UAV Toolbox

User's Guide

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Contents

3

3D Data Processing – User's Guide

4

Simulink Block Examples

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);
```
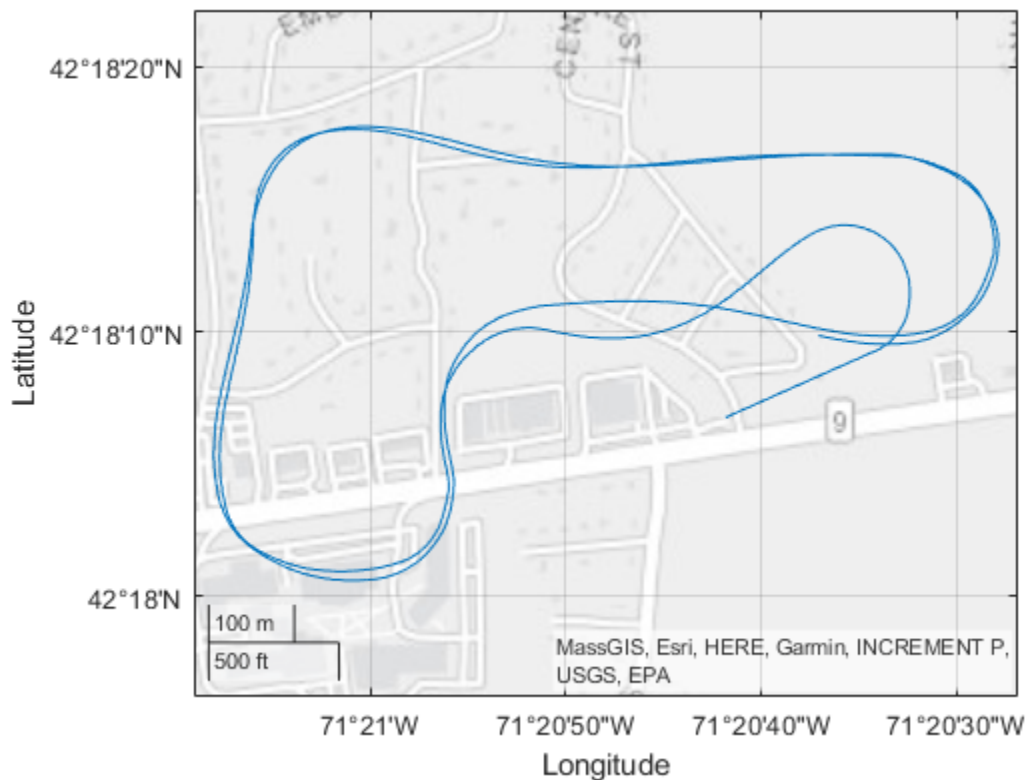

Extract the attitude messages from the TLOG. Specify the message name for attitude messages. Plot the roll, pitch, yaw data using `stackedplot`.

```
msgs = readmsg(logimport,'MessageName','ATTITUDE','Time',[0 100]);

figure()
stackedplot(msgs.Messages{1},{'roll','pitch','yaw'});
```
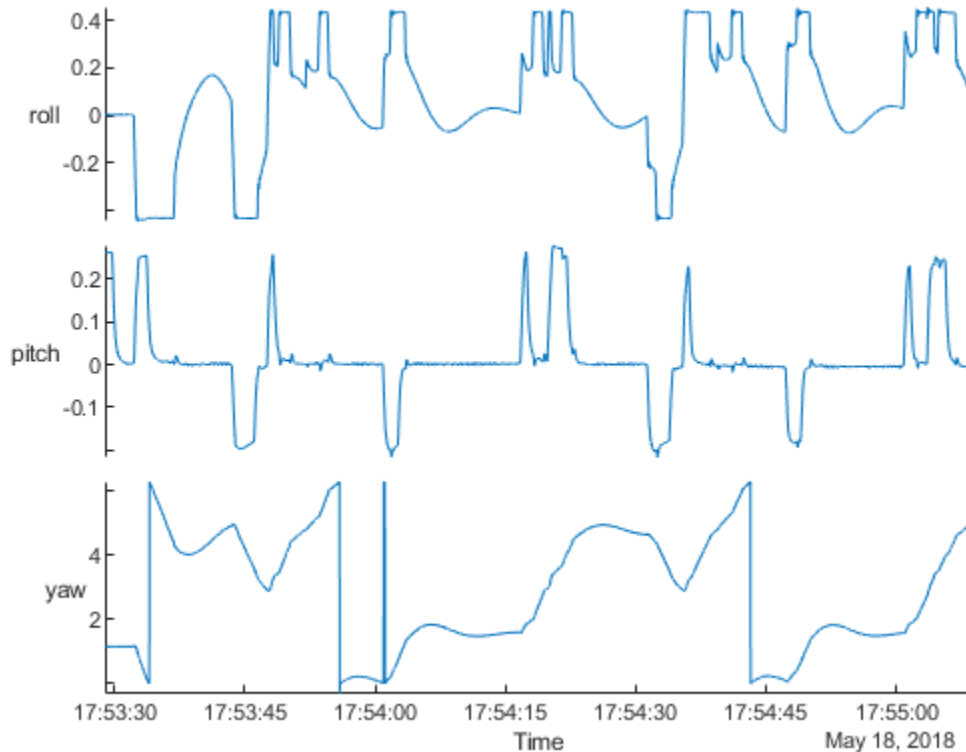

Playback MAVLink Log Entries

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 communcation 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 intrument 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)"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("flight.ulg");
plotter = flightLogSignalMapping("ulog");
info(plotter,"Signal")
```

```
ans=18×4 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,
    "GPS"                   true        "Latitude, Longitude, Altitude, GroundSpeed, CourseAngle
    "Gyro"                  true        "GyroX, GyroY, GyroZ"
    "LocalENU"              true        "X, Y, Z"
    "LocalENUTarget"        true        "XTarget, YTarget, ZTarget"
    "LocalENUVel"           true        "VX, VY, VZ"
    "LocalENUVelTarget"     true        "VXTarget, VYTarget, VZTarget"
    "LocalNED"              true        "X, Y, Z"
    "LocalNEDTarget"        true        "XTarget, YTarget, ZTarget"
    "LocalNEDVel"           true        "VX, VY, VZ"
        ⋮
```

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.

```matlab
% 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,'u

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

% 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 Simulik, you can create custom visualizations of signals using intrument 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');
```
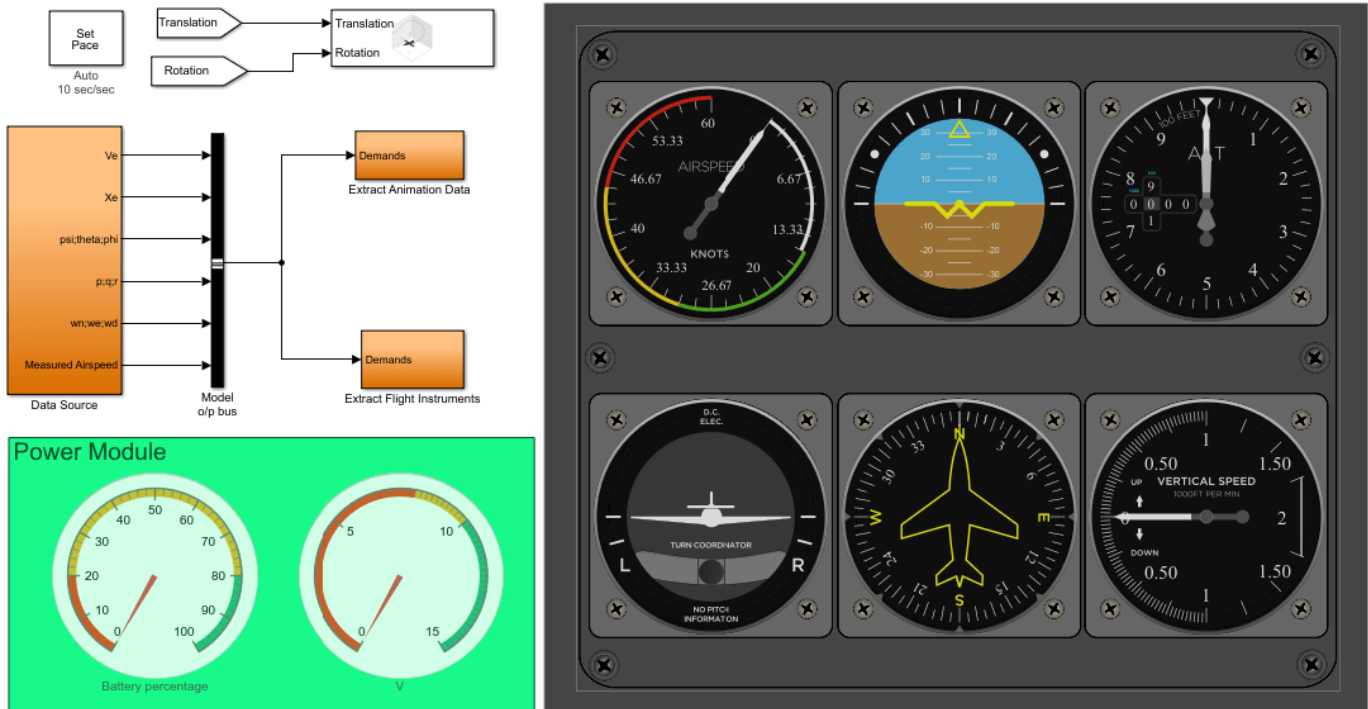

Run the model. The generated figure shows the trajectory of the UAV in real time and the gauges show the current status of the flight.

```
sim('dronegauges');
```
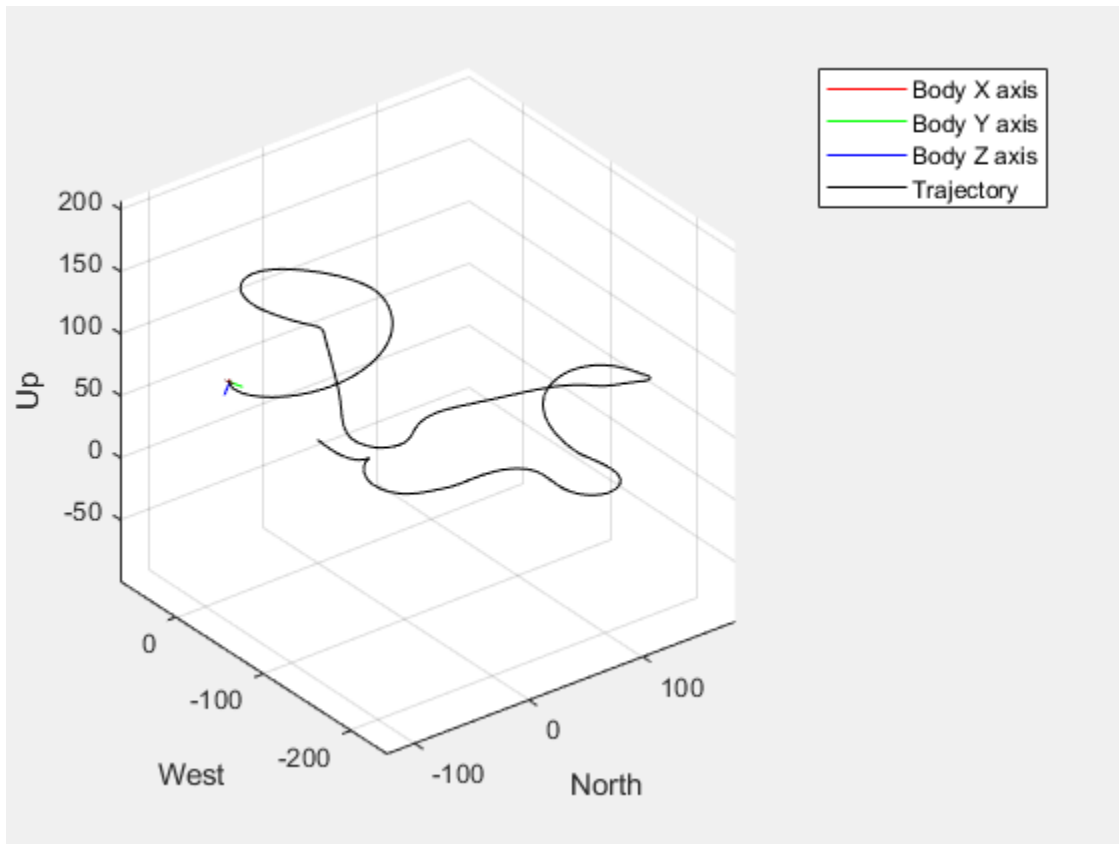
Visualize Custom Flight Log

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 can properly handle the log data saved in the MAT file and visualize it. The data, `customFlightData.mat`, stores a structure 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 data 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: [2785×9 double]
           Fs: 100
   Trajectory: [2785×10 double]
```

The `IMU` field in `logData` is an n-by-9 matrix, where the first 3 columns are 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 μT.

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

```
ans = 5×9

    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-9 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 = 5×10

    0.0200         0   -4.0000    2.0000         0   -0.0036    1.0000         0         0   -0.0
    0.0400         0   -4.0001    2.0000         0   -0.0072    1.0000         0         0   -0.0
    0.0600         0   -4.0002    2.0000         0   -0.0108    1.0000         0         0   -0.0
    0.0800         0   -4.0003    2.0000         0   -0.0143    1.0000         0         0   -0.0
    0.1000         0   -4.0004    2.0000         0   -0.0179    1.0000         0         0   -0.0
```

Visualize Custom Flight Log Using Predefined Signal Format and Plots

Create a `flightLogSignalMapping` object with no input argument since the custom log format does 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 have a "`#`" symbol suffix. For these signals, you can optionally add integers as suffixes to the signal names so that the flight log plotter can handle multiple of signals of this kind, such as secondary IMU signals and barometer readings. Call `info`.

```
% Predefined signals
info(customPlotter, "Signal")
```

ans=*18×4 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,
"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=*10×4 table*

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

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=*1×4 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 signalsfor 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 celocities
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 quickly check if the signals are correctly mapped call checkSignal and specify the logData.

```
checkSignal(customPlotter,logData);
```

```
-------------------------------------------
SignalName: Gyro
```

```
Pass
-------------------------------------------
SignalName: Accel
Pass
-------------------------------------------
SignalName: Mag
Pass
-------------------------------------------
SignalName: LocalNED
Pass
-------------------------------------------
SignalName: LocalNEDVel
Pass
-------------------------------------------
SignalName: AttitudeEuler
Pass
```

To get a preview of a mapped signal select the preview option in checkSignal.

```
checkSignal(customPlotter,logData,'Preview',"on",'Signal',"Accel");
```

```
-------------------------------------------
SignalName: Accel
Pass
Press a key to continue or 'q' to quit. Figure needs to be in focus.
```

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);
```
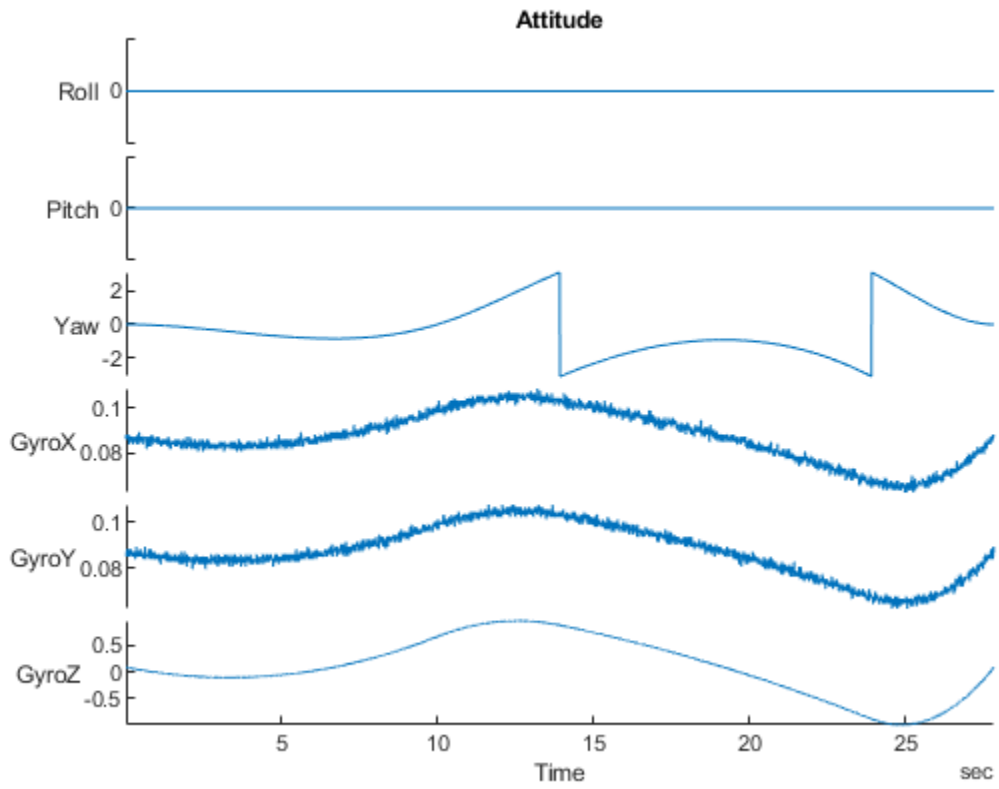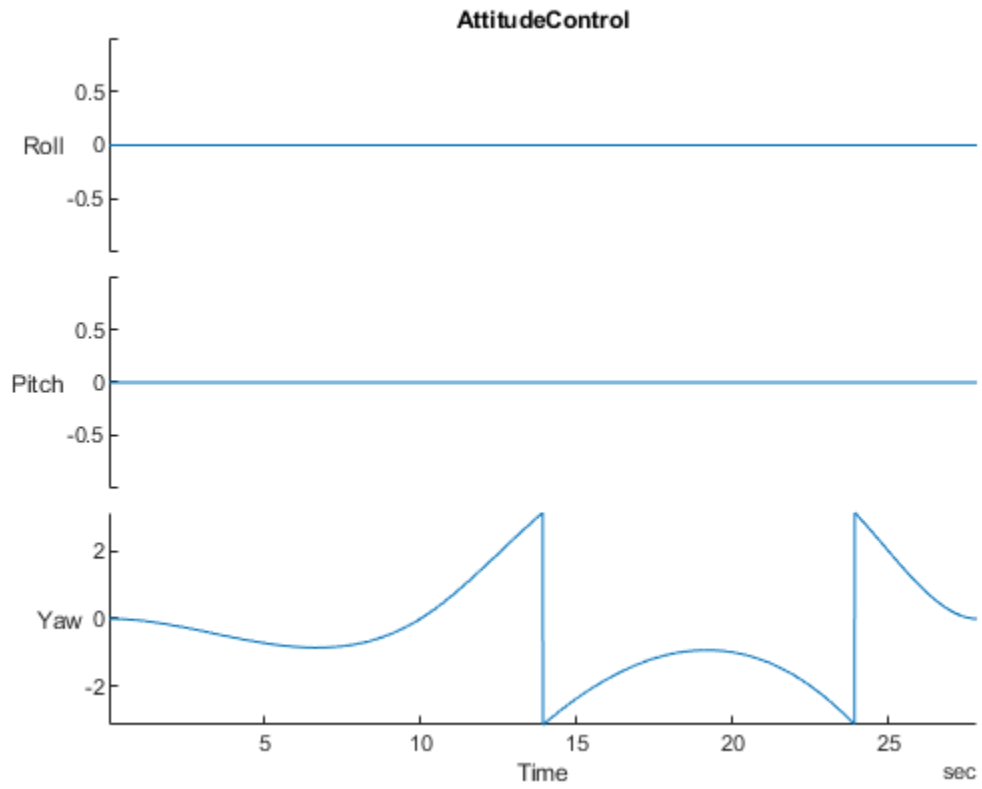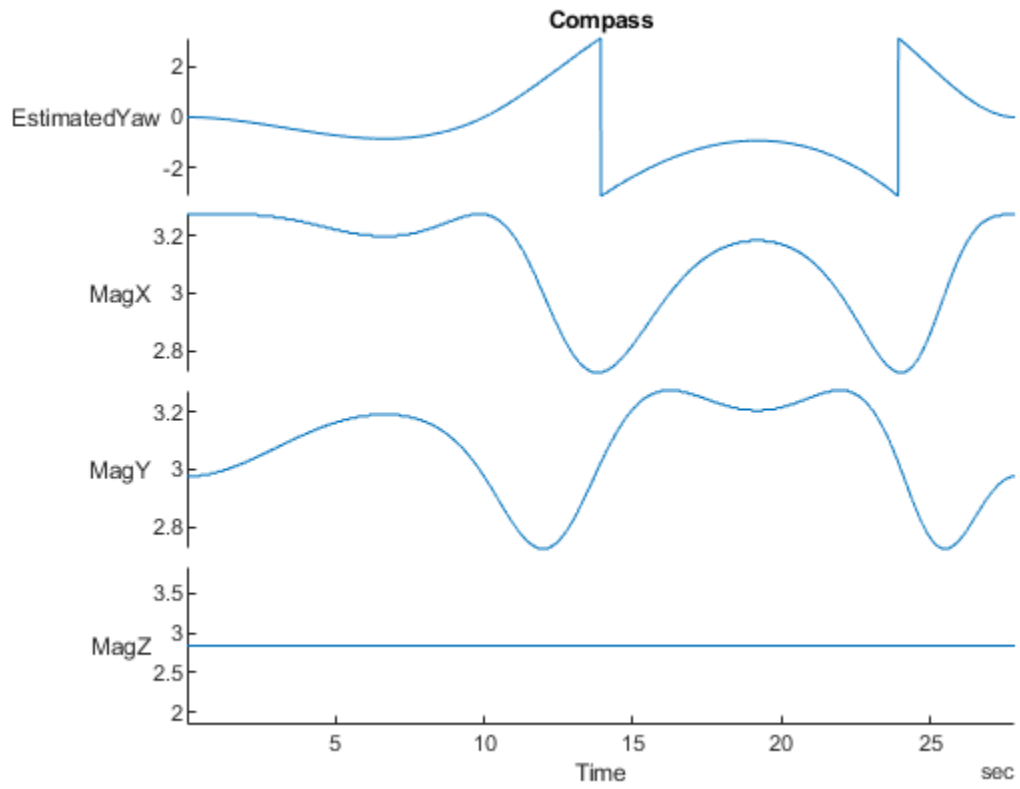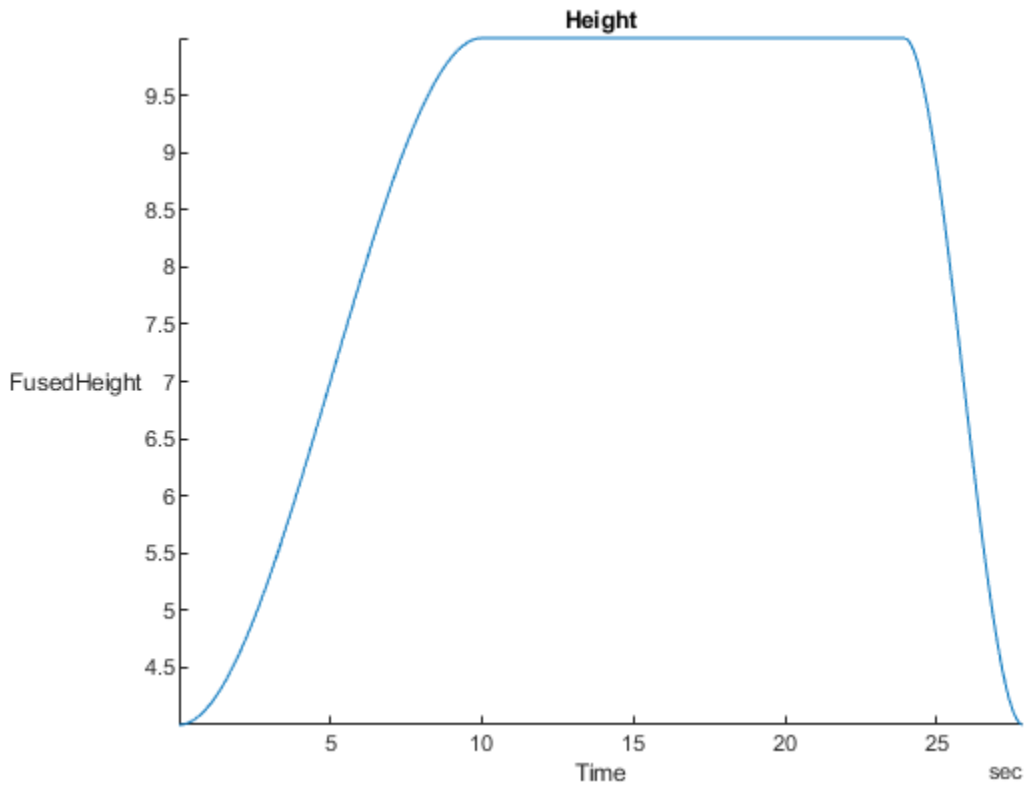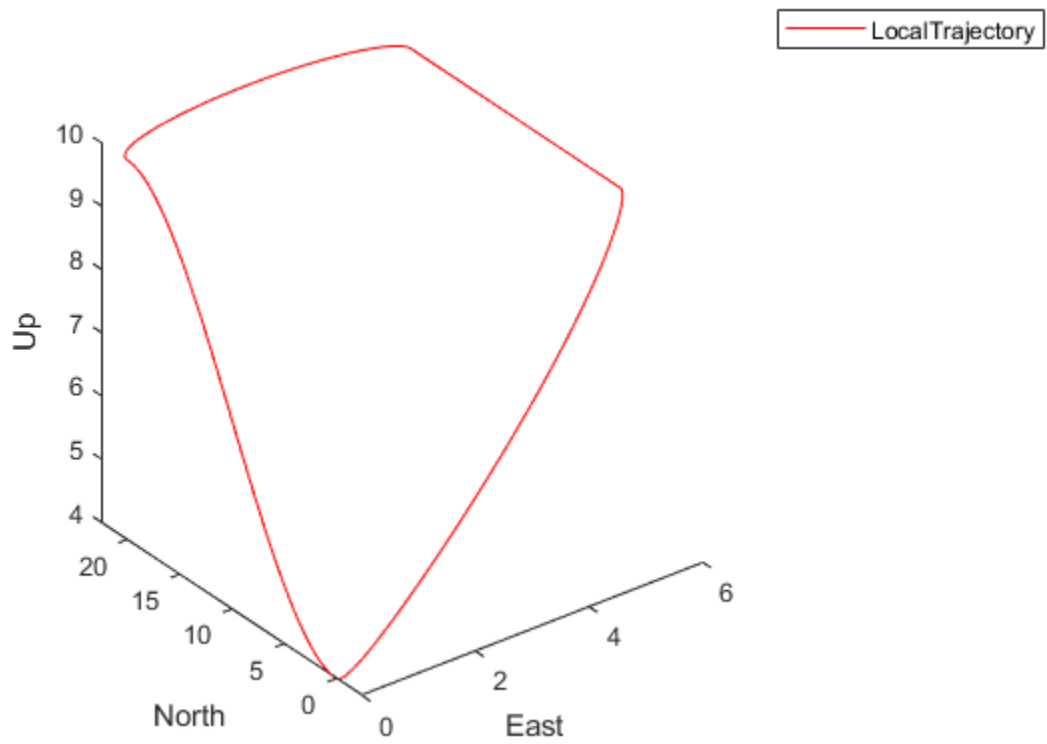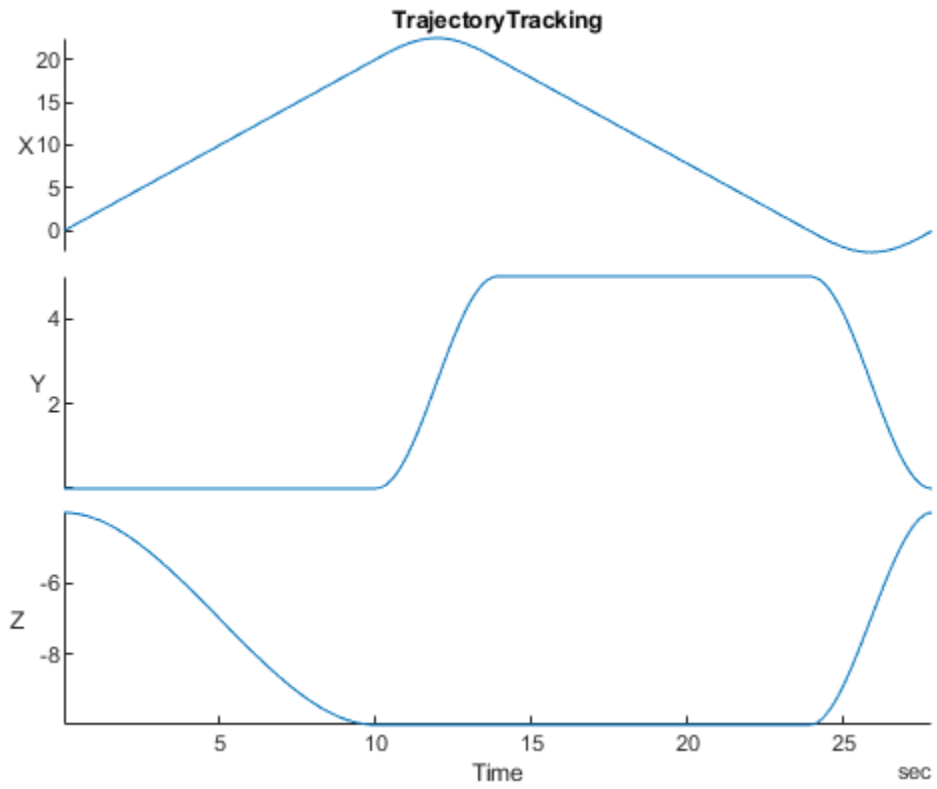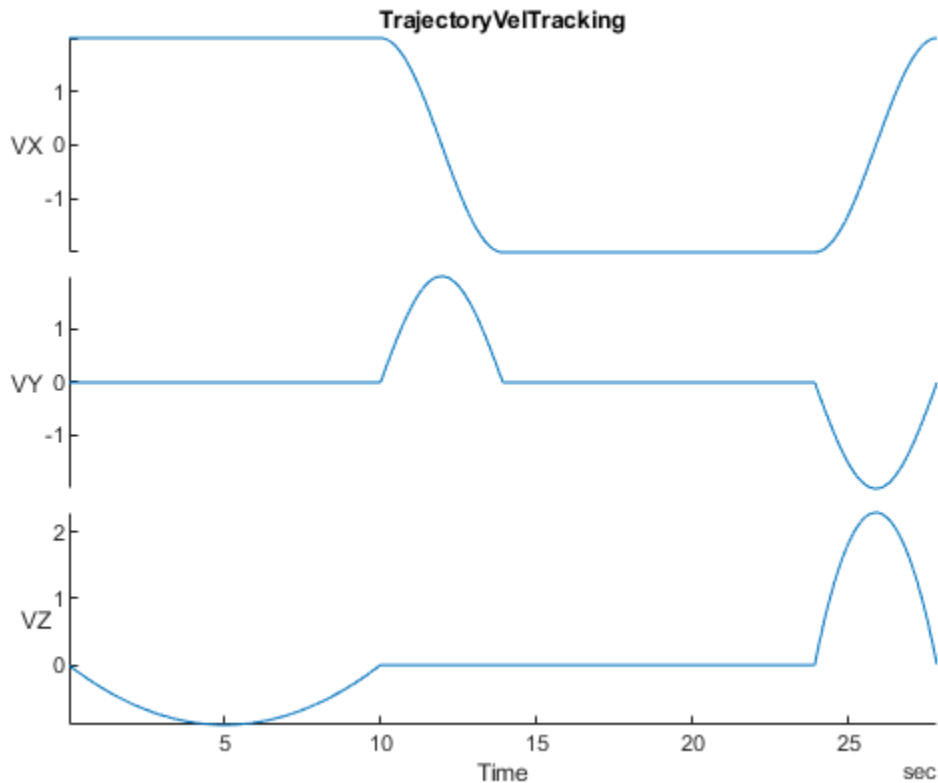
Visualize Custom Flight Log with Custom Plot

