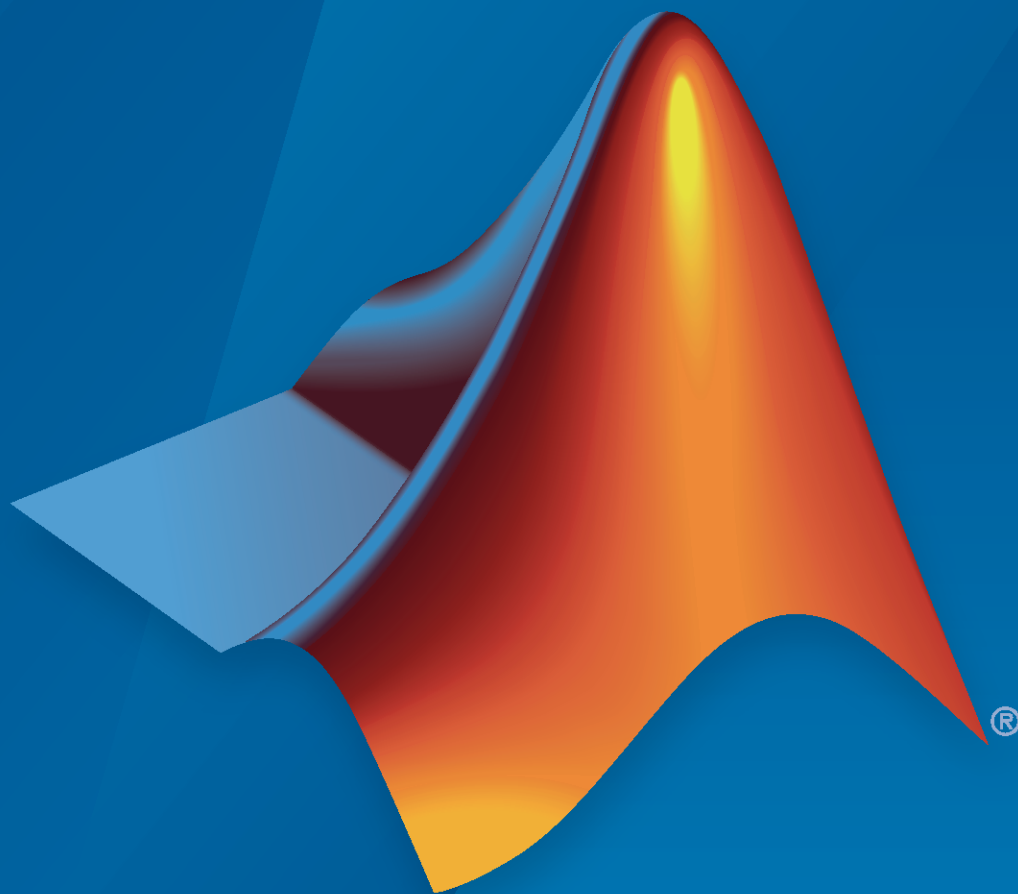


# UAV Toolbox

## User's Guide



# MATLAB® & SIMULINK®

R2020b





<b>1</b>	<b>UAV Toolbox Examples</b>
	<hr/>
	<b>Visualize and Playback MAVLink Flight Log</b> ..... 1-2
	<b>Flight Instrument Gauge Visualization for a Drone</b> ..... 1-5
	<b>Visualize Custom Flight Log</b> ..... 1-11
	<b>Tuning Waypoint Follower for Fixed-Wing UAV</b> ..... 1-25
	<b>Approximate High-Fidelity UAV model with UAV Guidance Model block</b> ..... 1-29
	<b>Motion Planning with RRT for Fixed-Wing UAV</b> ..... 1-40
	<b>UAV Package Delivery</b> ..... 1-46
	<b>UAV Scenario Tutorial</b> ..... 1-57
	<b>Tune UAV Parameters Using MAVLink Parameter Protocol</b> ..... 1-61
	<b>Exchange Data for MAVLink Microservices like Mission Protocol and Parameter Protocol Using Simulink</b> ..... 1-66

<b>2</b>	<b>3D Simulation - User's Guide</b>
	<hr/>
	<b>Unreal Engine Simulation for Unmanned Aerial Vehicles</b> ..... 2-2
	Unreal Engine Simulation Blocks ..... 2-2
	Algorithm Testing and Visualization ..... 2-3
	<b>Unreal Engine Simulation Environment Requirements and Limitations</b> ..... 2-5
	Software Requirements ..... 2-5
	Minimum Hardware Requirements ..... 2-5
	Limitations ..... 2-5
	<b>How Unreal Engine Simulation for UAVs Works</b> ..... 2-7
	Communication with 3D Simulation Environment ..... 2-7
	Block Execution Order ..... 2-7

<b>Coordinate Systems for Unreal Engine Simulation in UAV Toolbox . . . . .</b>	<b>2-9</b>
Earth-Fixed (Inertial) Coordinate System . . . . .	2-9
Body (Non-Inertial) Coordinate System . . . . .	2-9
Unreal Engine World Coordinate System . . . . .	2-11
<b>Choose a Sensor for Unreal Engine Simulation . . . . .</b>	<b>2-13</b>
<b>Simulate Simple Flight Scenario and Sensor in Unreal Engine Environment . . . . .</b>	<b>2-14</b>
<b>Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation . . . . .</b>	<b>2-19</b>
<b>Customize Unreal Engine Scenes for UAVs . . . . .</b>	<b>2-24</b>
<b>Install Support Package for Customizing Scenes . . . . .</b>	<b>2-25</b>
Verify Software and Hardware Requirements . . . . .	2-25
Install Support Package . . . . .	2-25
Set Up Scene Customization Using Support Package . . . . .	2-25
<b>Customize Unreal Engine Scenes Using Simulink and Unreal Editor . . . . .</b>	<b>2-28</b>
Open Unreal Editor from Simulink . . . . .	2-28
Reparent Actor Blueprint . . . . .	2-29
Create or Modify Scenes in Unreal Editor . . . . .	2-29
Run Simulation . . . . .	2-31
<b>Package Custom Scenes into Executable . . . . .</b>	<b>2-33</b>
Package Scene into Executable Using Unreal Engine . . . . .	2-33
<b>Apply Semantic Segmentation Labels to Custom Scenes . . . . .</b>	<b>2-35</b>

# UAV Toolbox Examples

---

## Visualize and Playback MAVLink Flight Log

This example shows how to load a telemetry log (TLOG) containing MAVLink packets into MATLAB®. Details of the messages are extracted for plotting. Then, to simulate the flight again, the messages are republished over the MAVLink communication interface. This publishing mimics an unmanned aerial vehicle (UAV) executing the flight recorded in the tlog.

### Load MAVLink TLOG

Create a `mavlinkdialect` object using the "common.xml" dialect. Use `mavlinktlog` with this dialect to load the TLOG data.

```
dialect = mavlinkdialect('common.xml');
logimport = mavlinktlog('mavlink_flightlog.tlog',dialect);
```

Extract the GPS messages from the TLOG and visualize them using `geoplot`.

```
msgs = readmsg(logimport, 'MessageName', 'GPS_RAW_INT', ...
               'Time',[0 100]);
latlon = msgs.Messages{1};
% filter out zero-valued messages
latlon = latlon(latlon.lat ~= 0 & latlon.lon ~= 0, :);
figure()
geoplot(double(latlon.lat)/1e7, double(latlon.lon)/1e7);
```

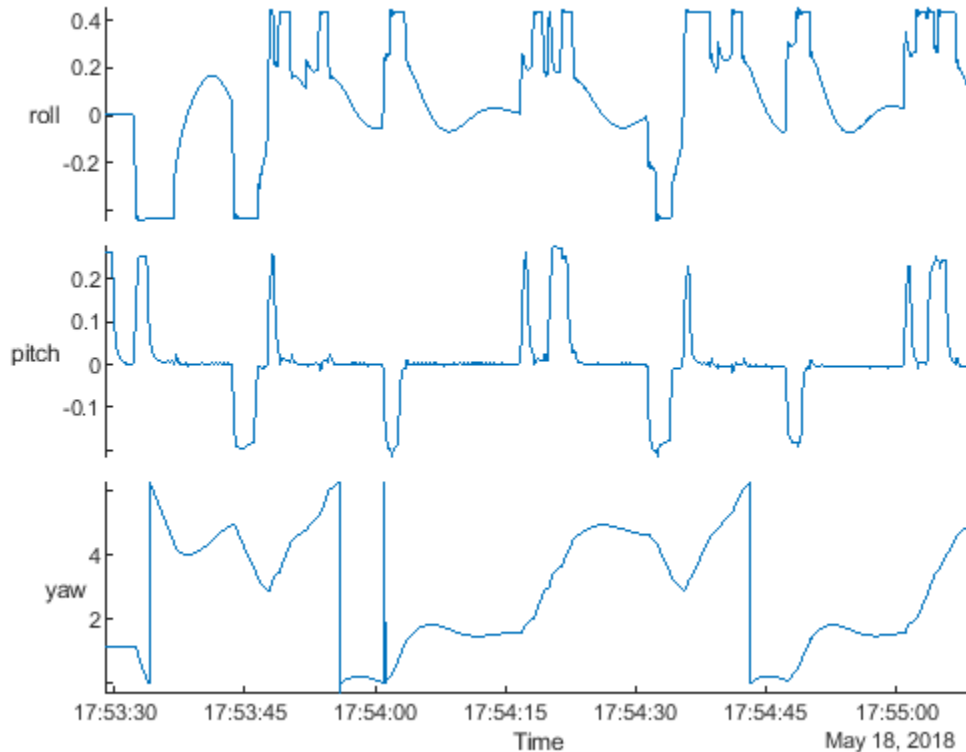


Extract the attitude messages from the TLOG. Specify the message name for attitude messages. Plot the roll, pitch, yaw data using `stackedplot`.

```

msgs = readmsg(logimport, 'MessageName', 'ATTITUDE', 'Time', [0 100]);
figure()
stackedplot(msgs.Messages{1}, {'roll', 'pitch', 'yaw'});

```



### Playback MAVLink Log Entries

Create a MAVLink communication interface and publish the messages from the TLOG to user defined UDP port. Create a sender and receiver for passing the MAVLink messages. This communication system works the same way that real hardware would publish messages using the MAVLink communication protocols.

```

sender = mavlinkio(dialect, 'SystemID', 1, 'ComponentID', 1, ...
                  'AutopilotType', "MAV_AUTOPILOT_GENERIC", ...
                  'ComponentType', "MAV_TYPE_QUADROTOR");
connect(sender, 'UDP');

destinationPort = 14550;
destinationHost = '127.0.0.1';

receiver = mavlinkio(dialect);
connect(receiver, 'UDP', 'LocalPort', destinationPort);

subscriber = mavlinksub(receiver, 'ATTITUDE', 'NewMessageFcn', @(~,msg) disp(msg.Payload));

```

Send the first 100 messages at a rate of 50 Hz.

```
payloads = table2struct(msgs.Messages{1});
attitudeDefinition = msginfo(dialect, 'ATTITUDE');
for msgIdx = 1:100
    sendudpmsg(sender,struct('MsgID', attitudeDefinition.MessageID, 'Payload', payloads(msgIdx))
    pause(1/50);
end
```

Disconnect from both MAVLink communication interfaces.

```
disconnect(receiver)
disconnect(sender)
```



## Flight Instrument Gauge Visualization for a Drone

Import and visualize a drone flight log using 3-D animations and flight instrument gauges. This example obtains a high level overview of flight performance in MATLAB® using “Flight Instruments” (Aerospace Toolbox) functions in Aerospace Toolbox™. Then, to view signals in a custom interface in Simulink®, the example uses the “Flight Instruments” (Aerospace Blockset) blocks from Aerospace Blockset™

The example extracts the signals of interest from a ULOG file and plays back the UAV flight trajectory in MATLAB. Then, those signals are replayed in a Simulink model using instrument blocks.

### Import a Flight log

A drone log file records information about the flight at regular time intervals. This information gives insight into the flight performance. Flight instrument gauges display navigation variables such as attitude, altitude, and heading of the drone. The ULOG log file for this example was obtained from an airplane model running in the Gazebo simulator.

Import the logfile using `ulogreader`. Create a `flightLogSignalMapping` object for ULOG files.

To understand the convention of the signals, the units, and their reference frame, inspect the information within the `plotter` object. This information about units within log file becomes important when connecting the signals to flight instrument gauges.

```
data = ulogreader("fwflight.ulg");
plotter = flightLogSignalMapping("ulog");
info(plotter, "Signal")
```

```
ans=14x4 table
      SignalName      IsMapped
-----
"Accel"              true      "AccelX, AccelY, AccelZ"
"Airspeed"           true      "PressDiff, IndicatedAirSpeed, Temperature"
"AttitudeEuler"      true      "Roll, Pitch, Yaw"
"AttitudeRate"       true      "BodyRotationRateX, BodyRotationRateY, BodyRotationRateZ"
"AttitudeTargetEuler" true      "RollTarget, PitchTarget, YawTarget"
"Barometer"          true      "PressAbs, PressAltitude, Temperature"
"Battery"             true      "Voltage_1, Voltage_2, Voltage_3, Voltage_4, Voltage_5, Voltage_6"
"GPS"                 true      "Latitude, Longitude, Altitude, GroundSpeed, CourseAngle"
"Gyro"                true      "GyroX, GyroY, GyroZ"
"LocalNED"            true      "X, Y, Z"
"LocalNEDTarget"     true      "XTarget, YTarget, ZTarget"
"LocalNEDVel"         true      "VX, VY, VZ"
"LocalNEDVelTarget"  true      "VXTarget, VYTarget, VZTarget"
"Mag"                 true      "MagX, MagY, MagZ"
```

### Extract Signals of Interest

To visualize the drone flight using instrument gauges, extract the attitude, position, velocity, and airspeed at each timestep. Specify the appropriate signal name from the info table in the previous step. Call the `extract` function with the appropriate signal names. The time vector element of signals are adjusted so they start at 0 seconds.

```
% Extract attitude and roll-pitch-yaw data.
rpy = extract(plotter, data, "AttitudeEuler");
```

```

rpy{1}.Time=rpy{1}.Time-rpy{1}.Time(1);

RollData = timetable(rpy{1}.Time,rpy{1}.Roll,...
    'VariableNames',{'Roll'});
PitchData = timetable(rpy{1}.Time,rpy{1}.Pitch,...
    'VariableNames',{'Pitch'});
YawData = timetable(rpy{1}.Time,rpy{1}.Yaw,...
    'VariableNames',{'Yaw'});

% Extract position and xyz data.
Position = extract(plotter, data,"LocalNED");
Position{1}.Time = Position{1}.Time-Position{1}.Time(1);

X = timetable(Position{1}.Time,Position{1}.X,...
    'VariableNames',{'X'});
Y = timetable(Position{1}.Time,Position{1}.Y,...
    'VariableNames',{'Y'});
Z = timetable(Position{1}.Time,Position{1}.Z,...
    'VariableNames',{'Z'});

% Extract velocity data.
vel = extract(plotter, data,"LocalNEDVel");
vel{1}.Time=vel{1}.Time-vel{1}.Time(1);

XVel = timetable(vel{1}.Time,vel{1}.VX,...
    'VariableNames',{'VX'});
YVel = timetable(vel{1}.Time,vel{1}.VY,...
    'VariableNames',{'VY'});
ZVel = timetable(vel{1}.Time,vel{1}.VZ,...
    'VariableNames',{'VZ'});

% Extract Airspeed magnitude data.
airspeed = extract(plotter, data,"Airspeed");
Airspeed = timetable(airspeed{1}.Time,airspeed{1}.IndicatedAirSpeed,...
    'VariableNames',{'Airspeed'});

```

### Convert Units and Preprocess Data for Gauges

Our flight log records data in SI Units. The flight instrument gauges require a conversion to Aerospace Standard Unit System represented by English System. This conversion is handled in the visualization block available in attached Simulink model for the user. The turn coordinator indicates the yaw rate of the aircraft using an indicative banking motion (which differs from the bank angle). In order to compute the yaw rate, convert the angular rates from body frame to vehicle frame as given below:

$$\dot{\psi} = \frac{q\cos(\phi) + r\sin(\phi)}{\cos\theta}$$

The inclinometer ball within turn coordinator indicates the sideslip of the aircraft. This sideslip angle is based on the angle between the body of the aircraft and computed airspeed. For an accurate airspeed, a good estimate of velocity and wind vector is required. Most small UAV's do not possess sensors to estimate wind vector data or airspeed while flying. UAV's can face between 20-50% of their airspeed in the form of crosswinds.

$$V_g - V_w = V_a$$

To compute sideslip and turn, extract wind and attitude rate data directly from the log file.

```

% Extract roll, pitch and yaw rates and an estimated windspeed.
[p,q,r,wn,we] = helperExtractUnmappedData(data);

% Merge timetables.
FlightData = synchronize(X,Y,Z,RollData,PitchData,YawData,XVel,YVel,ZVel,p,q,r,Airspeed,wn,we,'union');

% Assemble an array for the data.
FlightDataArray = double([seconds(FlightData.Time) FlightData.X FlightData.Y FlightData.Z FlightData.VX
FlightData.Pitch FlightData.Yaw,FlightData.VX,FlightData.VY,...
FlightData.VZ,FlightData.p,FlightData.q,FlightData.r,FlightData.Airspeed,FlightData.wn,FlightData.we]);

% Ensure time rows are unique.
[~,ind]=unique(FlightDataArray(:,1));
FlightDataArray=FlightDataArray(ind,:);

% Preprocess time data to specific times.
flightdata = double(FlightDataArray(FlightDataArray(:,1)>=0,1:end));

```

### Visualize Standard Flight Instrument Data in MATLAB

To get a quick overview of the flight , use the animation interface introduced in the “Display Flight Trajectory Data Using Flight Instruments and Flight Animation” (Aerospace Toolbox) example. The helper function `helperDroneInstruments` creates an instrument animation interface.

```
helperDroneInstruments;
```





The **Airspeed** indicator dial indicates the speed of the drone. The **Artificial Horizon** indicator reveals the attitude of the drone excluding yaw. The **Altimeter** and **Climb Rate** indicator reveal the altitude as recorded within the barometer and the climb rate sensors respectively. The **Turn Coordinator** indicates the yaw rate of the aircraft and sideslip. If the inclinometer skews towards left or right, this denotes a slip or skid situation. In a coordinated turn, the sideslip should be zero.

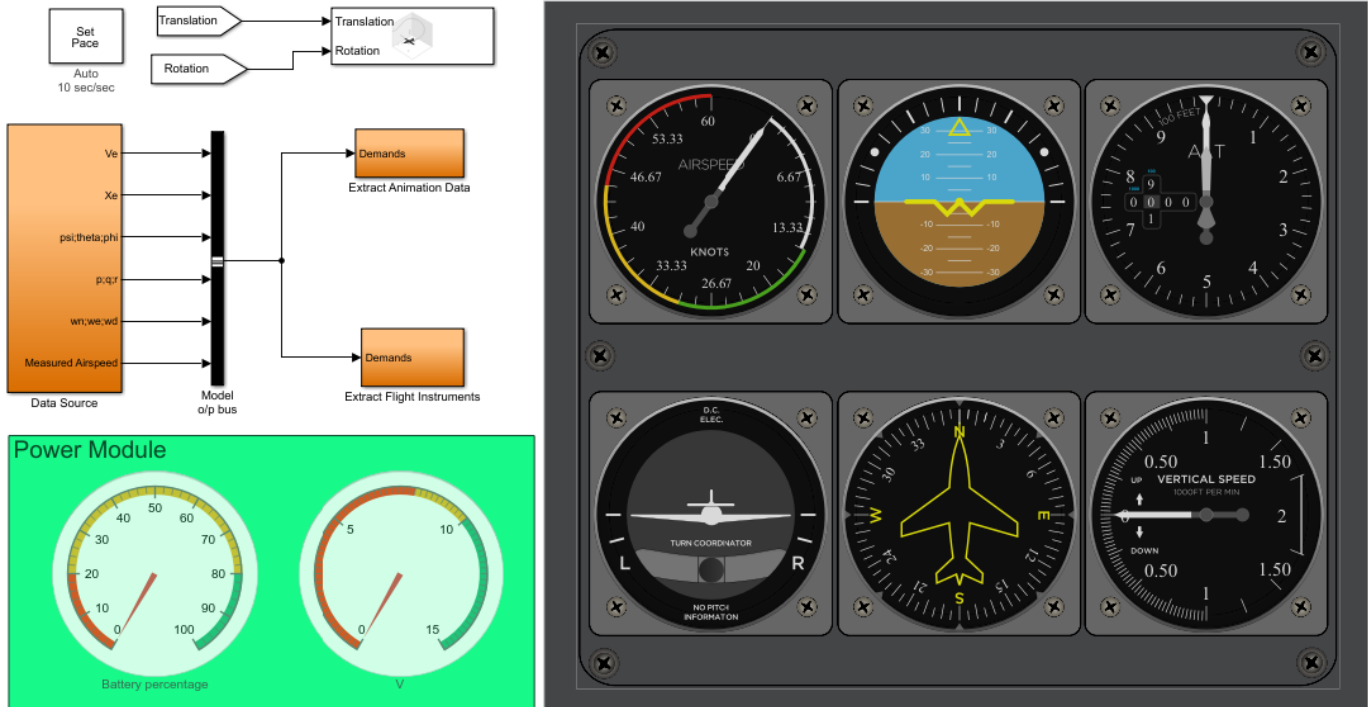
### Visualize Signals in Simulink

In Simulink, you can create custom visualizations of signals using instrument blocks to help diagnose problems with a flight. For example, voltage and battery data in log files can help diagnose failures due to inadequate power or voltage spikes. Extract this batter data below to visualize them.

```
% Extract battery data.
Battery = extract(plotter,data,"Battery");
% Extract voltage data from topic.
Voltage = timetable(Battery{1}.Time,Battery{1}.Voltage_1,...
    'VariableNames',{'Voltage_1'});
% Extract remaing battery capacity data from topic.
Capacity = timetable(Battery{1}.Time,Battery{1}.RemainingCapacity,...
    'VariableNames',{'RemainingCapacity'});
```

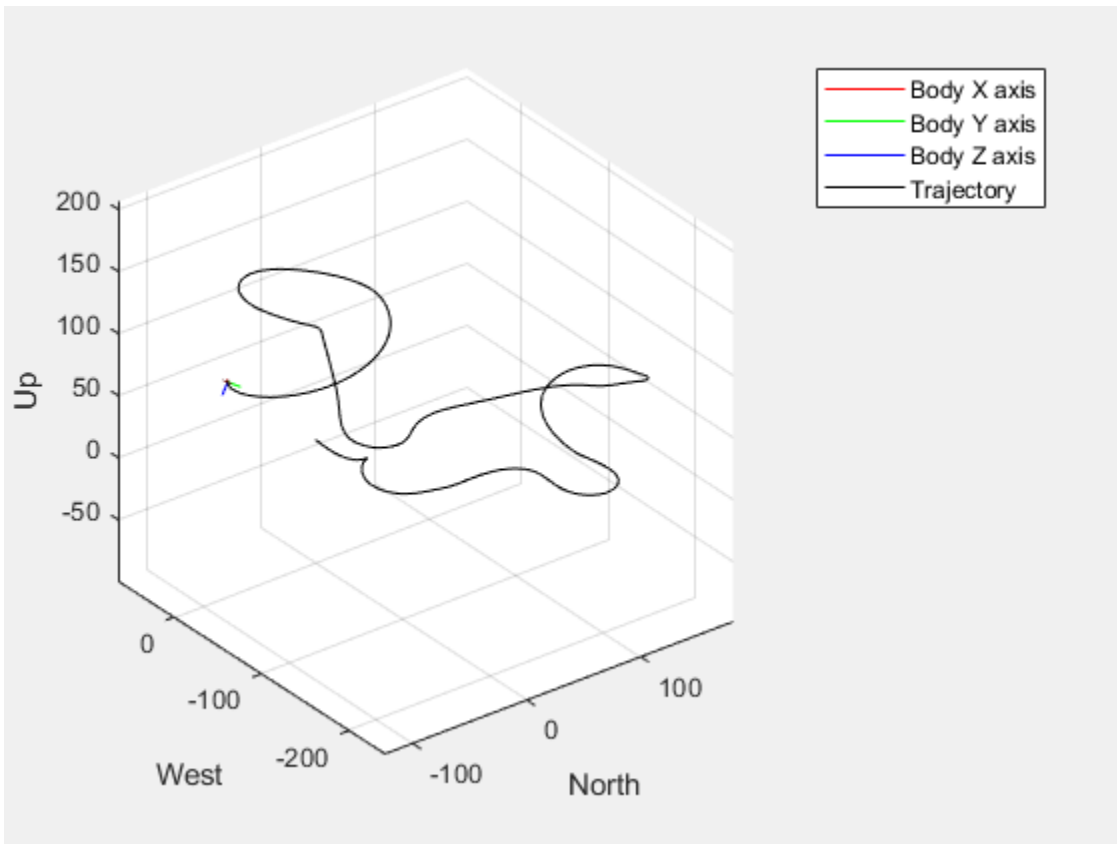
Open the 'dronegauge' model, which takes the loaded data and displays it on the different gauges and the UAV animation figure.

```
open_system('dronegauges');
```



**Run** the model. The generated figure shows the trajectory of the UAV in real time and the gauges show the current status of the flight.

```
sim('dronegauges');
```



## Visualize Custom Flight Log

Configure the `flightLogSignalMapping` object to visualize data from a custom flight log.

### Load Custom Flight Log

In this example, it is assumed that flight data is already parsed into MATLAB® and stored as a mat file. This example focuses on configuring the `flightLogSignalMapping` object so that it could properly handle the log data saved in the mat file and visualize them. The data, `customFlightData.mat`, stores a struct that contains 3 fields. `Fs` is the sampling frequency of the signals stored in the mat file. `IMU` and `Trajectory` are matrices containing actual flight information. The trajectory and IMU data are based on a simulated flight that follows a projected rectangular path on an XY-plane.

```
customData = load("customFlightData.mat");
logData = customData.logData

logData = struct with fields:
    IMU: [2785x9 double]
    Fs: 100
    Trajectory: [2785x10 double]
```

The `IMU` field in `logData` is an  $n$ -by-9 matrix, with the first 3 columns as accelerometer readings in  $m/s^2$ . The next 3 columns are gyroscope readings in  $rad/s$ , and the last 3 columns are magnetometer readings in  $\mu T$ .

```
logData.IMU(1:5, :)
```

```
ans = 5x9
```

```
    0.8208    0.7968    10.7424    0.0862    0.0873    0.0862    327.6000    297.6000    283.8000
    0.8016    0.8160    10.7904    0.0883    0.0873    0.0862    327.6000    297.6000    283.8000
    0.7680    0.7680    10.7568    0.0862    0.0851    0.0851    327.6000    297.6000    283.8000
    0.8208    0.7536    10.7520    0.0873    0.0883    0.0819    327.6000    297.6000    283.8000
    0.7872    0.7728    10.7328    0.0873    0.0862    0.0830    327.6000    297.6000    283.8000
```

The `Trajectory` field in `logData` is an  $n$ -by-10 matrix, with the first 3 columns are XYZ NED coordinates in  $m$ . The next 3 columns are velocity in XYZ NED direction in  $m/s$ , and the last 4 columns are quaternions describing the UAV rotation from the inertia NED frame to body frame. Each row is a single point of the trajectory with all these parameters defined.

```
logData.Trajectory(1:5, :)
```

```
ans = 5x10
```

```
    0.0200         0   -4.0000    2.0000         0   -0.0036    1.0000         0         0   -0.
    0.0400         0   -4.0001    2.0000         0   -0.0072    1.0000         0         0   -0.
    0.0600         0   -4.0002    2.0000         0   -0.0108    1.0000         0         0   -0.
    0.0800         0   -4.0003    2.0000         0   -0.0143    1.0000         0         0   -0.
    0.1000         0   -4.0004    2.0000         0   -0.0179    1.0000         0         0   -0.
```

### Visualize Custom Flight Log Using Pre-defined Signal Format and Plots

A `flightLogSignalMapping` object is created with no input argument since the custom log format is not following a standard "uLog" or "tlog" definition.

```
customPlotter = flightLogSignalMapping;
```

The object has a predefined set of signals that you can map. By mapping these predefined signals, you gain access to a set of predefined plots. Notice that a few signals has a "#" symbol suffix. This means that you can optionally suffix these signal names with integers so that the flight log plotter can handle multiple of signals of this kind, such as secondary IMU signals, barometer readings, etc. Call `info`

`% Predefined signals`

```
info(customPlotter, "Signal")
```

ans=18x4 table

SignalName	IsMapped	
"Accel#"	false	"AccelX, AccelY, AccelZ"
"Airspeed#"	false	"PressDiff, IndicatedAirSpeed, Temperature"
"AttitudeEuler"	false	"Roll, Pitch, Yaw"
"AttitudeRate"	false	"BodyRotationRateX, BodyRotationRateY, BodyRotationRateZ"
"AttitudeTargetEuler"	false	"RollTarget, PitchTarget, YawTarget"
"Barometer#"	false	"PressAbs, PressAltitude, Temperature"
"Battery"	false	"Voltage_1, Voltage_2, Voltage_3, Voltage_4, Voltage_5, Voltage_6"
"GPS#"	false	"Latitude, Longitude, Altitude, GroundSpeed, CourseAngle"
"Gyro#"	false	"GyroX, GyroY, GyroZ"
"LocalENU"	false	"X, Y, Z"
"LocalENUTarget"	false	"XTarget, YTarget, ZTarget"
"LocalENUVel"	false	"VX, VY, VZ"
"LocalENUVelTarget"	false	"VXTarget, VYTarget, VZTarget"
"LocalNED"	false	"X, Y, Z"
"LocalNEDTarget"	false	"XTarget, YTarget, ZTarget"
"LocalNEDVel"	false	"VX, VY, VZ"
:		

`% Predefined plots`

```
info(customPlotter, "Plot")
```

ans=10x4 table

PlotName	ReadyToPlot	MissingSignals	
"Attitude"	false	"AttitudeEuler, AttitudeRate, Gyro#"	"AttitudeEuler"
"AttitudeControl"	false	"AttitudeEuler, AttitudeTargetEuler"	"AttitudeEuler"
"Battery"	false	"Battery"	"Battery"
"Compass"	false	"AttitudeEuler, Mag#, GPS#"	"AttitudeEuler"
"GPS2D"	false	"GPS#"	"GPS#"
"Height"	false	"Barometer#, GPS#, LocalNED"	"Barometer#"
"Speed"	false	"GPS#, Airspeed#"	"GPS#, Airspeed#"
"Trajectory"	false	"LocalNED, LocalNEDTarget"	"LocalNED, LocalNEDTarget"
"TrajectoryTracking"	false	"LocalNED, LocalNEDTarget"	"LocalNED, LocalNEDTarget"
"TrajectoryVelTracking"	false	"LocalNEDVel, LocalNEDVelTarget"	"LocalNEDVel, LocalNEDVelTarget"



The `flightLogSignalMapping` object needs to know how data is stored in the flight log before it can visualize the data. To associate signal names with function handles that access the relevant information in the `logData`, you must map signals using `mapSignal`. Each signal is defined as a timestamp vector and a signal value matrix.

For example, to map the `Gyro#` signal, define a `timeAccess` function handle based on the sensor data sampling frequency. This function handle generates the timestamp vector for the signal values using a global timestamp interval for the data.

```
timeAccess = @(x)seconds(1/x.Fs*(1:size(x.IMU)));
```

Next, check what fields must be defined for the `Gyro#` signal using `info`.

```
info(customPlotter, "Signal", "Gyro#")
```

```
ans=1x4 table
   SignalName   IsMapped   SignalFields   FieldUnits
   _____   _____   _____   _____
   "Gyro#"      false      "GyroX, GyroY, GyroZ"  "rad/s, rad/s, rad/s"
```

The `Gyro#` signal needs three columns containing the gyroscope readings for the XYZ axes. Define the `gyroAccess` function handle accordingly and map it with `timeAccess` using `mapSignal`.

```
gyroAccess = @(x)x.IMU(:,4:6);
mapSignal(customPlotter, "Gyro", timeAccess, gyroAccess);
```

Similarly, map other predefined signals for data that is present in the flight log. Define the value function handles for the data. Map the signals using the same `timeAccess` timestamp vector function.

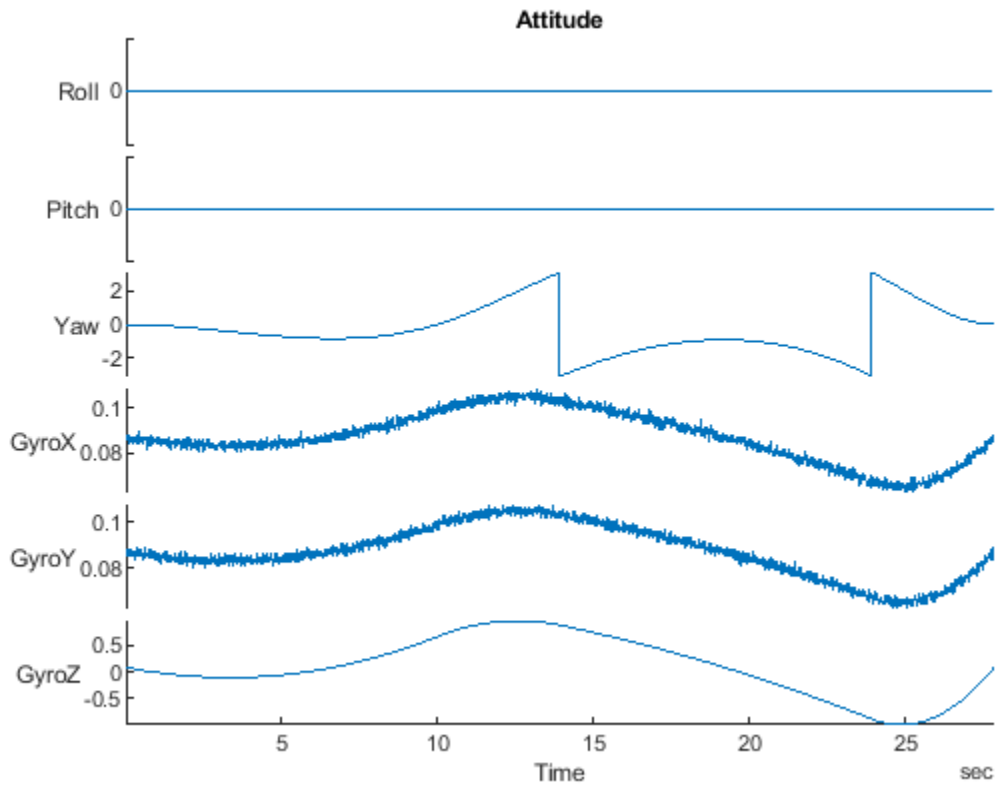
```
% IMU data stores accelerometer and magnetometer data.
accelAccess = @(x)x.IMU(:,1:3);
magAccess = @(x)x.IMU(:,7:9)*1e-2;

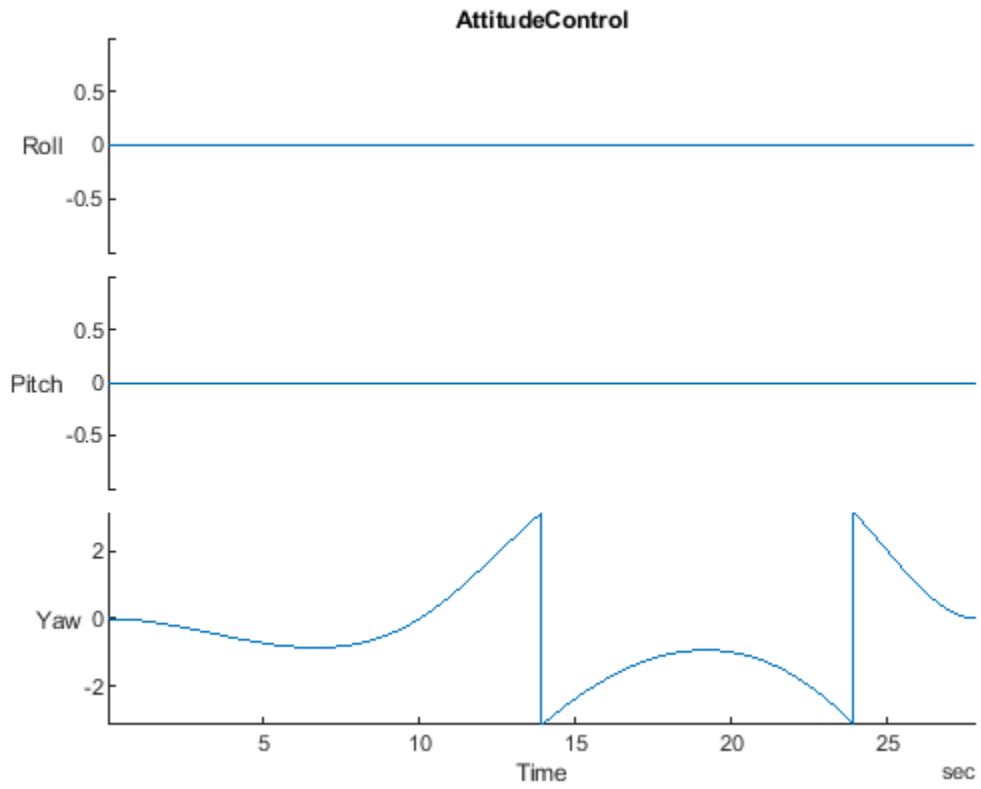
% Flight trajectory in local NED coordinates
% XYZ coordinates
nedAccess = @(x)x.Trajectory(:, 1:3);
% XYZ velocities
nedVelAccess = @(x)x.Trajectory(:, 4:6);
% Roll Pitch Yaw rotations converted from a quaternion
attitudeAccess = @(x)flip(quat2eul(x.Trajectory(:, 7:10)),2);

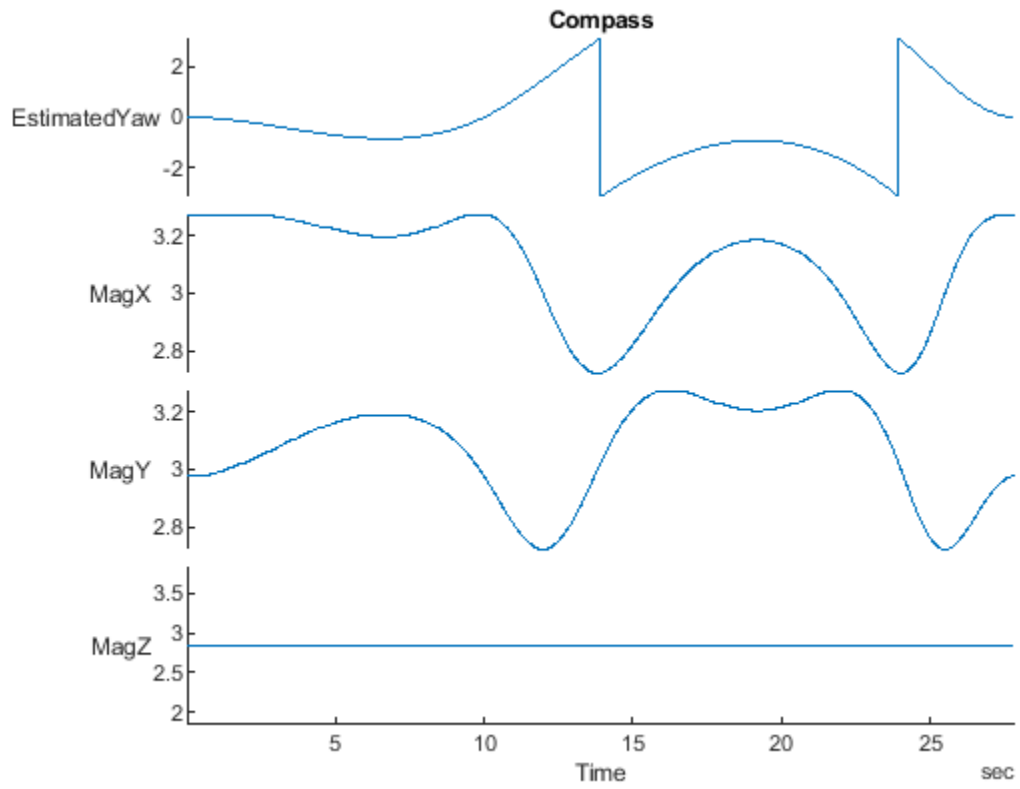
% Configure flightLogSignalMapping for custom data
mapSignal(customPlotter, "Accel", timeAccess, accelAccess);
mapSignal(customPlotter, "Mag", timeAccess, magAccess);
mapSignal(customPlotter, "LocalNED", timeAccess, nedAccess);
mapSignal(customPlotter, "LocalNEDVel", timeAccess, nedVelAccess);
mapSignal(customPlotter, "AttitudeEuler", timeAccess, attitudeAccess);
```

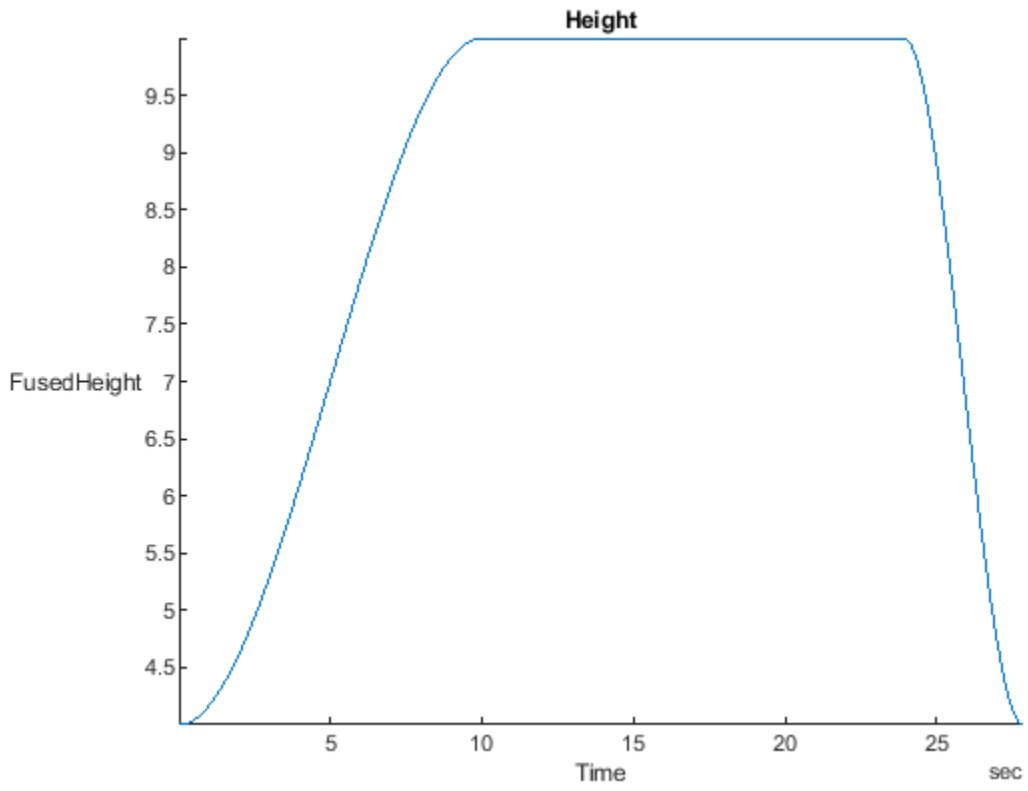
Once all signals are mapped, `customPlotter` is ready to generate plots based on signal data stored in the log. To visualize the flight log data, call `show` and specify `logData`. All the plots available based on the mapped signals are shown in figures.

