Digital twin development and deployment for a wind turbine

10

10.1 Introduction

This chapter guides the user in developing a digital twin of a wind turbine using MATLAB Simscape™ and then deploying it into Amazon Web Services (AWS) cloud. There will be an equivalent hardware prototype representing the wind turbine. In this case, the hardware is the DC motor hardware along with the ESP32 used in Chapter 9. The data from the DC motor speed sensor for a given wind speed input shall be compared with the digital twin on the cloud for off-board diagnostics (Off-BD). Fig. 10.1 shows the Off-BD process for the wind turbine, whereas Fig. 10.2 shows the boundary diagram of the wind turbine system along with the interaction with its twin model. The outline of this chapter is shown below:

1. Physical asset setup and considerations: wind turbine hardware
2. Understanding the input−output behavior of the wind turbine Simscape™ model
3. Developing the driver Simscape™ model for the hardware and communicating to AWS
4. Deploying the Simscape™ digital twin model to the AWS cloud and performing Off-BD

With the exception of the digital twin model, the rest of the files and settings can be carried forward from Chapter 9. This chapter will focus on utilizing an existing MATLAB Simscape™ Wind Turbine example and converting that model to a digital twin for deployment on AWS.

All the codes used in the chapter can be downloaded for free from MATLAB File Exchange. Follow the link below and search for the ISBN or title of this book:

https://www.mathworks.com/matlabcentral/fileexchange/

Alternatively, the reader can also download the material and other resources from the dedicated website or contact the authors for further help:

https://www.practicalmpc.com/

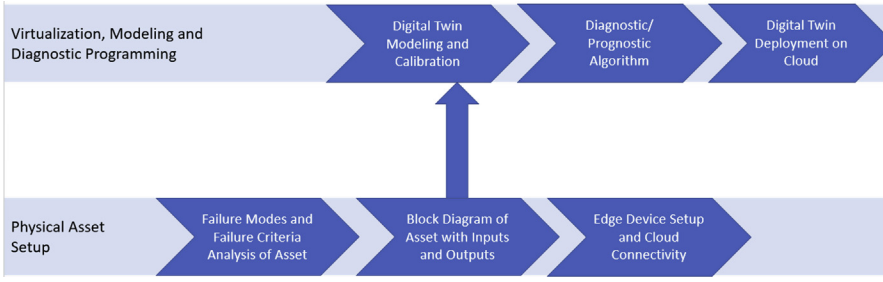


Figure 10.1 Off-board diagnostics process for wind turbine system.

Digital Twin Development and Deployment on the Cloud. https://doi.org/10.1016/B978-0-12-821631-6.00010-4

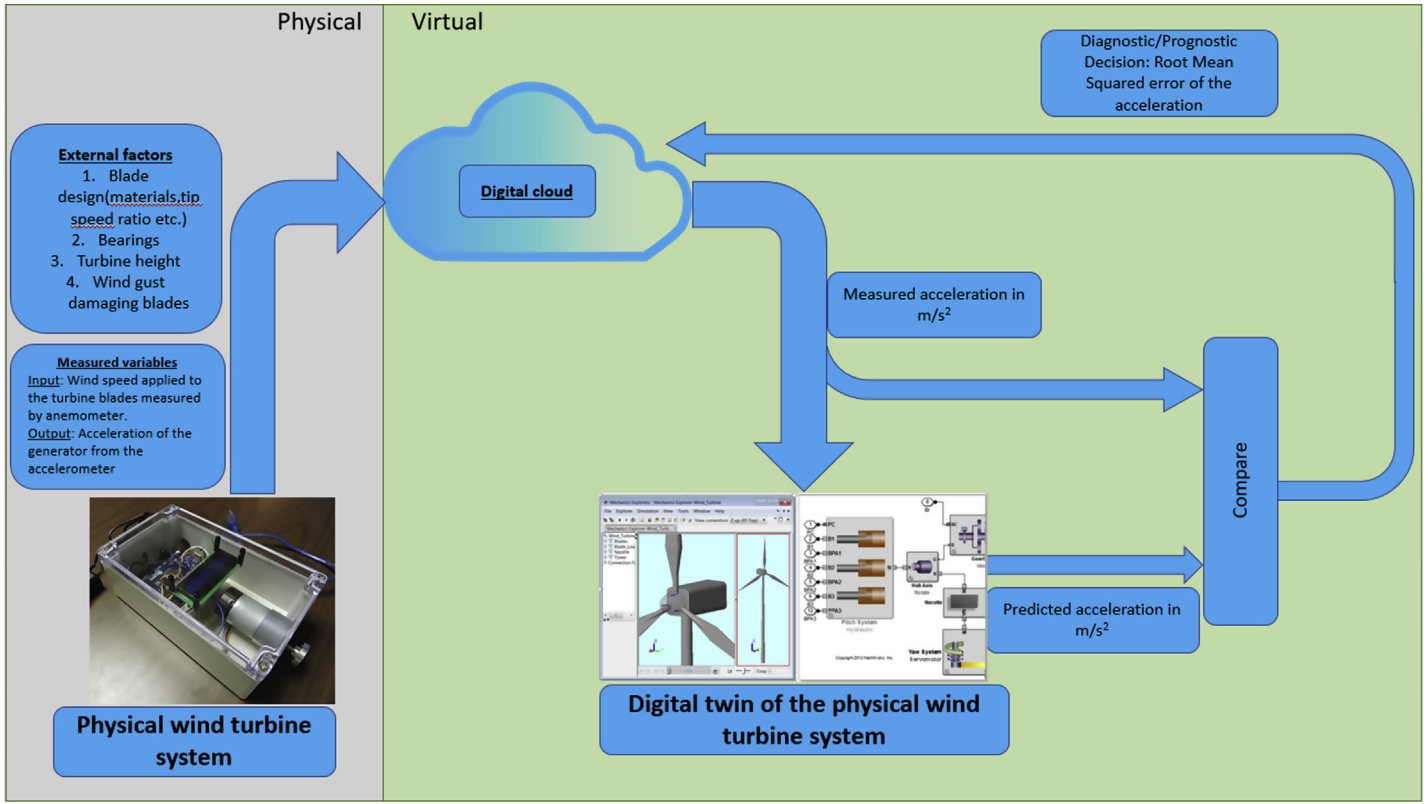


Figure 10.2 Block diagram of a wind turbine system.

10.2 Physical asset setup and considerations: wind turbine hardware

As mentioned in Section 10.1, the DC motor hardware along with the ESP32 module in Chapter 9 (Fig. 10.3) can be used for wind turbine purposes as well. The DC motor hardware is driven by the DC motor driver model described in Chapter 9. The goal of this chapter is to drive the DC motor using wind speed instead of predefined desired RPM as an input in the driver model. The DC motor is analogous to a generator in the wind turbine. There would be a relationship between wind speed and the RPM of the DC motor. Based on the RPM calculated in the model, a respective PWM command can be sent to the Arduino board. Fig. 10.4 shows the illustration of the Chapter 10 concept. Note that conversion from wind speed to motor speed would not be as straightforward as a Gain block as explained later in Section 10.3. The next section shall elaborate on this conversion process.

10.3 Understanding the input–output behavior of the wind turbine Simscape™ model

The digital twin for the wind turbine would be upon an existing MATLAB Simscape™ of a wind turbine model on MATLAB File Exchange [1]. The file may be

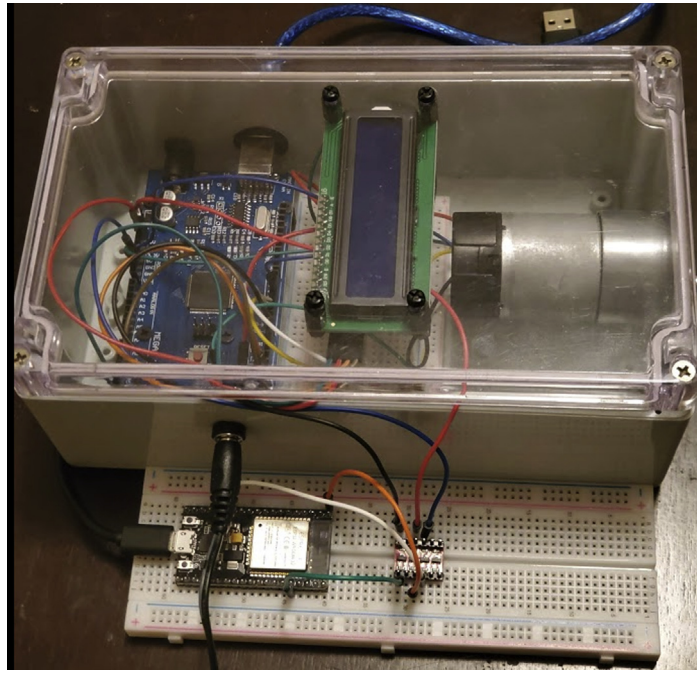


Figure 10.3 DC motor hardware with ESP32 module.

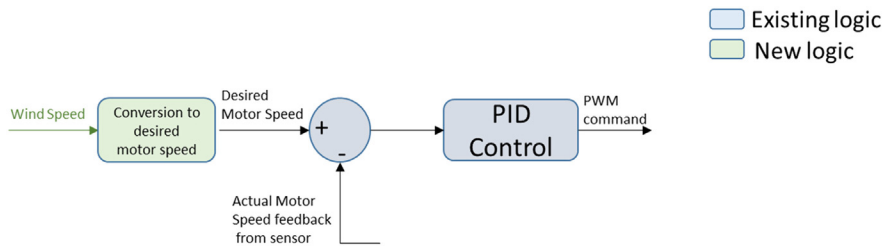


Figure 10.4 Control structure for wind speed to PWM command.

downloaded from https://www.mathworks.com/matlabcentral/fileexchange/25752-wind-turbine-model. It is recommended to read the instructions on how to download and setup this model given in the link. There is also a four-part webinar series based on the design of this model also shared in the link. The following paragraph below shall give a brief overview of the model as well as the key inputs and outputs that will correlate with the DC motor driver model.

The Wind Turbine model is shown in Fig. 10.5 with key areas highlighted. As seen in the figure, the inputs to the model are the wind speed along with the states of the wind turbine (explained below). The output is the RPM of the generator located in the Nacelle subsystem. The solver options are set to phasor 60 Hz as default.

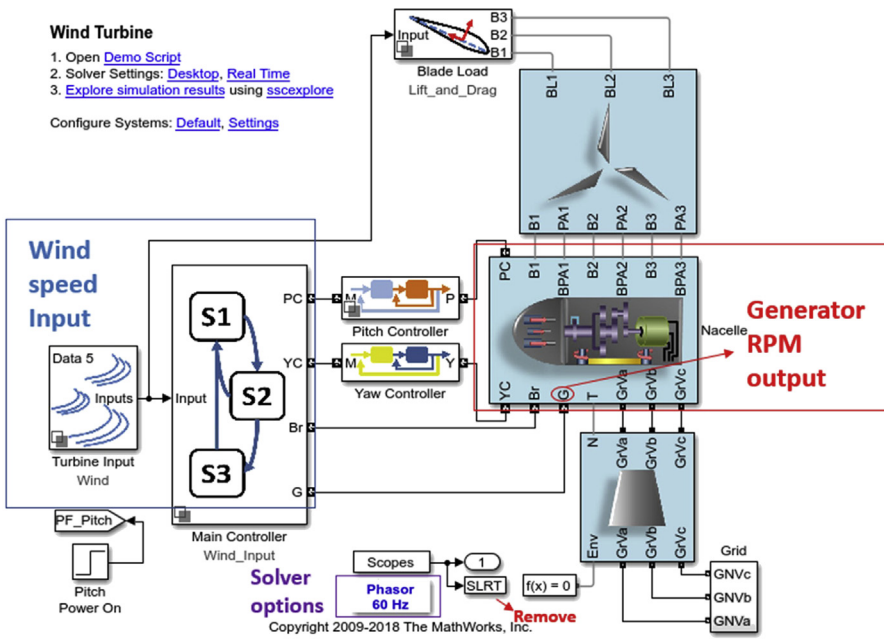


Figure 10.5 Wind turbine Simscape model.

10.3.1 Wind turbine model components

1. Firstly, the wind speed and wind direction is input to a stateflow logic. The different states for the wind turbine are PARK, STARTUP, GENERATING, and BRAKE. The stateflow always initializes in PARK state. The states can be seen in the model path Wind_Turbine/Main Controller/Wind Input/Turbine Stateflow also shown in Fig. 10.6.
   - **a.** If the wind speed is less than a "cut in lower" threshold, the wind turbine will remain in PARK state where the rotor for the generator is stopped.
   - **b.** Once the wind speed goes above the threshold in point 1a, and the wind speed is below a "cut out threshold", then the wind turbine will enter STARTUP state. The wind turbine blades will start rotating, hence energizing the generator until the rotor speed has achieved a steady-state value.
   - **c.** Once the rotor has reached a steady-state speed, the wind turbine will enter GENERATING state where the generator is able to deliver power to the power grid.
   - **d.** If the wind speed goes below the "cut in lower" threshold when the state is in STARTUP or GENERATING, then the state will transition to BRAKE where a pitch of the blades are adjusted to stall the motion. Note that there are multiple conditions that can cause the state to transition from GENERATING to BRAKE, but it is beyond the scope of this chapter.
   - **e.** If the rotor speed is less than the park speed threshold in BRAKE state, then the state will transition from BRAKE to PARK.
2. Depending on the state of the turbine, a pitch is applied to the turbine blade to move the blades when the state is in STARTUP or GENERATING and reduce the movement when the state is in PARK or BRAKE state.
3. The Blade Load subsystem in Fig. 10.5 is to calculate lift and drag loads on the turbine blade. In order to determine the lift and drag loads, the pitch and yaw commands are needed from the pitch and yaw controllers. This information is sent to the Simscape portion to simulate the movement in the Mechanics Explorer and then to the Nacelle subsystem.
4. The Nacelle subsystem includes the gear train that converts the low turbine blades RPM to high-speed RPM for the generator as per the gear ratio. The output RPM of the generator is used in the stateflow logic, and it is this output that will be compared with the actual DC motor RPM.
5. The yaw controller subsystem is used to turn the turbine around the Z axis (yaw) to make sure the wind is perpendicular to the blades for movement.
6. Lastly, the tower subsystem is used to deliver the power generated by the generator to the power grid subsystem.

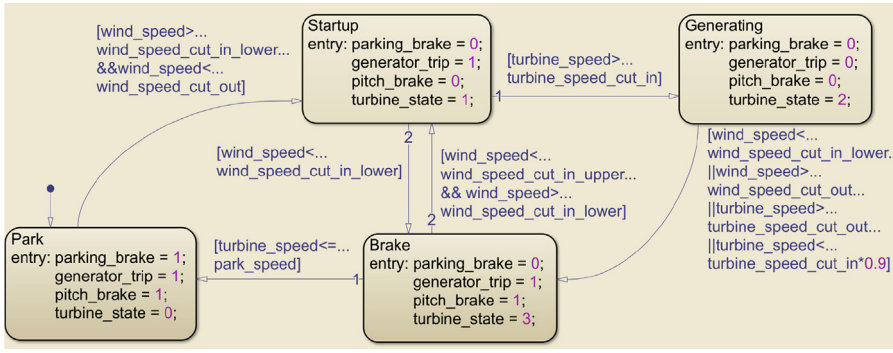


Figure 10.6 States of the wind turbine.

10.3.2 Input–output response of the model

The model is ready to be simulated with variable time step. Before running the simulation, the SLRT block adjacent to the Phasor 60 Hz block in the top level of the model needs to be removed as shown in Fig. 10.5. SLRT is for Simulink Real-Time Workshop for code generation purposes; however, in this chapter, embedded coder ert.tlc is used instead for code generation. Once the SLRT block is deleted, the model can be simulated by pressing the play button. As the simulation is running, the system performance of the wind turbine can be observed graphically in the Mechanics Explorer window. The key outputs to observe can be found in Wind_Turbine/Scopes/Generator Scopes path of the model. Double click on the Rotor Turbine speed scope to observe the RPM response of the generator as shown in Fig. 10.7. As seen from the second subplot in Fig. 10.7, the generator ramps up to a steady state speed of 1200 RPM after approximately 10 seconds. This behavior can be understood as a response to the wind speed input located in Wind_Turbine/Turbine Input/Wind shown in Fig. 10.8.

The input wind speed has a direct relationship to the generator RPM as per the stateflow logic shown in Fig. 10.6. It can be observed that when the wind speed is less than wind_speed_cut_in_lower (tuned to 4 m/s), the turbine is PARK mode; hence, the generator RPM is 0. After a few seconds when the wind speed ramps above 4 m/s, the generator RPM starts to ramp as the state is now in STARTUP. The PI control in the Pitch Control subsystem will now generate a pitch command as per the desired target rotor speed, which is set to 1200 RPM. Once the steady state has been achieved, the generator can output power. The generator RPM output plot can be divided into three sections as shown in Fig. 10.9.

10.3.3 Customizing the model as per the digital twin requirements

The input–output relationship can now be customized as per the digital twin and physical hardware needs of this chapter. Four factors need to be kept in mind:

1. The target speed of 1200 RPM needs to be changed to a smaller value to accommodate for the physical asset DC motor hardware limits.
2. The inputs to the stateflow would need to be modified to now allow steady-state speed in STARTUP instead of GENERATING. The rationale is that the power output in GENERATING state is not being considered in this chapter. In addition to this, it is found that there are divergence issues for GENERATING state when discretizing the model for digital twin purposes explained later in Section 10.4.
3. Any Proportional or Integral gains in the pitch control logic would need to be taken into consideration after changing the target speed.
4. The simulation time needs to be increased above 70 s to accommodate all wind speeds and understand the generator RPM output when the state transitions out of steady state.

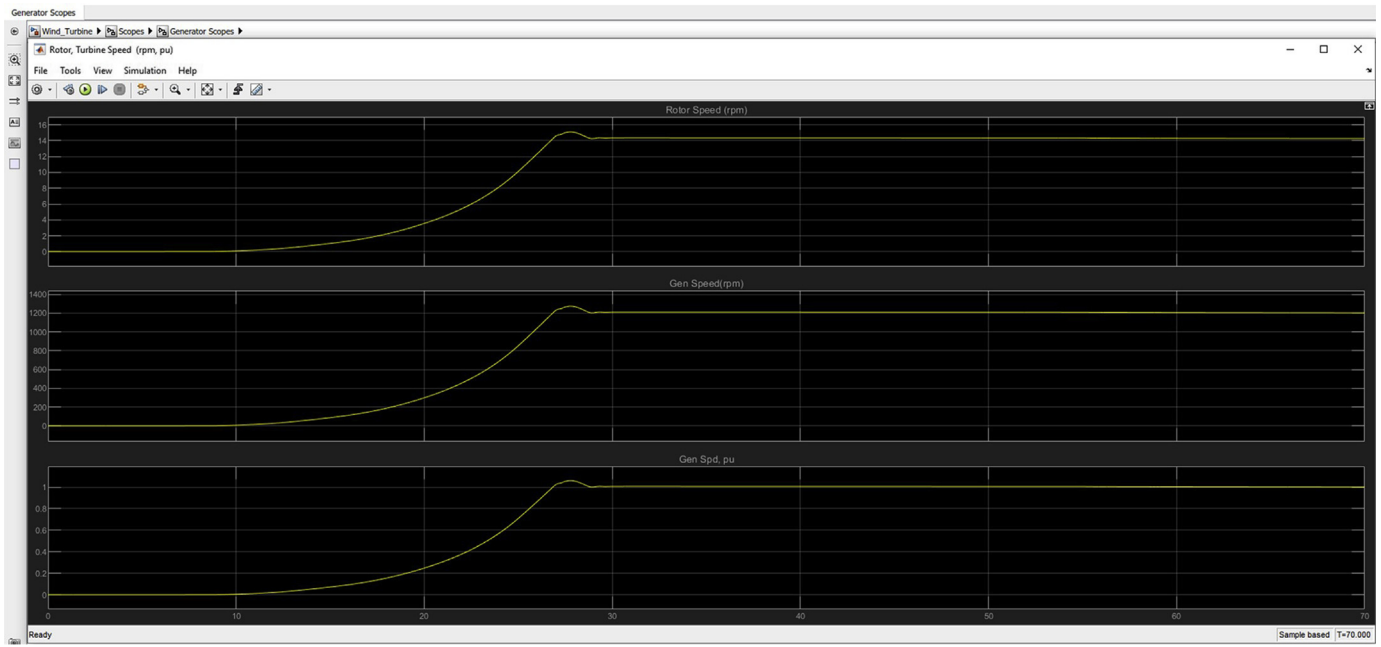


Figure 10.7 Output response of the generator.