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 arguments. To specify a time series of data, use `"Timeseries"` as the third argument, and then list the data.

```
updatePlot(customPlotter, "AnalyzeAccel","Timeseries",["HighAccel.AccelGreaterThan11", "LocalNED
```

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=19×4 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,
"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=12×4 table

PlotName	ReadyToPlot	MissingSignals	RequiredSignals
"AnalyzeAccel"	true	""	"HighAccel, LocalNEDVel"
"Attitude"	true	"AttitudeRate"	"AttitudeEuler, AttitudeR
"AttitudeControl"	true	"AttitudeTargetEuler"	"AttitudeEuler, AttitudeT
"Compass"	true	"GPS#"	"AttitudeEuler, Mag#, GPS#
"Height"	true	"Barometer#, GPS#"	"Barometer#, GPS#, LocalNE
"Trajectory"	true	"LocalNEDTarget"	"LocalNED, LocalNEDTarget"
"TrajectoryTracking"	true	"LocalNEDTarget"	"LocalNED, LocalNEDTarget"
"TrajectoryVelTracking"	true	"LocalNEDVelTarget"	"LocalNEDVel, LocalNEDVelT
"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);
```
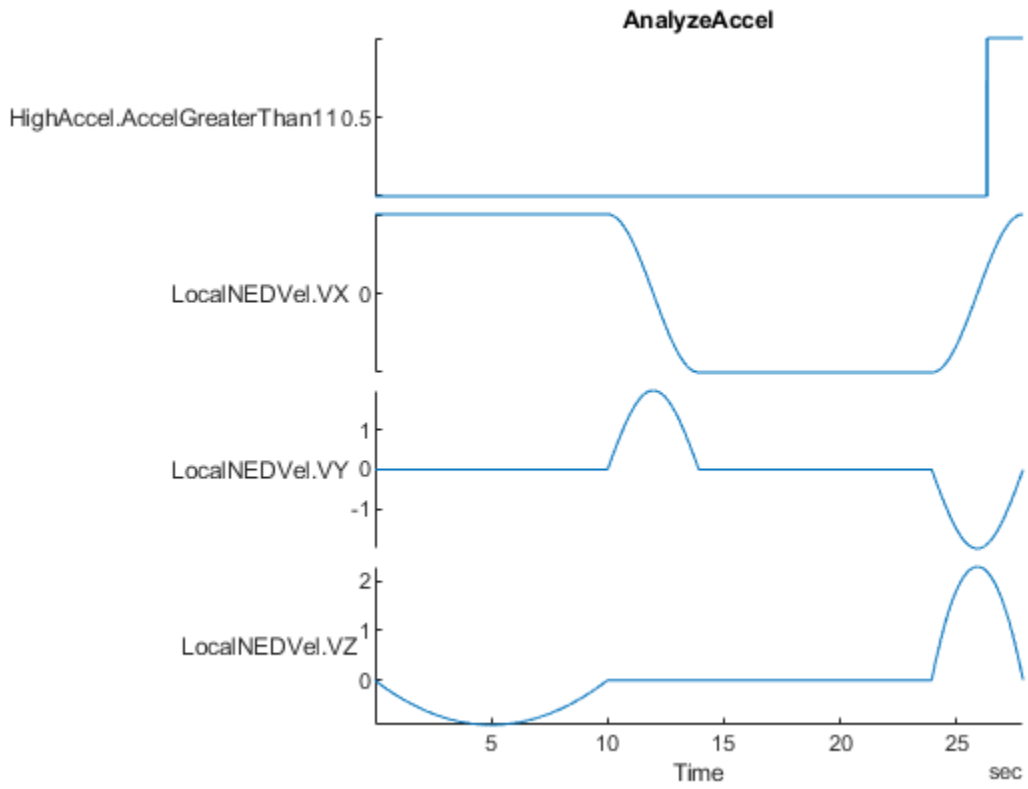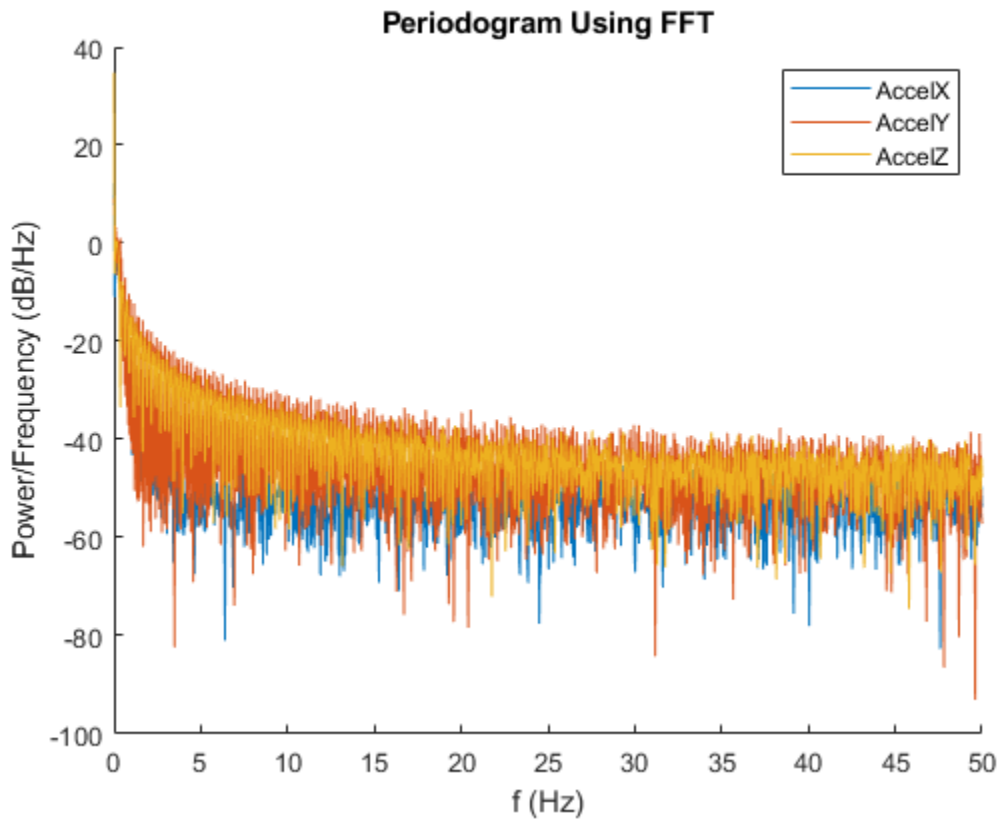
This example has shown how to use the `flightLogSignalMapping` object to look at predefined signals and plots, as well as customize your own plots for flight log analysis.

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
```

Analyze UAV Autopilot Flight Log Using Flight Log Analyzer

This example shows you how to launch the Flight Log Analyzer app, import flight log data, create figures and plots, export signals and using custom signal mapping in the app.

The Flight Log Analyzer app enables you to analyze log files generated by simulated or real flights.

Log analysis helps find the root cause of a crash, or monitor the health during a flight of a vehicle. You can perform basic analysis to determine:

- How well the controllers track their references
- Whether there is any strong vibration
- If the vehicle experiences power failures

Open Flight Log Analyzer App

In the **Apps** tab, under **Control System Design and Analysis,** click **Flight Log Analyzer**.

Alternatively, you can use the `flightLogAnalyzer` function from the MATLAB® command prompt.

Import ULOG File

Load the ULOG file.

```
ulg = ulogreader('flight.ulg')

ulg =
  ulogreader with properties:

             FileName: 'flight.ulg'
            StartTime: 00:00:00.176000
              EndTime: 00:02:15.224000
      AvailableTopics: [51×5 table]
    DropoutIntervals: [0×2 duration]
```

On the **Flight Log Analyzer** app toolstrip, select **Import > From Workspace**. In the **Log Data** section of the Import flight log signal mapping and log data from Workspace dialog box, select the `ulogreader` object `ulg` and click **Import**.
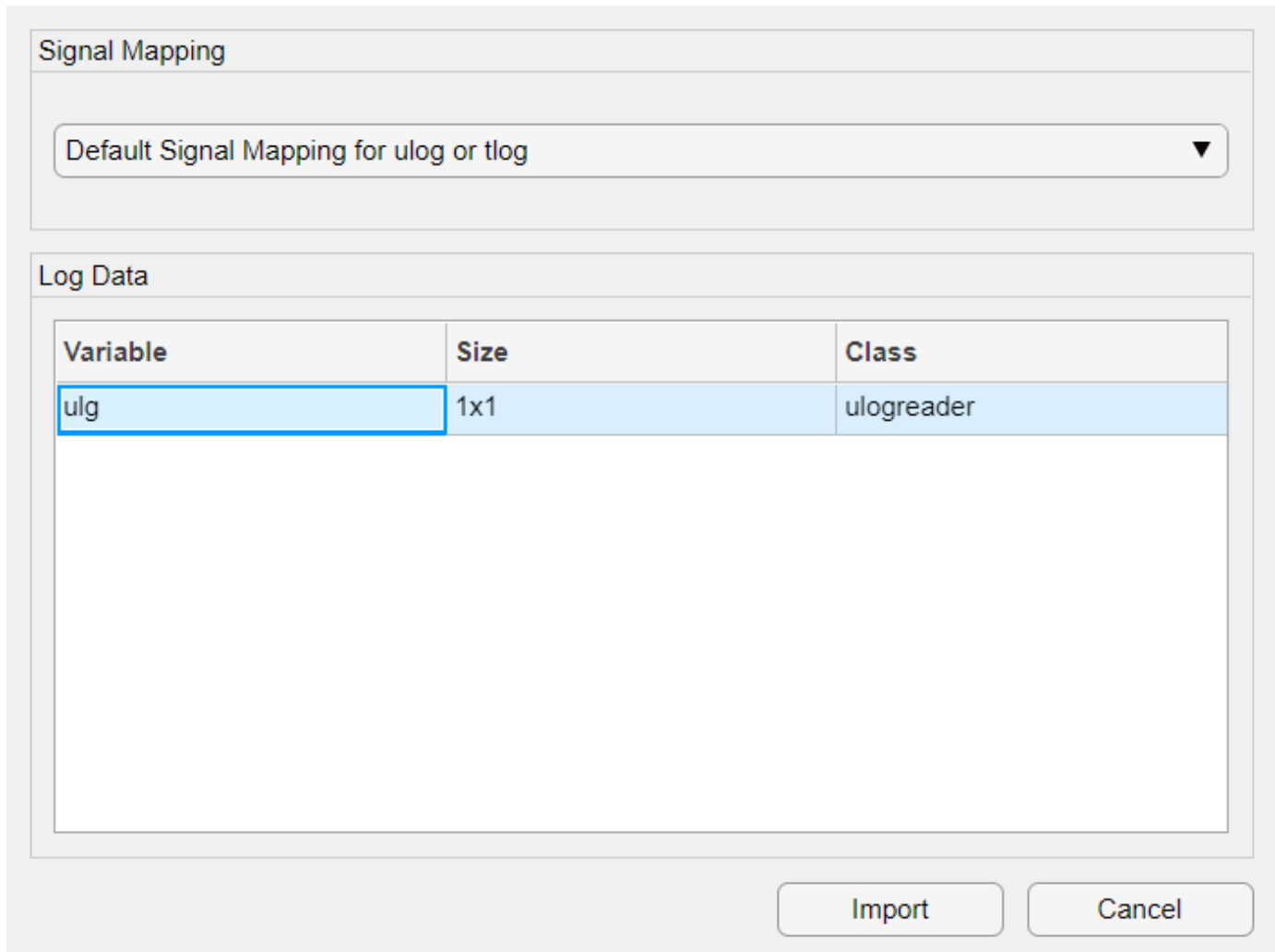
By default, the app displays a satellite map with logged GPS data and the flight modes as a table. The flight modes, along with their corresponding start and end times, are tabulated in the **Flight Modes** pane.
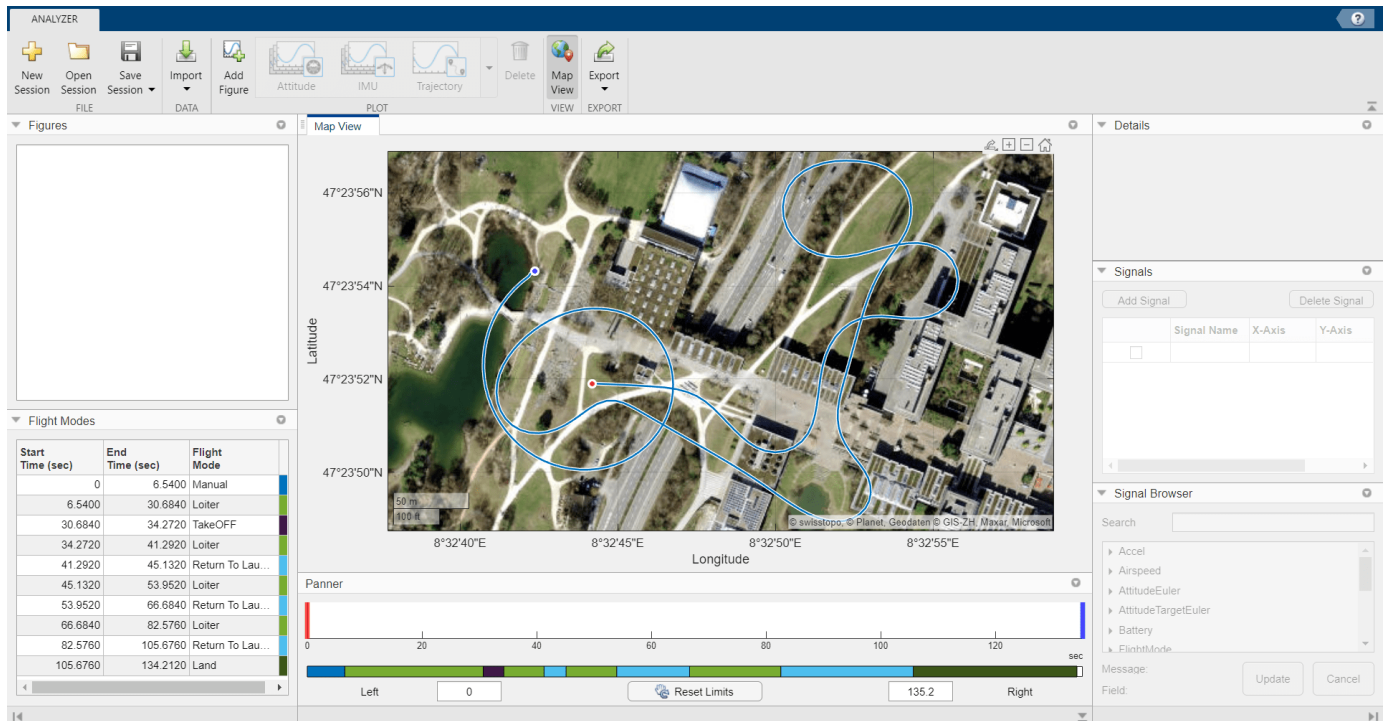
Create Figures and Plots

Every UAV (fixed or multi-rotor) is equipped with a set of sensors, such as a gyroscope, accelerometer, magnetometer, and barometer, to determine the vehicle state. A vehicle state includes the position, velocity, altitude, speed, and rates of rotation of the vehicle.

Add Predefined Plot

In the **Plot** section of the app toolstrip, click **Add Figure** to add an empty figure to the plotting pane. Then, in the plot gallery, click **IMU** to add plots to the figure for the gyroscope, **Gyro**, and accelerometer, **Accelerometer**.
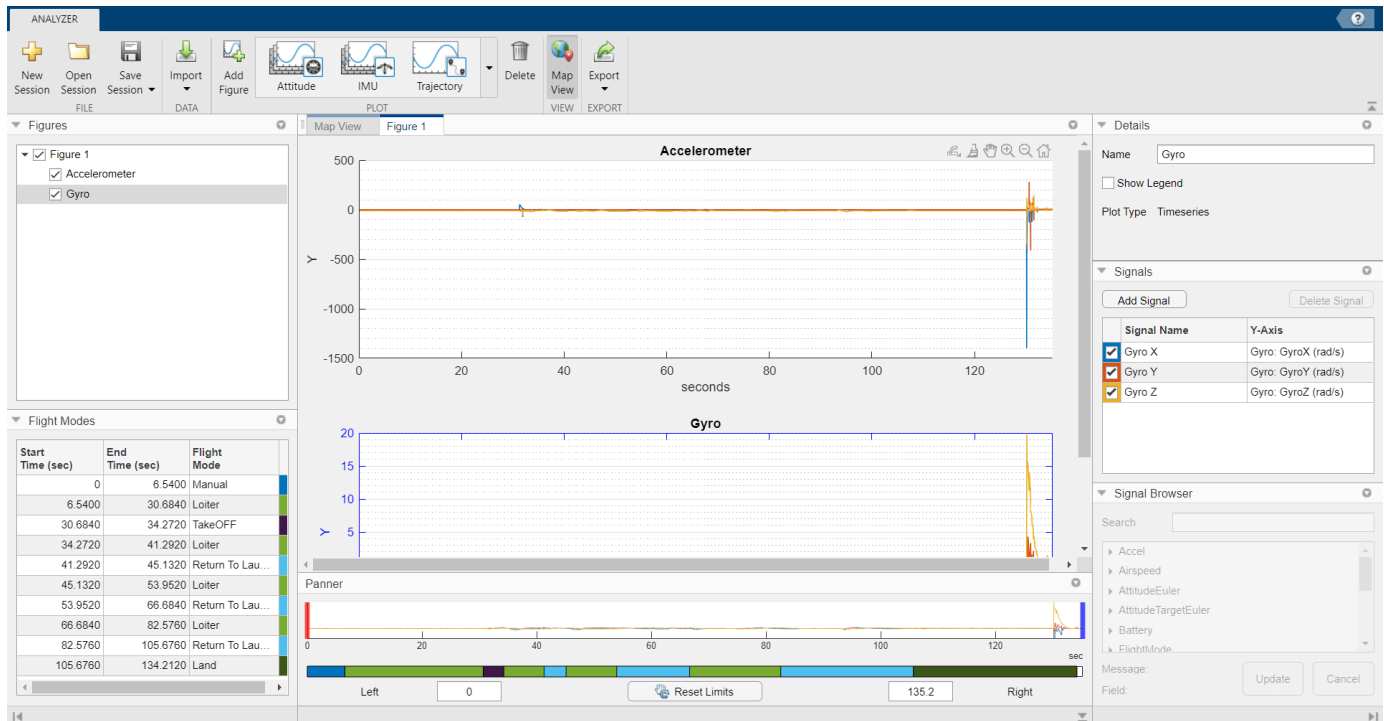
You can use various predefined plots from the plot gallery to visualize data from different sensors.

Change Plot Focus Using Panner

In the **Flight Modes** pane, find the first instance of the **Loiter** flight mode and note its **Start Time** and **End Time** values. Focus on the flight mode by, in the **Panner** pane, dragging the red and blue handles to the **Start Time** and **End Time**, respectively, of the desired flight mode. Alternatively, you can type the **Start Time** and **End Time** values in the **Left** and **Right** boxes, beneath the strip plot. Click the **Acceleration** plot to focus on it.
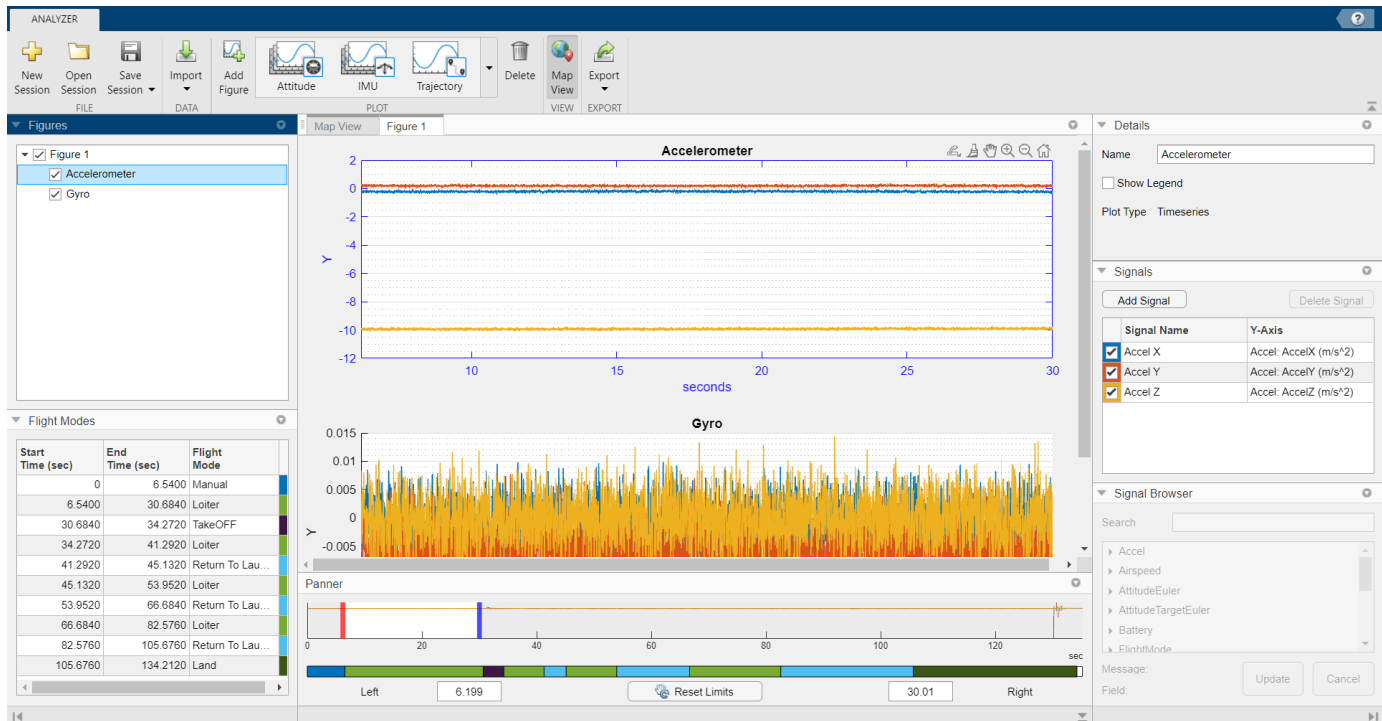
If UAV vibration is in a good range, then z-axis acceleration should remain below x-axis and y-axis acceleration. The plotted flight data indicates that, at this point in the flight, UAV vibration is in a good range. Using the **Panner**, focus on the other three **Loiter** flight modes and observe the acceleration of the UAV.

Add Custom Plot

Next, create a custom **Timeseries** plot to compare the estimated roll against the roll target.

1 First, in the **Custom Plots** section of the plot gallery, click **Timeseries**.

2 In the **Signals** pane, click **Add Signal** twice to add two signals.

3 Double-click the **Y-Axis** column of the first signal and, in the **Signal Browser** pane, type `RollTarget` in the **Search** box, and then click the arrow next to `AttitudeTargetEuler` and select `RollTarget`. Then, click **Update**.

4 Repeat the previous three steps for the second signal to add `Roll`.

5 Rename the first signal to `RollTarget` and the second signal to `Roll`. To rename a signal, double-click its entry in the **Signal Name** column and type the new name.