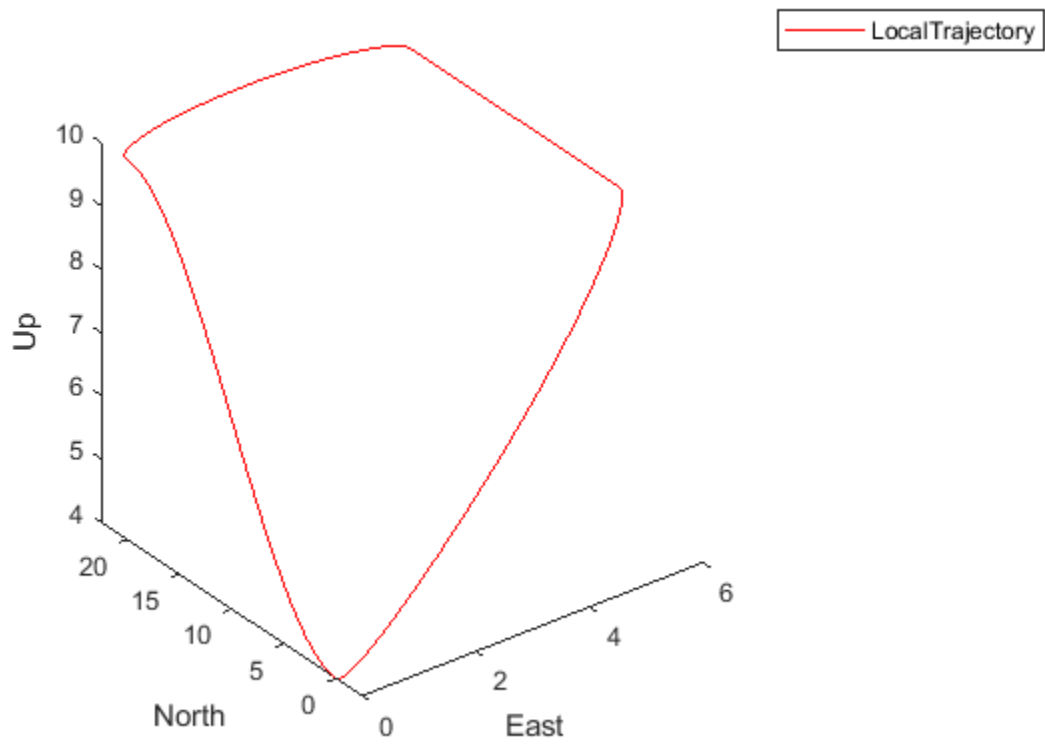
```
predefinedPlots = show(customPlotter, logData);
```

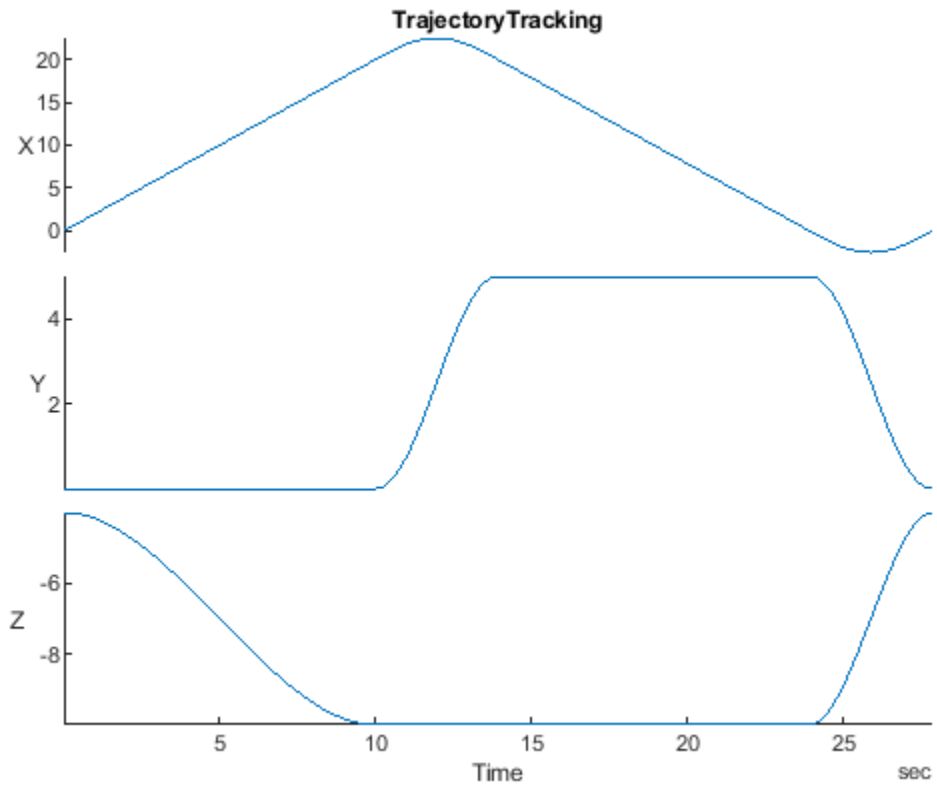


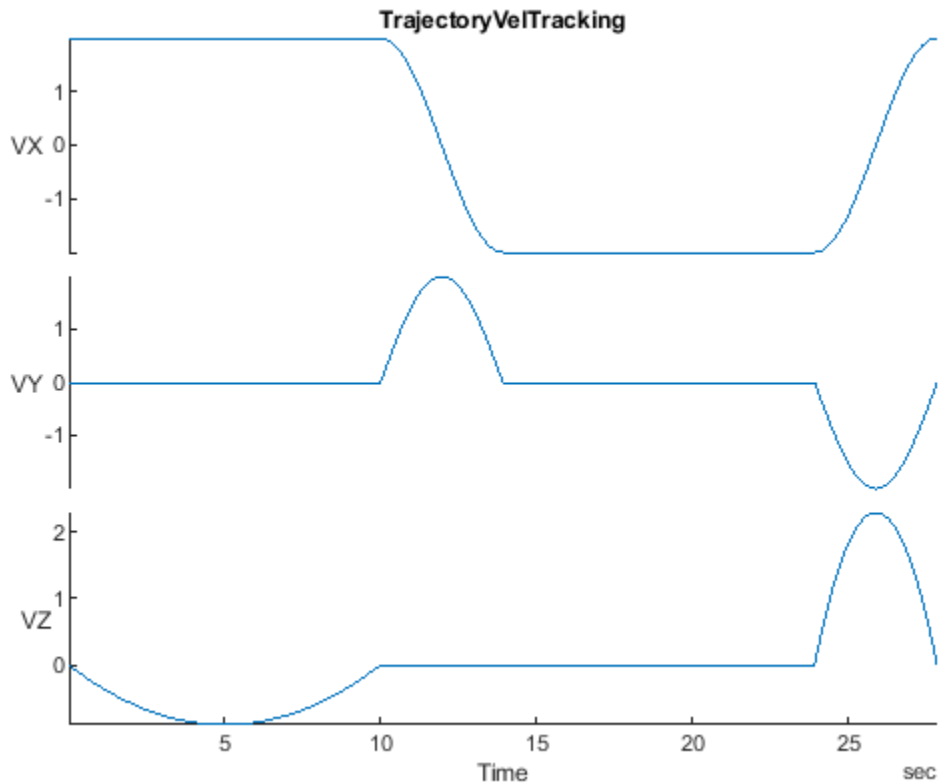












### Visualize Custom Flight Log with Custom Plot

For mod details log analysis, define more signals and add more plots other than predefined plots stored in `flightLogSignalMapping`. Specify a function handle that filters accelerations greater than 1.

```
accelThreshold = @(x)(vecnorm(accelAccess(x)')>11)';
mapSignal(customPlotter, "HighAccel", timeAccess, accelThreshold, "AccelGreaterThan11", "N/A");
```

Call `updatePlot` to add custom plots. Specify the flight log plotter object and a name for the plot as the first two argument. To specify a time series of data, use "Timeseries" as the third argument, and then list the data.

```
updatePlot(customPlotter, "AnalyzeAccel", "Timeseries", ["HighAccel.AccelGreaterThan11", "LocalNEDE"]);
```

Define a custom function handle for generating a figure handle (see function definition below). This function generates a periodogram using `fft` and other functions on the acceleration data and plots them. The function returns a function handle.

```
updatePlot(customPlotter, "plotFFTAccel", @(acc)plotFFTAccel(acc), "Accel");
```

Check that `customPlotter` now contains a new signal and two new plots using `info`.

```
info(customPlotter, "Signal")
```

```
ans=19x4 table
      SignalName      IsMapped
```



```

"Accel"           true      "AccelX, AccelY, AccelZ"
"AttitudeEuler"  true      "Roll, Pitch, Yaw"
"Gyro"           true      "GyroX, GyroY, GyroZ"
"HighAccel"      true      "AccelGreaterThan11"
"LocalNED"       true      "X, Y, Z"
"LocalNEDVel"    true      "VX, VY, VZ"
"Mag"           true      "MagX, MagY, MagZ"
"Airspeed#"      false     "PressDiff, IndicatedAirSpeed, Temperature"
"AttitudeRate"   false     "BodyRotationRateX, BodyRotationRateY, BodyRotationRateZ"
"AttitudeTargetEuler" false    "RollTarget, PitchTarget, YawTarget"
"Barometer#"     false     "PressAbs, PressAltitude, Temperature"
"Battery"        false     "Voltage_1, Voltage_2, Voltage_3, Voltage_4, Voltage_5, Voltage_6"
"GPS#"          false     "Latitude, Longitude, Altitude, GroundSpeed, CourseAngle"
"LocalENU"       false     "X, Y, Z"
"LocalENUTarget" false     "XTarget, YTarget, ZTarget"
"LocalENUVel"    false     "VX, VY, VZ"
:

```

```
info(customPlotter, "Plot")
```

```
ans=12x4 table
```

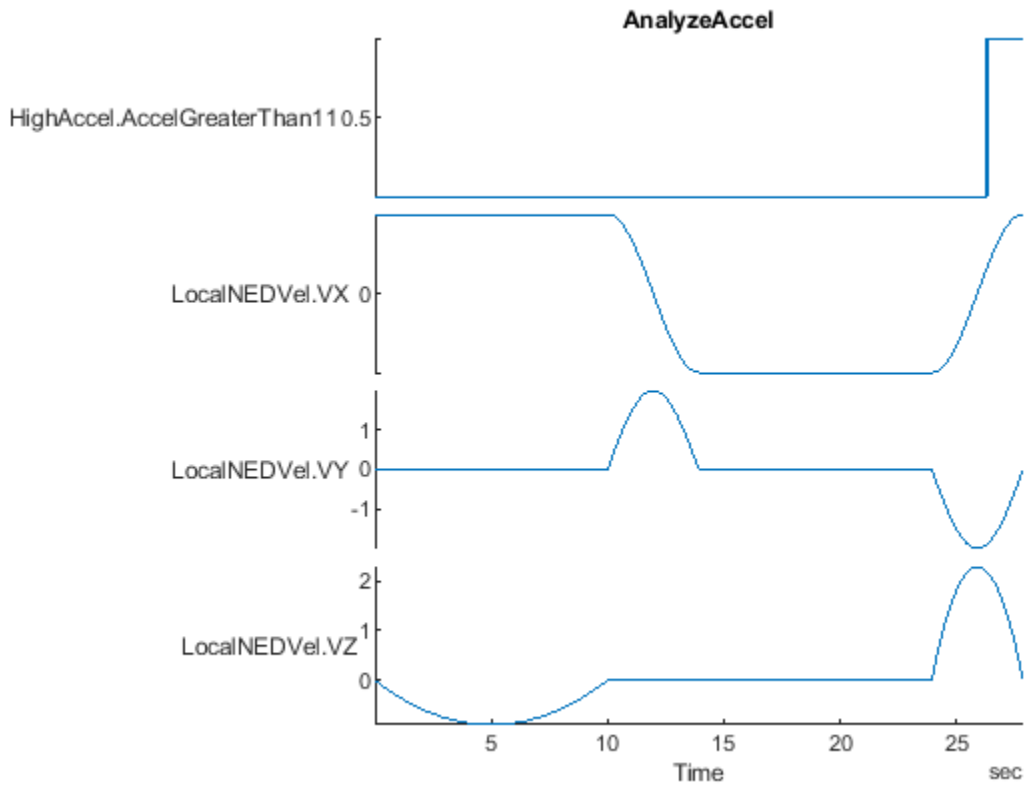
PlotName	ReadyToPlot	MissingSignals	RequiredSignals
"AnalyzeAccel"	true	" "	"HighAccel, LocalNEDVel"
"Attitude"	true	"AttitudeRate"	"AttitudeEuler, AttitudeRate"
"AttitudeControl"	true	"AttitudeTargetEuler"	"AttitudeEuler, AttitudeTarget"
"Compass"	true	"GPS#"	"AttitudeEuler, Mag#, GPS#"
"Height"	true	"Barometer#, GPS#"	"Barometer#, GPS#, LocalNED"
"Trajectory"	true	"LocalNEDTarget"	"LocalNED, LocalNEDTarget"
"TrajectoryTracking"	true	"LocalNEDTarget"	"LocalNED, LocalNEDTarget"
"TrajectoryVelTracking"	true	"LocalNEDVelTarget"	"LocalNEDVel, LocalNEDVelTarget"
"plotFFTAccel"	true	" "	"Accel"
"Battery"	false	"Battery"	"Battery"
"GPS2D"	false	"GPS#"	"GPS#"
"Speed"	false	"GPS#, Airspeed#"	"GPS#, Airspeed#"

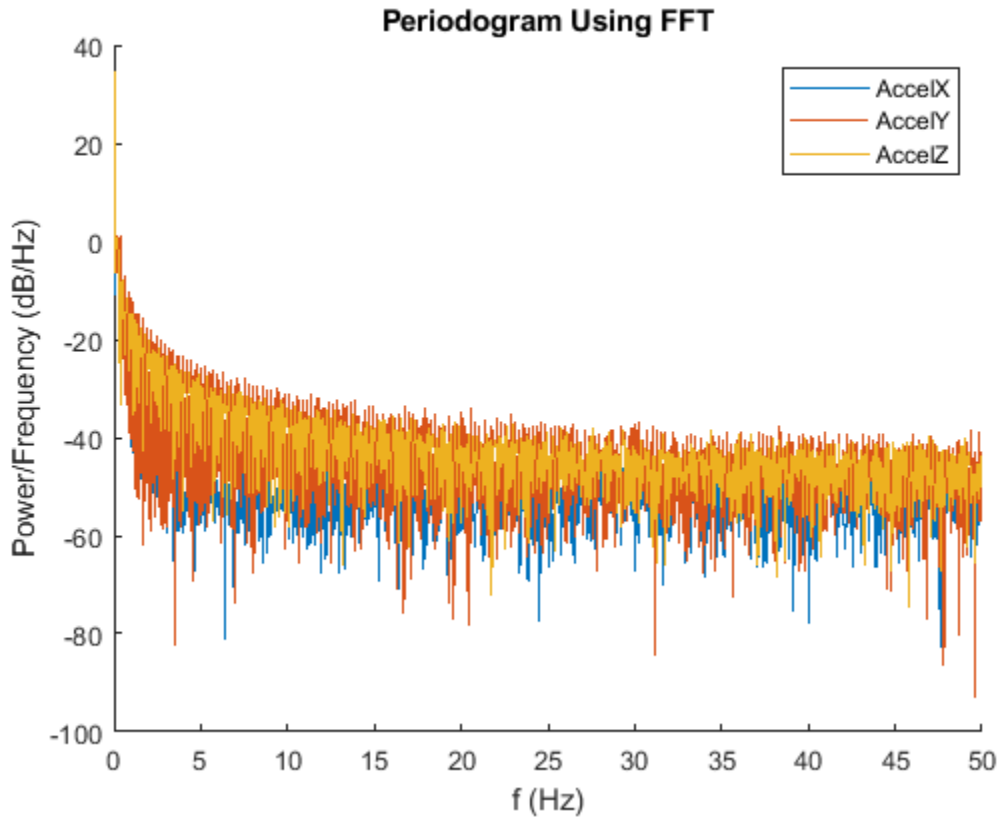
Specify which plot names you want to plot. Call show using "PlotsToShow" to visualize the analysis of the acceleration data.

```

accelAnalysisProfile = ["AnalyzeAccel", "plotFFTAccel"];
accelAnalysisPlots = show(customPlotter, logData, "PlotsToShow", accelAnalysisProfile);

```





This example has shown how to use the `flightLogSignalMapping` object to look at predefined signals and plots, as well as customize your own plots for flight log analysis.

### Analyze Acceleration Data Function Definition

```
function h = plotFFTAccel(acc)
    h = figure("Name", "AccelFFT");
    ax = newplot(h);
    v = acc.Values{1};
    Fs = v.Properties.SampleRate;
    N = floor(length(v.AccelX)/2)*2;
    hold(ax, "on");
    for idx = 1:3
        x = v{1:N, idx};
        xdft = fft(x);
        xdft = xdft(1:N/2+1);
        psdx = (1/(Fs*N)) * abs(xdft).^2;
        psdx(2:end-1) = 2*psdx(2:end-1);
        freq = 0:Fs/length(x):Fs/2;
        plot(ax, freq, 10*log10(psdx));
    end
    hold(ax, "off");
    title("Periodogram Using FFT");
    xlabel("f (Hz)");
    ylabel("Power/Frequency (dB/Hz)");
```

```
    legend("AccelX", "AccelY", "AccelZ");  
end
```

## Tuning Waypoint Follower for Fixed-Wing UAV

This example designs a waypoint following controller for a fixed-wing unmanned aerial vehicle (UAV) using the Guidance Model and Waypoint Follower blocks.

The example iterates through different control configurations and demonstrates UAV flight behavior by simulating a kinematic model for fixed-wing UAV.

### Guidance Model Configuration

The fixed-wing guidance model approximates the kinematic behavior of a closed-loop system consisting of the fixed-wing aerodynamics and an autopilot. This guidance model is suitable for simulating small UAV flights at a low-fidelity near the stable flight condition of the UAV. We can use the guidance model to simulate the flight status of the fixed-wing UAV guided by a waypoint follower.

The following Simulink® model can be used to observe the fixed-wing guidance model response to step control inputs.

```
open_system('uavStepResponse');
```

### Integration with Waypoint Follower

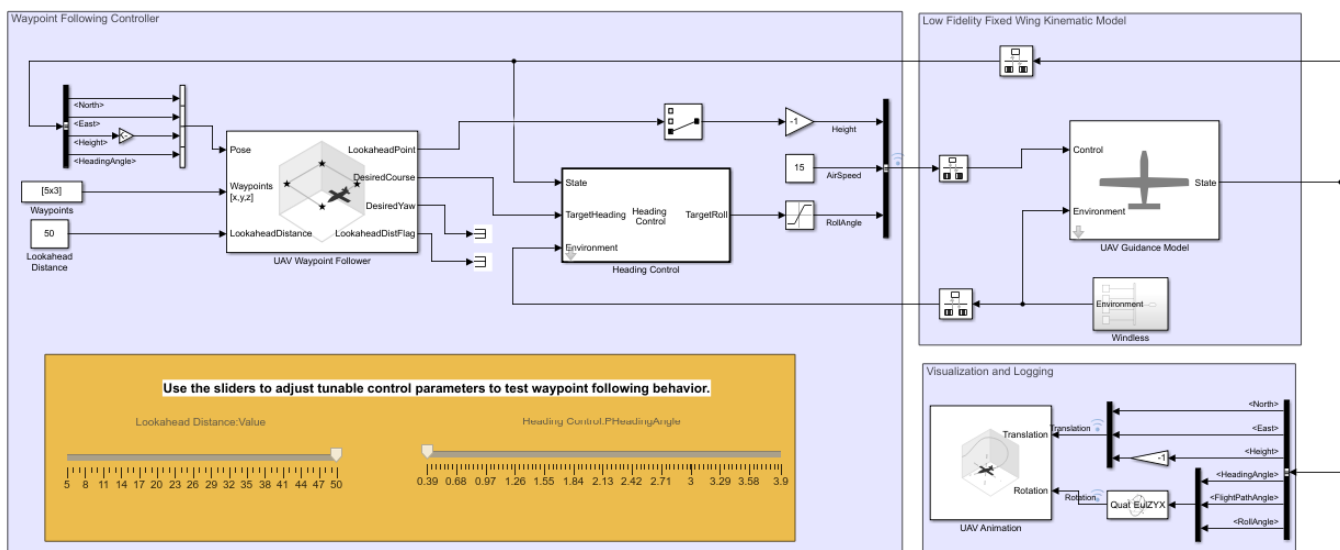
The `fixedWingPathFollowing` model integrates the waypoint follower with the fixed-wing guidance model. This model demonstrates how to extract necessary information from the guidance model output bus signal and feed them into the waypoint follower. The model assembles the control and environment inputs for the guidance model block.

```
open_system('fixedWingPathFollowing');
```

### Waypoint Follower Configuration

The waypoint follower controller includes two parts, a **UAV Waypoint Follower** block and a fixed-wing UAV heading controller.

Tune Waypoint Follower for Fixed-Wing UAV



The UAV Waypoint Follower block computes a desired heading for the UAV based on the current pose, lookahead distance, and a given set of waypoints. Flying along these heading directions, the UAV visits each waypoint (within the specified transition radius) in the list.

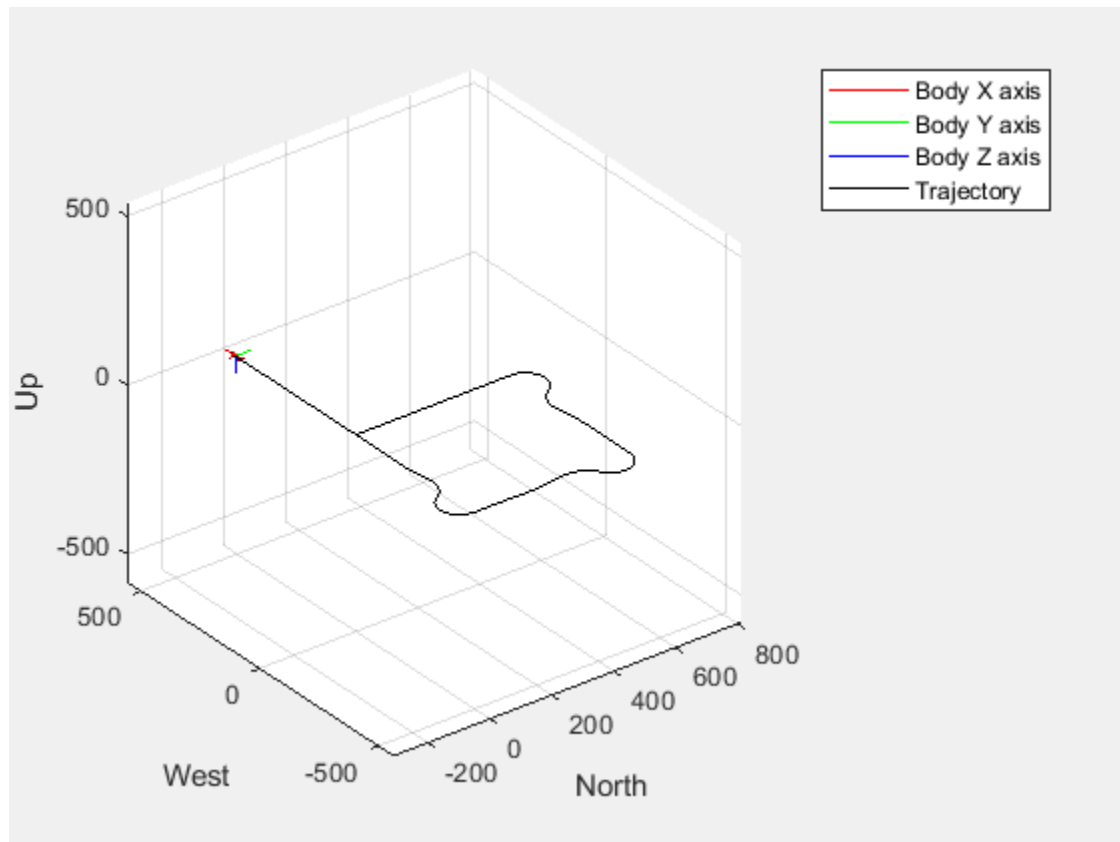
The Heading Control block is a proportional controller that regulates the UAV heading angle by controlling the roll angle under the coordinated-flight condition.

The UAV Animation block visualizes the UAV flight path and attitude. For fixed-wing simulation in a windless condition, the body pitch angle is the sum of the flight path angle and the attack angle. For small fixed-wing UAV, the attack angle is usually controlled by the autopilot and remains relatively small. For visualization purposes, we approximate the pitch angle with the flight path angle. In a windless, zero side-slip condition, the body yaw angle is the same as heading angle.

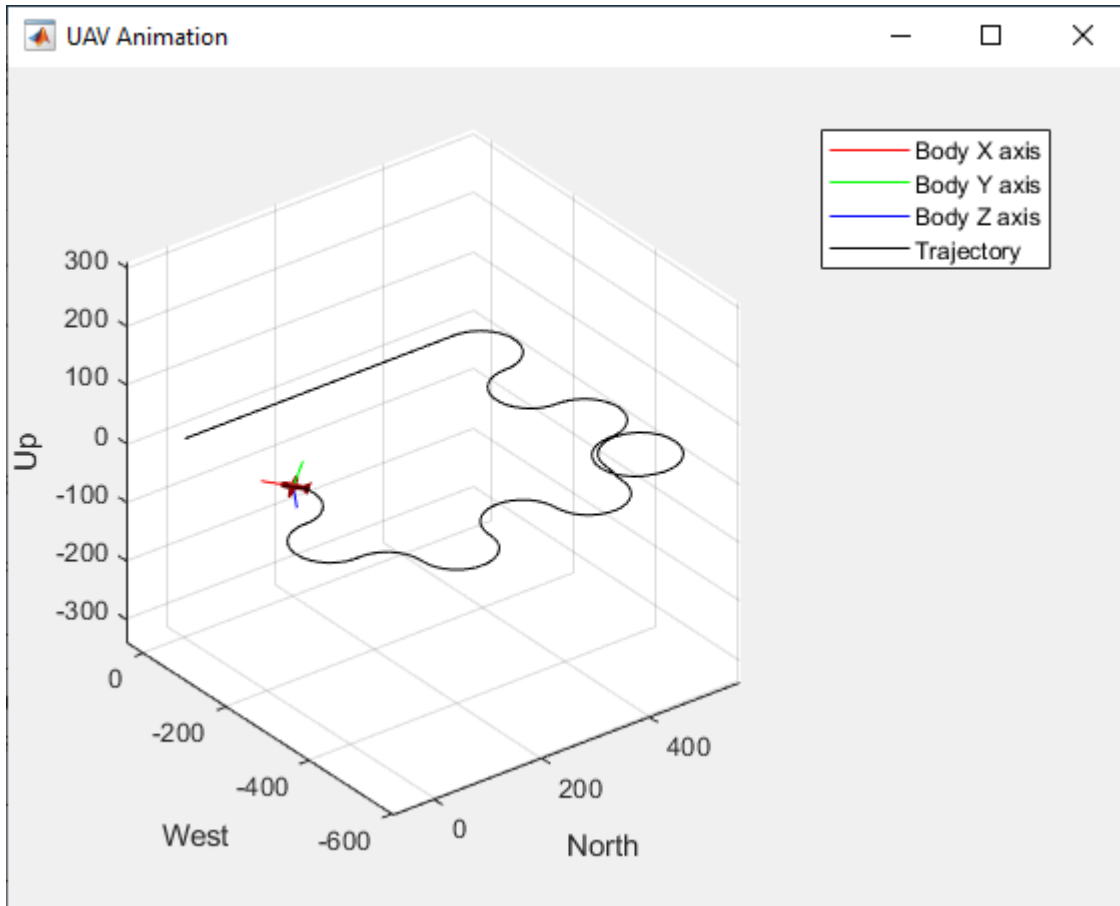
### Tune Waypoint Following Controller through Simulation

Simulate the model. Use the slider to adjust the controller waypoint following.

```
sim("fixedWingPathFollowing")
```

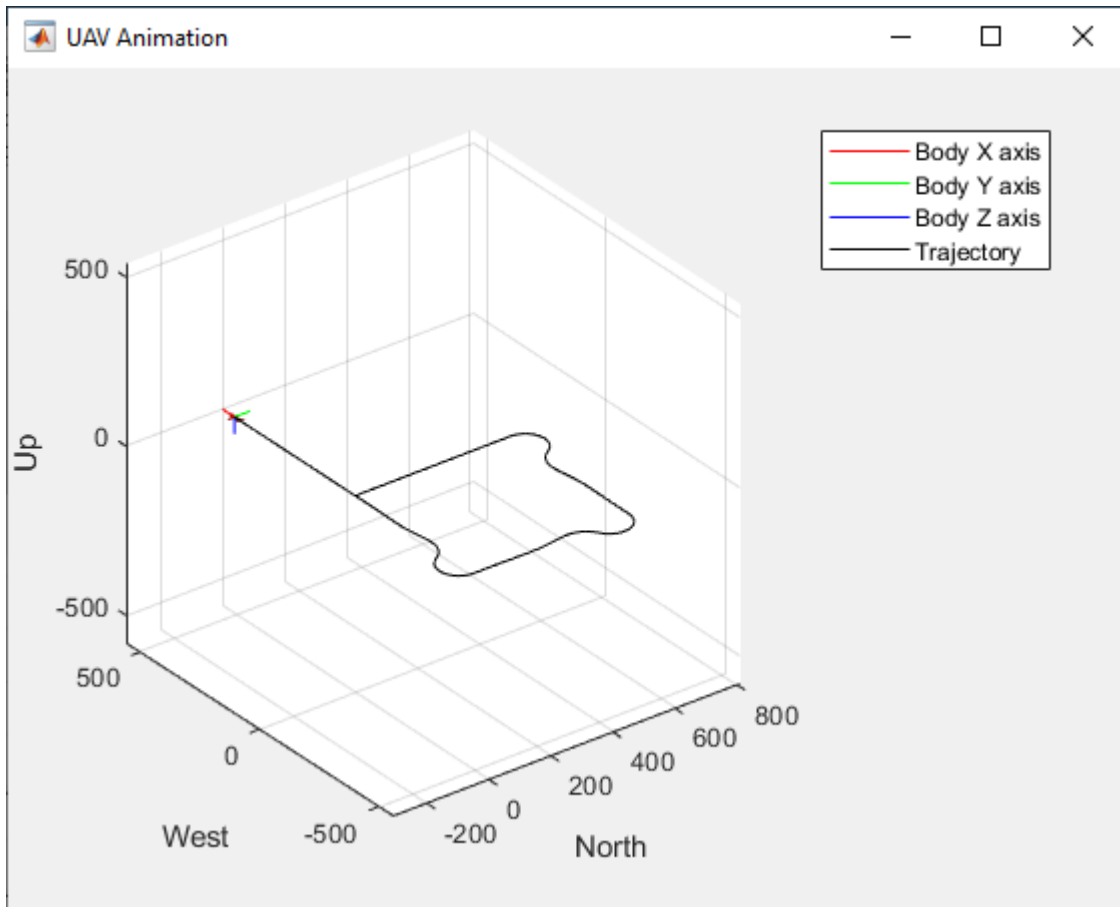


The next figures shows the flight behavior with a small lookahead distance (5) and a fast heading control (3.9). Notice the UAV follows a very curvy path between the waypoints.



The next figure shows the flight behavior with a large lookahead distance and slow heading control.





### Summary

This example tunes UAV flight controller by manually iterating through multiple sets of control parameters. This process can be extended to automatically sweep large set of control parameters to obtain optimal control configurations for customized navigation controllers.

Once the flight behavior satisfies design specification, consider testing the chosen control parameters with high-fidelity models built with Aerospace Blockset or with external flight simulators.

```
% close Simulink models
close_system("uavStepResponse");
close_system("fixedWingPathFollowing");
```



## Approximate High-Fidelity UAV model with UAV Guidance Model block

Simulation models often need different levels of fidelity during different development stages. During the rapid-prototyping stage, we would like to quickly experiment and tune parameters to test different autonomous algorithms. During the production development stage, we would like to validate our algorithms against models of increasing fidelities. In this example, we demonstrate a method to approximate a high-fidelity model with the Guidance Model block and use it to prototype and tune a waypoint following navigation system. See “Tuning Waypoint Follower for Fixed-Wing UAV” on page 1-25. The same navigation system is tested against a high-fidelity model to verify its performance.

The example model uses a high-fidelity unmanned aerial vehicle (UAV) model consisting of a plant model and a mid-level built-in autopilot. This model contains close to a thousand blocks and it is quite complicated to work with. As a first step in the development process, we created a variant system that can switch between this high-fidelity model and the UAV Guidance Model block. The high-fidelity model is extracted from a File Exchange entry, Simulink Drone Reference Application.

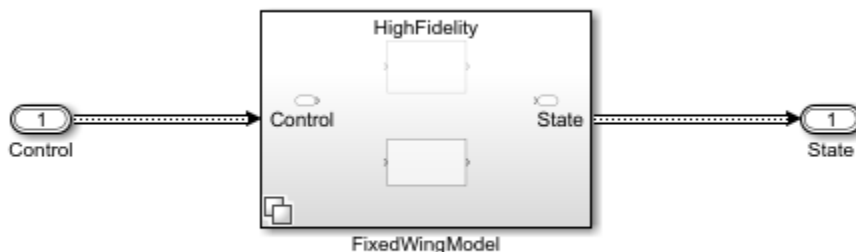
### UAV model of different fidelity

```
uavModel = 'FixedWingModel.slx';
open_system(uavModel);
```

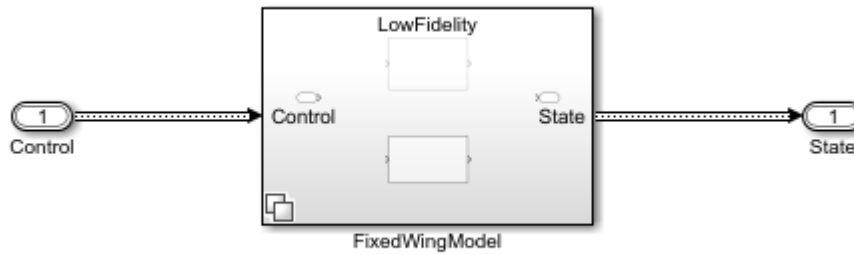
You can switch between the low and high-fidelity models by changing a MATLAB® variable value stored in the data dictionary associated with this model.

```
plantDataDictionary = Simulink.data.dictionary.open('pathFollowingData.slidd');
plantDataSet = getSection(plantDataDictionary, 'Design Data');
```

```
% Switch to high-fidelity model
assignin(plantDataSet, 'useHighFidelity', 1);
```



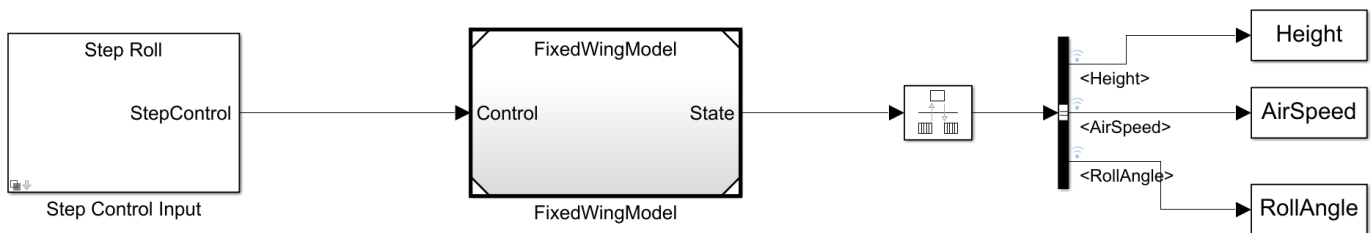
```
% Switch to low-fidelity model
assignin(plantDataSet, 'useHighFidelity', 0);
```



### Approximate high-fidelity fixed-wing model with low-fidelity guidance model

To approximate the high-fidelity model with the UAV Guidance Model block, create step control signals to feed into the model and observe the step response to RollAngle, Height, and AirSpeed commands.

```
stepModel = 'stepResponse';
open_system(stepModel)
```



First, command a change in roll angle.

```
controlBlock = get_param('stepResponse/Step Control Input','Object');
controlBlock.StepControl = 'RollAngle Step Control';
```

```
assignin(plantDataSet, 'useHighFidelity', 1);
```

```
sim(stepModel);
```

```
### Starting serial model reference simulation build
### Successfully updated the model reference simulation target for: PlantModel
### Successfully updated the model reference simulation target for: FixedWingModel
```

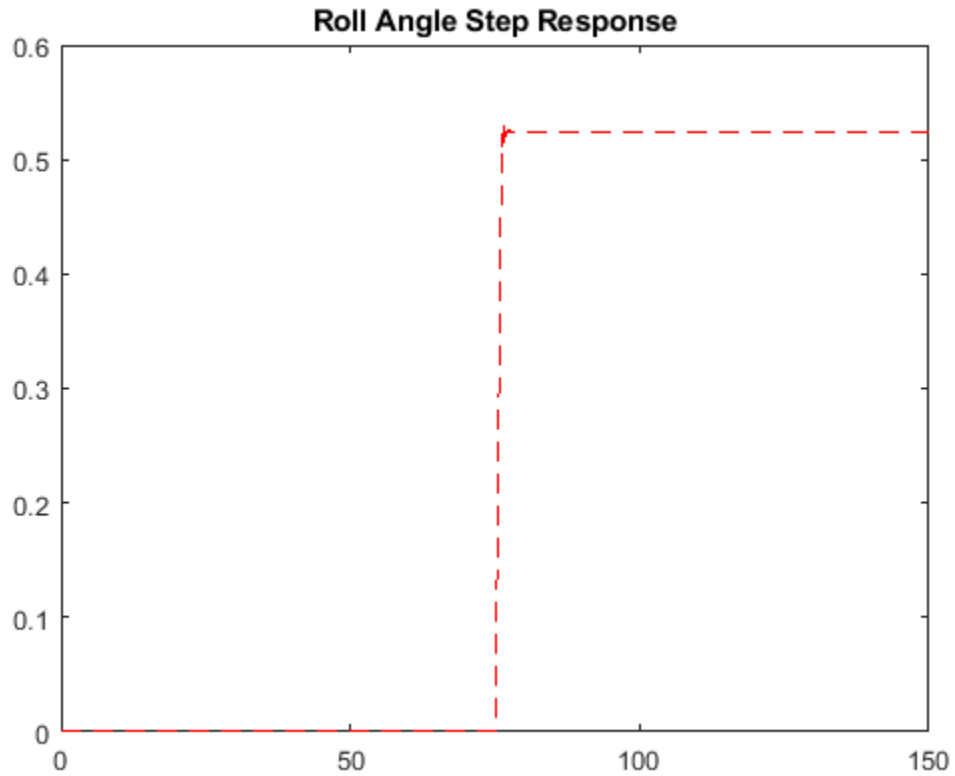
Build Summary

Simulation targets built:

Model	Action	Rebuild Reason
PlantModel	Code generated and compiled	PlantModel_msf.mexw64 does not exist.
FixedWingModel	Code generated and compiled	FixedWingModel_msf.mexw64 does not exist.