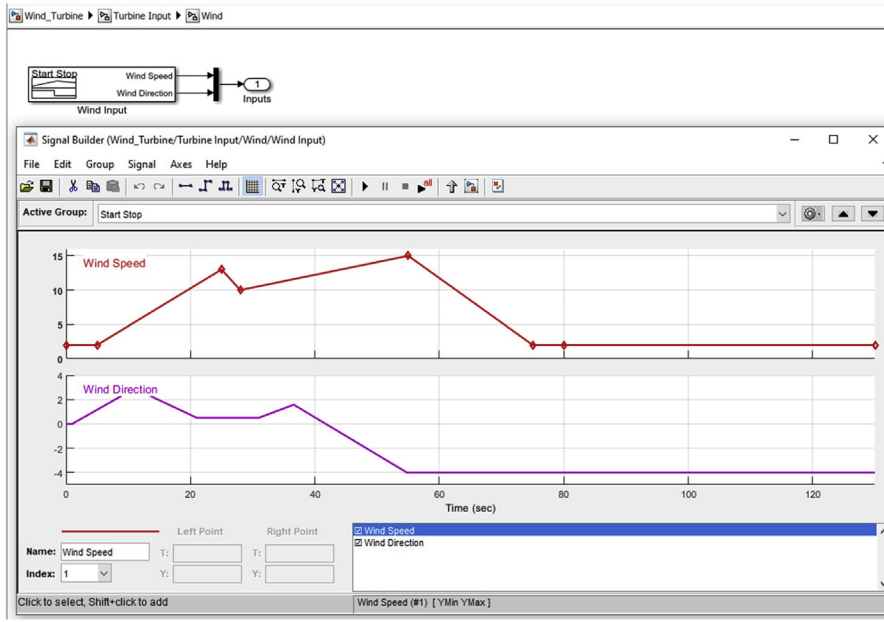


Figure 10.8 Wind speed and wind direction input.

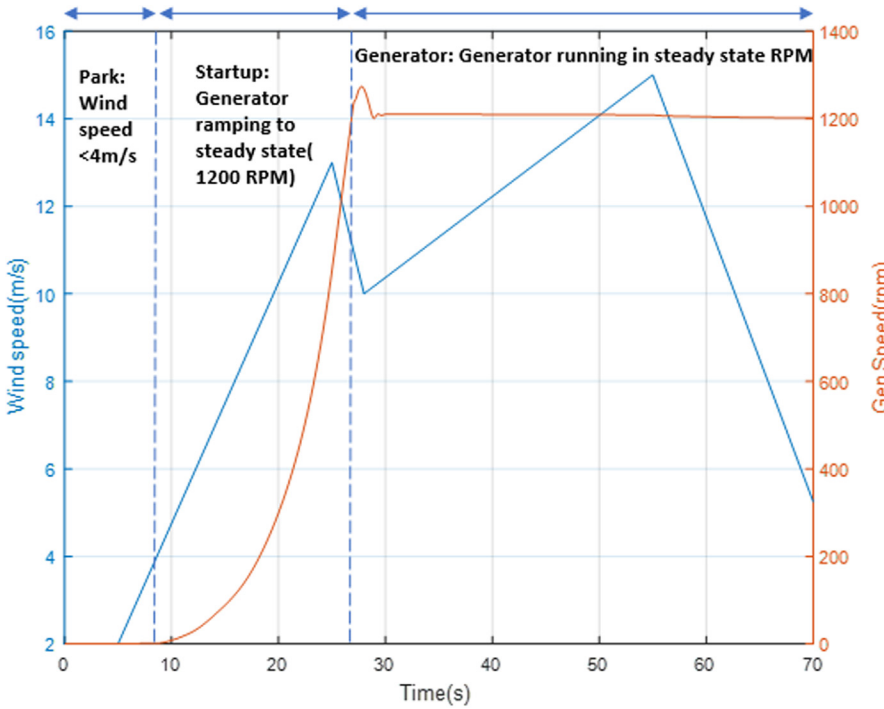


Figure 10.9 Relationship between wind speed and generator RPM.

10.3.3.1 Target speed

To change the target speed, navigate to Wind_Turbine/Pitch Controller/PI on AoA/Pitch Controller/Determine Pitch Command/Determine Desired Angle of Attack and change the Nominal RPM value to the value shown in Fig. 10.10. WT_Params.Rotor.-nominal_rpm is set as 1200 as per the Wind_Turbine_Parameters.m script. This script is run automatically when the model is run as per its design. Dividing this number by 5 will set the desired RPM as 240, which is within the acceptable range of speeds of the DC motor.

10.3.3.2 Steady-state speed in STARTUP state

With the target speed changed, navigate to the stateflow logic Wind_Turbine/Main Controller/Wind Input/Turbine Stateflow. Observe that the speed changed in 10.3.3.1 is not part of the stateflow. To make sure the state remains in STARTUP when the wind speed has reached the target speed, verify the turbine_speed_cut_in is set to 1200 RPM. This should be the default value in the Wind_Turbine_Parameters.m. Notice the condition to transition from STARTUP to GENERATING is turbine_speed>turbine_speed_cut_in. Since the turbine_speed stead state is 240 RPM, it will not transition to GENERATING state.

10.3.3.3 Proportional (P) and I (Integral) gains change in the pitch actuation control logic

Changing the target speed may affect the transient behavior of the pitch actuation, which is needed to derive drag and lift forces for the turbine blades. The P and I gains can be tuned by using the MATLAB Control System Tuner and Parameter Estimator. The P and I gains for the pitch control are located in Wind_Turbine/Pitch Controller/PI on AoA/Pitch Controller/Actuator Controller/PI Controller. Change only the P gain as shown in Fig. 10.11.

10.3.3.4 Increasing the simulation time

As seen in Fig. 10.8, the wind speed input is set for 130 s. Increase the simulation time from 70 to 130 s next to the play button in Simulink. This way, the transition from STARTUP to BRAKE is captured when wind speed falls below wind_speed_cut_in_lower.

Run the simulation again and navigate to Wind_Turbine/Scopes/Generator Scopes. Double click on the Rotor Turbine Speed scope. It can be observed that the generator speed now stabilizes its speed around 240 RPM with an overshoot shown in Fig. 10.12. The reader may tune the P and I gains to reduce overshoot; however, fine tuning these gains is outside the scope of this chapter. Also, look into the turbine states scope in Wind_Turbine/Scopes/Main Controller Scopes. As seen in Fig. 10.13, the state now transitions from 0 (PARK) to 1 (STARTUP) to 3 (BRAKE).

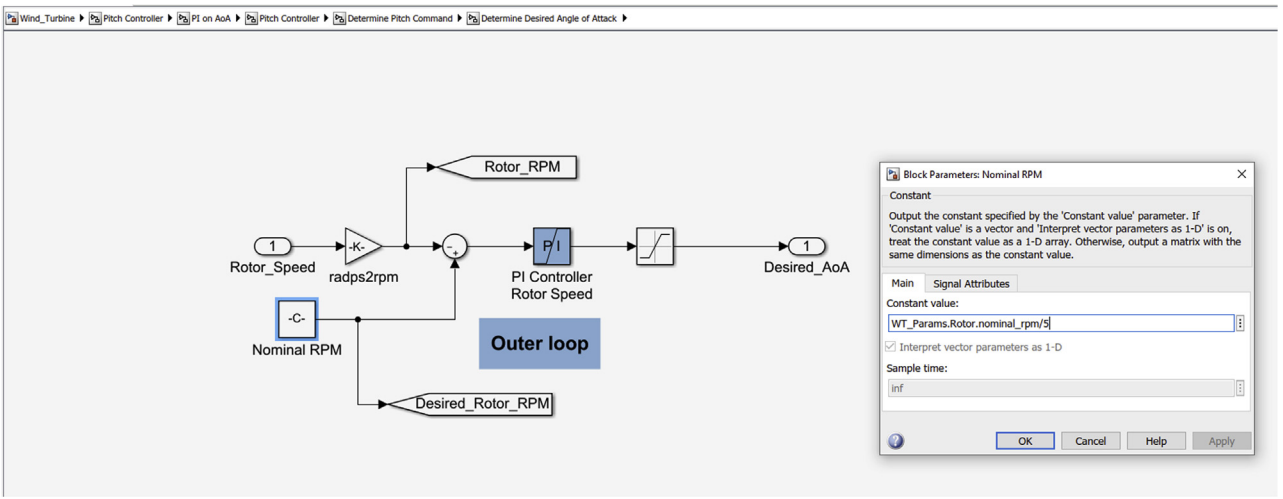


Figure 10.10 Changing the target speed.

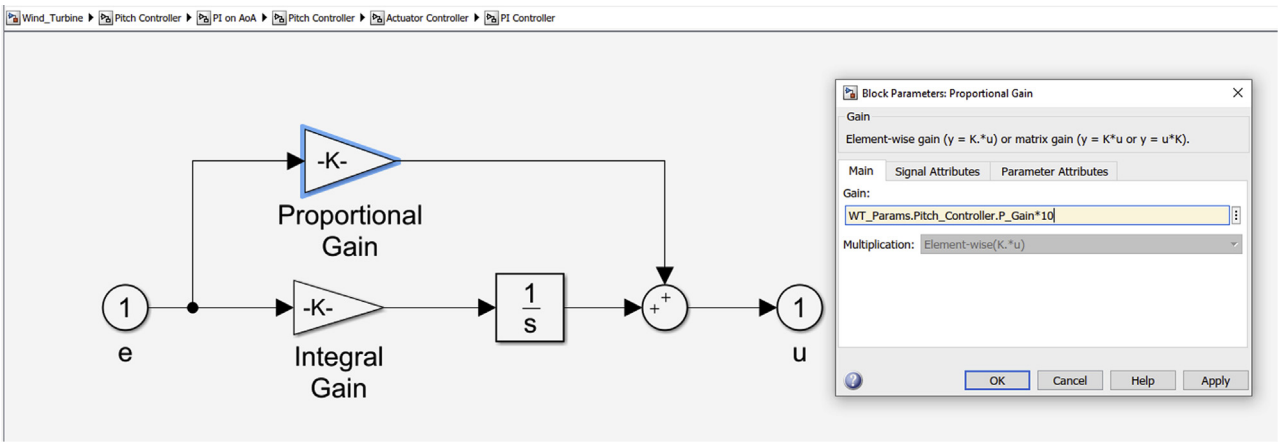


Figure 10.11 Changing the P gain.

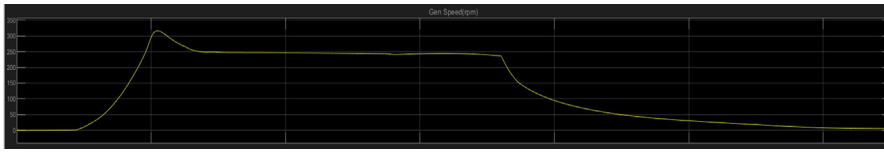


Figure 10.12 Generator speed RPM profile for a 1200 RPM target setpoint.

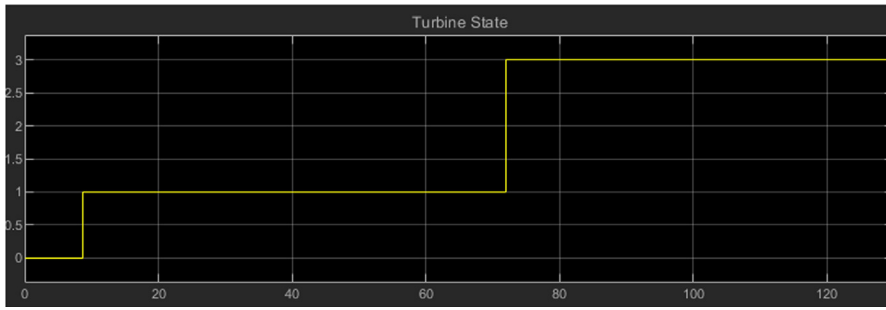


Figure 10.13 Turbine state transitions.

10.3.4 Developing relationship between wind speed and generator speed

The wind speed time series and generator speed time series according to the digital twin needs are plotted together as shown in Fig. 10.14. This profile can now be divided into separate regions where each region has one piecewise function. This is done to replicate this profile on the DC motor hardware model.

Fig. 10.15 shows the generator speed curve split into five piecewise functions described below:

1. In the first piecewise function, RPM is to remain 0 if turbine state is in PARK state.
2. In the second piecewise function, RPM is taken as function of wind speed in STARTUP state. This function may be in the form of a third-degree polynomial equation. Note that the overshoot is not being modeled in this equation.
3. In the third piecewise function, RPM is equal to 240 in STARTUP state.
4. In the fourth piecewise function, RPM is taken as function of wind speed provided the fact that the wind speed does not go below 2 m/s. This function is applicable to BRAKE state.
5. In the fifth piecewise function, once the wind speed is constant 2 m/s, then RPM is a function of the simulation time. This function is applicable to BRAKE or when state transitions from BRAKE to PARK. This last function is just for the sake of completing the generator speed profile since wind speed has no influence on the decay of generator speed.

The piecewise function can be formulated by either using the "polyfit" command in MATLAB or the polynomial regression in Excel. The things to keep in mind are

1. For the second piecewise function, use only the points when wind speed starting point is 4 m/s and end point would be when the generator speed reaches 240 RPM. This way the overshoot is not modeled for the DC motor driver model. Use wind speed as the x input and the generator speed as the y output.

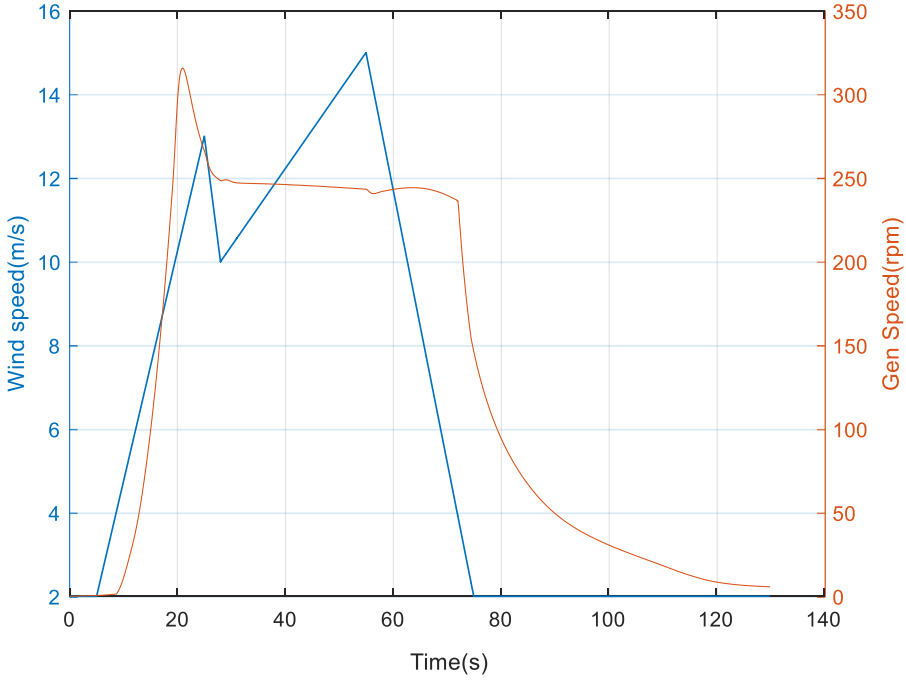


Figure 10.14 Wind speed input and generator speed output.

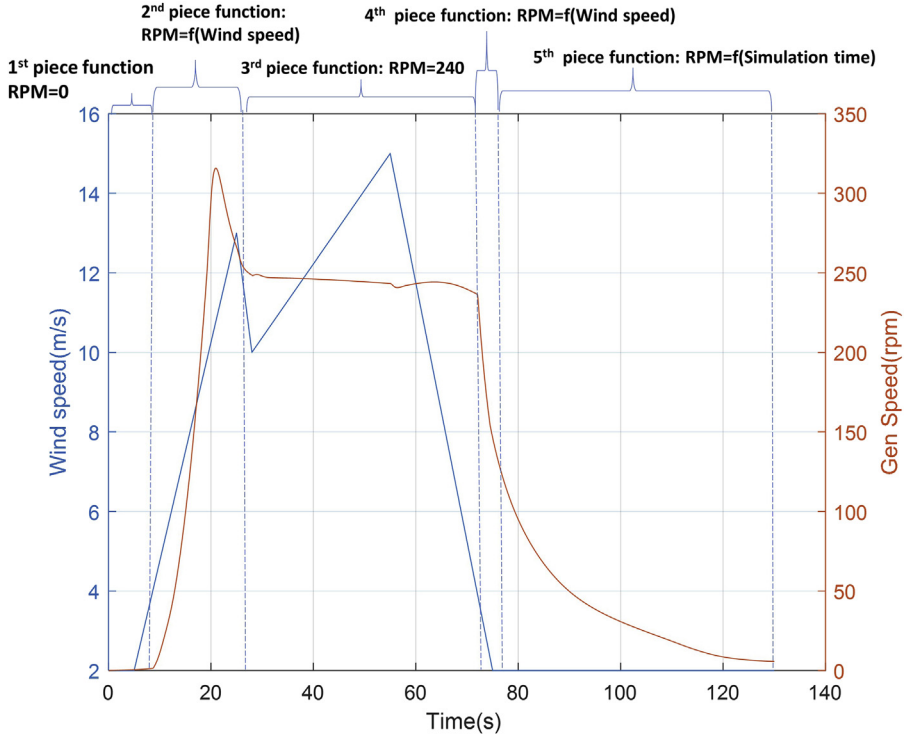


Figure 10.15 Generator speed piecewise functions.

2. For the fourth piecewise function, the start point would be when the state transitions from STARTUP to BRAKE and the end point would be when wind speed has decayed to 2 m/s. Use wind speed as the x input and the generator speed as the y output.
3. For the fifth piecewise function, the start point would be when the wind speed has decayed to 2 m/s in BRAKE or PARK state. Use simulation time as the x input and the generator speed as the y output.

The piecewise functions for this chapter were developed in Excel using polynomial regression.

1. Second piecewise function:

$$\text{Generator RPM} = 0.4362 * \left(\text{windspeed}^3\right) + \left(-1.7756 * \text{windspeed}^2\right) + 0.3608 * \text{windspeed} \tag{10.1}$$

Eq. (10.1): Wind speed to generator RPM when state is STARTUP

2. Fourth piecewise function:

$$\text{Generator RPM} = 0.219 * \left(\text{windspeed}^3\right) + \left(-9.5573 * \left(\text{windspeed}^2\right)\right) + 95.988 * \text{windspeed} \tag{10.2}$$

Eq. (10.2): Wind speed to generator RPM when state has transition from STARTUP to BRAKE

3. Fifth piecewise function:

$$\text{Generator RPM} = -0.0021 * \left(\text{time}^3\right) + \left(0.6993 * \left(\text{time}^2\right)\right) - 79.429 * \text{time} + 3041.3 \tag{10.3}$$

Eq. (10.3): Simulation time to generator RPM relationship when state is BRAKE or PARK and wind speed has decayed to 2 m/s.

With the piecewise functions developed, the DC motor driver model can now be modified to include the wind speed to motor RPM relationship.

10.4 Developing the driver Simscape™ model for the hardware and communicating to AWS

10.4.1 Developing the driver Simscape™ model for the hardware

1. Firstly, open the driver model for DC motor that was developed in Chapter 9, it is named as Chapter_9_Section_6_3.slx in Chapter 9. Save it as Chapter_10.slx.

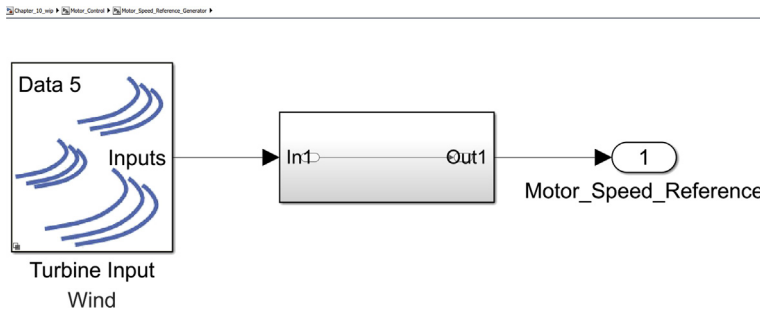


Figure 10.16 Changes inside Motor_Speed_Reference subsystem.

2. Double click the Motor_Speed_Reference_Generator subsystem and delete everything except the Motor_Speed_Reference outport. Then copy the Turbine Input subsystem block from the Wind_Turbine.slx Simulink model and paste into the Chapter_10.slx model in the Motor_Speed_Reference_Generator subsystem. Then add an empty subsystem and connect the Turbine Input to the inport of the subsystem. The outport of the empty subsystem should be connected to the Motor_Speed_Reference outport. Fig. 10.16 shows the changes.
3. Rename the empty subsystem as Main Controller, the In1 inport as Input, and Out1 outport as RPM. Delete the connection between Input and RPM. Then click anywhere on the model and type **Demux** as shown in Fig. 10.17. Note for the rest of this section, any Simulink blocks that needed to be added can be added in this way. A **Demux** block separates the wind speed and wind direction from the Turbine Input subsystem.
4. Connect Input to the Demux. Connect the second output of the Demux to a **terminator.** A **terminator** can be found in the same manner as a **Demux.** Then copy the Turbine Stateflow logic in Wind_Turbine/Main Controller/Wind Input in Wind_Turbine.slx model and paste it to Chapter_10/Motor_Control/Motor_Speed_Reference_Generator/Main_Controller in Chapter 10.slx. Connect the first output of the Bus selector to the wind_speed input of the stateflow logic. The changes are shown in Fig. 10.18.
5. Before proceeding with the stateflow logic, the parameters inside the stateflow need to be defined. These parameters are defined in Wind_Turbine_Parameters.m file of the Wind_Turbine.slx. Enter Wind_Turbine_Parameters in the MATLAB Command window to define the variables. In addition to this, go to File->Model Properties->Model Properties and enter Wind_Turbine_Parameters in the PreLoadFcn as shown in Fig. 10.19. This shall run the m file before the Chapter_10.slx model is opened.