6 In the **Details** pane, select **Show Legend** to show the legend on the plot.

The plot shows that the estimated roll closely follows the roll target until the last few seconds of the flight.
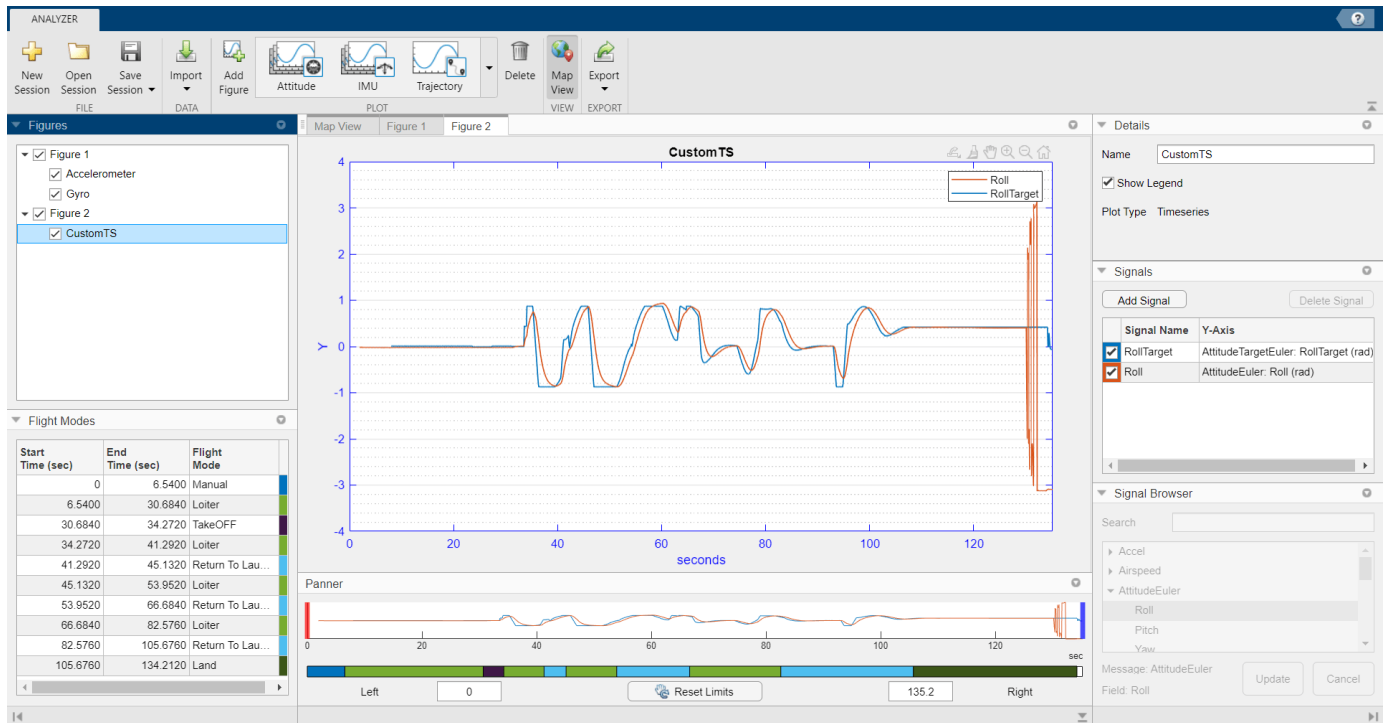
Export Signals

To further analyze the data, you can export signals. On the app toolstrip, click **Export**, and then select **Export Signal**.
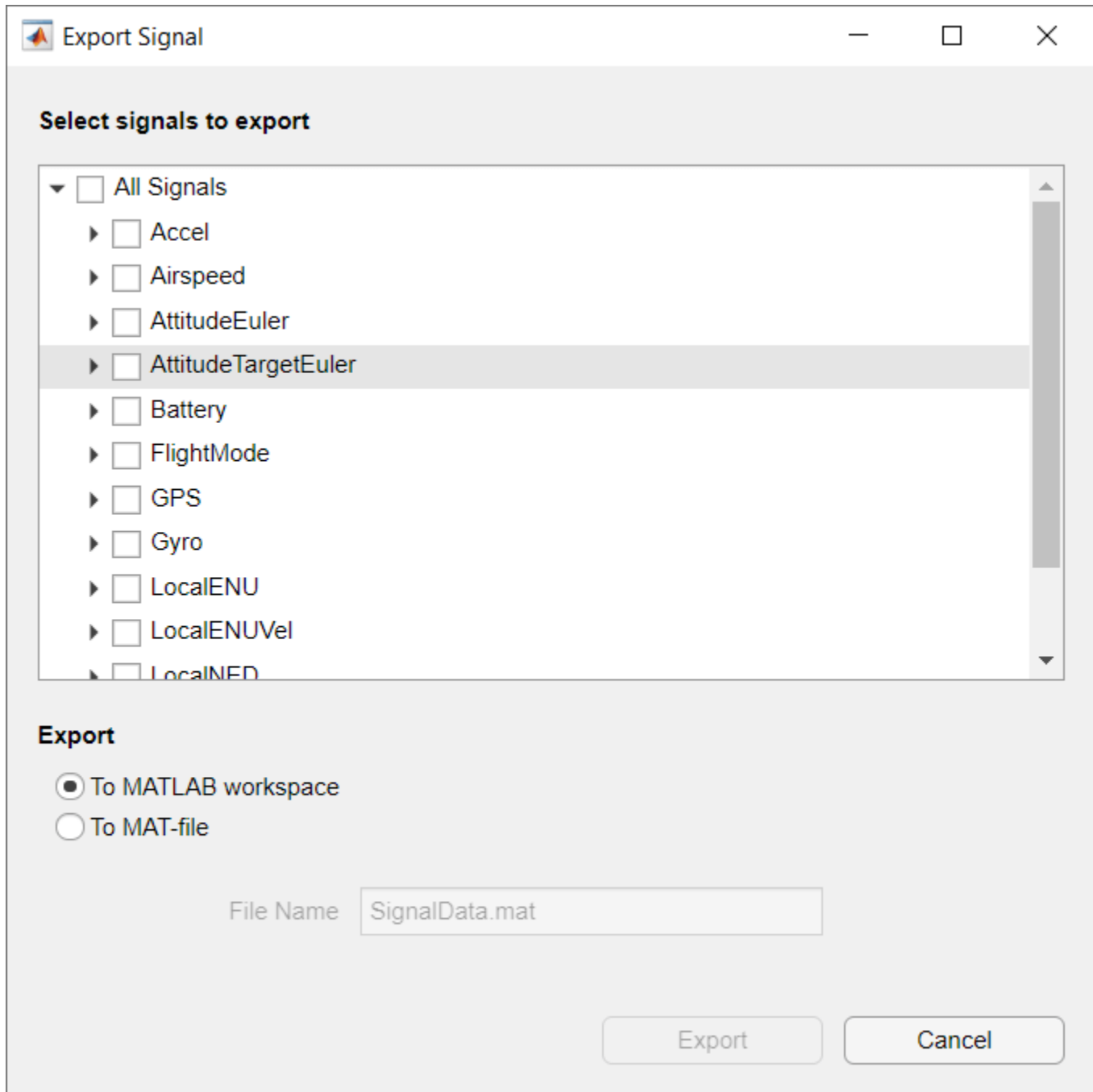
Use the **Export Signal** dialog box to select signals of interest and export them **To Workspace** or **To MAT-file**. The signals are exported as a timetable.

Using Custom Signal Mapping in Flight Log Analyzer App

The default signal mapping returns a predefined set of signals.

```
flsmObj = flightLogSignalMapping('ulog');
info(flsmObj,"Signal")
```

```
ans=18×4 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,
    "GPS"                        true          "Latitude, Longitude, Altitude, GroundSpeed, CourseAngle
    "Gyro"                       true          "GyroX, GyroY, GyroZ"
    "LocalENU"                   true          "X, Y, Z"
    "LocalENUTarget"             true          "XTarget, YTarget, ZTarget"
    "LocalENUVel"                true          "VX, VY, VZ"
    "LocalENUVelTarget"          true          "VXTarget, VYTarget, VZTarget"
    "LocalNED"                   true          "X, Y, Z"
    "LocalNEDTarget"             true          "XTarget, YTarget, ZTarget"
    "LocalNEDVel"                true          "VX, VY, VZ"
        ⋮
```

In addition the predefined set of signals, you can map other signals present in the flight log. For example, use `mapSignal` to map `WindSpeed` to the `flightLogSignalMapping` object, `flsmObj`.

```
mapSignal(flsmObj,"WindSpeed", ...
    @(data)getTime(getTable(data,"wind_estimate")), ...
    @(data)getModeValue(getTable(data,"wind_estimate")), ...
    ["WindSpeed_East","WindSpeed_North"]);
```

Check that `flsmObj` now contains the new signal.

```
info(flsmObj,"Signal")
```

```
ans=19×4 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,
    "GPS"                        true          "Latitude, Longitude, Altitude, GroundSpeed, CourseAngle
    "Gyro"                       true          "GyroX, GyroY, GyroZ"
    "LocalENU"                   true          "X, Y, Z"
    "LocalENUTarget"             true          "XTarget, YTarget, ZTarget"
    "LocalENUVel"                true          "VX, VY, VZ"
    "LocalENUVelTarget"          true          "VXTarget, VYTarget, VZTarget"
    "LocalNED"                   true          "X, Y, Z"
    "LocalNEDTarget"             true          "XTarget, YTarget, ZTarget"
    "LocalNEDVel"                true          "VX, VY, VZ"
        ⋮
```

To use this custom signal mapping in the **Flight Log Analyzer** app:

1 On the app toolstrip, click **Import** and select **From Workspace**.

2 In the dialog box, select `flsmObj` from the **Signal Mapping** list.

3 Select the `ulogreader` object `ulg` from the **Log Data** section.

4 Click **Import**.
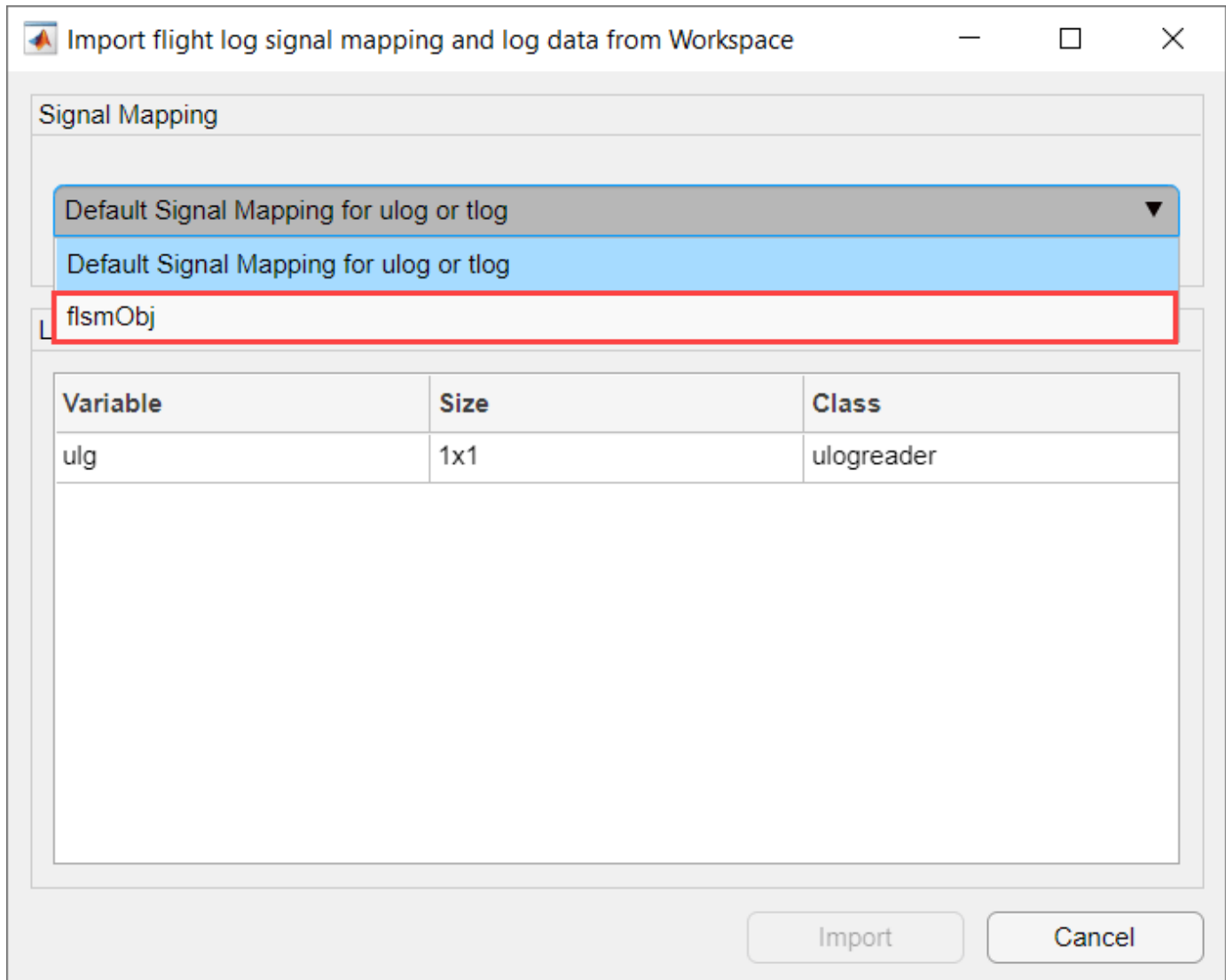
Create a custom **Timeseries** plot and follow the steps in Add Custom Plot on page 1-0 to add the `WindSpeed_East` and `WindSpeed_North` signals from the **Signal Browser**.
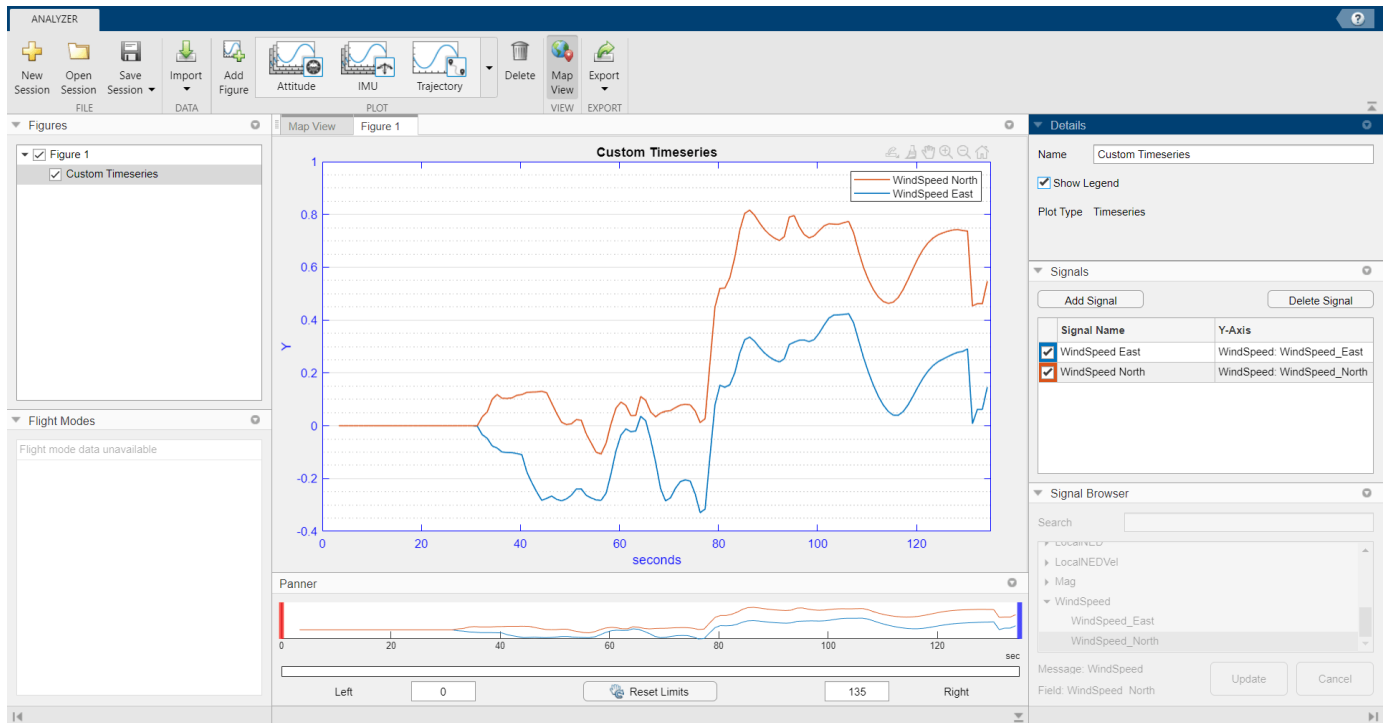
You can use this process to map other signals to the custom signal mapping object and visualize them in the app.

References

[1] PX4 Autopilot. "Flight Log Analysis." PX4 User Guide. Accessed December 14, 2020. https://docs.px4.io/master/en/log/flight_log_analysis.html

[2] PX4 Autopilot. "Log Analysis Using Flight Review." PX4 User Guide. Accessed December 14, 2020. https://docs.px4.io/master/en/log/flight_review.html

Tuning Waypoint Follower for Fixed-Wing UAV

This example designs a waypoint following controller for a fixed-wing unmanned aerial vehicle (UAV). The Guidance Model and Waypoint Follower blocks are the main components that simulate the UAV guidance model and generate commands for following waypoints.

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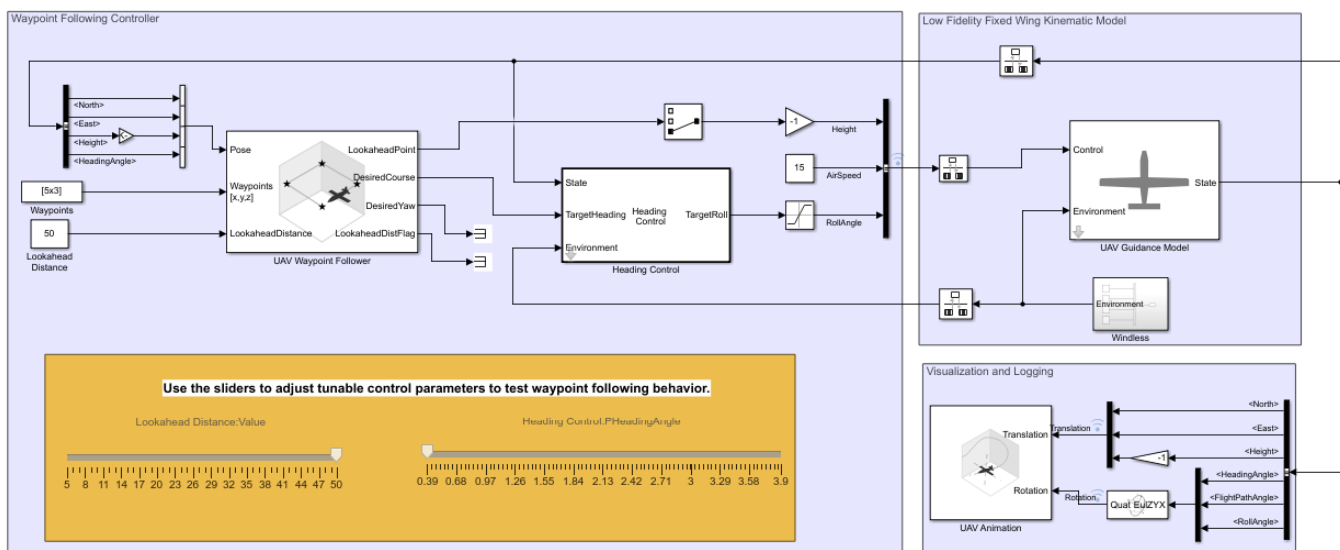
The UAV Waypoint Follower block computes a desired heading for the UAV based on the current pose, lookahead distance, and a given set of waypoints. Flying along these heading directions, the UAV visits each waypoint (within the specified transition radius) in the list.

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")
```
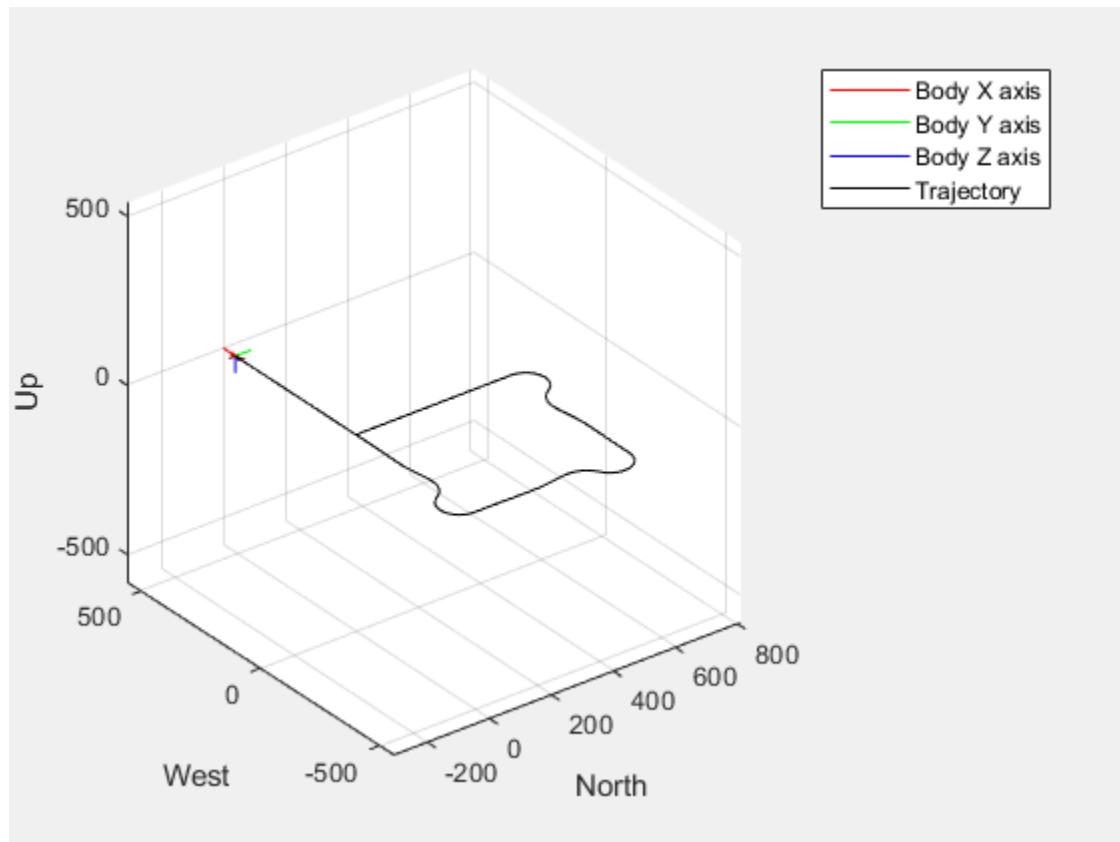

The next figures shows the flight behavior with a small lookahead distance (5) and a fast heading control (3.9). Notice the UAV follows a very curvy path between the waypoints.

Lookahead Distance:Value

|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|
5   8 11 14 17 20 23 26 29 32 35 38 41 44 47 50

Heading Conrol:PHeadingAngle

|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|
0.39 0.69 0.99 1.29 1.59 1.89 2.19 2.49 2.79 3.09 3.39 3.69 3.9
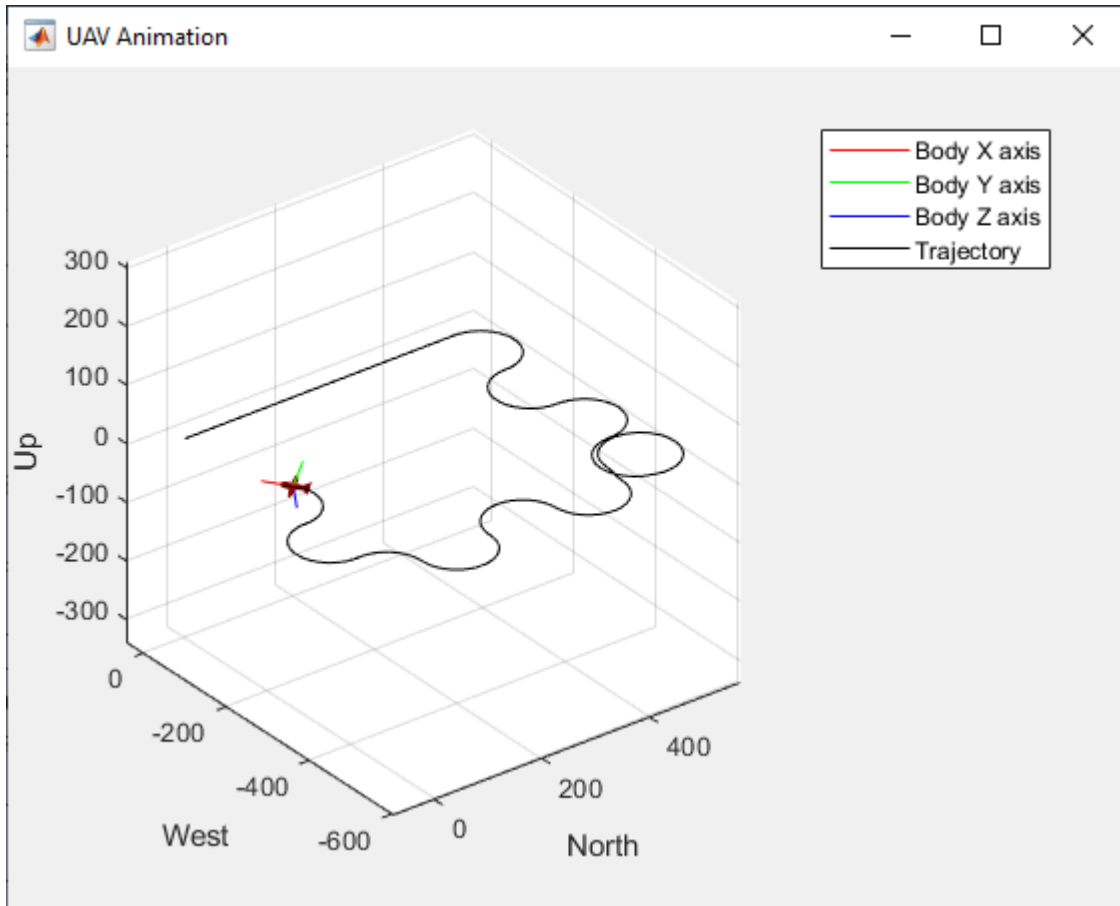
The next figure shows the flight behavior with a large lookahead distance and slow heading control.

Lookahead Distance:Value

|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|'''|
5   8 11 14 17 20 23 26 29 32 35 38 41 44 47 50

Heading Conrol:PHeadingAngle

|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|'''''|
0.39 0.69 0.99 1.29 1.59 1.89 2.19 2.49 2.79 3.09 3.39 3.69 3.9
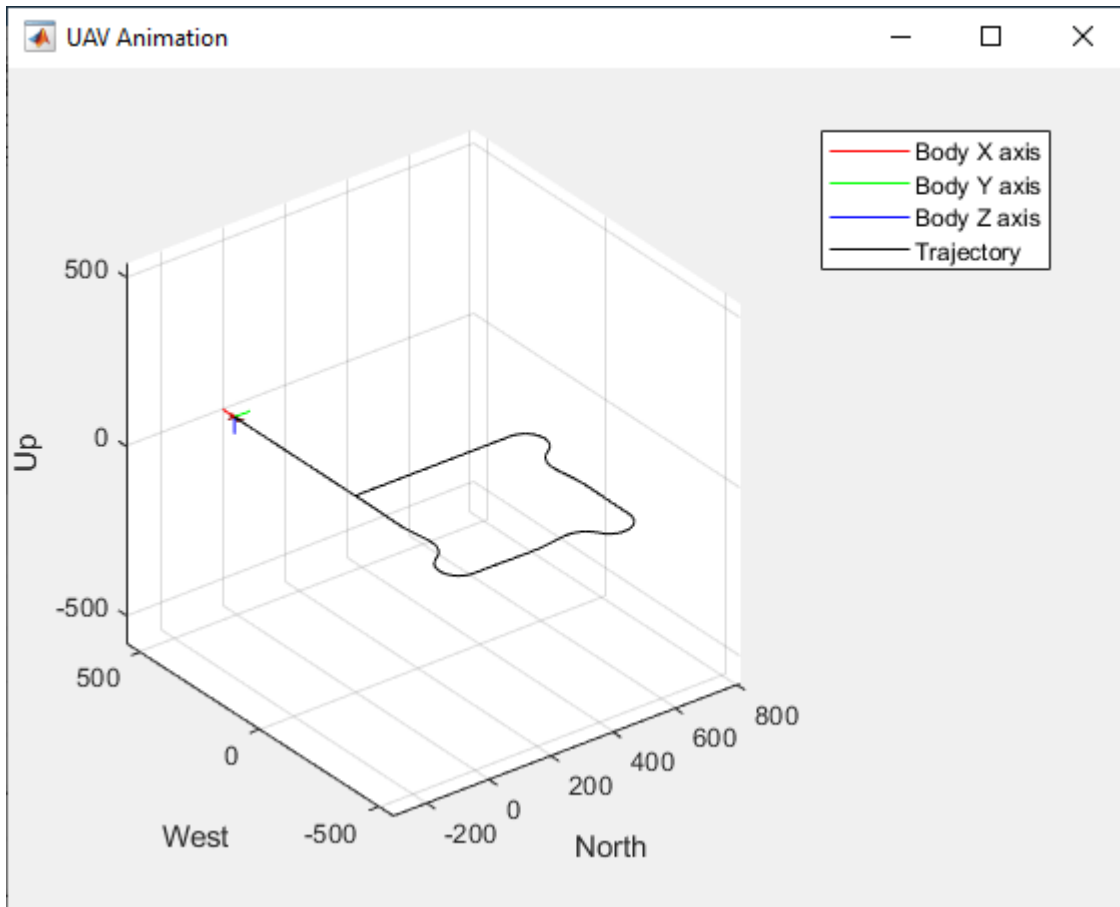
Summary

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 1-37. 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.sldd');
plantDataSet = getSection(plantDataDictionary,'Design Data');

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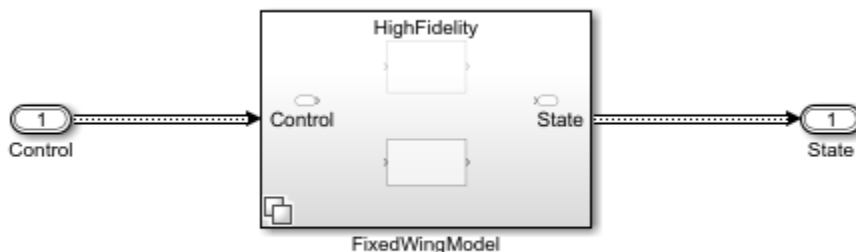