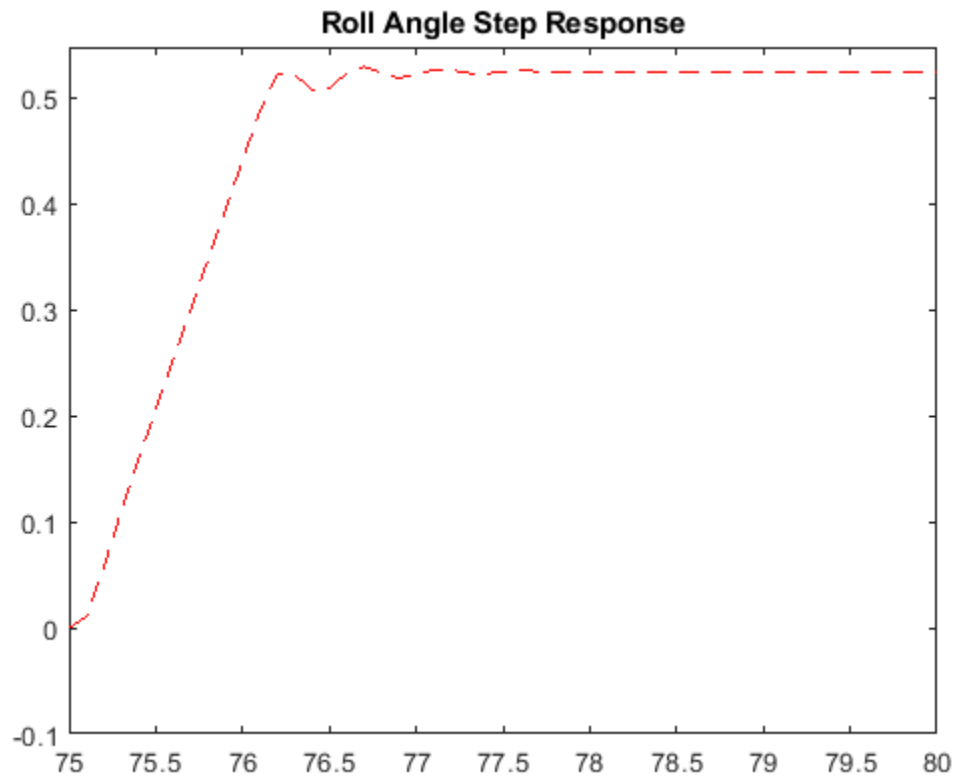
2 of 2 models built (0 models already up to date)  
Build duration: 0h 2m 51.221s

```
highFidelityRollAngle = RollAngle.Data(:);  
highFidelityTime = RollAngle.Time;  
  
figure()  
plot(highFidelityTime, highFidelityRollAngle, '--r');  
title('Roll Angle Step Response')
```



Zooming into the simulation result above, you see the characteristics of the roll angle controller built into the high-fidelity model. The settling time for the roll angle is close to 2.5 seconds.

```
xlim([75 80])  
ylim([-0.1 0.548])
```



For a second-order PD controller, to achieve this settling time with a critically damped system, the following gains should be used to configure the UAV Guidance Model block inside the low-fidelity variant of the UAV model. For this example, the **UAV Guidance Model** block is simulated using code generation to increase speed for multiple runs. See the block parameters.

```
zeta = 1.0; % critically damped
ts = 2.5; % 2 percent settling time
wn = 5.8335/(ts*zeta);
newRollPD = [wn^2 2*zeta*wn];
```

Set the new gains and simulate the step response for the low-fidelity model. Compare it to the original response.

```
load_system(uavModel)
set_param('FixedWingModel/FixedWingModel/LowFidelity/Fixed Wing UAV Guidance Model',...
    'PDRollFixedWing',strcat('[' ,num2str(newRollPD), ']'))
save_system(uavModel)

assignin(plantDataSet, 'useHighFidelity', 0);

sim(stepModel);

### Starting serial model reference simulation build
### Successfully updated the model reference simulation target for: FixedWingModel

Build Summary
```

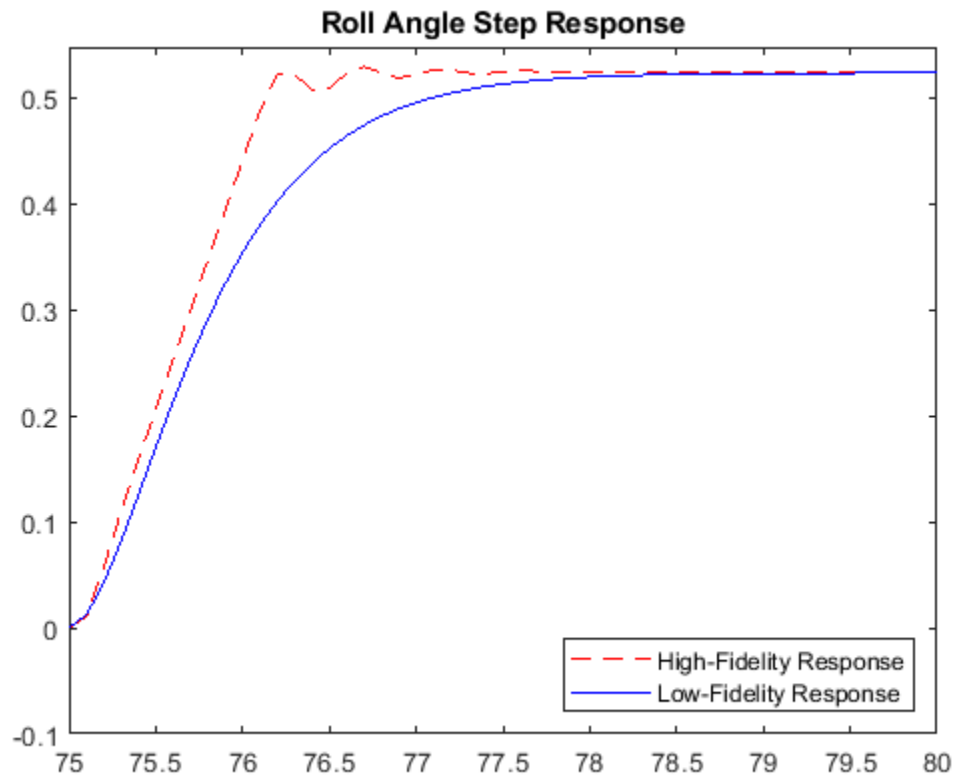
Simulation targets built:

Model	Action	Rebuild Reason
FixedWingModel	Code generated and compiled	Model or library FixedWingModel has changed.

1 of 1 models built (0 models already up to date)  
Build duration: 0h 0m 38.007s

```
lowFidelityRollAngle = RollAngle.Data(:);
lowFidelityTime = RollAngle.Time;

hold on;
plot(lowFidelityTime, lowFidelityRollAngle, '-b');
legend('High-Fidelity Response', 'Low-Fidelity Response', 'Location', 'southeast');
```



The low-fidelity model achieves a similar step response. Similarly, we can tune the other two control channels: Height and AirSpeed. More sophisticated methods can be used here to optimize the control gains instead of visual inspection of the control response. Consider using System Identification Toolbox® to perform further analysis of the high-fidelity UAV model behavior.

```
controlBlock.StepControl = 'AirSpeed Step Control';
assignin(plantDataSet, 'useHighFidelity', 0);

sim(stepModel);
```

```
### Starting serial model reference simulation build
### Model reference simulation target for FixedWingModel is up to date.

Build Summary

0 of 1 models built (1 models already up to date)
Build duration: 0h 0m 5.424s

lowFidelityAirSpeed = AirSpeed.Data(:);
lowFidelityTime = AirSpeed.Time;

assignin(plantDataSet, 'useHighFidelity', 1);

sim(stepModel);

### Starting serial model reference simulation build
### Model reference simulation target for PlantModel is up to date.
### Successfully updated the model reference simulation target for: FixedWingModel

Build Summary

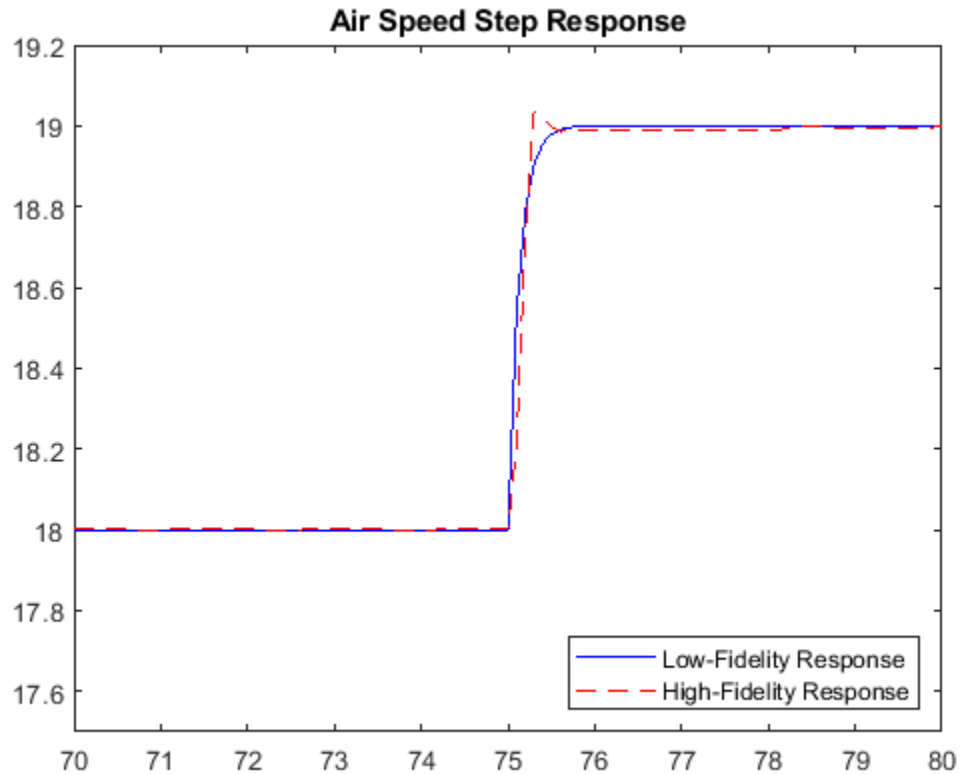
Simulation targets built:

Model          Action          Rebuild Reason
=====
FixedWingModel Code generated and compiled Variant control useHighFidelity == 0 value has changed

1 of 2 models built (1 models already up to date)
Build duration: 0h 1m 3.131s

highFidelityAirSpeed = AirSpeed.Data(:);
highFidelityTime = AirSpeed.Time;

figure()
plot(lowFidelityTime, lowFidelityAirSpeed, '-b');
hold on;
plot(highFidelityTime, highFidelityAirSpeed, '--r');
legend('Low-Fidelity Response', 'High-Fidelity Response', 'Location', 'southeast');
title('Air Speed Step Response')
xlim([70 80])
ylim([17.5 19.2])
```



```
controlBlock.StepControl = 'Height Step Control';
assignin(plantDataSet, 'useHighFidelity', 0);
```

```
sim(stepModel);
```

```
### Starting serial model reference simulation build
### Successfully updated the model reference simulation target for: FixedWingModel
```

Build Summary

Simulation targets built:

Model	Action	Rebuild Reason
FixedWingModel	Code generated and compiled	Variant control useHighFidelity == 1 value has changed

=====

1 of 1 models built (0 models already up to date)

Build duration: 0h 0m 38.647s

```
lowFidelityHeight = Height.Data(:);
lowFidelityTime = Height.Time;
```

```
assignin(plantDataSet, 'useHighFidelity', 1);
```

```
sim(stepModel);
```

```
### Starting serial model reference simulation build
### Model reference simulation target for PlantModel is up to date.
```

```
### Successfully updated the model reference simulation target for: FixedWingModel
```

```
Build Summary
```

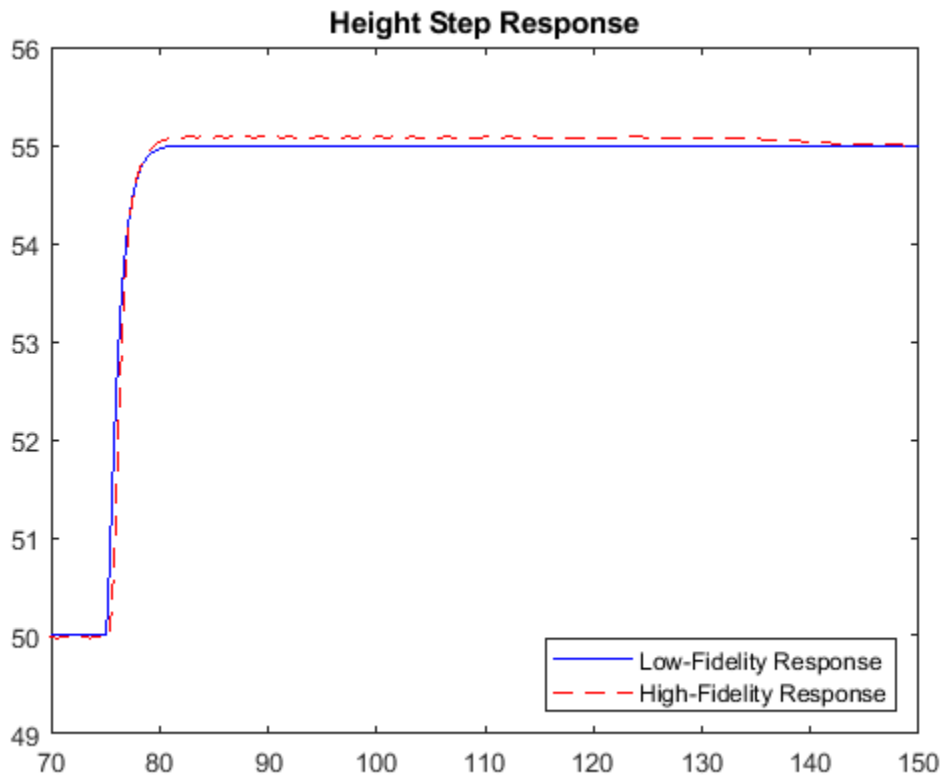
```
Simulation targets built:
```

Model	Action	Rebuild Reason
FixedWingModel	Code generated and compiled	Variant control useHighFidelity == 0 value has changed

```
1 of 2 models built (1 models already up to date)
Build duration: 0h 0m 56.413s
```

```
highFidelityHeight = Height.Data(:);
highFidelityTime = Height.Time;
```

```
figure()
plot(lowFidelityTime, lowFidelityHeight, '-b');
hold on;
plot(highFidelityTime, highFidelityHeight, '--r');
legend('Low-Fidelity Response', 'High-Fidelity Response', 'Location', 'southeast');
title('Height Step Response')
xlim([70 150])
ylim([49 56])
```





## Test navigation algorithm with low-fidelity model

Now that we have approximated the high-fidelity model with the **UAV Guidance Model** block, we can try to replace it with the UAV Guidance Model block in the “Tuning Waypoint Follower for Fixed-Wing UAV” on page 1-25 example. Test the effect of the lookahead distance and heading control gains against these models of different fidelities.

```

navigationModel = 'pathFollowing';
open_system(navigationModel);

assignin(plantDataSet,'useHighFidelity',0);

sim(navigationModel);

### Starting serial model reference simulation build
### Successfully updated the model reference simulation target for: FixedWingModel

Build Summary

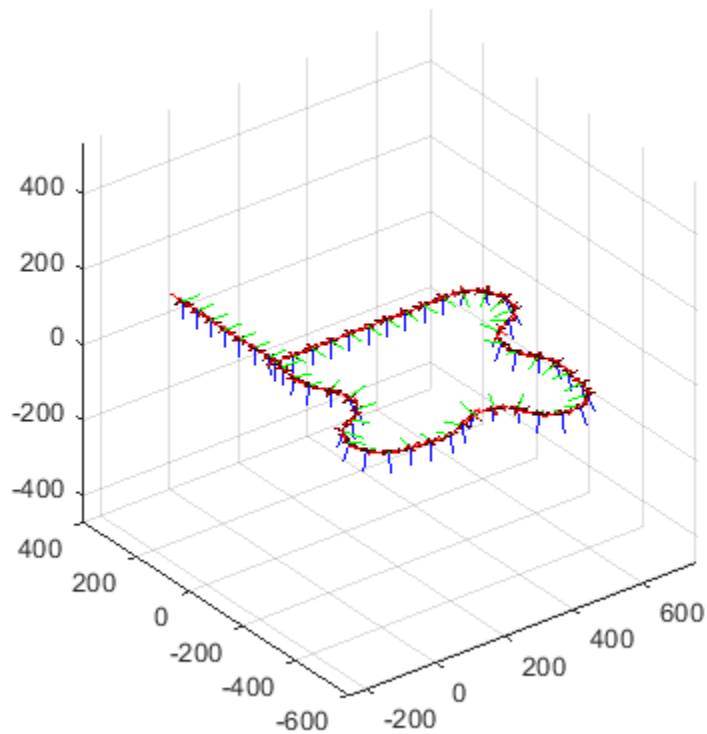
Simulation targets built:

Model          Action          Rebuild Reason
=====
FixedWingModel Code generated and compiled Variant control useHighFidelity == 1 value has changed

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 30.997s

figure
visualizeSimStates(simStates);

```



### Validate with high-fidelity model

```
assignin(plantDataSet, 'useHighFidelity', 1);
```

```
sim(navigationModel);
```

```
### Starting serial model reference simulation build
### Model reference simulation target for PlantModel is up to date.
### Successfully updated the model reference simulation target for: FixedWingModel
```

Build Summary

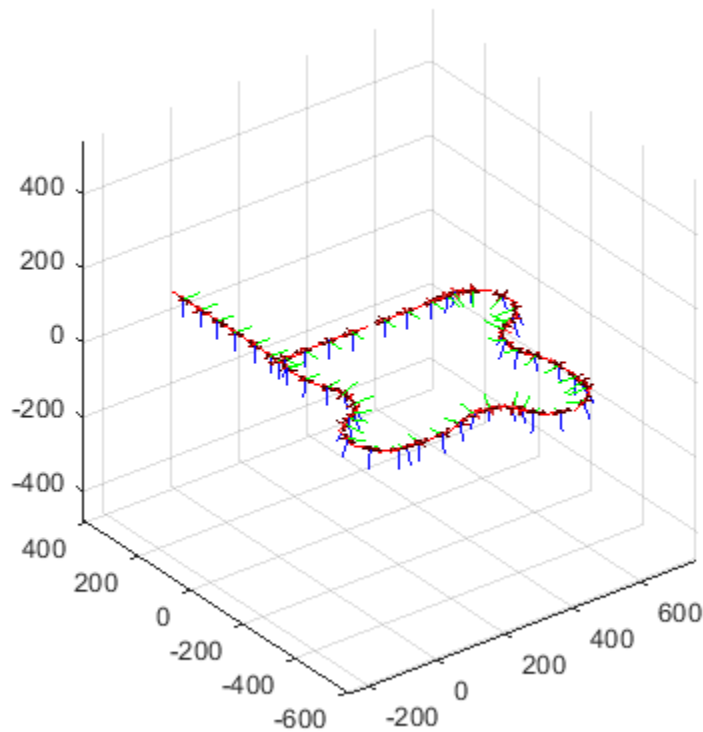
Simulation targets built:

Model	Action	Rebuild Reason
FixedWingModel	Code generated and compiled	Variant control useHighFidelity == 0 value has changed

1 of 2 models built (1 models already up to date)

Build duration: 0h 0m 52.308s

```
figure
visualizeSimStates(simStates);
```



## Conclusion

This example shows how we can approximate a high-fidelity model with a low-fidelity abstraction of a fixed-wing UAV. The opposite approach can be used as well to help with choosing autopilot control gains for the high-fidelity model. You can first decide acceptable characteristics of an autopilot control response by simulating a low-fidelity model in different test scenarios and then tune the high-fidelity model autopilot accordingly.

```
discardChanges(plantDataDictionary);  
clear plantDataSet  
clear plantDataDictionary  
close_system(uavModel, 0);  
close_system(stepModel, 0);  
close_system(navigationModel, 0);
```

## See Also

UAV Guidance Model | fixedwing | multirotor

## More About

- “Tuning Waypoint Follower for Fixed-Wing UAV” on page 1-25
- “Simulink Bus Signals” (Simulink)

## Motion Planning with RRT for Fixed-Wing UAV

This example demonstrates motion planning of a fixed-wing unmanned aerial vehicle (UAV) using the rapidly exploring random tree (RRT) algorithm given a start and goal pose on a 3-D map. A fixed-wing UAV is nonholonomic in nature, and must obey aerodynamic constraints like maximum roll angle, flight path angle, and airspeed when moving between waypoints.

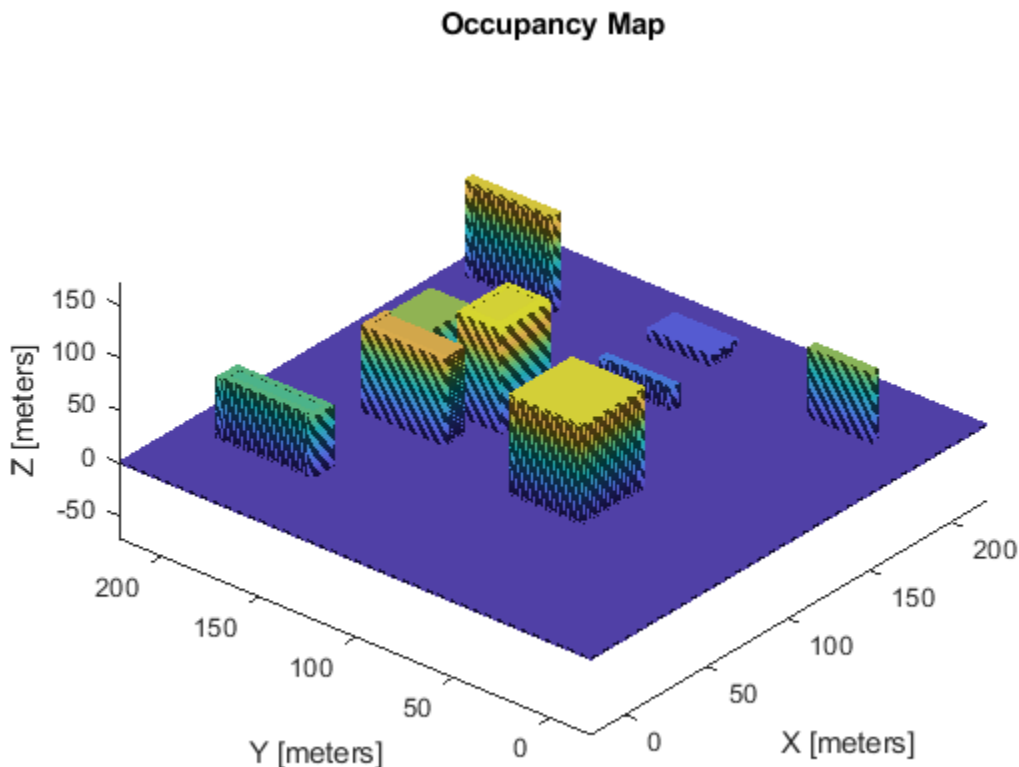
In this example you will set up a 3-D map, provide the start pose and goal pose, plan a path with RRT using 3-D Dubins motion primitives, smooth the obtained path, and simulate the flight of the UAV.

```
% Set RNG seed for repeatable result
rng(1, "twister")
```

### Load Map

Load the 3-D occupancy map `uavMapCityBlock.mat`, which contains a set of pregenerated obstacles, into the workspace. Inflate the map uniformly by 1 m to increase path safety and account for the wingspan of the fixed-wing UAV. The occupancy map is in an ENU (East-North-Up) frame.

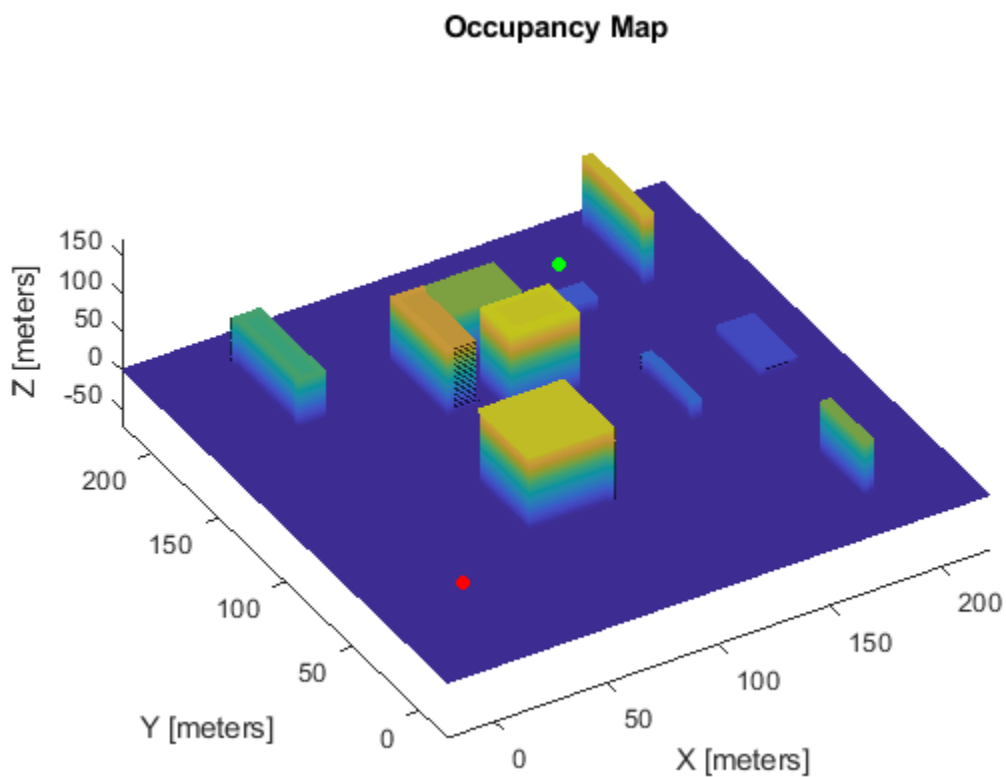
```
mapData = load("uavMapCityBlock.mat","omap");
omap = mapData.omap;
% Consider unknown spaces to be unoccupied
omap.FreeThreshold = omap.OccupiedThreshold;
inflate(omap,1)
figure("Name","CityBlock")
show(omap)
```



## Set Start Pose and Goal Pose

Using the map for reference, select an unoccupied start pose and goal pose.

```
startPose = [12 22 25 pi/2];
goalPose = [150 180 35 pi/2];
figure("Name", "StartAndGoal")
hMap = show(omap);
hold on
scatter3(hMap, startPose(1), startPose(2), startPose(3), 30, "red", "filled")
scatter3(hMap, goalPose(1), goalPose(2), goalPose(3), 30, "green", "filled")
hold off
view([-31 63])
```



## Plan a Path with RRT Using 3-D Dubins Motion Primitives

RRT is a tree-based motion planner that builds a search tree incrementally from random samples of a given state space. The tree eventually spans the search space and connects the start state and the goal state. Connect the two states using a `uavDubinsConnection` object that satisfies aerodynamic constraints. Use the `validatorOccupancyMap3D` object for collision checking between the fixed-wing UAV and the environment.

### Define the State Space Object

This example provides a predefined state space, `ExampleHelperUavStateSpace`, for path planning. The state space is defined as `[x y z headingAngle]`, where `[x y z]` specifies the position of the UAV and `headingAngle` specifies the heading angle in radians. The example uses a

`uavDubinsConnection` object as the kinematic model for the UAV, which is constrained by maximum roll angle, airspeed, and flight path angle. Create the state space object by specifying the maximum roll angle, airspeed, and flight path angle limits properties of the UAV as name-value pairs. Use the "Bounds" name-value pair argument to specify the position and orientation boundaries of the UAV as a 4-by-2 matrix, where the first three rows represent the x-, y-, and z-axis boundaries inside the 3-D occupancy map and the last row represents the heading angle in the range  $[-\pi, \pi]$  radians.

```
ss = ExampleHelperUAVStateSpace("MaxRollAngle",pi/6,...
                                "AirSpeed",6,...
                                "FlightPathAngleLimit",[-0.1 0.1],...
                                "Bounds",[-20 220; -20 220; 10 100; -pi pi]);
```

Set the threshold bounds of the workspace based on the target goal pose. This threshold dictates how large the target workspace goal region around the goal pose is, which is used for bias sampling of the workspace goal region approach.

```
threshold = [(goalPose-0.5)' (goalPose+0.5)'; -pi pi];
```

Use the `setWorkspaceGoalRegion` function to update the goal pose and the region around it.

```
setWorkspaceGoalRegion(ss,goalPose,threshold)
```

### Define the State Validator Object

The `validatorOccupancyMap3D` object determines that a state is invalid if the xyz-location is occupied on the map. A motion between two states is valid only if all intermediate states are valid, which means the UAV does not pass through any occupied location on the map. Create a `validatorOccupancyMap3D` object by specifying the state space object and the inflated map. Then set the validation distance, in meters, for interpolating between states.

```
sv = validatorOccupancyMap3D(ss,"Map",omap);
sv.ValidationDistance = 0.1;
```

### Set Up the RRT Path Planner

Create a `plannerRRT` object by specifying the state space and state validator as inputs. Set the `MaxConnectionDistance`, `GoalBias`, and `MaxIterations` properties of the planner object, and then specify a custom goal function. This goal function determines that a path has reached the goal if the Euclidean distance to the target is below a threshold of 5 m.

```
planner = plannerRRT(ss,sv);
planner.MaxConnectionDistance = 50;
planner.GoalBias = 0.10;
planner.MaxIterations = 400;
planner.GoalReachedFcn = @(~,x,y)(norm(x(1:3)-y(1:3)) < 5);
```

### Execute Path Planning

Perform RRT-based path planning in 3-D space. The planner finds a path that is collision-free and suitable for fixed-wing flight.

```
[pthObj,solnInfo] = plan(planner,startPose,goalPose);
```

### Simulate a UAV Following the Planned Path

Visualize the planned path. Interpolate the planned path based on the UAV Dubins connections. Plot the interpolated states as a green line.

Simulate the UAV flight using the provided helper function, `exampleHelperSimulateUAV`, which requires the waypoints, airspeed, and time to reach the goal (based on airspeed and path length). The helper function uses the `fixedwing` guidance model to simulate the UAV behavior based on control inputs generated from the waypoints. Plot the simulated states as a red line.

Notice that the simulated UAV flight deviates slightly from the planned path because of small control tracking errors. Also, the 3-D Dubins path assumes instantaneous changes in the UAV roll angle, but the actual dynamics have a slower response to roll commands. One way to compensate for this lag is to plan paths with more conservative aerodynamic constraints.

```

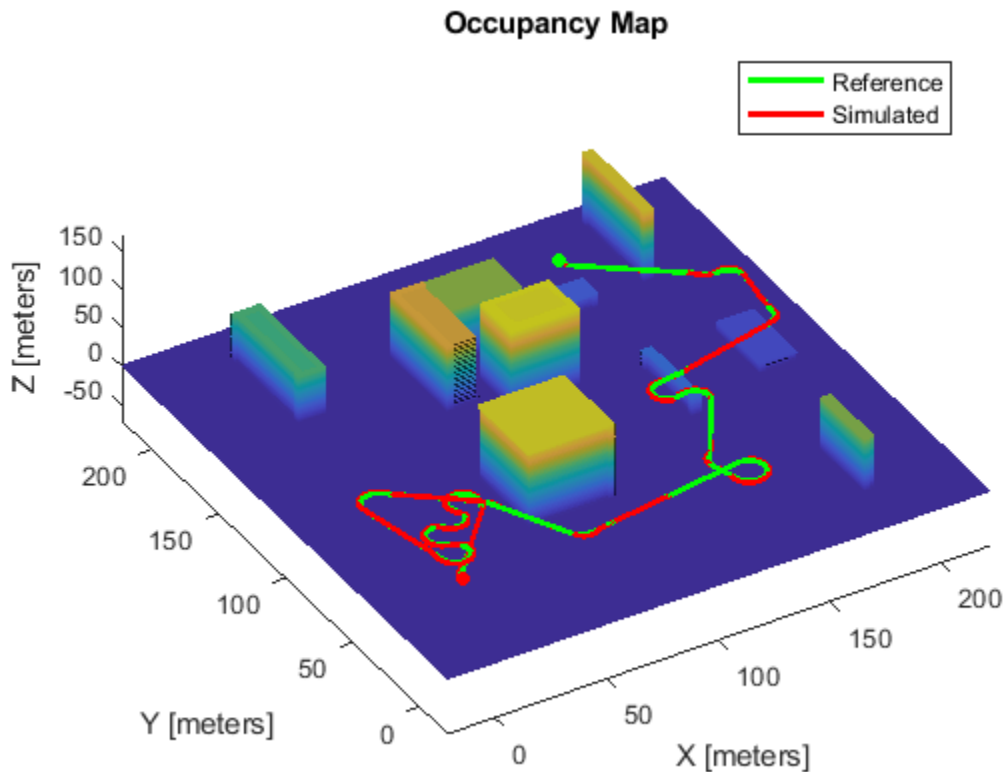
if (solnInfo.IsPathFound)
    figure("Name","OriginalPath")
    % Visualize the 3-D map
    show(omap)
    hold on
    scatter3(startPose(1),startPose(2),startPose(3),30,"red","filled")
    scatter3(goalPose(1),goalPose(2),goalPose(3),30,"green","filled")

    interpolatedPathObj = copy(pthObj);
    interpolate(interpolatedPathObj,1000)

    % Plot the interpolated path based on UAV Dubins connections
    hReference = plot3(interpolatedPathObj.States(:,1), ...
        interpolatedPathObj.States(:,2), ...
        interpolatedPathObj.States(:,3), ...
        "LineWidth",2,"Color","g");

    % Plot simulated UAV trajectory based on fixed-wing guidance model
    % Compute total time of flight and add a buffer
    timeToReachGoal = 1.05*pathLength(pthObj)/ss.AirSpeed;
    waypoints = interpolatedPathObj.States;
    [xENU,yENU,zENU] = exampleHelperSimulateUAV(waypoints,ss.AirSpeed,timeToReachGoal);
    hSimulated = plot3(xENU,yENU,zENU,"LineWidth",2,"Color","r");
    legend([hReference,hSimulated],"Reference","Simulated","Location","best")
    hold off
    view([-31 63])
end

```



### Smooth Dubins Path and Simulate UAV Trajectory

The original planned path makes some unnecessary turns while navigating towards the goal. Simplify the 3-D Dubins path by using the path smoothing algorithm provided with the example, `exampleHelperUAVPathSmoothing`. This function removes intermediate 3-D Dubins poses based on an iterative strategy. For more information on the smoothing strategy, see [1 on page 1-0 ]. The smoothing function connects non-sequential 3-D Dubins poses with each other as long as doing so does not result in a collision. The smooth paths generated by this process improve tracking characteristics for the fixed-wing simulation model. Simulate the fixed-wing UAV model with these new, smoothed waypoints.

```
if (solnInfo.IsPathFound)
    smoothWaypointsObj = exampleHelperUAVPathSmoothing(ss,sv,pthObj);

    figure("Name","SmoothedPath")
    % Plot the 3-D map
    show(omap)
    hold on
    scatter3(startPose(1),startPose(2),startPose(3),30,"red","filled")
    scatter3(goalPose(1),goalPose(2),goalPose(3),30,"green","filled")

    interpolatedSmoothWaypoints = copy(smoothWaypointsObj);
    interpolate(interpolatedSmoothWaypoints,1000)

    % Plot smoothed path based on UAV Dubins connections
    hReference = plot3(interpolatedSmoothWaypoints.States(:,1), ...
        interpolatedSmoothWaypoints.States(:,2), ...
```



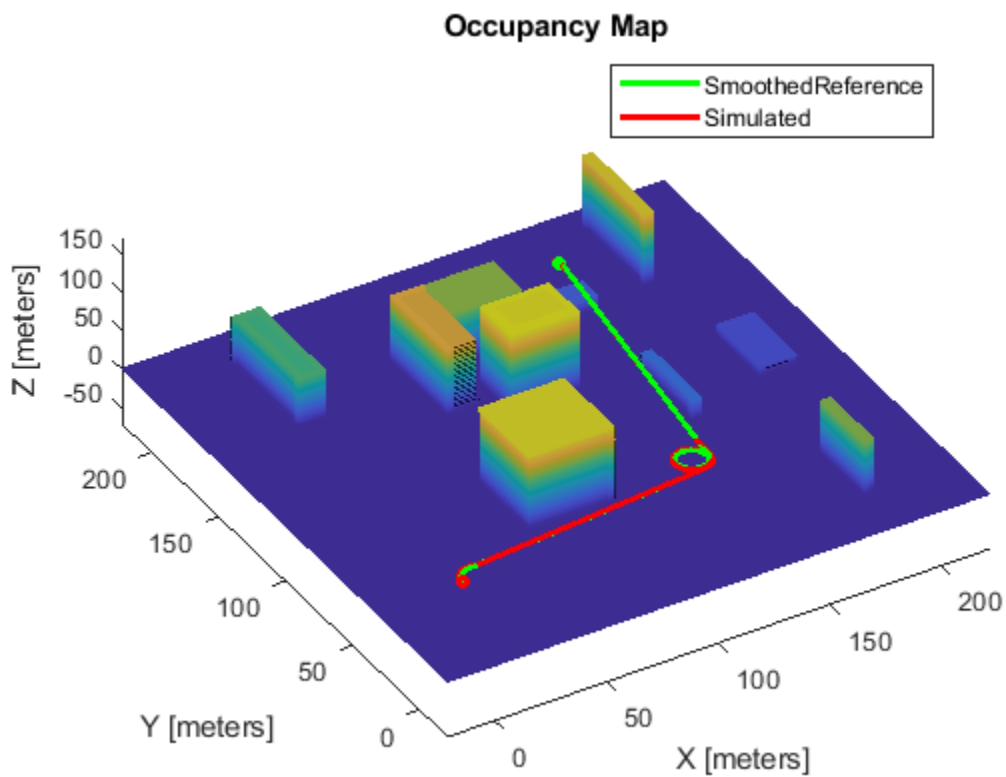
```

interpolatedSmoothWaypoints.States(:,3), ...
"LineWidth",2,"Color","g");

% Plot simulated flight path based on fixed-wing guidance model
waypoints = interpolatedSmoothWaypoints.States;
timeToReachGoal = 1.05*pathLength(smoothWaypointsObj)/ss.AirSpeed;
[xENU,yENU,zENU] = exampleHelperSimulateUAV(waypoints,ss.AirSpeed,timeToReachGoal);
hSimulated = plot3(xENU,yENU,zENU,"LineWidth",2,"Color","r");

legend([hReference,hSimulated],"SmoothedReference","Simulated","Location","best")
hold off
view([-31 63])
end

```



The smoothed path is much shorter and shows improved tracking overall.