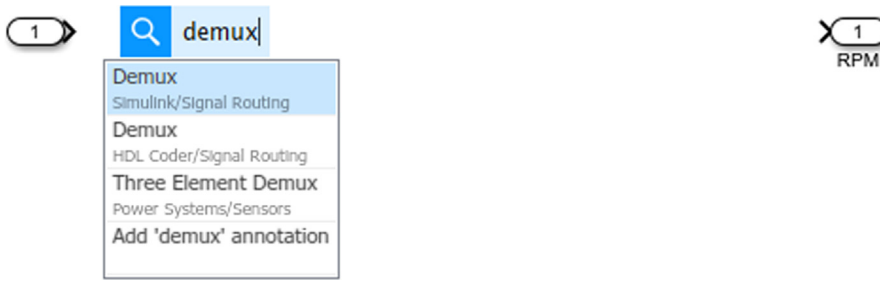


Figure 10.17 Demux to separate wind speed and wind direction.

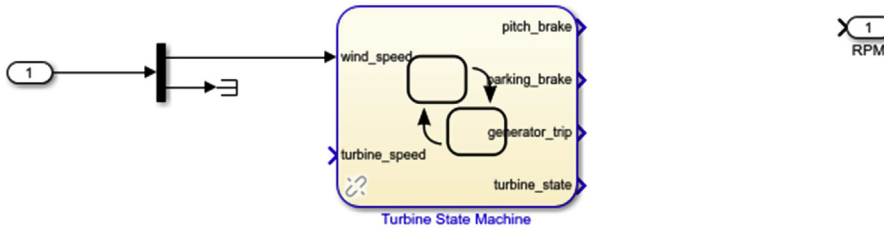


Figure 10.18 Demux to separate wind speed and wind direction.

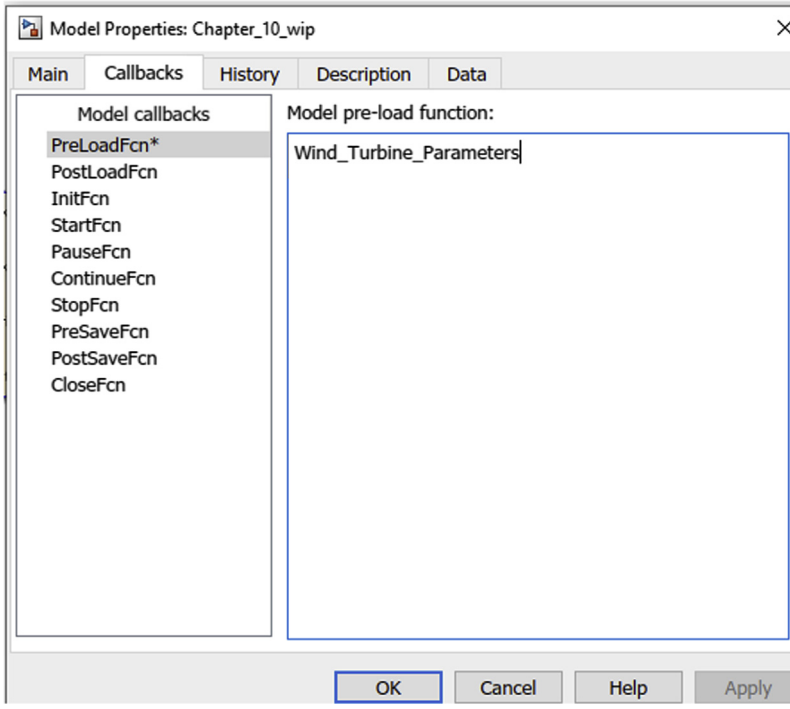


Figure 10.19 PreLoad function for the model.

6. Connect the pitch_brake, parking_brake, and generator_trip outputs of the stateflow to terminators. Then add an **If** block. Double click on the **If** block and set the Parameter settings as shown in Fig. 10.20. In this case, u1 is the turbine_state output of the stateflow logic whereas u2 is the simulation time. There will be four cases for this if condition:
   a. u1==0 and u2<60 is for the case when turbine state is in PARK and simulation time is less than 60 s. This is for the case when motor RPM needs to be 0. The reason time is used here is because the turbine state can go from BRAKE to PARK state when the RPM is decaying and it is a nonzero value.
   b. u1==1|u1==2 is for the case when turbine state is in STARTUP or PREPARING state. This will have the second piecewise function developed in the previous section.

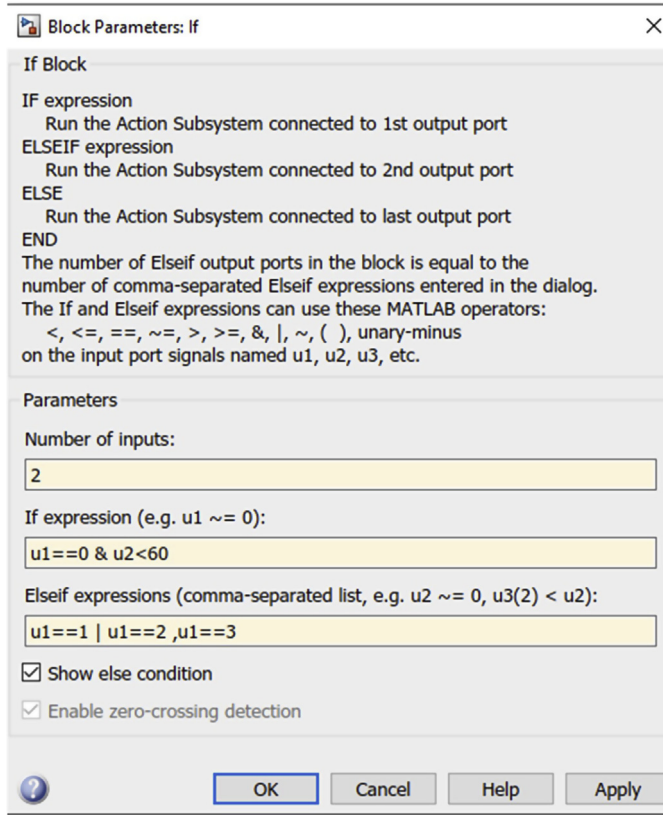


Figure 10.20 Parameter settings for If block.

c. u1==3 is for the case when turbine state is in BRAKE state. This will have the fourth and fifth piecewise function.

d. The else condition is for the case when u1==0 only. This case activates when state changes from BRAKE to PARK only for the generator RPM profile shown in the previous section.

7. Connect the turbine_state output of the stateflow logic to u1 of the If block. Then insert a **clock** and connect it to u2 of the **If** block. The changes can be seen in Fig. 10.21.

8. Insert four **If Action Subsystem** blocks and align them vertically to the right of the **If** block. Connect the case of the **If** block to the respective **If Action Subsystem** block. Next, delete the inport of the **If Action Subsystem** connected to the u1==0 and u2<60 case and also the else case. Name outports of **all the If Action Subsystem** as RPM and the remaining inports as Wind_speed. The updated model is shown in Fig. 10.22.

9. Now, the logic for each subsystem can be put in place. Go inside the **If action subsystem** block for the u1==0 and u2>60 and connect a **Constant** block of 0 to the RPM outport as shown in Fig. 10.23

10. Go inside the **If action subsystem** block for the u1==1|u1==2 case and follow these steps:

a. Insert a **Fcn** block, a **Switch** block, and two **Unit delay** blocks.

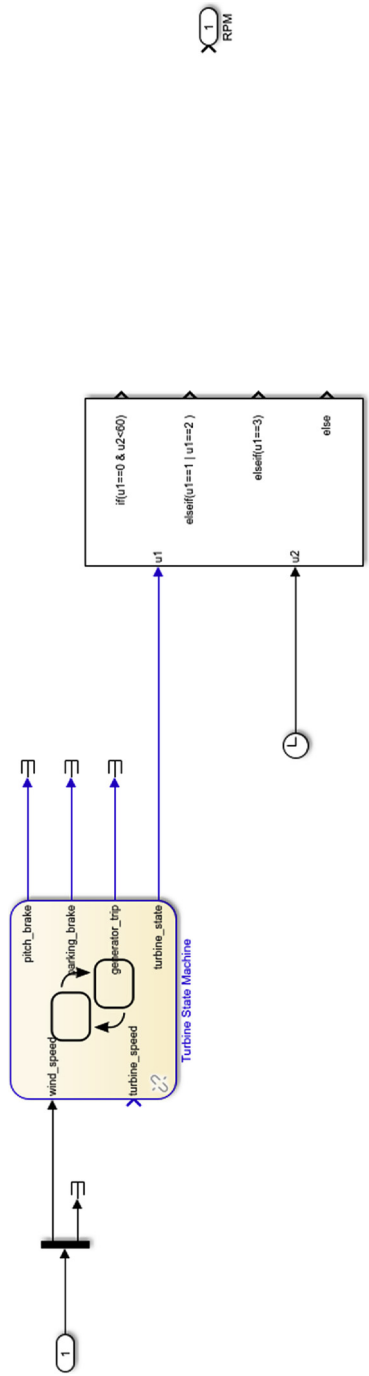


Figure 10.21 Inputs for the If block.

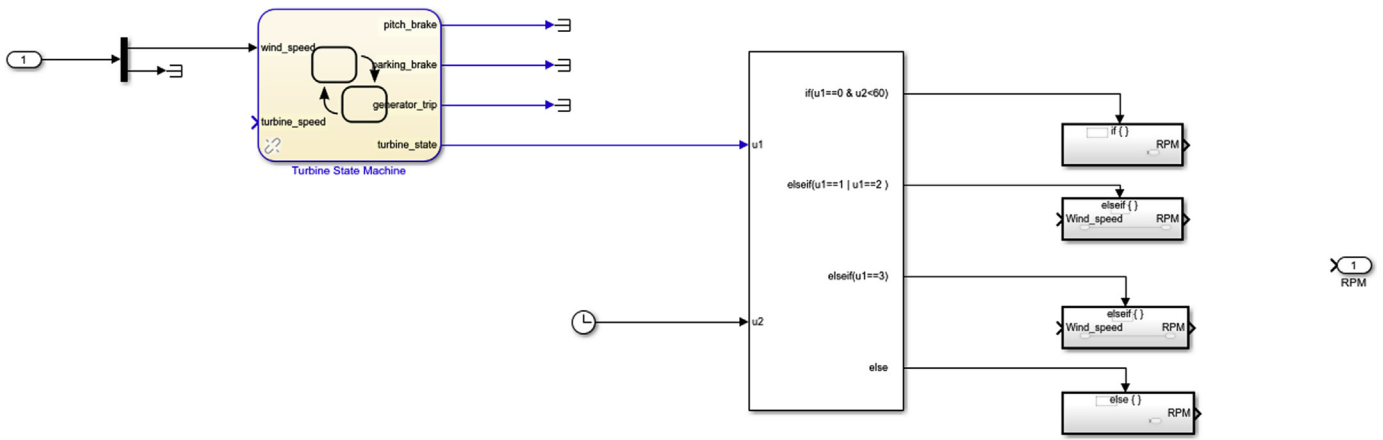


Figure 10.22 If action subsystems for the If cases.

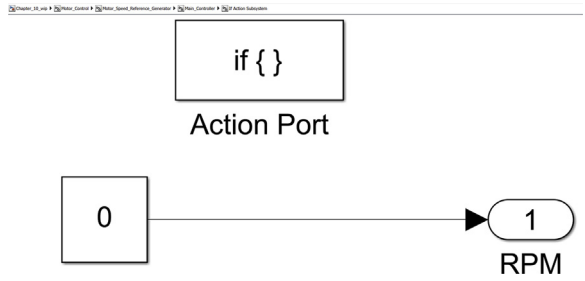


Figure 10.23 Inside the first If action subsystem block.

b. Connect these blocks as shown in Fig. 10.24.
c. Double click on the **Fcn** block and insert the second piecewise function developed in the previous section as shown in Fig. 10.25.
d. Double click on the **Switch** block and set the u2>=Threshold as 240 as shown in Fig. 10.26. This will ensure that the piecewise function is followed until motor speed reaches 240 RPM.

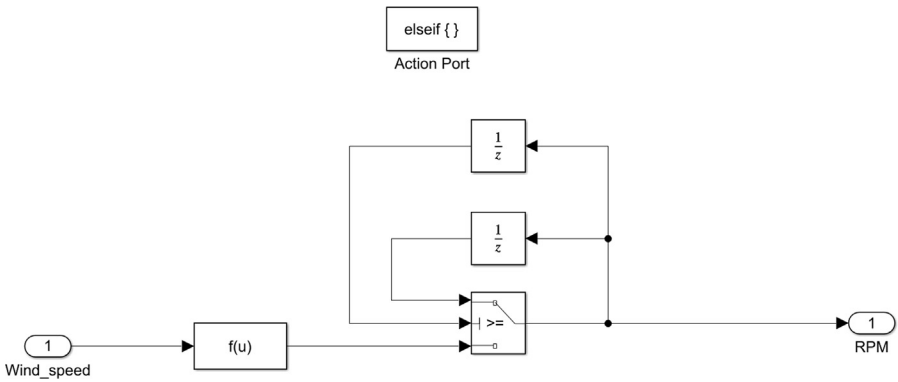


Figure 10.24 Inside the second If action subsystem block.

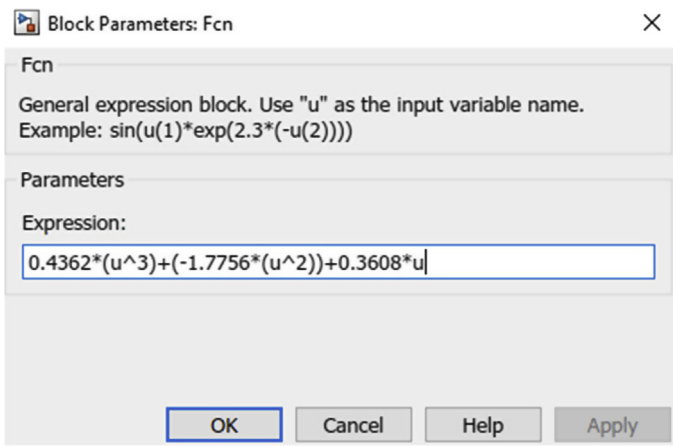


Figure 10.25 Second piecewise function.

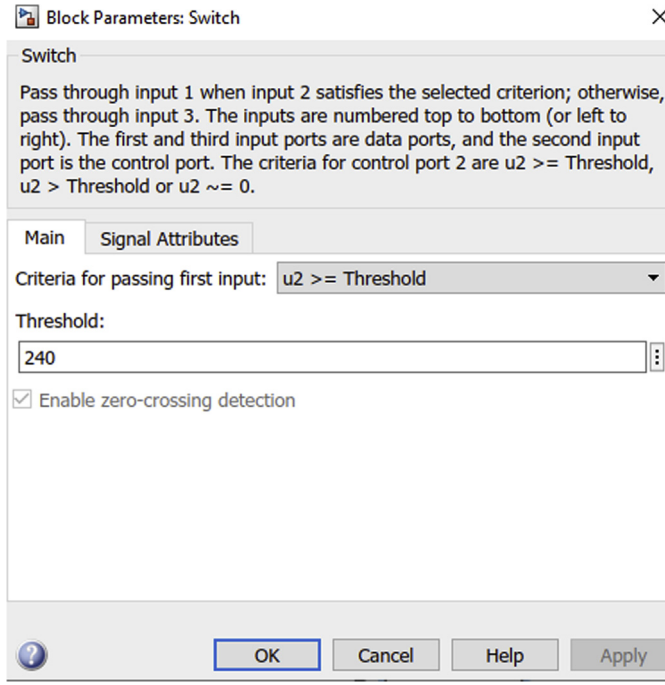


Figure 10.26 Switch threshold for second If action subsystem.

11. Go inside the **If action subsystem** block for the u1==3 case and follow these steps:
- **a.** Insert a Clock block, two Fcn blocks, a Switch block, and a Saturation block.
- **b.** Connect them as shown in Fig. 10.27. Rename the Fcn blocks as WindFcn and TimeFcn.
- **c.** Double click on the Wind Fcn block as insert the fourth piecewise function as shown in Fig. 10.28.

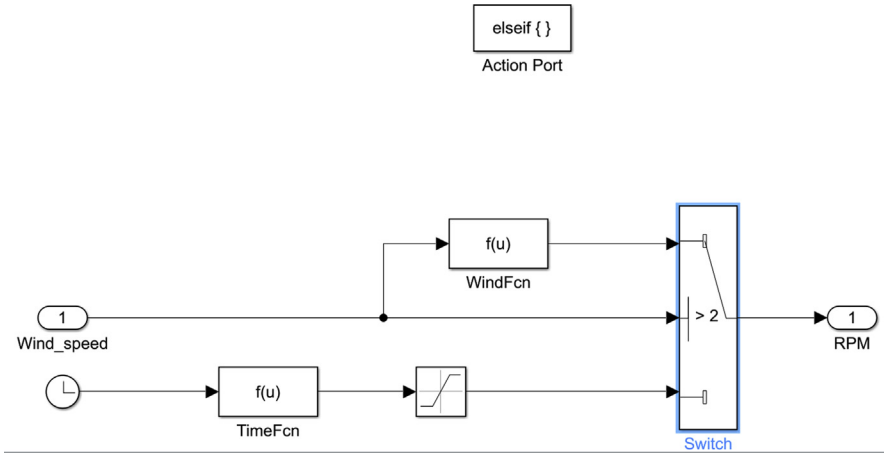


Figure 10.27 Inside the third If action subsystem block.

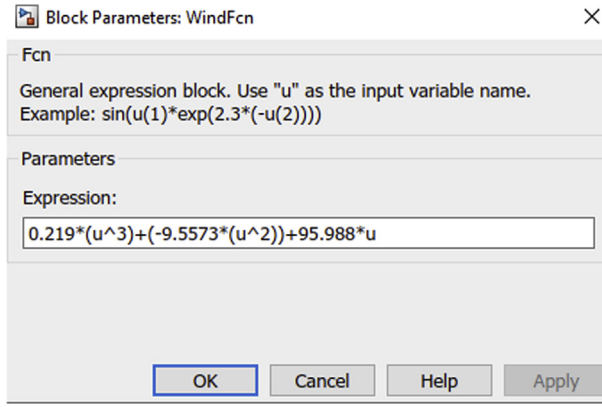


Figure 10.28 Piecewise function for wind to RPM.

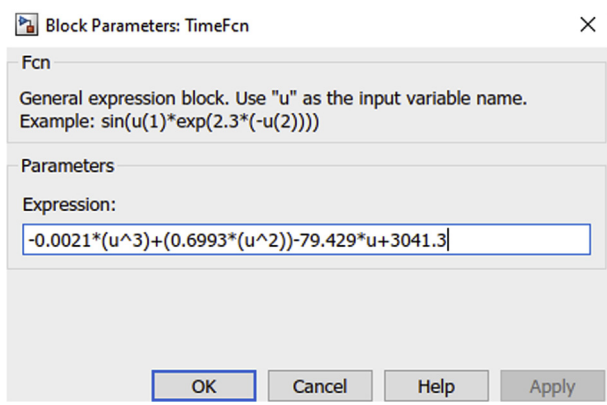


Figure 10.29 Piecewise function for time to RPM.

d. Insert the fifth piecewise function for the TimeFcn block as shown in Fig. 10.29.
e. Double click on the Saturation block and set the upper limit as 240 and lower limit as 0.
f. Double click on the **Switch** block and for the "Criteria for passing first input" field, set it as u2>threshold where the threshold is 2.

12. Go inside the **If action subsystem** block for the else case and follow these steps:
a. Insert a **Clock** block, a **Fcn** block, and a **Saturation** block.
b. Connect them as shown in Fig. 10.30.
c. Double click on the Fcn block as insert the fifth piecewise function as shown in Fig. 10.31.
d. Double click on the **Saturation** block and set the upper limit as 240 and lower limit as 0.

13. Navigate to Chapter10/Motor_Control/Motor_Speed_Reference_Generator/Main_Controller. Place a **Goto** flag and connect it to the top signal of the two signals coming out of the **Demux** block. Rename it as Wind_Speed. Double click on the flag and change tag visibility to global as shown in Fig. 10.32. The following **Goto** flags shall also have global visibility. Then, place two **From** flags and connect it to the second and third **If action subsystems**.

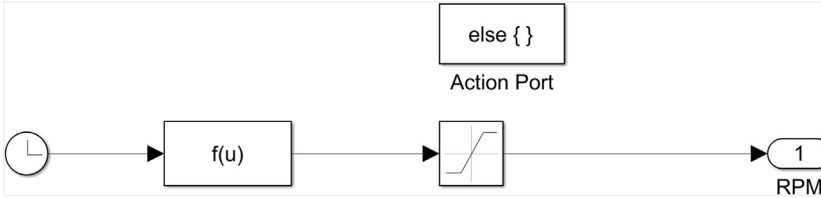


Figure 10.30 Inside the fourth If action subsystem block.

