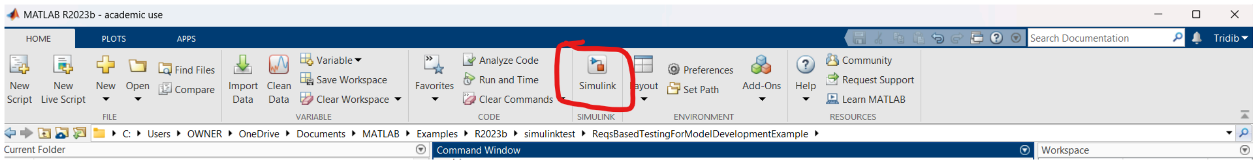
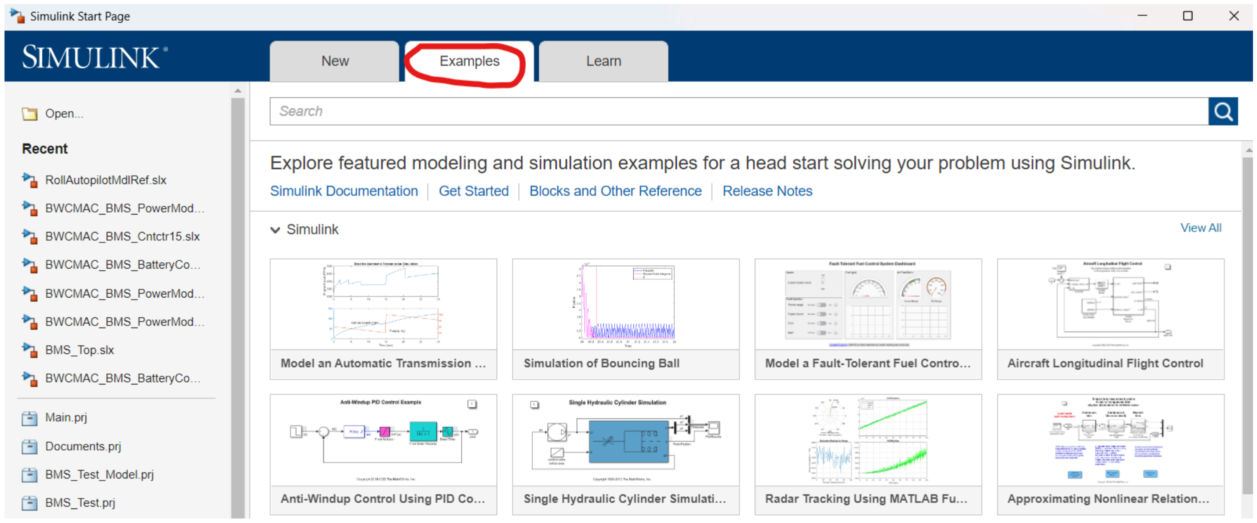
Recommended Example to follow:

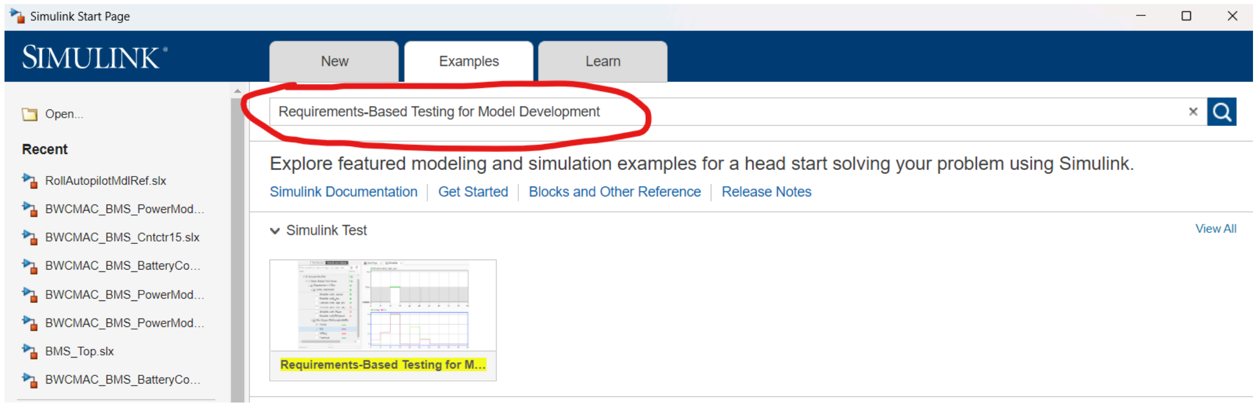
- Open Simulink by writing *Simulink* in the MATLAB command window and pressing Enter. Or press the *Simulink* icon.



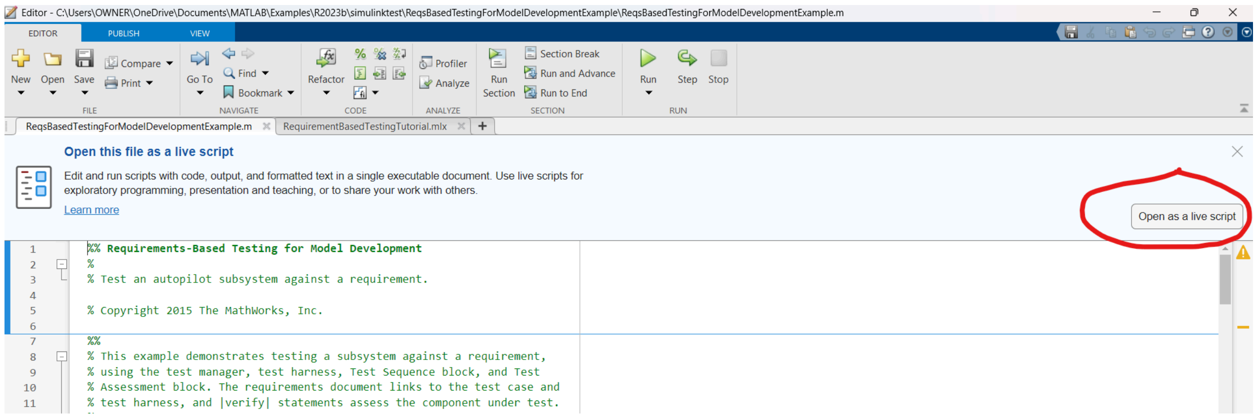
- Go to *Examples*.


Simulink Start Page
SIMULINK
New
Examples
Learn
Search
Open...
Recent
RollAutopilotMdlRef.slx
BWCMAC_BMS_PowerMod...
BWCMAC_BMS_Cntctr15.slx
BWCMAC_BMS_BatteryCo...
BWCMAC_BMS_PowerMod...
BWCMAC_BMS_PowerMod...
BMS_Top.slx
BWCMAC_BMS_BatteryCo...
Main.prj
Documents.prj
BMS_Test_Model.prj
BMS_Test.prj
Explore featured modeling and simulation examples for a head start solving your problem using Simulink.
Simulink Documentation
Get Started
Blocks and Other Reference
Release Notes
Simulink
View All
Model an Automatic Transmission ...
Simulation of Bouncing Ball
Model a Fault-Tolerant Fuel Contro...
Aircraft Longitudinal Flight Control
Anti-Windup Control Using PID Co...
Single Hydraulic Cylinder Simulati...
Radar Tracking Using MATLAB Fu...
Approximating Nonlinear Relation...

- Search for '*Requirements-Based Testing for Model Development*' and open the example.



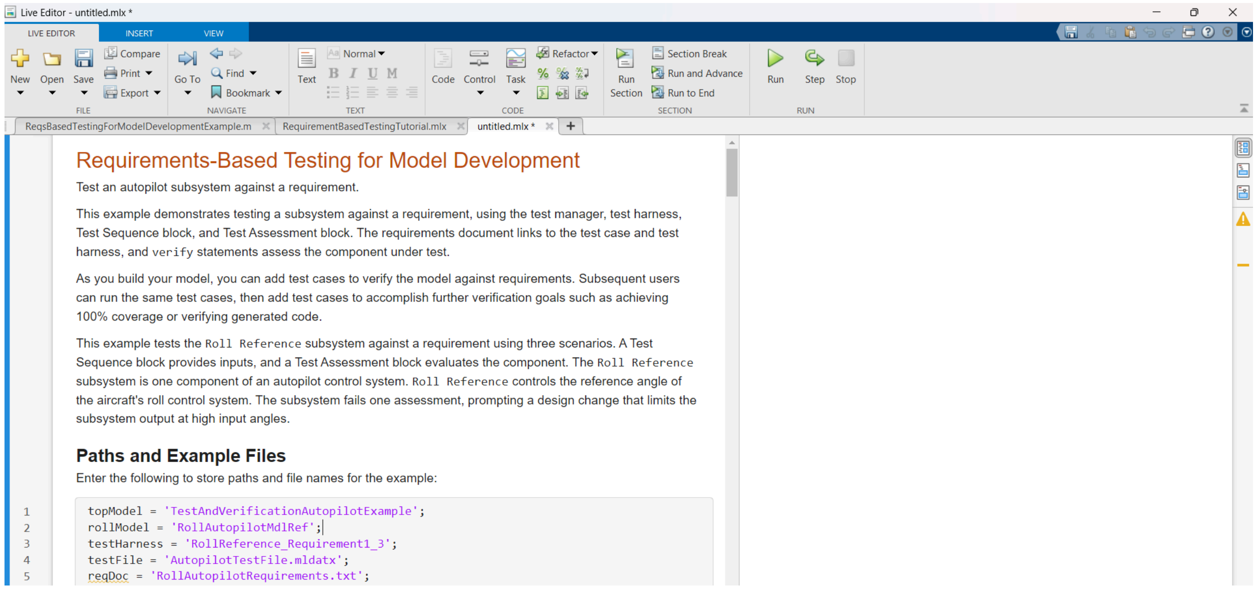
- You will likely land in an Editor. For convenience, open up the live editor by pressing *'Open as a live script'*.


Editor - C:\Users\OWNER\OneDrive\Documents\MATLAB\Examples\R2023b\simulinktest\ReqsBasedTestingForModelDevelopmentExample\ReqsBasedTestingForModelDevelopmentExample.m
EDITOR
PUBLISH
VIEW
New
Open
Save
Compare
Print
Go To
Find
Bookmark
Refactor
Profiler
Analyze
Run Section
Section Break
Run and Advance
Run to End
Run
Step
Stop
FILE
NAVIGATE
CODE
ANALYZE
SECTION
RUN
ReqsBasedTestingForModelDevelopmentExample.m
RequirementBasedTestingTutorial.mlx
Open this file as a live script
Learn more
Open as a live script

Edit and run scripts with code, output, and formatted text in a single executable document. Use live scripts for exploratory programming, presentation and teaching, or to share your work with others.

```
%% Requirements-Based Testing for Model Development
%
% Test an autopilot subsystem against a requirement.

% Copyright 2015 The MathWorks, Inc.

%%
% This example demonstrates testing a subsystem against a requirement,
% using the test manager, test harness, Test Sequence block, and Test
% Assessment block. The requirements document links to the test case and
% test harness, and |verify| statements assess the component under test.
```

- A Live Editor will start. First, save it as a .mlx file by giving it any convenient name you want using Ctrl+S on a Windows machine.



Requirements-Based Testing for Model Development

Test an autopilot subsystem against a requirement.

This example demonstrates testing a subsystem against a requirement, using the test manager, test harness, Test Sequence block, and Test Assessment block. The requirements document links to the test case and test harness, and `verify` statements assess the component under test.

As you build your model, you can add test cases to verify the model against requirements. Subsequent users can run the same test cases, then add test cases to accomplish further verification goals such as achieving 100% coverage or verifying generated code.