## References

- [1] Beard, Randal W., and Timothy W. McLain. *Small Unmanned Aircraft: Theory and Practice*. Princeton, N.J: Princeton University Press, 2012.
- [2] Hornung, Armin, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees." *Autonomous Robots* 34, no. 3 (April 2013): 189-206. <https://doi.org/10.1007/s10514-012-9321-0>.

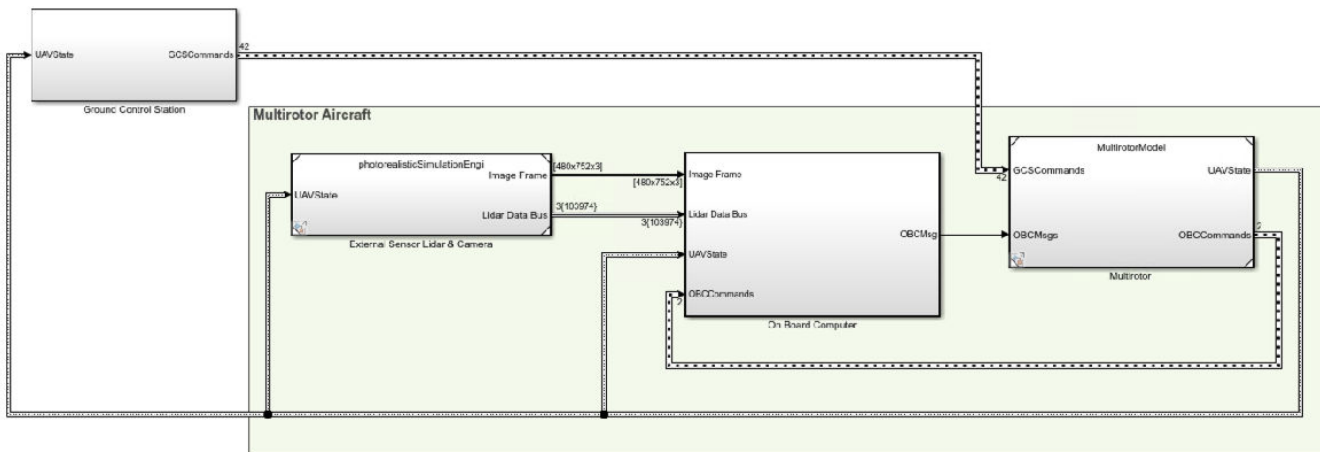
## UAV Package Delivery

This example shows through incremental design iterations how to implement a small multicopter simulation to takeoff, fly, and land at a different location in a city environment.

### Open the Project

To get started, launch the project from the command line.

```
prj = openProject('uavPackageDelivery.prj');
```



### Model Architecture and Conventions

The top model consists of the following subsystems and model references:

- 1 **Ground Control Station:** Used to control and monitor the aircraft while in-flight.
- 2 **External Sensors - Lidar & Camera:** Used to connect to previously-designed scenario or a Photorealistic simulation environment. These produce Lidar readings from the environment as the aircraft flies through it.
- 3 **On Board Computer:** Used to implement algorithms meant to run in an on-board computer independent from the Autopilot
- 4 **Multicopter:** Includes a low-fidelity and mid-fidelity multicopter mode, a flight controller including its guidance logic.

The model's design data is contained in a Simulink data dictionary in the **data** folder (uavPackageDeliveryDataDict.slidd). Additionally, the model uses "Variant Subsystems" (Simulink) to manage different configurations of the model. Variables placed in the base workspace configure these variants without the need to modify the data dictionary.

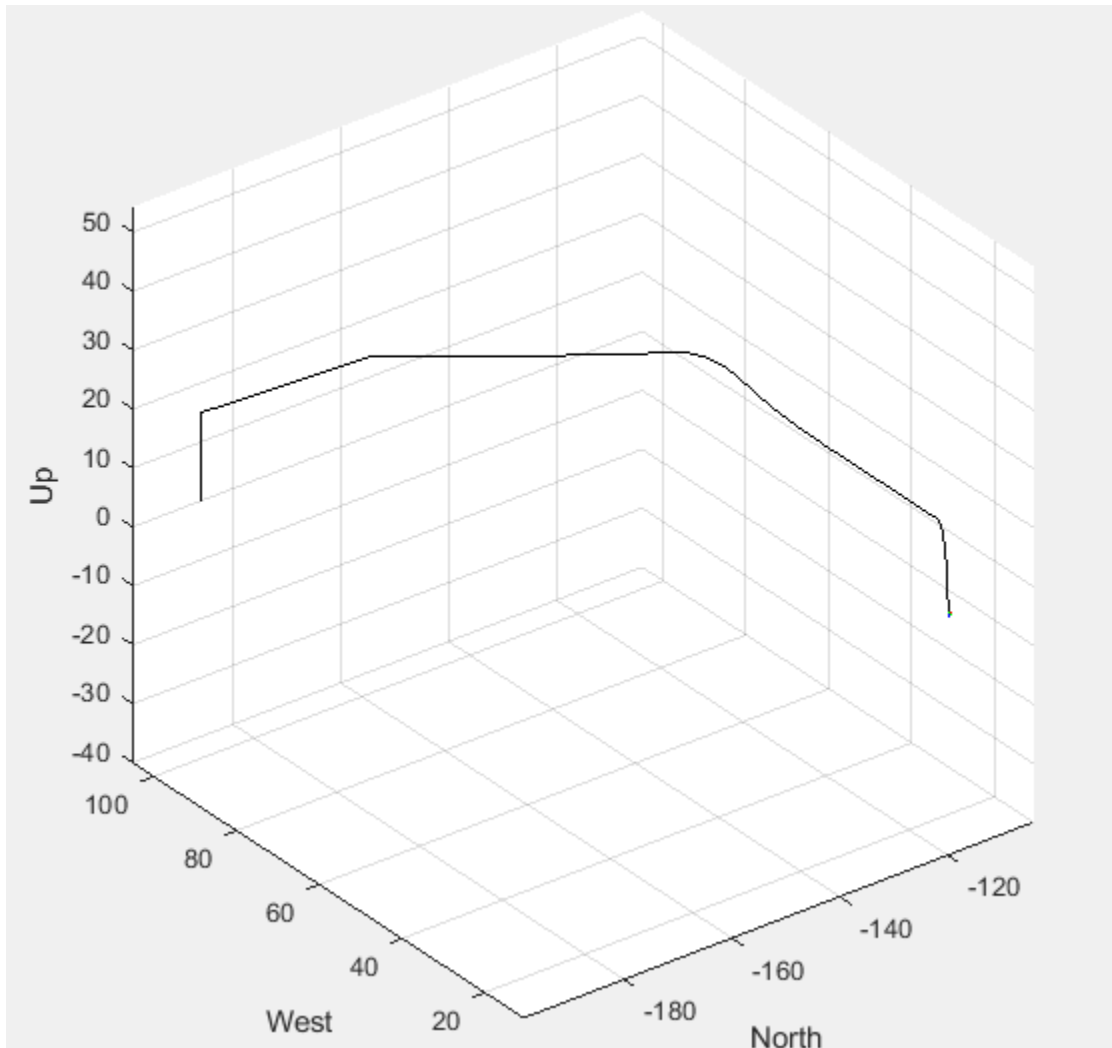
### Following Example Steps

Use the **Project Shortcuts** to step through the example. Each shortcut sets up the required variables for the project.

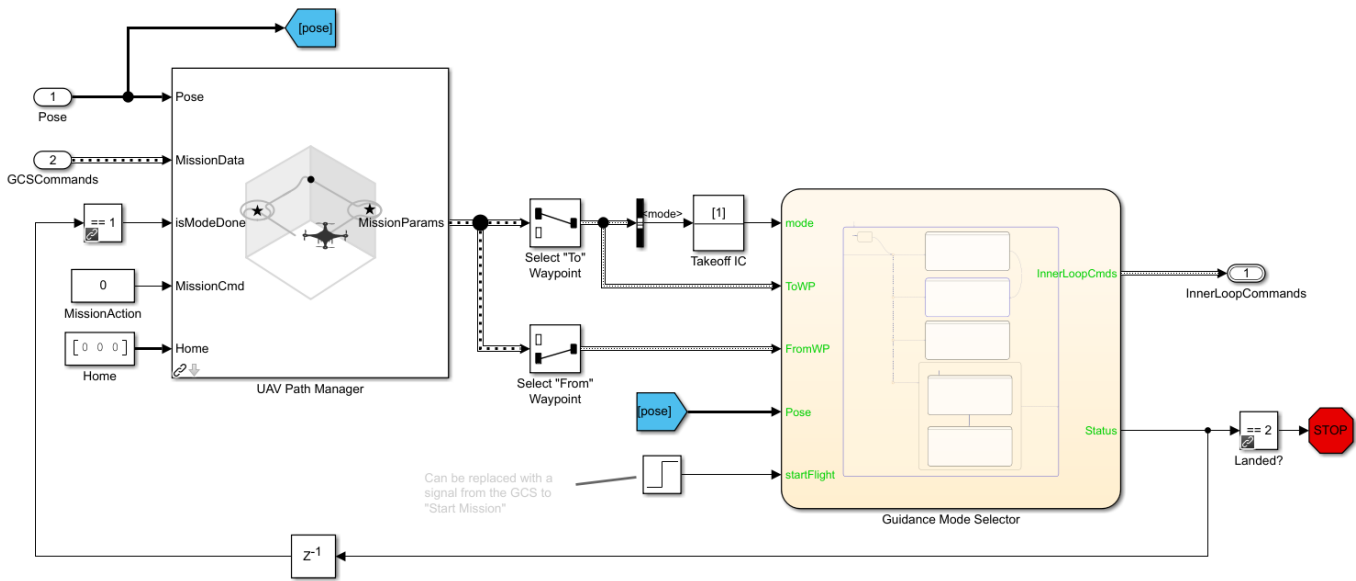
PLOTS	APPS	PROJECT	PROJECT SHORTCUTS				
1	2	3	4	5	6	7	8

## 1. Getting Started

Click the **Getting Started** project shortcut, which sets up the model for a four-waypoint mission using a low-fidelity multirotor plant model. **Run** the `uavPackageDelivery` model, which shows the multirotor takeoff, fly, and land in a 3-D plot.



The model uses the **UAV Path Manager** block to determine which is the active waypoint throughout the flight. The active waypoint is passed into the **Guidance Mode Selector** Stateflow™ chart to generate the necessary inner loop control commands.



## 2. Connecting to a GCS

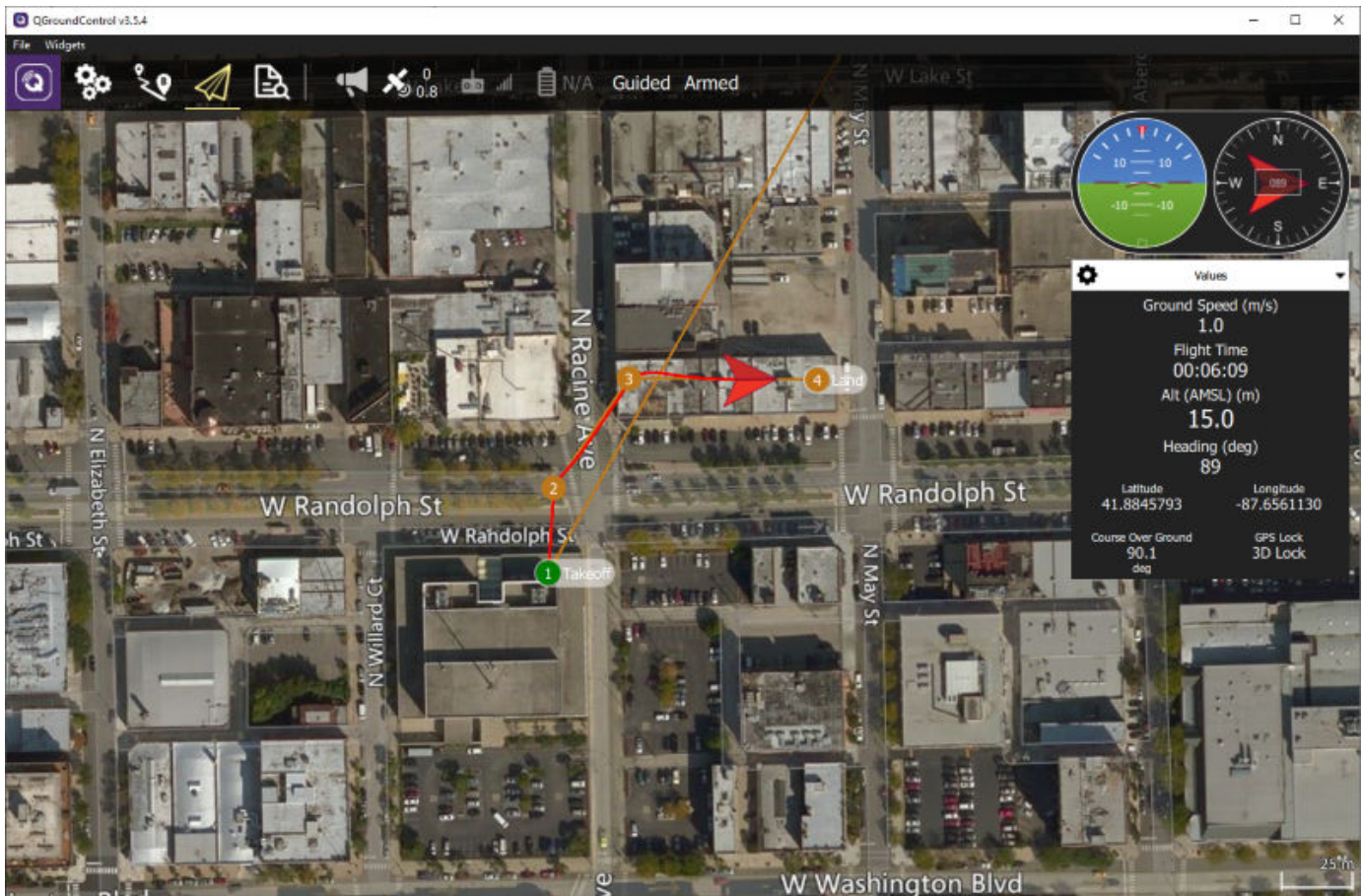
Once you are able to fly a basic mission, you are ready to integrate your simulation with a Ground Station Software so you can better control the aircraft's mission. For this, you need to download and install QGroundControl Ground Control Station software.

The model uses the UAV Toolbox™ `mavlinkio` to establish a connection between Simulink and QGroundControl. The connection is implemented as a MATLAB System Block located in **uavPackageDelivery/Ground Control Station/Get Flight Mission/QGC/MAVLink Interface**.

To test the connectivity between Simulink and QGroundControl follow these steps:

- 1 Click the **Connecting to a GCS** project shortcut
- 2 Launch QGroundControl
- 3 In QGroundControl, load the mission plan named `shortMission.plan` located in `/utilities/qgc`.
- 4 **Run** the simulation.
- 5 When QGroundControl indicates that it is connected to the system, upload the mission.

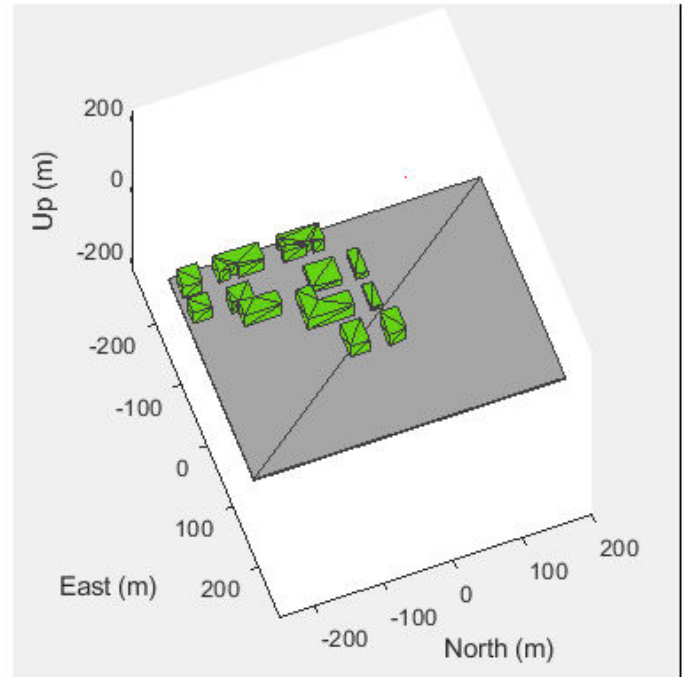
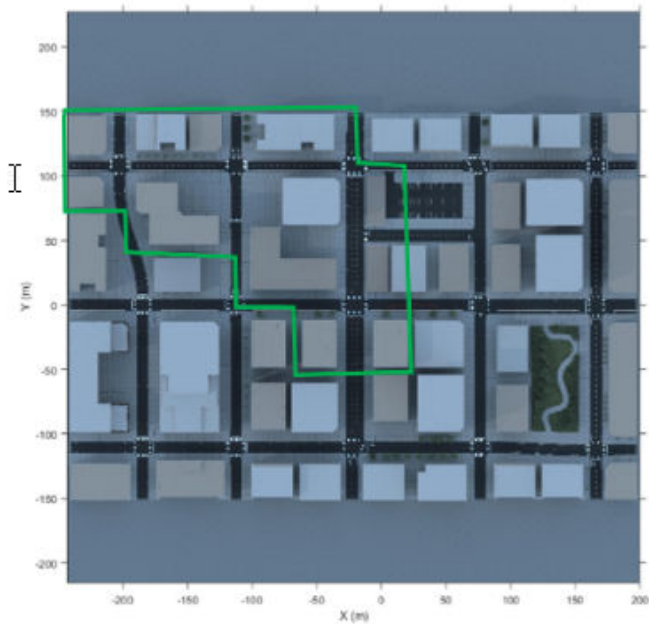
Once the aircraft takes off, you should see the UAV fly its mission as sent by QGC as shown below.



You can modify the mission by adding waypoints or moving those that are already in the mission. Upload the mission and the aircraft should respond to these changes.

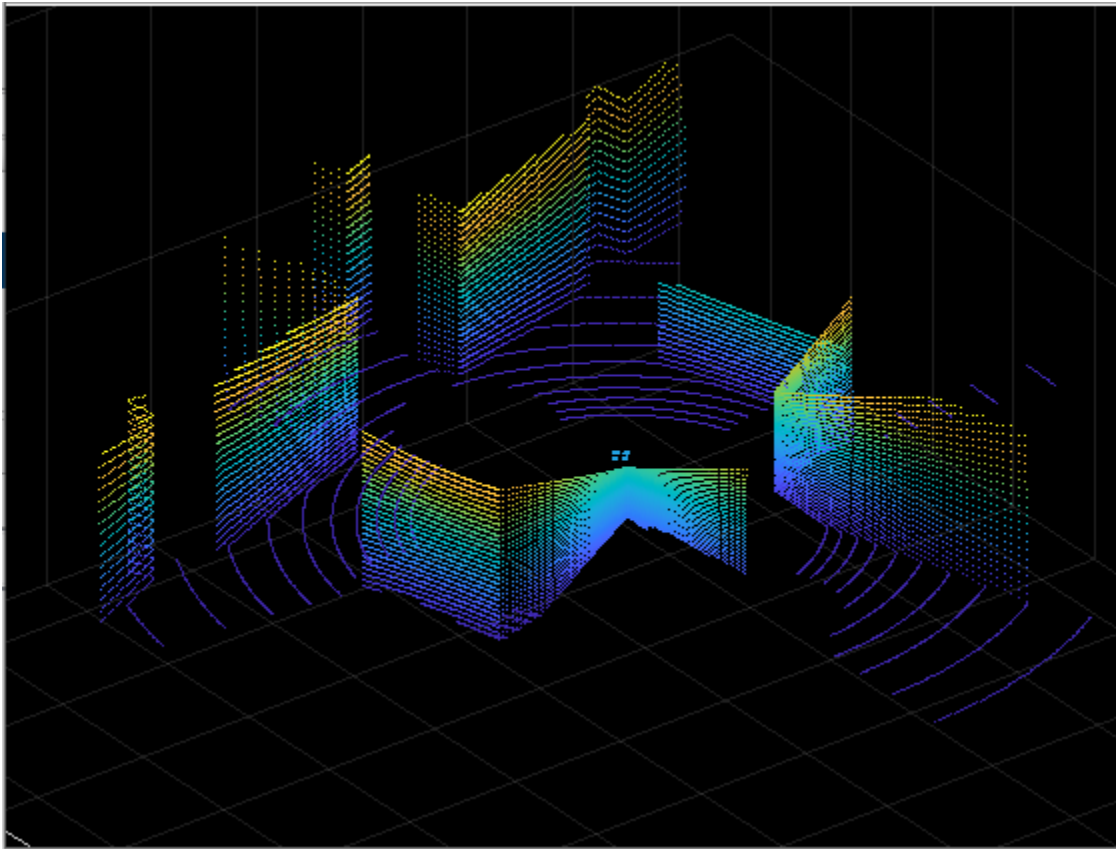
### 3. Setting a Cuboid Scenario

Now that aircraft's model can be flown from a ground control station, consider the environment the aircraft flies in. For this example, a few city blocks are modelled in a cuboid scenario using the `uavScenario` object. The scenario is based on the city block shown in the left figure below.



To safely fly the aircraft in this type of scenario, you need a sensor that provides information about the environment such as a lidar sensor to the model. This example uses a `uavLidarPointCloudGenerator` object added to the UAV scenario with a `uavSensor` object. The lidar sensor model generates readings based on the pose of the sensor and the obstacles in the environment.

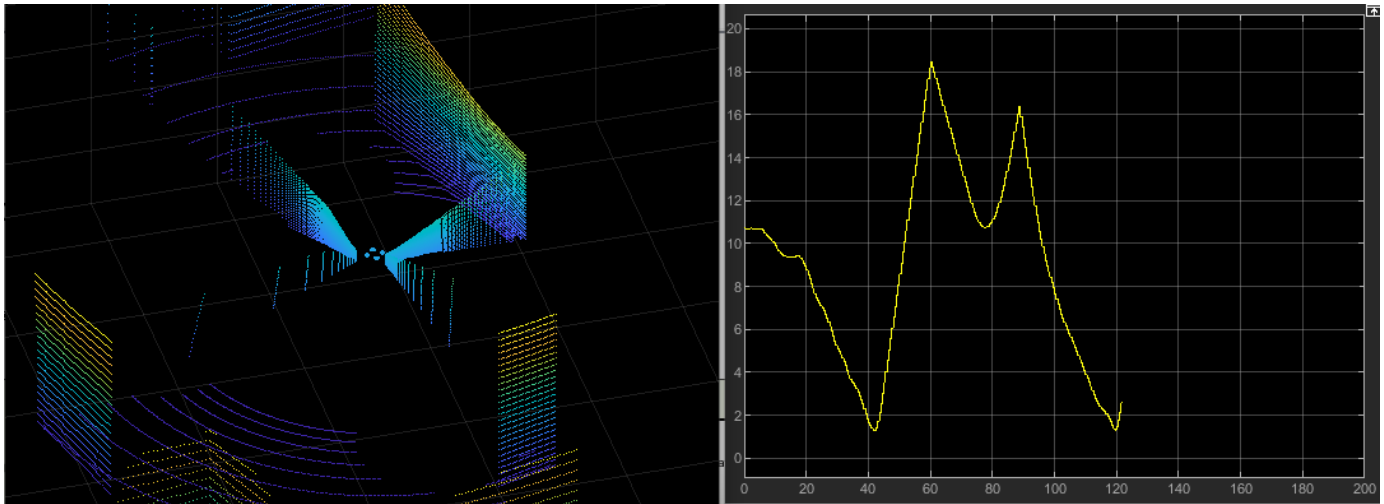
Click the **Setting a Cuboid Scenario** shortcut and **Run** the model. As the model runs, a lidar point cloud image is displayed as the aircraft flies through the cuboid environment:



#### 4. Obstacle Avoidance

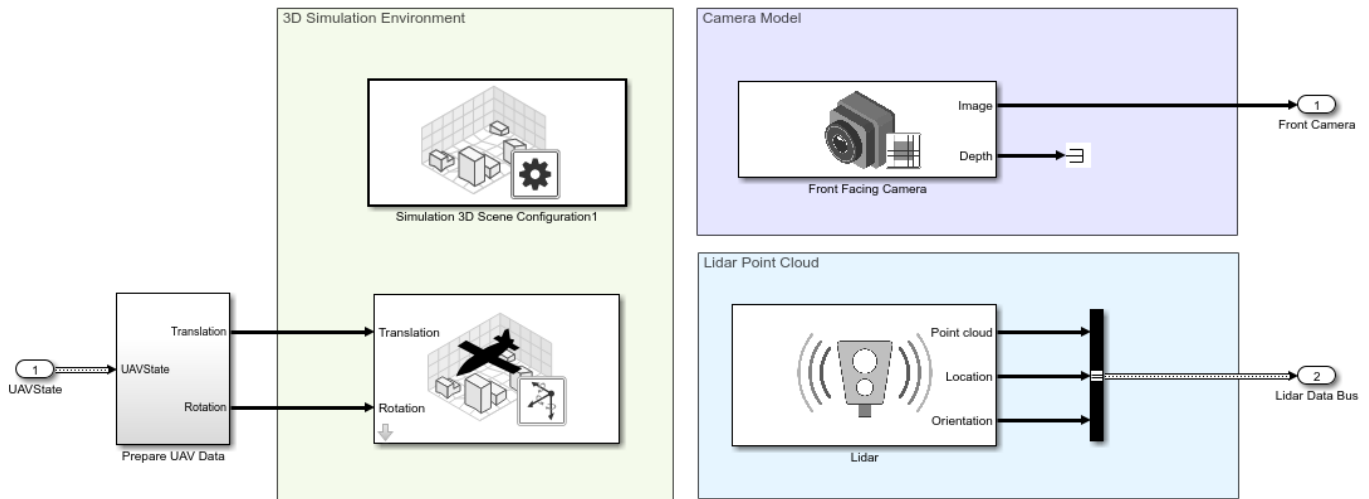
To avoid obstacles in the environment, the model must use the available sensor data as the UAV flies the mission in the environment. To modify the model configuration, click the **Obstacle Avoidance** shortcut. A scope appears that shows the closest point to a building in the cuboid environment.

**Run** the model. As the model runs, the aircraft attempts to fly in a straight path between buildings to a drop site and avoids obstacles along the way. Notice the change in distance to obstacles over time.



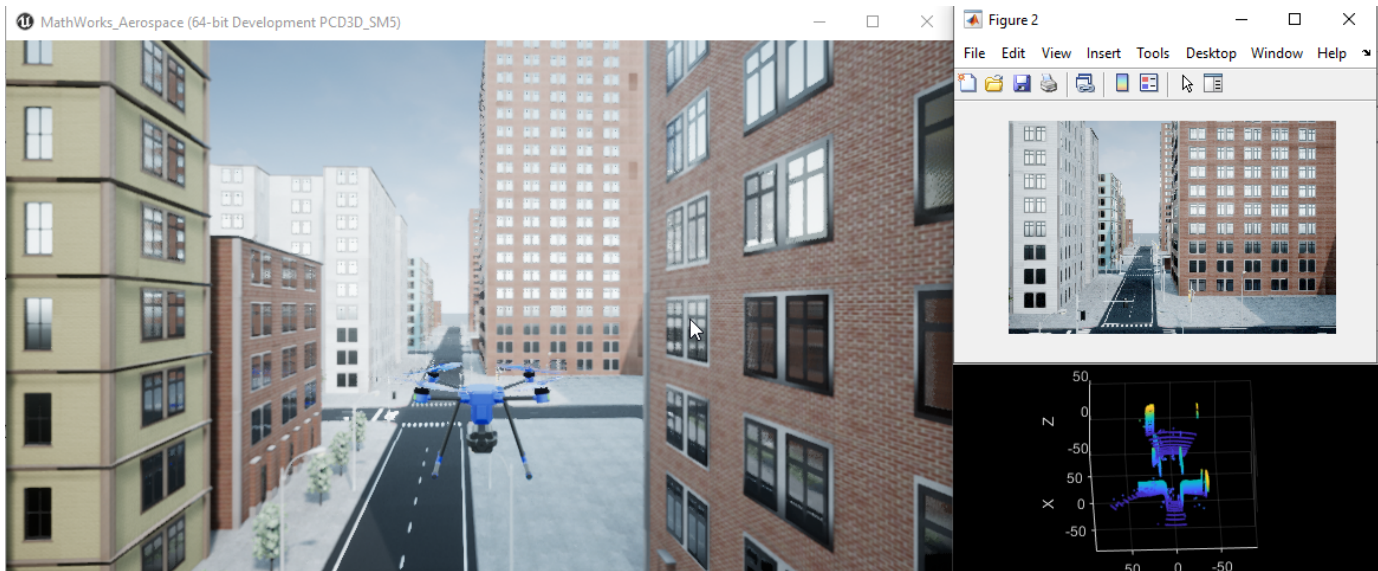
### 5. Photorealistic Simulation

Up to this point, the environment has been a simple cuboid scenario. To increase the fidelity of the environment, click the **Photorealistic Simulation** shortcut, which places the aircraft in a more realistic world to fly through. The PhotorealisticQuadrotor variant located at `uavPackageDelivery/photorealisticSimulationEngi/SimulationEnvironmentVariant` becomes active. This variant contains the necessary blocks to configure the simulation environment and the sensors mounted on the aircraft:



**Run** the model. The aircraft is setup to fly the same mission from steps 1 and 2. Notice as the aircraft flies the mission the lidar point clouds update and an image from the front-facing camera is shown.



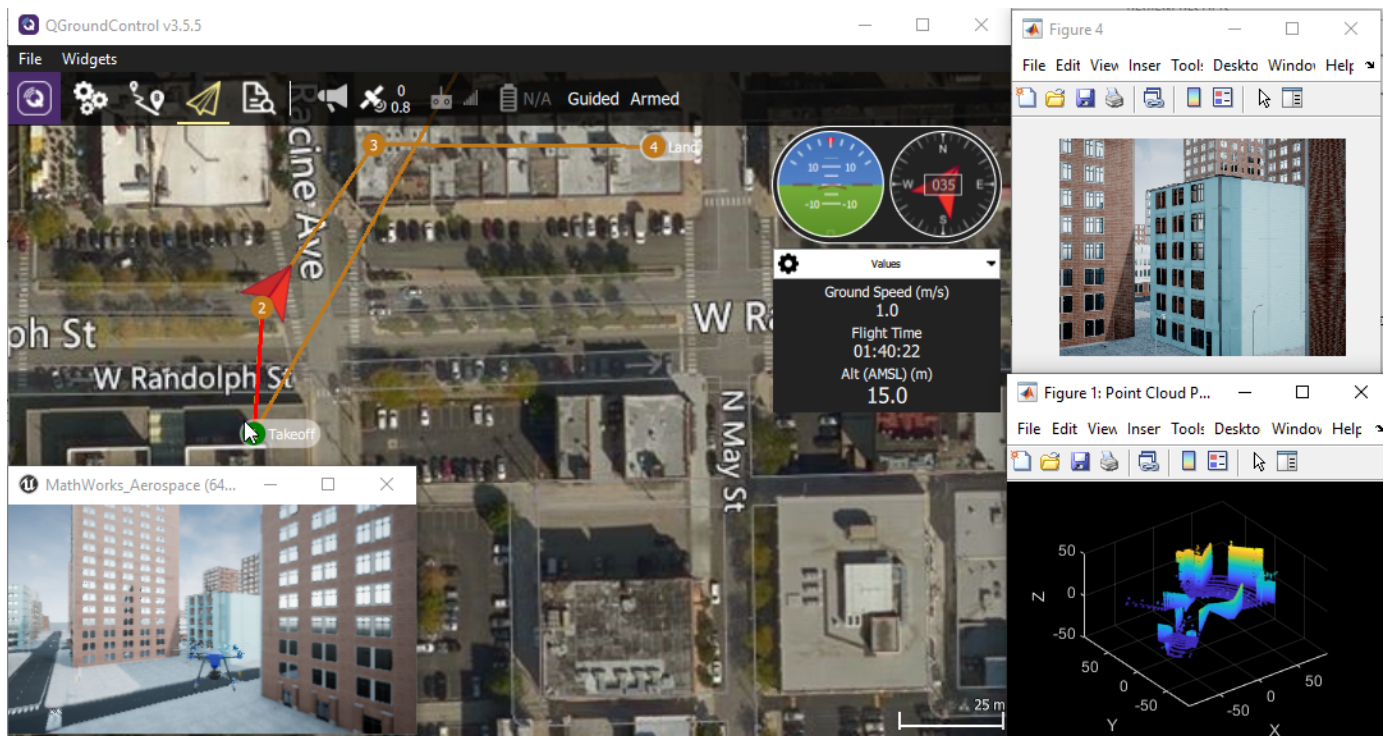


## 6. Fly Full Mission in a Photorealistic Simulation Environment

Next, click the **Fly full mission** shortcut, which sets up the connectivity to QGroundControl from step 2 for uploading the mission inside the photorealistic environment. Follow these steps to run the simulation:

- 1 Launch QGroundControl.
- 2 In QGroundControl, load the mission plan named `shortMission.plan` located in `/utilities/qgc`.
- 3 **Run** the Simulation.
- 4 When QGroundControl indicates that it is connected to a system, upload the mission.

As the aircraft starts to fly, you can modify the mission in QGroundControl by adding waypoints or moving those that are already in the mission. Upload the mission and the aircraft should respond to these changes. Throughout the flight you'll see the aircraft flying in the scenario.

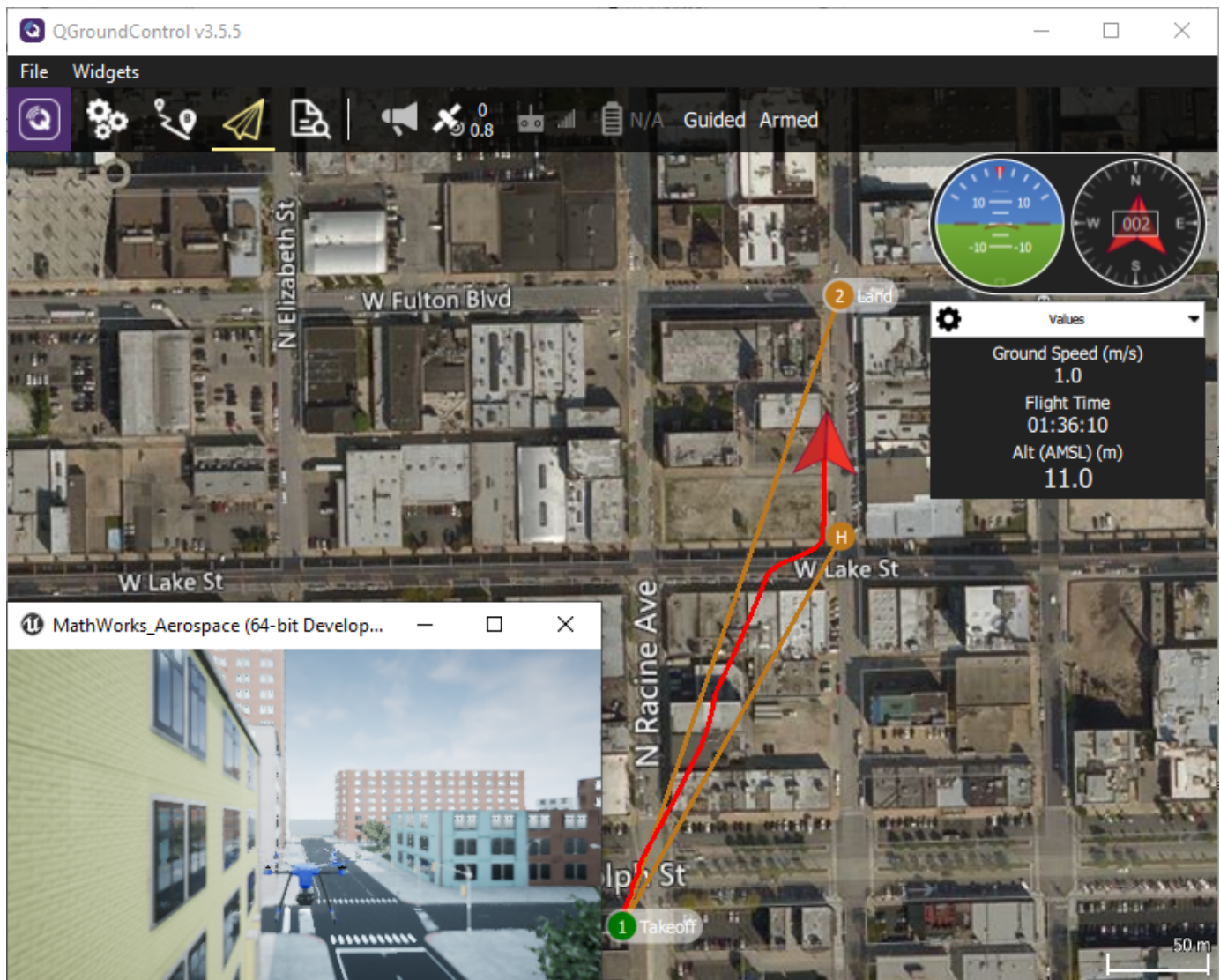


## 7. Flying Obstacle Avoidance in a Photorealistic Simulation Environment

Next, the goal is to fly a mission by specifying a takeoff and landing point in QGroundControl and using the obstacle avoidance to navigate around the obstacles along the path. Click the **Fly full Obstacle Avoidance** shortcut and follow these steps to run the simulation:

- 1 Launch QGroundControl.
- 2 In QGroundControl, load the mission plan named `oaMission.plan` located in `/utilities/qgc`.
- 3 **Run** the Simulation.
- 4 When QGroundControl indicates that it is connected to a system, upload the mission.

Throughout the flight, watch the aircraft try to follow the commanded path in QGroundControl, while at the same time attempting to avoid colliding with the buildings in the environment.

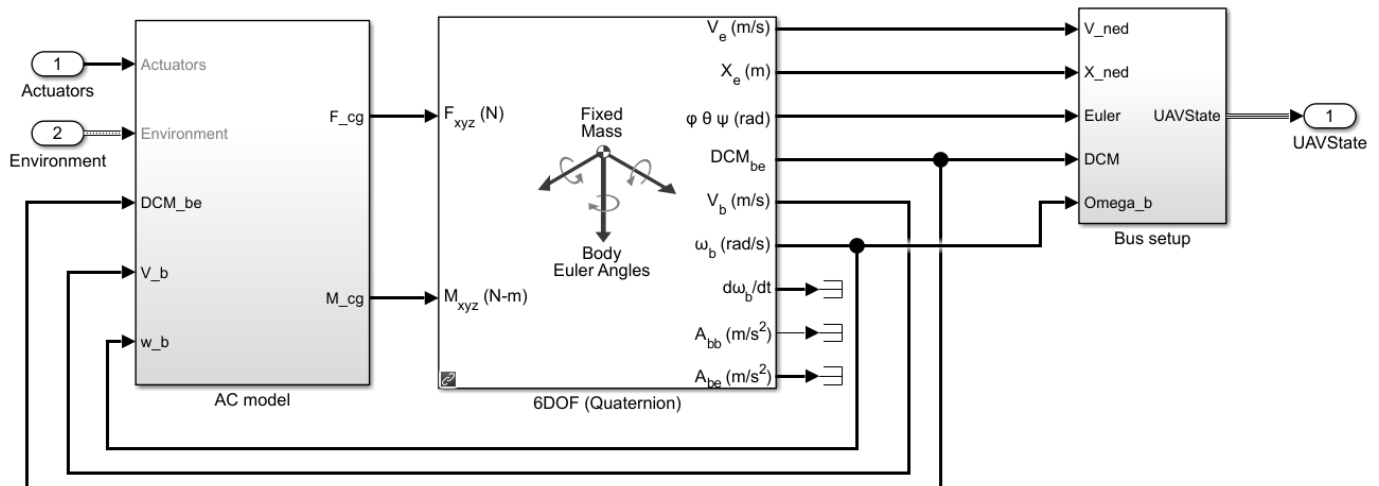


At some point during the flight, you will see the UAV pass through a narrow pass between two buildings.