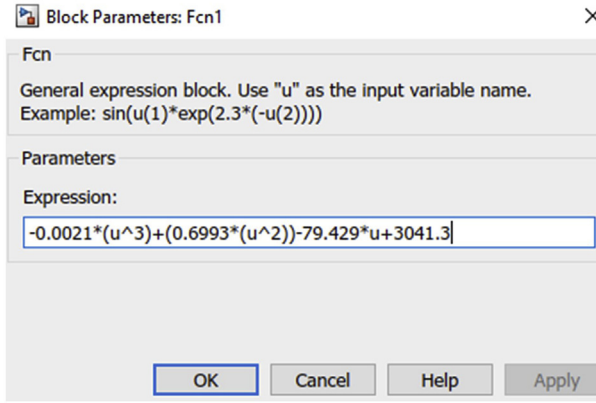


Figure 10.31 Piecewise function for time to RPM.

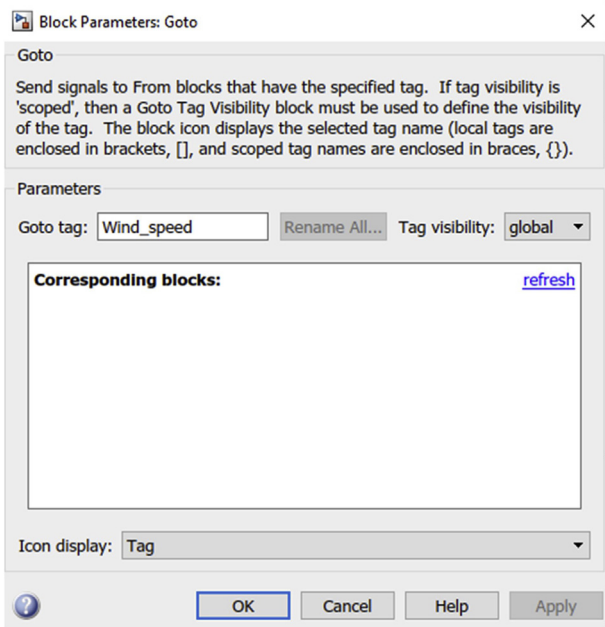


Figure 10.32 Wind_speed GoTo flag global Tag visibility.

14. Insert a **Merge** block and set the inputs as 4. Connect the output of each **If action subsystem** to the respective input of the **Merge** block. The output of the Merge block should be connected to the outport named RPM. The changes in this step and the previous one are shown in Fig. 10.33.
15. For the turbine_speed input, place a **Goto** flag named Turbine_speed in Chapter_10/Motor_Control/Motor_Speed_Sensor as shown in Fig. 10.34. Make sure the tag visibility is global. Place a **From** flag in Chapter_10/Motor_Control/Motor_Speed_Reference_Generator/Main_Controller with the name Turbine_speed.
16. Place a **Goto** flag named state for the turbine_state output of the stateflow logic in Chapter_10/Motor_Control/Motor_Speed_Sensor and name it State. Make sure tag visibility is global as this will be used outside this subsystem later on. In addition to this, add a **Rate Transition** and a **Scope** block for the wind_speed input and turbine_state output. Also add a **ToWorkspace** block connected to wind_speed and Turbine_speed inputs. Name them as Wind_speed and Turbine_speed, respectively. Make sure the save format for the **ToWorkspace** block is Structure with Time as shown in Fig. 10.35. Completed section for Chapter_10/Motor_Control/Motor_Speed_Reference_Generator/Main_Controller is shown in Fig. 10.36.
17. With the Main_Controller subsystem completed, the Serial_Data_Transmit_to_ESP32 subsystem in the top level of the model can be modified now. As mentioned in the previous section, only the steady-state motor RPM is going to be compared for Off-BD detection. Currently, the Serial_Data_Transmit_to_ESP32 sends data all the time; hence, a few modifications would need to be made for this subsystem to only send data when system has entered steady state. Firstly, go inside Chapter_10/Motor_Control/Serial_Data_Transmit_-to_ESP32/Chart, right click anywhere, and click Explore. It should open the Model Explorer. In the top bar menu, click on "Add Data" two times to add two parameters as shown in Fig. 10.37.
18. Name the two new parameters as time and State, respectively. Click on each of the two new parameters and enter the settings on the right side as shown in Figs. 10.38 and 10.39. Rename the first input as Wind_speed instead of PWM_Command. Close the explorer window once done.
19. Right click on the solid blue (gray in print version) dot which is the initial state and select the If-Else option as shown in Fig. 10.40.
20. Insert the conditions for the If-Else as shown in Fig. 10.41. Notice that the If action is the same statement that was there previously when stateflow was sending messages all the time. The If condition will now only send messages when state is in STARTUP, time is greater than 30 s, and wind speed is greater than 10 m/s. The time condition is to make sure we are not capturing any overshoot. This is initial guess based on the results shown in the previous section. The wind speed greater than 10 m/s in conjunction with the time condition will make sure only the wind increasing speed option is chosen. Since this wind speed is going to be an input for the digital twin and it will take some time for the digital twin to reach steady state, decreasing wind speeds may yield lower RPM value than steady-state value as per the piecewise function. The If-Else loop is shown in Fig. 10.42.
21. Delete the first (blue) and second node (transparent) that has the Serial_Write_Value action as this is not needed. The Serial_Write_Value condition is now in the If-Else loop. Format the Serial_Write_Value action in the If-Else by pressing enter after each semicolon. The changes are shown in Fig. 10.43.

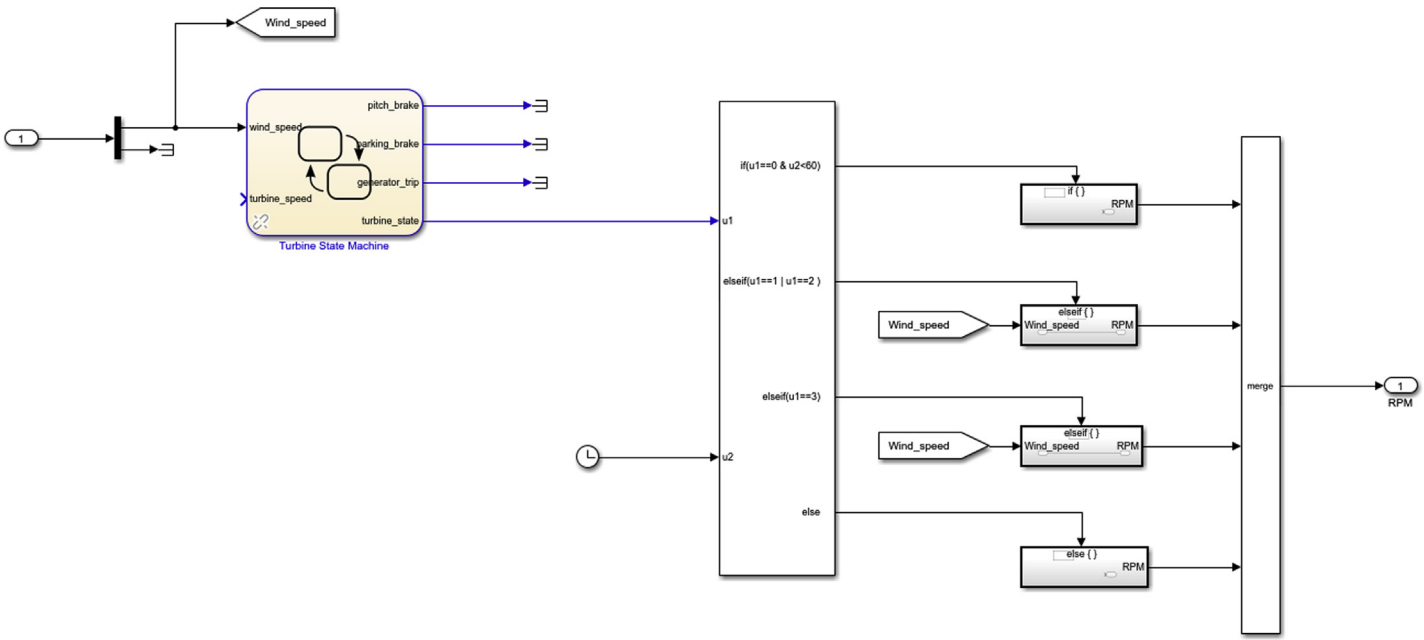


Figure 10.33 Merge block inputs and output.

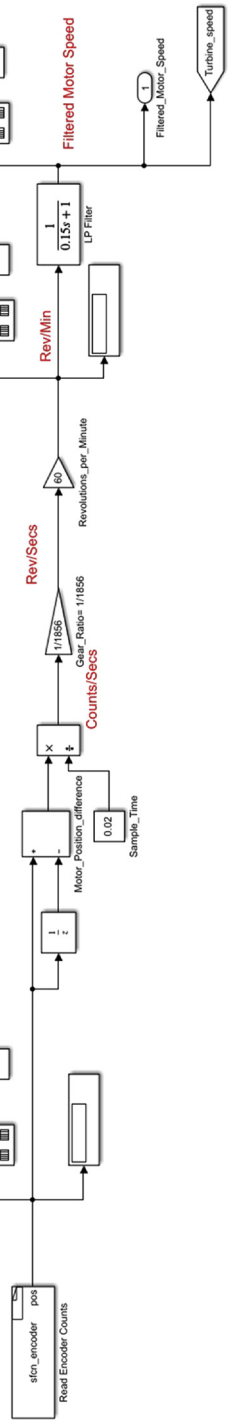


Figure 10.34 Turbine_speed Goto flag.

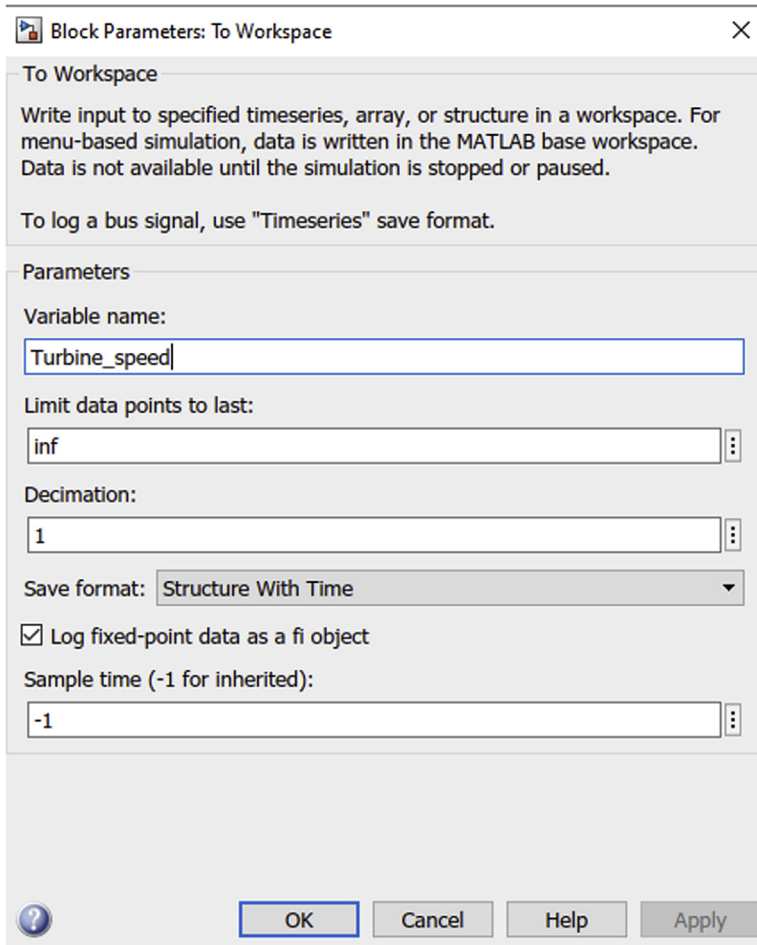


Figure 10.35 Save format for ToWorkspace variables.

22. Go outside the stateflow chart and add two inports. Rename the first inport as Wind_speed, third inport as time, and the fourth one as State as shown in Fig. 10.44.
23. Navigate to the top level of the model and add a **Clock** block, a **From** block, and two **Rate Transition** blocks. Rename the **From** block as State and connect them to the time and State inputs of the Serial_Data_Transmit_to_ESP32 subsystem. In addition to this, delete the arrow coming from the Arduino Mega Running MPC subsystem to the Wind_speed input of the Serial_Data_Transmit_to_ESP32 subsystem. Add another **From** block and rename it as Wind_speed. The changes are shown in Fig. 10.45.

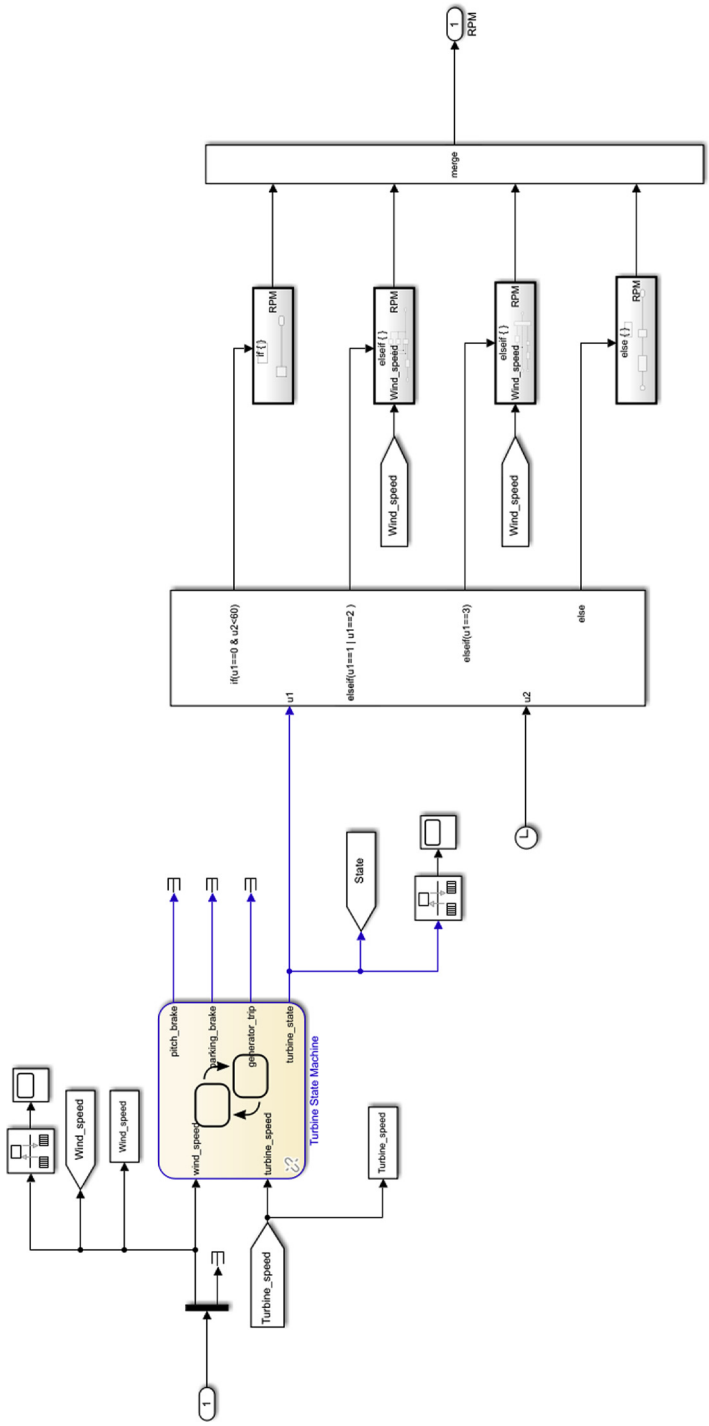


Figure 10.36 Completed Chapter_10/Motor_Control/Motor_Speed_Reference_Generator/Main_Controller.

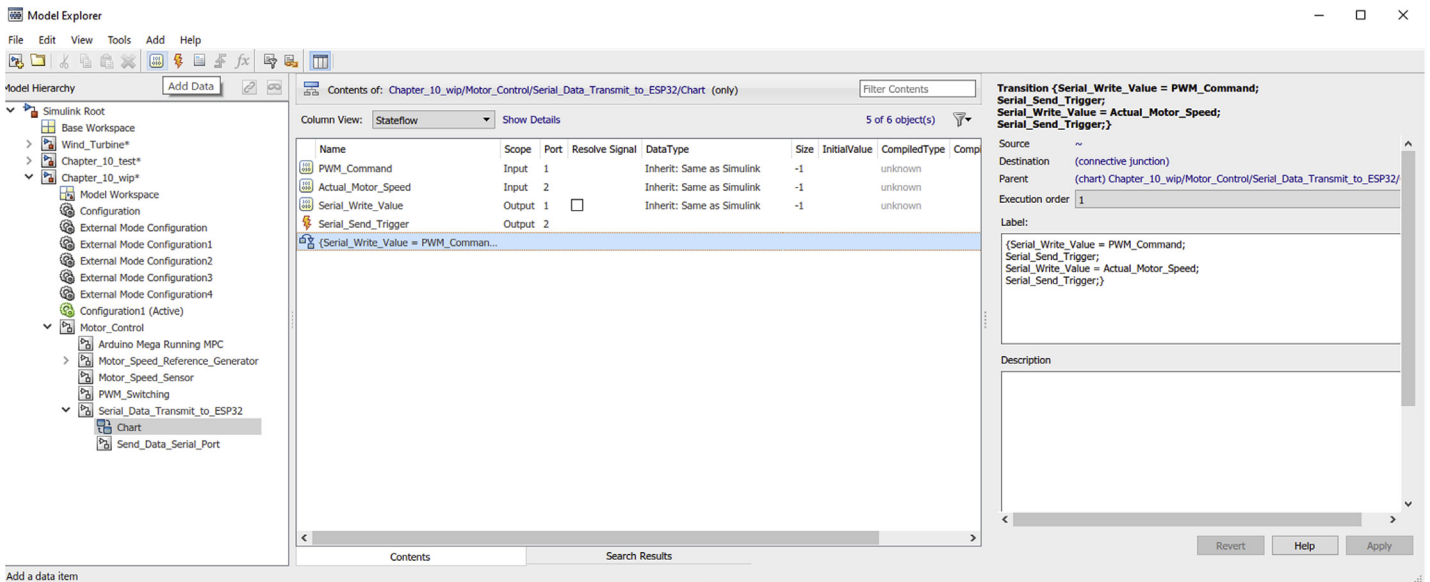


Figure 10.37 Add Data option in Model Explorer.

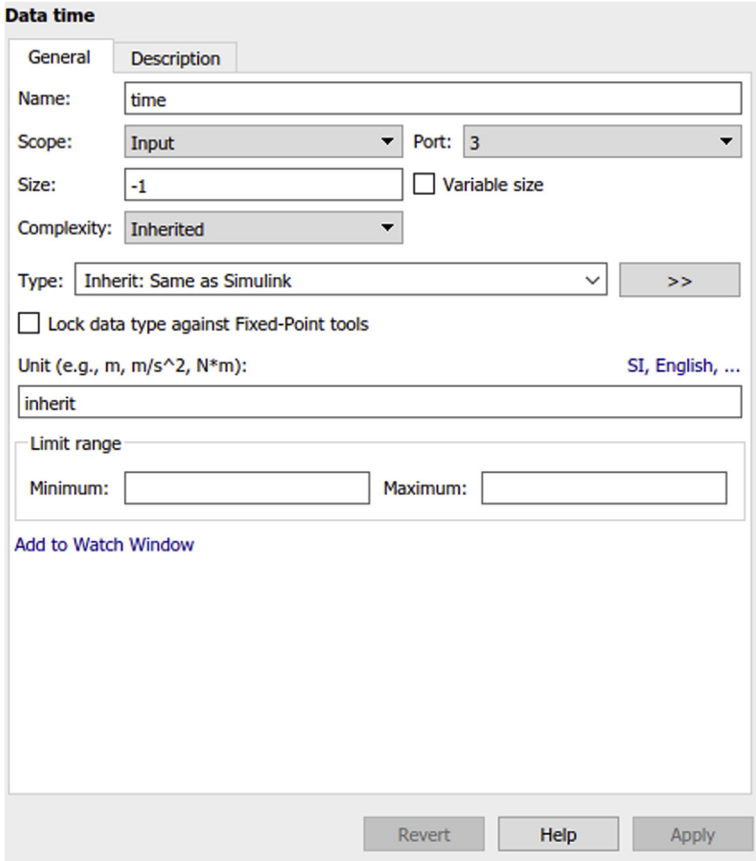


Figure 10.38 "Time" parameter settings.

24. The changes for the DC motor driver model are complete. Connect the Arduino board to the USB port making sure the right port is selected for this model. Set the simulation time to 130 s, and press play to simulate the results. Double click on the wind speed scope in Chapter_10/Motor_Control/Motor_Speed_Reference_Generator/Main_Controller and the filtered motor speed scope in Chapter_10/Motor_Control/Motor_Speed_Sensor to visualize the results. The scope outputs for wind speed and motor speed are shown in Figs. 10.46 and 10.47, respectively.

25. Notice that with the hardware prototype, the overshoot impact is smaller due to how the PID logic was tuned for the DC motor driver. Also, the Wind_speed and the Turbine_speed are now saved to the MATLAB workspace as structure due to the **ToWorkspace** variables that were created in step 16. Save these two structures into a separate .mat file named as chapt10.mat as it would be needed for the digital twin C code generation verification. This concludes the DC motor driver model changes section.