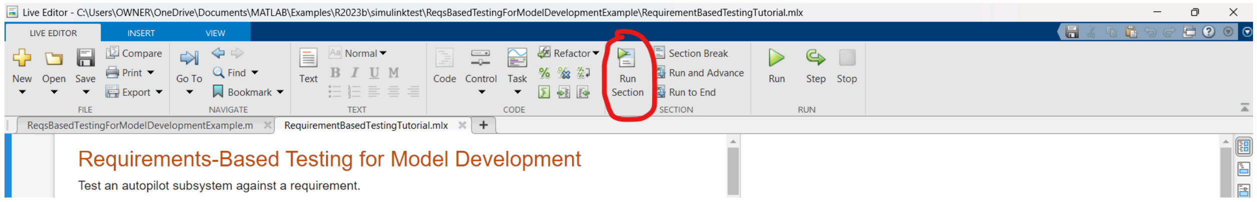
This example tests the `Roll Reference` subsystem against a requirement using three scenarios. A Test Sequence block provides inputs, and a Test Assessment block evaluates the component. The `Roll Reference` subsystem is one component of an autopilot control system. `Roll Reference` controls the reference angle of the aircraft's roll control system. The subsystem fails one assessment, prompting a design change that limits the subsystem output at high input angles.

Paths and Example Files

Enter the following to store paths and file names for the example:

```
topModel = 'TestAndVerificationAutopilotExample';
rollModel = 'RollAutopilotMdlRef';
testHarness = 'RollReference_Requirement1_3';
testFile = 'AutopilotTestFile.mldatx';
reqDoc = 'RollAutopilotRequirements.txt';
```

- Each Gray area is a code section. To run any section, go to that section by clicking anywhere inside the gray area. Then, click *Run Section* or press Ctrl+Enter from the keyboard.


Live Editor - C:\Users\OWNER\OneDrive\Documents\MATLAB\Examples\R2023b\simulinktest\ReqsBasedTestingForModelDevelopmentExample\RequirementBasedTestingTutorial.mlx
LIVE EDITOR
INSERT
VIEW
New
Open
Save
Compare
Print
Export
Go To
Find
Bookmark
Text
Normal
Code
Control
Task
Refactor
Run Section
Section Break
Run and Advance
Run to End
Run
Step
Stop
FILE
NAVIGATE
TEXT
CODE
SECTION
RUN
ReqsBasedTestingForModelDevelopmentExample.m
RequirementBasedTestingTutorial.mlx

Requirements-Based Testing for Model Development

Test an autopilot subsystem against a requirement.

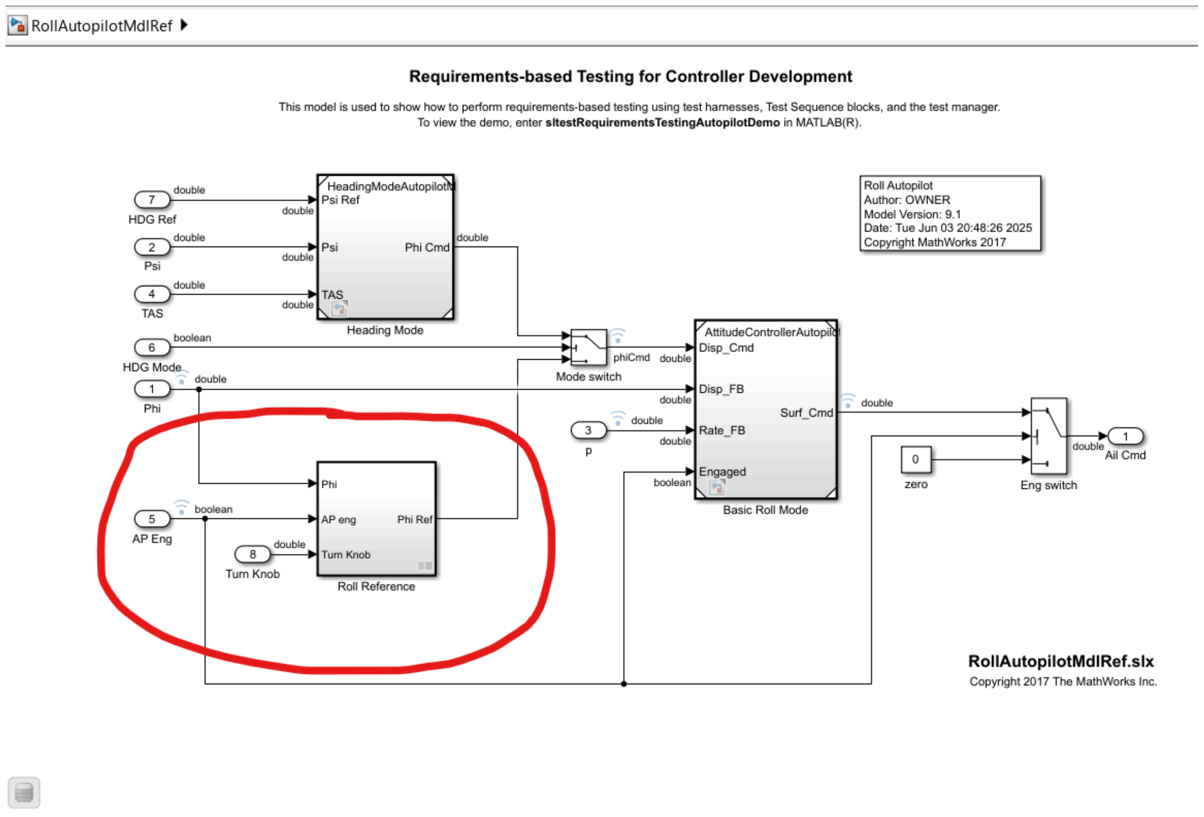
- Next steps are to follow along with the live script. It shows the entire workflow of Simulink Test, i.e., first, creating a Test harness for the subsystem you want to test, adding a *Test*

Sequence block and a *Test Assessment* block and adding steps to each of them in the Test Harness, creating a new test file from the Test Manager and adding Requirements, model and test harness, running the test to see if it passes or not, finally generating a test report.

To get a general idea of the workflow, please follow along with the live script instructions and run each section to open up the example test harness, test file, etc. To learn about starting from how to create a test harness to how to generate a test file, please follow along with the rest of this document.

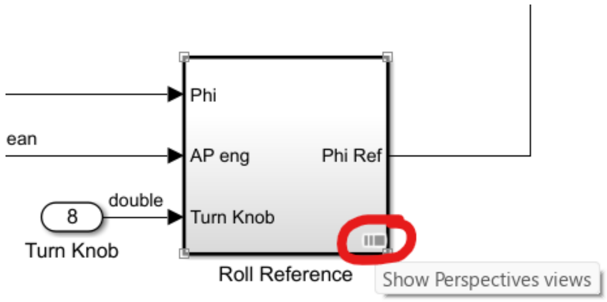
Test Harness

- In the Example, we were supposed to test the *Roll Reference* subsystem.

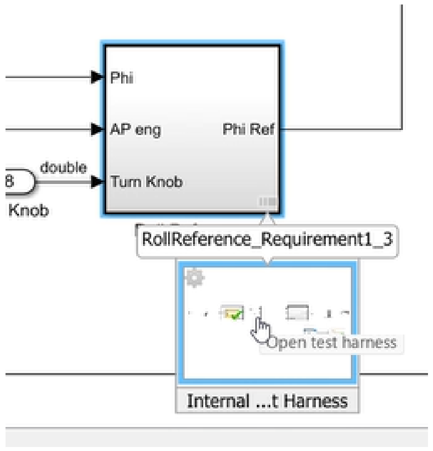

RollAutopilotMdlRef
Requirements-based Testing for Controller Development
This model is used to show how to perform requirements-based testing using test harnesses, Test Sequence blocks, and the test manager.
To view the demo, enter sltestRequirementsTestingAutopilotDemo in MATLAB(R).
HDG Ref
Psi
TAS
HDG Mode
Phi
AP Eng
Turn Knob
Heading Mode
Psi Ref
Phi Cmd
Mode switch
phiCmd
Disp_Cmd
Disp_FB
Rate_FB
Engaged
Surf_Cmd
Basic Roll Mode
p
Roll Reference
Phi
AP eng
Turn Knob
Phi Ref
zero
Eng switch
Ail Cmd
Roll Autopilot
Author: OWNER
Model Version: 9.1
Date: Tue Jun 03 20:48:26 2025
Copyright MathWorks 2017
RollAutopilotMdlRef.slx
Copyright 2017 The MathWorks Inc.

- Firstly, we need to open the Test Harness.
- A test harness is a model that isolates the component under test, with inputs, outputs, and verification blocks configured for testing scenarios.
- You can create a test harness for a model component or for a full model.
- A test harness gives you a separate testing environment for a model or a model component, which is useful for unit testing.

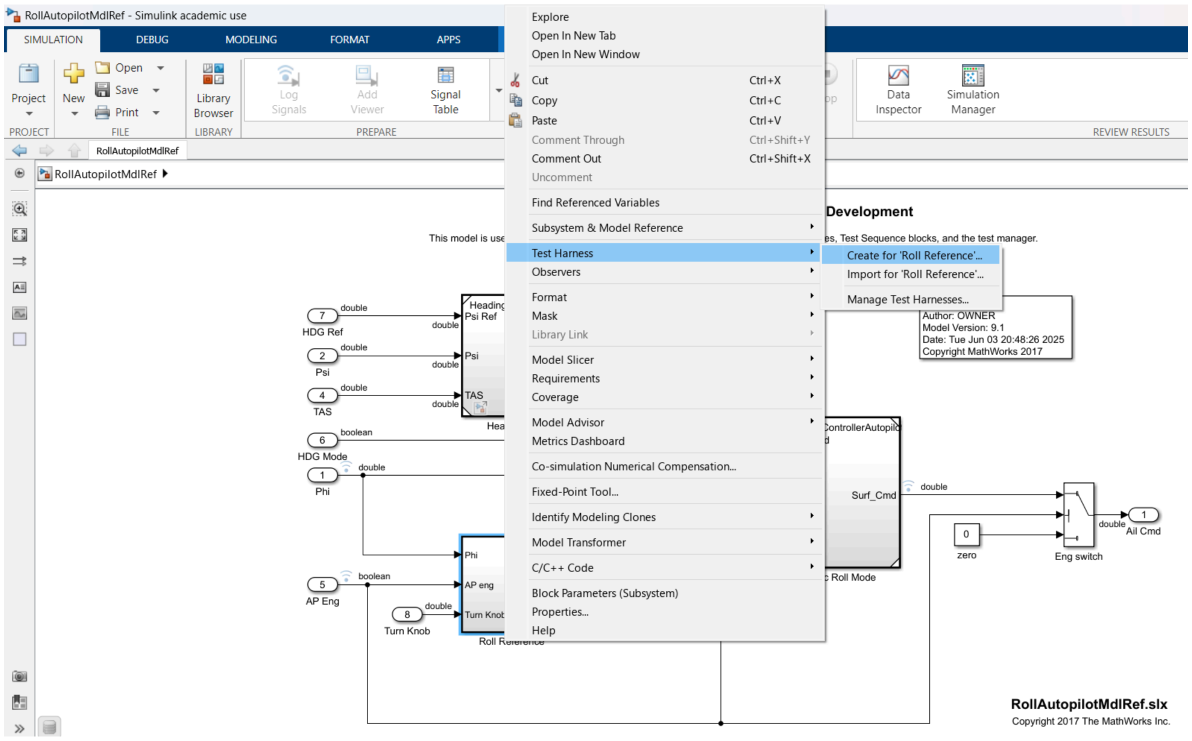
- To open the already created test harness for the Example, press *Show Perspective Views* icon.