### 8. Adding a 6DOF Plant Model for Higher-Fidelity Simulation

As a final step, click the **Adding a High-fidelity Plan** shortcut, which activates the high-fidelity variant of the UAV model located at **uavPackageDelivery/MultirotorModel/Inner Loop and Plant Model/High-FidelityModel**. This variant contains an inner-loop controller and a high-fidelity plant model



**Run** the model. There are minor changes in behavior due to the high-fidelity model, but the UAV flies the same mission.

When you are done exploring the models, close the project file.

```
close(prj);
```

# UAV Scenario Tutorial

Create a scenario to simulate unmanned aerial vehicle (UAV) flights between a set of buildings. The example demonstrates updating the UAV pose in open-loop simulations. Use the UAV scenario to visualize the UAV flight and generate simulated point cloud sensor readings.

## Introduction

To test autonomous algorithms, a UAV scenario enables you to generate test cases and generate sensor data from the environment. You can specify obstacles in the workspace, provide trajectories of UAVs in global coordinates, and convert data between coordinate frames. The UAV scenario enables you to visualize this information in the reference frame of the environment.

## Create Scenario with Polygon Building Meshes

A `uavScenario` object is model consisting of a set of static obstacles and movable objects called platforms. Use `uavPlatform` objects to model fixed-wing UAVs, multirotors, and other objects within the scenario. This example builds a scenario consisting of a ground plane and 11 buildings as by extruded polygons. The polygon data for the buildings is loaded and used to add polygon meshes.

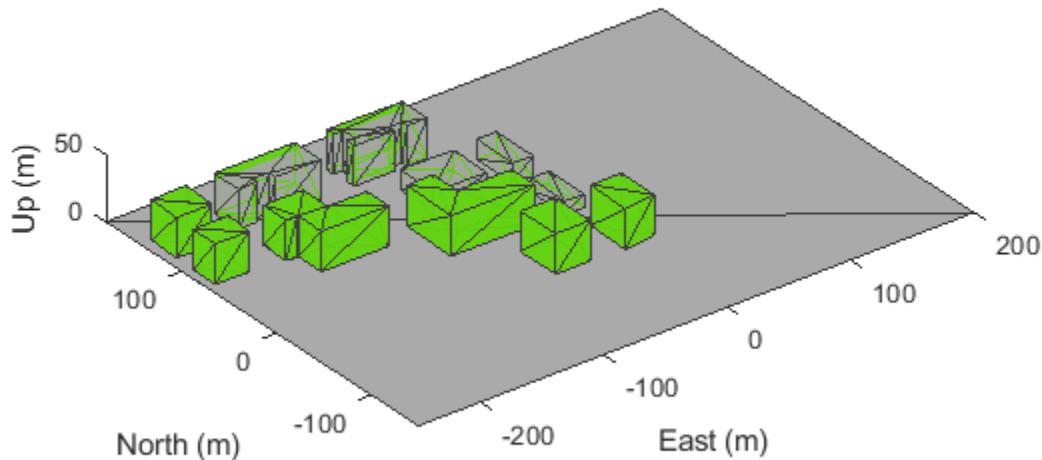
```
% Create the UAV scenario.
scene = uavScenario("UpdateRate",2,"ReferenceLocation",[75 -46 0]);

% Add a ground plane.
color.Gray = 0.651*ones(1,3);
color.Green = [0.3922 0.8314 0.0745];
color.Red = [1 0 0];
addMesh(scene,"polygon",{[-250 -150; 200 -150; 200 180; -250 180],[-4 0]},color.Gray)

% Load building polygons.
load("buildingData.mat");

% Add sets of polygons as extruded meshes with varying heights from 10-30.
addMesh(scene,"polygon",{buildingData{1}(1:4,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{2}(2:5,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{3}(2:10,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{4}(2:9,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{5}(1:end-1,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{6}(1:end-1,:),[0 15]},color.Green)
addMesh(scene,"polygon",{buildingData{7}(1:end-1,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{8}(2:end-1,:),[0 10]},color.Green)
addMesh(scene,"polygon",{buildingData{9}(1:end-1,:),[0 15]},color.Green)
addMesh(scene,"polygon",{buildingData{10}(1:end-1,:),[0 30]},color.Green)
addMesh(scene,"polygon",{buildingData{11}(1:end-2,:),[0 30]},color.Green)

% Show the scenario.
show3D(scene);
xlim([-250 200])
ylim([-150 180])
zlim([0 50])
```



### Define UAV Platform and Mount Sensor

You can define a `uavPlatform` in the scenario as a carrier of your sensor models and drive them through the scenario to collect simulated sensor data. You can associate the platform with various meshes, such as `fixedwing`, `quadrotor`, and `cuboid` meshes. You can define a custom mesh defined ones represented by vertices and faces. Specify the reference frame for describing the motion of your platform.

Load flight data into the workspace and create a quadrotor platform using an NED reference frame. Specify the initial position and orientation based on loaded flight log data. The configuration of the UAV body frame orients the  $x$ -axis as forward-positive, the  $y$ -axis as right-positive, and the  $z$ -axis downward-positive.

```
load("flightData.mat")

% Set up platform
plat = uavPlatform("UAV",scene,"ReferenceFrame","NED", ...
    "InitialPosition",position(:,:,1),"InitialOrientation",eul2quat(orientation(:,:,1)));

% Set up platform mesh. Add a rotation to orient the mesh to the UAV body frame.
updateMesh(plat,"quadrotor",{10},color.Red,[0 0 0],eul2quat([0 0 pi]))
```

You can choose to mount different sensors, such as the `insSensor`, `gpsSensor`, or `uavLidarPointCloudGenerator` System objects to your UAV. Mount a lidar point cloud generator and a `uavSensor` object that contains the lidar sensor model. Specify a mounting location of the sensor that is relative to the UAV body frame.

```
lidarmodel = uavLidarPointCloudGenerator("AzimuthResolution",0.3324099,...
    "ElevationLimits",[-20 20],"ElevationResolution",1.25,...
    "MaxRange",90,"UpdateRate",2,"HasOrganizedOutput",true);
```

```
lidar = uavSensor("Lidar",plat,lidarmodel,"MountingLocation",[0,0,-1]);
```

### Fly the UAV Platform Along Pre-defined Trajectory and Collect Point Cloud Sensor Readings

Move the UAV along a pre-defined trajectory, and collect the lidar sensor readings along the way. This data could be used to test lidar-based mapping and localization algorithms.

Preallocate the `traj` and `scatterPlot` line plots and then specify the plot-specific data sources. During the simulation of the `uavScenario`, use the provided `plotFrames` output from the scene as the parent axes to visualize your sensor data in the correct coordinate frames.

```
% Visualize the scene
[ax,plotFrames] = show3D(scene);

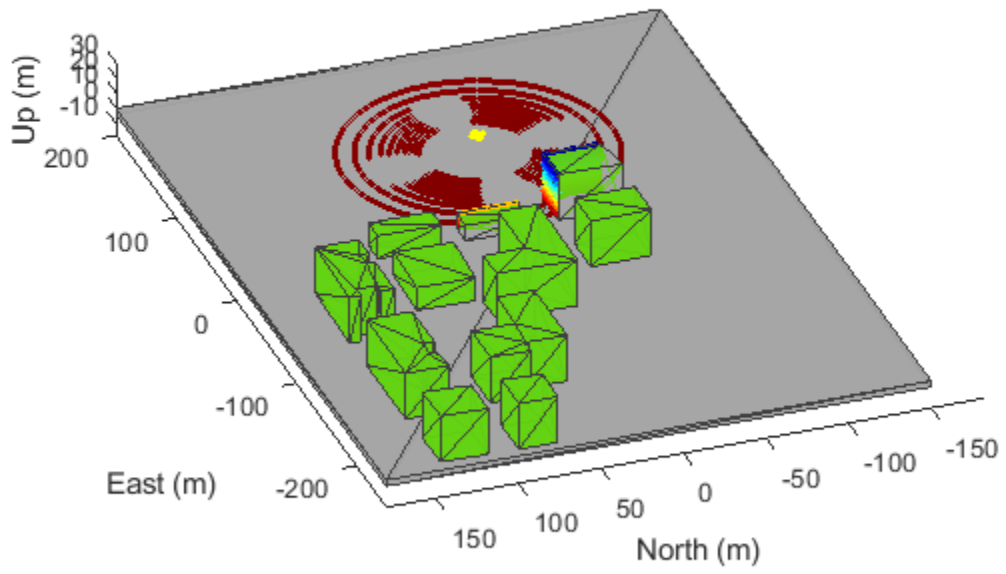
% Update plot view for better visibility.
xlim([-250 200])
ylim([-150 180])
zlim([0 50])
view([-110 30])
axis equal
hold on

% Create a line plot for the trajectory.
traj = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
traj.XDataSource = "position(:,1,1:idx+1)";
traj.YDataSource = "position(:,2,1:idx+1)";
traj.ZDataSource = "position(:,3,1:idx+1)";

% Create a scatter plot for the point cloud.
colormap("jet")
pt = pointCloud(nan(1,1,3));
scatterplot = scatter3(nan,nan,nan,1,[0.3020 0.7451 0.9333],...
    "Parent",plotFrames.UAV.Lidar);
scatterplot.XDataSource = "reshape(pt.Location(:,:,1),[],1)";
scatterplot.YDataSource = "reshape(pt.Location(:,:,2),[],1)";
scatterplot.ZDataSource = "reshape(pt.Location(:,:,3),[],1)";
scatterplot.CDataSource = "reshape(pt.Location(:,:,3),[],1) - min(reshape(pt.Location(:,:,3),[],1)";

% Start Simulation
setup(scene)
for idx = 0:size(position, 3)-1
    [isupdated,lidarSampleTime, pt] = read(lidar);
    if isupdated
        % Use fast update to move platform visualization frames.
        show3D(scene,"Time",lidarSampleTime,"FastUpdate",true,"Parent",ax);
        % Refresh all plot data and visualize.
        refreshdata
        drawnow limitrate
    end
    % Advance scene simulation time and move platform.
    advance(scene);
    move(plat,[position(:,:,idx+1),zeros(1,6),eul2quat(orientation(:,:,idx+1)),zeros(1,3)])
    % Update all sensors in the scene.
    updateSensors(scene)
```

```
end  
hold off
```





## Tune UAV Parameters Using MAVLink Parameter Protocol

This example shows how to use a MAVLink parameter protocol in MATLAB and communicate with external ground control stations. A sample parameter protocol is provided for sending parameter updates from a simulated unmanned aerial vehicle (UAV) to a ground control station using MAVLink communication protocols. You setup the communication between the two MAVLink components, the UAV and the ground control station. Then, you send and receive parameter updates to tune parameter values for the UAV. Finally, if you use QGroundControl© as a ground control station, you can get these parameter updates from QGroundControl and see them reflected in the program window.

### Parameter Protocol

MAVLink clients exchange information within the network using commonly defined data structures as messages. MAVLink parameter protocol is used to exchange configuration settings between UAV and ground control station (GCS). Parameter protocol follows a client-server pattern. For example, GCS initiates a request in the form of messages and the UAV responds with data.

### Setup common dialect

MAVLink messages are defined in an XML file. Standard messages that are common to all systems are defined in the "common.xml" file. Other vendor-specific messages are stored in separate XML files. For this example, use the "common.xml" file to setup a common dialect between the MAVLink clients.

```
dialect = mavlinkdialect("common.xml");
```

This dialect is used to create mavlinkio objects which can understand messages within the dialect.

### Setup UAV Connection

Create a mavlinkio object to represent a simulated UAV. Specify the SystemID, ComponentID, AutoPilotType, and ComponentType parameters as name-value pairs. For this example, we use a generic autopilot type, 'MAV\_AUTOPILOT\_GENERIC', with a quadrotor component type, 'MAV\_TYPE\_QUADROTOR'.

```
uavNode = mavlinkio(dialect, 'SystemID', 1, 'ComponentID', 1, ...
    'AutopilotType', "MAV_AUTOPILOT_GENERIC", 'ComponentType', "MAV_TYPE_QUADROTOR");
```

The simulated UAV is listening on a UDP port for incoming messages. Connect to this UDP port using the uavNode object.

```
uavPort = 14750;
connect(uavNode, "UDP", 'LocalPort', uavPort);
```

### Setup GCS Connection

Create a simulated ground control station (GCS) that listens on a different UDP port.

```
gcsNode = mavlinkio(dialect);
gcsPort = 14560;
connect(gcsNode, "UDP", 'LocalPort', gcsPort);
```

## Setup Client and Subscriber

Setup a client interface for the simulated UAV to communicate with the ground control station. Get the `LocalClient` information as a structure and specify the system and component ID info to the `mavlinkclient` object.

```
clientStruct = uavNode.LocalClient;  
uavClient = mavlinkclient(gcsNode,clientStruct.SystemID,clientStruct.ComponentID);
```

Create a `mavlinksub` object to receive messages and process those messages using a callback. This subscriber receives messages on the 'PARAM\_VALUE' topic and specifically looks for messages matching the system and component ID of `uavClient`. A callback function is specified to display the payload of each new message received.

```
paramValueSub = mavlinksub(gcsNode,uavClient,'PARAM_VALUE','BufferSize',10,...  
                           'NewMessageFcn', @(~,msg)disp(msg.Payload));
```

## Parameter Operations

Now that you have setup the connections between the UAV and ground control station. You can now query and update the simulated UAV configuration using operations defined in parameter protocol, `exampleHelperMAVParamProtocol`. There are 4 GCS operations that describe the workflow of parameter protocol. Each message type listed has a brief description what the message executes based on the specified parameter protocol.

- 1 `PARAM_REQUEST_LIST`: Requests all parameters from the recipients. All values are broadcasted using `PARAM_VALUE` messages.
- 2 `PARAM_REQUEST_READ`: Requests a single parameter. The specified parameter value is broadcasted using a `PARAM_VALUE` message.
- 3 `PARAM_SET`: Commands to set the value of the specific parameter. After setting up the value, the current value is broadcasted using a `PARAM_VALUE` message.
- 4 `PARAM_VALUE`: Broadcasts the current value of a parameter in response to the above requests (`PARAM_REQUEST_LIST`, `PARAM_REQUEST_READ` or `PARAM_SET`).

```
paramProtocol = exampleHelperMAVParamProtocol(uavNode);
```

This parameter protocol has three parameter values: 'MAX\_ROLL\_RATE', 'MAX\_PITCH\_RATE', and 'MAX\_YAW\_RATE'. These values represent the maximum rate for roll, pitch, and yaw for the UAV in degrees per second. In a real UAV systems, these rates can be tuned to adjust performance for more or less acrobatic control.

## Read All Parameters

To read all parameters from a UAV system, send a "PARAM\_REQUEST\_LIST" message from `gcsNode` to `uavNode`. The sequence of operations are as follows:

- 1 GCS node sends a message whose topic is "PARAM\_REQUEST\_LIST" to the UAV node specifying the target system and component using `uavClient` as defined above.
- 2 UAV node sends out all parameters individually in the form of "PARAM\_VALUE" messages, since we have a subscriber on the GCS node which is subscribed to the topic 'PARAM\_VALUE', message payload is being displayed right away.

```
msg = createmsg(dialect,"PARAM_REQUEST_LIST");
```

Assign values for the system and component ID into the message, use ( : )= indexing to make sure the assignment doesn't change the struct field data type.

```
msg.Payload.target_system(:) = uavNode.LocalClient.SystemID;
msg.Payload.target_component(:) = uavNode.LocalClient.ComponentID;
```

Send the parameter request to the UAV, which is listening on a port at local host IP address '127.0.0.1'. Pause to allow the message to be processed. The parameter list is displayed in the command window.

```
sendudpmsg(gcsNode,msg,"127.0.0.1",uavPort)
pause(1);
```

```
param_value: 90
param_count: 3
param_index: 0
  param_id: 'MAX_ROLL_RATE    '
  param_type: 9

param_value: 90
param_count: 3
param_index: 1
  param_id: 'MAX_YAW_RATE    '
  param_type: 9

param_value: 90
param_count: 3
param_index: 2
  param_id: 'MAX_PITCH_RATE   '
  param_type: 9
```

### Read Single Parameter

Read a single parameter by sending a "PARAM\_REQUEST\_READ" message from the GCS node to the UAV node. Send a message on the "PARAM\_REQUEST\_READ" topic to the UAV node. Specify the parameter index of 0, which refers to the 'MAX\_ROLL\_RATE' parameter. This index value queries the first parameter value.

The UAV sends the updated parameter as a "PARAM\_VALUE" message back to the GCS node. Because we setup a subscriber to the "PARAM\_VALUE" on the GCS node, the message payload is displayed to the command window.

```
msg = createmsg(gcsNode.Dialect,"PARAM_REQUEST_READ");
msg.Payload.param_index(:) = 0;
msg.Payload.target_system(:) = uavNode.LocalClient.SystemID;
msg.Payload.target_component(:) = uavNode.LocalClient.ComponentID;
```

```
sendudpmsg(gcsNode,msg,"127.0.0.1",uavPort);
pause(1);
```

```
param_value: 90
param_count: 3
param_index: 0
  param_id: 'MAX_ROLL_RATE    '
  param_type: 9
```

## Write Parameters

To write a parameter, send a "PARAM\_SET" message from GCS node to UAV node. Specify the ID, type, and value of the message and send using the `gcsNode` object. The UAV sends the updated parameter value back and the GCS subscriber displays the message payload. This message updates the maximum yaw rate of the UAV by reducing it to 45 degrees per second.

```
msg = createmsg(gcsNode.Dialect, "PARAM_SET");
msg.Payload.param_id(1:12) = "MAX_YAW_RATE";
msg.Payload.param_type(:) = 9;
msg.Payload.param_value(:) = 45;
msg.Payload.target_system(:) = uavNode.LocalClient.SystemID;
msg.Payload.target_component(:) = uavNode.LocalClient.ComponentID;

sendudpmsg(gcsNode, msg, "127.0.0.1", uavPort);
pause(1);

    param_value: 45
    param_count: 3
    param_index: 2
    param_id: 'MAX_YAW_RATE'
    param_type: 9
```

## Working with QGroundControl

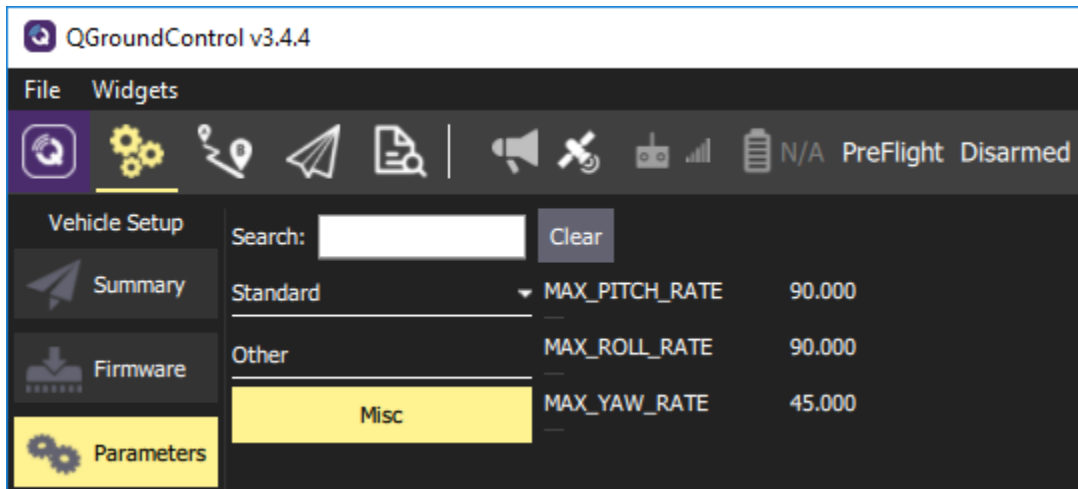
QGroundControl© is an app that is used to perform flight control and mission planning for any MAVLink-enabled UAV. You can use QGroundControl as a GCS to demonstrate how to access parameters of our simulated UAV:

- 1 Download and launch QGroundControl. Define `qgcPort` number as 14550, which is the default UDP port for the QGroundControl app.
- 2 Create a heartbeat message.
- 3 Send heartbeat message from UAV node to QGroundControl using the MATLAB timer object. By default, the timer object executes the `TimerFcn` every 1 second. The `TimerFcn` is a `sendudpmsg` call that sends the heartbeat message.
- 4 Once QGroundControl receives the heartbeat from the simulated UAV, QGroundControl creates a Parameter panel widget for the user to read and update UAV parameters

```
qgcPort = 14550;
heartbeat = createmsg(dialect, "HEARTBEAT");
heartbeat.Payload.type(:) = enum2num(dialect, 'MAV_TYPE', uavNode.LocalClient.ComponentType);
heartbeat.Payload.autopilot(:) = enum2num(dialect, 'MAV_AUTOPILOT', uavNode.LocalClient.AutopilotType);
heartbeat.Payload.system_status(:) = enum2num(dialect, 'MAV_STATE', "MAV_STATE_STANDBY");

heartbeatTimer = timer;
heartbeatTimer.ExecutionMode = 'fixedRate';
heartbeatTimer.TimerFcn = @(~,~)sendudpmsg(uavNode, heartbeat, '127.0.0.1', qgcPort);
start(heartbeatTimer);
```

While the timer runs, QGroundControl shows it has received the heartbeat message and is connected to a UAV. In the **Vehicle Setup** tab, click **Other > Misc** to see the parameter values set are reflected in the app.



**Note:** Because we use a generic autopilot type, "MAV\_AUTOPILOT\_GENERIC", QGroundControl does not recognize the connection as a known autopilot type. This does not affect the connection and the parameter values should still update as shown.

### Close MAVLink connections

After experimenting with the QGroundControl parameter widget, stop the `heartbeatTimer` to stop sending any more heartbeat messages. Delete the `heartbeatTimer` and the `paramProtocol` objects. Finally, disconnect the UAV and GCS nodes to clean up the communication between systems.

```
stop(heartbeatTimer);
delete(heartbeatTimer);
delete(paramProtocol);
```

```
disconnect(uavNode);
disconnect(gcsNode);
```

## Exchange Data for MAVLink Microservices like Mission Protocol and Parameter Protocol Using Simulink

This example shows how to implement MAVLink microservices like Mission protocol and Parameter protocol using the MAVLink Serializer and MAVLink Deserializer blocks in Simulink®.

This example uses:

- MATLAB®
- Simulink®
- UAV Toolbox™
- Stateflow™
- Instrument Control Toolbox™
- DSP System Toolbox™

The Mission protocol microservice in MAVLink allows a Ground Control Station (GCS) to communicate with a drone to send and receive mission information needed to execute a mission. The Mission protocol microservice allows you to:

- Upload a mission from the GCS to the drone
- Download a mission from the drone
- Set the current mission item

The Parameter protocol microservice in MAVLink allows you to exchange parameters representing important configuration information between the drone and the GCS. The parameters are represented as key-value pairs.

This example explains how to:

- Upload a mission consisting of 10 waypoints from the GCS to a drone emulated in Simulink. Use QGroundControl (QGC) as the GCS. If you do not have the QGC installed on the host computer, download it from [here](#).
- Read and write data to a list of 28 parameters from the QGC and the drone.

### Design Model

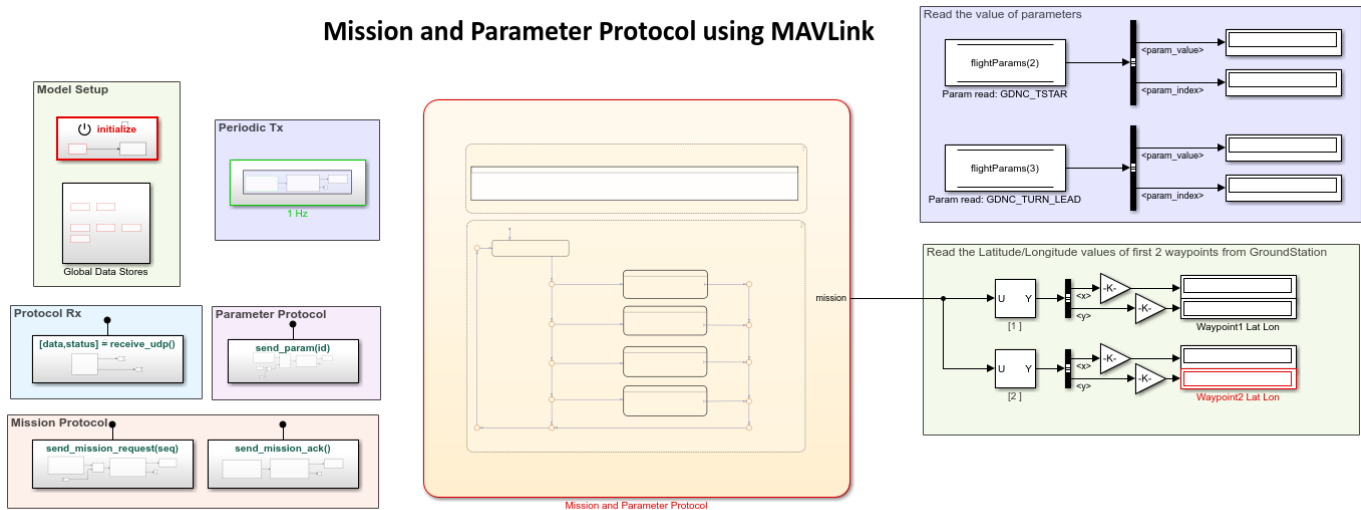
To get started, follow these steps:

1. Open the `exampleHelperMAVLinkMissionAndParamProtocol` file in MATLAB and click **Run**. This creates the workspace variables required to initialize the data in Simulink and upload the autopilot parameters to the QGC.

2. Launch the example model in Simulink by clicking **Open Model** at the top of this page. You can also use the following command to launch the model anytime after you clicked the **Open Model** button once:

```
open_system('MissionAndParameterProtocolUsingMAVLink.slx');
```

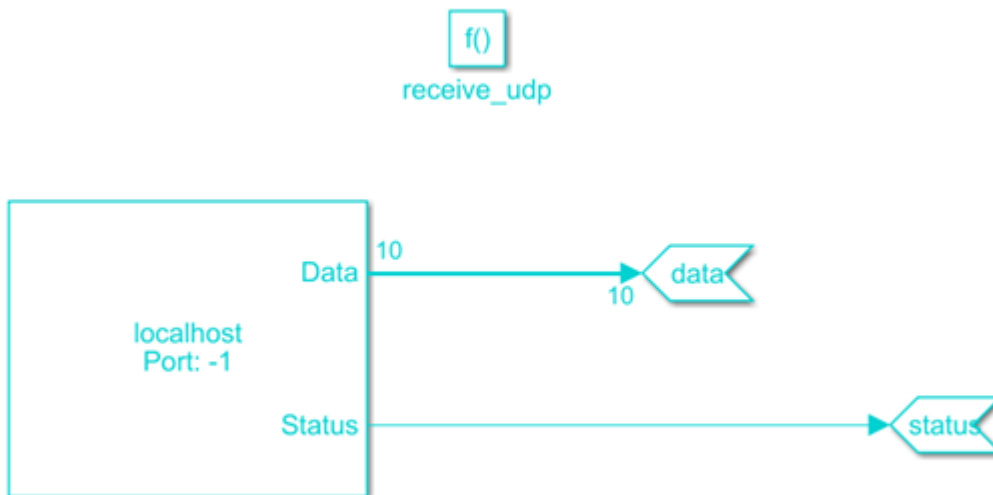
### Mission and Parameter Protocol using MAVLink



Copyright 2020 The MathWorks, Inc.

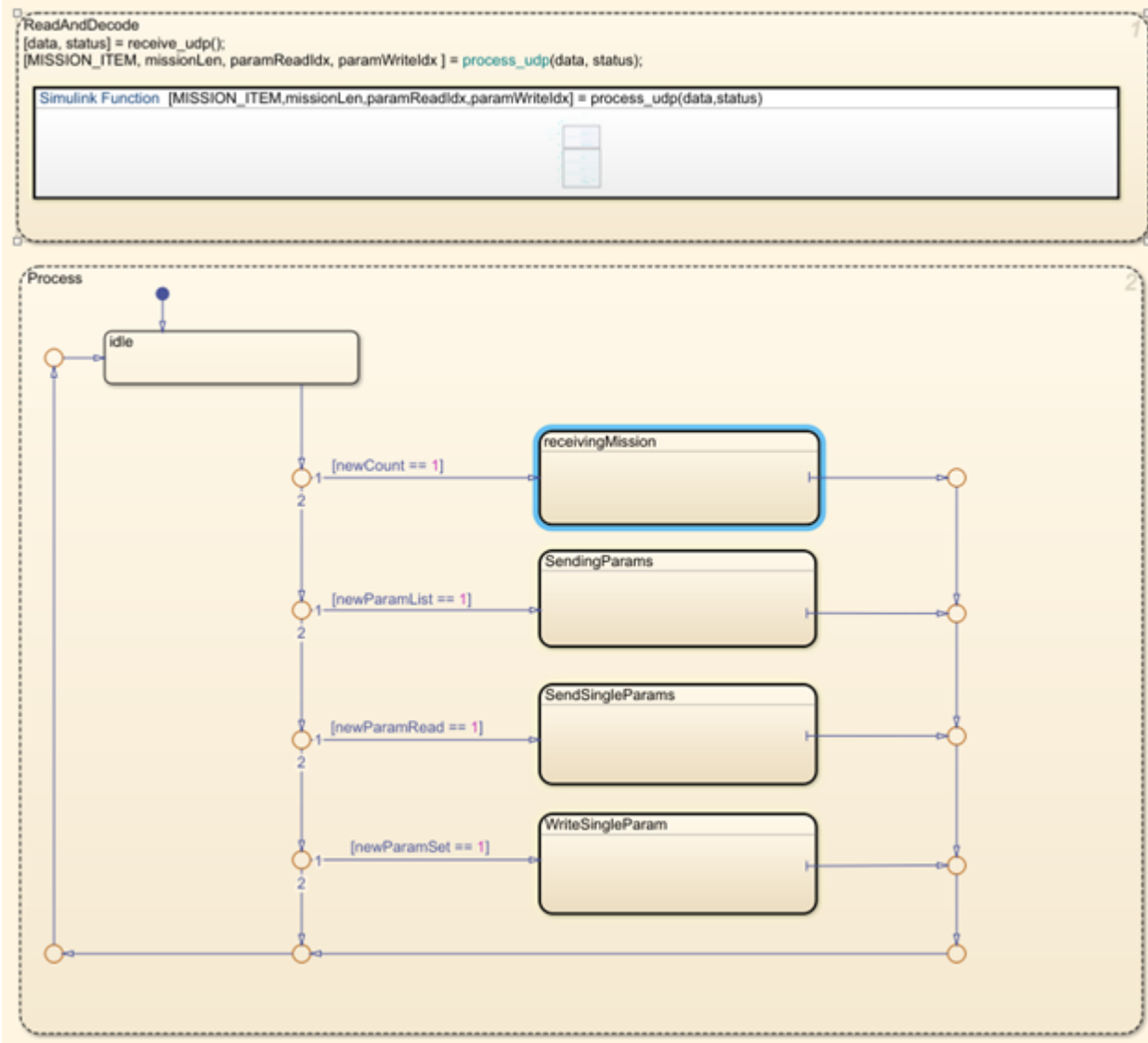
The Simulink model consists of:

1. **Model Setup:** This area in the model consists of two subsystem blocks - Initialize Function and Global Data Stores. These blocks are used to initialize the data that will be used in the model from the signals generated in the base workspace.
2. **Protocol Rx:** This area in the model consists of the `receive_udp` subsystem block that is used to receive UDP data from QGC. The subsystem contains a Simulink function that reads the MAVLink data over UDP from the QGC, at each simulation step. The received MAVLink data is passed to a Stateflow chart for decoding and parsing.



3. **Mission Protocol:** This area in the model consists of two subsystem blocks that send mission requests and mission acknowledgements to the QGC. These functions are called from the Stateflow chart that implements the mission microservice.

4. **Mission and Parameter Protocol:** The Stateflow chart that implements the mission and parameter logic in the model.



The received MAVLink data is deserialized in the `process_udp` Simulink function and then passed to the Stateflow logic that performs four tasks:

a. *ReceivingMission*: This Stateflow subchart receives a mission from the QGC and decodes the waypoints in the mission. It implements the protocol of Mission microservice that uploads a mission from QGC to drone, as described in Upload a Mission to the Vehicle.



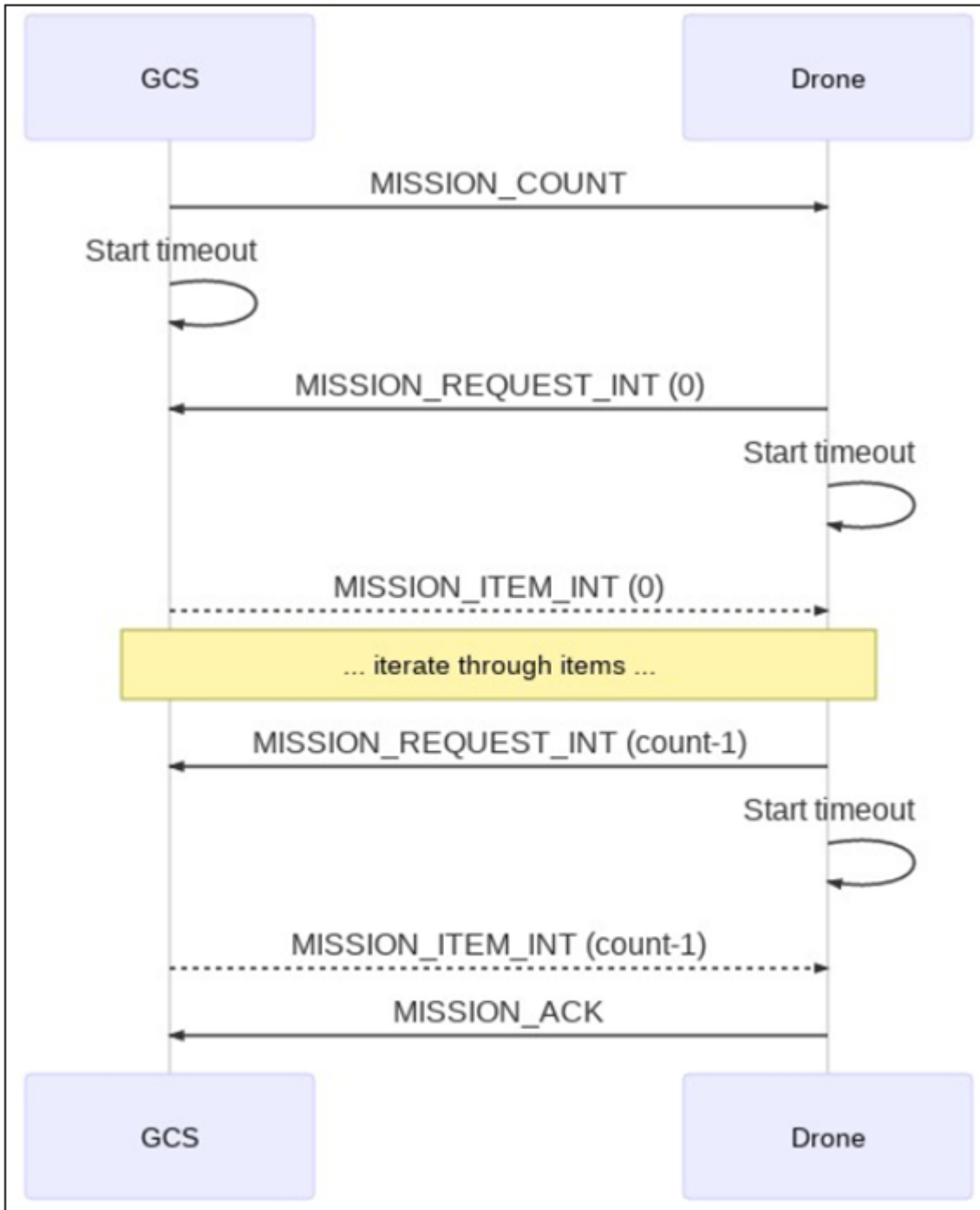


Image courtesy: [https://mavlink.io/en/services/mission.html#uploading\\_mission](https://mavlink.io/en/services/mission.html#uploading_mission)

**b. SendingParams:** This Stateflow subchart uploads the parameters created in the base workspace to the QGC by following the parameter protocol, as described in Read All Parameters.

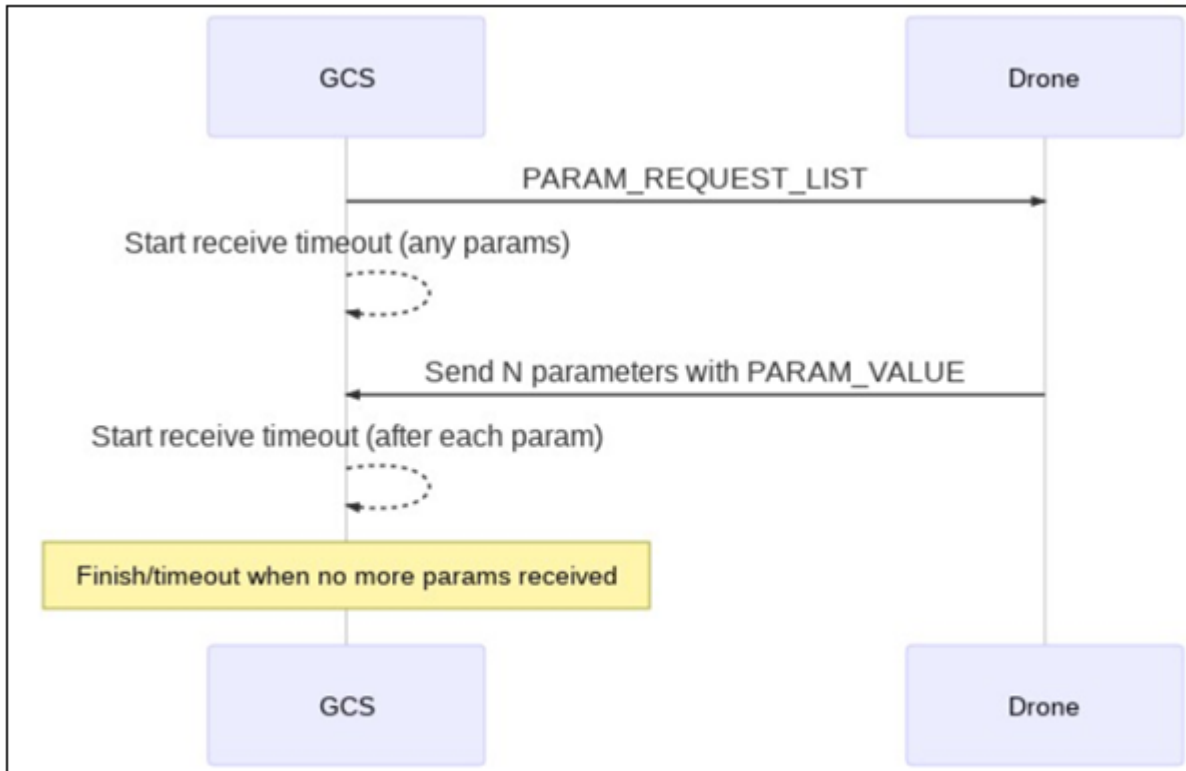


Image courtesy: [https://mavlink.io/en/services/parameter.html#read\\_all](https://mavlink.io/en/services/parameter.html#read_all)

- c. *SendSingleParams*: This Stateflow subchart defines how to send a single parameter from the drone to the QGC, as described in Read Single Parameter.
- d. *WriteSingleParam*: This Stateflow subchart defines how to update the parameter values from the QGC and see them on the drone, as described in Write Parameters.