```
% Switch to low-fidelity model
assignin(plantDataSet,'useHighFidelity',0);
```
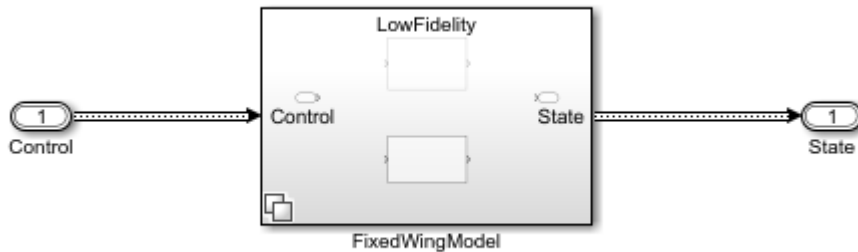
Approximate high-fidelity fixed-wing model with low-fidelity guidance model

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)
```
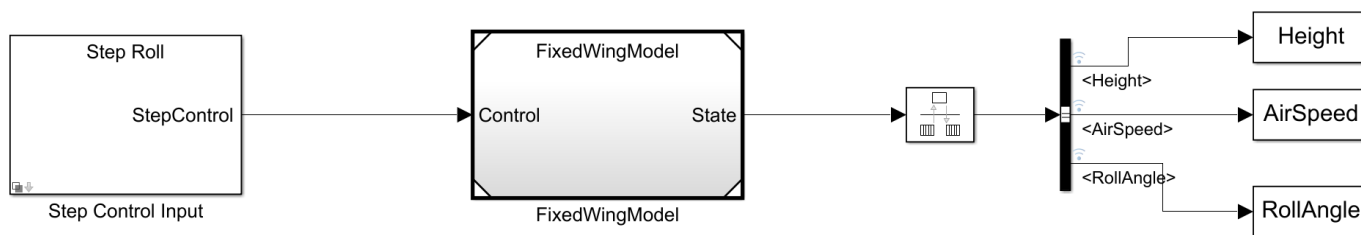

First, command a change in roll angle.

```
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 3m 25.623s
```

```
highFidelityRollAngle = RollAngle.Data(:);
highFidelityTime = RollAngle.Time;

figure()
plot(highFidelityTime, highFidelityRollAngle,'--r');
title('Roll Angle Step Response')
```
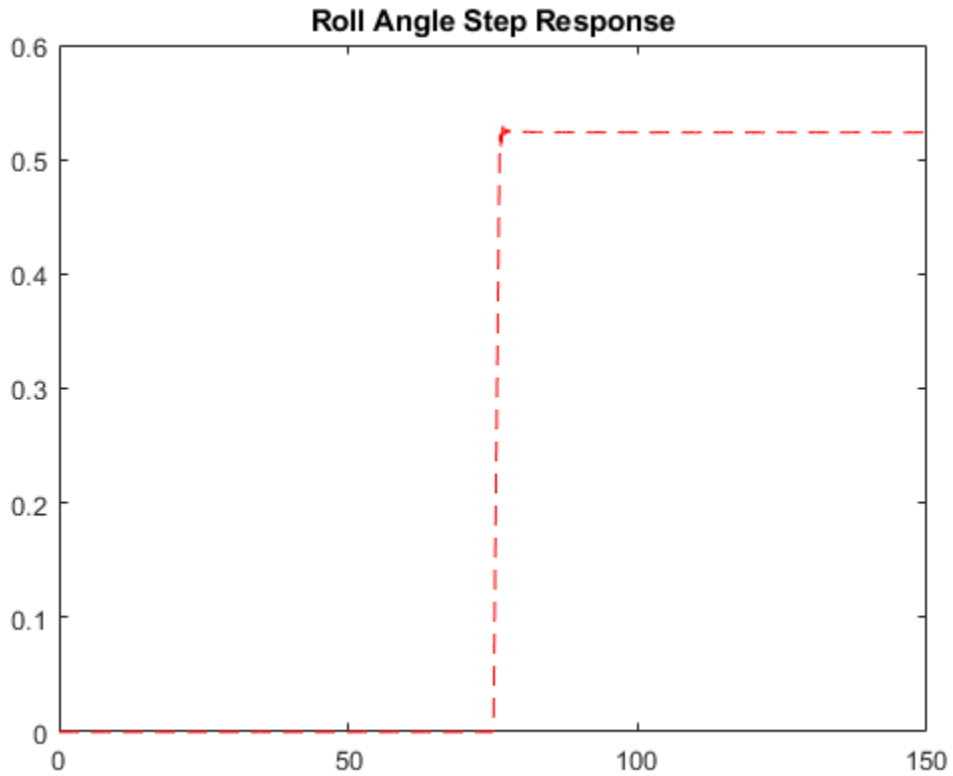

Zooming into the simulation result above, you see the characteristics of the roll angle controller built into the high-fidelity model. The settling time for the roll angle is close to 2.5 seconds.

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

Roll Angle Step Response
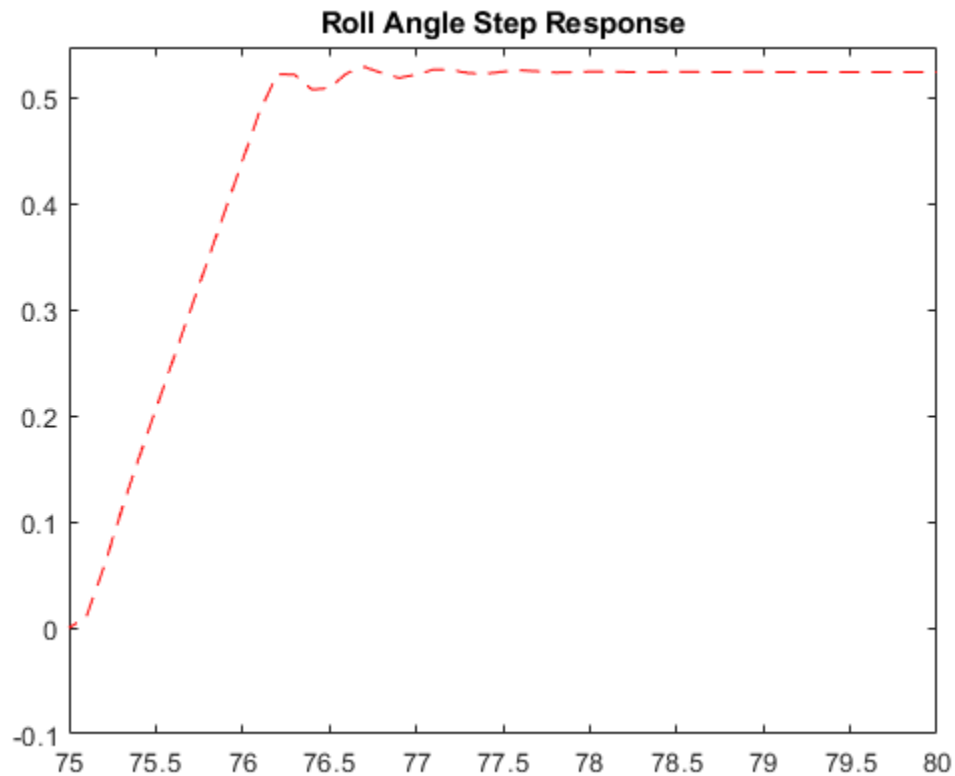
For a second-order PD controller, to achieve this settling time with a critically damped system, the following gains should be used to configure the UAV Guidance Model block inside the low-fidelity variant of the UAV model. For this example, the **UAV Guidance Model** block is simulated using code generation 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 32.72s
```

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

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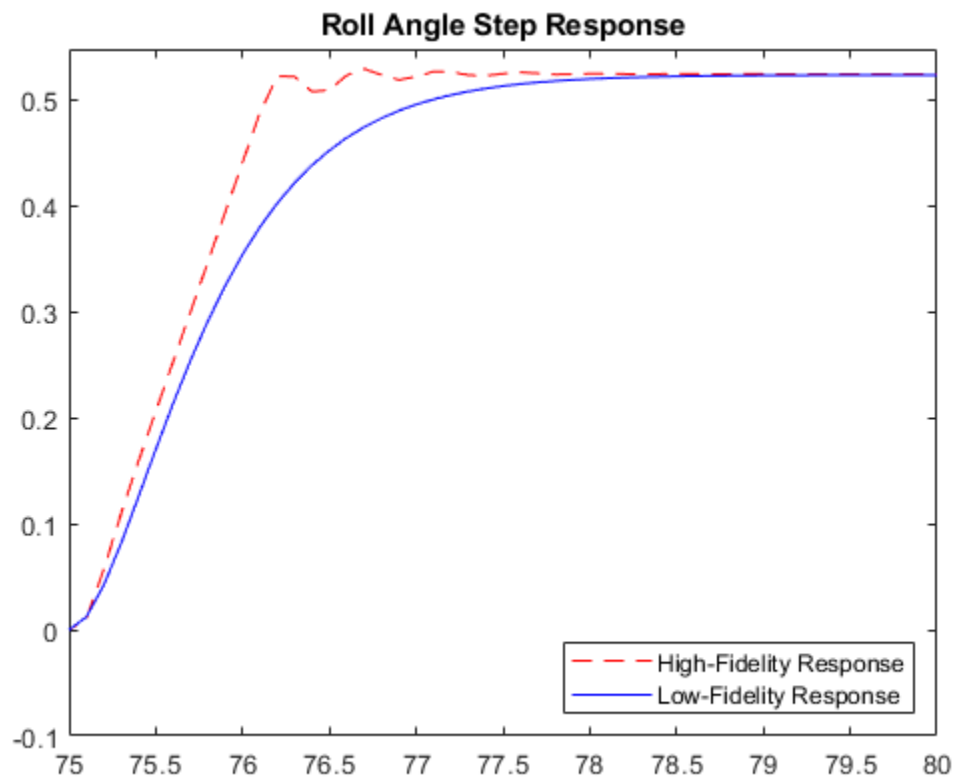

The low-fidelity model achieves a similar step response. Similarly, we can tune the other two control channels: `Height` and `AirSpeed`. More sophisticated methods can be used here to optimize the control gains instead of visual inspection of the control response. Consider using System Identification 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 4.191s

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 chang

1 of 2 models built (1 models already up to date)
Build duration: 0h 1m 2.104s

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])
```

Air Speed Step Response

Legend: Low-Fidelity Response; High-Fidelity Response
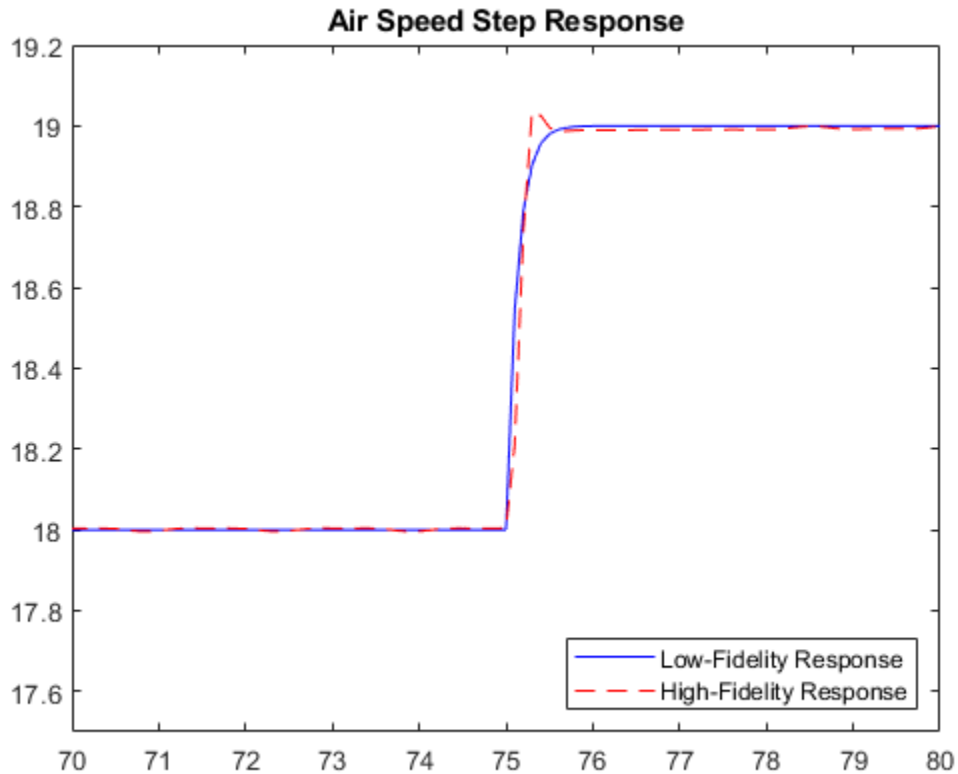
```
controlBlock.StepControl = 'Height Step Control';
assignin(plantDataSet, 'useHighFidelity', 0);

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 chang

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 27.077s

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 chang

1 of 2 models built (1 models already up to date)
Build duration: 0h 1m 1.299s
```

```matlab
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])
```
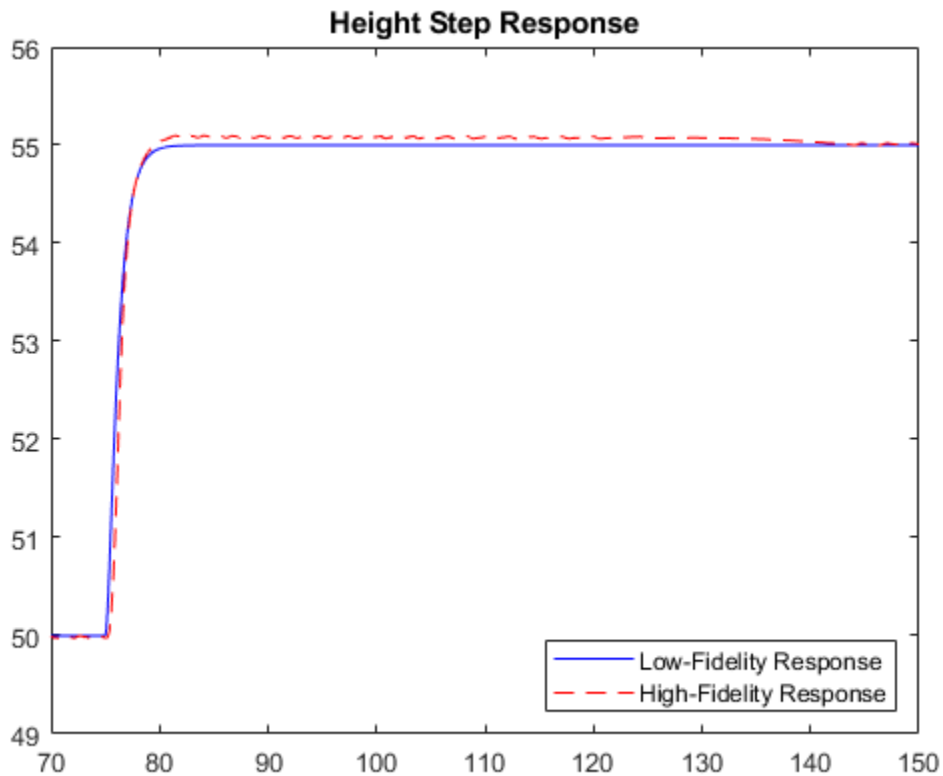
Test navigation algorithm with low-fidelity model

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 1-37 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 chang

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 30.518s


figure
visualizeSimStates(simStates);
```
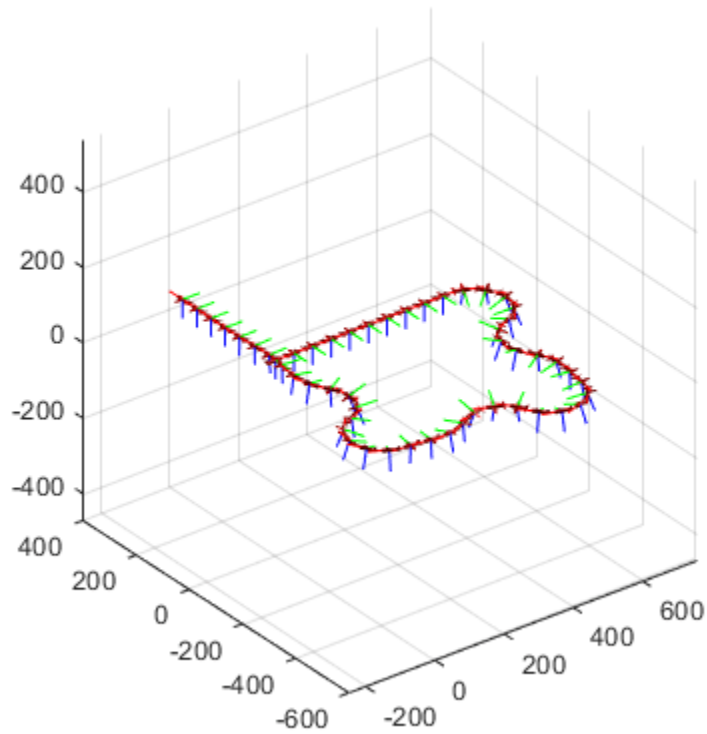
Validate with high-fidelity model

```
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 chang

1 of 2 models built (1 models already up to date)
Build duration: 0h 1m 1.491s
```

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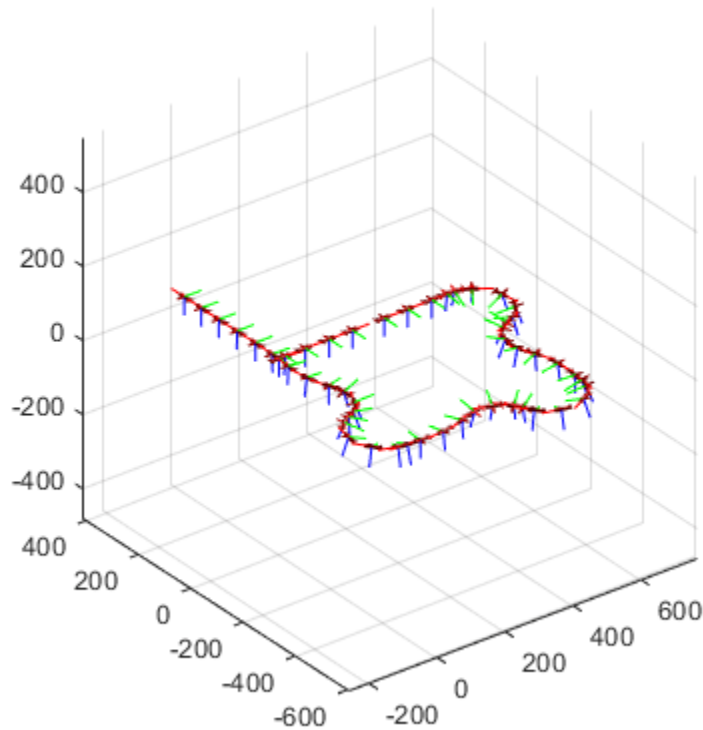
Conclusion

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 senarios 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 1-37
- "Explore Simulink Bus Capabilities" (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.

```matlab
% 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. The occupancy map is in an ENU (East-North-Up) frame.

```matlab
mapData = load("uavMapCityBlock.mat","omap");
omap = mapData.omap;
% Consider unknown spaces to be unoccupied
omap.FreeThreshold = omap.OccupiedThreshold;
```

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

```matlab
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])
```

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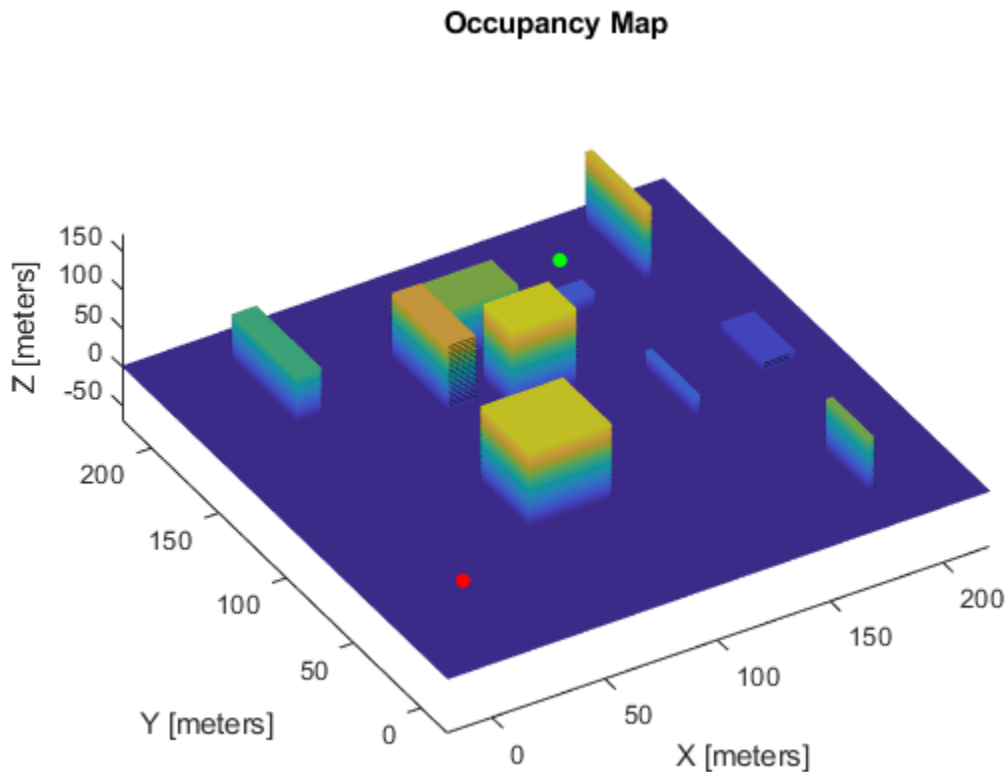
Plan a Path with RRT Using 3-D Dubins Motion Primitives

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")
```

```matlab
    % 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
```


Occupancy Map
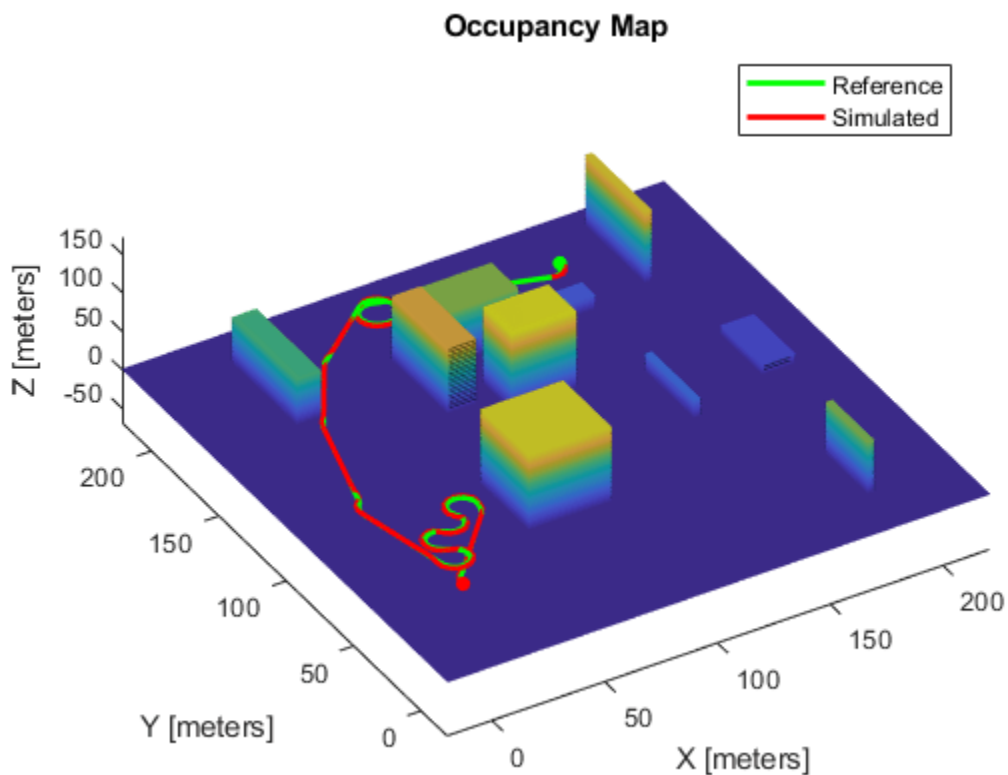
Smooth Dubins Path and Simulate UAV Trajectory

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.