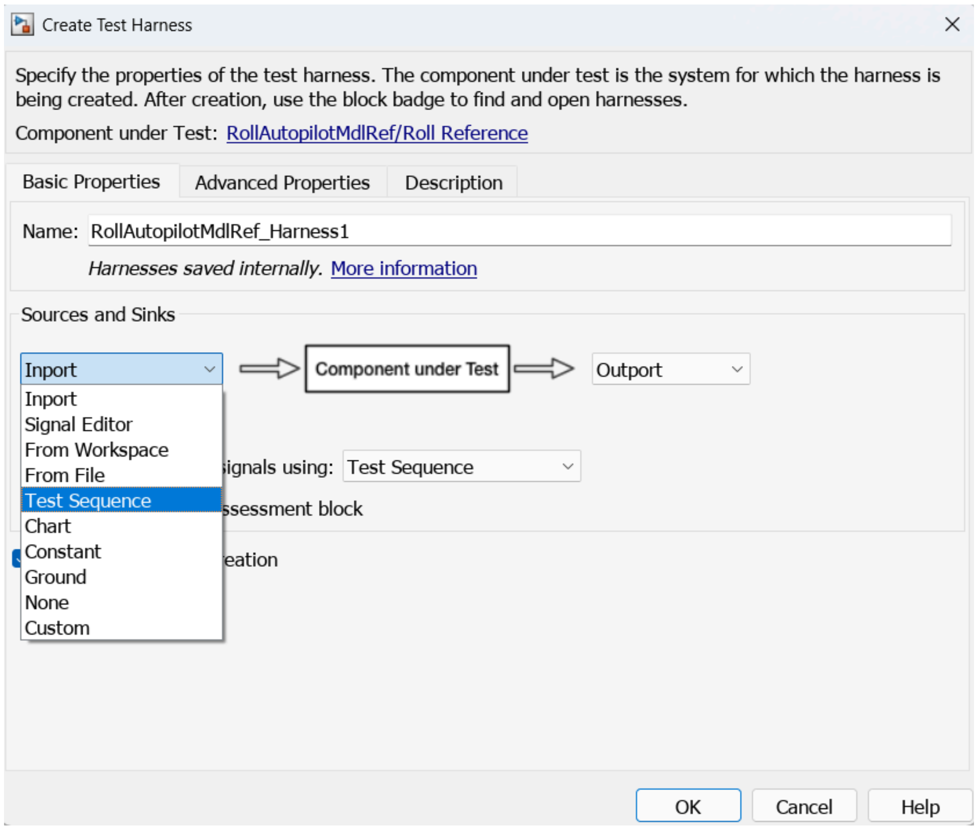
Open the test harness by clicking on it.



- To create a new test harness for any subsystem, first, right-click on the subsystem. Select *Test Harness* and select *Create for 'Subsystem_name'*.



- In the pop-up window, select *Test Sequence* instead of *Inport* from the drop-down menu of *Sources and Sinks.*



Create Test Harness

Specify the properties of the test harness. The component under test is the system for which the harness is being created. After creation, use the block badge to find and open harnesses.

Component under Test: RollAutopilotMdlRef/Roll Reference

Basic Properties | Advanced Properties | Description

Name: RollAutopilotMdlRef_Harness1

Harnesses saved internally. More information

Sources and Sinks

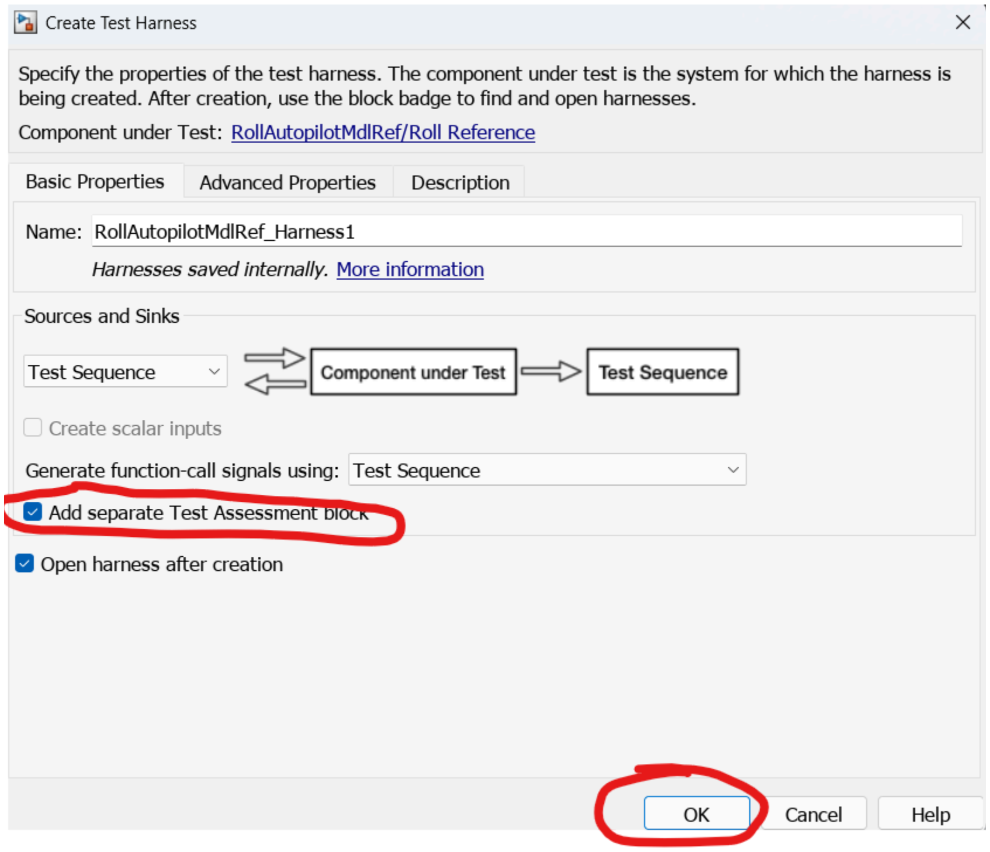
Inport → Component under Test → Outport

ignals using: Test Sequence

ssessment block

eation

- Check off *Add separate Test Assessment Block* and press OK.



Create Test Harness

Specify the properties of the test harness. The component under test is the system for which the harness is being created. After creation, use the block badge to find and open harnesses.

Component under Test: RollAutopilotMdlRef/Roll Reference

Basic Properties | Advanced Properties | Description

Name: RollAutopilotMdlRef_Harness1

Harnesses saved internally. More information

Sources and Sinks

Test Sequence ⇄ Component under Test ⇒ Test Sequence

[ ] Create scalar inputs

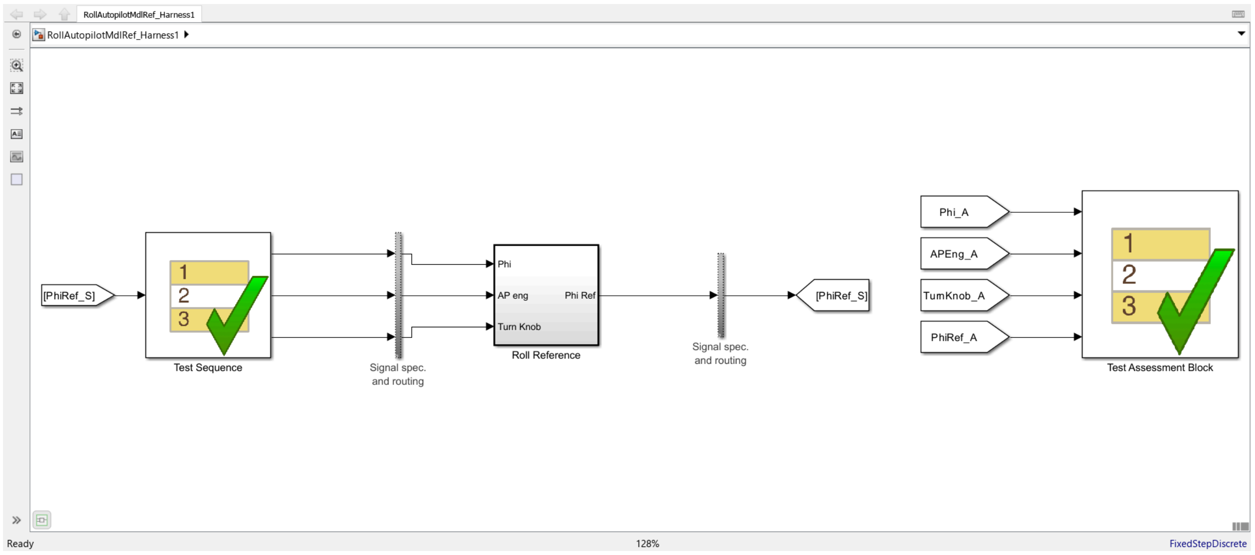
Generate function-call signals using: Test Sequence

[✓] Add separate Test Assessment block

[✓] Open harness after creation

OK | Cancel | Help

- It should create a Test Harness with a *Test Sequence* block on the left-hand side and a *Test Assessment* block on the right-hand side.



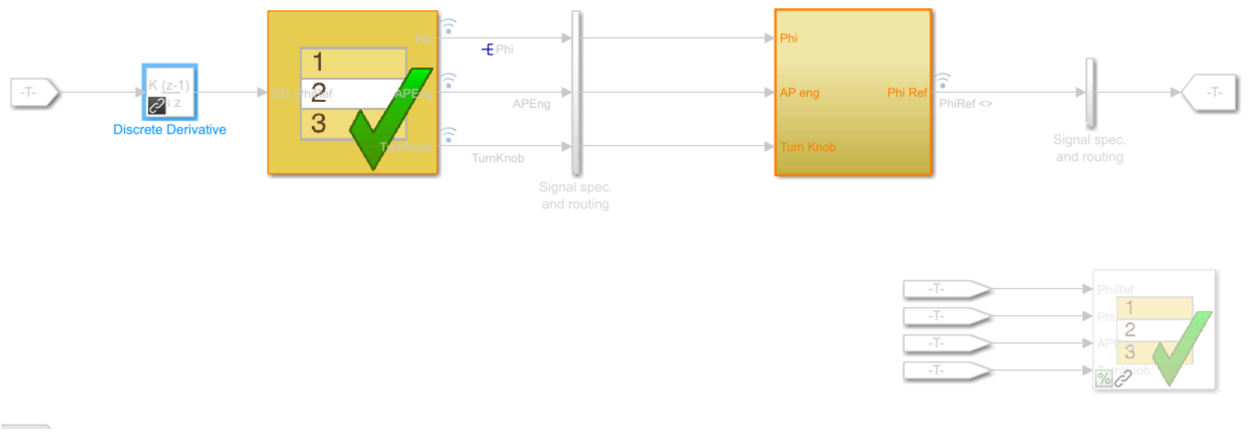
- Direct link to [Test Harness Breakdown]() to learn more about
    - Signal Conversion

- Function Calls
- Physical Signal Connections
- Bus Signals
- String Signals
- Non-Graphical Connections
- Export Function Models
- Execution Semantics
- Sample Time Specification

- Direct link (please save the Test Manager related documentation to learn after completing the Test Assessment related documentation) to Create and Simulate a Test Harness to learn more about
  - How to create a Test Harness and simulate a Test Harness
- Direct link to Author a Test Sequence and Test Assessment to learn more about
  - How to create a test sequence
  - How to create a test assessment
  - How to view the signal data in the *Scope* block
  - How to view the results of the *verify* statements in the *Simulation Data Inspector*.