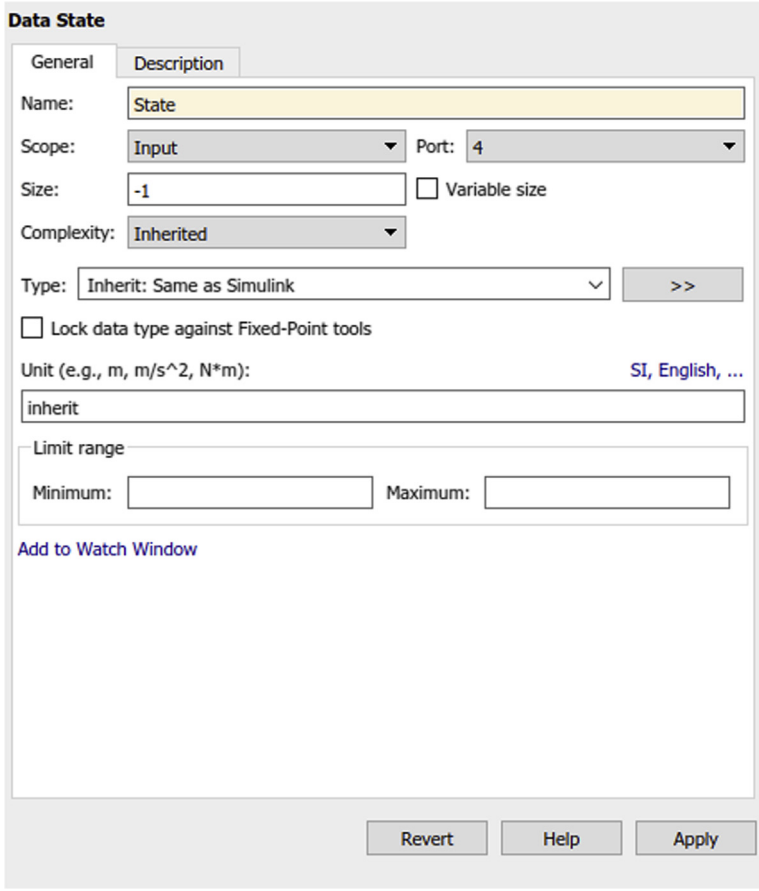


Figure 10.39 "State" parameter settings.

10.4.2 DC motor hardware communication to AWS

The communication between the DC motor hardware model (now renamed as Chapter_10.slx) to ESP32 and then to AWS cloud has already been handled in the DC motor chapter. However, there are some modifications that need to be made in the Arduino script for ESP32 to send messages to AWS.

1. Open Arduino IDE script that was used to send messages to AWS in the previous chapter. Rename it as Chapter_10_ESP32_AWS.
2. Press Ctrl+F and replace PWM_Command with Wind_speed.
3. Press Ctrl+F and find "if(message_counter==30)". Then under that statement, add the following statements as shown in Fig. 10.48. This makes sure that the previous wind speed and motor speed values are not 0 since the messages sent to ESP32 are only sent when system has entered steady state; hence, previous messages cannot be set as 0 as before in the DC motor chapter.

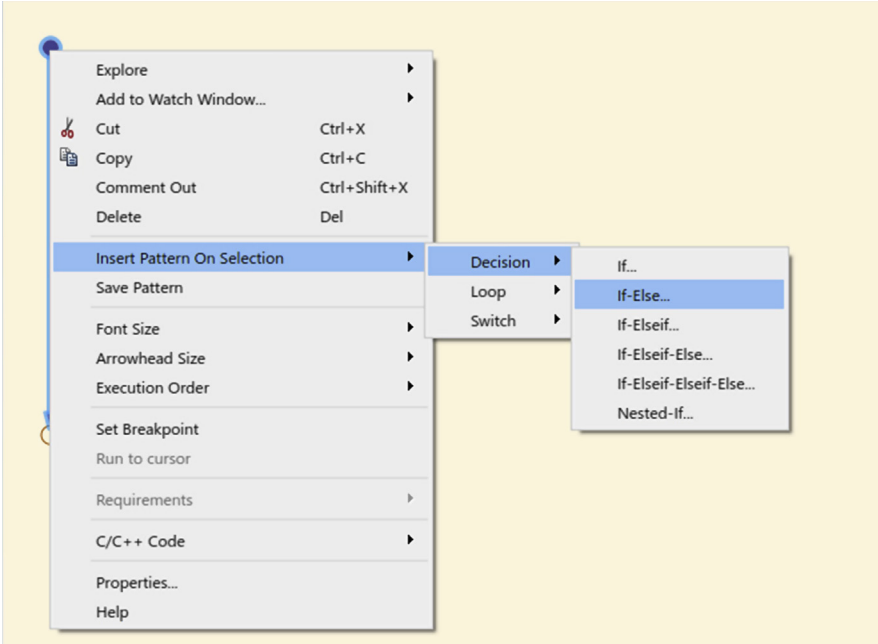


Figure 10.40 Selecting the If-Else decision pattern for stateflow.

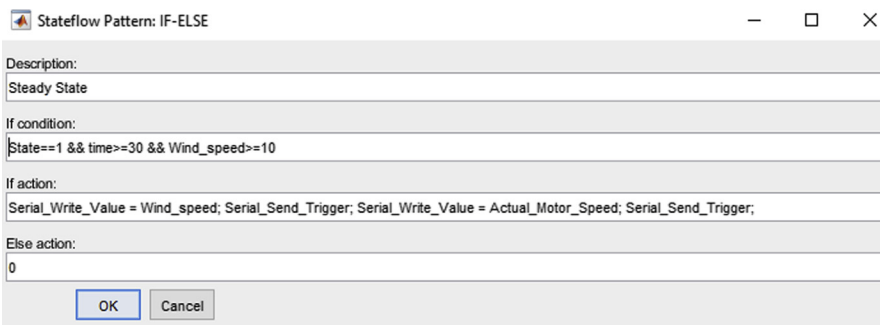


Figure 10.41 Conditions for the If-Else.

Note: The only time the "count=count+1" is referenced is in Fig. 10.48. If the variable "count" is not defined, set "int count=0" before the void setup function.

4. Connect the ESP32 to the USB port on the computer and run the Arduino script. Once, the messages "Connected to AWS" and "Subscribe Successful" are seen in the COM serial monitor of the Arduino IDE, connect the Arduino Mega 2560 to a USB port and run the "Chapter_10.slx" on the microcontroller by pressing play. The messages in the COM Serial Monitor will come up as shown in Fig. 10.49.

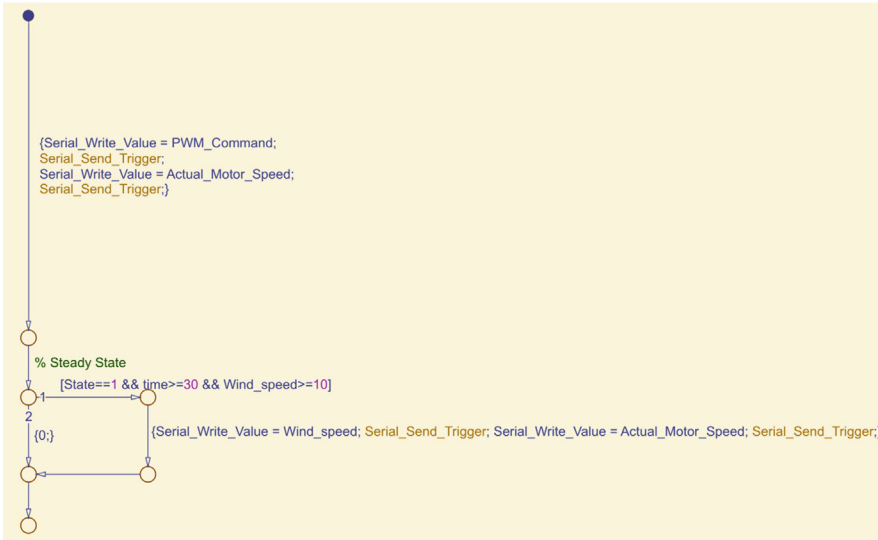


Figure 10.42 Added If-Else loop.

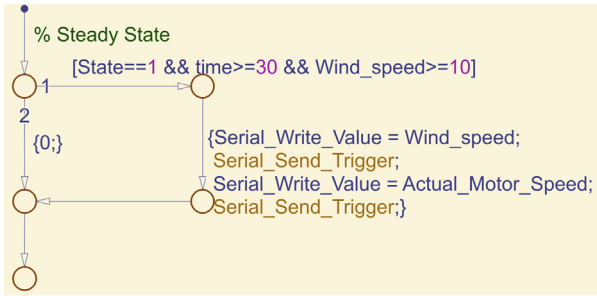


Figure 10.43 Formatted If-Else loop.

Note: Ignore any email notification you may receive by AWS Lambda function. The AWS Lambda is still using the C code for the DC motor digital twin; hence, that C Code needs to be changed to the Wind_Turbine.slx generated code.

10.5 Deploying the Simscape™ digital twin model to the AWS cloud and performing Off-BD

This section requires a Linux OS to generate the C code for the Wind_Turbine digital twin model. The Linux OS used in this chapter is Ubuntu. Once booted into Ubuntu and after launching MATLAB, follow these steps below:

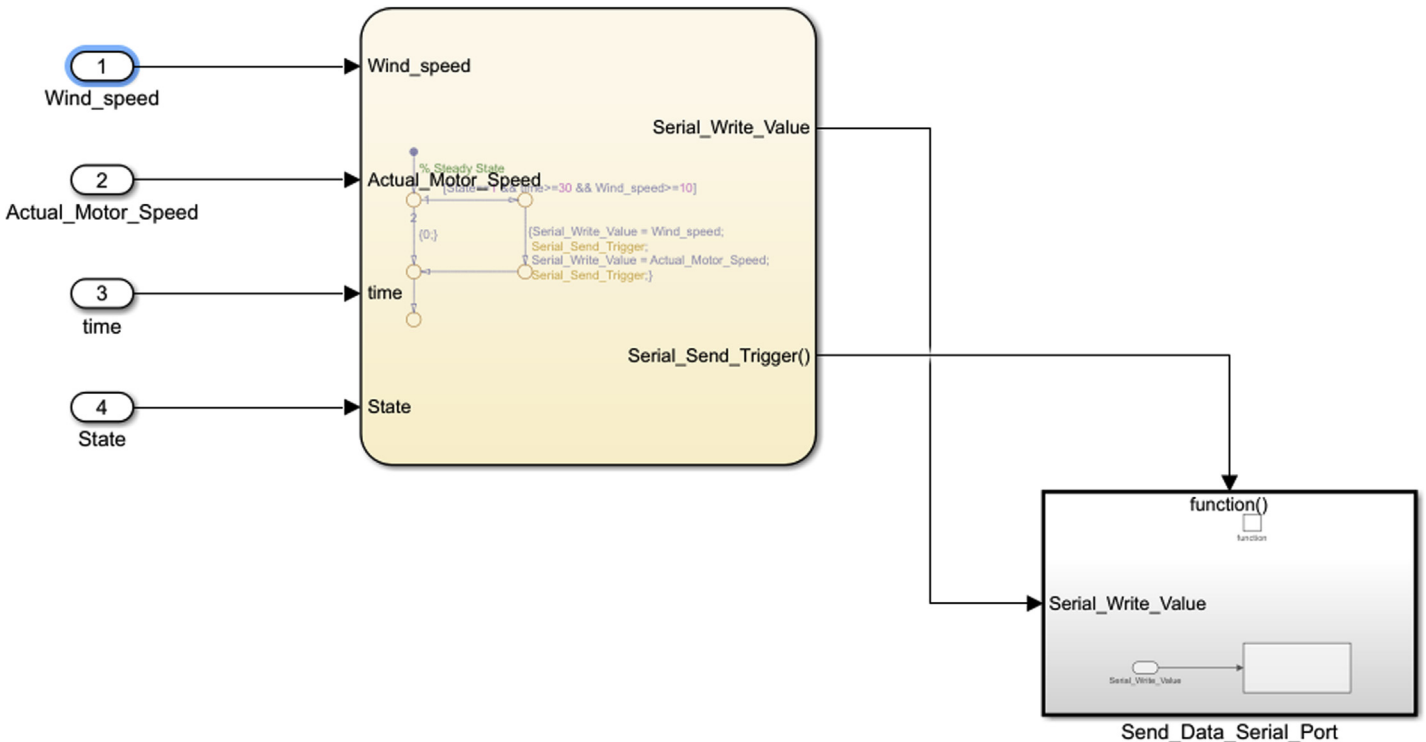


Figure 10.44 First, third, and fourth input for stateflow.

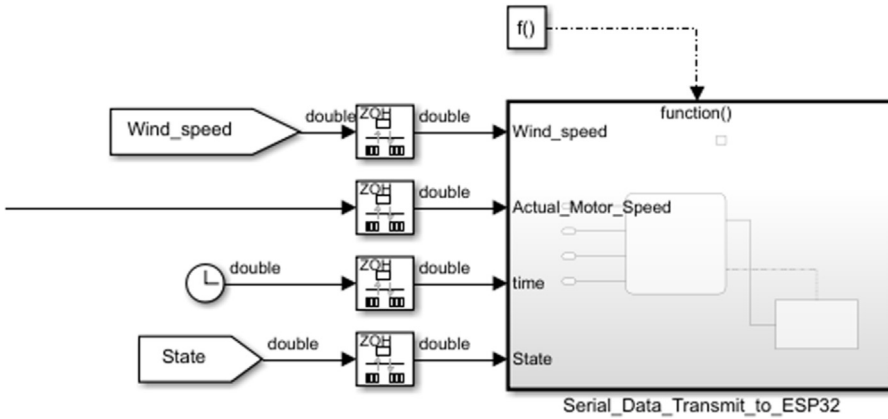


Figure 10.45 Connections to the time and state ports of the Serial_Data_Transmit_to_ESP32 subsystem.

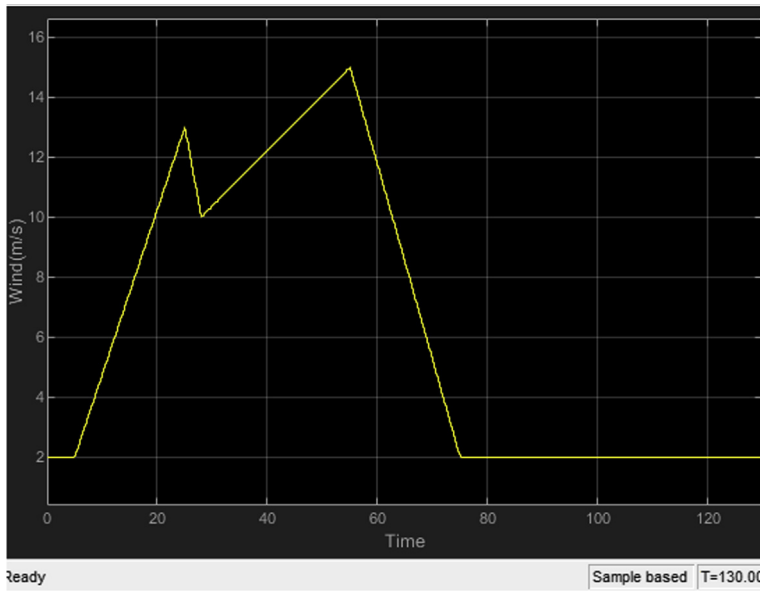


Figure 10.46 Wind speed input on DC motor driver model.

1. Open the Wind_Turbine.slx model and perform the changes done in Section 10.3.3 if not already done so for this model.
2. In the top level of the model, delete the Scopes block and outport 1. This is not needed for code generation. Also delete the SLRT block as mentioned in Section 10.3.1.
3. Go to Wind_Turbine/Turbine Input/Wind and double click on the Wind Input Signal Generator block. Right click on the Wind speed signal and select delete to delete the signal. The modified Signal builder block should only have the Wind Direction signal as shown in Fig. 10.50.

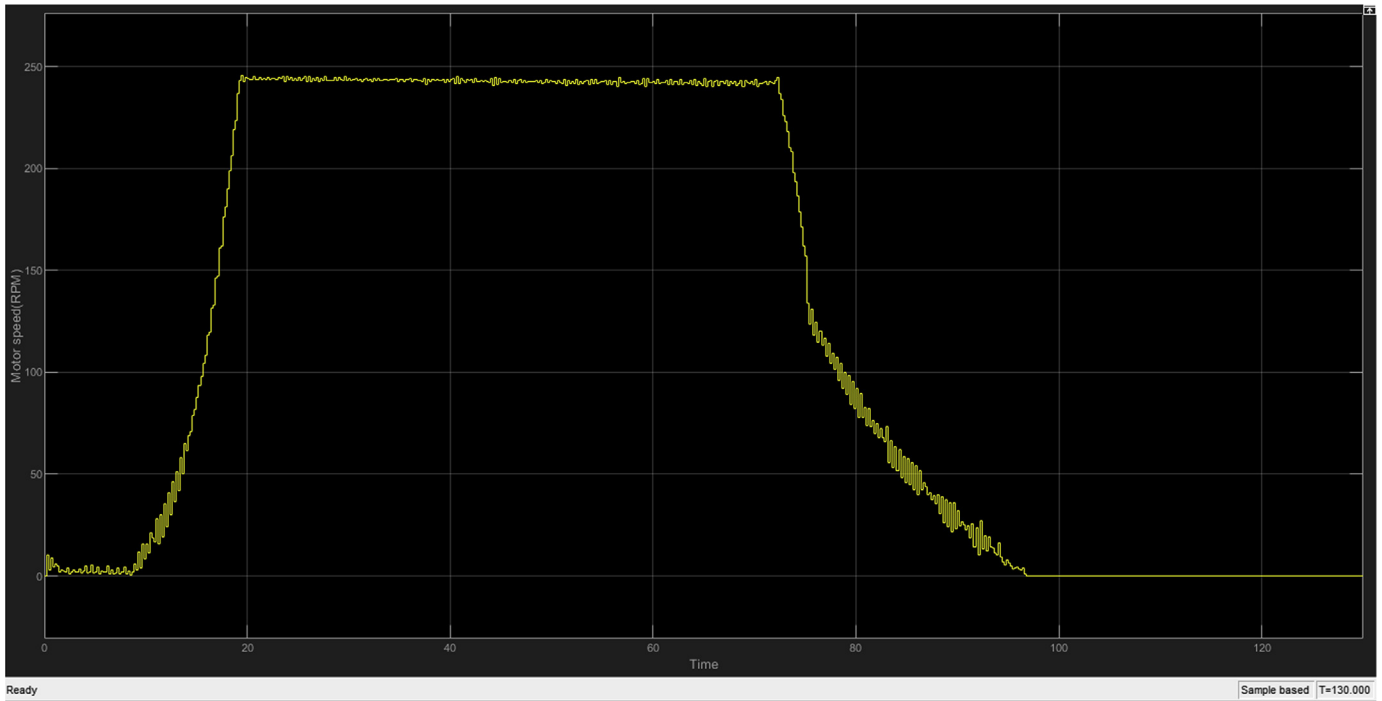


Figure 10.47 DC motor speed RPM on DC motor driver model.

```
// If message_counter = 30, that means we have received 3 seconds data. The below section of code makes a JSON structure with the data in the format
//   "{\"state\":{\"desired\":{\"Input\":[data],\"Output\":[data]}}}
if(message_counter ==30)
{
  // Increment the counter to indicate the total message count
  count = count + 1;
  // Previous wind speed and previous actual motor speed values the firs time the messages are sent to ESP32 in steady state.
  if (count ==1)
    {
    prev_Wind_speed=Wind_speed[0];
    prev_Actual_Motor_Speed=Actual_Motor_Speed[0];
    Serial.print("Entering stead state");
    }
  sprintf(payload,"{\"state\":{\"desired\":{\"Input\":[\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\",\"%d\"
  prev_Wind_speed,Wind_speed[0],Wind_speed[1],Wind_speed[2],Wind_speed[3],Wind_speed[4],Wind_speed[5],Wind_speed[6],Wind_speed[7],Wind_speed[8],Wind_speed[9],Wind_speed[10],
  Wind_speed[11],Wind_speed[12],Wind_speed[13],Wind_speed[14],Wind_speed[15],Wind_speed[16],Wind_speed[17],Wind_speed[18],Wind_speed[19],Wind_speed[20],
  Wind_speed[21],Wind_speed[22],Wind_speed[23],Wind_speed[24],Wind_speed[25],Wind_speed[26],Wind_speed[27],Wind_speed[28],Wind_speed[29],
  prev_Actual_Motor_Speed,Actual_Motor_Speed[0],Actual_Motor_Speed[1],Actual_Motor_Speed[2],Actual_Motor_Speed[3],Actual_Motor_Speed[4],Actual_Motor_Speed[5],Actual_Motor_Speed[6],Actual_Motor_Speed[7],Actual_Motor_
  Actual_Motor_Speed[11],Actual_Motor_Speed[12],Actual_Motor_Speed[13],Actual_Motor_Speed[14],Actual_Motor_Speed[15],Actual_Motor_Speed[16],Actual_Motor_Speed[17],Actual_Motor_Speed[18],Actual_Motor_Speed[19],Actual
  Actual_Motor_Speed[21],Actual_Motor_Speed[22],Actual_Motor_Speed[23],Actual_Motor_Speed[24],Actual_Motor_Speed[25],Actual_Motor_Speed[26],Actual_Motor_Speed[27],Actual_Motor_Speed[28],Actual_Motor_Speed[29]);
```

Figure 10.48 Modification to ESP32 script to set previous input and output messages.

```
Connected to wifi
Connected to AWS
Subscribe Successfull
Entering stead state{"state":{"desired":{"Input":["10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10","10"],"Output":["244","244",
Publish failed
{"state":{"desired":{"Input":["10","10","10","10","10","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11"],"Output":["243","242","244","246","242","2
Publish Message:{"state":{"desired":{"Input":["10","10","10","10","10","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11"],"Output":["243","242","244
{"state":{"desired":{"Input":["11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","12","12"],"Output":["244","242","241","243","244","2
Publish Message:{"state":{"desired":{"Input":["11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","11","12","12"],"Output":["244","242","241
{"state":{"desired":{"Input":["12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12"],"Output":["242","243","243","243","242","2
Publish Message:{"state":{"desired":{"Input":["12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12"],"Output":["242","243","243
{"state":{"desired":{"Input":["12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","13","13","13","13","13","13","13","13"],"Output":["241","242","244","243","242","2
Publish Message:{"state":{"desired":{"Input":["12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","13","13","13","13","13","13","13","13"],"Output":["241","242","244
{"state":{"desired":{"Input":["13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13"],"Output":["243","242","241","242","243","2
Publish Message:{"state":{"desired":{"Input":["13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13"],"Output":["243","242","241
{"state":{"desired":{"Input":["13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14"],"Output":["241","242","244","242","241","2
Publish Message:{"state":{"desired":{"Input":["13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","13","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14"],"Output":["241","242","244
{"state":{"desired":{"Input":["14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14"],"Output":["243","243","242","241","242","2
Publish Message:{"state":{"desired":{"Input":["14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14"],"Output":["243","243","242
{"state":{"desired":{"Input":["14","14","14","14","14","14","14","14","14","14","14","15","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","13","13","13","13","13"],"Output":["241","241","243","243","241","2
Publish Message:{"state":{"desired":{"Input":["14","14","14","14","14","14","14","14","14","14","14","15","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","14","13","13","13","13","13"],"Output":["241","241","243
{"state":{"desired":{"Input":["13","13","13","13","13","13","13","13","13","13","13","13","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","11","11","11"],"Output":["242","244","242","241","241","2
Publish Message:{"state":{"desired":{"Input":["13","13","13","13","13","13","13","13","13","13","13","13","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","12","11","11","11"],"Output":["242","244","242
```


Figure 10.49 Messages sent to AWS.