```matlab
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
```
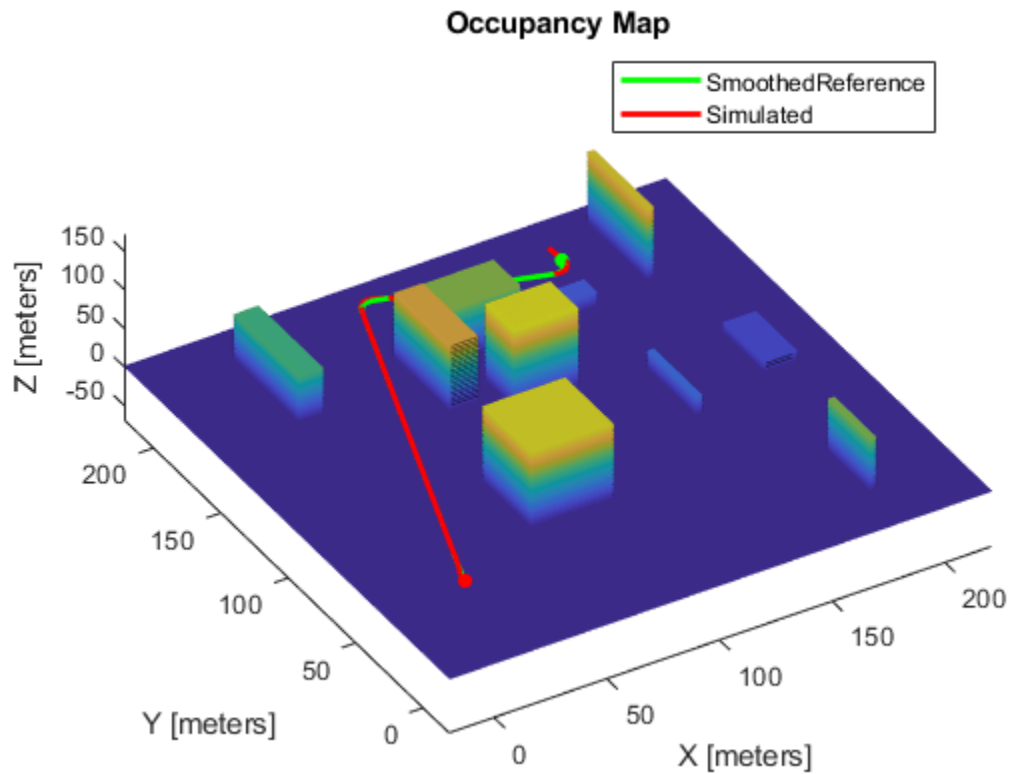
Occupancy Map

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.

Transition from Low to High-Fidelity UAV Models in Three Stages

This example shows how to continuously evolve your UAV plant model to keep in sync with the latest information available.

Background

An unmanned aerial vehicle (UAV) design cycle provides incrementally better access to UAV characteristics as the design progresses. By increasing its fidelity, this information can be used to continuously evolve a plant model through a Model Based Design approach.
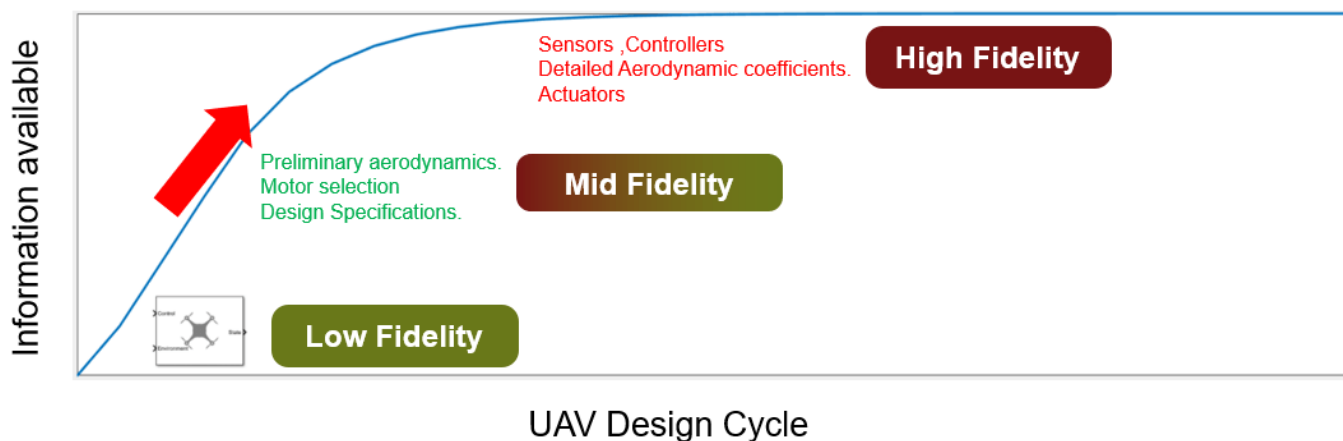
UAV Design Cycle

Towards the end of the design cycle, there is enough information to develop a high-fidelity plant. To accurately model the UAV, a high-fidelity model incorporates modeling all forces and moments, wind and environmental effects and sensors in detail. However, this level of information may be unavailable to a designer early in the design process. To build such a complex model, it can take several flight and wind tunnel tests to create enough detailed aerodynamic coefficients to compute all forces and moments that affect the UAV. These factors can potentially block guidance algorithm design until the end of the design process, when a more realistic estimate of UAV dynamics is obtained.

To concurrently design a guidance algorithm sooner, a UAV algorithm designer can start with a low-fidelity model and evolve their plant model as and when additional data becomes available.

Designing a guidance algorithm using only a low-fidelity model can also pose a risk. Without controller or aerodynamic constraints, an optimistic guidance technique can fail for a real UAV with slower aircraft dynamics.

This example highlights an alternative approach. You progress from the low-fidelity Guidance Block to a medium and then high-fidelity model by progressively adding layers of control and dynamics to the simulation. In this process, the medium-fidelity model becomes a useful tool for leveraging limited information about a plant model to tune and test guidance algorithms.

The medium-fidelity model is thus used to test a given path following an algorithm. Since the high-fidelity model is unavailable until the end of the design process, the high-fidelity model is only used later to validate our modelling approach by comparing step response and path following behavior.

Open Example and Project Files

To access the example files, click **Open Live Script** or use the `openExample` function.

```
openExample('shared_uav_aeroblks/UAVFidelityExample')
```

Open the Simulink™ project provided in this example.

```
cd fidelityExample
openProject('fidelityExample.prj')
```

The project contains three versions of a UAV model, low-fidelity, medium-fidelity and high-fidelity with steps to study their step response and path following behaviour.

Low-Fidelity Model

Assume your UAV has the following design specifications shown in the table below. The low-fidelity variant provided in this model is tuned to achieve the desired response, but you can tune these gains for your specific requirements. The low-fidelity plant uses the UAV Guidance Block which is a reduced order model for a UAV. To run the low-fidelity variant, click the **Simulate Plant** shortcut under the **Low Fidelity** group of the project toolstrip.

Design Specification	Response Time (within 2%)	Step Change
Roll	2.5 seconds	30 degrees
Height	4.5 seconds	5 m
Airspeed	0.6 seconds	1 m/s

This shortcut sets the FidelityStage parameter to 1, configures the FidelityStepResponse model to simulate the low-fidelity model, and outputs the step response. The step response is computed for height, airspeed, and roll response.
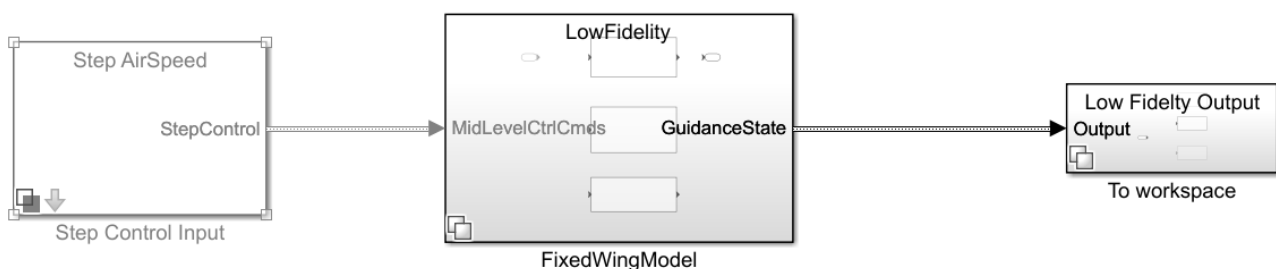

Copyright 2021 The MathWorks, Inc.

Open the UAV Fixed Wing Guidance Model block in the **FidelityStepResponse/FixedWingModel/LowFidelity** subsystem. In the Configuration tab, inspect the gains set for height, airspeed, and roll response. This guidance block integrates the controller with the dynamics of the aircraft. The low-fidelity variant gives a first estimate of how fast the UAV can realistically respond to help tune high-level planners.

Medium-Fidelity Model

As the UAV design progresses, the lift and drag coefficients become available. A motor for the aircraft is selected by the user, which defines the thrust curves. To test the validity of the guidance algorithm against this new information, the example adds this information to the plant model in this step.

To design a medium-fidelity model, the model needs only preliminary aerodynamic coefficients, thrust curves, and response time specifications. To model a medium-fidelity UAV, you can use the Fixed-Wing Point Mass Block. The block only requires lift, drag and thrust force inputs, which are much easier to approximate at an early design stage than detailed forces and moments of an aircraft. To set up the medium-fidelity variant, click the **Setup Plant** shortcut under the **Medium Fidelity** group of the project toolstrip.

Examine the Vehicle Dynamics tab in the model under **FidelityStepResponse/ FixedWingModel/Mid Fidelty/UAV Plant Dynamics/Vehicle Dynamics**.
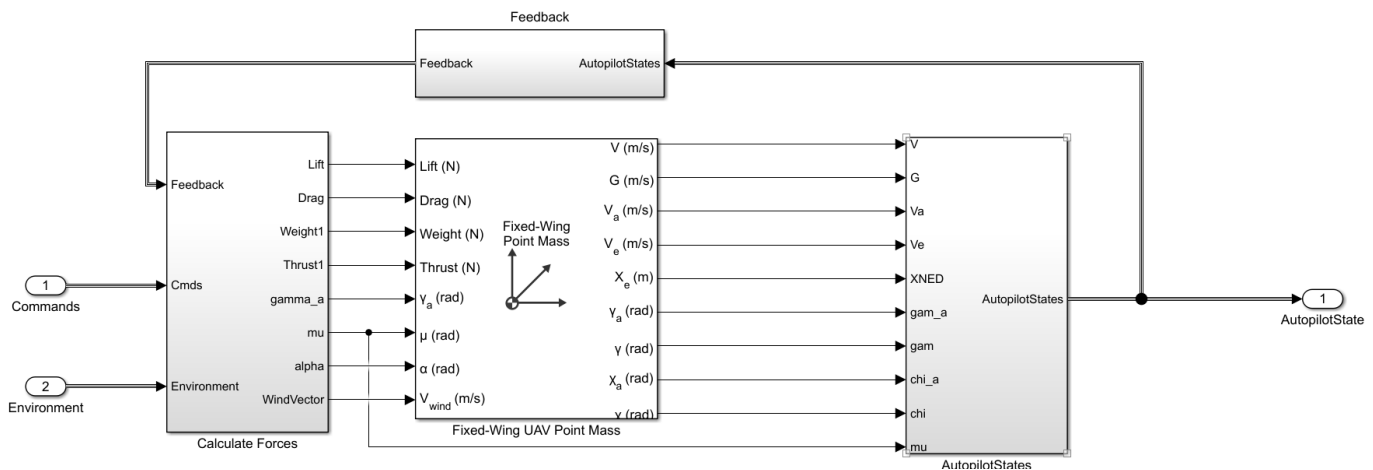
The medium-fidelity model represents the UAV as a point mass with the primary control variables being the angle of attack and roll. This medium-fidelity plant model takes in roll, pitch, thrust as control inputs. The point mass block assumes instantaneous dynamics of roll and angle of attack. This model uses a transfer function to model roll lag based on our roll-response specification shared in the table within the previous step.

The medium-fidelity aircraft controls pitch instead of angle of attack. Since the angle of attack is an input to the point mass block, the plant model converts pitch to alpha using the following equation.

$\Theta = \gamma_a + \alpha$

Θ, γ_a and α represent pitch, flight path angle in the wind frame, and angle of attack respectively.

Unlike the low-fidelity model, the medium-fidelity model splits the autopilot from the plant dynamics. The medium-fidelity plant needs an outer-loop controller for height-pitch and airspeed-throttle control to be added. The predefined controllers provided are using standard PID-tuning loops to reach satisfactory response without overshoot. To inspect the outer-loop controller, open the `Outer_Loop_Autopilot` Simulink model.

Medium-Fidelity Step Response

The low-fidelity plant was tuned in the previous step by assuming that all response time specifications are met by the UAV. To test this assumption, use the medium-fidelity plant. The study of the step

response of the improved plant is used to contrast the performance of the low-fidelity and medium-fidelity variant. To simulate the medium-fidelity step response, click the **Simulate Plant** shortcut under the **Medium Fidelity** group of the project toolstrip. The step response plots appear as figures.

Notice that the model meets the design criteria shown in the table below by achieving an air speed settling time of 0.6 seconds and a height response of 4.1 seconds. However, the height response is slower than the low-fidelity variant. This lag in response is expected due to the additional aerodynamic constraints placed on the medium-fidelity plant.

Design Specification	Response Time (within 2%)		Step Change
Roll	2.5 seconds		30 degrees
Height	4.5 seconds	✔	5 m
Airspeed	0.6 seconds	✔	1 m/s

### Simulate Path Following Algorithm

With a more accurate response from the UAV medium-fidelity model, you can now test waypoint follower or guidance algorithms to follow waypoints. For the guidance algorithm design, see the "Tuning Waypoint Follower for Fixed-Wing UAV" example.

To simulate and visualize the medium-fidelity UAV path following the model, click the **Simulate Path Follower** shortcut under the **Medium Fidelity** group of the project toolstrip.

Tune Waypoint Follower for Fixed-Wing UAV
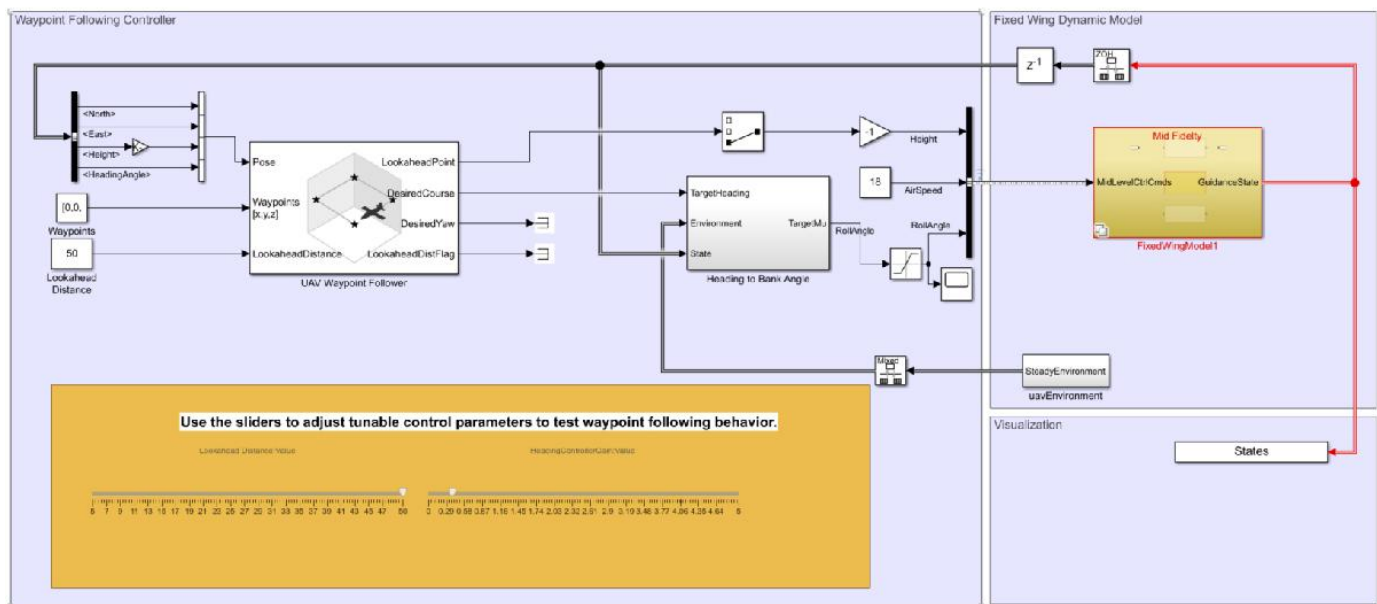
Notice that the medium-fidelity UAV follows the desired path accurately.
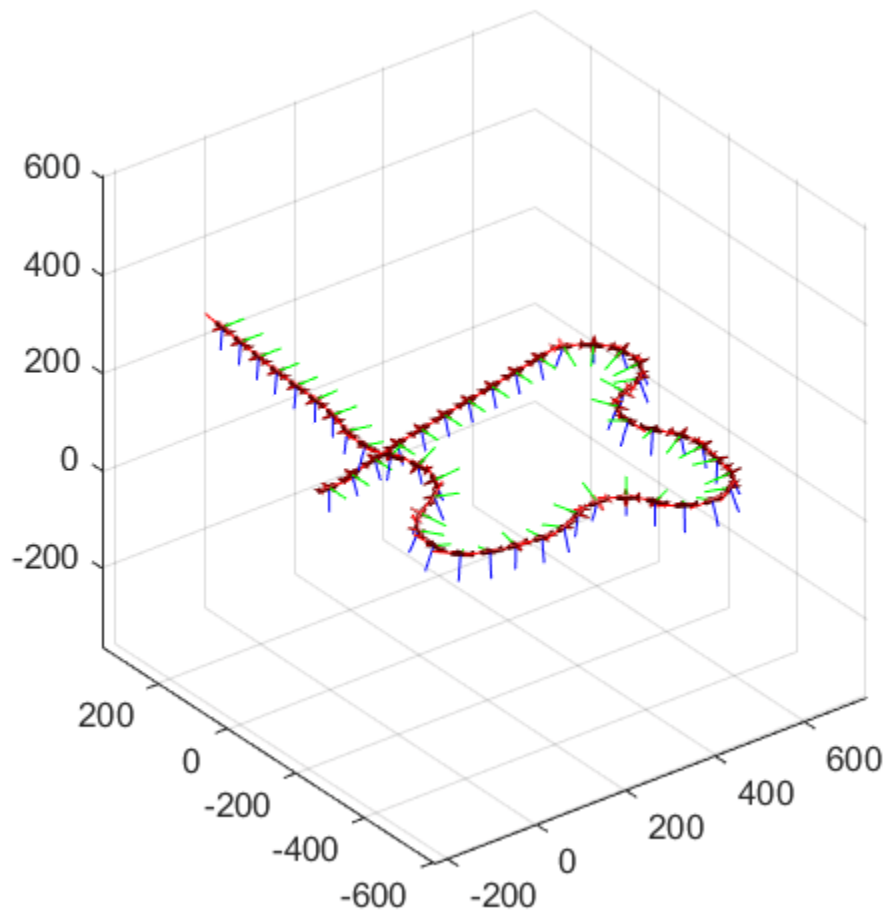
High-Fidelity Step Response

The medium-fidelity model was used to test a path follower design using simple aircraft parameters available at an early design state. However, it is important to continue adding fidelity to capture UAV control response to study more complex situations. For example, the use of more detailed aerodynamics coefficients allows analysis of complex motions such as doublet maneuvers. Another example is, adding actuator dynamics lets you study the subsequent effect on inner loop controllers for attitude, which can cause destabilization. In this way, the high-fidelity plant allows refinement of control system design. In this step, to study the change in response, we look at a high-fidelity plant with these added dynamics.

The high-fidelity plant inputs all forces and moments to a 6-DOF block, adds on-board sensors, and models actuator dynamics for the UAV. Unlike the mid-fidelity plant, the high-fidelity version does not take attitude inputs directly. Instead, an inner loop controller is added to control attitude. Additionally, a yaw compensation loop balances the non-zero sideslip. The model reuses the outer-loop controller designed for the medium-fidelity model. To validate that the medium-fidelity model provided useful intermediate information, use the response of the higher fidelity model.

To simulate and visualize the high-fidelity step response, click the **Simulate Plant** shortcut under the **High-Fidelity** group of the project toolstrip. Notice that despite added complexity, the trajectory matches well with the medium-fidelity model. Also, notice the design specifications are relatively the same for the high-fidelity stage. This similarity shows that the medium-fidelity plant modelled UAV dynamics accurately.

Design Specification	Response Time (within 2%)	Step Change
Roll	2.5 seconds	30 degrees
Height	~~4.1~~ **3.9 seconds**	5 m
Airspeed	0.6 seconds	1 m/s

Simulate Path Following Algorithm for High-Fidelity

Towards the end of the design cycle, the high-fidelity model finally becomes available. To get the final UAV path following characteristics, you can now test the guidance algorithm developed in previous steps on the high-fidelity plant. Click the **Simulate Path Follower** shortcut under the **High-Fidelity** group of the project toolstrip.
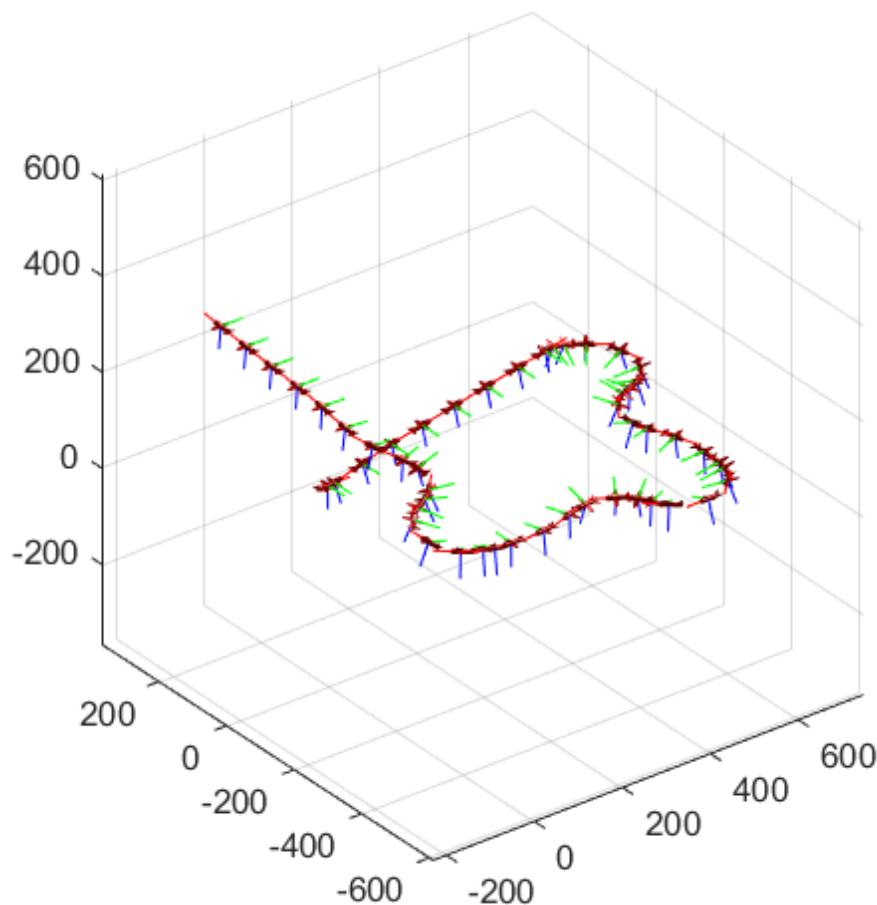
Notice that the model obtains a similar response to the medium-fidelity model using the guidance and outer-loop control parameters. This validates the guidance algorithm with a high-fidelity plant.

Conclusion

The medium-fidelity model accurately predicts the UAV dynamics making optimum use of limited information available during design. The example designs the outer loop controller and tests a waypoint follower without needing all the information in a high-fidelity plant model.

To model additional dynamics such as actuator lag, the medium-fidelity plant is flexible and can continuously evolve alongside design. The example obtains results under zero-wind conditions. In the presence of wind disturbances, the controller and path follower performance tracking might be adversely affected. To augment the autopilot controller to compensate for wind effects, leverage the atmospheric wind model in the high-fidelity plant model.

See Also

Blocks
Guidance Model | Fixed-Wing UAV Point Mass

Related Examples

- "Tuning Waypoint Follower for Fixed-Wing UAV" on page 1-37

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.
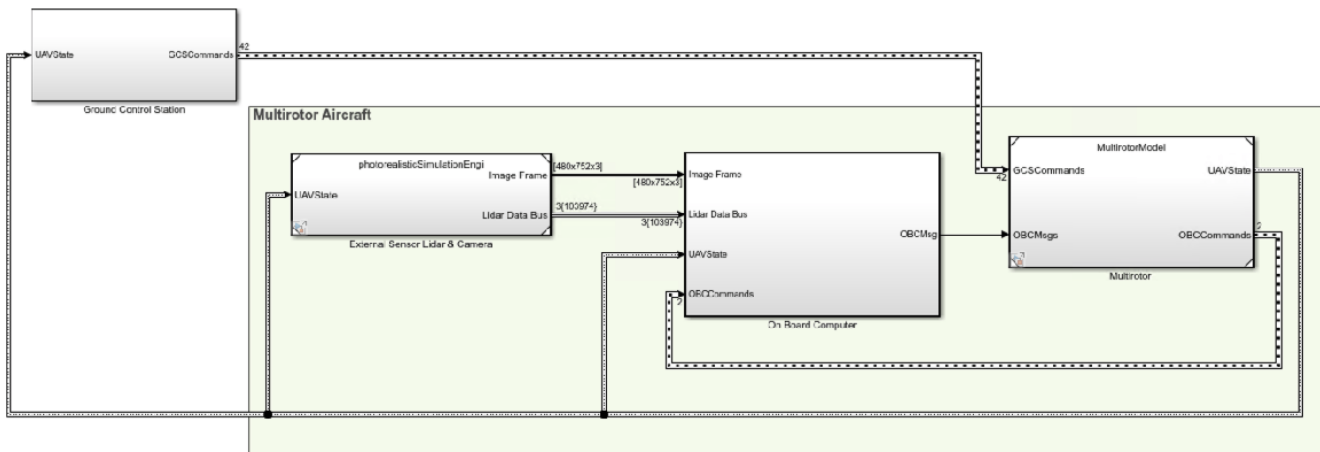
Open the Project

To get started, open the example live script and access the supporting files by either clicking **Open Live Script** in the documentation or using the `openExample` function.