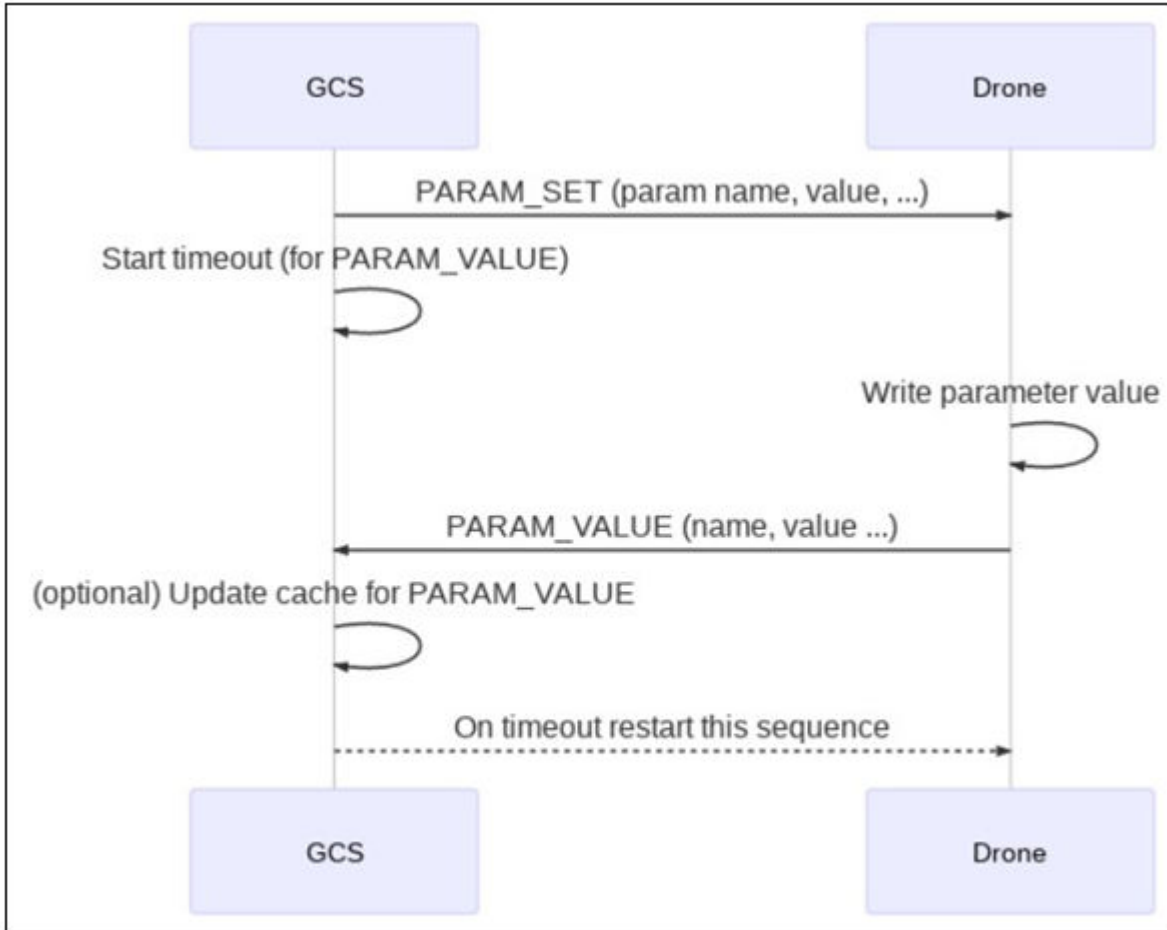
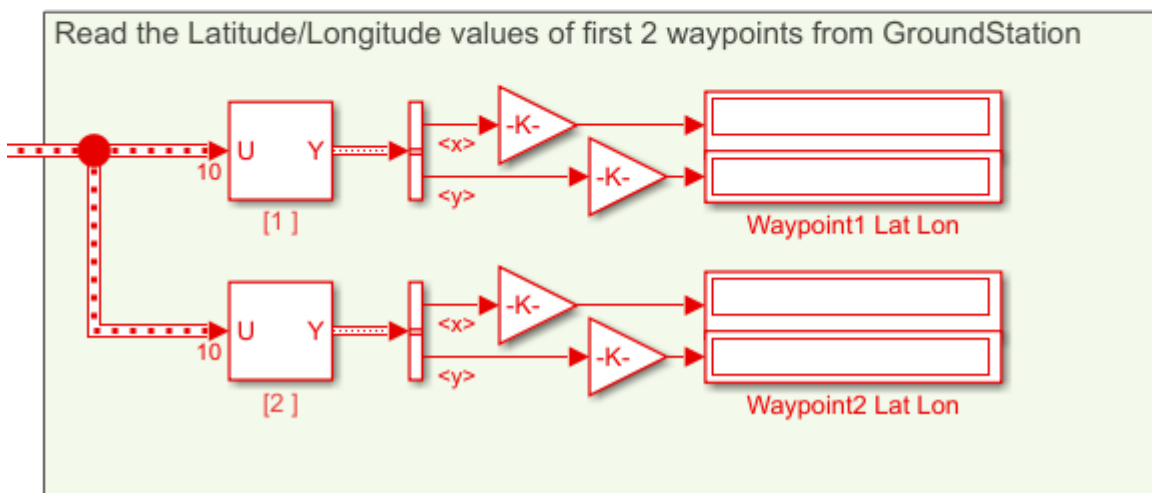
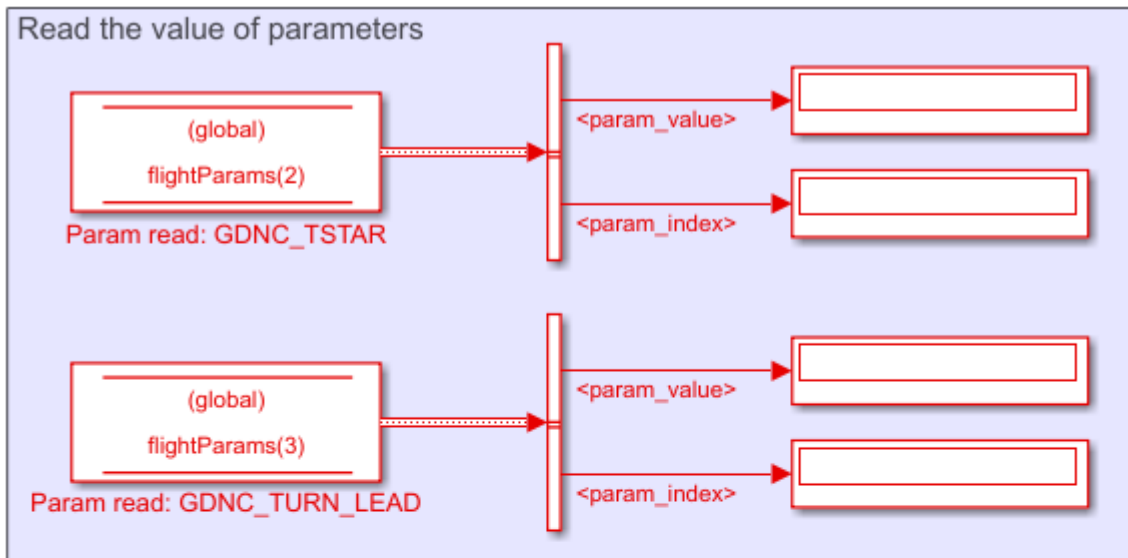


Image courtesy: <https://mavlink.io/en/services/parameter.html#write>

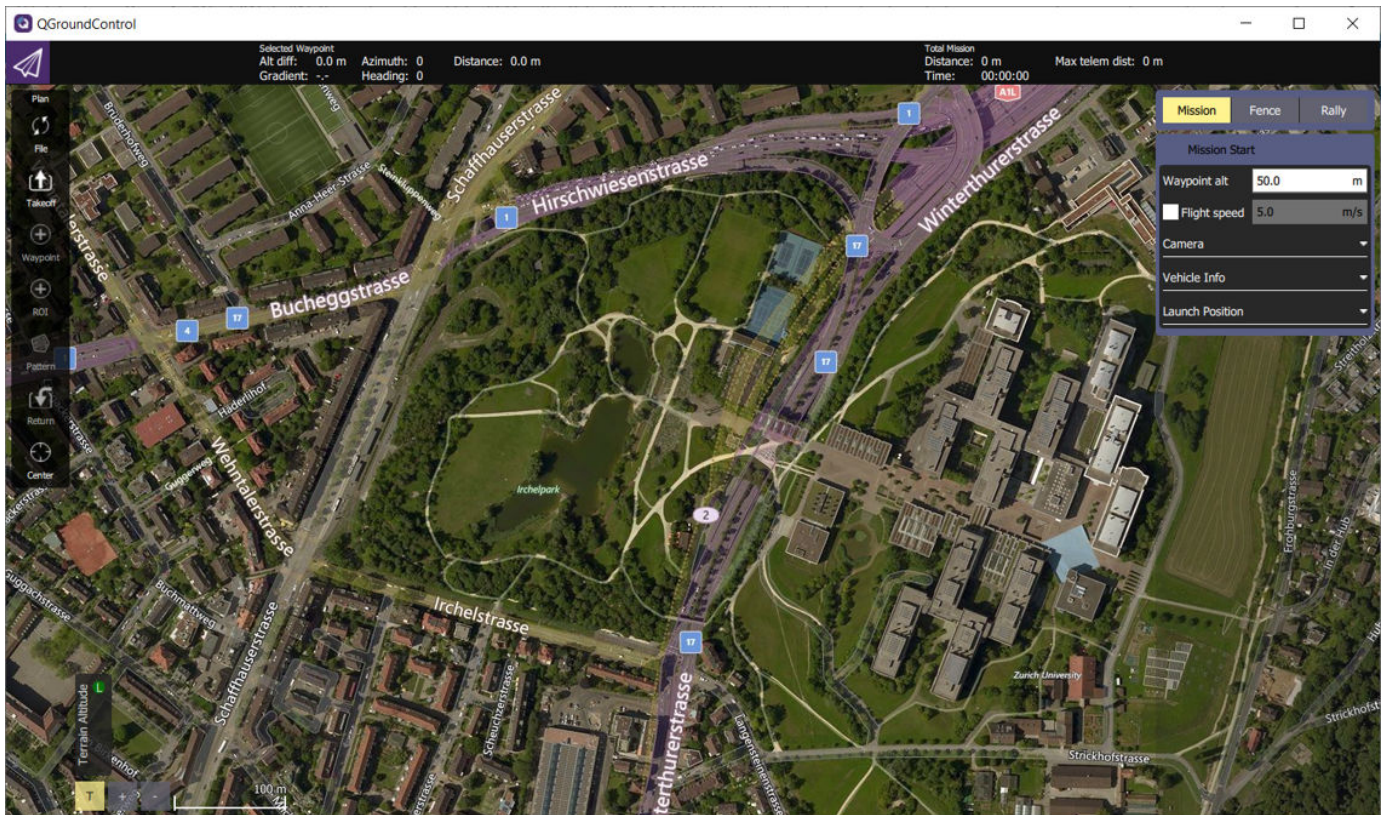
**5. Logic to read received waypoints and parameters:** Stateflow implements the two protocols and outputs the received waypoints and uploaded parameter values.



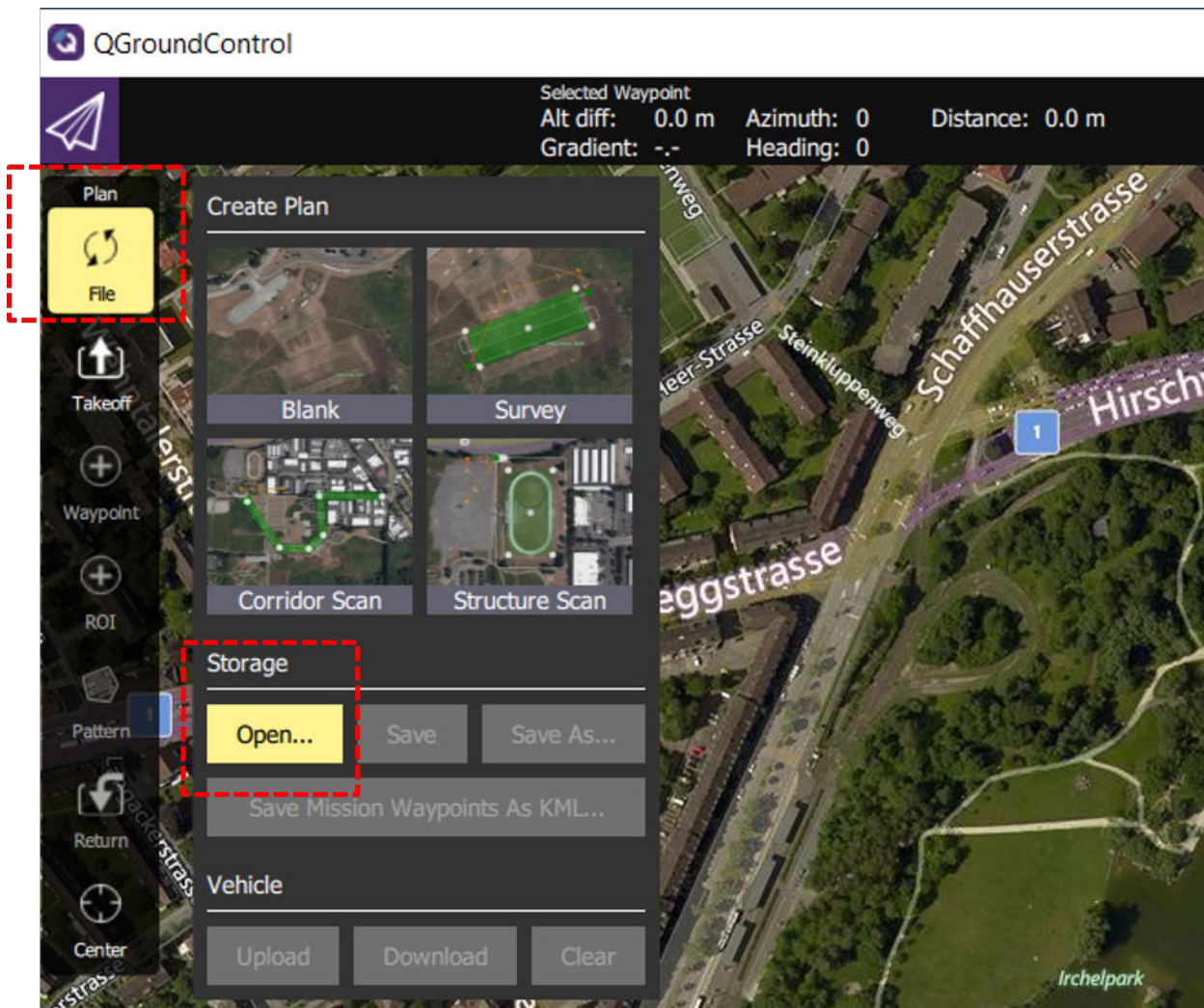
The next section explains how to upload a mission from the QGC to the drone.

### Upload a Mission from QGC to Drone and Run the Simulink Model

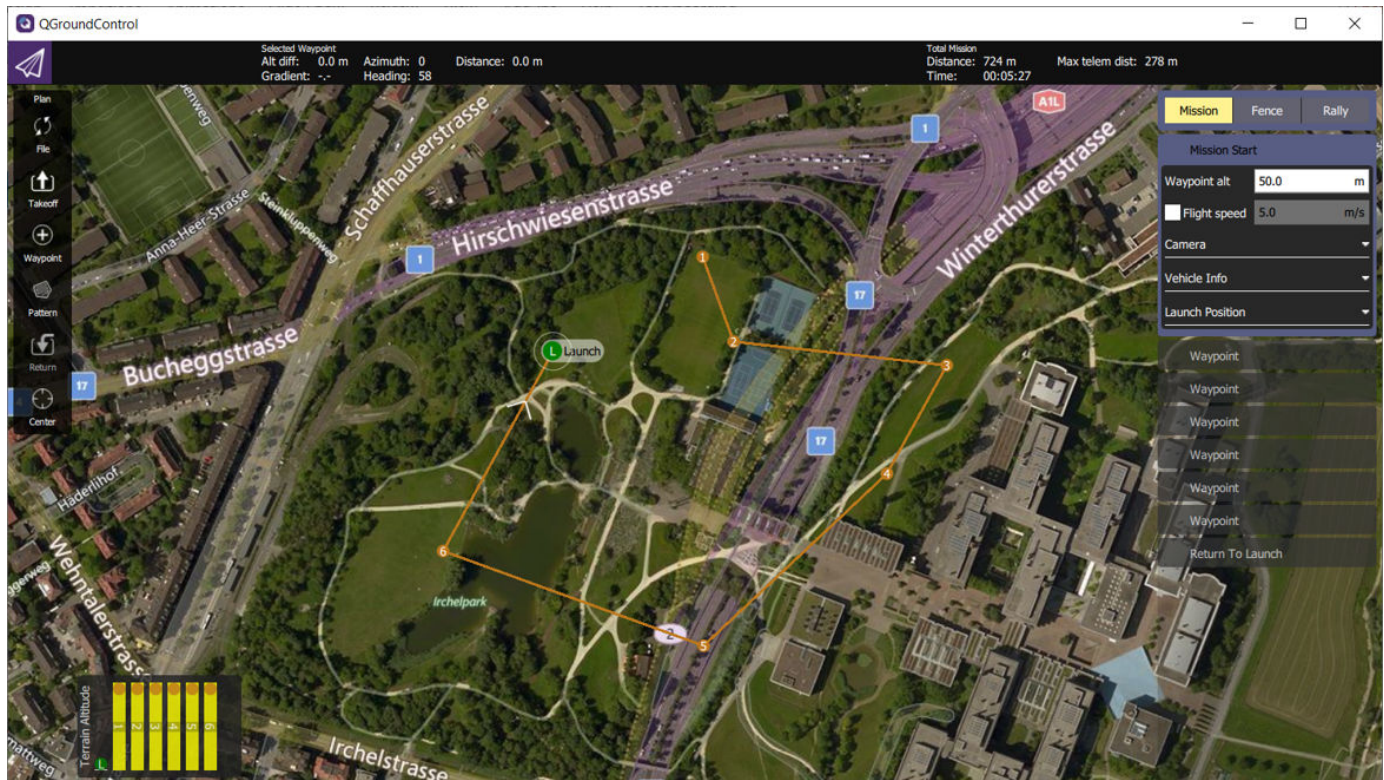
1. Launch the QGC and navigate to the *Plan View*.



2. A preplanned mission, *MissionProtocol.plan*, is available with this example. Click **Open Model** at the top of this page to save the plan file to your computer. After you save the .plan file, launch QGC, and click **File > Open** to upload the plan to the QGC.

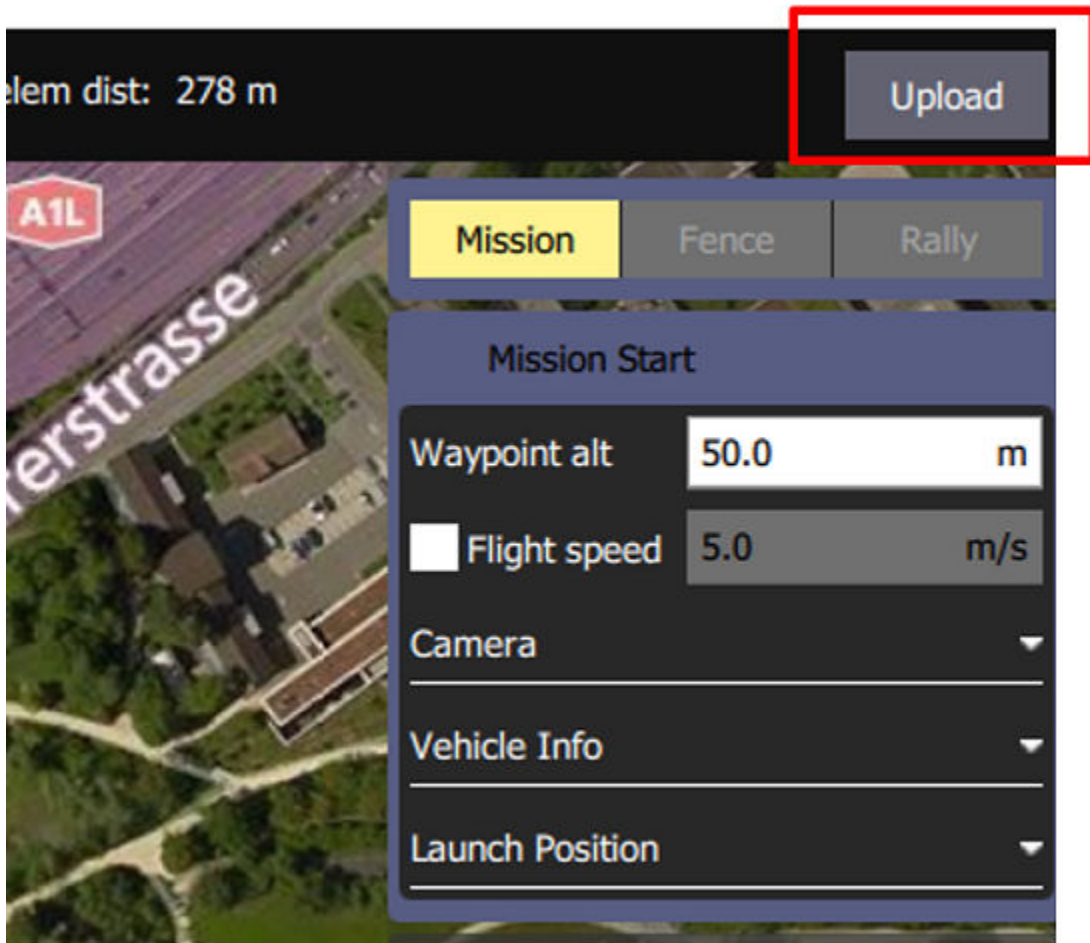


After you upload the plan, the mission is visible in QGC.



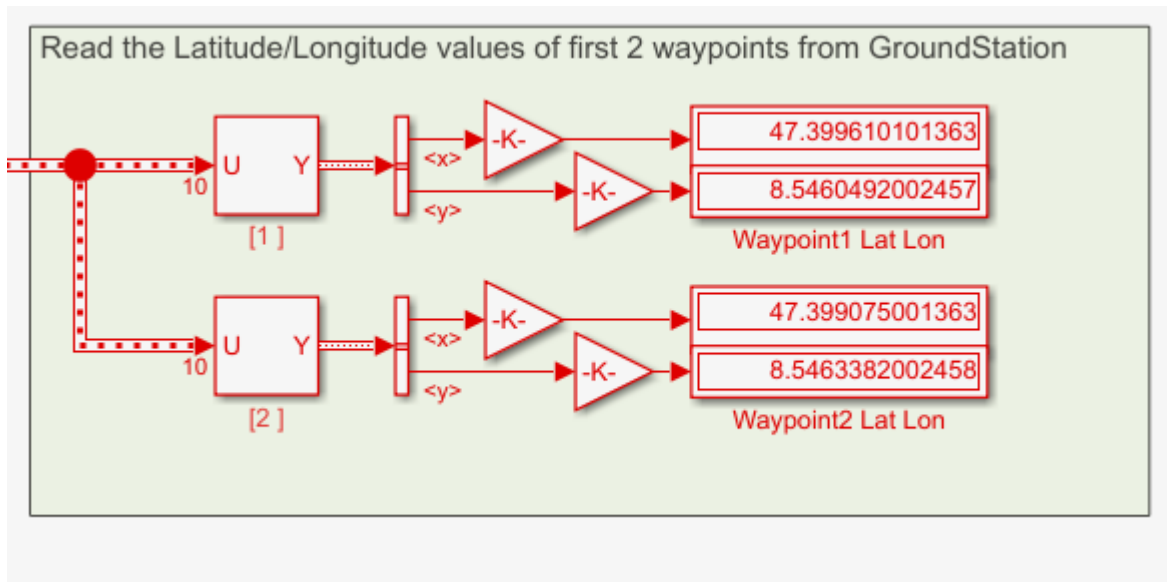
3. Run the Simulink model. The Simulink model sends HEARTBEAT message over MAVLink to QGC and thus establish connection with QGC.

4. Click **Upload** at the top right of QGC interface to upload the mission from QGroundControl.

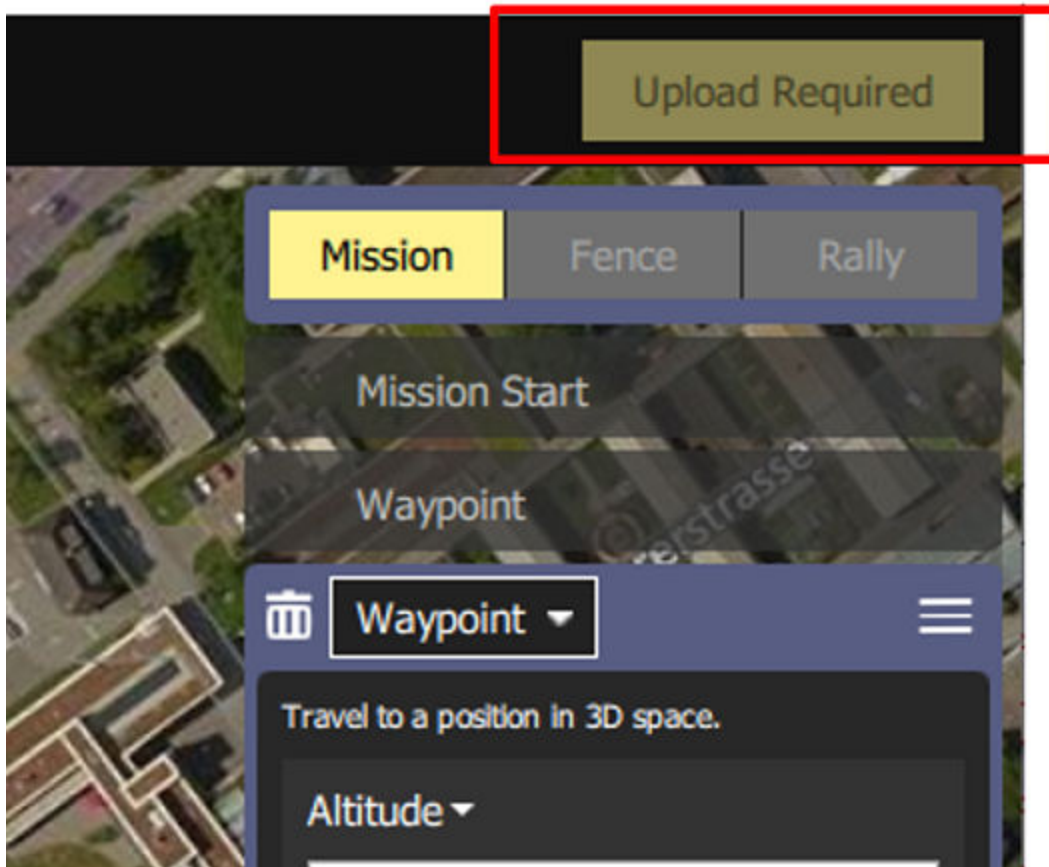


5. Observe that the latitude and longitude values from the first two waypoints of the uploaded mission are being displayed in Simulink.

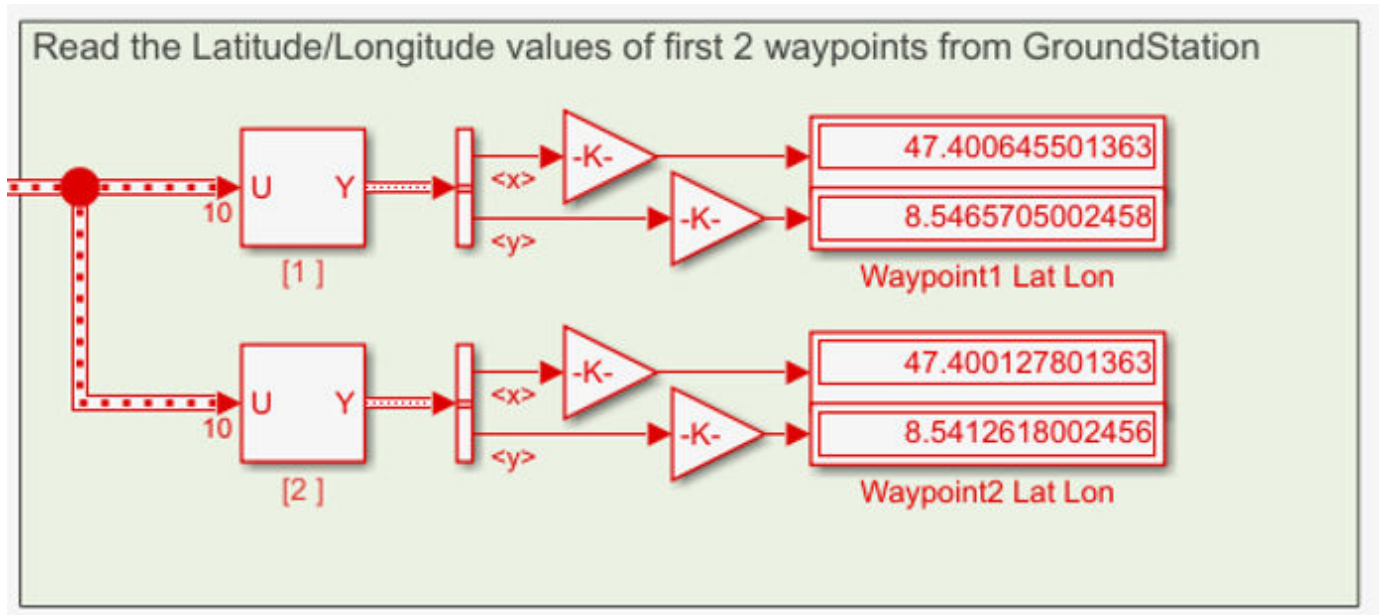




6. Change the waypoint1 and waypoint2 in the QGC by dragging the waypoints to a different location in the plan. Upload the modified mission by clicking **Upload Required**.



7. Observe the modified Latitude/Longitude values for waypoint 1 and 2 in Simulink.



### Modify Parameters in QGC and Send Them to Simulink

When you run the `exampleHelperMAVLinkMissionAndParamProtocol` file in the MATLAB Command Window, a workspace variable `apParams` is created, which is an array of 28 flight parameters.

When you run the Simulink model, it connects to the QGC, and the QGC reads the parameters from Simulink.

The parameters can be visualized and modified in the QGC:

1. Navigate to the **Vehicle Setup** pane in the QGC. Select the **Parameters** tab.
2. In the **Parameters** tab, select **Other** to list all the parameters that the QGC read from Simulink.

QGroundControl

Vehicle Setup

Summary

Firmware

Parameters

Search:  Clear  Show modified only

Standard

Other

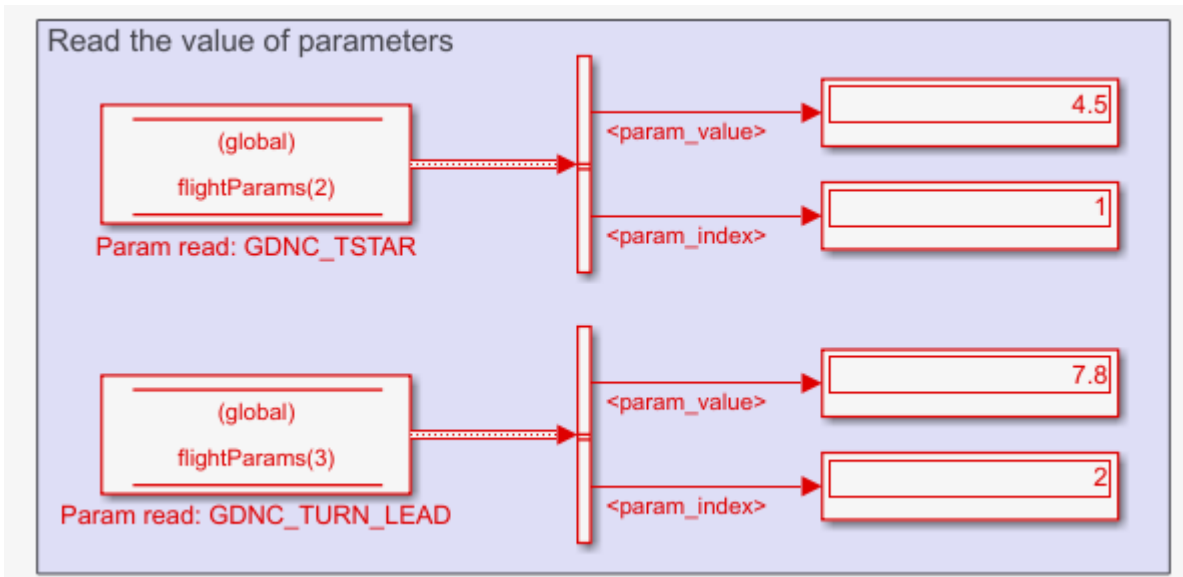
Misc

CMDS_FSID_C_M	0.000
CMDS_RTB	0.000
CMDS_U_C	18.000
CMDS_VIEW_IDX	2.000
CTRL_DE_FFWD	0.000
CTRL_DT_FFWD	0.000
CTRL_HE2THE_D	0.000
CTRL_HE2THE_I	0.000
CTRL_HE2THE_P	0.070
CTRL_ROLL_D	0.300
CTRL_ROLL_I	2.000
CTRL_ROLL_P	10.000
CTRL_TH2DT_D	0.000
CTRL_TH2DT_I	-0.200
CTRL_TH2DT_P	-1.100
CTRL_UHOLD_D	0.000
CTRL_UHOLD_I	0.150
CTRL_UHOLD_P	1.300
CTRL_YAW_D	0.000
CTRL_YAW_I	0.050
CTRL_YAW_P	0.030
GDNC_TSTAR	2.500
GDNC_TURN_LEAD	6.500
SYS_ID	12.000

3. The model displays the values for GDNC\_TSTAR and GDNC\_TURN\_LEAD parameters. Click the GDNC\_TSTAR and GDNC\_TURN\_LEAD parameters and modify their corresponding values in the QGC.

CTRL_YAW_I	0.050
CTRL_YAW_P	0.030
GDNC_TSTAR	4.500
GDNC_TURN_LEAD	7.800
SYS_ID	12.000

4. The QGC writes the values of these modified parameters using the parameter protocol microservice to Simulink. Observe the parameter values being modified in Simulink.



### Other Things to try

The Stateflow charts explained in this example do not implement the following scenario:

- If the communication between the drone and the QGC breaks off at some point and reconnects, the mission protocol upload should resume after the waypoint from which the drone had transmitted data before disconnecting.

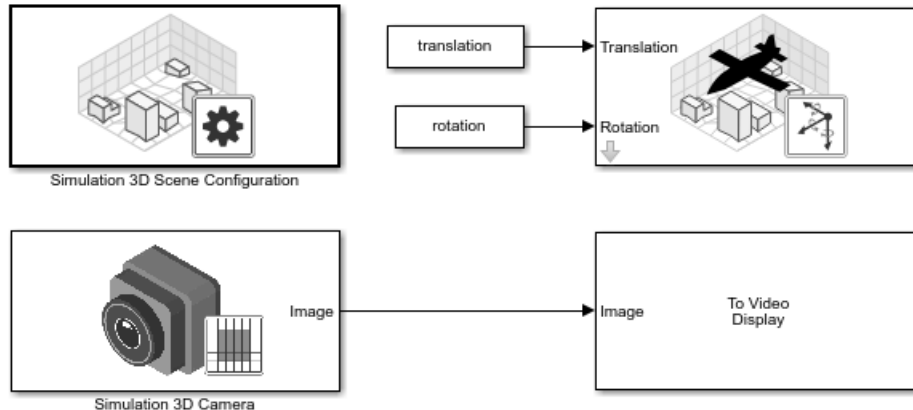
You can modify the Stateflow charts, so that even when the communication snaps, Stateflow remembers the last waypoint transmitted.

# 3D Simulation - User's Guide

---

## Unreal Engine Simulation for Unmanned Aerial Vehicles

UAV Toolbox provides a co-simulation framework that models driving algorithms in Simulink and visualizes their performance in a virtual simulation environment. This environment uses the Unreal Engine from Epic Games.



Simulink blocks related to the simulation environment can be found in the **UAV Toolbox > Simulation 3D** block library. These blocks provide the ability to:

- Configure prebuilt scenes in the simulation environment.
- Place and move UAVs within these scenes.
- Set up camera and lidar sensors on the vehicles.
- Simulate sensor outputs based on the environment around the UAV.
- Obtain ground truth data for semantic segmentation and depth information.

This simulation tool is commonly used to supplement real data when developing, testing, and verifying the performance of UAV flight algorithms. In conjunction with a UAV vehicle model, you can use these blocks to perform realistic closed-loop simulations that encompass the entire UAV flight-control stack, from perception to control.

For more details on the simulation environment, see “How Unreal Engine Simulation for UAVs Works” on page 2-7.

### Unreal Engine Simulation Blocks

To access the **UAV Toolbox > Simulation 3D** library, at the MATLAB® command prompt, enter `uavsim3dlib`.

#### Scenes

To configure a model to co-simulate with the simulation environment, add a Simulation 3D Scene Configuration block to the model. Using this block, you can choose from a prebuilt scene where you can test and visualize your driving algorithms. The following image is from the **US City Block** scene.

The toolbox includes these scenes.

Scene	Description
US City Block	City block with intersections, barriers, and traffic lights

If you have the UAV Toolbox Interface for Unreal Engine Projects support package, then you can modify these scenes or create new ones. For more details, see “Customize Unreal Engine Scenes for UAVs” on page 2-24.

## Vehicles

To define a virtual vehicle in a scene, add a Simulation 3D UAV Vehicle block to your model. Using this block, you can control the movement of the vehicle by supplying the X, Y, and yaw values that define its position and orientation at each time step. The vehicle automatically moves along the ground.

You can also specify the color and type of vehicle. The toolbox includes these vehicle types:

- **Quadrotor**
- **Fixed Wing Aircraft**

## Sensors

You can define virtual sensors and attach them at various positions on the vehicles. The toolbox includes these sensor modeling and configuration blocks.

Block	Description
Simulation 3D Camera	Camera model with lens. Includes parameters for image size, focal length, distortion, and skew.
Simulation 3D Fisheye Camera	Fisheye camera that can be described using the Scaramuzza camera model. Includes parameters for distortion center, image size, and mapping coefficients.
Simulation 3D Lidar	Scanning lidar sensor model. Includes parameters for detection range, resolution, and fields of view.

For more details on choosing a sensor, see “Choose a Sensor for Unreal Engine Simulation” on page 2-13.

## Algorithm Testing and Visualization

UAV Toolbox simulation blocks provide the tools for testing and visualizing path planning, UAV control, and perception algorithms.

### Path Planning and Vehicle Control

You can use the Unreal Engine simulation environment to visualize the motion of a vehicle in a prebuilt scene. This environment provides you with a way to analyze the performance of path planning and vehicle control algorithms. After designing these algorithms in Simulink, you can use the `uavsim3dlib` library to visualize vehicle motion in one of the prebuilt scenes.

### **Perception**

UAV Toolbox provides several blocks for detailed camera and lidar sensor modeling. By mounting these sensors on UAVs within the virtual environment, you can generate synthetic sensor data or sensor detections to test the performance of your sensor models against perception algorithms.

### **Closed-Loop Systems**

After you design and test a perception system within the simulation environment, you can then use it to drive a control system that actually steers a vehicle. In this case, rather than manually set up a trajectory, the UAV uses the perception system to fly itself. By combining perception and control into a closed-loop system in the 3D simulation environment, you can develop and test more complex algorithms, such as automated delivery.

### **See Also**



# Unreal Engine Simulation Environment Requirements and Limitations

UAV Toolbox provides an interface to a simulation environment that is visualized using the Unreal Engine from Epic Games. Version 4.23 of this visualization engine comes installed with UAV Toolbox. When simulating in this environment, keep these requirements and limitations in mind.