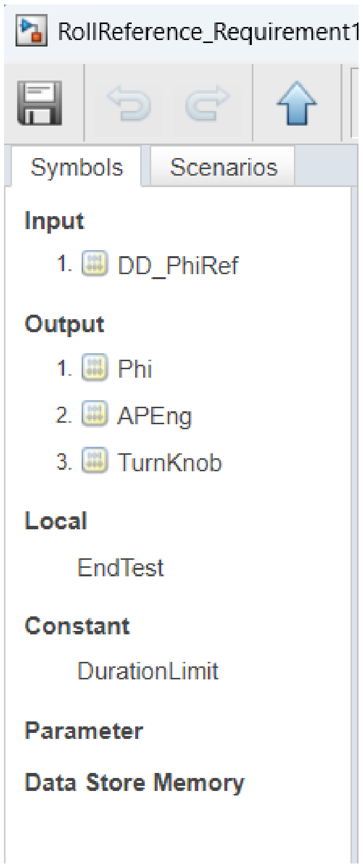
Test Sequence Block

- Open the Test Sequence Editor for the Test Sequence block from the Example (The Test Harness name shown in the Example is *RollReference_Requirement1_3*). Open it by double-clicking the Test Sequence block.



- Subsystem’s output is Test Sequence’s input.
- Subsystem’s input is Test Sequence’s output.
- A test sequence consists of test steps arranged in a hierarchy.
- You can use a test sequence to define test inputs and to define how a test will progress in response to the simulation.
- A test step contains actions that execute at the beginning of the step.
- A test step can contain transitions that define when the step stops executing and which test step executes next.
- Actions and transitions use MATLAB® as the action language.

- The *Input*, *Output*, *Local* and *Constant* to the Test Sequence block can be viewed and modified from the Symbols menu on the left-hand side of the Test Sequence Editor.



Input
1. DD_PhiRef

Output
1. Phi
2. APEng
3. TurnKnob

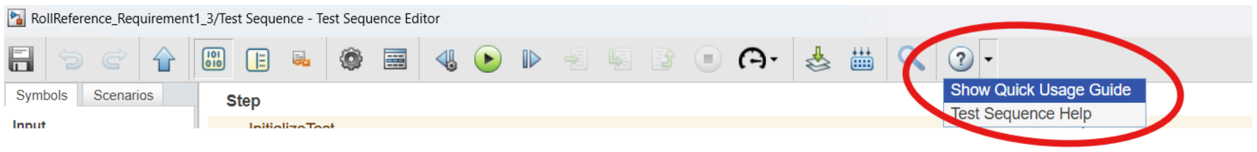
Local
EndTest

Constant
DurationLimit

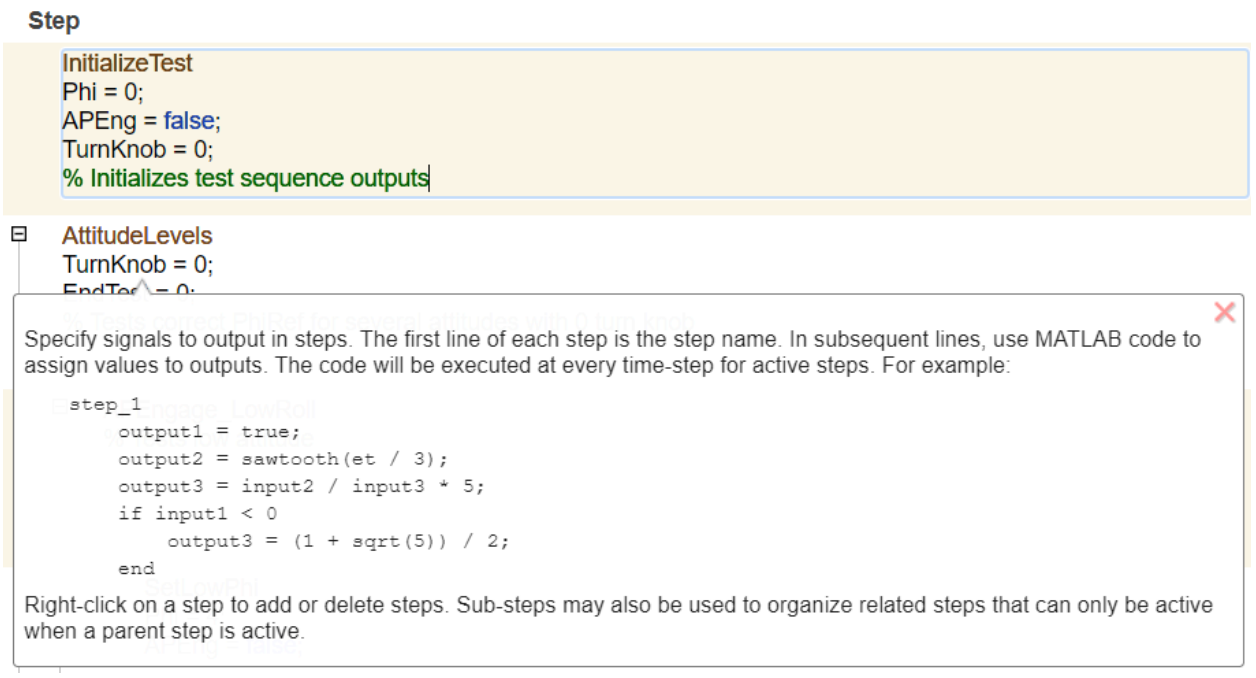
Parameter

Data Store Memory

- The details of how to define *Step* are below. The details of how to define a *Step* can be found by selecting *Show Quick Usage Guide*.



- The syntax and the interpretation of *Step* are shown in the screenshot below.



Step

InitializeTest
Phi = 0;
APEng = false;
TurnKnob = 0;
% Initializes test sequence outputs

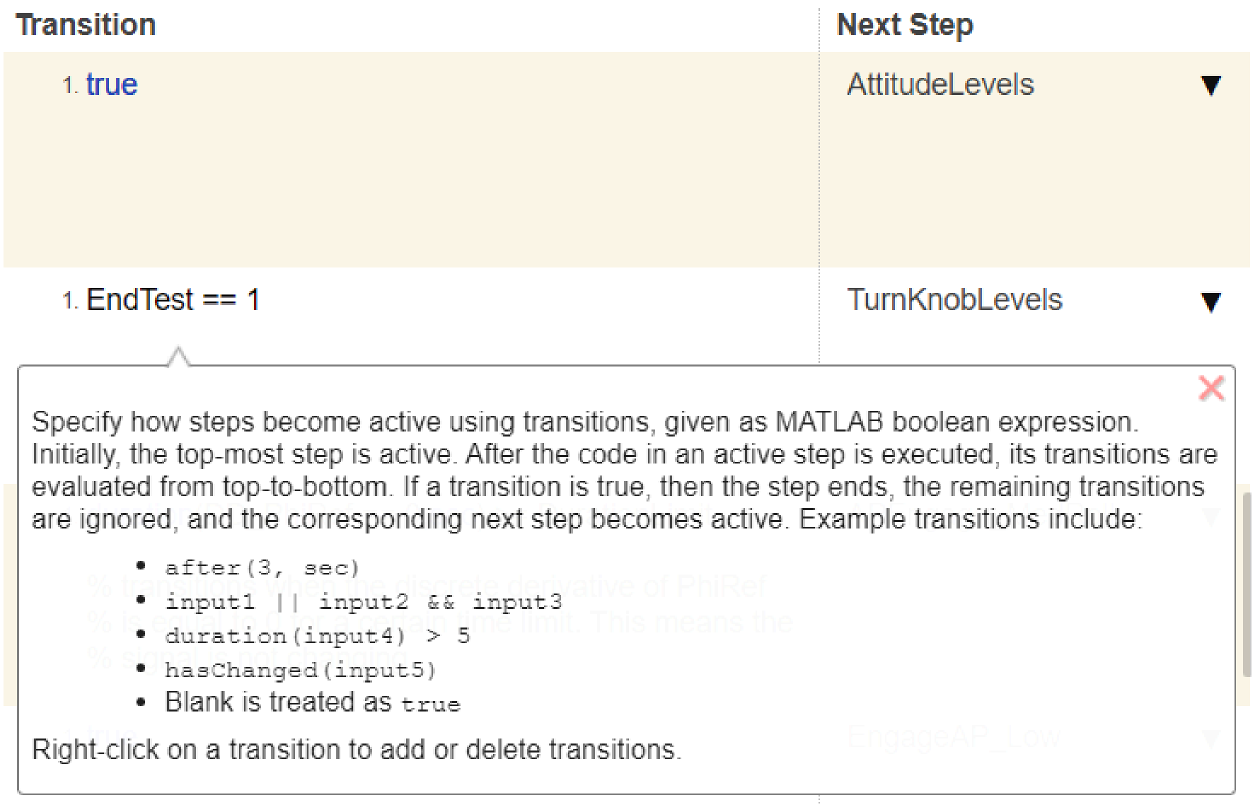
AttitudeLevels
TurnKnob = 0;

Specify signals to output in steps. The first line of each step is the step name. In subsequent lines, use MATLAB code to assign values to outputs. The code will be executed at every time-step for active steps. For example:

```
step_1
    output1 = true;
    output2 = sawtooth(et / 3);
    output3 = input2 / input3 * 5;
    if input1 < 0
        output3 = (1 + sqrt(5)) / 2;
    end
```

Right-click on a step to add or delete steps. Sub-steps may also be used to organize related steps that can only be active when a parent step is active.

- The syntax and the interpretation of *Transition* are shown in the screenshot below.



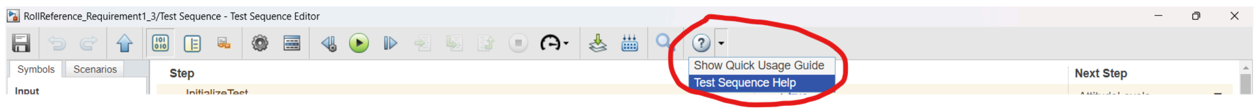
Transition	Next Step
1. true	AttitudeLevels
1. EndTest == 1	TurnKnobLevels

Specify how steps become active using transitions, given as MATLAB boolean expression. Initially, the top-most step is active. After the code in an active step is executed, its transitions are evaluated from top-to-bottom. If a transition is true, then the step ends, the remaining transitions are ignored, and the corresponding next step becomes active. Example transitions include:

- `after(3, sec)`
- `input1 || input2 && input3`
- `duration(input4) > 5`
- `hasChanged(input5)`
- Blank is treated as `true`

Right-click on a transition to add or delete transitions.

- To learn more about various syntax and interpretation, please go to the MathWorks Help Centre by pressing *Test Sequence Help*.



- Direct links to open each of the corresponding pages in a web browser. [Documentation](), [Examples](), [Functions](), [Blocks](), and [Apps]().