```
openExample('uav/UAVPackageDeliveryExample');
```

Then, open the Simulink™ project file.

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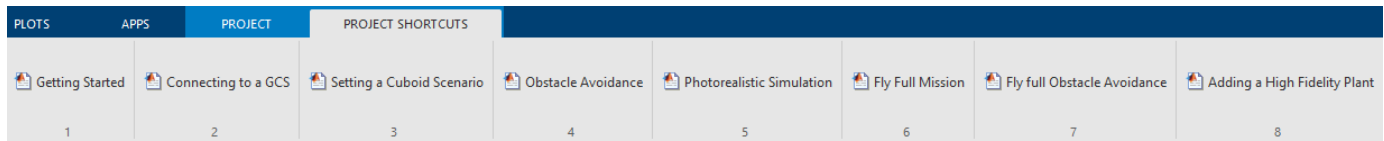

Model Architecture and Conventions

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 **Multirotor:** 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.sldd`). 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					
Getting Started	Connecting to a GCS	Setting a Cuboid Scenario	Obstacle Avoidance	Photorealistic Simulation	Fly Full Mission	Fly full Obstacle Avoidance	Adding a High Fidelity Plant	
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.
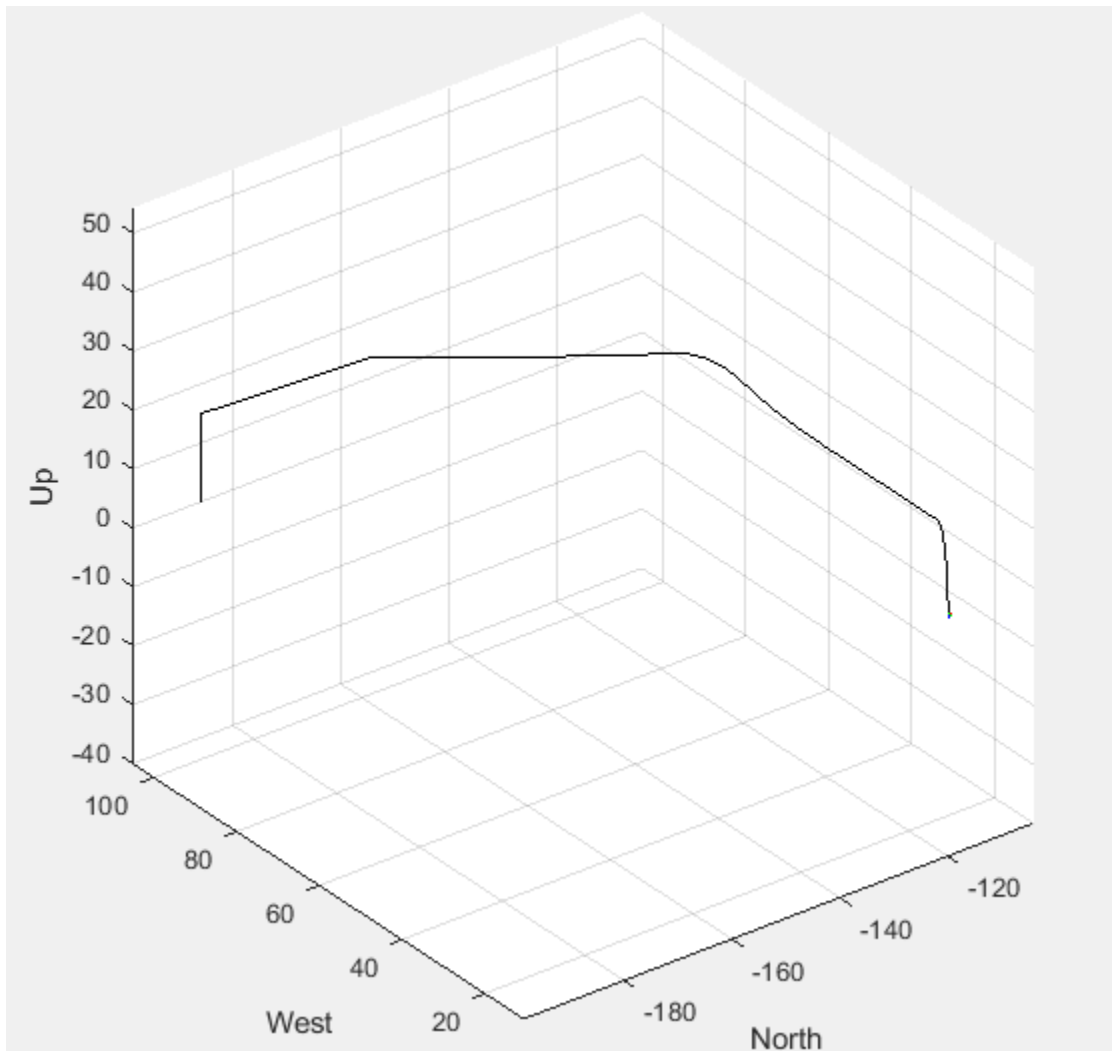
The model uses **UAV Path Manager** block to determine which is the active waypoint throughout the flight. The active waypoint is passed into the **Guidance Mode Selector** Stateflow™ chart to generate the necessary inner loop control commands.
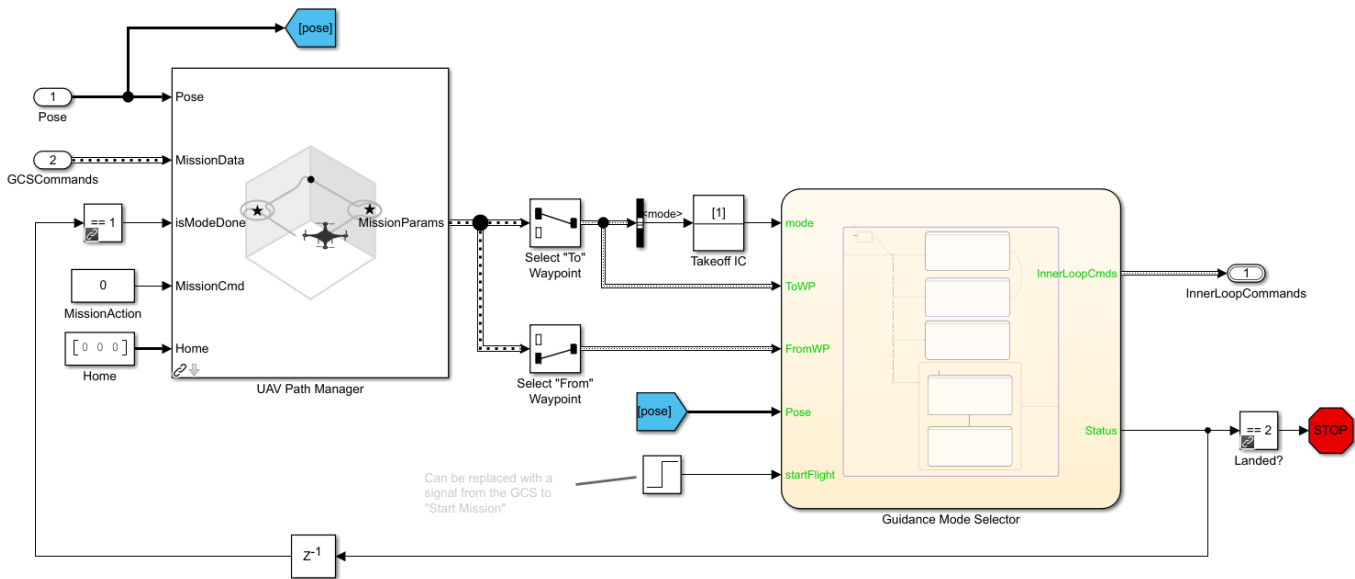
2. Connecting to a GCS

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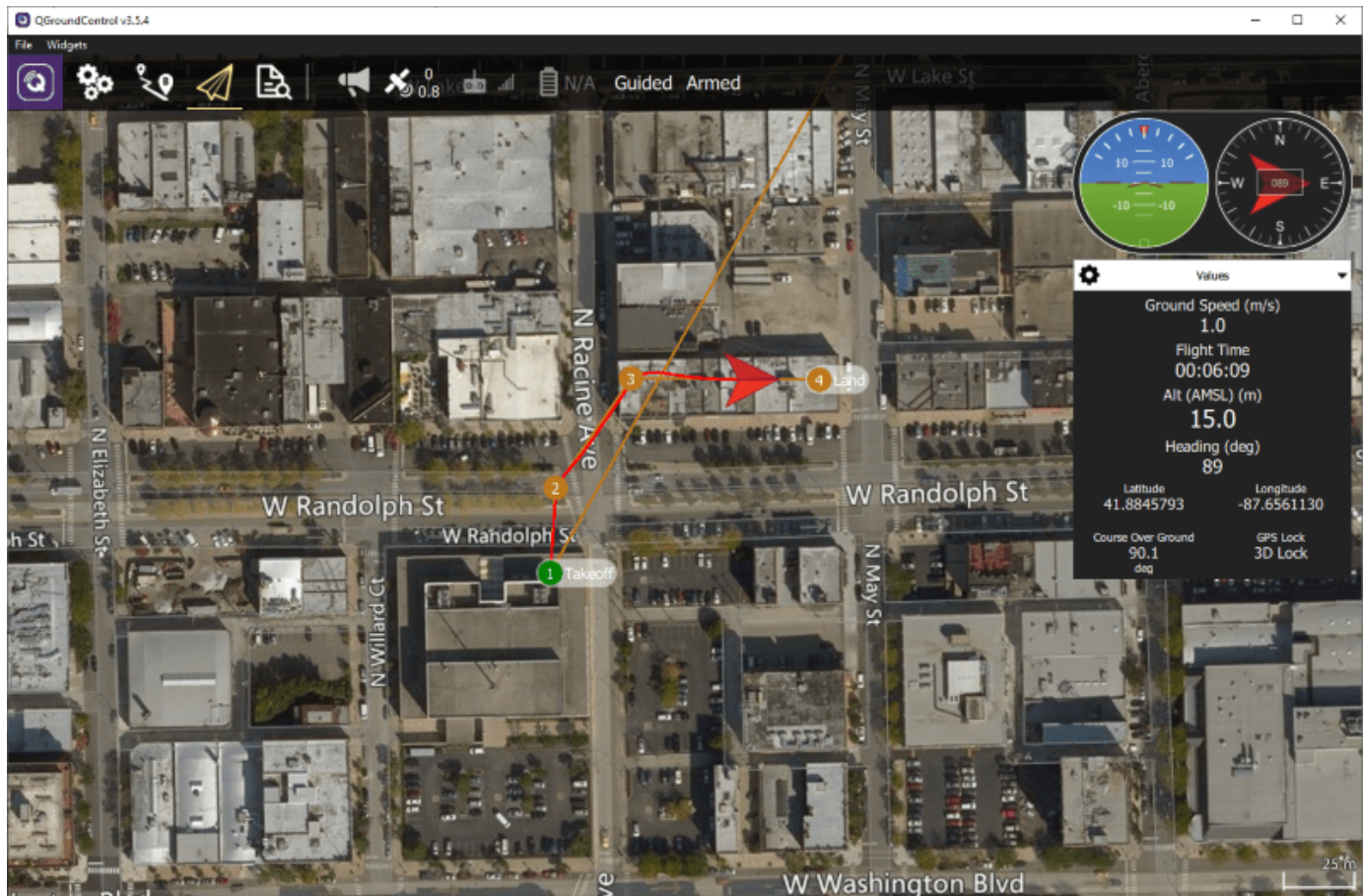
You can modify the mission by adding waypoints or moving those that are already in the mission. Upload the mission and the aircraft should respond to these changes.

3. Setting a Cuboid Scenario

Now that aircraft's model can be flow 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.
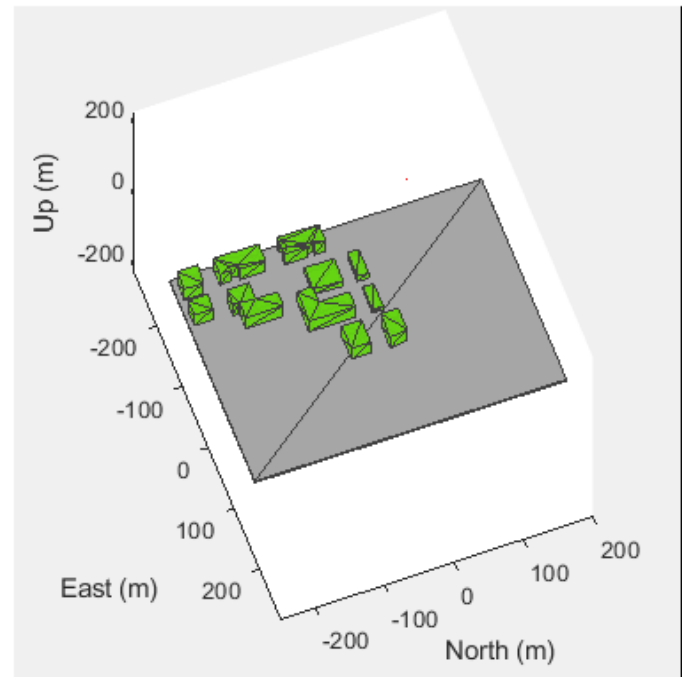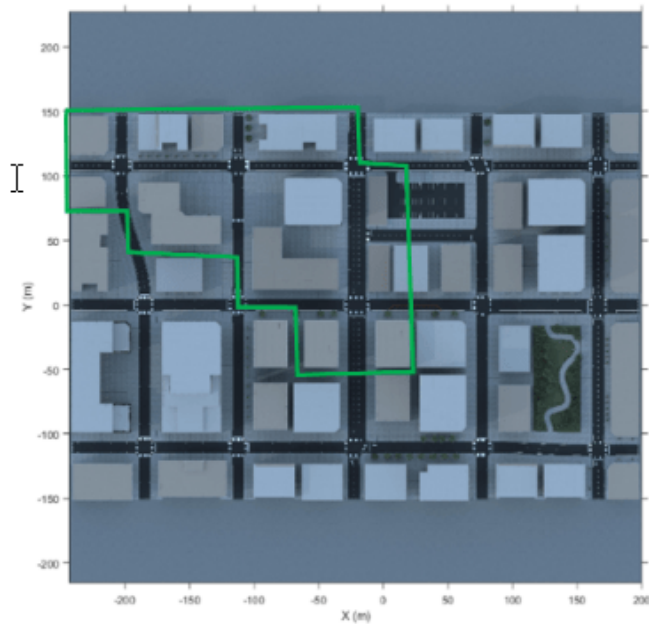
To safely fly the aircraft in this type of scenario, you need a sensor that provides information about the environment such as a lidar sensor to the model. This example uses a `uavLidarPointCloudGenerator` object added to the UAV scenario with a `uavSensor` object. The lidar sensor model generates readings 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:
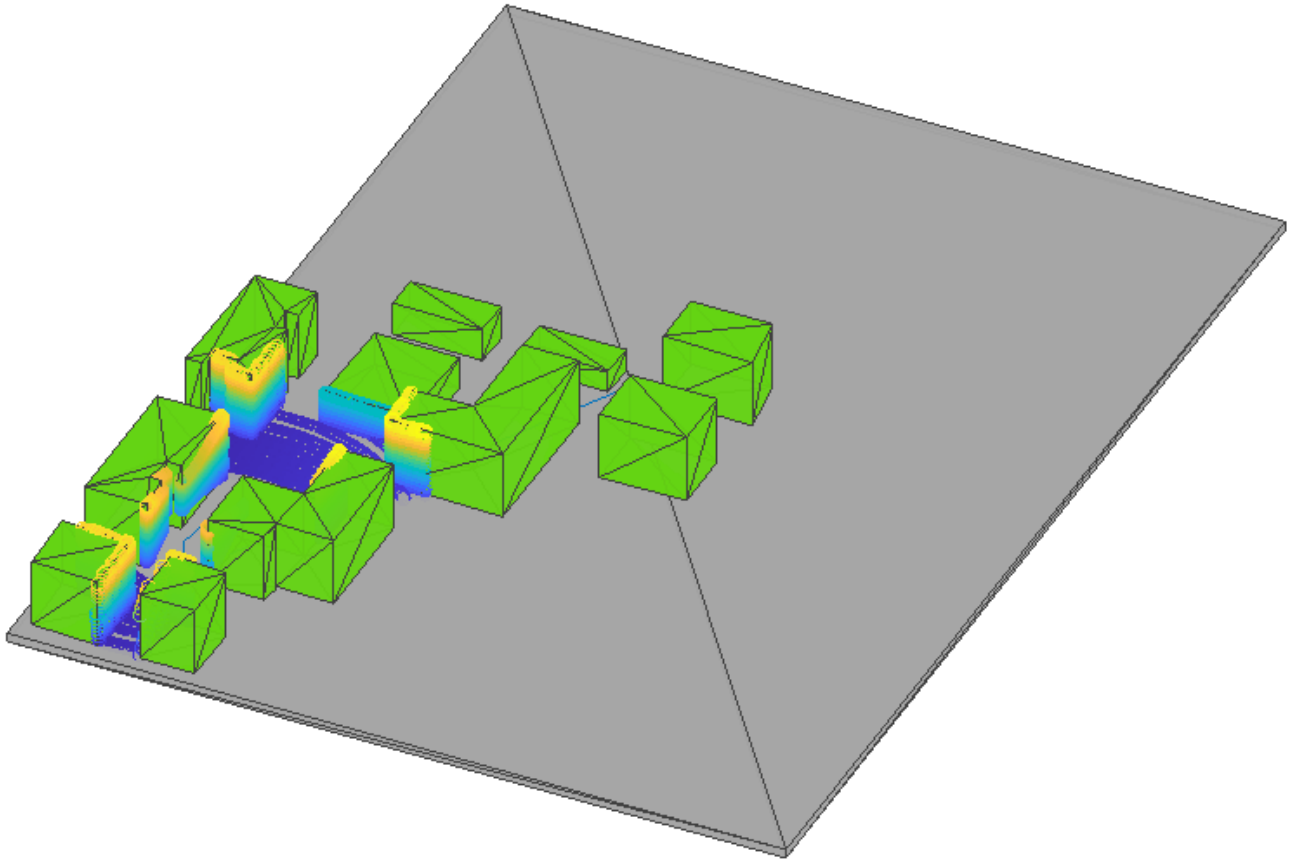
4. Obstacle Avoidance

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 buidling 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.
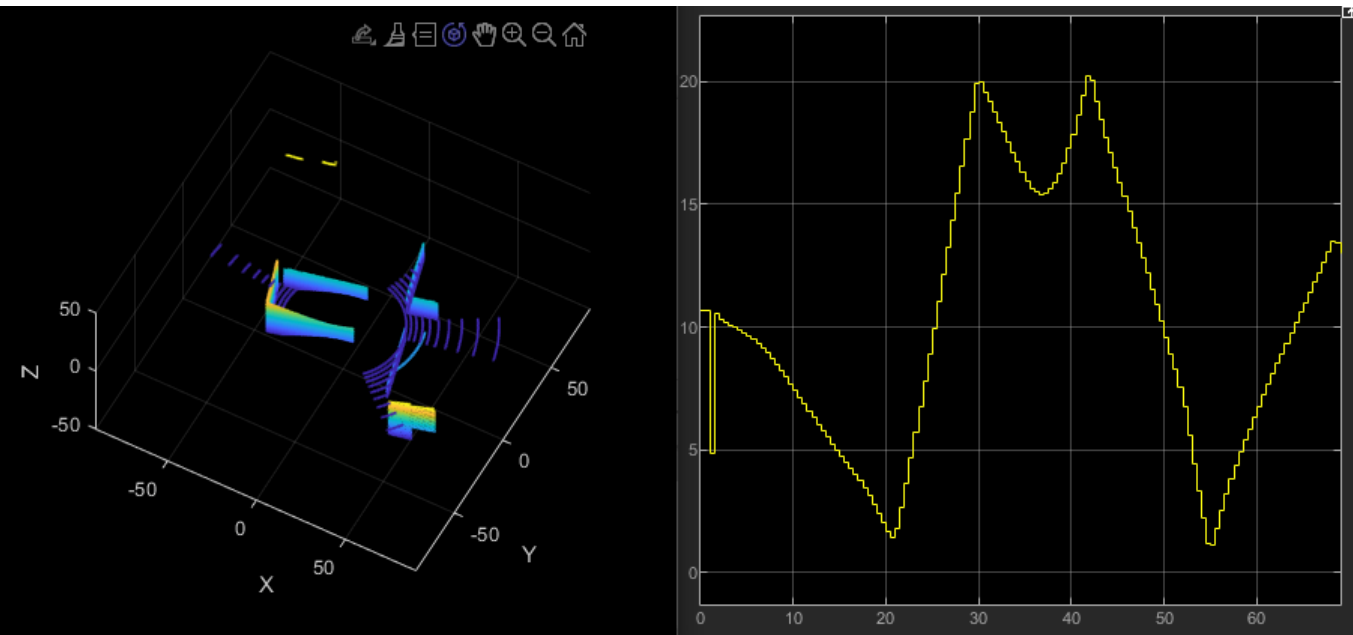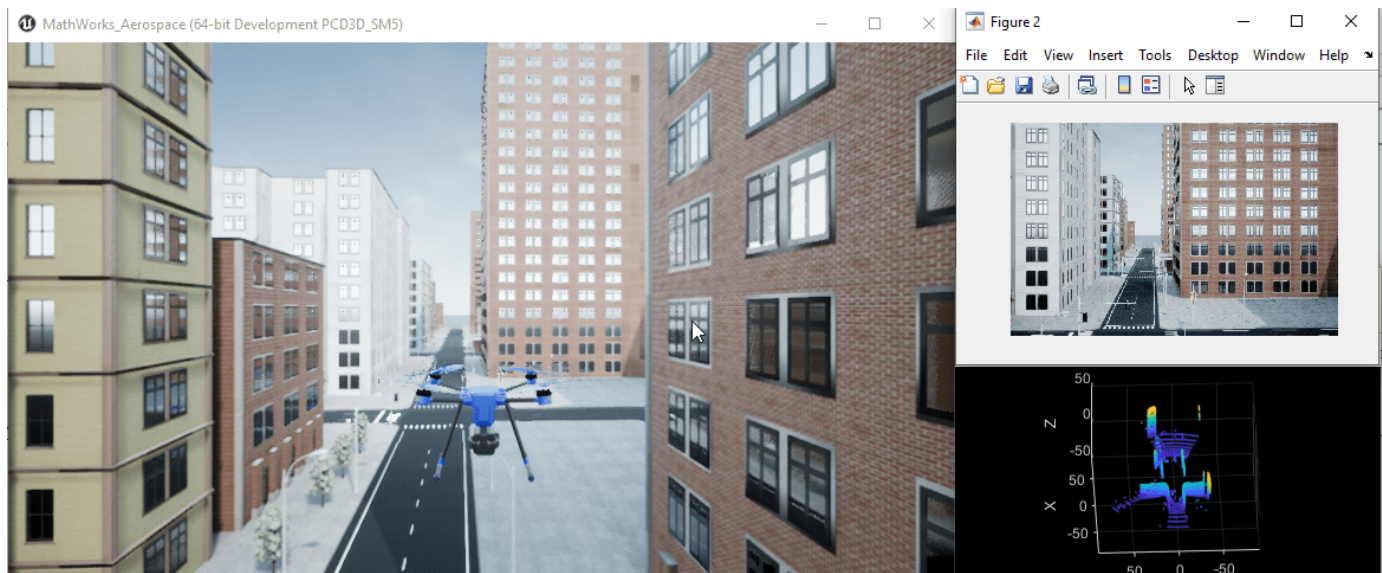
5. Photorealistic Simulation

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:
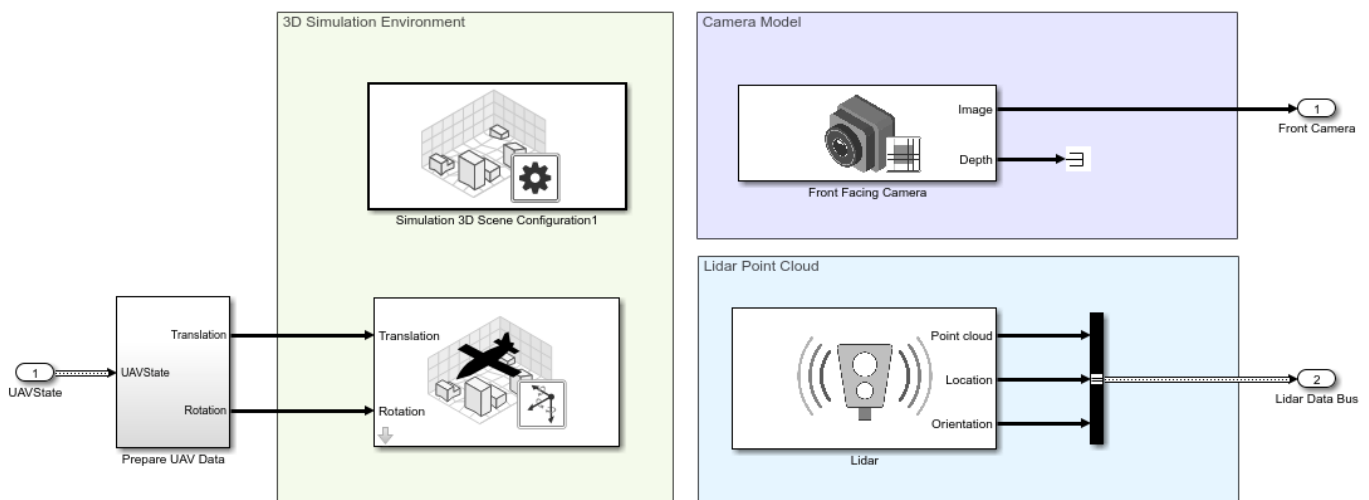
Run the model. The aircraft is set up to fly the same mission from steps 1 and 2. Notice as the aircraft flies the mission the lidar point clouds update and an image from the front-facing camera is shown.
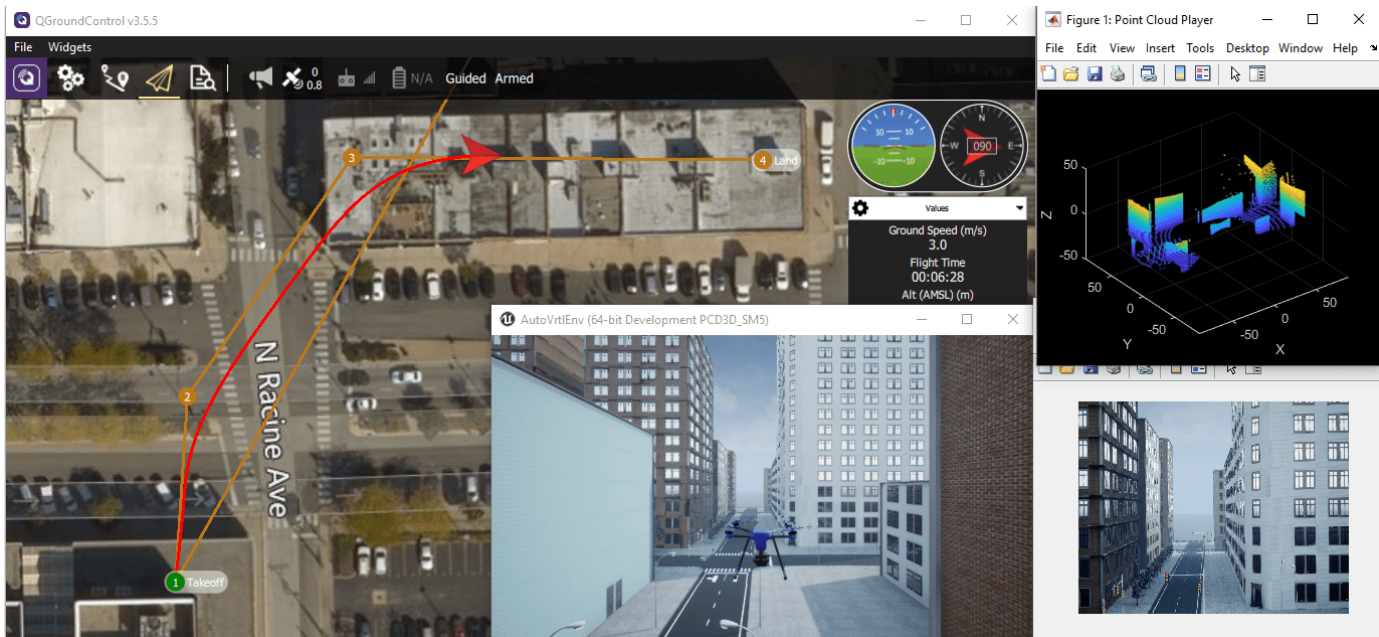
6. Fly Full Mission in a Photorealistic Simulation Environment

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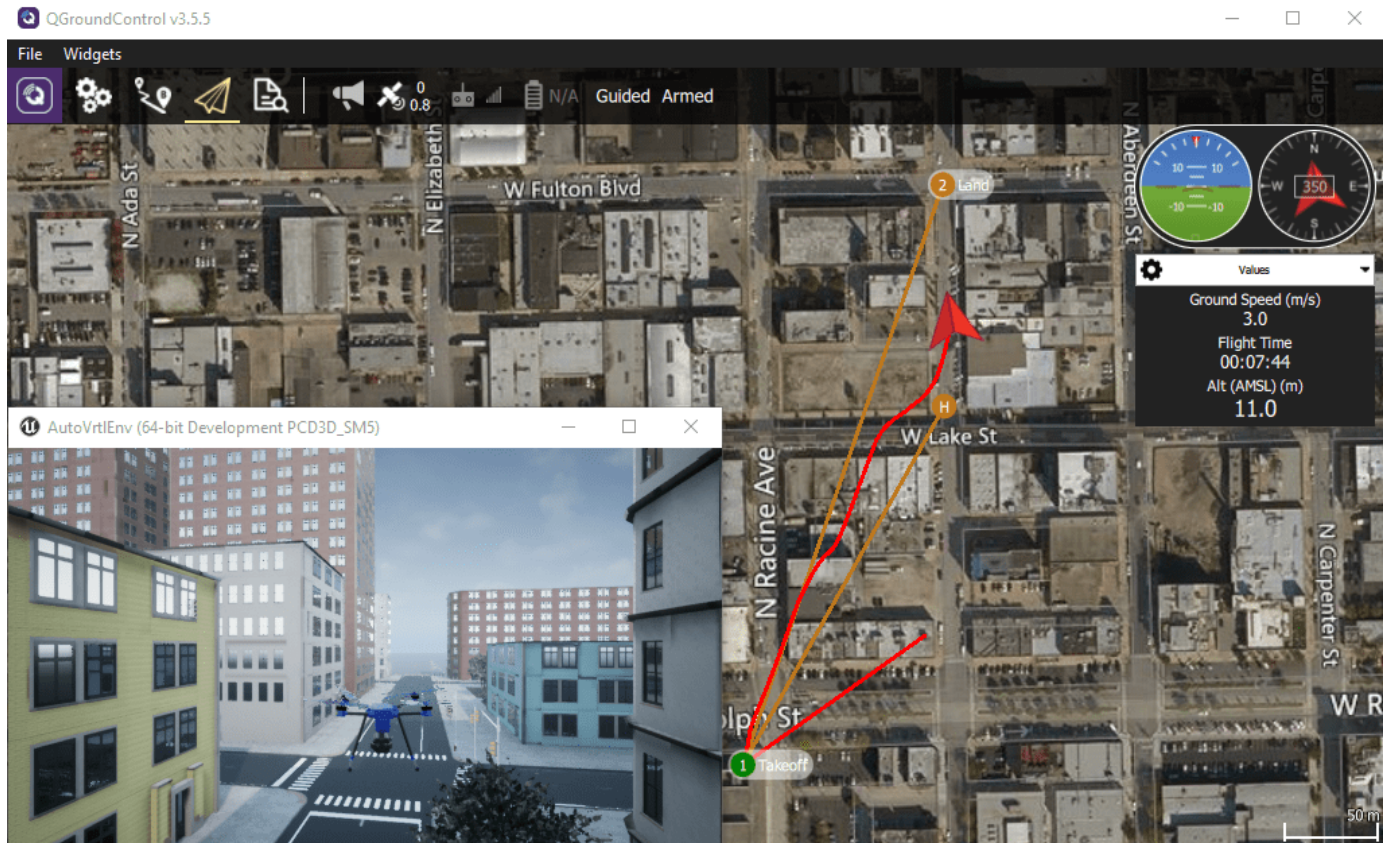
At some point during the flight, you will see the UAV pass through a narrow pass between two buildings.

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

As a final step, click the **Adding a High Fidelity Plant** 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.
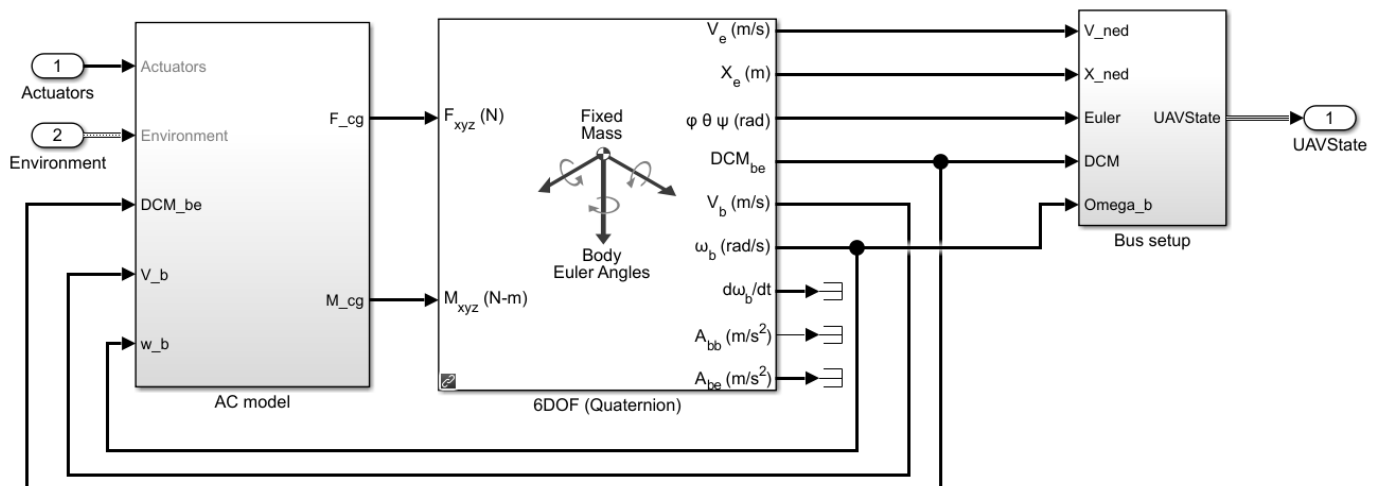
Run the model. There are minor changes in behavior due to the high-fidelity model, but the UAV flies the same mission.

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