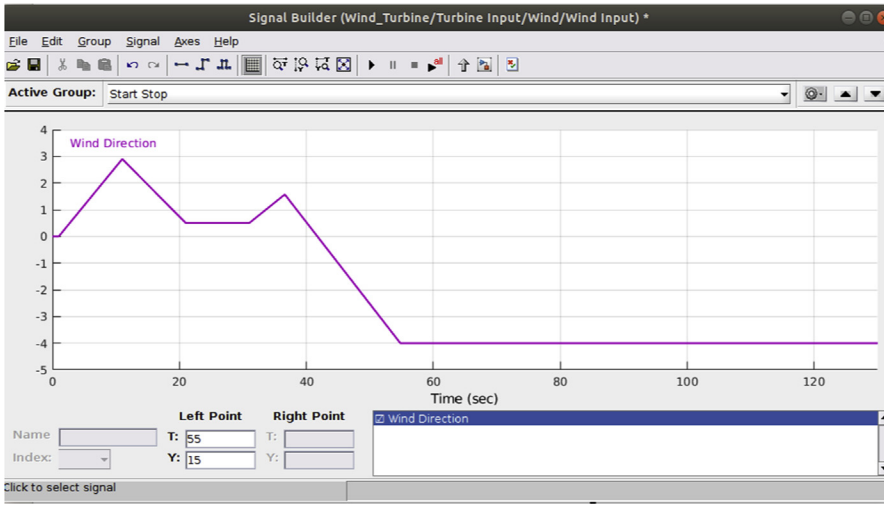


Figure 10.50 Modified signal builder for wind direction.

4. For the first input of the **Mux** block in Wind_Turbine/Turbine Input/Wind, add an **inport**, rename it as WS, and connect it to the first input of the **Mux** block as shown in Fig. 10.51. Copy the WS **inport** and paste it in Wind_Turbine/Turbine Input as shown in Fig. 10.52. Repeat the same for Wind_Turbine top level. Connect the WS **inport** to the Turbine Input subsystem as shown in Fig. 10.53.
5. In the top level of the model, place a **From** and an **outport** block. Rename the **From** flag as Gen_Spd_rpm and the **outport** block as Turb. The **Goto** flag for this is already in Wind_Turbine/Nacelle/Generator/Full. Connect the **From** block to the **outport** and place them above the Phasor 60 Hz block as shown in Fig. 10.54. This shall serve as the turbine speed outport for the given wind speed input from WS inport.
6. In the top level of the model, double click on the phasor 60 Hz block and input the settings as shown in Fig. 10.55. Notice that the discrete sampling time is set as 0.0015 s, which is different than the 0.1 s execution rate of the function block sending messages to ESP32 in Chapter 10.slx. The reason 0.1 s cannot be set as a sampling rate for this Wind_Turbine digital twin model is because the Asynchronous Machine block in Wind_Turbine/Nacelle/Generator will not be able to converge on a solution if the sample time is too large. There would be some further modifications needed in the generated C code to address the difference between the sampling rates of the motor hardware and the digital twin shown later in this chapter.

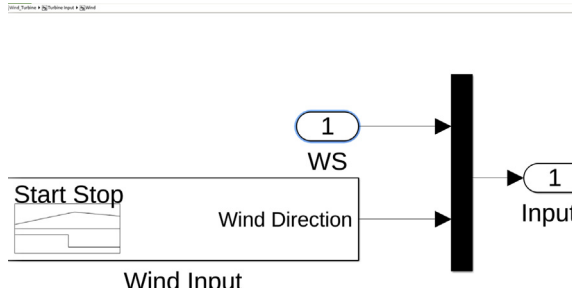


Figure 10.51 WS inport to Mux block.

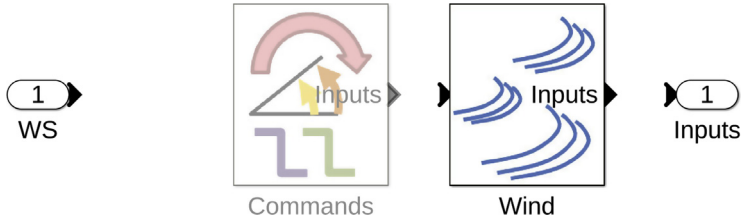


Figure 10.52 WS inport in Wind_Turbine/Turbine_Input.

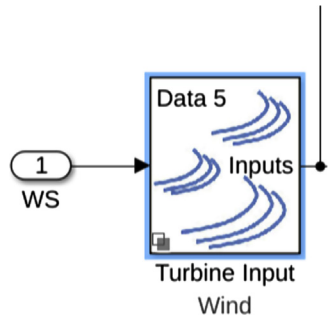


Figure 10.53 WS inport in top level.

Figure 10.54 Turb outport in top level.

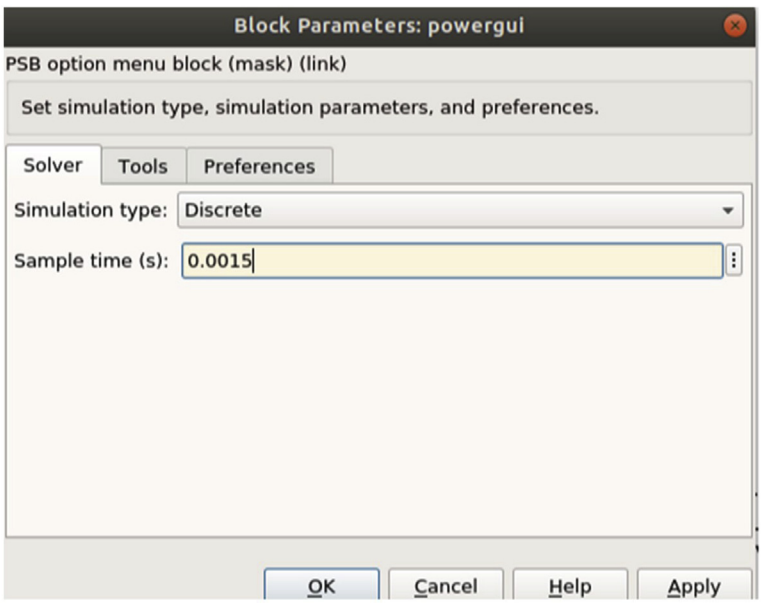


Figure 10.55 Discrete sampling time settings.

7. In the top level of the model, in the bottom right corner, double click on the Pitch Power On block and set the sample time as 0.0015 s as shown in Fig. 10.56.
8. Save and close the model. Open the model again, and in the top bar menu of Simulink, go to Code->C/C++ Code->Embedded Coder as shown in Fig. 10.57. The quick start shall guide you to the process of generating C code for the model while selecting the correct settings for the C Code. Once the C code is generated, open the ert_main.c file in the Wind_Turbine_ert_rtw folder that is created after code generation. The two main functions of the generated ert_main.c file are shown in Figs. 10.58 and 10.59.
9. For this step, replace the contents of the Wind_Turbine ert_main.c after the header file definitions with the contents of the ert_main.c that was used in the DC motor chapter. Some modifications need to be made in the Wind_Turbine ert_main.c after the replacement as shown on Figs. 10.60−10.63. The new changes in the Wind_Turbine ert_main.c are shown in the left half of each figure.

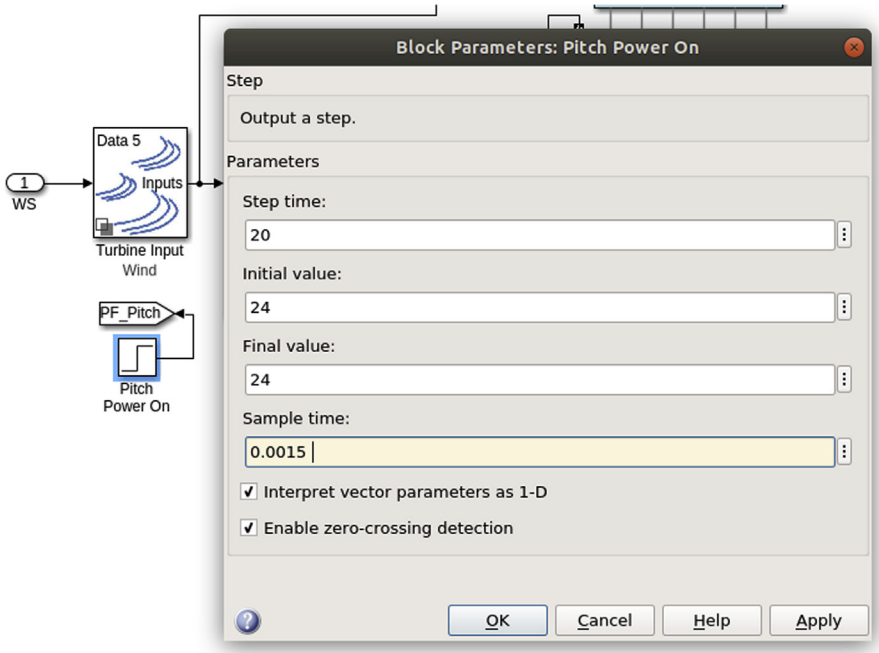


Figure 10.56 Pitch Power On settings.

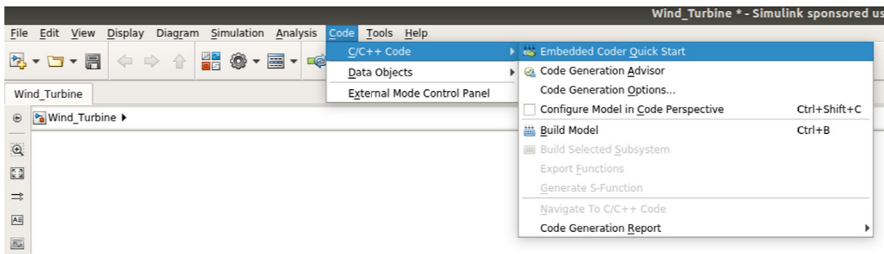


Figure 10.57 Embedded coder quick start.

```
#include <stddef.h>
#include <stdio.h>                      /* This ert_main.c example uses printf/fflush */
#include "Wind_Turbine.h"               /* Model's header file */
#include "rtwtypes.h"

/*
 * Associating rt_OneStep with a real-time clock or interrupt service routine
 * is what makes the generated code "real-time".  The function rt_OneStep is
 * always associated with the base rate of the model.  Subrates are managed
 * by the base rate from inside the generated code.  Enabling/disabling
 * interrupts and floating point context switches are target specific.  This
 * example code indicates where these should take place relative to executing
 * the generated code step function.  Overrun behavior should be tailored to
 * your application needs.  This example simply sets an error status in the
 * real-time model and returns from rt_OneStep.
 */
void rt_OneStep(void);
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = false;

  /* Disable interrupts here */

  /* Check for overrun */
  if (OverrunFlag) {
    rtmSetErrorStatus(rtM, "Overrun");
    return;
  }

  OverrunFlag = true;

  /* Save FPU context here (if necessary) */
  /* Re-enable timer or interrupt here */
  /* Set model inputs here */

  /* Step the model for base rate */
  Wind_Turbine_step();

  /* Get model outputs here */

  /* Indicate task complete */
  OverrunFlag = false;

  /* Disable interrupts here */
  /* Restore FPU context here (if necessary) */
  /* Enable interrupts here */
}

/*
 * The example "main" function illustrates what is required by your
```

Figure 10.58 void rt_OnStep function in generated ert_main.c file.

There are a few observations in the ert_main.c differences listed below:

(a) In Fig. 10.60, it can be seen for the DC motor hardware ert_main.c that there were input and output arguments to pass into the step function in parenthesis. This may not be done for the Wind_Turbine ert_main.c due to how the code was generated. The input and output argument instead will be explicitly mentioned in Fig. 10.63.

(b) In Fig. 10.61, line 148 of the Wind_Turbine ert_main.c file, 7100 samples are being used instead of 100 because Wind_Turbine needs about 7100 samples at a sampling rate of

```
/*
 * The example "main" function illustrates what is required by your
 * application code to initialize, execute, and terminate the generated code.
 * Attaching rt_OneStep to a real-time clock is target specific.  This example
 * illustrates how you do this relative to initializing the model.
 */
int_T main(int_T argc, const char *argv[])
{
  /* Unused arguments */
  (void)(argc);
  (void)(argv);

  /* Initialize model */
  Wind_Turbine_initialize();

  /* Simulating the model step behavior (in non real-time) to
   *  simulate model behavior at stop time.
   */
  while ((rtmGetErrorStatus(rtM) == (NULL)) && !rtmGetStopRequested(rtM)) {
    rt_OneStep();
  }

  /* Disable rt_OneStep() here */
  return 0;
}

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

Figure 10.59 int main function in generated ert_main.c file.

0.0015 s to reach steady state. This would be equivalent to 10 s data as it was in the DC motor chapter. The next 2000 samples are the new data received from the hardware equivalent to 3 s of data as before.

Note: Even though we are asking for more samples, we will still only send 30 samples sent from ESP32 to AWS. An interpolating function would need to be used in the lambda function to interpolate data to 9100 samples.

(c) In Fig. 10.62, line 199 of the Wind_Turbine ert_main.c file, if the sample count is less than 7100, then the padd_hw_input_wind and padded_hw_output_rpm arrays are fed with previous data.

(d) In Fig. 10.63, line 222 of the Wind_Turbine ert_main.c file, the input wind speed is explicitly defined as rTU.WS=padded_hw_input_wind[ii] where rTU.WS is the WS inport in the model. The same is done for equating arg_Turb_speed to rtY.Turb where rtY.Turb is the Turb outport in the model. Note that line 225 is "printf("Digital Twin Predicted Motor speed [%d] with Wind Input %f is = %f\n",ii,padded_hw_input_wind[ii],arg_Turb_speed_1"); In Fig. 10.63, there is no terminate function for the Wind_Turbine ert_main.c file.

Wind_Turbine

```
40  void rt_OneStep(void);
41  void rt_OneStep(void)
42  {
43    static boolean_T OverrunFlag = false;
44
45
46
47
48
49
50
51    /* Disable interrupts here */
52
53    /* Check for overrun */
54    if (OverrunFlag) {
55      rtmSetErrorStatus(rtM, "Overrun");
56      return;
57    }
58
59    OverrunFlag = true;
60
61    /* Save FPU context here (if necessary) */
62    /* Re-enable timer or interrupt here */
63    /* Set model inputs here */
64
65    /* Step the model for base rate */
66    Wind_Turbine_step();
67
68    /* Get model outputs here */
69
70    /* Indicate task complete */
71    OverrunFlag = false;
72
73    /* Disable interrupts here */
74    /* Restore FPU context here (if necessary) */
75    /* Enable interrupts here */
76  }
```

DC_Motor