## Software Requirements

- Windows® 64-bit platform
- Visual Studio® 2017 with minimum Version 15.9
- Microsoft® DirectX® — If this software is not already installed on your machine and you try to simulate in the 3D environment, UAV Toolbox prompts you to install it. Once you install the software, you must restart the simulation.

In you are customizing scenes, verify that your Unreal Engine project is compatible with the Unreal Engine version supported by your MATLAB release.

MATLAB Release	Unreal Engine Version
R2020b	4.23

**Note** Mac and Linux® platforms are not supported.

## Minimum Hardware Requirements

- Graphics card (GPU) — Virtual reality-ready with 8 GB of on-board RAM
- Processor (CPU) — 2.60 GHz
- Memory (RAM) — 12 GB

## Limitations

The Unreal Engine simulation environment blocks do not support:

- Code generation
- Model reference
- Multiple instances of the Simulation 3D Scene Configuration block
- Multiple instances of the 3D simulation environment
- Rapid accelerator mode

In addition, when using these blocks in a closed-loop simulation, all Unreal Engine simulation environment blocks must be in the same subsystem.

## See Also

Simulation 3D Scene Configuration

### **More About**

- “Unreal Engine Simulation for Unmanned Aerial Vehicles” on page 2-2
- “How Unreal Engine Simulation for UAVs Works” on page 2-7

### **External Websites**

- Unreal Engine

## How Unreal Engine Simulation for UAVs Works

UAV Toolbox provides a co-simulation framework that you can use to model UAV algorithms in Simulink and visualize their performance in a virtual simulation environment. This environment uses the Unreal Engine by Epic Games.

Understanding how this simulation environment works can help you troubleshoot issues and customize your models.

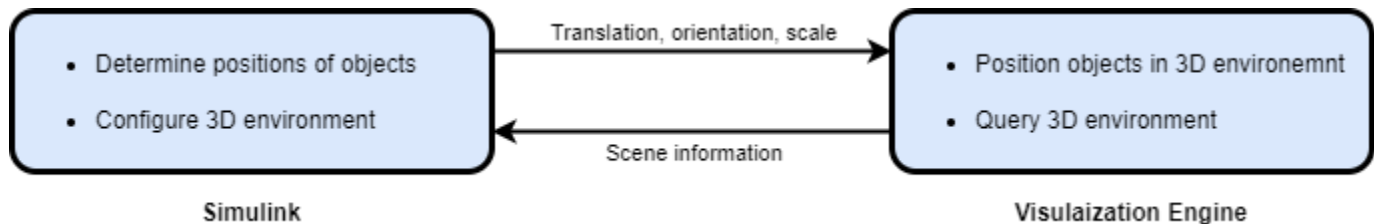
### Communication with 3D Simulation Environment

When you use UAV Toolbox to run your algorithms, Simulink co-simulates the algorithms in the visualization engine.

In the Simulink environment, UAV Toolbox:

- Configures the visualization environment, specifically the ray tracing, scene capture from cameras, and initial object positions
- Determines the next position of the objects by using the simulation environment feedback

The diagram summarizes the communication between Simulink and the visualization engine.



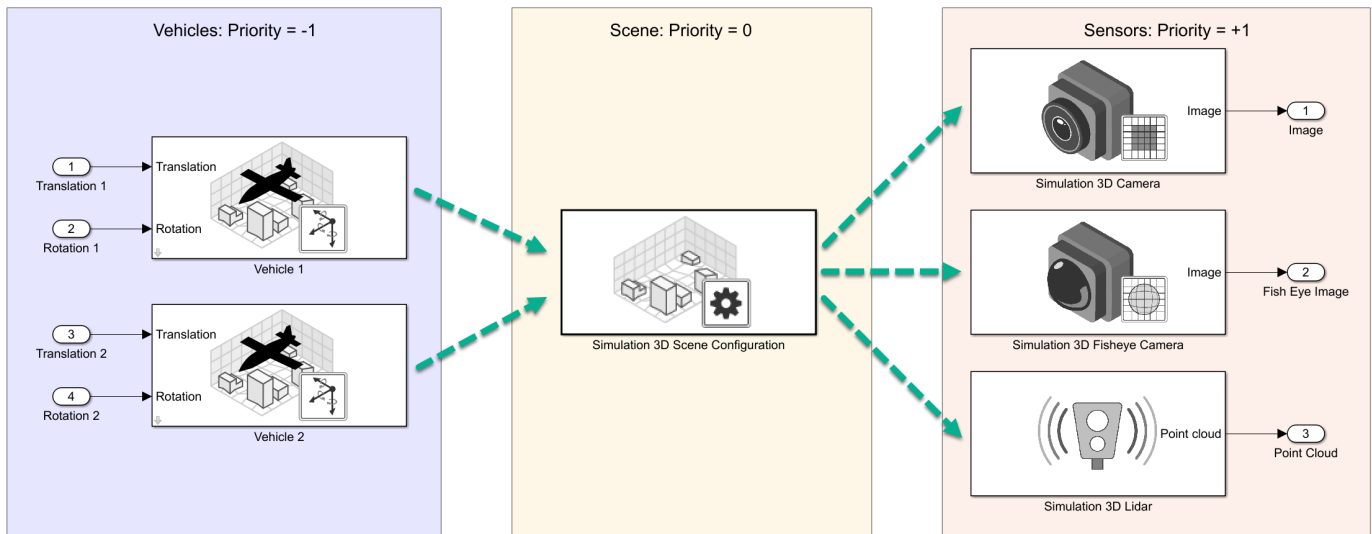
### Block Execution Order

During simulation, the Unreal Engine simulation blocks follow a specific execution order:

- 1 The Simulation 3D UAV Vehicle blocks initialize the vehicles and send their **Translation**, and **Rotation** signal data to the Simulation 3D Scene Configuration block.
- 2 The Simulation 3D Scene Configuration block receives the vehicle data and sends it to the sensor blocks.
- 3 The sensor blocks receive the vehicle data and use it to accurately locate and visualize the vehicles.

The **Priority** property of the blocks controls this execution order. To access this property for any block, right-click the block, select **Properties**, and click the **General** tab. By default, Simulation 3D UAV Vehicle blocks have a priority of **-1**, Simulation 3D Scene Configuration blocks have a priority of **0**, and sensor blocks have a priority of **1**.

The diagram shows this execution order.



If your sensors are not detecting vehicles in the scene, it is possible that the Unreal Engine simulation blocks are executing out of order. Try updating the execution order and simulating again. For more details on execution order, see “Control and Display Execution Order” (Simulink).

Also be sure that all 3D simulation blocks are located in the same subsystem. Even if the blocks have the correct **Priority** settings, if they are located in different subsystems, they still might execute out of order.

## See Also

### More About

- “Unreal Engine Simulation for Unmanned Aerial Vehicles” on page 2-2
- “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-5
- “Choose a Sensor for Unreal Engine Simulation” on page 2-13
- “Coordinate Systems for Unreal Engine Simulation in UAV Toolbox” on page 2-9

## Coordinate Systems for Unreal Engine Simulation in UAV Toolbox

UAV Toolbox enables you to simulate your driving algorithms in a virtual environment that uses the Unreal Engine from Epic Games. In general, the coordinate systems used in this environment follow the conventions described in “Coordinate Systems for Modeling” (Aerospace Toolbox). However, when simulating in this environment, it is important to be aware of the specific differences and implementation details of the coordinate systems.

UAV Toolbox uses these coordinate systems to calculate the vehicle dynamics and position objects in the Unreal Engine visualization environment.

Environment	Description	Coordinate Systems
UAV vehicle dynamics in Simulink	The <i>right-hand rule</i> establishes the X-Y-Z sequence and rotation of the coordinate axes used to calculate the vehicle dynamics. The UAV Toolbox interface to the Unreal Engine simulation environment uses the <i>right-handed</i> (RH) <i>Cartesian</i> coordinate systems: <ul style="list-style-type: none"> <li>• Earth-fixed (inertial)</li> <li>• Vehicle</li> </ul>	“Earth-Fixed (Inertial) Coordinate System” on page 2-9  “Body (Non-Inertial) Coordinate System” on page 2-9
Unreal Engine visualization	To position objects and query the Unreal Engine visualization environment, the UAV Toolbox uses a world coordinate system.	“Unreal Engine World Coordinate System” on page 2-11

### Earth-Fixed (Inertial) Coordinate System

The earth-fixed coordinate system ( $X_E$ ,  $Y_E$ ,  $Z_E$ ) axes are fixed in an inertial reference frame. The inertial reference frame has zero linear and angular acceleration and zero angular velocity. In Newtonian physics, the earth is an inertial reference.

Axis	Description
$X_E$	The $X_E$ axis is in the forward direction of the vehicle.
$Y_E$	The $X_E$ and $Y_E$ axes are parallel to the ground plane. The ground plane is a horizontal plane normal to the gravitational vector.
$Z_E$	In the Z-up orientation, the positive $Z_E$ axis points upward.  In the Z-down orientation, the positive $Z_E$ axis points downward.

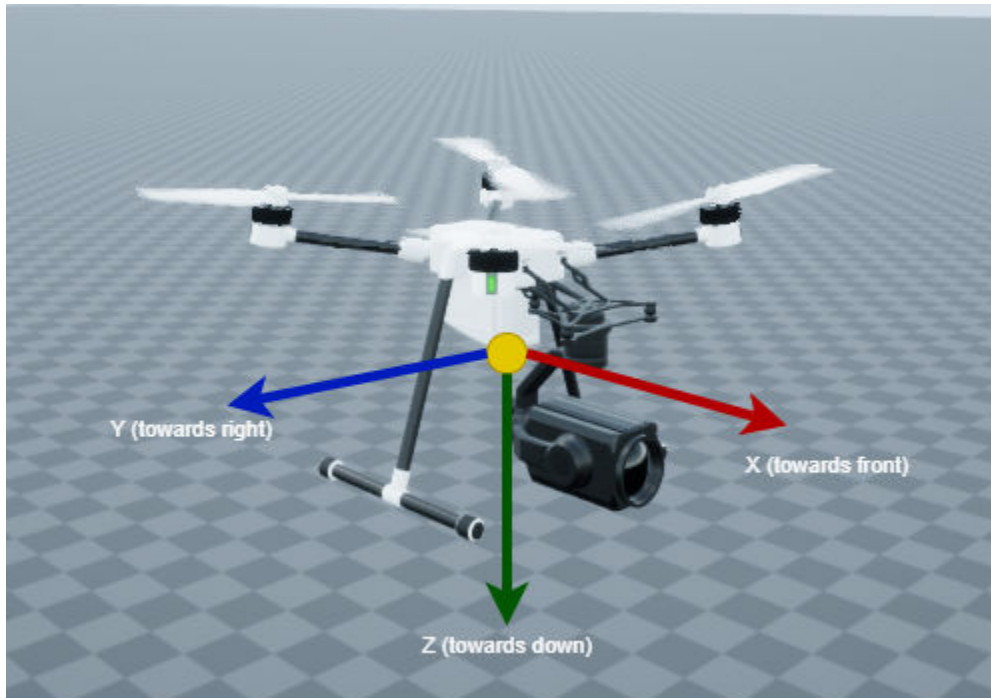
### Body (Non-Inertial) Coordinate System

Modeling aircraft and spacecraft are simplest if you use a coordinate system fixed in the body itself. In the case of aircraft, the forward direction is modified by the presence of wind, and the craft's motion through the air is not the same as its motion relative to the ground. The non-inertial body coordinate system is fixed in both origin and orientation to the moving craft. The craft is assumed to be rigid. The orientation of the body coordinate axes is fixed in the shape of body.

- The  $x$ -axis points through the nose of the craft.
- The  $y$ -axis points to the right of the  $x$ -axis (facing in the pilot's direction of view), perpendicular to the  $x$ -axis.
- The  $z$ -axis points down through the bottom of the craft, perpendicular to the  $x$ - $y$  plane and satisfying the RH rule.

### Translational Degrees of Freedom

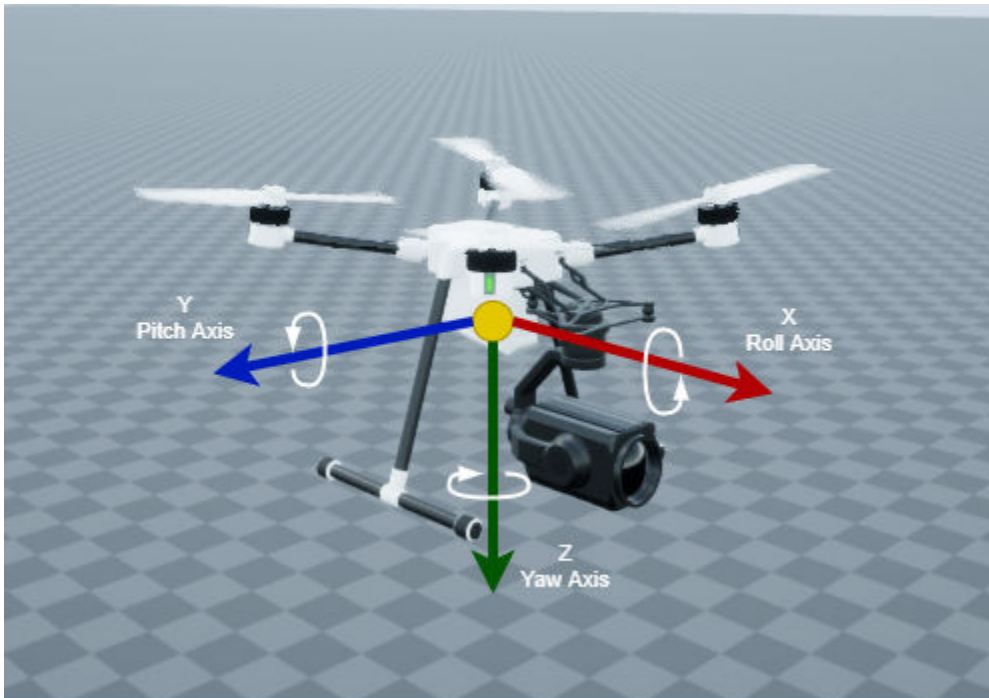
Translations are defined by moving along these axes by distances  $x$ ,  $y$ , and  $z$  from the origin.



### Rotational Degrees of Freedom

Rotations are defined by the Euler angles  $P$ ,  $Q$ ,  $R$  or  $\Phi$ ,  $\Theta$ ,  $\Psi$ . They are

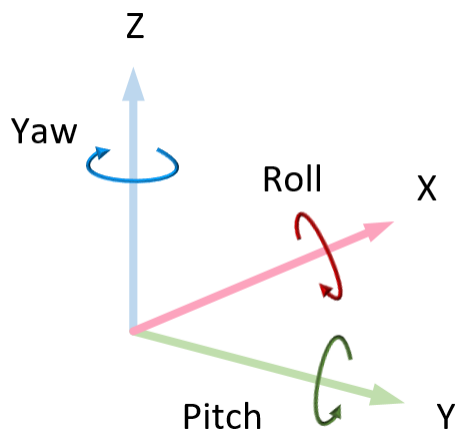
- $P$  or  $\Phi$ : Roll about the  $x$ -axis
- $Q$  or  $\Theta$ : Pitch about the  $y$ -axis
- $R$  or  $\Psi$ : Yaw about the  $z$ -axis



Unless otherwise specified, by default the software uses ZYX rotation order for Euler angles.

## Unreal Engine World Coordinate System

The Unreal Engine environment uses a world coordinate system with axes that are fixed in the inertial reference frame.



Axis	Description
X	Forward direction of the vehicle Roll — Right-handed rotation about X-axis

<b>Axis</b>	<b>Description</b>
Y	Extends to the right of the vehicle, parallel to the ground plane Pitch — Right-handed rotation about Y-axis
Z	Extends upwards Yaw — Left-handed rotation about Z-axis

## **See Also**

**Fixed Wing Aircraft | Quadrotor**

## **More About**

- “How Unreal Engine Simulation for UAVs Works” on page 2-7
- “Simulate Simple Flight Scenario and Sensor in Unreal Engine Environment” on page 2-14



## Choose a Sensor for Unreal Engine Simulation

In UAV Toolbox, you can obtain high-fidelity sensor data from a virtual environment. This environment is rendered using the Unreal Engine from Epic Games.

The table summarizes the sensor blocks that you can simulate in this environment.

Sensor Block	Description	Visualization	Example
Simulation 3D Camera	<ul style="list-style-type: none"> <li>• Camera with lens that is based on the ideal pinhole camera. See “What Is Camera Calibration?” (Computer Vision Toolbox)</li> <li>• Includes parameters for image size, focal length, distortion, and skew</li> <li>• Includes options to output ground truth for depth estimation and semantic segmentation</li> </ul>	Display camera images by using a Video Viewer or To Video Display block. Sample visualization:	“Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” on page 2-19
		Display depth maps by using a Video Viewer or To Video Display block. Sample visualization:	“Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” on page 2-19
		Display semantic segmentation maps by using a Video Viewer or To Video Display block. Sample visualization:	“Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” on page 2-19
Simulation 3D Fisheye Camera	<ul style="list-style-type: none"> <li>• Fisheye camera that can be described using the Scaramuzza camera model. See “Fisheye Calibration Basics” (Computer Vision Toolbox)</li> <li>• Includes parameters for distortion center, image size, and mapping coefficients</li> </ul>	Display camera images by using a Video Viewer or To Video Display block. Sample visualization:	“Simulate Simple Flight Scenario and Sensor in Unreal Engine Environment” on page 2-14
Simulation 3D Lidar	<ul style="list-style-type: none"> <li>• Scanning lidar sensor model</li> <li>• Includes parameters for detection range, resolution, and fields of view</li> </ul>	Display point cloud data by using <code>pcplayer</code> within a MATLAB Function block. Sample visualization:	“UAV Package Delivery” on page 1-46

### See Also

Simulation 3D Scene Configuration

## Simulate Simple Flight Scenario and Sensor in Unreal Engine Environment

UAV Toolbox™ provides blocks for visualizing sensors in a simulation environment that uses the Unreal Engine® from Epic Games®. This model simulates a simple flight scenario in a prebuilt scene and captures data from the scene using a fisheye camera sensor. Use this model to learn the basics of configuring and simulating scenes, vehicles, and sensors. For more background on the Unreal Engine simulation environment, see “Unreal Engine Simulation for Unmanned Aerial Vehicles” on page 2-2.

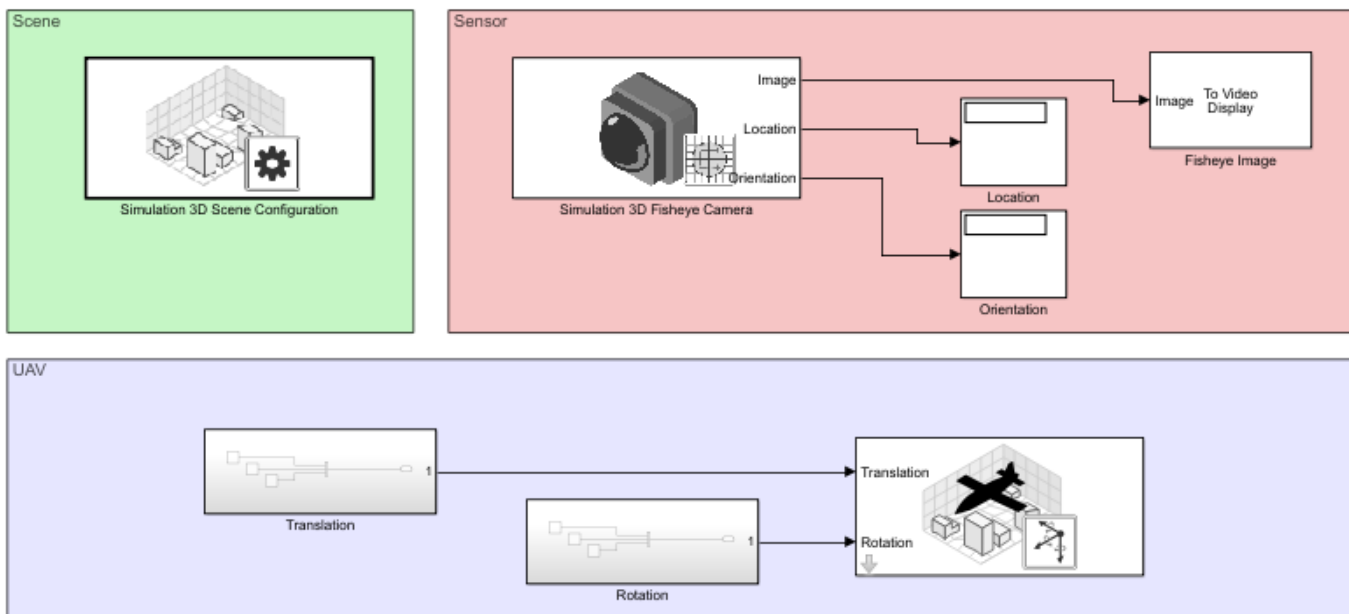
### Model Overview

The model consists of these main components:

- Scene - A Simulation 3D Scene Configuration block configures the scene in which you simulate.
- Vehicles - A Simulation 3D UAV Vehicle blocks configures the quadrotor within the scene and specifies its trajectory.
- Sensor - A Simulation 3D Fisheye Camera configures the mounting position and parameters of the fisheye camera used to capture simulation data. A Video Viewer (Computer Vision Toolbox) block visualizes the simulation output of this sensor.

You can open the model using the following command.

```
open_system("uav_simple_flight_model.slx")
```

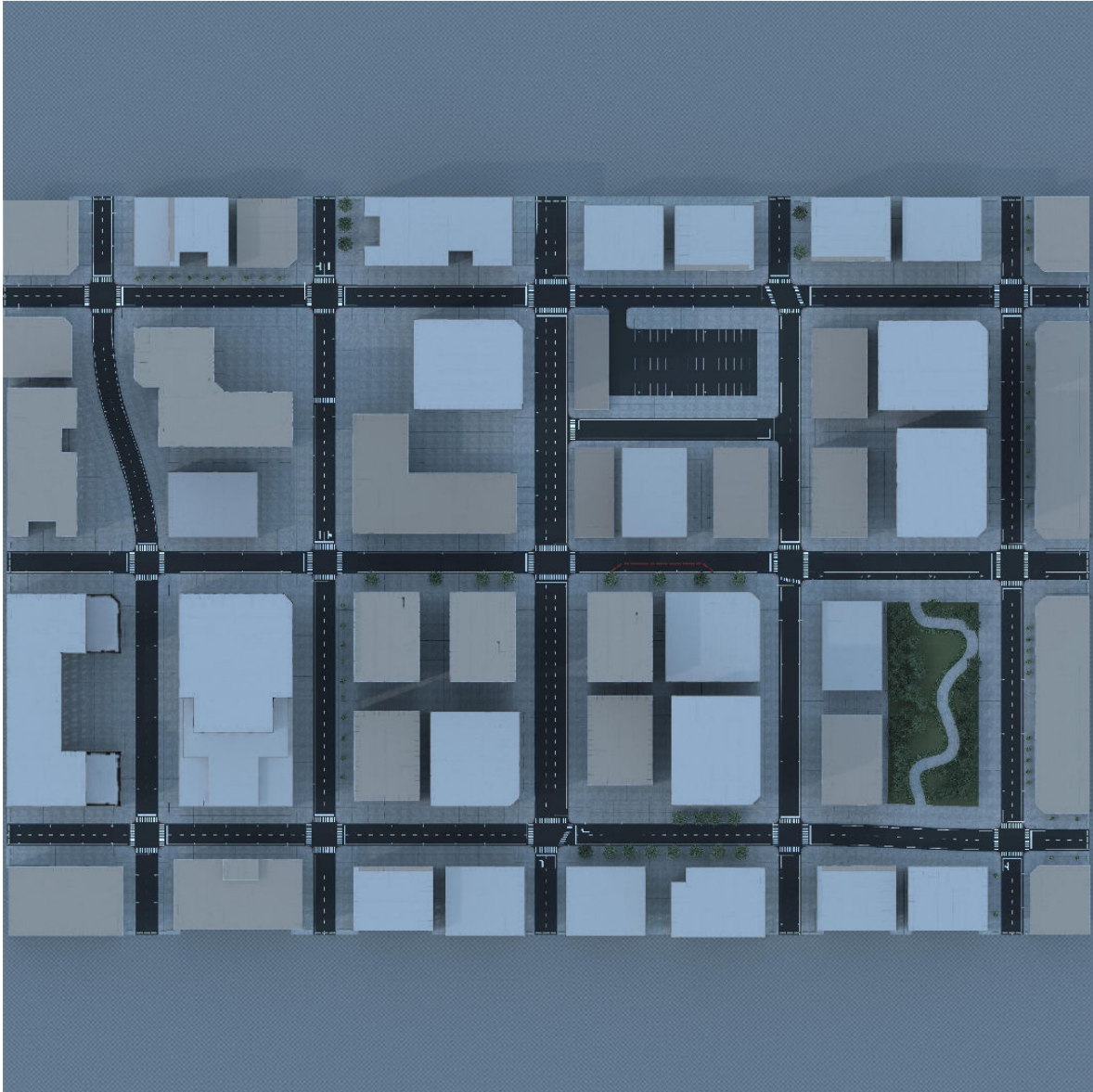


### Inspect Scene

In the Simulation 3D Scene Configuration block, the Scene name parameter determines the scene where the simulation takes place. This model uses the US City Block scene. To explore a scene, you can open the 2D image corresponding to the Unreal Engine scene.

```
imshow('USCityBlock.jpg',...
       'XData', [-242.998152046784, 200.198152046784],...)
```

```
'YData', [-215.598152046784, 227.598152046784]);
set(gca, 'YDir', 'normal')
```



The Scene view parameter of this block determines the view from which the Unreal Engine window displays the scene. In this block, Scene view is set to the root of the scene (the scene origin), select root. You can also change the scene view to the quadrotor UAV.

### **Inspect Vehicle**

The Simulation 3D UAV Vehicle block models the quadcopter, named Quadrotor1, in the scenario. During simulation, the quadrotor flies one complete circle with a radius of 5m and elevation of 1.5m

around the center of the scene. The viewpoint of the quadrotor yaw oscillates yaw from left to right in the direction of travel.

To create more realistic trajectories, you can obtain waypoints from a scene interactively and specify these waypoints as inputs to the Simulation 3D UAV Vehicle block. See [Select Waypoints for Unreal Engine Simulation](#).

### Inspect Sensor

The Simulation 3D Fisheye Camera block models the sensor used in the scenario. Open this block and inspect its parameters.

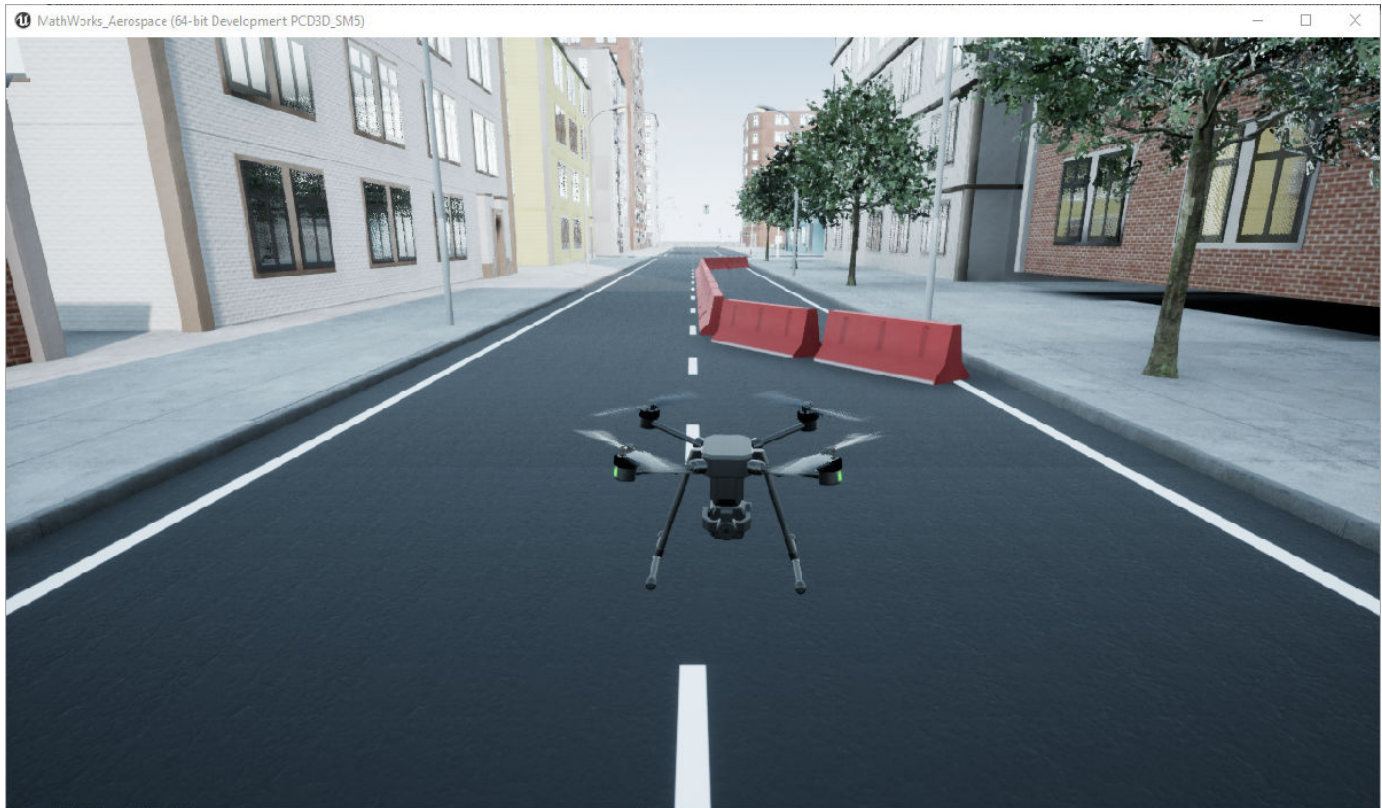
- The **Mounting** tab contains parameters that determine the mounting location of the sensor. The fisheye camera sensor is mounted to the center of the roof of the ego vehicle.
- The **Parameters** tab contains the intrinsic camera parameters of a fisheye camera. These parameters are set to their default values.
- The **Ground Truth** tab contains a parameter for outputting the location and orientation of the sensor in meters and radians. In this model, the block outputs these values so you can see how they change during simulation.

The block outputs images captured from the simulation. During simulation, the Video Viewer block displays these images.



### **Simulate Model**

Simulate the model. When the simulation begins, it can take a few seconds for the visualization engine to initialize, especially when you are running it for the first time. The MathWorks\_Aerospace window shows a view of the scene in the Unreal Engine environment.



To change the view of the scene during simulation, use the numbers 1-9 on the numeric keypad. For a bird's-eye view of the scene, press 0.

After simulating the model, try modifying the intrinsic camera parameters and observe the effects on simulation. You can also change the type of sensor block. For example, try substituting the 3D Simulation Fisheye Camera with a 3D Simulation Camera block. For more details on the available sensor blocks, see “Choose a Sensor for Unreal Engine Simulation” on page 2-13.

### **See Also**

[Simulation 3D Camera](#) | [Simulation 3D Scene Configuration](#) | [Simulation 3D UAV Vehicle](#)

# Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation

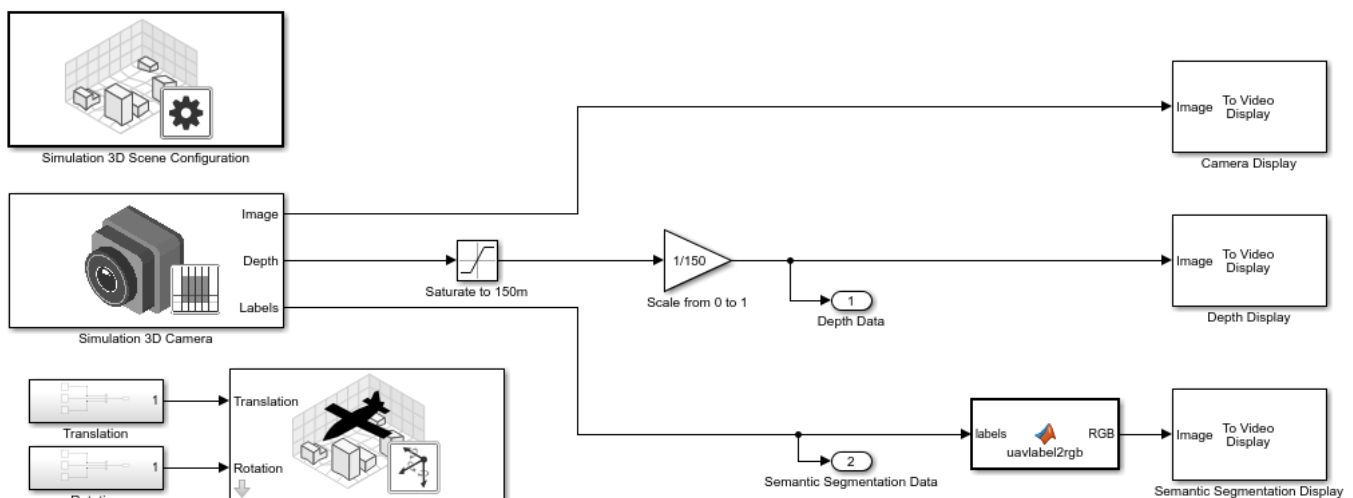
This example shows how to visualize depth and semantic segmentation data captured from a camera sensor in a simulation environment. This environment is rendered using the Unreal Engine® from Epic Games®.

You can use depth visualizations to validate depth estimation algorithms for your sensors. You can use semantic segmentation visualizations to analyze the classification scheme used for generating synthetic semantic segmentation data from the Unreal Engine environment.

## Model Setup

The model used in this example simulates a vehicle driving in a city scene.

- A Simulation 3D Scene Configuration block sets up simulation with the US City Block scene.
- A Simulation 3D UAV Vehicle block specifies the driving route of the vehicle.
- A Simulation 3D Camera block mounted to the quadrotor captures data from the flight. This block outputs the camera, depth, and semantic segmentation displays by using To Video Display (Computer Vision Toolbox) (Computer Vision Toolbox) blocks.



## Depth Visualization

A depth map is a grayscale representation of camera sensor output. These maps visualize camera images in grayscale, with brighter pixels indicating objects that are farther away from the sensor. You can use depth maps to validate depth estimation algorithms for your sensors.

The **Depth** port of the Simulation 3D Camera block outputs a depth map of values in the range of 0 to 1000 meters. In this model, for better visibility, a Saturation block saturates the depth output to a maximum of 150 meters. Then, a Gain block scales the depth map to the range [0, 1] so that the To Video Display block can visualize the depth map in grayscale.

## Semantic Segmentation Visualization

*Semantic segmentation* describes the process of associating each pixel of an image with a class label, such as *road*, *building*, or *traffic sign*. In the 3D simulation environment, you generate synthetic

semantic segmentation data according to a label classification scheme. You can then use these labels to train a neural network for UAV flight applications, such as landing zone identification. By visualizing the semantic segmentation data, you can verify your classification scheme.

The **Labels** port of the Simulation 3D Camera block outputs a set of labels for each pixel in the output camera image. Each label corresponds to an object class. For example, in the default classification scheme used by the block, 1 corresponds to buildings. A label of 0 refers to objects of an unknown class and appears as black. For a complete list of label IDs and their corresponding object descriptions, see the **Labels** port description on the Simulation 3D Camera block reference page.

The MATLAB® Function block uses the `label2rgb` function to convert the labels to a matrix of RGB triplets for visualization. The colormap is based on the colors used in the CamVid dataset, as shown in the “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox) example. The colors are mapped to the predefined label IDs used in the default Unreal Engine simulation scenes. The helper function `sim3dColormap` defines the colormap. Inspect these colormap values.

```
open sim3dColormap.m
```

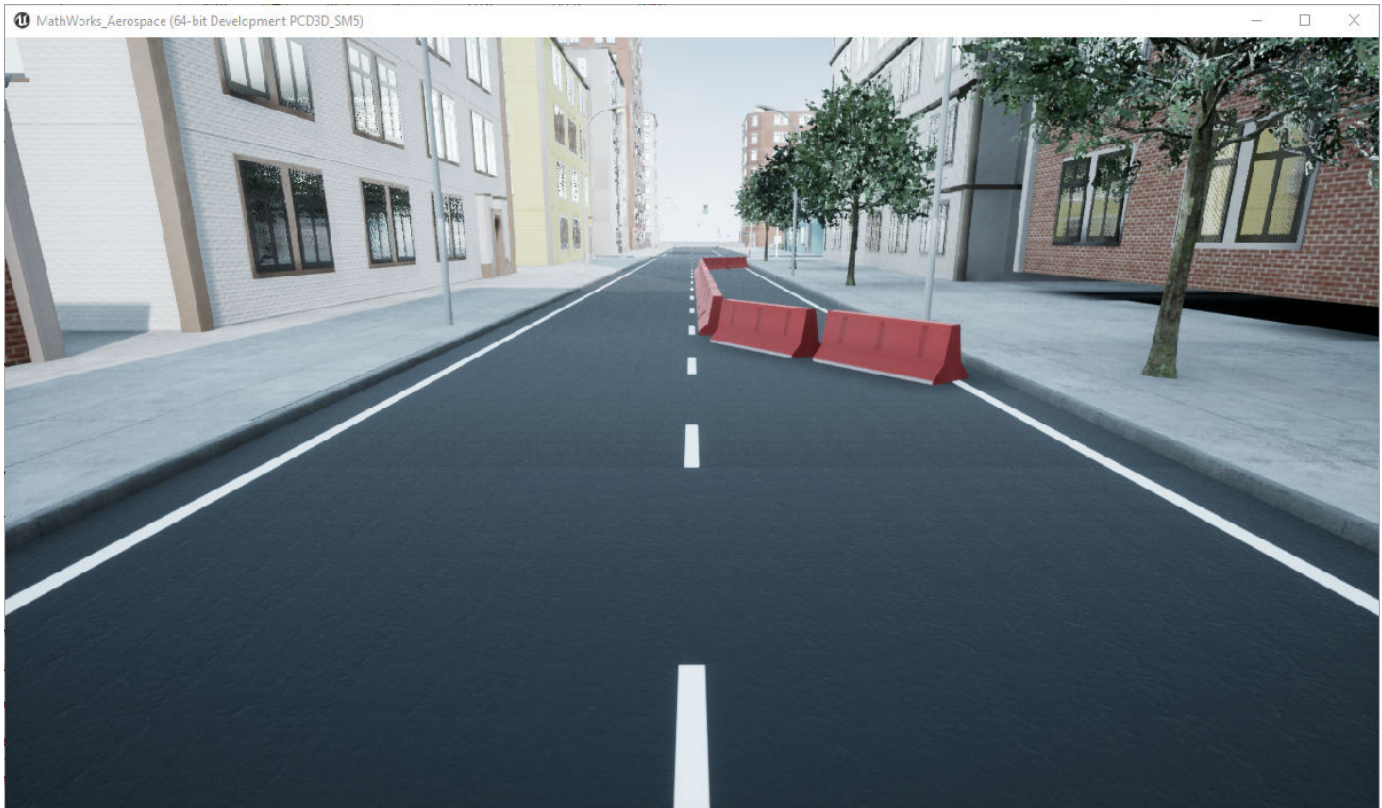
### **Model Simulation**

Run the model.

```
sim('uav_ue4_depth_imaging.slx');
```

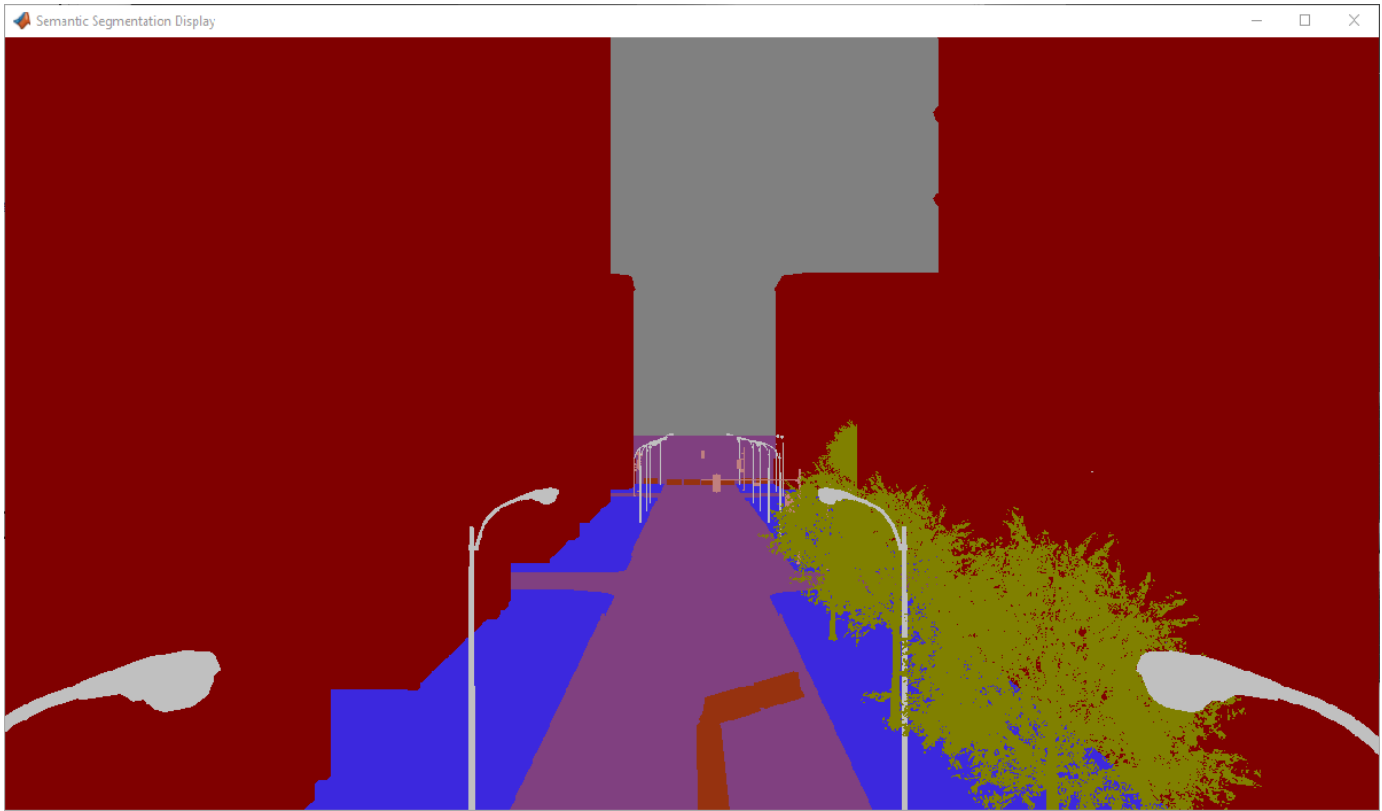
When the simulation begins, it can take a few seconds for the visualization engine to initialize, especially when you are running it for the first time. The `MathWorks_Aerospace` window displays the scene from the scene origin. In this scene, the quadrotor UAV flies a short distance down one city block.