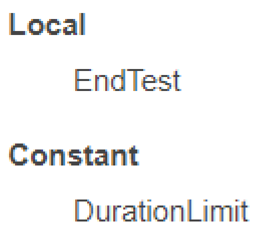
- Direct link to Test Sequence and Assessment Syntax to learn more about
  - Assessment Statements (verify, assert)
  - Temporal Operators (et, t, after, before, duration)
  - Transition Operators (hasChanged, hasChangedFrom, hasChangedTo)
  - Singal Generation Functions (sin, cos, square, exp, etc.)
  - Logical Operators (~, &&, ||, etc.)
  - Relational Operators (>, <, >=, <=, ==, ~=)

- Direct link to Test Sequence Editor to learn more about
  - How to define test sequences and scenarios
  - How to add and delete, copy and paste and reorder test steps
  - How to manage input, output, and data objects from *Symbols* Sidebar button
  - How to output and view active step data by going to *Model Explorer.*

- Direct link to Test Sequence Basics to learn more about
  - Test Sequence Hierarchy
  - Test Sequence Scenarios
  - Transition Types (Standard Transition using *after* keyword and When decomposition using *when* and *verify* keywords)

Inside the Test Sequence block of the *Example,* the
Local => EndTest
- Complexity: Inherited
- Initial Value: 0

Constant => DurationLimit
- Constant Value: 5

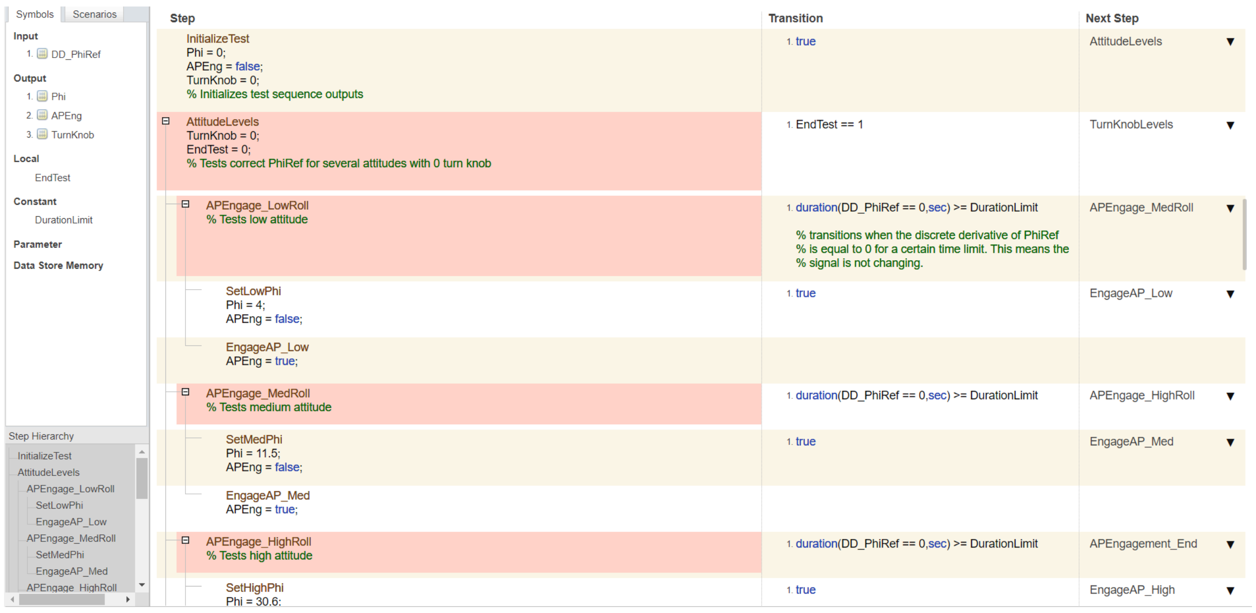


Local

EndTest

Constant

DurationLimit



Symbols | Scenarios

Input
1. DD_PhiRef

Output
1. Phi
2. APEng
3. TurnKnob

Local
EndTest

Constant
DurationLimit

Parameter

Data Store Memory

Step Hierarchy
- InitializeTest
- AttitudeLevels
  - APEngage_LowRoll
    - SetLowPhi
    - EngageAP_Low
  - APEngage_MedRoll
    - SetMedPhi
    - EngageAP_Med
  - APEngage_HighRoll

Step	Transition	Next Step

InitializeTest
Phi = 0;
APEng = false;
TurnKnob = 0;
% Initializes test sequence outputs
	1. true	AttitudeLevels

AttitudeLevels
TurnKnob = 0;
EndTest = 0;
% Tests correct PhiRef for several attitudes with 0 turn knob
	1. EndTest == 1	TurnKnobLevels

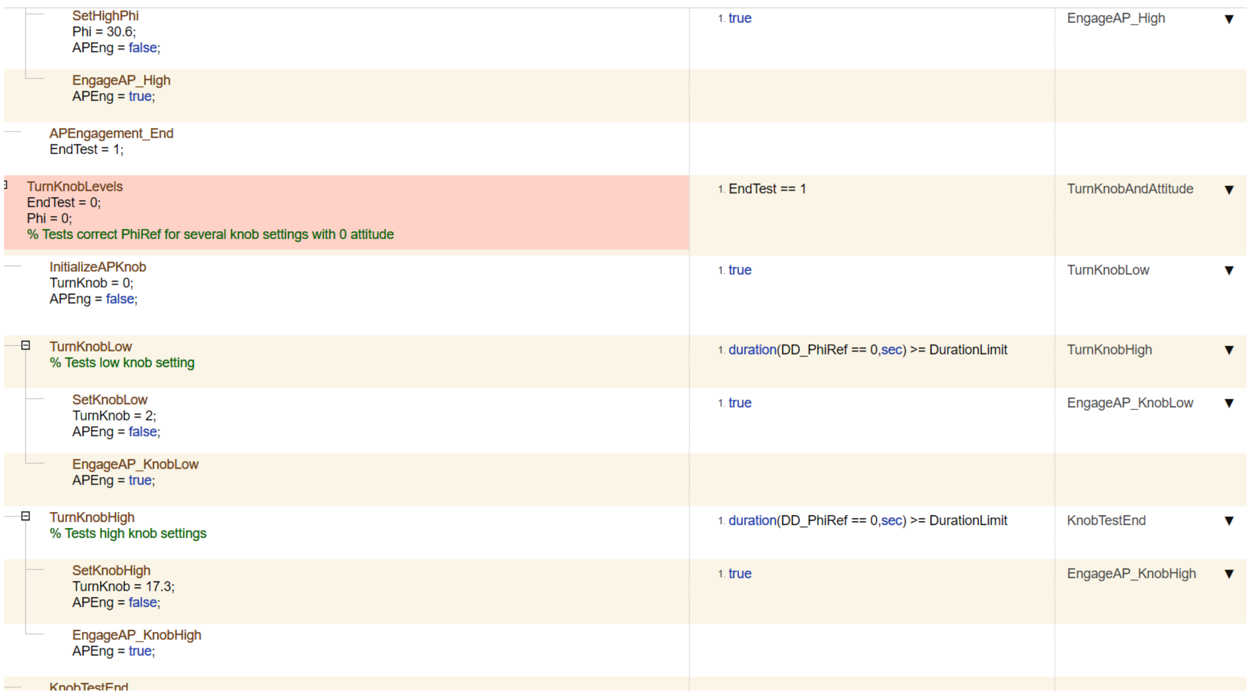
APEngage_LowRoll % Tests low attitude	1. duration(DD_PhiRef == 0,sec) >= DurationLimit % transitions when the discrete derivative of PhiRef % is equal to 0 for a certain time limit. This means the % signal is not changing.	APEngage_MedRoll

SetLowPhi
Phi = 4;
APEng = false;
	1. true	EngageAP_Low

EngageAP_Low APEng = true;		
APEngage_MedRoll % Tests medium attitude	1. duration(DD_PhiRef == 0,sec) >= DurationLimit	APEngage_HighRoll

SetMedPhi
Phi = 11.5;
APEng = false;
	1. true	EngageAP_Med

EngageAP_Med APEng = true;		
APEngage_HighRoll % Tests high attitude	1. duration(DD_PhiRef == 0,sec) >= DurationLimit	APEngagement_End
SetHighPhi Phi = 30.6;	1. true	EngageAP_High



Step	Transition	Next Step

SetHighPhi
Phi = 30.6;
APEng = false;
	1. true	EngageAP_High

EngageAP_High APEng = true;		
APEngagement_End EndTest = 1;		

TurnKnobLevels
EndTest = 0;
Phi = 0;
% Tests correct PhiRef for several knob settings with 0 attitude
	1. EndTest == 1	TurnKnobAndAttitude

InitializeAPKnob
TurnKnob = 0;
APEng = false;
	1. true	TurnKnobLow

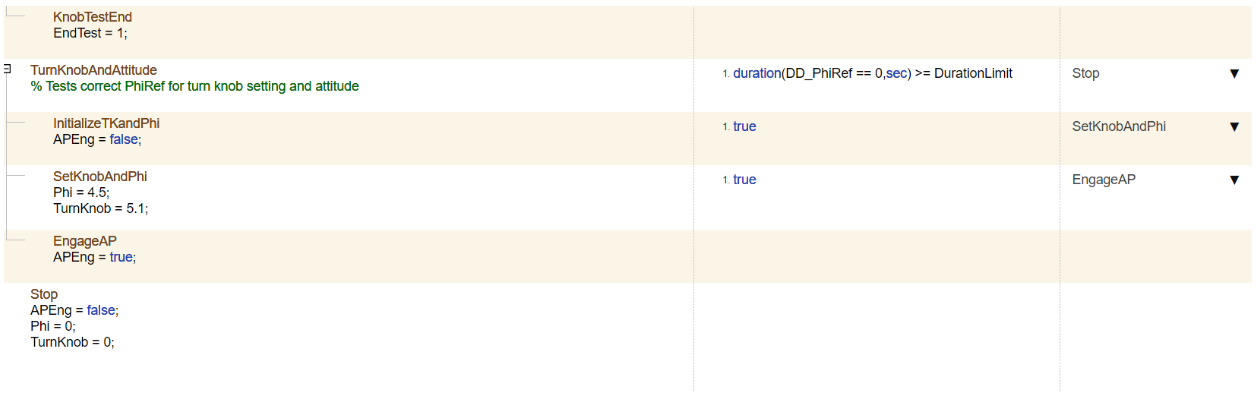
TurnKnobLow % Tests low knob setting	1. duration(DD_PhiRef == 0,sec) >= DurationLimit	TurnKnobHigh

SetKnobLow
TurnKnob = 2;
APEng = false;
	1. true	EngageAP_KnobLow

EngageAP_KnobLow APEng = true;		
TurnKnobHigh % Tests high knob settings	1. duration(DD_PhiRef == 0,sec) >= DurationLimit	KnobTestEnd

SetKnobHigh
TurnKnob = 17.3;
APEng = false;
	1. true	EngageAP_KnobHigh

EngageAP_KnobHigh APEng = true;		
KnobTestEnd		



Step	Transition	Next Step
KnobTestEnd EndTest = 1;		
TurnKnobAndAttitude % Tests correct PhiRef for turn knob setting and attitude	1. duration(DD_PhiRef == 0,sec) >= DurationLimit	Stop
InitializeTKandPhi APEng = false;	1. true	SetKnobAndPhi

SetKnobAndPhi
Phi = 4.5;
TurnKnob = 5.1;
	1. true	EngageAP

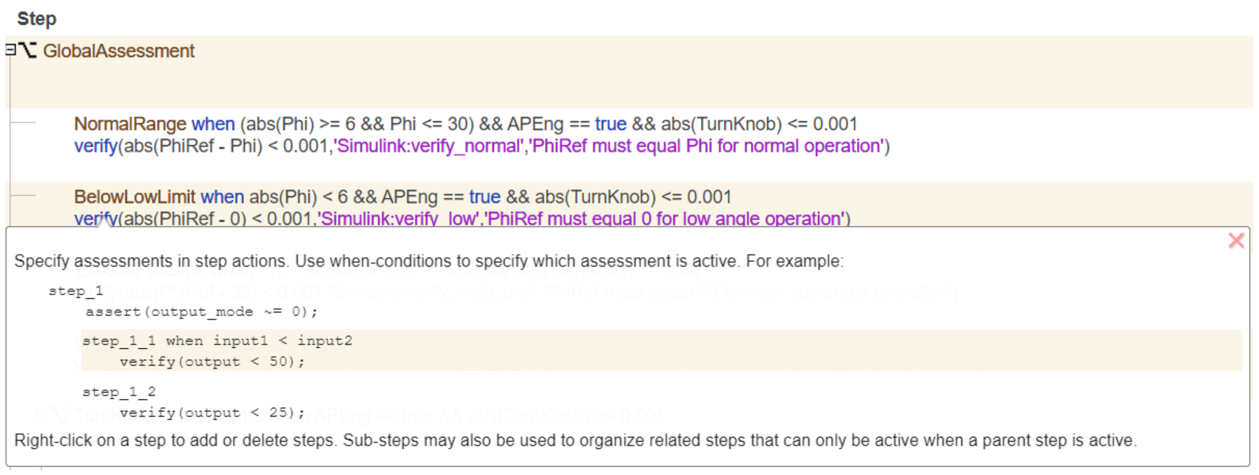
EngageAP APEng = true;		

Stop
APEng = false;
Phi = 0;
TurnKnob = 0;		

- **AttitudeLevels:** Keep TurnKnob = 0 (constant) and change AttitudeLevels (phi) (Low, Medium and High).
- **TurnKnobLevels:** Keep Phi = 0 (constant) and change TurnKnob (Low and High).
- **TurnKnobAndPhi:** Change both Phi and TurnKnob together.