```
close(prj);
```

Automate Testing for UAV Package Delivery Example

This example shows how to edit requirements, link requirements to their implementation in a model, and verify their functionality in the context of a UAV application. The components of the model and requirements include guidance and control of a UAV implemented by the "UAV Package Delivery" on page 1-65 example.

Introduction

The "UAV Package Delivery" on page 1-65 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. In this example we go through the process of editing a small but representative requirement set, linking these requirements to sections in the model that implment these requirements, and finally validate these through a test suite.
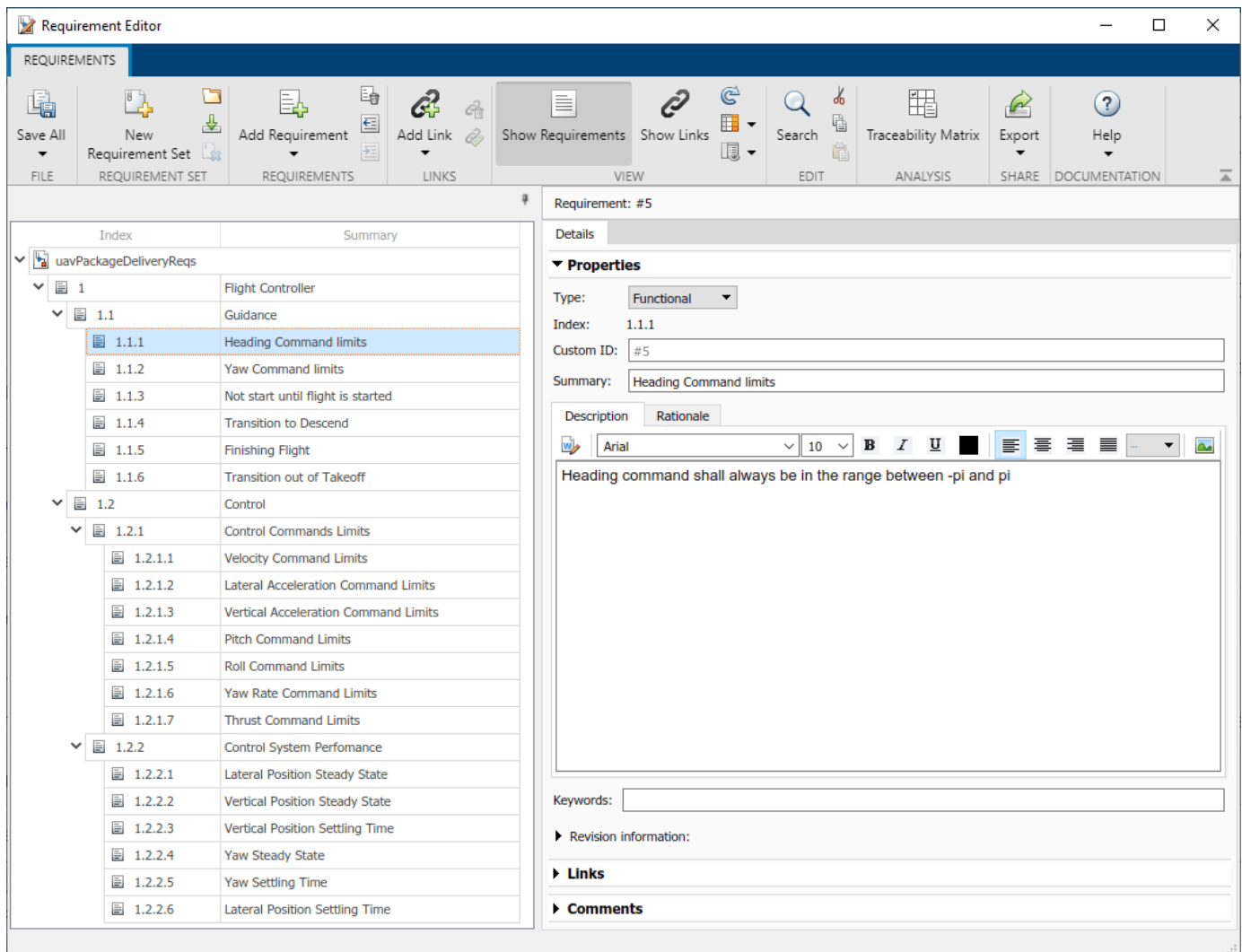
Requirements Review

To load the required project and files, click **Open Live Script** or run the `openExample` function.

```
openExample('uav/AutomateTestingForUAVPackageDeliveryExample')
```

Simulink Requirements™ lets you author, analyze, and manage requirements within Simulink™. This example contains twenty functional requirements defined for the **Guidance** and **Control** of a UAV flight controller. Open the provided Simulink project and the requirement set. Alternatively, you can also open the file from the **Requirements** tab of the **Requirements Manager** app in Simulink.

```
prj = openProject("verifications/AutomatedTestsPackageDelivery.prj");
open('uavPackageDeliveryReqs.slreqx')
```

Requirements are separated into Guidance and Control sections. These requirements map directly to sections in the multirotor model of the UAV Package Delivery example. Look through the list of requirements, and click items to see and edit details on the right.

Linking Requirements to Implementation

Simulink Requirements enables you to link each individual requirement to the Simulink model component that implements such requirement. To link requirements, first open the multirotor model.

```
open_system('MultirotorModel')
```

Enter the requirements perspective by clicking in the **Perspectives** control in the lower-right corner of the model canvas.
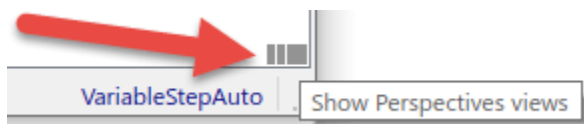

Select the **Requirements** perspective.
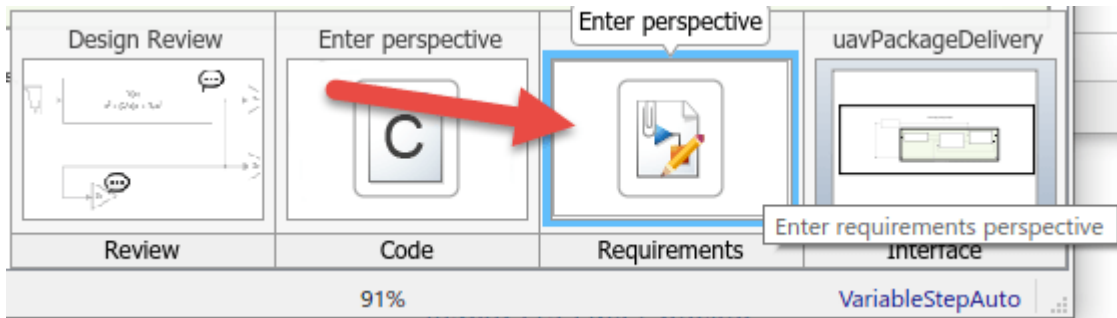
In the requirements perspective, navigate to the **Guidance Logic** and inspect some if the the guidance requirements are implemented. Requiremenst #5 and #6 are labeled in gray. The heading and yaw command limits are implemented by the **Waypoint Follower** block.

```
open_system('MultirotorModel/Guidance Logic/Full Guidance Logic/Guidance Stateflow/Guidance Mode
```
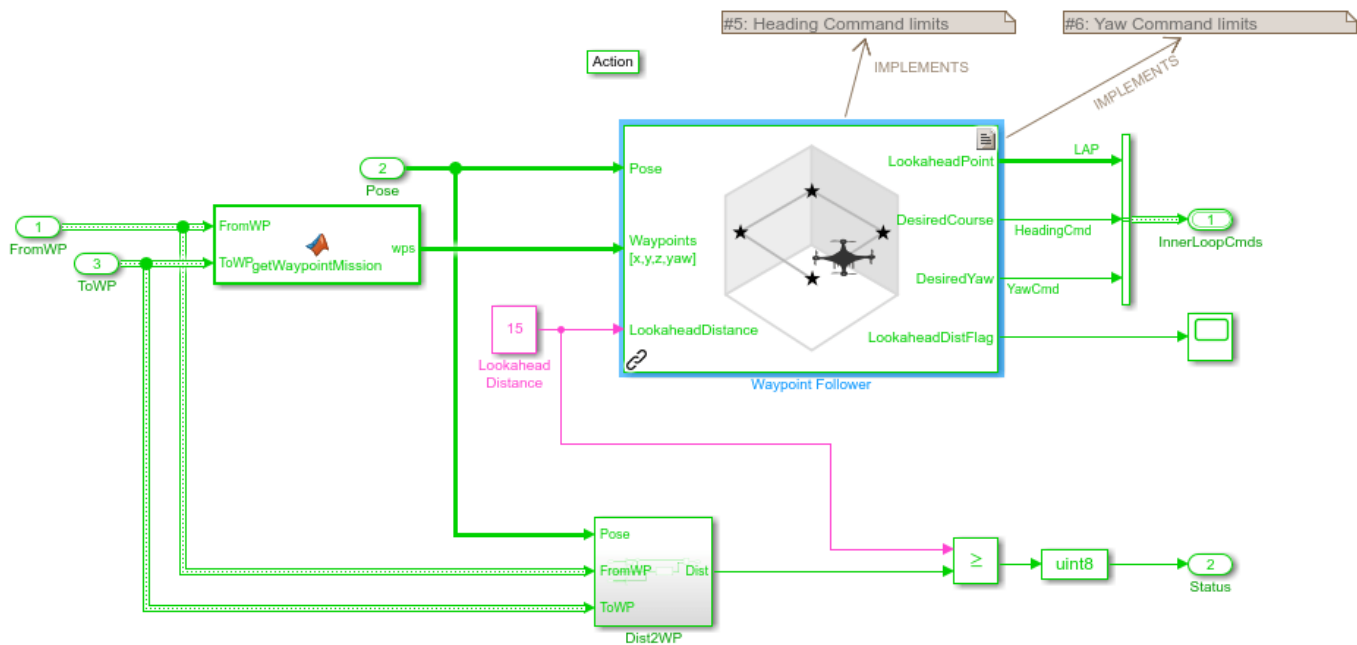

Alternatively, you can navigate to the implementation of each requirement from the **Links** section of each requirement in the requirement editor. Open the Requirement Editor. Select a functional requirement and navigate to the **Links >> Implemented by** section in the **Details** tab on the right.

```
open('uavPackageDeliveryReqs.slreqx')
```

Click on Requirement #19 (Index 1.2.1.4). In the **Details** tab under **Links**, click the **theta_limit** link to go to where the requirement is implemented in the multirotor model. The **theta Limit** block implements this requirement.
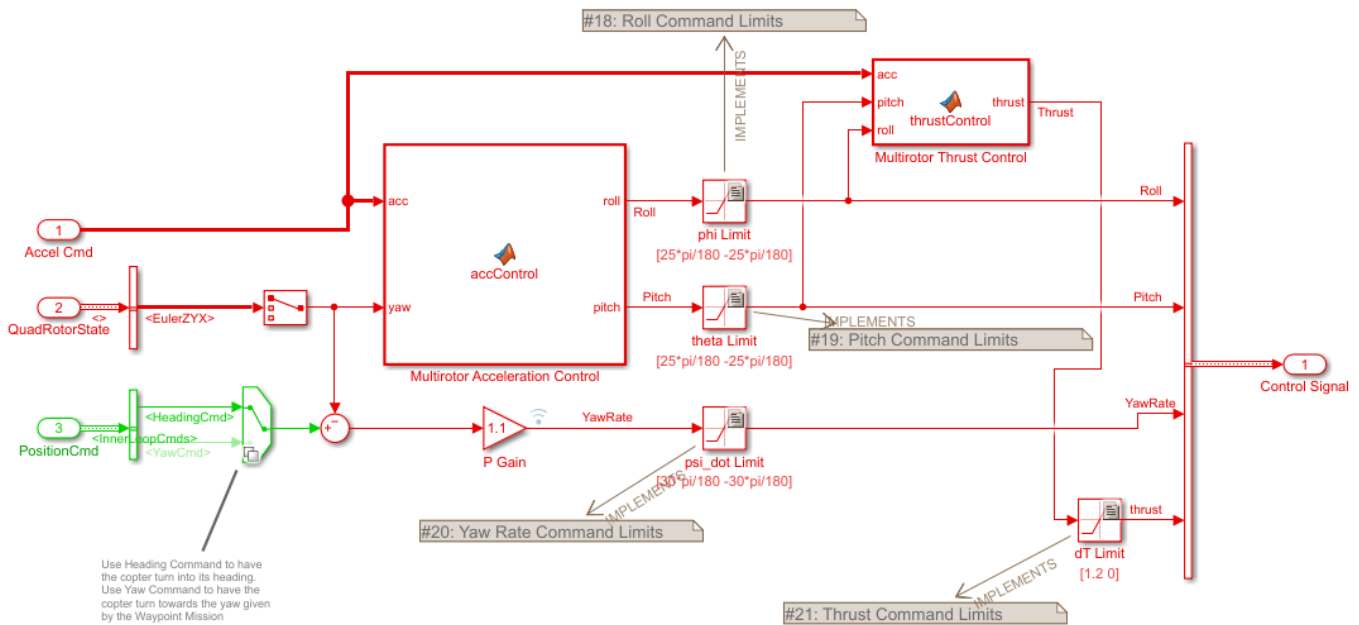
Automate Testing

To ensure each of the requirements is met, this example includes three automatic tests to run on the model. To see how these tests are implemented, open the test file in the **Test Manager**. You should see two test suites, **GuidanceLogicSuite** and **ControlSystemSuite**.

```
uiopen('MultirotorModelTest.mldatx',1)
```
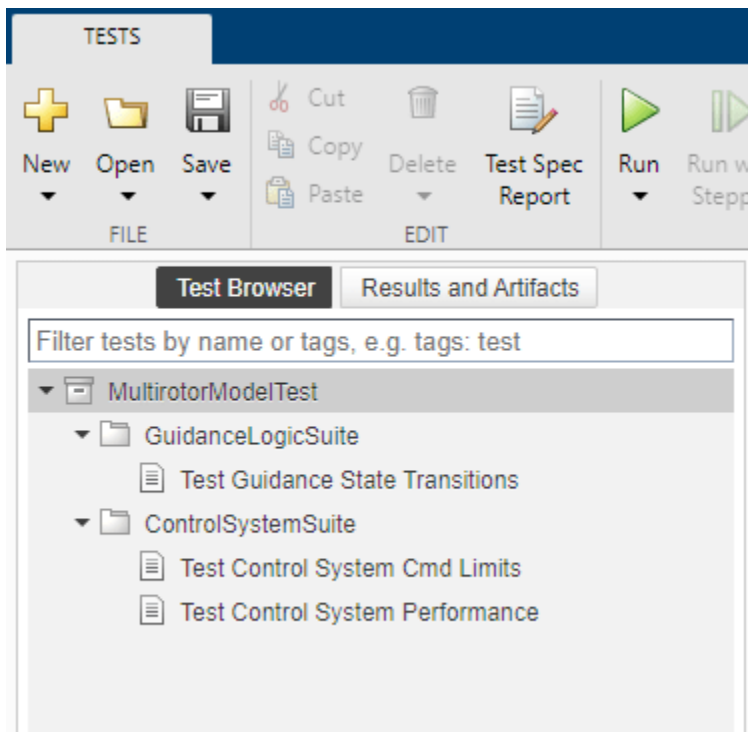
Testing the Guidance Logic

The **Test Guidance State Transitions** test makes use of a "Manage Test Harnesses" (Simulink Test) for the model. To see the test harness, click the **Test Guidance State Transitions** test and expand the **System Under Test** section of the test. Click on the arrow button to open the model:

The harness contains a Signal Editor with a pre-defined set of inputs to test all the phases of the guidance logic state machine, from takeoff to land.
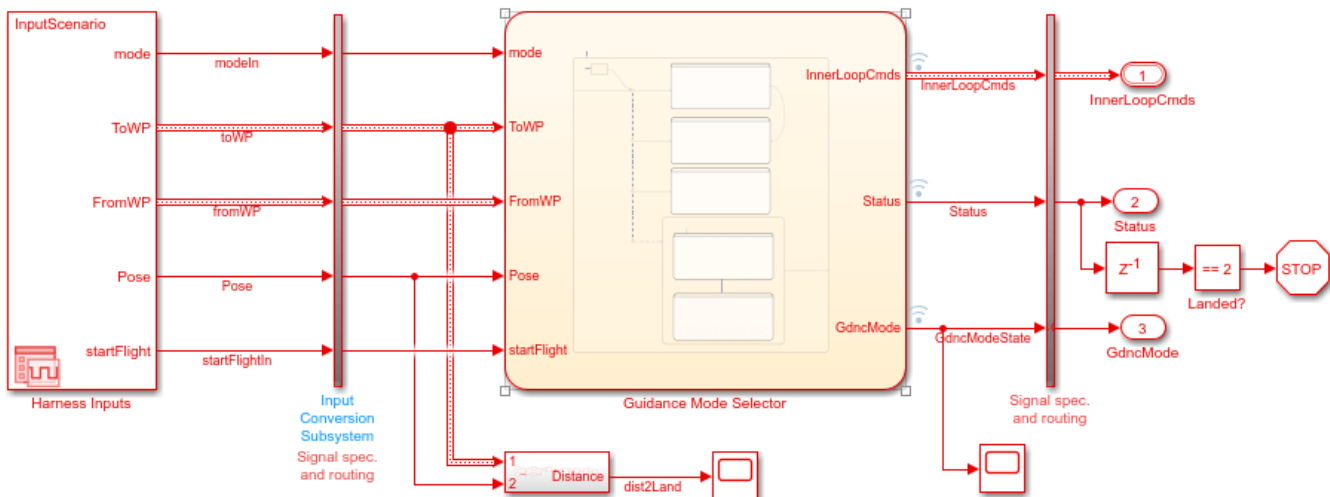
To validate the requirements are met during simuliation, the test implements six "Assess Temporal Logic by Using Temporal Assessments" (Simulink Test) and links each of these with a requirement.

Testing the Control System

The **Control System** test suite consists of two tests. One focused on testing all the command limits of the controller and the other asessing the controller performance. Both tests make use of a Simulink test harness configured to drive the control system under some reasonable inputs and evaluate the response.
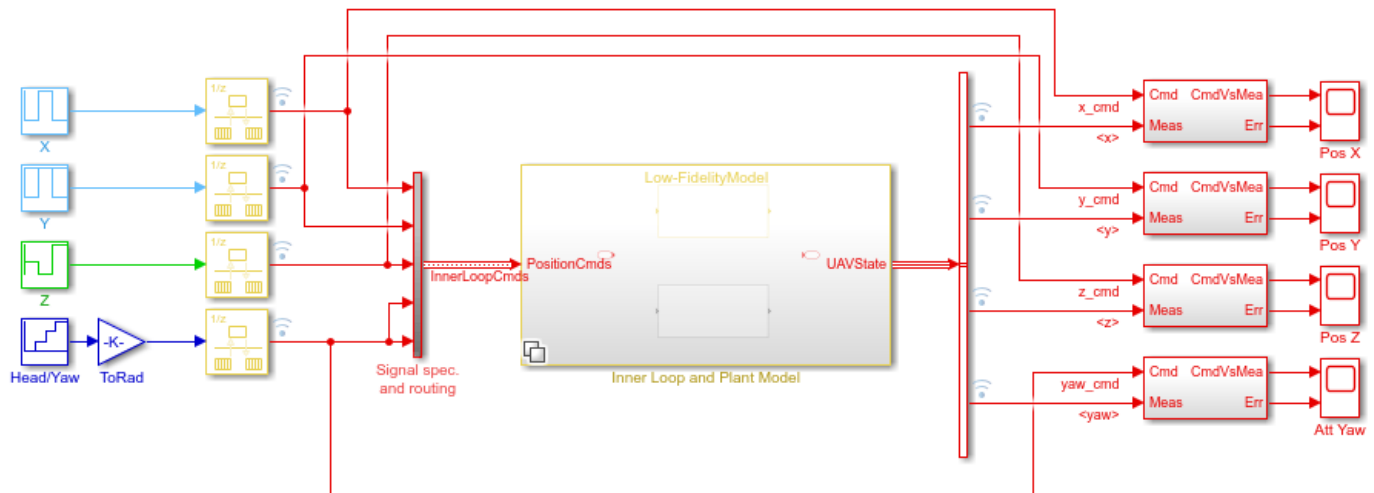
The **Test Control System Cmd Limits** test implements ten "Assess Temporal Logic by Using Temporal Assessments" (Simulink Test) assesments to make sure all commands in the control system are properly saturated to values established by the requirements. These assesments are linked to the corresponding requirements.

The **Test Control System Performance** test uses a "Compare Model Output to Baseline Data" (Simulink Test) test to asses whether the control system is within the bounds or diverges from a prerecorded baseline.

Running All Tests

To run both test suites, click **Run** on the **Test Manager** toolstrip. Once the tests run, you will see the results status in the **Results and Artifacts** tree.

Validating Requirements

As a final step, open the **Requirement Editor** and enable the **Implementation State** and **Validation Status** columns from the toolstrip. The column colors indicate whether each requirement has been implemented and verified by a test.

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.

```matlab
% 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])
```
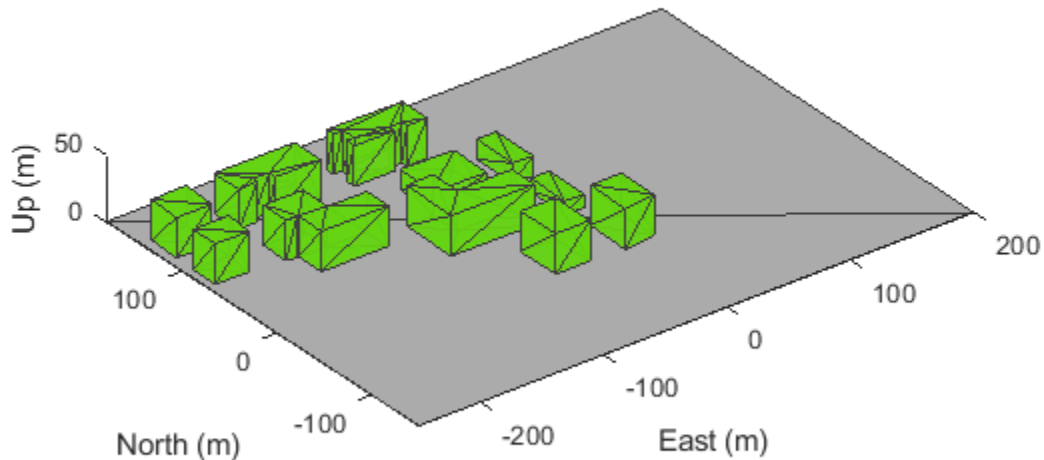
Define UAV Platform and Mount Sensor

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. First create the plot with `plot3`, then manually modify the data source properties of the plot. This improves performance of the plotting.

```
traj = plot3(nan,nan,nan,"Color",[1 1 1],"LineWidth",2);
traj.XDataSource = "position(:,2,1:idx+1)";
traj.YDataSource = "position(:,1,1:idx+1)";
traj.ZDataSource = "-position(:,3,1:idx+1)";
```

Create a scatter plot for the point cloud. Update the data source properties again.

```
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
```

Set up the simulation. Then, iterate through the positions and show the scene each time the lidar sensor updates. Advance the scene, move the UAV platform, and update the sensors.

```
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
```
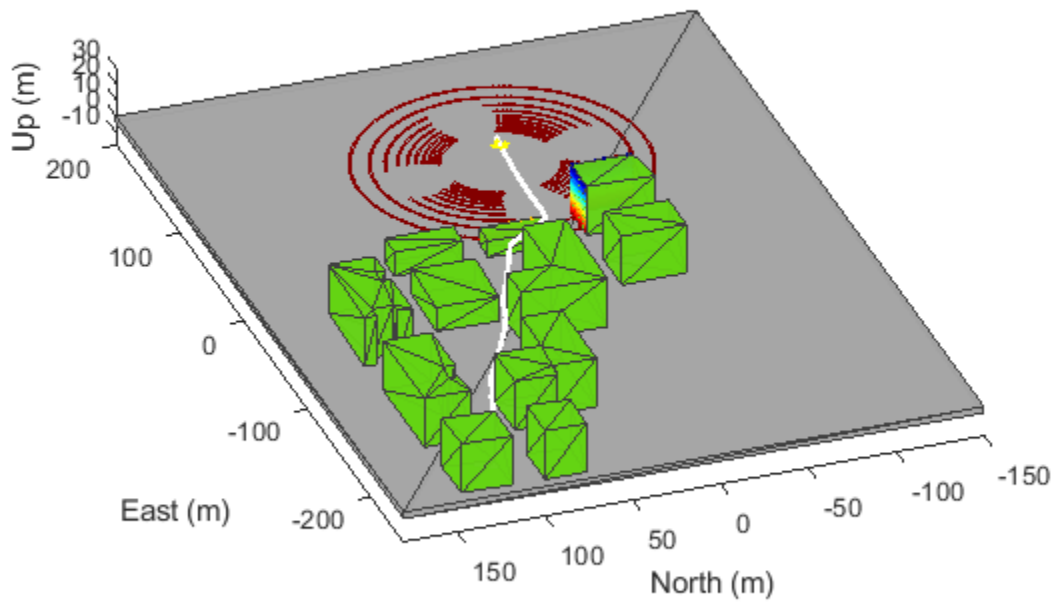
Simulate IMU Sensor Mounted on UAV

Create a sensor adaptor for an `imuSensor` from Navigation Toolbox™ and gather readings for a simulated UAV flight scenario.

Create Sensor Adaptor

Use the `createSensorAdaptorTemplate` function to generate a template sensor and update it to adapt an `imuSensor` object for usage in UAV scenario.

```
createCustomSensorTemplate
```

This example provivdes the adaptor class `uavIMU`, which can be viewed using the following command.

```
edit uavIMU.m
```

Use Sensor Adaptor in UAV Scenario Simulation

Use the IMU sensor adaptor in a UAV Scenario simulation. First, create the scenario.

```
scenario = uavScenario("StopTime", 8, "UpdateRate", 100);
```

Create a UAV platform and specify the trajectory. Add a fixed-wing mesh for visualization.

```
plat = uavPlatform("UAV", scenario, "Trajectory", ...
    waypointTrajectory([0 0 0; 100 0 0; 100 100 0], "TimeOfArrival", [0 5 8], "AutoBank", true));
updateMesh(plat,"fixedwing", {10}, [1 0 0], eul2tform([0 0 pi]));
```

Attach the IMU sensor using the `uavSensor` object and specify the `uavIMU` as an input. Load parameters for the sensor model.