```
40  void rt_OneStep(void);
41  void rt_OneStep(void)
42  {
43    static boolean_T OverrunFlag = false;
44
45    /* '<Root>/PWM_Input' */
46    static real_T arg_PWM_Input = 0.0;
47
48    /* '<Root>/RPM_Output' */
49    static real_T arg_RPM_Output;
50
51    /* Disable interrupts here */
52
53    /* Check for overrun */
54    if (OverrunFlag) {
55      rtmSetErrorStatus(Chapter_9_Section_7_M, "Overrun");
56      return;
57    }
58
59    OverrunFlag = true;
60
61    /* Save FPU context here (if necessary) */
62    /* Re-enable timer or interrupt here */
63    /* Set model inputs here */
64
65    /* Step the model */
66    Chapter_9_Section_7_step(&arg_PWM_Input, &arg_RPM_Output);
67
68    /* Get model outputs here */
69
70    /* Indicate task complete */
71    OverrunFlag = false;
72
73    /* Disable interrupts here */
74    /* Restore FPU context here (if necessary) */
75    /* Enable interrupts here */
76  }
```

Figure 10.60 ert_main.c differences between DC motor chapter and wind turbine chapter with regard to input and output arguments.

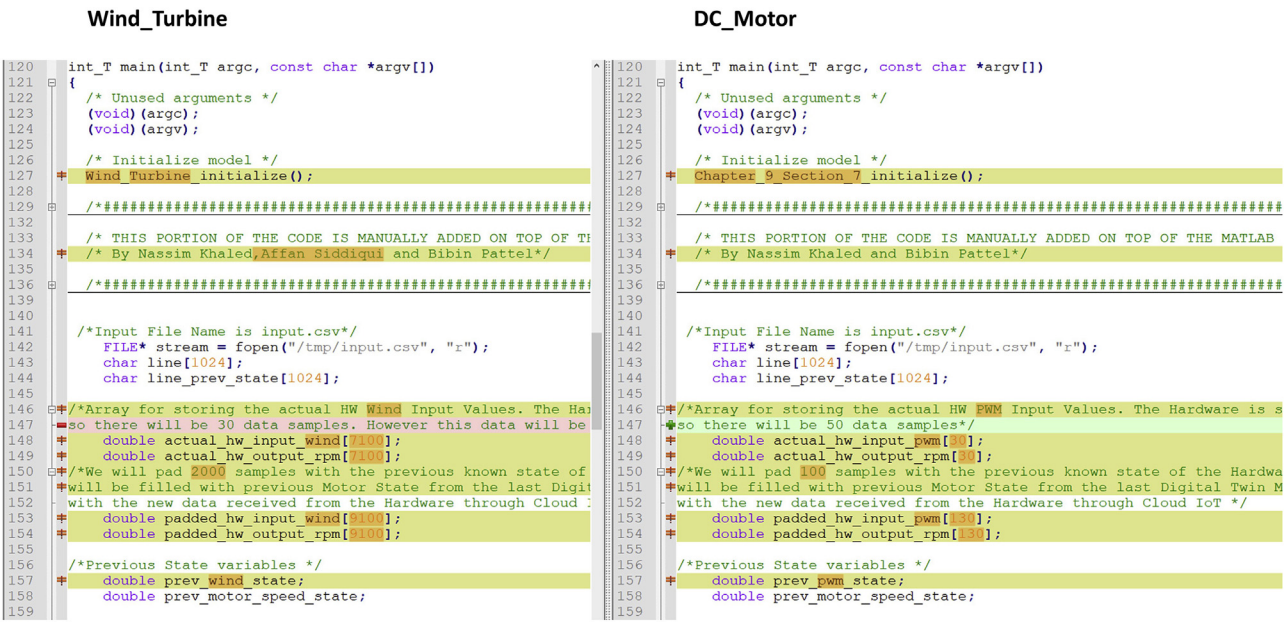


Figure 10.61 ert_main.c differences between DC motor chapter and wind turbine chapter with regard to storing information.

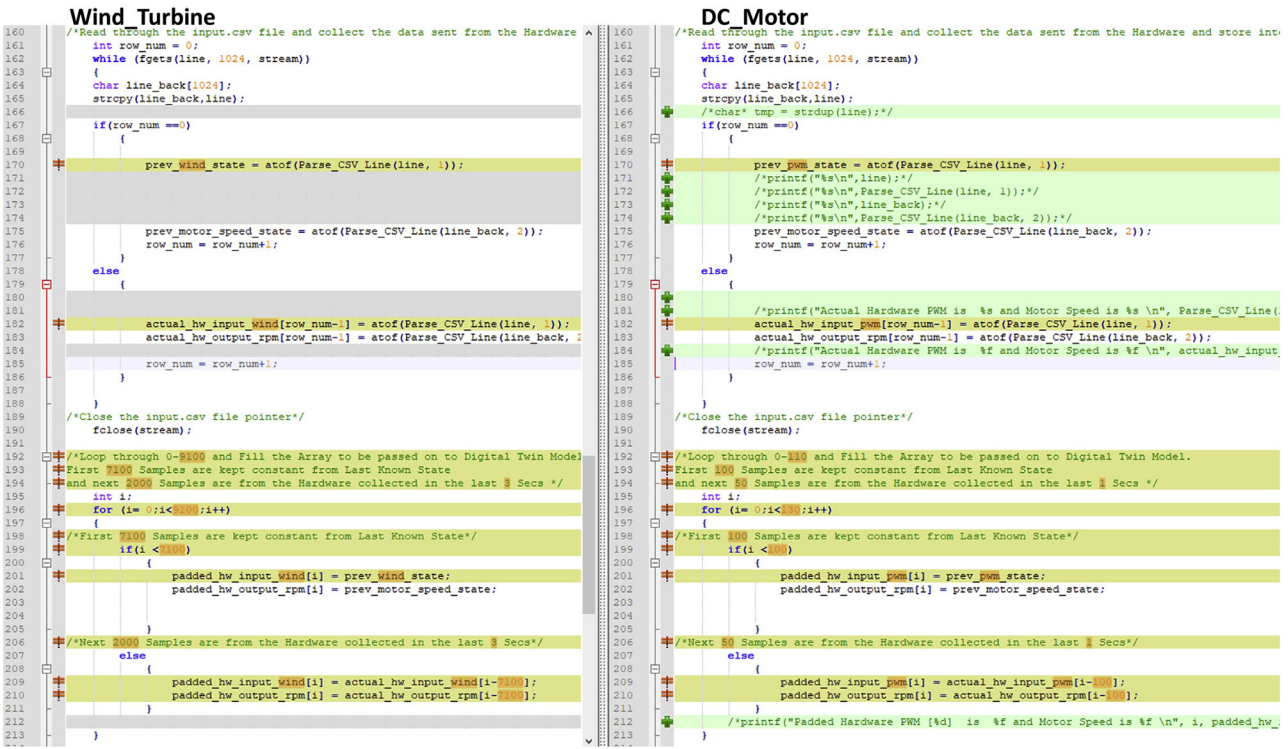


Figure 10.62 ert_main.c differences between DC motor chapter and wind turbine chapter with regard to processing input and output.

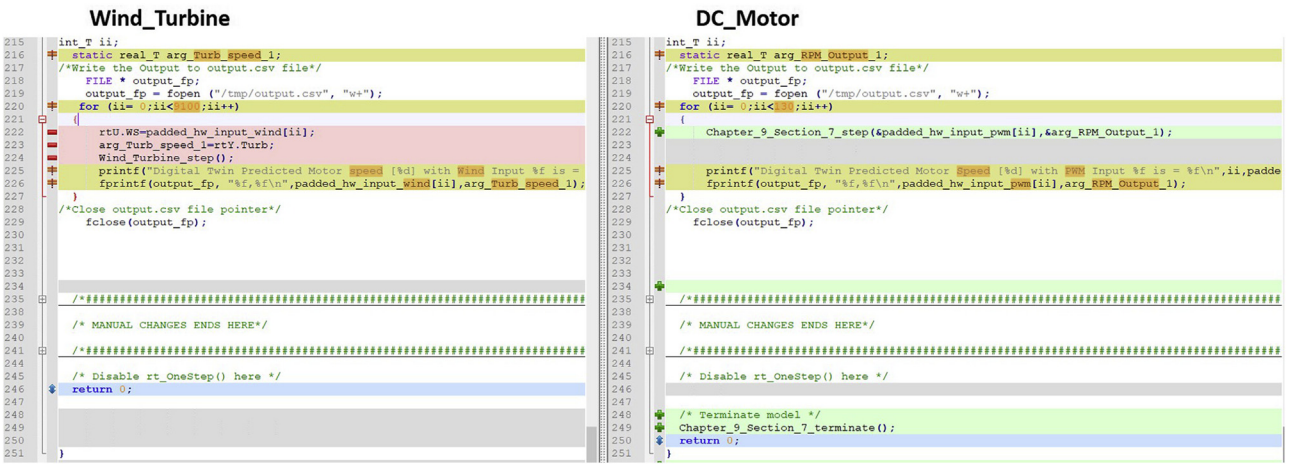


Figure 10.63 ert_main.c differences between DC motor chapter and wind turbine chapter with regard to terminate function.

10. Next, navigate to the instrumented folder in the Wind_Turbine_ert_rtw folder and type the following commands in sequence in the MATLAB workspace to generate a new executable based on the modifications made in step 9:
    i. !cp Wind_Turbine.mk Makefile
    ii. !rm ert_main.o
    iii. !make -f Makefile

Once done, navigate to outside Wind_Turbine_ert_rtw folder and notice a new Wind_Turbine executable.

11. After the executable is created, it is time to verify this executable through a MATLAB script. Use the same .m script that was used to verify the DC motor chapter. There are a few changes that need to be made to this script as shown in Figs. 10.64 and 10.65.

There are a few observations to see in the MATLAB script:

(a) In Fig. 10.64, data is only read from 1501 to 3135 elements as this is the range where the turbine is supposed to enter steady state.
(b) Notice that in Fig. 10.64 for the Wind_Turbine, resample function is used instead of itnerp1 function as there will be NaN values that may come up with interp1 function for the Wind_Turbine code. The resampling function will set the time divisions to 0.0015 s and appropriately compute a y value based on the resampled time.
(c) In line 55 of the Wind_Turbine, notice that the count threshold is set as 2002 instead of 32 since 3 s of data would be equivalent to 0.0015 s*2002.
d) In Fig. 10.65, line 81 of the Wind_Turbine, only the 7100 to 9100 elements are used since the first 7100 values are the previous wind speed input to make the system to go to steady state.

12. Run the verification script and observe in the MATLAB window, for 7100 samples, the same wind speed is being used bringing the generator speed to steady-state value. Notice that there is an overshoot in the RPM values as mentioned in Section 10.3. The script will finish plotting the results after approximately 5 min. The plotted RMSE is shown in Fig. 10.66.
13. With the executable verified, it is time to develop the Python function that will be used in AWS Lambda. Use the same python script that was used for the DC motor chapter. Modifications are need to be made to this python script as shown in Figs. 10.67 and 10.68.

Here are a few observations:

(a) In Fig. 10.67, line 8 of the Wind_Turbine, numpy library would be needed for the interpolation.
(b) In Fig. 10.67, from line 17 onwards of the Wind_Turbine, the data from the hardware are first sent to a temporary input_1.csv file. Then the input_1.csv is read and interpolated to match the sampling rate of 0.0015 s sampling rate of the digital twin.
(c) In Fig. 10.68, the RMSE error processing function is adjusted to read data from the 7100 to 9100 elements of the interpolated data.

14. Save the above developed Lambda function into a new folder. In this case, we named the folder to be **Chapter_10** under the folder **Digtal_Twin_Simscape_book.** Copy the Digital Twin compiled executable also to this folder. Fig. 10.69 shows a set of Linux commands to give the necessary executable permissions and packaging of the Lambda function and

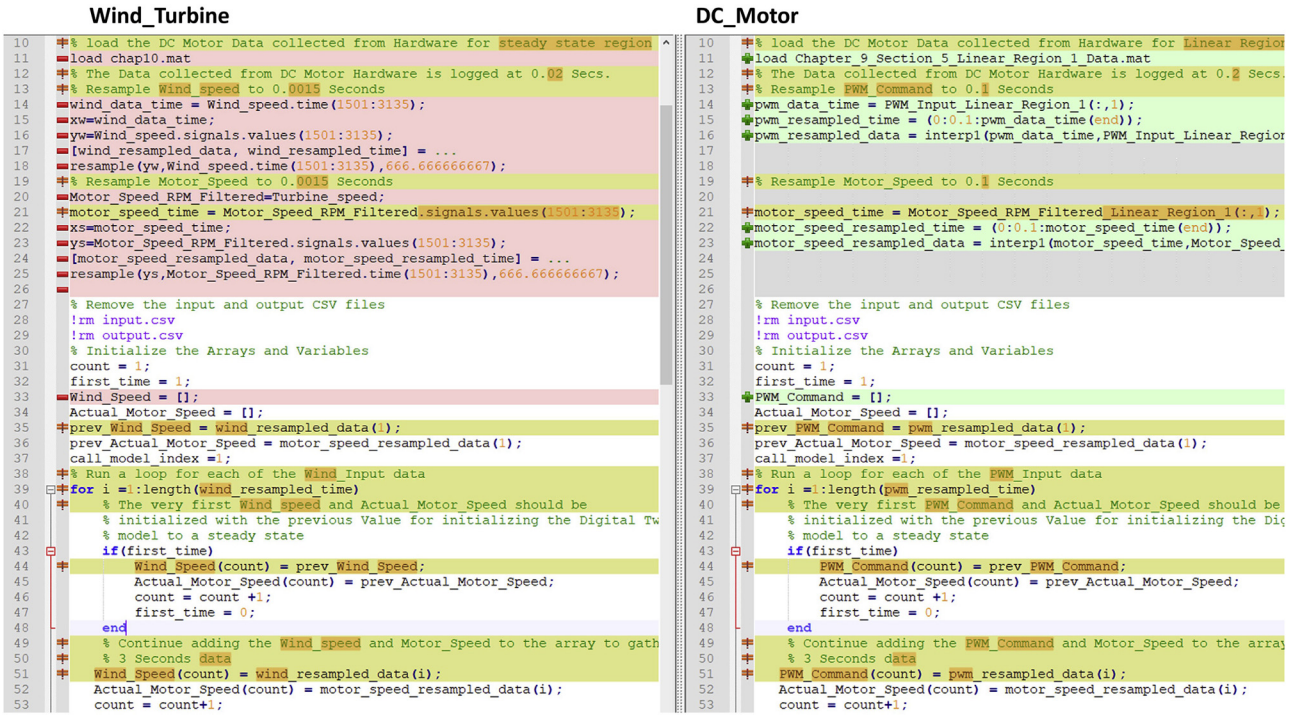


Figure 10.64 Executable verification script differences between DC motor chapter and wind turbine chapter with regard to processing input and output.

Wind_Turbine

```
% When count becomes 2002 the we have gathered 3 Seconds data
if(count ==2002)
    % Update the previous values
    prev_Wind_Speed = Wind_Speed(count-1);
    prev_Actual_Motor_Speed = Actual_Motor_Speed(count-1);
    count =1;
    first_time = 1;
     % Print the collected 3 Seconds data to input.csv file
    fid = fopen('input.csv','w+');
    for j =1:2001
        fprintf(fid,'%s,%s\n',num2str(Wind_Speed(j)),num2str(Actual_Motor_Speed(j)));
    end
    fclose(fid);
    pause(1);
    % Move the input.csv file to /tmp folder
    !cp input.csv /tmp
    % Run the Digital Twin Model
    !./Wind_Turbine
    % Digital Twin Model creates an output.csv file , copy it back from
    % the /tmp folder to the current working directory
    !cp /tmp/output.csv .
    pause(1);
    % Read the output.csv file
    model_predicted_data = csvread('output.csv');
    temp = model_predicted_data(:,2);
    % Calculate RMSE between Actual Hardware and Model Predicted generator
    % Speed
    Error = Actual_Motor_Speed(2:end) - temp(7101:9100)';
    Squared_Error = Error.^2;
    Mean_Squared_Error = mean(Squared_Error);
    Root_Mean_Squared_Error = sqrt(Mean_Squared_Error);
%     Plot the RMSE of Speed Prediction for every 3 Seconds when the
%     Digital Twin Model Runs
    figure(101);
    plot(call_model_index,Root_Mean_Squared_Error,'b','Marker','Diamond','linewidth',2
    hold all
    call_model_index = call_model_index+1;
    % Delete the input.csv and output.csv for the next 3 Seconds data
    !rm input.csv
    !rm output.csv
    pause(1);

end
end
```

DC_Motor

```
% When count becomes 32 the we have gathered 3 Seconds data
if(count ==32)
    % Update the previous values
    prev_PWM_Command = PWM_Command(count-1);
    prev_Actual_Motor_Speed = Actual_Motor_Speed(count-1);
    count =1;
    first_time = 1;
     % Print the collected 3 Seconds data to input.csv file
    fid = fopen('input.csv','w+');
    for j =1:31
        fprintf(fid,'%s,%s\n',num2str(PWM_Command(j)),num2str(Actual_Mo
    end
    fclose(fid);
    pause(1);
    % Move the input.csv file to /tmp folder
    !cp input.csv /tmp
    % Run the Digital Twin Model
    !./Chapter_9_Section_7
    % Digital Twin Model creates an output.csv file , copy it back from
    % the /tmp folder to the current working directory
    !cp /tmp/output.csv .
    pause(1);
    % Read the output.csv file
    model_predicted_data = csvread('output.csv');
    temp = model_predicted_data(:,2);
    % Calculate RMSE between Actual Hardware and Model Predicted Motor
    % Speed
    Error = Actual_Motor_Speed(2:end) - temp(101:130)';
    Squared_Error = Error.^2;
    Mean_Squared_Error = mean(Squared_Error);
    Root_Mean_Squared_Error = sqrt(Mean_Squared_Error);
    % Plot the RMSE of Speed Prediction for every 3 Seconds when the
    % Digital Twin Model Runs
    figure(101);
    plot(call_model_index,Root_Mean_Squared_Error,'b','Marker','Diamond
    hold all
    call_model_index = call_model_index+1;
    % Delete the input.csv and output.csv for the next 3 Seconds data
    !rm input.csv
    !rm output.csv
    pause(1);

end
end
```

Figure 10.65 Executable verification differences between DC motor chapter and wind turbine chapter with regard to terminate function.

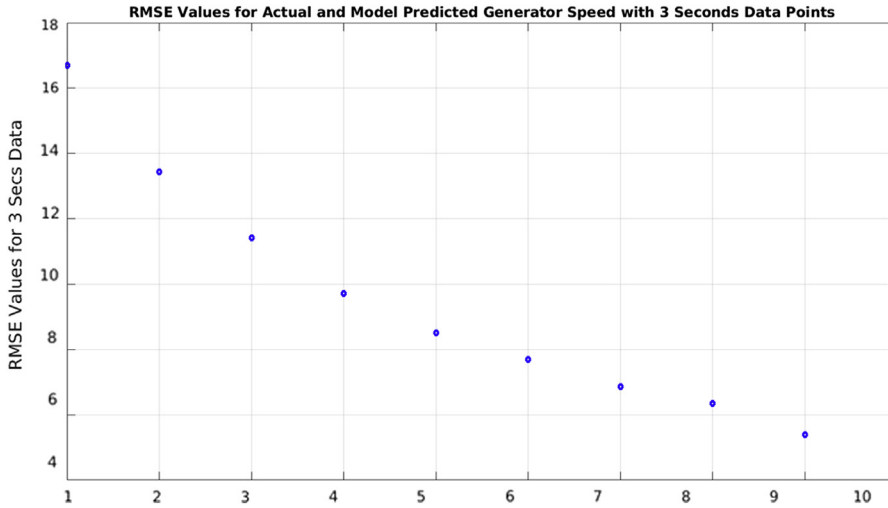


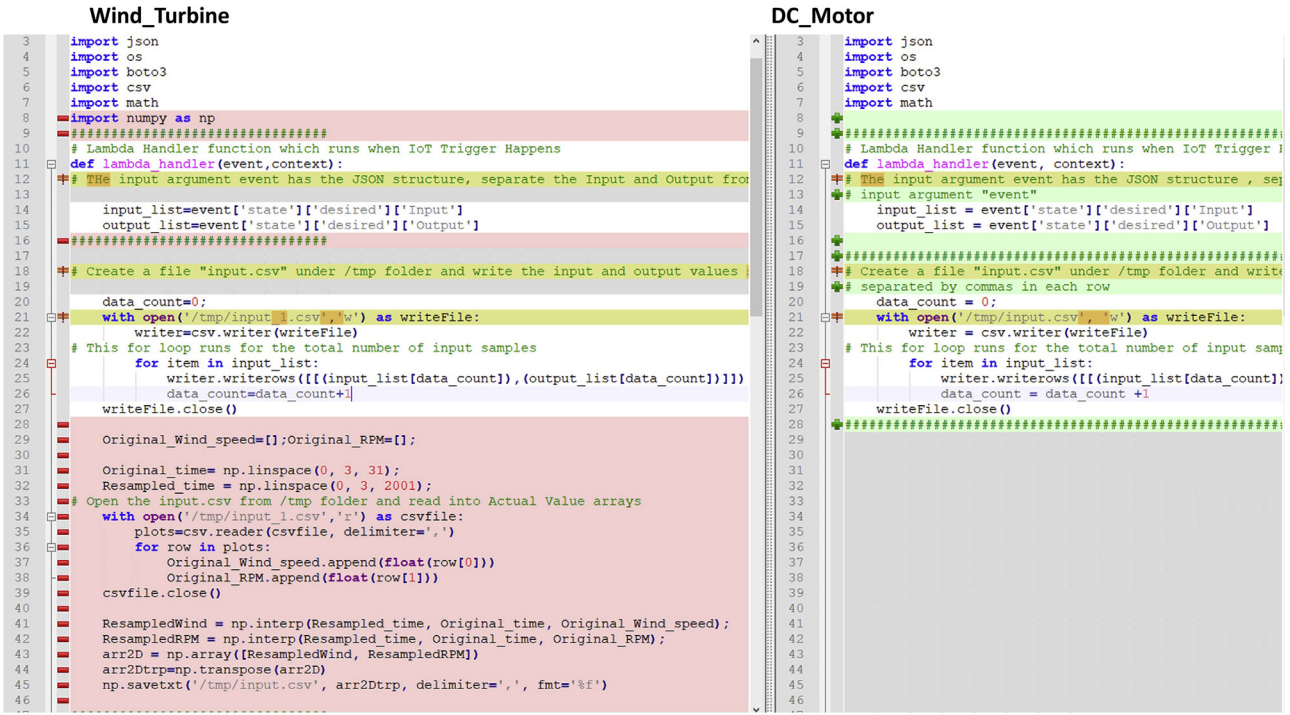
Figure 10.66 RMSE plot for verification script.

Digital Twin executable. The commands are explained below. Open a Linux terminal and run the below commands in steps:

- **a.** The ***ls*** command shows the folder has the compiled executable **Chapter_9_Section_7** and the lambda function **lambda_function.py**.
- **b.** Using the ***cp*** command, copy the executable to a new file name **Wind_Turbine_Digital_Twin,** which we used in the Lambda function.
- **c.** We need to give full read/write/executable permissions to all files in this folder for the AWS to be able to run it. Use the command ***sudo chmod -R 777 <folder_name>***.
- **d.** Run the command ***ls -l.*** This will list all the files and folders in the current folder and their permissions. Everything should be showing ***rwxrwxrwx***.
- **e.** Now package the lambda function and compiled executable using the command ***zip bundle.zip lambda_function.py Wind_Turbine***. This will zip the files ***lambda_function.py*** and ***Wind_Turbine*** into a file named ***bundle.zip***.
- **f.** After the bundle.zip is created once again repeat the step c to allow read/write/executable permissions to the ***bundle.zip*** file as well.
- **g.** The ***ls*** command will show the newly created ***bundle.zip*** file as well. Now we are ready to deploy the Lambda function to AWS and do the final testing.

15. Now open a web browser and then the AWS Management Console and then open the dc_motor_digital_twin lambda function created in the last chapter. Select the option from the drop down "Upload a .zip file." Browse and select the bundle.zip file that was packaged in step 14. Click on "Save" when done.

16. There are a couple of more steps to execute before running the lambda function. The lambda python script includes numpy as a library; however, AWS on its own does not recognize that library unless that library was zipped as well. However, there is now a way in AWS such that his library can be used without the need of zipping it. In the Designer area, click on Layers and then click on Add a layer as shown in Fig. 10.70.