Test Assessment Block

- Open the Test Sequence Editor for the Test Assessment block from the Example (The Test Harness name shown in the Example is *RollReference_Requirement1_3*). Open it by double-clicking the Test Assessment block.
- Test Assessment block takes all signals (both input and output) of the subsystem.
- It defines test assessments in a tabular series of steps.
- Like the Test Sequence block, the Test Assessment block uses MATLAB® as the action language.
- *verify* statements return *pass*, *fail*, or *untested* results for both the overall simulation and individual time steps.
- A *fail* at any time step results in an overall *fail*.
- If there are no failing results, a *pass* at any time step results in an overall *pass*.
- Otherwise, the overall result is *untested*.
- Results appear in the *Test Manager*.
- The syntax and the interpretation of *Step* are shown in the screenshot below.



```
Step
GlobalAssessment

    NormalRange when (abs(Phi) >= 6 && Phi <= 30) && APEng == true && abs(TurnKnob) <= 0.001
    verify(abs(PhiRef - Phi) < 0.001,'Simulink:verify_normal','PhiRef must equal Phi for normal operation')

    BelowLowLimit when abs(Phi) < 6 && APEng == true && abs(TurnKnob) <= 0.001
    verify(abs(PhiRef - 0) < 0.001,'Simulink:verify_low','PhiRef must equal 0 for low angle operation')
```

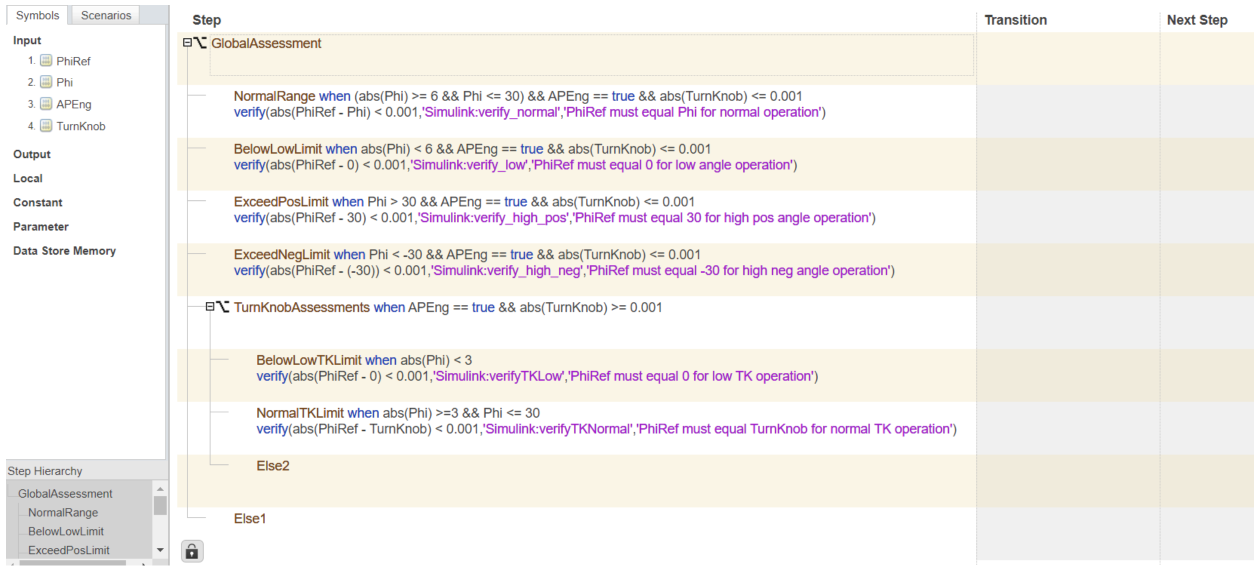
Specify assessments in step actions. Use when-conditions to specify which assessment is active. For example:

```
  step_1
     assert(output_mode ~= 0);

     step_1_1 when input1 < input2
        verify(output < 50);

     step_1_2
        verify(output < 25);
```

Right-click on a step to add or delete steps. Sub-steps may also be used to organize related steps that can only be active when a parent step is active.

- Syntax to write *verify* statements: verify(expression, identifier, errorMessage)
- Direct link to [Assess Model Simulation Using verify Statements]() to learn more about
    - How to activate *verify* statements with *Test Sequence Steps* by using *When* decomposition sequence.
    - How to activate *verify* statements with *Signal Conditions.*

Inside the Test Assessment block of the *Example,*

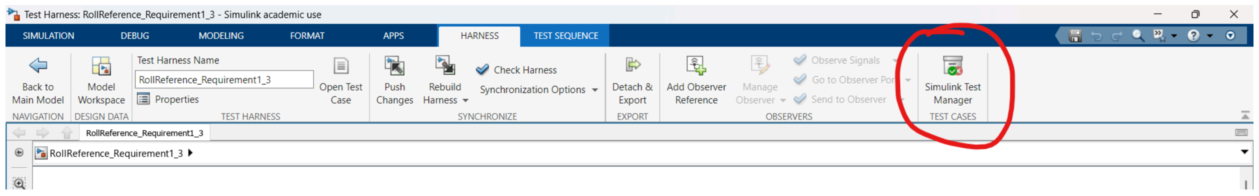


```
GlobalAssessment

    NormalRange when (abs(Phi) >= 6 && Phi <= 30) && APEng == true && abs(TurnKnob) <= 0.001
    verify(abs(PhiRef - Phi) < 0.001,'Simulink:verify_normal','PhiRef must equal Phi for normal operation')

    BelowLowLimit when abs(Phi) < 6 && APEng == true && abs(TurnKnob) <= 0.001
    verify(abs(PhiRef - 0) < 0.001,'Simulink:verify_low','PhiRef must equal 0 for low angle operation')

    ExceedPosLimit when Phi > 30 && APEng == true && abs(TurnKnob) <= 0.001
    verify(abs(PhiRef - 30) < 0.001,'Simulink:verify_high_pos','PhiRef must equal 30 for high pos angle operation')

    ExceedNegLimit when Phi < -30 && APEng == true && abs(TurnKnob) <= 0.001
    verify(abs(PhiRef - (-30)) < 0.001,'Simulink:verify_high_neg','PhiRef must equal -30 for high neg angle operation')

    TurnKnobAssessments when APEng == true && abs(TurnKnob) >= 0.001

        BelowLowTKLimit when abs(Phi) < 3
        verify(abs(PhiRef - 0) < 0.001,'Simulink:verifyTKLow','PhiRef must equal 0 for low TK operation')

        NormalTKLimit when abs(Phi) >=3 && Phi <= 30
        verify(abs(PhiRef - TurnKnob) < 0.001,'Simulink:verifyTKNormal','PhiRef must equal TurnKnob for normal TK operation')

        Else2

    Else1
```

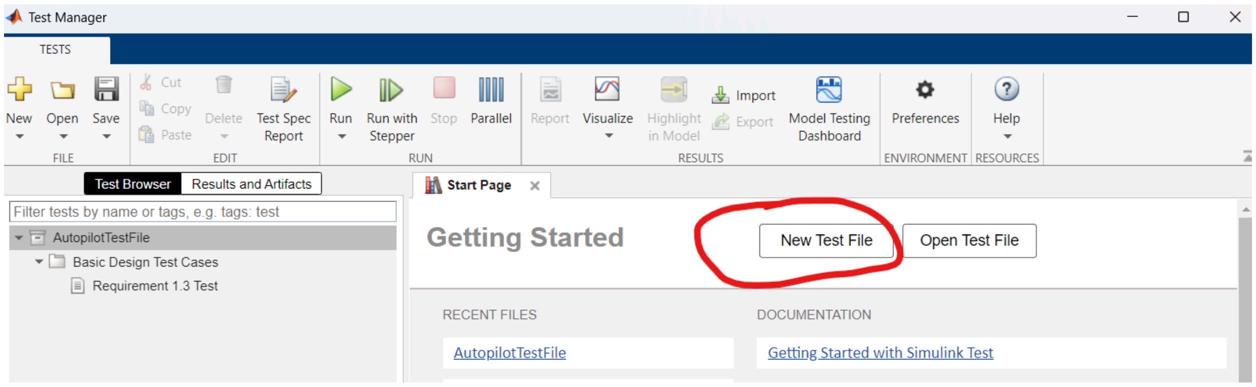
- **GlobalAssessment:** Change Phi to imitate all possible values
- **TurnKnobAssessments:** Test TurnKnob by making it below low and normal limit (req 1.3.1.3)

Test Manager

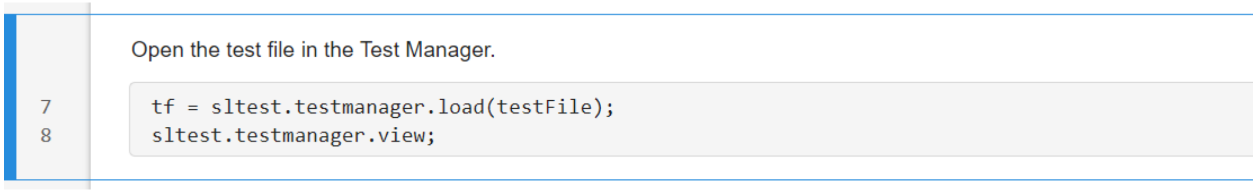
- Write *sltest.testmanager.view;* to the MATLAB command window to open the Test Manager App.
- Alternatively, click on the *Test Manager* from the Test Harness top menu bar under the *Harness* tab.



- To create a new Test file (any file with an extension .mldatx is a Test file), click *New Test FIle*.