The Camera Display, Depth Display, and Semantic Segmentation Display blocks display the outputs from the camera sensor.





To change the visualization range of the output depth data, try updating the values in the Saturation and Gain blocks.

To change the semantic segmentation colors, try modifying the color values defined in the `sim3dColormap` function. Alternatively, in the `uavLabel2rgb` MATLAB Function block, try replacing the input colormap with your own colormap or a predefined colormap. See `colormap`.

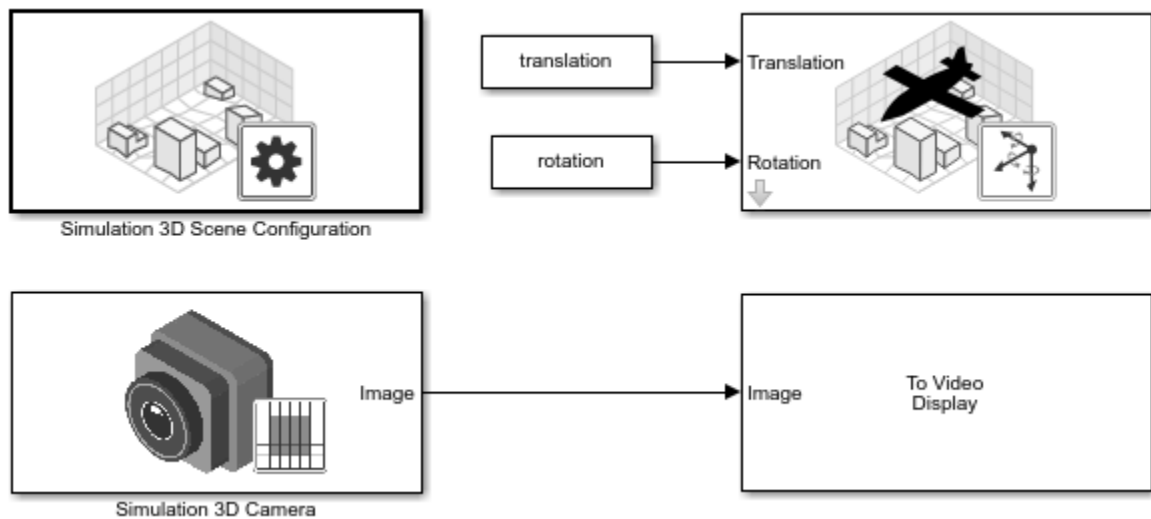
### See Also

[Simulation 3D Camera](#) | [Simulation 3D Scene Configuration](#) | [Simulation 3D UAV Vehicle](#)

## Customize Unreal Engine Scenes for UAVs

UAV Toolbox comes installed with prebuilt scenes in which to simulate and visualize the performance of UAV algorithms modeled in Simulink. These scenes are visualized using the Unreal Engine from Epic Games. By using the Unreal® Editor and the UAV Toolbox Interface for Unreal Engine Projects, you can customize these scenes. You can also use the Unreal Editor and the support package to simulate within scenes from your own custom project.

With custom scenes, you can co-simulate in both Simulink and the Unreal Editor so that you can modify your scenes between simulation runs. You can also package your scenes into an executable file so that you do not have to open the editor to simulate with these scenes.



To customize Unreal Engine scenes for UAV flight simulations, follow these steps:

- 1 "Install Support Package for Customizing Scenes" on page 2-25
- 2 "Customize Unreal Engine Scenes Using Simulink and Unreal Editor" on page 2-28
- 3 "Package Custom Scenes into Executable" on page 2-33

### See Also

Simulation 3D Scene Configuration

## Install Support Package for Customizing Scenes

To customize scenes in the Unreal Editor and use them in Simulink, you must install the UAV Toolbox Interface for Unreal Engine Projects.

### Verify Software and Hardware Requirements

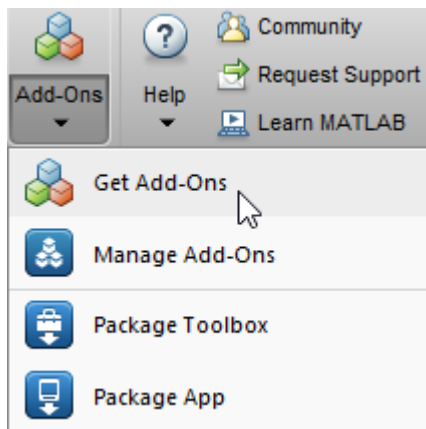
Before installing the support package, make sure that your environment meets the minimum software and hardware requirements described in “Unreal Engine Simulation Environment Requirements and Limitations” on page 2-5. In particular, verify that you have version 15.9 or higher of Visual Studio 2017 installed. This software is required for using the Unreal Editor to customize scenes.

In addition, verify that your project is compatible with Unreal Engine, Version 4.23. If your project was created with an older version of the Unreal Editor, upgrade your project to version 4.23.

### Install Support Package

To install the UAV Toolbox Interface for Unreal Engine Projects support package, follow these steps:

- 1 On the MATLAB **Home** tab, in the **Environment** section, select **Add-Ons > Get Add-Ons**.



- 2 In the Add-On Explorer window, search for the UAV Toolbox Interface for Unreal Engine Projects support package. Click **Install**.

---

**Note** You must have write permission for the installation folder.

---

### Set Up Scene Customization Using Support Package

The UAV Toolbox Interface for Unreal Engine Projects support package includes these components:

- An Unreal Engine project file (`MathWorks_Aerospace.uproject`) and its associated files. This project file includes editable versions of the prebuilt 3D scenes that you can select from the **Scene source** parameter of the Simulation 3D Scene Configuration block.
- Two plugin files, `MathWorkSimulation.uplugin` and `MathWorksUAV.uplugin`. This plugin establishes the connection between Simulink and the Unreal Editor and is required for co-simulation.

To set up scene customization, you must copy this project and plugin onto your local machine.

### Copy Project to Local Folder

Copy the mwas project folder into a folder on your local machine.

- 1 Specify the path to the support package folder that contains the project. If you previously downloaded the support package, specify only the latest download path, as shown here. Also specify a local folder destination in which to copy the project. This code specifies a local folder of C:\Local.

```
supportPackageFolder = fullfile( ...  
    matlabshared.supportpkg.getSupportPackageRoot, ...  
    "toolbox", "uav", "spkg", "uavunrealengine");  
localFolder = "C:\Local";
```

- 2 Copy the MathWorks\_Aerospace project from the support package folder to the local destination folder.

```
projectFolderName = "mwas";  
projectSupportPackageFolder = fullfile(supportPackageFolder, projectFolderName);  
projectLocalFolder = fullfile(localFolder, projectFolderName);  
if ~exist(projectLocalFolder, "dir")  
    copyfile(projectSupportPackageFolder, projectLocalFolder);  
end
```

The MathWorks\_Aerospace.uproject file and all of its supporting files are now located in a folder named mwas within the specified local folder. For example: C:\Local\mwas.

### Copy Plugin to Unreal Editor

Copy the MathWorksSimulation and MathWorksUAV plugins into the Plugins folder of your Unreal Engine installation.

- 1 Specify the local folder containing your Unreal Engine installation. This code shows the default installation location for the editor on a Windows machine.

```
ueInstallFolder = "C:\Program Files\Epic Games\UE_4.23";
```

- 2 Copy the plugins from the support package into the Plugins folder.

```
supportPackageFolder = fullfile( ...  
    matlabshared.supportpkg.getSupportPackageRoot, ...  
    "toolbox", "uav", "spkg", "uavunrealengine");  
  
mwSimPluginName = "MathWorksSimulation.uplugin";  
mwSimPluginFolder = fullfile(supportPackageFolder, "mwas_plugins", "MathWorksSimulation");  
mwUAVPluginName = "MathWorksUAV.uplugin";  
mwUAVPluginFolder = fullfile(supportPackageFolder, "mwas_plugins", "MathWorksUAV");  
  
uePluginFolder = fullfile(ueInstallFolder, "Engine", "Plugins");  
uePluginDestination = fullfile(uePluginFolder, "Marketplace", "MathWorks");  
  
cd(uePluginFolder)  
foundPlugins = [dir("**/" + mwSimPluginName) dir("**/" + mwUAVPluginName)];  
  
if ~isempty(foundPlugins)  
    numPlugins = size(foundPlugins, 1);  
    msg2 = cell(1, numPlugins);
```

```
pluginCell = struct2cell(foundPlugins);

msg1 = "Plugin(s) already exist here:" + newline + newline;
for n = 1:numPlugins
    msg2{n} = "      " + pluginCell{2,n} + newline;
end
msg3 = newline + "Please remove plugin folder(s) and try again.";
msg = msg1 + msg2 + msg3;
warning(msg);
else
    copyfile(mwSimPluginFolder, fullfile(uePluginDestination,"MathWorksSimulation"));
    disp("Successfully copied MathWorksSimulation plugin to UE4 engine plugins!")
    copyfile(mwUAVPluginFolder, fullfile(uePluginDestination,"MathWorksUAV"));
    disp("Successfully copied MathWorksUAV plugin to UE4 engine plugins!")
end
```

After you install and set up the support package, you can begin customizing scenes. See “Customize Unreal Engine Scenes Using Simulink and Unreal Editor” on page 2-28.

## See Also

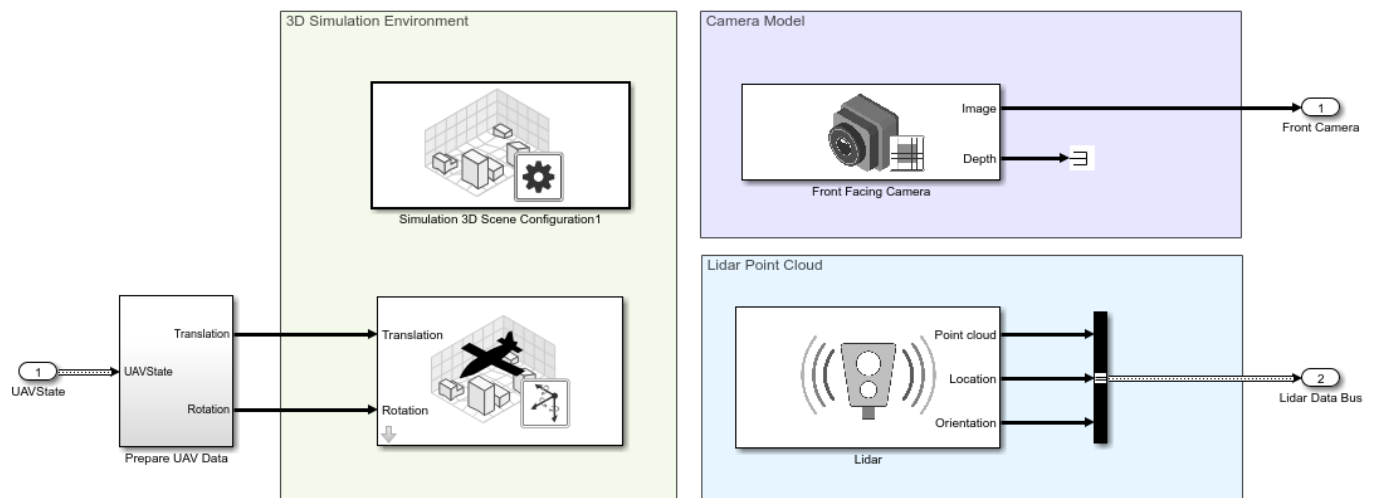
## Customize Unreal Engine Scenes Using Simulink and Unreal Editor

After you install the UAV Toolbox Interface for Unreal Engine Projects support package as described in “Install Support Package for Customizing Scenes” on page 2-25, you can simulate in custom scenes simultaneously from both the Unreal Editor and Simulink. By using this co-simulation framework, you can add vehicles and sensors to a Simulink model and then run this simulation in your custom scene.

### Open Unreal Editor from Simulink

If you open your Unreal project file directly in the Unreal Editor, Simulink is unable to establish a connection with the editor. To establish this connection, you must open your project from a Simulink model.

- 1 Open a Simulink model configured to simulate in the 3D environment. At a minimum, the model must contain a Simulation 3D Scene Configuration block. For example, open a simple model that simulates a UAV flying in a US city block. This model here is the photo-realistic simulation variant from the “UAV Package Delivery” on page 1-46 example.



- 2 In the Simulation 3D Scene Configuration block of this model, set the **Scene source** parameter to Unreal Editor.
- 3 In the **Project** parameter, browse for the project file that contains the scenes that you want to customize.

For example, this sample path specifies the MathWorks\_Aerospace project that comes installed with the UAV Toolbox Interface for Unreal Engine Projects support package.

```
C:\Local\mwas\MathWorks_Aerospace.uproject
```

This sample path specifies a custom project.

```
Z:\UnrealProjects\myProject\myProject.uproject
```

- 4 Click **Open Unreal Editor**. The Unreal Editor opens and loads a scene from your project.



The first time that you open the Unreal Editor from Simulink, you might be asked to rebuild UE4Editor DLL files or the MathWorks\_Aerospace module. Click **Yes** to rebuild these files or modules. The editor also prompts you that new plugins are available. Click **Manage Plugins** and verify that the **MathWorks Interface** plugin is installed. This plugin is the MathWorksSimulation.uplugin file that you copied into your Unreal Editor installation in “Install Support Package for Customizing Scenes” on page 2-25.

When the editor opens, you can ignore any warning messages about files with the name '\_BuiltData' that failed to load.

If you receive a warning that the lighting needs to be rebuilt, from the toolbar above the editor window, select **Build > Build Lighting Only**. The editor issues this warning the first time you open a scene or when you add new elements to a scene.

## Reparent Actor Blueprint

**Note** If you are using a scene from the MathWorks\_Aerospace project that comes installed with the UAV Toolbox Interface for Unreal Engine Projects support package, skip this section. However, if you create a new scene based off of one of the scenes in this project, then you must complete this section.

The first time that you open a custom scene from Simulink, you need to associate, or reparent, this project with the **Sim3dLevelScriptActor** level blueprint used in UAV Toolbox. The level blueprint controls how objects interact with the Unreal Engine environment once they are placed in it. Simulink returns an error at the start of simulation if the project is not reparented. You must reparent each scene in a custom project separately.

To reparent the level blueprint, follow these steps:

- 1 In the Unreal Editor toolbar, select **Blueprints > Open Level Blueprint**.
- 2 In the Level Blueprint window, select **File > Reparent Blueprint**.
- 3 Click the **Sim3dLevelScriptActor** blueprint. If you do not see the **Sim3dLevelScriptActor** blueprint listed, use these steps to check that you have the MathWorksSimulation plugin installed and enabled:
  - a In the Unreal Editor toolbar, select **Settings > Plugins**.
  - b In the Plugins window, verify that the **MathWorks Interface** plugin is listed in the installed window. If the plugin is not already enabled, select the **Enabled** check box.
 

If you do not see the **MathWorks Interface** plugin in this window, repeat the steps under “Copy Plugin to Unreal Editor” on page 2-26 and reopen the editor from Simulink.
  - c Close the editor and reopen it from Simulink.
- 4 Close the Level Blueprint window.

## Create or Modify Scenes in Unreal Editor

After you open the editor from Simulink, you can modify the scenes in your project or create new scenes.

## Open Scene

In the Unreal Editor, scenes within a project are referred to as levels. Levels come in several types, and scenes have a level type of map.

To open a prebuilt scene from the `MathWorks_Aerospace.uproject` file, in the **Content Browser** pane below the editor window, navigate to the **Content > Maps** folder. Then, select the map that corresponds to the scene you want to modify.

Unreal Editor Map	UAV Toolbox Scene
USCityBlock	US City Block

To open a scene within your own project, in the **Content Browser** pane, navigate to the folder that contains your scenes.

## Create New Scene

To create a new scene in your project, from the top-left menu of the editor, select **File > New Level**.

Alternatively, you can create a new scene from an existing one. This technique is useful if you want to use one of the prebuilt scenes in the `MathWorks_Aerospace` project as a starting point for creating your own scene. To save a version of the currently opened scene to your project, from the top-left menu of the editor, select **File > Save Current As**. The new scene is saved to the same location as the existing scene.

## Add Assets to Scene

In the Unreal Editor, elements within a scene are referred to as assets. To add assets to a scene, you can browse or search for them in the **Content Browser** pane at the bottom and drag them into the editor window.

When adding assets to a scene that is in the `MathWorks_Aerospace` project, you can choose from a library of driving-related assets. These assets are built as static meshes and begin with the prefix `SM_`. Search for these objects in the **Content Browser** pane.

For example, add a stop sign to a scene in the `MathWorks_Aerospace` project.

- 1 In the **Content Browser** pane at the bottom of the editor, navigate to the **Content** folder.
- 2 In the search bar, search for `SM_StopSign`. Drag the stop sign from the **Content Browser** into the editing window. You can then change the position of the stop sign in the editing window or on the **Details** pane on the right, in the **Transform** section.

The Unreal Editor uses a left-hand Z-up coordinate system, where the Y-axis points to the right. UAV Toolbox uses a right-hand Z-down coordinate system, where the Y-axis points to the left. When positioning objects in a scene, keep this coordinate system difference in mind. In the two coordinate systems, the positive and negative signs for the Y-axis and pitch angle values are reversed.

For more information on modifying scenes and adding assets, see Unreal Engine 4 Documentation.

To migrate assets from the `MathWorks_Aerospace` project into your own project file, see Migrating Assets in the Unreal Engine documentation.

To obtain semantic segmentation data from a scene, then you must apply stencil IDs to the objects added to a scene. For more information, see “Apply Semantic Segmentation Labels to Custom Scenes” on page 2-35.

## Run Simulation

Verify that the Simulink model and Unreal Editor are configured to co-simulate by running a test simulation.

- 1 In the Simulink model, click **Run**.

Because the source of the scenes is the project opened in the Unreal Editor, the simulation does not start. Instead, you must start the simulation from the editor.

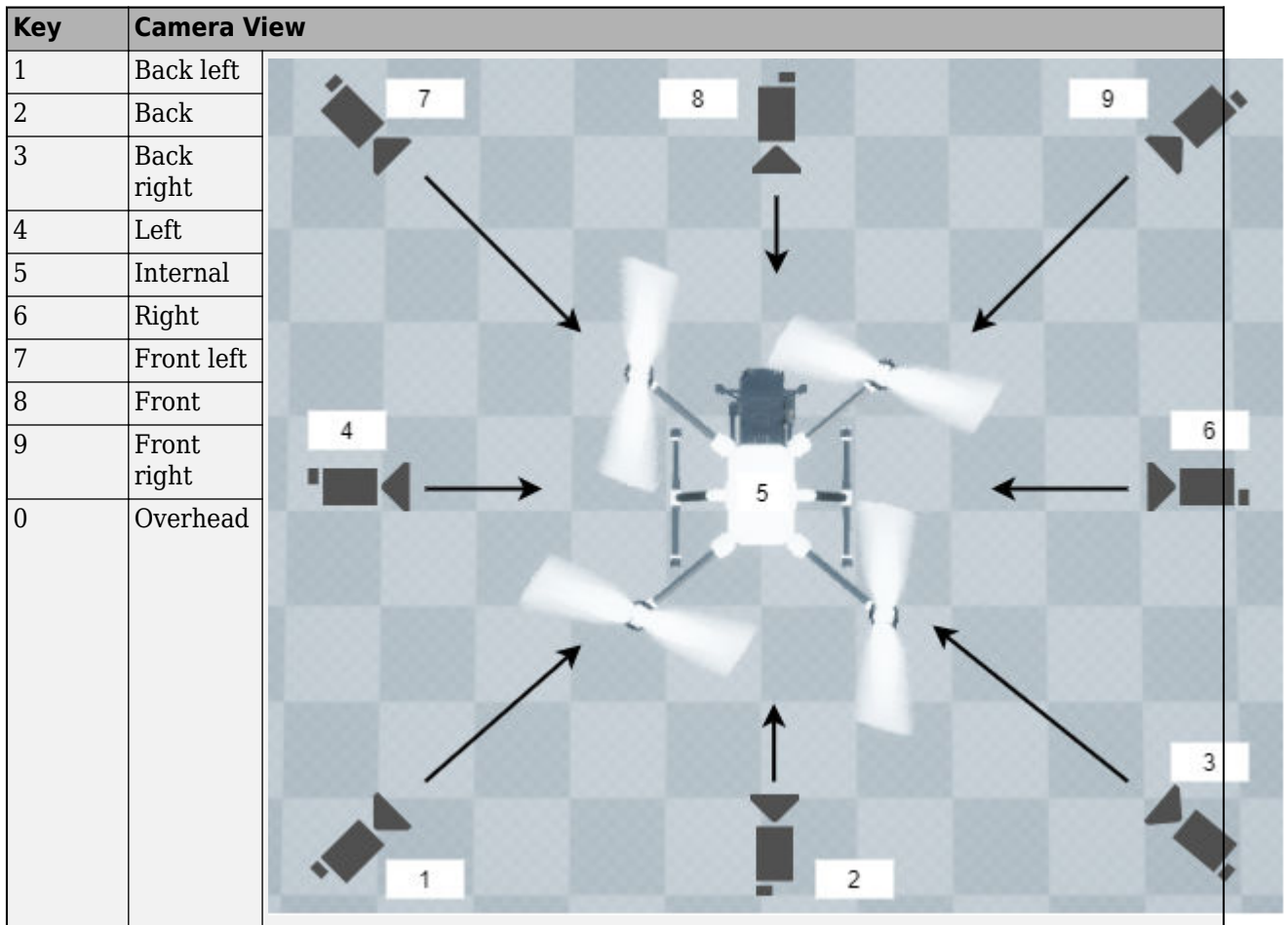
- 2 Verify that the Diagnostic Viewer window in Simulink displays this message:

In the Simulation 3D Scene Configuration block, you set the scene source to 'Unreal Editor'. In Unreal Editor, select 'Play' to view the scene.

This message confirms that Simulink has instantiated vehicles and other objects in the Unreal Engine 3D environment.

- 3 In the Unreal Editor, click **Play**. The simulation runs in the scene currently open in the Unreal Editor.
  - If your Simulink model contains vehicles, these vehicles drive through the scene that is open in the editor.
  - If your Simulink model includes sensors, these sensors capture data from the scene that is open in the editor.

To control the view of the scene during simulation, in the Simulation 3D Scene Configuration block, select the vehicle name from the **Scene view** parameter. To change the scene view as the simulation runs, use the numeric keypad in the editor. The table shows the position of the camera displaying the scene, relative to the vehicle selected in the **Scene view** parameter.



To restart a simulation, click **Run** in the Simulink model, wait until the Diagnostic Viewer displays the confirmation message, and then click **Play** in the editor. If you click **Play** before starting the simulation in your model, the connection between Simulink and the Unreal Editor is not established, and the editor displays an empty scene.

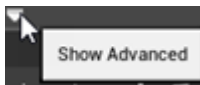
After tuning your custom scene based on simulation results, you can then package the scene into an executable. For more details, see “Package Custom Scenes into Executable” on page 2-33.

## See Also


## Package Custom Scenes into Executable

### Package Scene into Executable Using Unreal Engine

- 1 Open the project containing the scene in the Unreal Editor. You must open the project from a Simulink model that is configured to co-simulate with the Unreal Editor. For more details on this configuration, see “Customize Unreal Engine Scenes Using Simulink and Unreal Editor” on page 2-28.
- 2 In the Unreal Editor toolbar, select **Settings > Project Settings** to open the Project Settings window.
- 3 In the left pane, in the **Project** section, click **Packaging**.
- 4 In the **Packaging** section, set or verify the options in the table. If you do not see all these options, at the bottom of the **Packaging** section, click the **Show Advanced** expander.



Packaging Option	Enable or Disable
Use Pak File	Enable
Cook everything in the project content directory (ignore list of maps below)	Disable
Cook only maps (this only affects cookall)	Enable
Create compressed cooked packages	Enable
Exclude editor content while cooking	Enable

- 5 Specify the scene from the project that you want to package into an executable.
  - a In the **List of maps to include in a packaged build** option, click the **Adds Element** button .
    - b Specify the path to the scene that you want to include in the executable. By default, the Unreal Editor saves maps to the /Game/Maps folder. For example, if the /Game/Maps folder has a scene named myScene that you want to include in the executable, enter /Game/Maps/myScene.
    - c Add or remove additional scenes as needed.
- 6 Rebuild the lighting in your scenes. If you do not rebuild the lighting, the shadows from the light source in your executable file are incorrect and a warning about rebuilding the lighting displays during simulation. In the Unreal Editor toolbar, select **Build > Build Lighting Only**.
- 7 (Optional) If you plan to semantic segmentation data from the scene by using a Simulation 3D Camera block, enable rendering of the stencil IDs. In the left pane, in the **Engine** section, click **Rendering**. Then, in the main window, in the **Postprocessing** section, set **Custom Depth-Stencil Pass** to Enabled with Stencil. For more details on applying stencil IDs for semantic segmentation, see “Apply Semantic Segmentation Labels to Custom Scenes” on page 2-35.
- 8 Close the **Project Settings** window.

- 9 In the top-left menu of the editor, select **File > Package Project > Windows > Windows (64-bit)**. Select a local folder in which to save the executable, such as to the root of the project file (for example, `C:/Local/myProject`).

---

**Note** Packaging a project into an executable can take several minutes. The more scenes that you include in the executable, the longer the packaging takes.

---

Once packaging is complete, the folder where you saved the package contains a `WindowsNoEditor` folder that includes the executable file. This file has the same name as the project file.

---

**Note** If you repackage a project into the same folder, the new executable folder overwrites the old one.

---

Suppose you package a scene that is from the `myProject.uproject` file and save the executable to the `C:/Local/myProject` folder. The editor creates a file named `myProject.exe` with this path:

```
C:/Local/myProject/WindowsNoEditor/myProject.exe
```

### Simulate Scene from Executable in Simulink

- 1 In the Simulation 3D Scene Configuration block of your Simulink model, set the **Scene source** parameter to `Unreal Executable`.
- 2 Set the **File name** parameter to the name of your Unreal Editor executable file. You can either browse for the file or specify the full path to the file by using backslashes. For example:

```
C:\Local\myProject\WindowsNoEditor\myProject.exe
```

- 3 Set the **Scene** parameter to the name of a scene from within the executable file. For example:

```
/Game/Maps/myScene
```

- 4 Run the simulation. The model simulates in the custom scene that you created.

If you are simulating a scene from a project that is not based on the `mwas` project, then the scene simulates in full screen mode. To use the same window size as the default scenes, copy the `DefaultGameUserSettings.ini` file from the support package installation folder to your custom project folder. For example, copy `DefaultGameUserSettings.ini` from:

```
C:\ProgramData\MATLAB\SupportPackages\<MATLABrelease>\toolbox\uav\spkg\uavunrealengine\mwas\Config
```

to:

```
C:\<yourproject>.project\Config
```

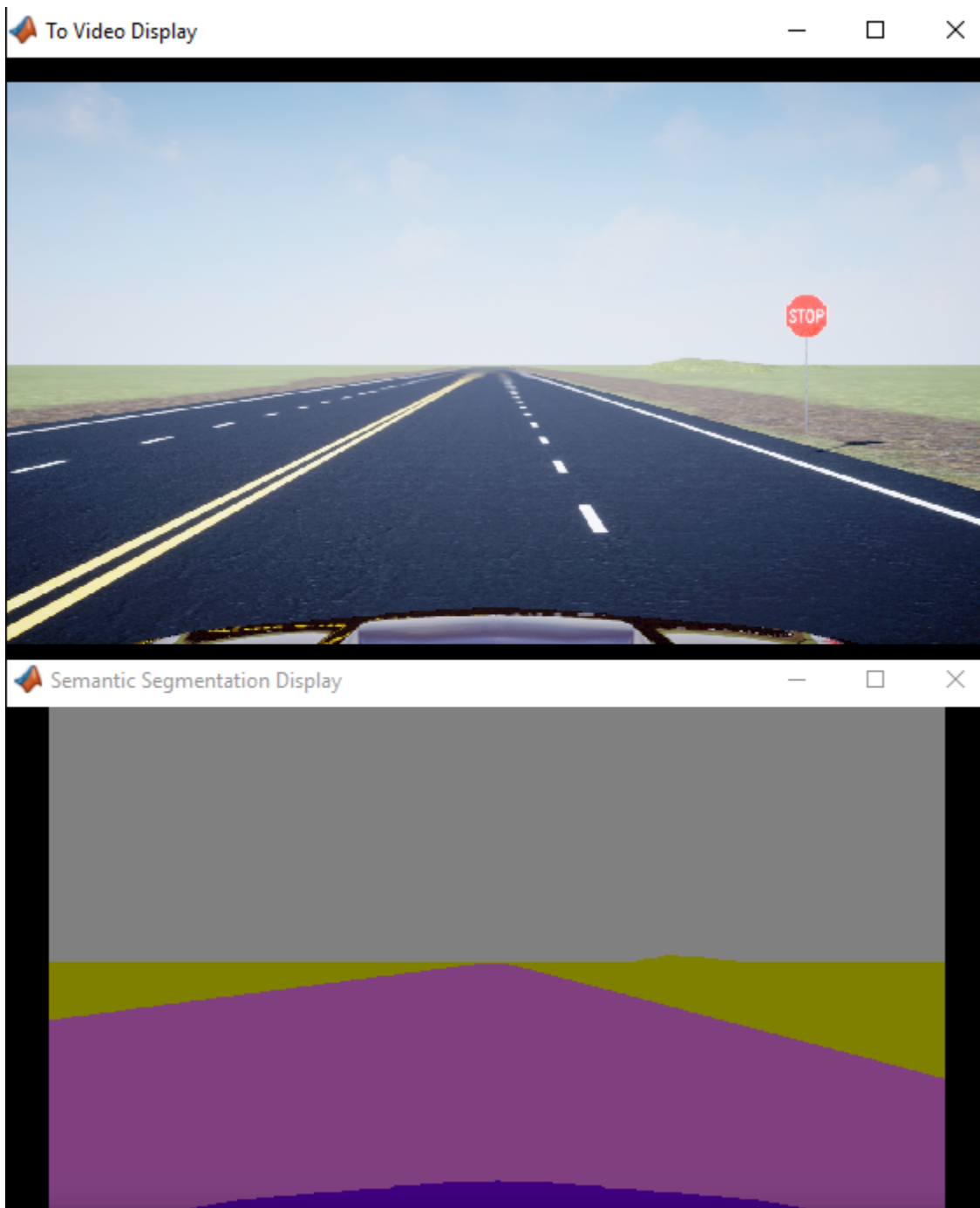
Then, package scenes from the project into an executable again and retry the simulation.

### See Also

## Apply Semantic Segmentation Labels to Custom Scenes

The Simulation 3D Camera block provides an option to output semantic segmentation data from a scene. If you add new scene elements, or assets (such as traffic signs or roads), to a custom scene, then in the Unreal Editor, you must apply the correct ID to that element. This ID is known as a stencil ID. Without the correct stencil ID applied, the Simulation 3D Camera block does not recognize the scene element and does not display semantic segmentation data for it.

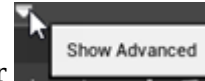
For example, this To Video Display window shows a stop sign that was added to a custom scene. The Semantic Segmentation Display window does not display the stop sign, because the stop sign is missing a stencil ID.



To apply a stencil ID label to a scene element, follow these steps:

- 1 Open the Unreal Editor from a Simulink model that is configured to simulate in the 3D environment. For more details, see "Customize Unreal Engine Scenes Using Simulink and Unreal Editor" on page 2-28.
- 2 In the editor window, select the scene element with the missing stencil ID.
- 3 On the **Details** pane on the right, in the **Rendering** section, select **Render CustomDepth Pass**.





If you do not see this option, click the **Show Advanced** expander to show all rendering options.

- 4 In the **CustomDepth Stencil Value** box, enter the stencil ID that corresponds to the asset. If you are adding an asset to a scene from the UAV Toolbox Interface for Unreal Engine Projects support package, then enter the stencil ID corresponding to that asset type, as shown in the table. If you are adding assets other than the ones shown, then you can assign them to unused IDs. If you do not assign a stencil ID to an asset, then the Unreal Editor assigns that asset an ID of 0.


**Note** The Simulation 3D Camera block does not support the output of semantic segmentation data for lane markings. Even if you assign a stencil ID to lane markings, the block ignores this setting.

ID	Type
0	None/default
1	Building
2	<i>Not used</i>
3	Other
4	<i>Not used</i>
5	Pole
6	<i>Not used</i>
7	Road
8	Sidewalk
9	Vegetation
10	Vehicle
11	<i>Not used</i>
12	Generic traffic sign
13	Stop sign
14	Yield sign
15	Speed limit sign
16	Weight limit sign
17 - 18	<i>Not used</i>
19	Left and right arrow warning sign
20	Left chevron warning sign
21	Right chevron warning sign
22	<i>Not used</i>
23	Right one-way sign
24	<i>Not used</i>

ID	Type
25	School bus only sign
26 - 38	<i>Not used</i>
39	Crosswalk sign
40	<i>Not used</i>
41	Traffic signal
42	Curve right warning sign
43	Curve left warning sign
44	Up right arrow warning sign
45 - 47	<i>Not used</i>
48	Railroad crossing sign
49	Street sign
50	Roundabout warning sign
51	Fire hydrant
52	Exit sign
53	Bike lane sign
54 - 56	<i>Not used</i>
57	Sky
58	Curb
59	Flyover ramp
60	Road guard rail
61 - 66	<i>Not used</i>
67	Deer
68 - 70	<i>Not used</i>
71	Barricade
72	Motorcycle
73 - 255	<i>Not used</i>

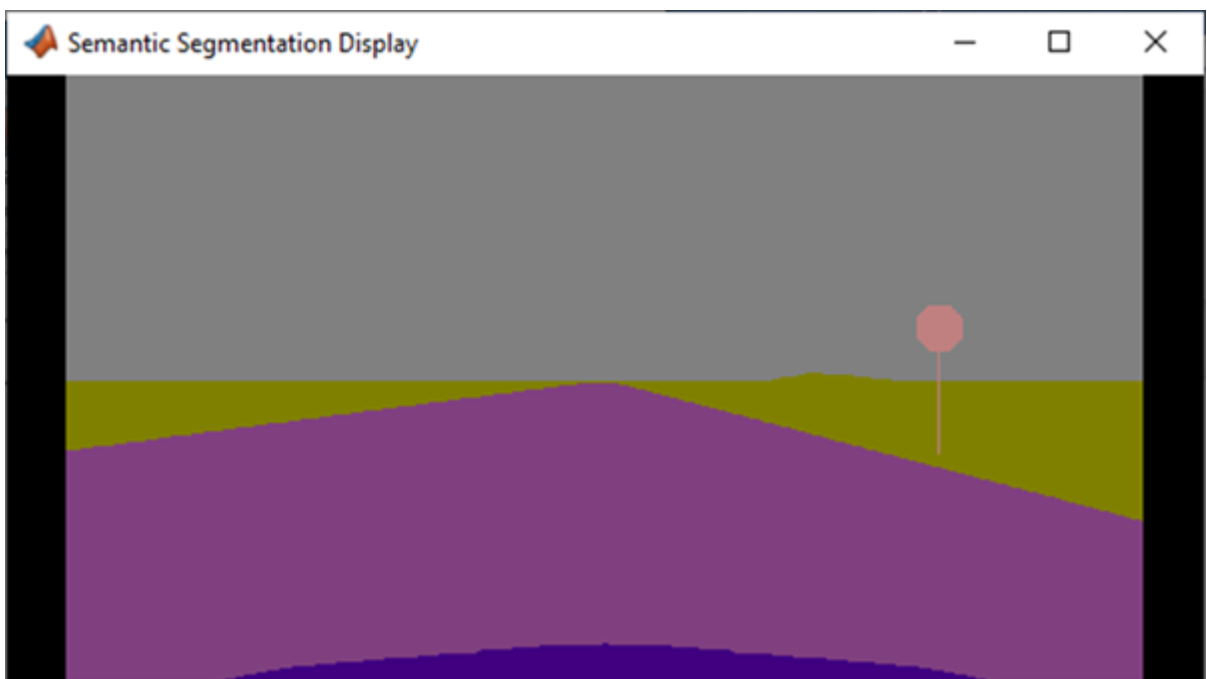
For example, for a stop sign that is missing a stencil ID, enter 13.

**Tip** If you are adding stencil ID for scene elements of the same type, you can copy (**Ctrl+C**) and paste (**Ctrl+V**) the element with the added stencil ID. The copied scene element includes the stencil ID.

- 5 Visually verify that the correct stencil ID shows by using the custom stencil view. In the top-left corner of the editor window, click  and select **Buffer Visualization > Custom Stencil**. The scene displays the stencil IDs specified for each scene element. For example, if you added the correct stencil ID to a stop sign (13) then the editor window, the stop sign displays a stencil ID value of 13.



- If you did not set a stencil ID value for a scene element, then the element appears in black and displays no stencil ID.
  - If you did not select **CustomDepth Stencil Value**, then the scene element does not appear at all in this view.
- 6 Turn off the custom stencil ID view. In the top-left corner of the editor window, click **Buffer Visualization** and then select **Lit**.
  - 7 If you have not already done so, set up your Simulink model to display semantic segmentation data from a Simulation 3D Camera block. For an example setup, see “Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” on page 2-19.
  - 8 Run the simulation and verify that the Simulation 3D Camera block outputs the correct data. For example, here is the Semantic Segmentation Display window with the correct stencil ID applied to a stop sign.



### **See Also**

[Simulation 3D Camera](#) | [Simulation 3D Scene Configuration](#) | [Simulation 3D UAV Vehicle](#)

### **More About**

- “Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation” on page 2-19
- “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)