```
import json
import os
import boto3
import csv
import math
import numpy as np
################################
# Lambda Handler function which runs when IoT Trigger Happens
def lambda_handler(event,context):
# THe input argument event has the JSON structure, separate the Input and Output fro

    input_list=event['state']['desired']['Input']
    output_list=event['state']['desired']['Output']
################################

# Create a file "input.csv" under /tmp folder and write the input and output values

    data_count=0;
    with open('/tmp/input_1.csv', 'w') as writeFile:
        writer=csv.writer(writeFile)
# This for loop runs for the total number of input samples
        for item in input_list:
            writer.writerows([[(input_list[data_count]),(output_list[data_count])]])
            data_count=data_count+1
    writeFile.close()

    Original_Wind_speed=[];Original_RPM=[];

    Original_time= np.linspace(0, 3, 31);
    Resampled_time = np.linspace(0, 3, 2001);
# Open the input.csv from /tmp folder and read into Actual Value arrays
    with open('/tmp/input_1.csv','r') as csvfile:
        plots=csv.reader(csvfile, delimiter=',')
        for row in plots:
            Original_Wind_speed.append(float(row[0]))
            Original_RPM.append(float(row[1]))
    csvfile.close()

    ResampledWind = np.interp(Resampled_time, Original_time, Original_Wind_speed);
    ResampledRPM = np.interp(Resampled_time, Original_time, Original_RPM);
    arr2D = np.array([ResampledWind, ResampledRPM])
    arr2Dtrp=np.transpose(arr2D)
    np.savetxt('/tmp/input.csv', arr2Dtrp, delimiter=',', fmt='%f')
```

```
import json
import os
import boto3
import csv
import math

################################################
# Lambda Handler function which runs when IoT Trigger
def lambda_handler(event, context):
# The input argument event has the JSON structure , se
# input argument "event"
    input_list = event['state']['desired']['Input']
    output_list = event['state']['desired']['Output']

################################################
# Create a file "input.csv" under /tmp folder and writ
# separated by commas in each row
    data_count = 0;
    with open('/tmp/input.csv', 'w') as writeFile:
        writer = csv.writer(writeFile)
# This for loop runs for the total number of input samp
        for item in input_list:
            writer.writerows([[(input_list[data_count]]
            data_count = data_count +1
    writeFile.close()
################################################
```

Figure 10.67 Lambda function differences between DC motor chapter and wind turbine chapter with regard to interpolating the input data.

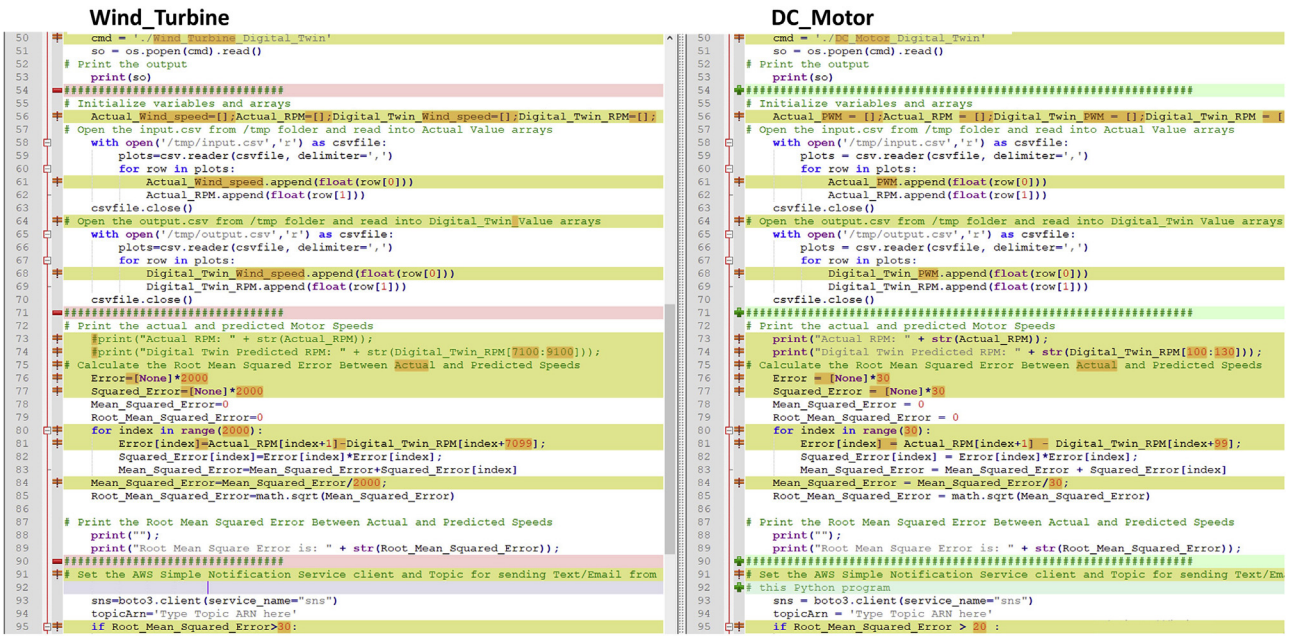


Figure 10.68 Lambda function differences between DC motor chapter and wind turbine chapter with regard to processing RMSE.

```
affan@affan-Lenovo-Y50-70:~/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10$ ls
lambda_function.py  Wind_Turbine
affan@affan-Lenovo-Y50-70:~/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10$ cp Wind_Turbine Wind_Turbine_Digital_Twin
affan@affan-Lenovo-Y50-70:~/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10$ ls
lambda_function.py  Wind_Turbine  Wind_Turbine_Digital_Twin
affan@affan-Lenovo-Y50-70:~/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10$ sudo chmod -R 777 /home/affan/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10
[sudo] password for affan:
affan@affan-Lenovo-Y50-70:~/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10$ ls -l
total 8024
-rwxrwxrwx 1 affan affan    4608 Jan  5 18:07 lambda_function.py
-rwxrwxrwx 1 affan affan 4103592 Jan  5 16:19 Wind_Turbine
-rwxrwxrwx 1 affan affan 4103592 Jan  5 18:20 Wind_Turbine_Digital_Twin
affan@affan-Lenovo-Y50-70:~/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10$ zip bundle.zip lambda_function.py Wind_Turbine_Digital_Twin
  adding: lambda_function.py (deflated 67%)
  adding: Wind_Turbine_Digital_Twin (deflated 65%)
affan@affan-Lenovo-Y50-70:~/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10$ sudo chmod -R 777 /home/affan/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10
affan@affan-Lenovo-Y50-70:~/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10$ ls -l
total 9436
-rwxrwxrwx 1 affan affan 1442546 Jan  5 18:24 bundle.zip
-rwxrwxrwx 1 affan affan    4608 Jan  5 18:07 lambda_function.py
-rwxrwxrwx 1 affan affan 4103592 Jan  5 16:19 Wind_Turbine
-rwxrwxrwx 1 affan affan 4103592 Jan  5 18:20 Wind_Turbine_Digital_Twin
affan@affan-Lenovo-Y50-70:~/Downloads/test/Wind_Turbine_Simulink/Wind_Turbine_Model_R18b/Chapter_10$
```

Figure 10.69 Linux terminal commands for packaging of lambda function and digital twin executable.

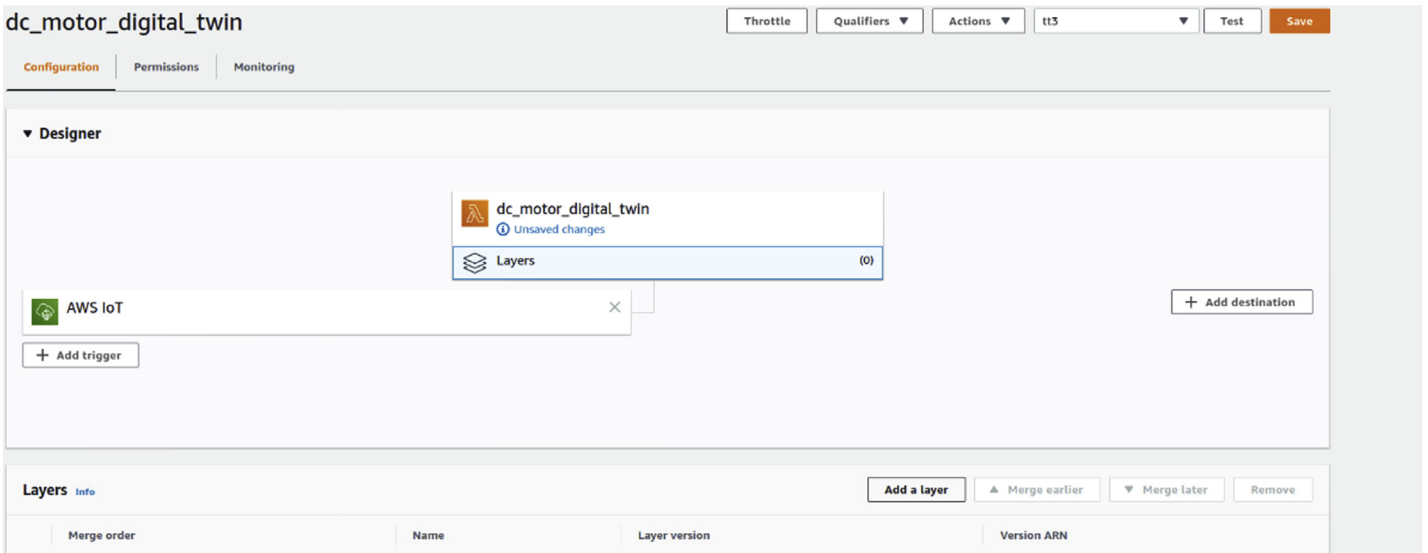


Figure 10.70 Layer options in AWS Lambda.

17. Then, in the Name field, from the drop down list, choose AWSLambda-Python37-SciPy1x as shown in Fig. 10.71. Choose the latest Version in the Version in drop down menu. Then click on Add. The layer is now added as noticed in the designer area.
18. Next, click on dc_motor_digital_twin in the Designer area and scroll to the Basic settings area shown in Fig. 10.72. Increase the Timeout to 1 min and the Memory (MB) to 3008 MB or to the maximum limit. The reasoning is because the lambda function takes some time to run (approximately 30−40 s). However, the lambda function triggers can run concurrently. As in, when the DC motor data are sent as a trigger to AWS Lambda, it will execute the function each time a trigger is sent without waiting for the previous trigger to end. Each execution will take about 30−40 s after which an email notification will be sent according to the AWS SNS service.

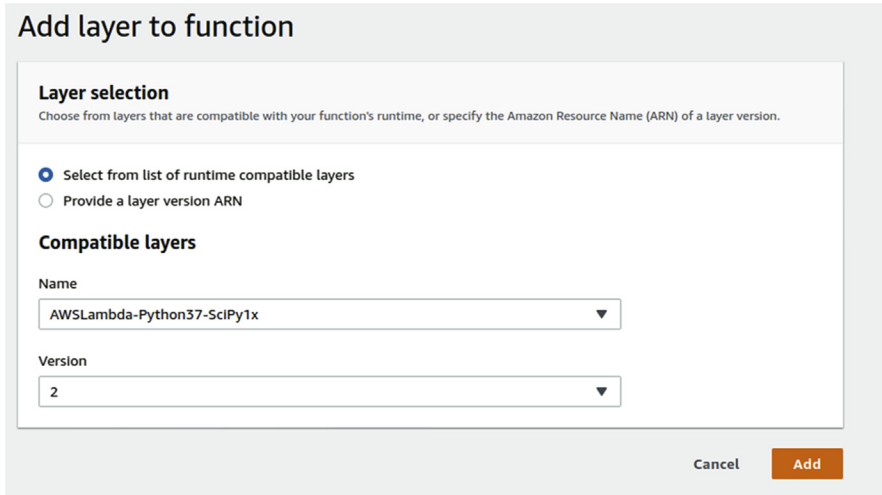


Figure 10.71 SciPy lambda layer.

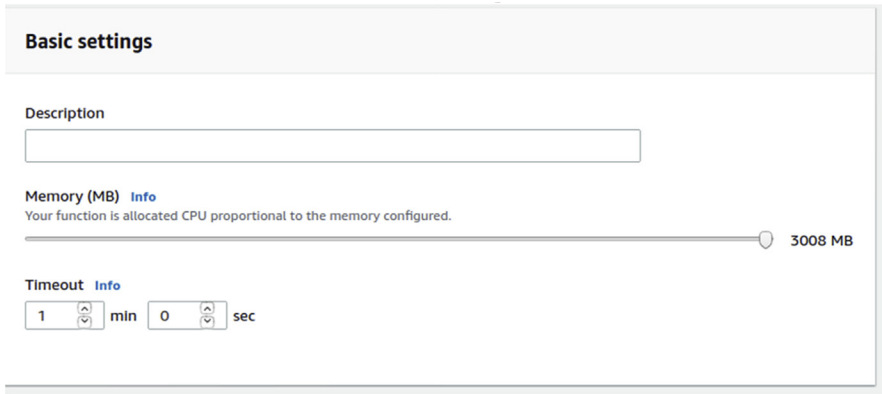


Figure 10.72 Memory size and timeout options for lambda function.

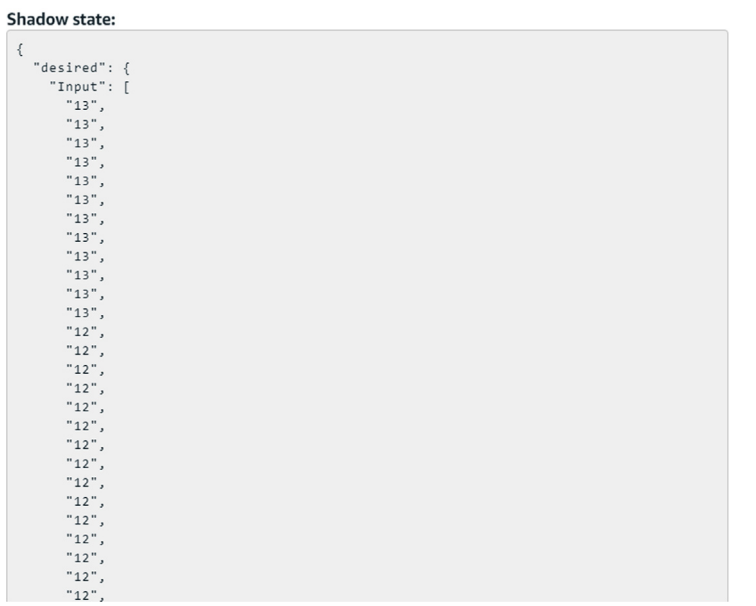


Figure 10.73 Shadow update in AWS.

19. This brings a conclusion to the deployment of the digital twin on AWS. We are now ready to test the hardware and digital twin. You can now switch to Windows OS where you saved the DC motor hardware model and the Arduino script. Connect the ESP32 and 2560 to the respective COM port and run the Arduino IDE script and the DC Motor hardware model on MATLAB. You should be seeing the messages now being sent to AWS when hardware has achieved steady state in AWS IoT Core digital_twin_thing shadow update as shown in Fig. 10.73. Once the lambda function finishes executing one trigger, you should receive an email according to SNS stating there is no failure since we have not introduced any failure into the system. Fig. 10.74 shows the messages for a healthy system.

20. Now introduce a failure into the system by disconnecting the power supply to the DC motor. When the DC motor hardware model is run, it will still send as per the conditions in Fig. 10.41 since there is no circular reference to motor speed. However, motor speed will show 0 RPM. Once the lambda function finishes executing, you will see the failure email messages as shown in Fig. 10.75.

10.6 Application problem

Modify the output of this chapter to use acceleration instead of RPM and compute the RMSE between the acceleration of the DC motor and digital twin generator when the system has entered steady state. You may observe that this comparison is easier than RPM as you will get an acceleration close to 0 when system as entered steady state.

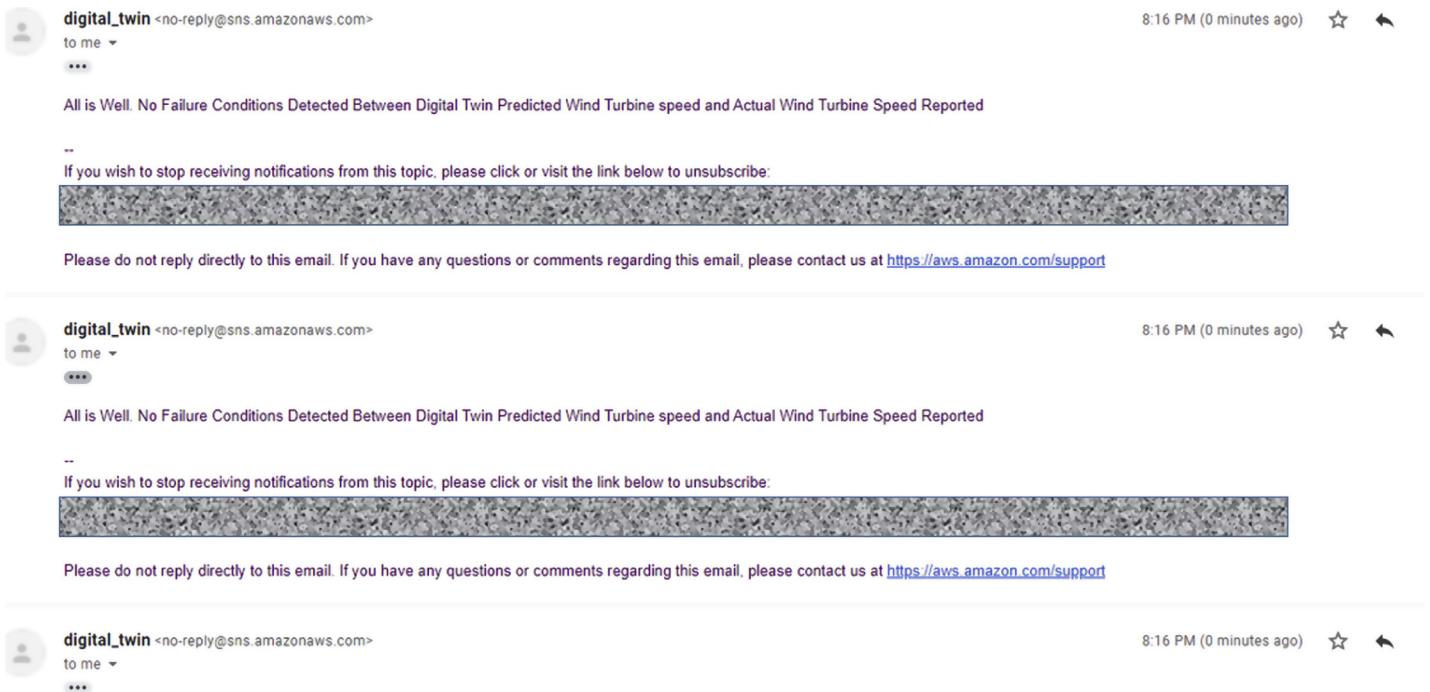


Figure 10.74 Email messages for a healthy system.

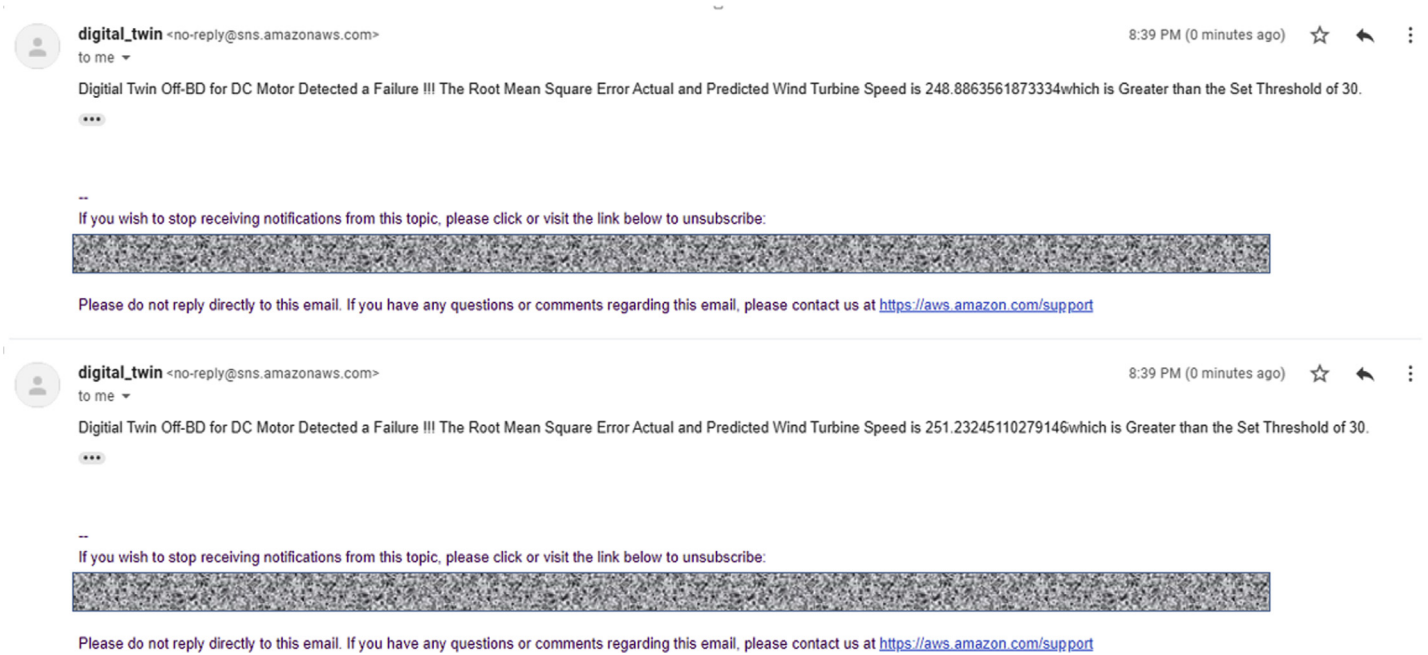


Figure 10.75 Email messages when a fault is introduced.

Hint: You may find the **Discrete Derivative** block useful for this problem. Keep in mind the difference in sample rates between the hardware driver model and the digital twin model when using this block.

Download the material for the chapter from MATLAB® Central. The final models and codes are provided in **Application_Problem_windturbine** folder.

Reference

[1] S. Miller, Wind Turbine Model, 2020. https://www.mathworks.com/matlabcentral/fileexchange/25752-wind-turbine-model. MATLAB Central File Exchange. Retrieved November 11, 2019.