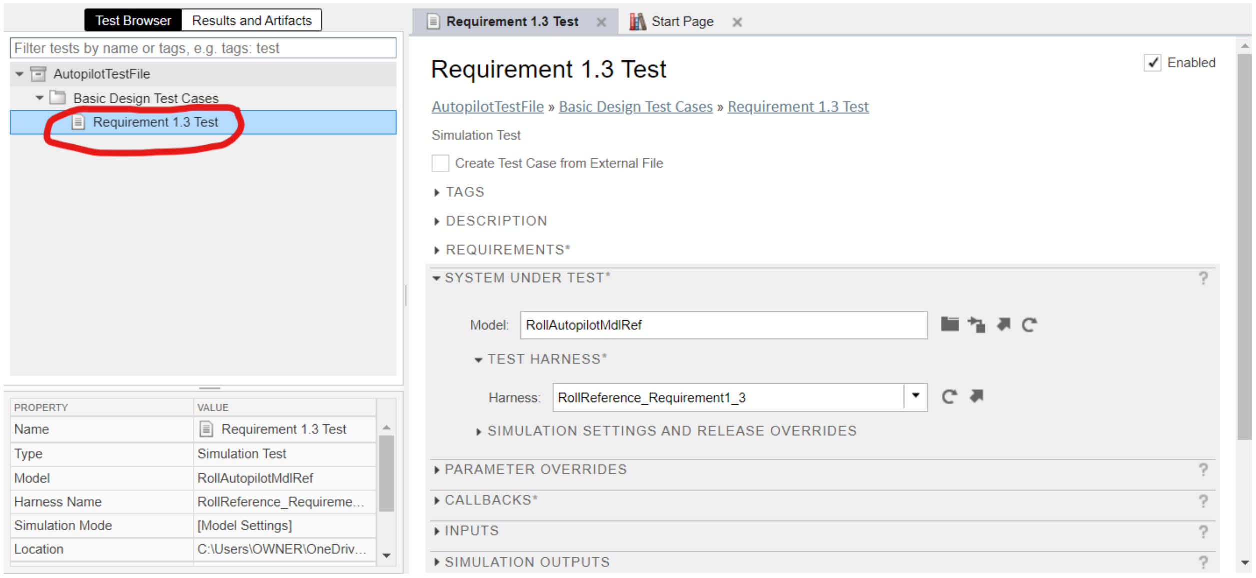
- Let's work through the test file provided in the Example. Open the test file by running the corresponding section of the live script.



Open the test file in the Test Manager.

```
tf = sltest.testmanager.load(testFile);
sltest.testmanager.view;
```

- From the Test Browser, click on Requirement 1.3 Test. Here, AutopilotTestFile is the test file name, Basic Design Test Cases is the Test suite that usually contains a group of test cases, and Requirement 1.3 Test is a single test case.

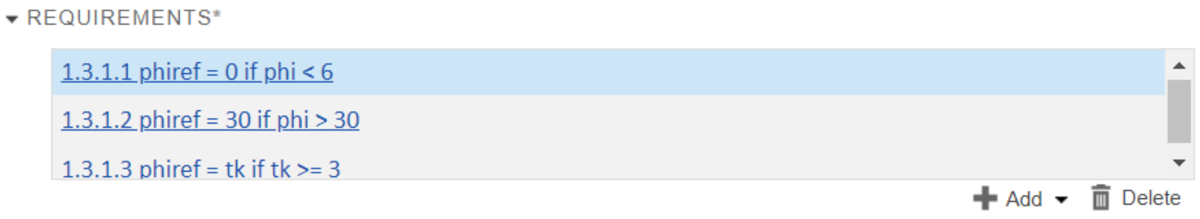


Model: RollAutopilotMdlRef

Harness: RollReference_Requirement1_3

PROPERTY	VALUE
Name	Requirement 1.3 Test
Type	Simulation Test
Model	RollAutopilotMdlRef
Harness Name	RollReference_Requireme...
Simulation Mode	[Model Settings]
Location	C:\Users\OWNER\OneDriv...

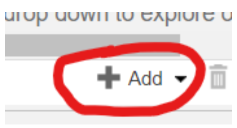
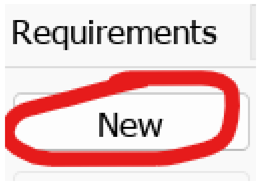
- First, go to *System Under Test* section in *Requirement 1.3 Test* window. The Example already has test requirements added 1.3.1.1, 1.3.1.2, and 1.3.1.3. To view them, simply click on one of the links as instructed in the live script.

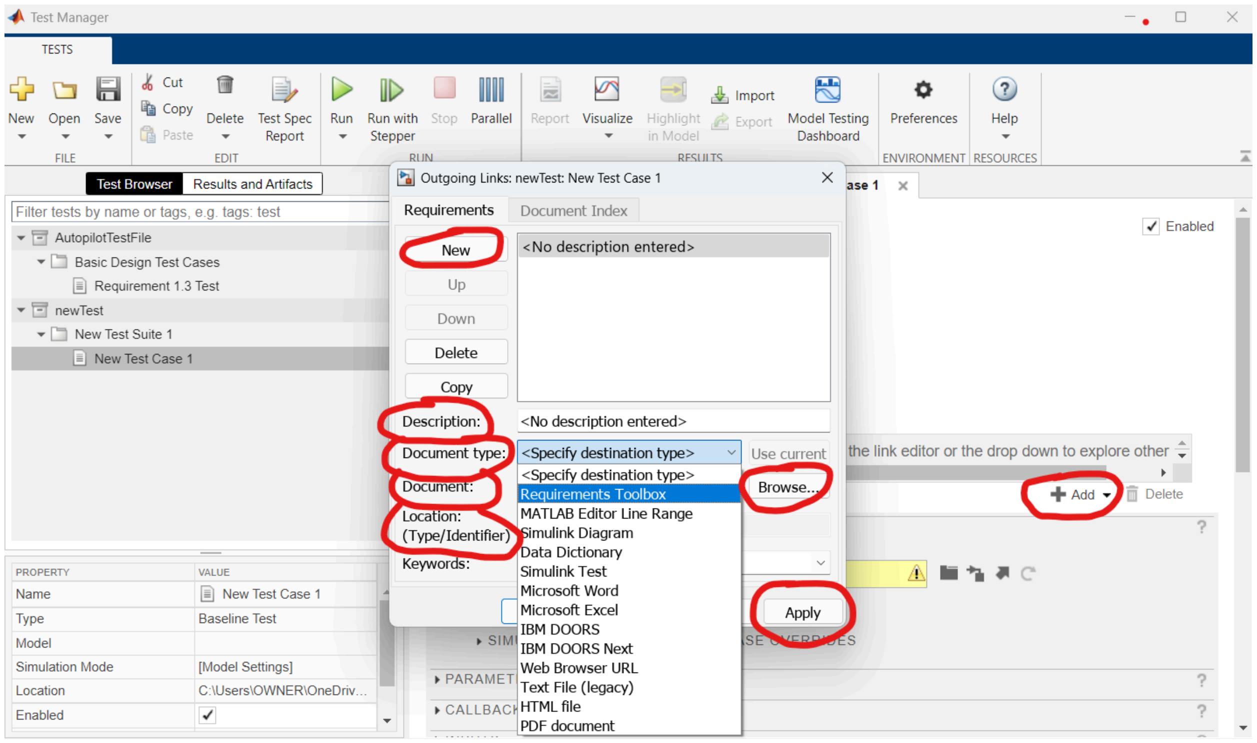

REQUIREMENTS*

1.3.1.1 phiref = 0 if phi < 6

1.3.1.2 phiref = 30 if phi > 30

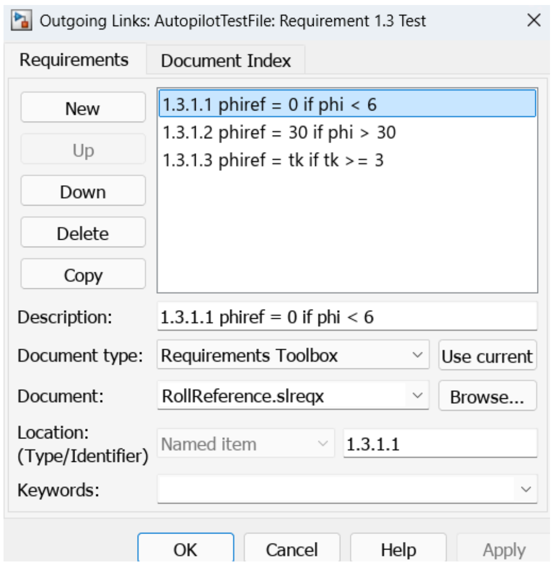
1.3.1.3 phiref = tk if tk >= 3

- To add a new requirement, first click on *Add* . Then, click on New in the popped-up window . Add a brief description of the requirement. In the *Document type,* select the correct file type. Get the particular requirements document, click *browse* from the *Document* field. This opens a Windows File Explorer pop-up window. Choose the correct requirements file. Add an *identifier* for the requirement as well from the *Location* field for easier distinction. Finally, press *Apply* to add it to the test case.



PROPERTY	VALUE
Name	New Test Case 1
Type	Baseline Test
Model	
Simulation Mode	[Model Settings]
Location	C:\Users\OWNER\OneDriv...
Enabled	✓

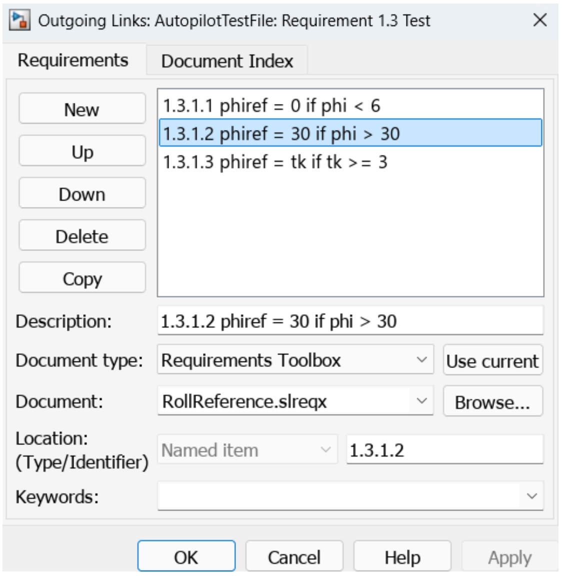
- The example has 3 requirements added. Here they are.


Outgoing Links: AutopilotTestFile: Requirement 1.3 Test

Requirements | Document Index

1.3.1.1 phiref = 0 if phi < 6
1.3.1.2 phiref = 30 if phi > 30
1.3.1.3 phiref = tk if tk >= 3

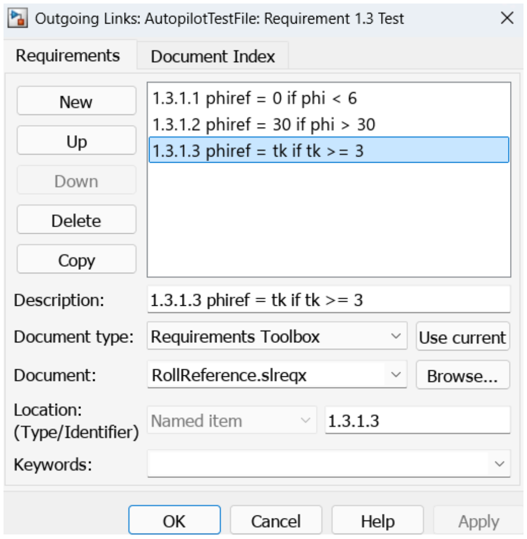
Description:	1.3.1.1 phiref = 0 if phi < 6
Document type:	Requirements Toolbox
Document:	RollReference.slreqx
Location: (Type/Identifier)	Named item 1.3.1.1
Keywords:	


Outgoing Links: AutopilotTestFile: Requirement 1.3 Test

Requirements | Document Index

1.3.1.1 phiref = 0 if phi < 6
1.3.1.2 phiref = 30 if phi > 30
1.3.1.3 phiref = tk if tk >= 3

Description:	1.3.1.2 phiref = 30 if phi > 30
Document type:	Requirements Toolbox
Document:	RollReference.slreqx
Location: (Type/Identifier)	Named item 1.3.1.2
Keywords:	

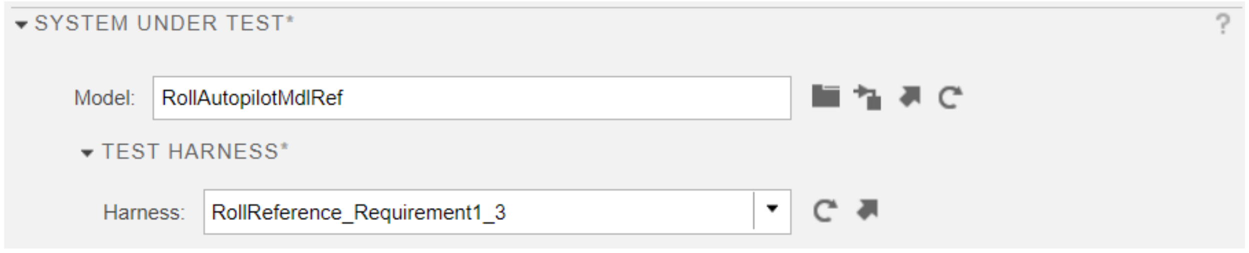

Outgoing Links: AutopilotTestFile: Requirement 1.3 Test

Requirements | Document Index

1.3.1.1 phiref = 0 if phi < 6
1.3.1.2 phiref = 30 if phi > 30
1.3.1.3 phiref = tk if tk >= 3

Description:	1.3.1.3 phiref = tk if tk >= 3
Document type:	Requirements Toolbox
Document:	RollReference.slreqx
Location: (Type/Identifier)	Named item 1.3.1.3
Keywords:	

- Next, go to *System Under Test* section in *Requirement 1.3 Test* window and add the corresponding model and test harness following the live script instructions.


SYSTEM UNDER TEST*

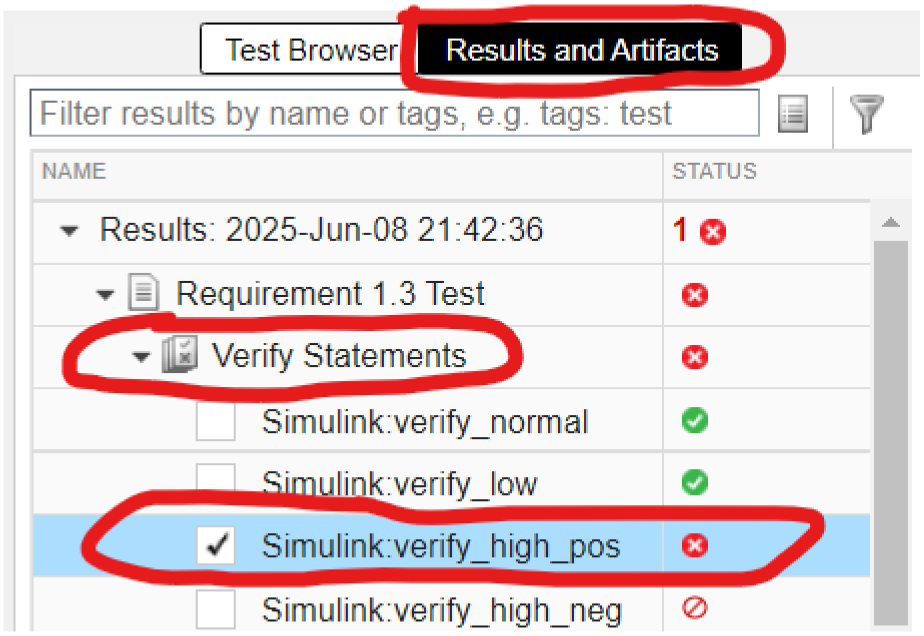
Model: RollAutopilotMdlRef

TEST HARNESS*

Harness: RollReference_Requirement1_3

- Now are the debugging steps. The Example model purposefully had an error to demonstrate how to debug. Due to the error, initially, the test case will fail. But after adding a *Saturation* block to the *Roll Reference* subsystem, the error is fixed, and the test case passes after running it again. To view a particular signal, go to *Results and*

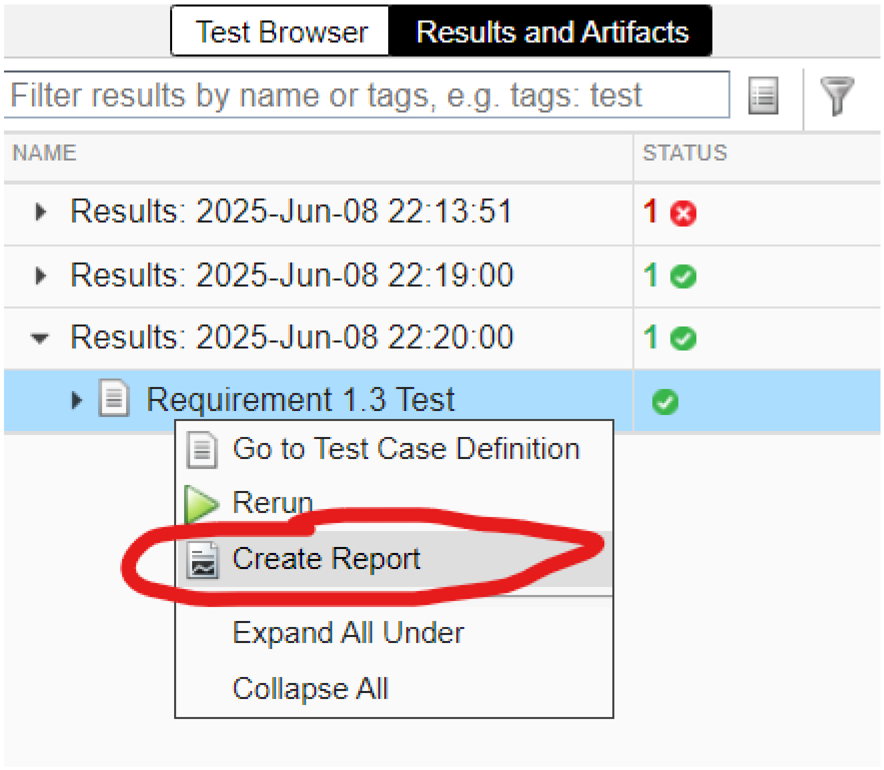
Artifacts and check off the particular signal (i.e., in the Example verify_high_pos is checked off to view its behaviour, since it is not showing expected behaviour). The signals can be found from the *Verify Statements* dropdown.



NAME	STATUS
Results: 2025-Jun-08 21:42:36	1 ✗
Requirement 1.3 Test	✗
Verify Statements	✗
☐ Simulink:verify_normal	✓
☐ Simulink:verify_low	✓
☑ Simulink:verify_high_pos	✗
☐ Simulink:verify_high_neg	⊘

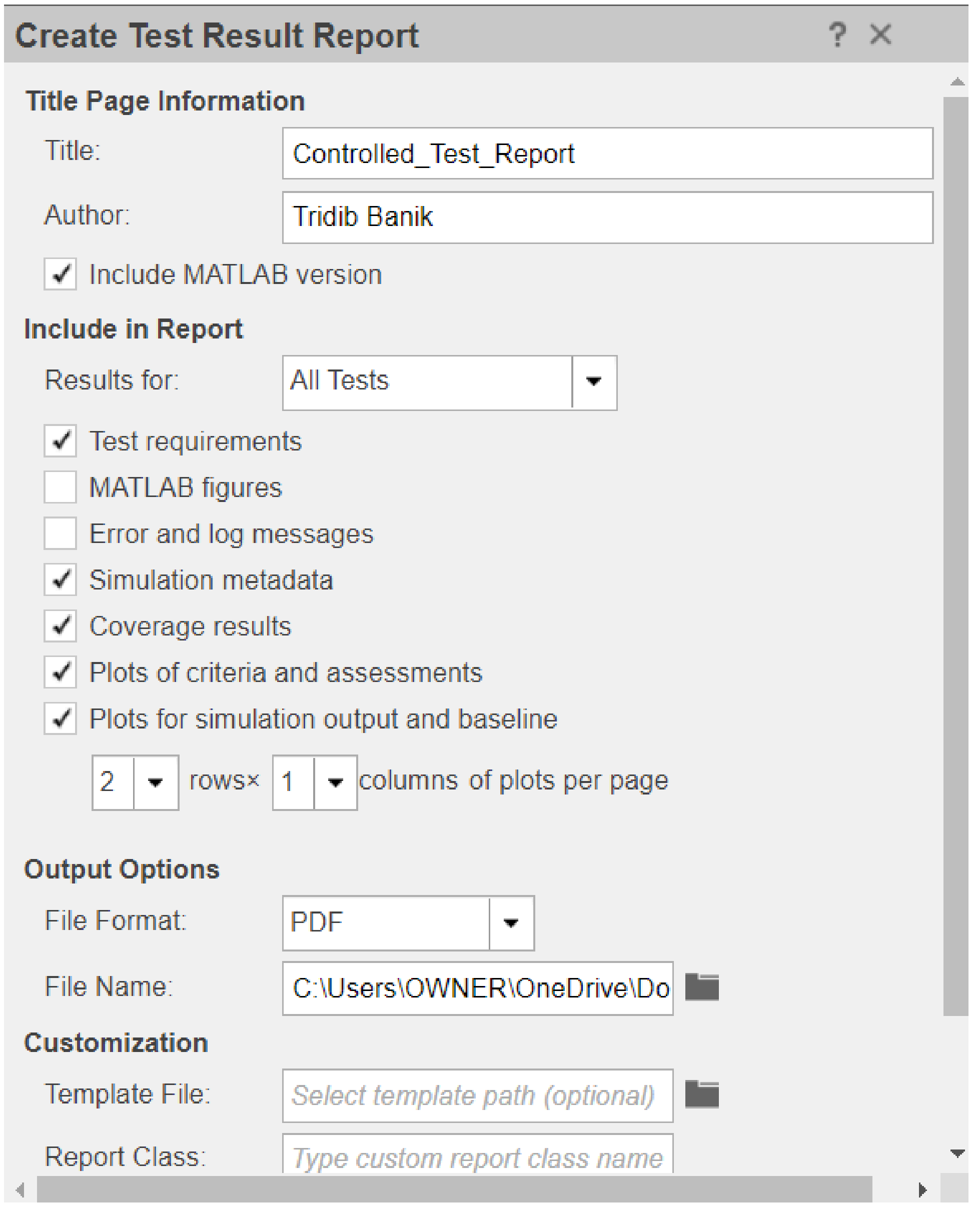
- First, close the test harness to fix the error. Follow the live script to add the *Saturation* block to the actual model (not the test harness model) and add proper upper and lower limits, and come back to *Test Manager* and run the test again. The test case will pass eventually.

- To generate the test report, right click on the passed test case and click *Create Report*



Test Browser
Results and Artifacts
Filter results by name or tags, e.g. tags: test
NAME
STATUS
Results: 2025-Jun-08 22:13:51
1
Results: 2025-Jun-08 22:19:00
1
Results: 2025-Jun-08 22:20:00
1
Requirement 1.3 Test
Go to Test Case Definition
Rerun
Create Report
Expand All Under
Collapse All

- In the popped-up window, choose title and author to show in the report. Check off what to include in the test report. Choose the appropriate File Format (pdf, docx, and zip are available). Finally, press *Create*.


Create Test Result Report

Title Page Information

Title:	Controlled_Test_Report
Author:	Tridib Banik

- [✓] Include MATLAB version

Include in Report

Results for:	All Tests

- [✓] Test requirements
- [ ] MATLAB figures
- [ ] Error and log messages
- [✓] Simulation metadata
- [✓] Coverage results
- [✓] Plots of criteria and assessments
- [✓] Plots for simulation output and baseline

2 rows× 1 columns of plots per page

Output Options

File Format:	PDF
File Name:	C:\Users\OWNER\OneDrive\Do

Customization

Template File:	*Select template path (optional)*
Report Class:	*Type custom report class name*

- A new window with the chosen file format will pop up. Download it for future reference.
- Direct link (please scroll down to the middle of the page to view Test Manager-related documentation) to Test Using Test Manager to learn more about
    - How to create a new test case in *Test Manager*
    - How to use external data with a test harness, and simulate from the Test Manager