```
imu = uavSensor("IMU", plat, uavIMU(imuSensor));

fn = fullfile(matlabroot,'toolbox','shared',...
    'positioning','positioningdata','generic.json');
loadparams(imu.SensorModel,fn,"GenericLowCost9Axis");
```

Visualize the scenario.

```
figure
ax = show3D(scenario);
xlim([-20 200]);
ylim([-20 200]);
```

Preallocate the `simData` structure and fields to store simulation data. The IMU sensor will output acceleration and angular rates.

```
simData = struct;
simData.Time = duration.empty;
simData.AccelerationX = zeros(0,1);
simData.AccelerationY = zeros(0,1);
simData.AccelerationZ = zeros(0,1);
simData.AngularRatesX = zeros(0,1);
simData.AngularRatesY = zeros(0,1);
simData.AngularRatesZ = zeros(0,1);
```

Setup the scenario.

```
setup(scenario);
```

Run the simulation using the `advance` function. Update the sensors and record the data.

```matlab
updateCounter = 0;
while true
    % Advance scenario.
    isRunning = advance(scenario);
    updateCounter = updateCounter + 1;
    % Update sensors and read IMU data.
    updateSensors(scenario);
    [isUpdated, t, acc, gyro] = read(imu);
    % Store data in structure.
    simData.Time = [simData.Time; seconds(t)];
    simData.AccelerationX = [simData.AccelerationX; acc(1)];
    simData.AccelerationY = [simData.AccelerationY; acc(2)];
    simData.AccelerationZ = [simData.AccelerationZ; acc(3)];
    simData.AngularRatesX = [simData.AngularRatesX; gyro(1)];
    simData.AngularRatesY = [simData.AngularRatesY; gyro(2)];
    simData.AngularRatesZ = [simData.AngularRatesZ; gyro(3)];

    % Update visualization every 10 updates.
    if updateCounter > 10
        show3D(scenario, "FastUpdate", true, "Parent", ax);
        updateCounter = 0;
        drawnow limitrate
    end
    % Exit loop when scenario is finished.
    if ~isRunning
        break;
    end
end
```
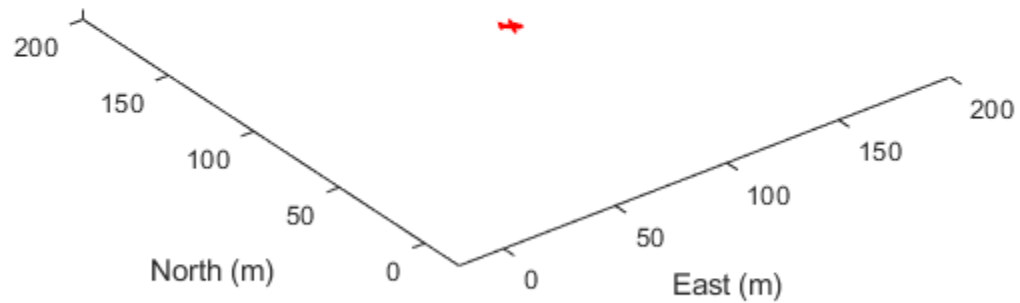
Visualize the simulated IMU readings.

```
simTable = table2timetable(struct2table(simData));
figure
stackedplot(simTable, ["AccelerationX", "AccelerationY", "AccelerationZ", ...
    "AngularRatesX", "AngularRatesY", "AngularRatesZ"], ...
    "DisplayLabels", ["AccX (m/s^2)", "AccY (m/s^2)", "AccZ (m/s^2)", ...
    "AngularRateX (rad/s)", "AngularRateY (rad/s)", "AngularRateZ (rad/s)"]);
```
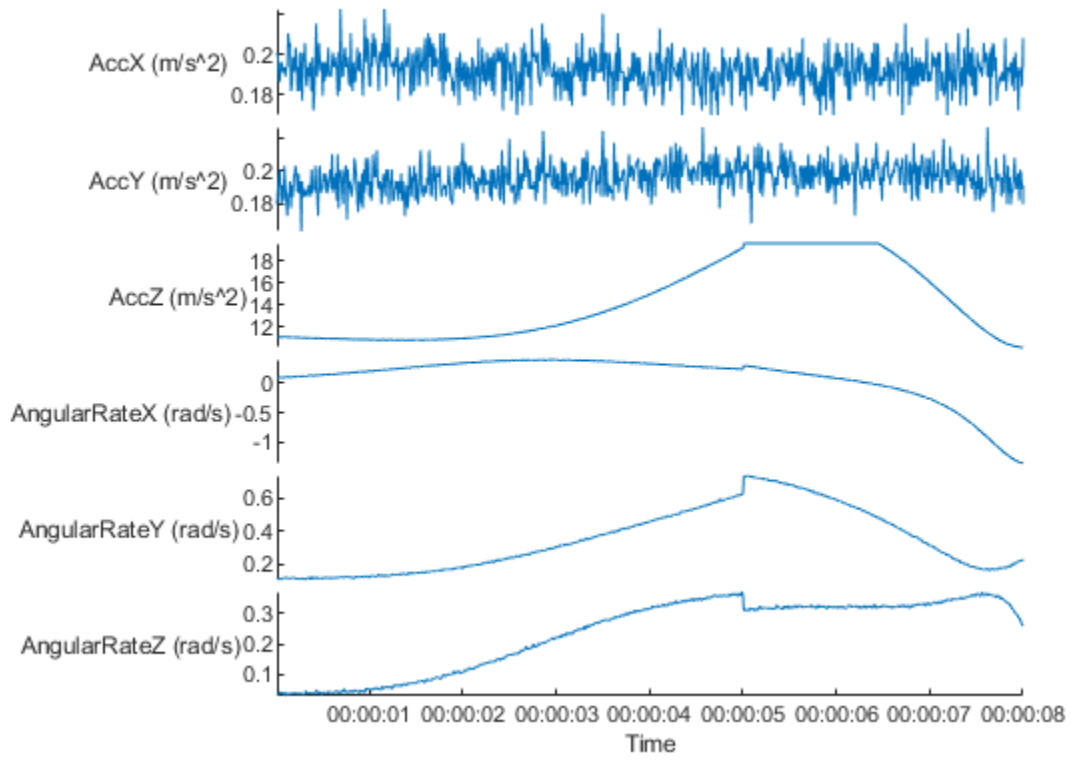
Simulate Radar Sensor Mounted On UAV

The radar sensor enables a UAV to detect other vehicles in the airspace, so that the UAV can predict other vehicle motion and make decisions to ensure clearance from other vehicles. This example shows how to simulate a radar sensor mounted on a UAV using the `uavScenario` and `radarDataGenerator` objects. During the scenario simulation, the `radarDataGenerator` object generates flight tracks of another vehicle in the scenario. The ego vehicle can utilize such track information to decide whether a collision is about to happen and decide whether a flight plan change is required.

Creating UAV Scenario with Custom Radar Sensor

The testing scenario consists of two UAVs. The fixed-wing UAV is the target vehicle and the multirotor UAV is tracking the fixed-wing UAV using a mounted radar sensor.

```
% Use fixed random seed for simulation repeatablity.
rng(0)

% Create a scenario that runs for 10 seconds.
s = uavScenario("StopTime",10,"HistoryBufferSize",200);

% Create a fixed-wing target that moves from [30 0 0] to [20 10 0].
target = uavPlatform("Target",s,"Trajectory",waypointTrajectory([30 0 0; 20 10 0],"TimeOfArrival"
updateMesh(target,"fixedwing", {1}, [1 0 0], eul2tform([0 0 pi]));

% Create a quadrotor that moves from [0 0 0] to [10 10 0].
egoMultirotor = uavPlatform("EgoVehicle",s,"Trajectory",waypointTrajectory([0 0 0; 10 10 0],"Time
updateMesh(egoMultirotor,"quadrotor",{1},[0 1 0],eul2tform([0 0 pi]));

% Mount a radar on the quadrotor.
radarSensor = radarDataGenerator("no scanning","SensorIndex",1,"UpdateRate",10,...
    "FieldOfView",[120 80],...
    "HasElevation", true,...
    "ElevationResolution", 3,...
    "AzimuthResolution", 1, ...
    "RangeResolution", 10, ... meters
    "RangeRateResolution",3,...
    "RangeLimits", [0 750],...
    "TargetReportFormat","Tracks",...
    "TrackCoordinates",'Scenario',...
    "HasINS", true,...
    "HasFalseAlarms",true,...
    "FalseAlarmRate",1e-5,...
    "HasRangeRate",true,...
    "FalseAlarmRate", 1e-7);

% Create the sensor. ExampleHelperUAVRadar inherits from the uav.SensorAdaptor class.
radar = uavSensor("Radar",egoMultirotor,ExampleHelperUAVRadar(radarSensor),"MountingAngles", [0 0
```

Preview the scenario using the `show3D` function.

```
[ax,plotFrames] = show3D(s);
xlim([-5,15]);
ylim([-5,35]);
hold on
```
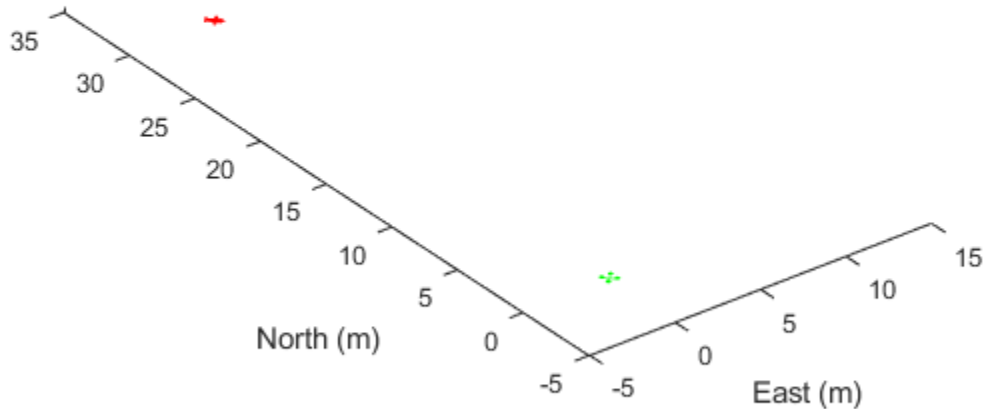
Simulate and Visualize Radar Detections

Setup the scenario, run the simulation, and check the detections.

```
% Add detection and sensor field of view to the plot.
trackSquare = plot3(plotFrames.NED,nan,nan,nan,"-");
radarDirection = hgtransform("Parent",plotFrames.EgoVehicle.Radar,"Matrix",eye(4));
coverageAngles = linspace(-radarSensor.FieldOfView(1)/360*pi, radarSensor.FieldOfView(1)/360*pi,1
coveragePatch = patch([0 radarSensor.RangeLimits(2)*cos(coverageAngles) 0], ...
    [0 radarSensor.RangeLimits(2)*sin(coverageAngles) 0],...
    "blue","FaceAlpha",0.3,...
    "Parent",radarDirection);
hold(ax,"off");

% Start simulation.
setup(s);
while advance(s)
    % Update sensor readings and read data.
    updateSensors(s);

    % Plot updated radar FOV.
    egoPose = read(egoMultirotor);
    radarFOV = coverageConfig(radarSensor, egoPose(1:3),quaternion(egoPose(10:13)));
    radarDirection.Matrix = eul2tform([radarFOV.LookAngle(1)/180*pi 0 0]);

    % Obtain detections from the radar and visualize them.
    [isUpdated,time,confTracks,numTracks,config] = read(radar);
    if numTracks > 0
```

```
        trackSquare.XData = [trackSquare.XData,confTracks(1).State(1)];
        trackSquare.YData = [trackSquare.YData,confTracks(1).State(3)];
        trackSquare.ZData = [trackSquare.ZData,confTracks(1).State(5)];
        drawnow limitrate
    end

    show3D(s,"FastUpdate", true,"Parent",ax);
    pause(0.1);
end
```
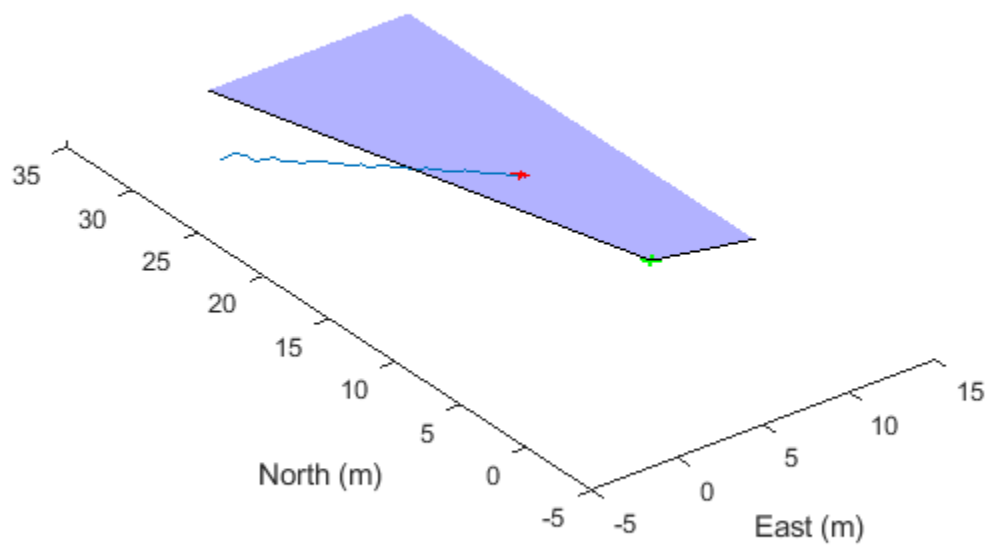

The target UAV track is visualized during simulation. Using this track prediction, the ego vehicle can now make decisions about whether a collision is about to happen. This enables you to implement obstacle avoidance algorithms and test them with this scenario.

Map Environment For Motion Planning Using UAV Lidar

This example demonstrates how to use a UAV, equipped with a lidar sensor, to map an environment in a 3D occupancy map. That generated occupancy map can then be used to execute motion planning. The UAV follows a specified trajectory through the environment. Knowledge of the UAV's position and the point clouds obtained at corresponding locations is used to build a map of the environment. The pose of the UAV is assumed to be completely known throughout the mapping flight; this is also known as mapping with known poses. The generated map is used for motion planning. The goal of the motion planning is to plan the path of a UAV in an apartment complex where it is used to drop off packages, delivered at the gate, to a desired rooftop location within the complex.

The East-North-Up (ENU) coordinate system is used throughout this example.

Table of Contents

Create Scenario

Create a simple `uavScenario` to build a representation of a simple apartment complex scene. During the scenario simulation, the scene will be used to create simulate lidar data points.

Specify the time and update rate of the simulation. In this case, the simulated flight takes 60 seconds and the simulation is updated at 2 Hz.

```
close all
close all hidden

simTime = 60;    % in seconds
updateRate = 2;  % in Hz
scene = uavScenario("UpdateRate",updateRate,"StopTime",simTime);
```

Add static meshes to the scenario to represent different buildings in the environment. We define each building with a rectangular footprint and varying height. All the coordinates are in meters.

```
% Floor
addMesh(scene, "Polygon",{[0 0;80 0;80 80;40 80;40 40;0 40],[-1 0]},[0.3 0.3 0.3]);

% Features
addMesh(scene, "Polygon",{[10 0;30 0;30 20;10 20],[0 40]},[0.4660 0.6740 0.1880]);    % Building
addMesh(scene, "Polygon",{[45 0;80 0;80 30;45 30],[0 60]},[0.9290 0.6980 0.1250]);    % Building
addMesh(scene, "Polygon",{[0 35;10 35;10 40;0 40],[0 5]},[0 0.5 0]);                   % Generato
addMesh(scene, "Polygon",{[50 40;80 40;80 70;50 70],[0 5]},[0 0.4470 0.7410]);         % Swimming
addMesh(scene, "Polygon",{[0 0;2 0;2 4;0 4],[0 3]},[0.6350 0.0780 0.1840]);            % Security
```

View the created scenario in 3D.

```
show3D(scene);
axis equal
view([-115 20])
```
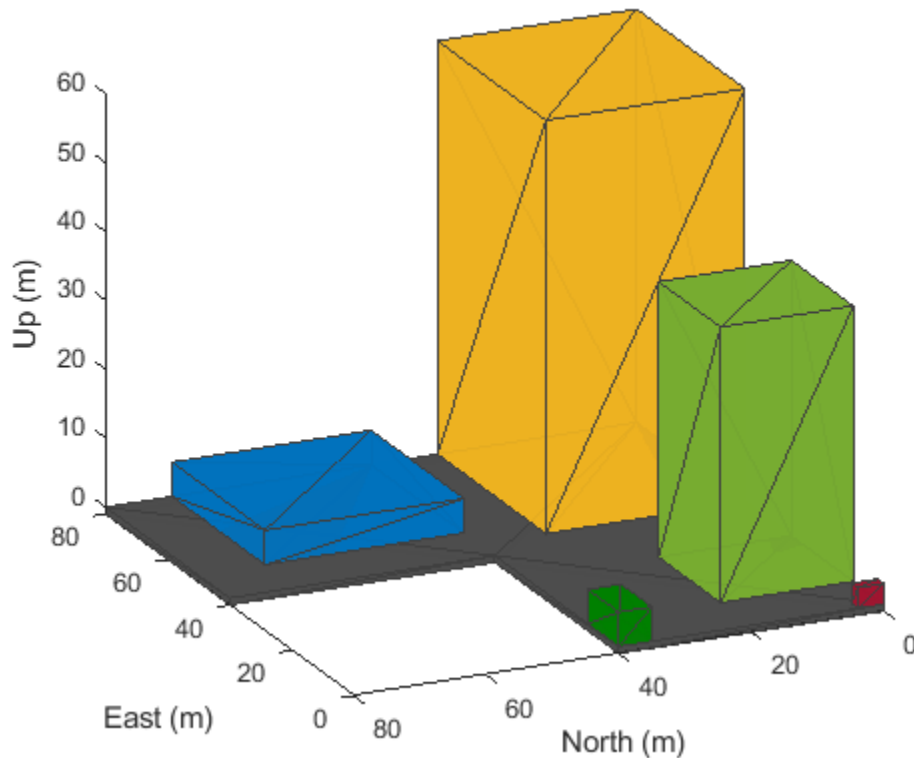
Create Mapping Trajectory

Specify the trajectory that the UAV follows for mapping the environment. Specify waypoints and assign orientation for each waypoint.

The complete pose vector for UAV is as follows

pose = [x y z a b c d]

x, y and z specify the position of the UAV with respect to the scenario's frame of reference. a, b, c and d are the parts of the quaternion number that specifies the orientation of the UAV with respect to the scenario's reference frame. a is the real part of the quaternion.

```
% Waypoints
x = -20:80;
y = -20:80;
z = 100*ones(1,length(x));

waypoints = [x' y' z'];
```

Specify the orientation as Euler angles (in radians) and convert the Euler angles to a quaternion:

```
orientation_eul = [0 0 0];
orientation_quat = quaternion(eul2quat(orientation_eul));
orientation_vec = repmat(orientation_quat,length(x),1);
```

Specify time vector

```
 time = 0:(simTime/(length(x)-1)):simTime;
```

Generate trajectory from the specified waypoints and orientations using `waypointTrajectory` system object. Specify the reference frame as 'ENU'.

```
trajectory = waypointTrajectory("Waypoints",waypoints,"Orientation",orientation_vec, ...
    "SampleRate",updateRate,"ReferenceFrame","ENU","TimeOfArrival",time);
```

Specify the initial pose of the UAV

```
initial_pose = [-20 -20 100 1 0 0 0];
```

Create UAV Platform

Create a `uavPlatform` object and update the scenario with a mesh of the UAV.

```
plat = uavPlatform("UAV",scene,"Trajectory",trajectory,"ReferenceFrame","ENU");
updateMesh(plat,"quadrotor",{4},[1 0 0],eye(4));
```

Introduce a 3D lidar into the scenario using the `uavLidarPointCloudGenerator` system object. Specify relevant lidar sensor parameters. For example, the lidar in this example has a maximum range of 200 meters, but you can adjust the parameters as needed.

```
lidarmodel = uavLidarPointCloudGenerator("AzimuthResolution",0.6, ...
    "ElevationLimits",[-90 -20],"ElevationResolution",2.5, ...
    "MaxRange",200,"UpdateRate",2,"HasOrganizedOutput",true);
```

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

Simulate Mapping Flight And Map Building

```
[ax,plotFrames] = show3D(scene);
xlim([-15 80]);
ylim([-15 80]);
zlim([0 80]);
view([-115 20]);
axis equal
hold on

colormap('jet');
ptc = pointCloud(nan(1,1,3));
scatterplot = scatter3(nan,nan,nan,1,[0.3020 0.7451 0.9333],...
    "Parent",plotFrames.UAV.Lidar);
scatterplot.XDataSource = "reshape(ptc.Location(:,:,1), [], 1)";
scatterplot.YDataSource = "reshape(ptc.Location(:,:,2), [], 1)";
scatterplot.ZDataSource = "reshape(ptc.Location(:,:,3), [], 1)";
scatterplot.CDataSource = "reshape(ptc.Location(:,:,3), [], 1) - min(reshape(ptc.Location(:,:,3)
hold off;

lidarSampleTime = [];
pt = cell(1,((updateRate*simTime) +1));
ptOut = cell(1,((updateRate*simTime) +1));
```

Create an occupancy map for a more efficient way to store the point cloud data. Use a minimum resolution of 1 cell per meter.

```
map3D = occupancyMap3D(1);
```

Simulate the mapping flight in the scenario. Store lidar sensor readings for each simulation step in a cell array after removing invalid points. Insert point clouds into the map using the function. Ensure that the pose vector accounts for sensor offset.

```matlab
setup(scene);

ptIdx = 0;
while scene.IsRunning
    ptIdx = ptIdx + 1;
    % Read the simulated lidar data from the scenario
    [isUpdated,lidarSampleTime,pt{ptIdx}] = read(lidar);

    if isUpdated
        % Get Lidar sensor's pose relative to ENU reference frame.
        sensorPose = getTransform(scene.TransformTree, "ENU","UAV/Lidar",lidarSampleTime);
        % Process the simulated Lidar pointcloud.
        ptc = pt{ptIdx};
        ptOut{ptIdx} = removeInvalidPoints(pt{ptIdx});
        % Construct the occupancy map using Lidar readings.
        insertPointCloud(map3D,[sensorPose(1:3,4)' tform2quat(sensorPose)],ptOut{ptIdx},500);

        figure(1)
        show3D(scene,"Time",lidarSampleTime,"FastUpdate",true,"Parent",ax);
        xlim([-15 80]);
        ylim([-15 80]);
        zlim([0 110]);
        view([-110 20]);

        refreshdata
        drawnow limitrate
    end

    % Show map building real time
    figure(2)
    show(map3D);
    view([-115 20]);
    axis equal

    advance(scene);
    updateSensors(scene);

end
```
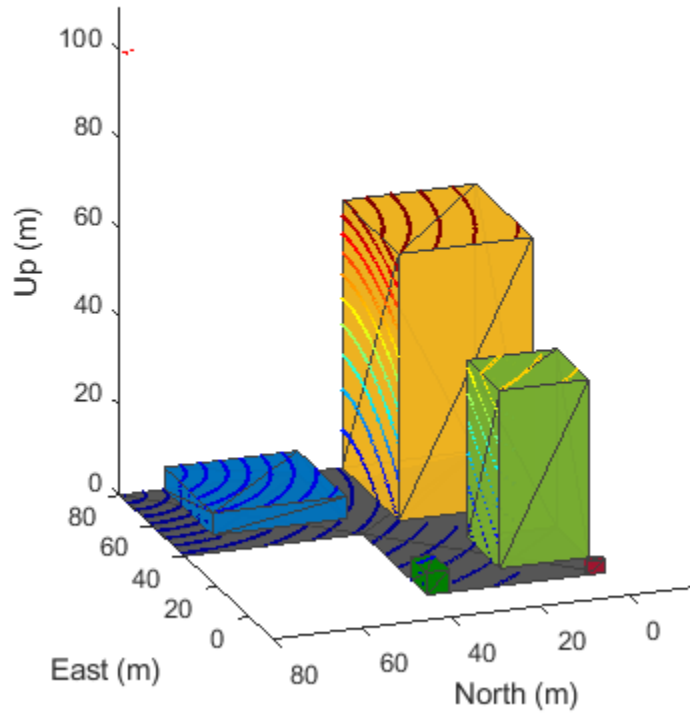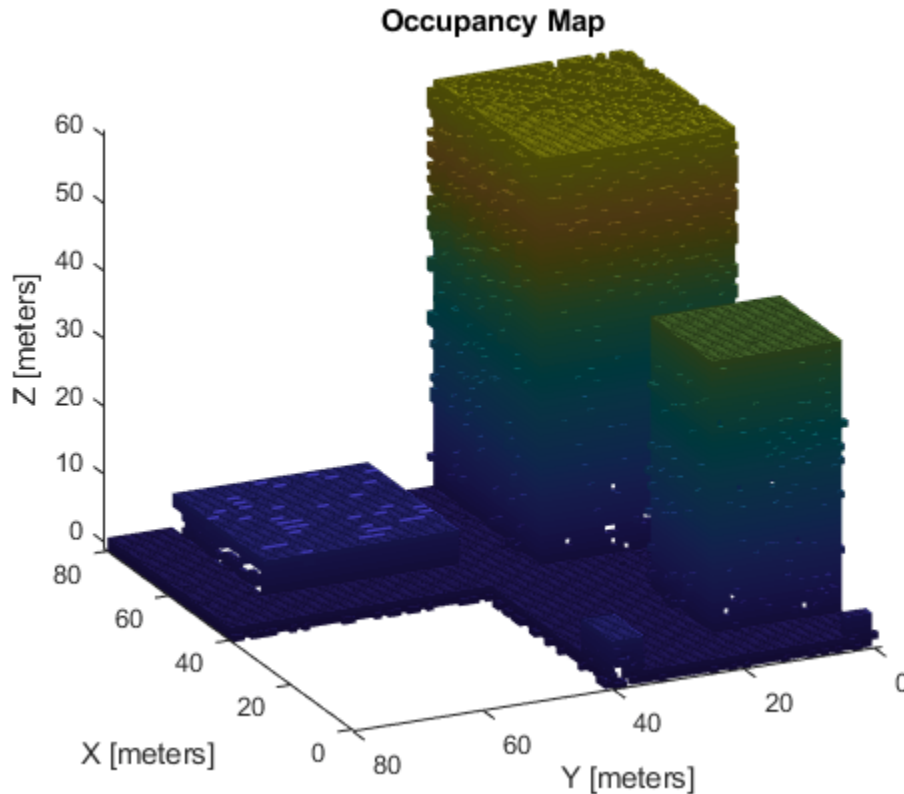
Occupancy Map

Perform Motion Planning

assigns probability observational value of 0.4 for unoccupied cells and 0.7 for occupied cells. Specify the same as the thresholds for the occupancy map.

```
map3D.FreeThreshold = 0.4;
map3D.OccupiedThreshold = 0.7;
```

Use as the state space object for validator as the state vector of the state space is same as the pose vector.

```
ss = stateSpaceSE3([0 80;0 40;0 120;inf inf;inf inf;inf inf;inf inf]);
sv = validatorOccupancyMap3D(ss);
sv.Map = map3D;
sv.ValidationDistance = 0.1;
```

Use an optimal planner, RRT* is for planning. Planning continues even after goal is reached and terminates only when iteration limit is reached.

```
planner = plannerRRTStar(ss,sv);
planner.MaxConnectionDistance = 20;
planner.ContinueAfterGoalReached = true;
planner.MaxIterations = 500;
```

Specify a custom goal function that determines that a path reaches the goal if the Euclidean distance to the target is below a threshold of 1 meter.

```
planner.GoalReachedFcn = @(~,x,y)(norm(x(1:3)-y(1:3))<1);
planner.GoalBias = 0.1;
```

Specify start and goal poses and plan the path.

```
start = [3 5 5 1 0 0 0];
goal = [60 10 65 1 0 0 0];

rng(1,"twister"); % For repeatable results
[pthObj,solnInfo] = plan(planner,start,goal);
```

View Planned Path

```
close all
close all hidden

show(map3D)
axis equal
view([-115 20])
hold on
scatter3(start(1,1),start(1,2),start(1,3),'g','filled')                              % draw start
scatter3(goal(1,1),goal(1,2),goal(1,3),'r','filled')                                 % draw goal
plot3(pthObj.States(:,1),pthObj.States(:,2),pthObj.States(:,3),'r-','LineWidth',2)  % draw path
```
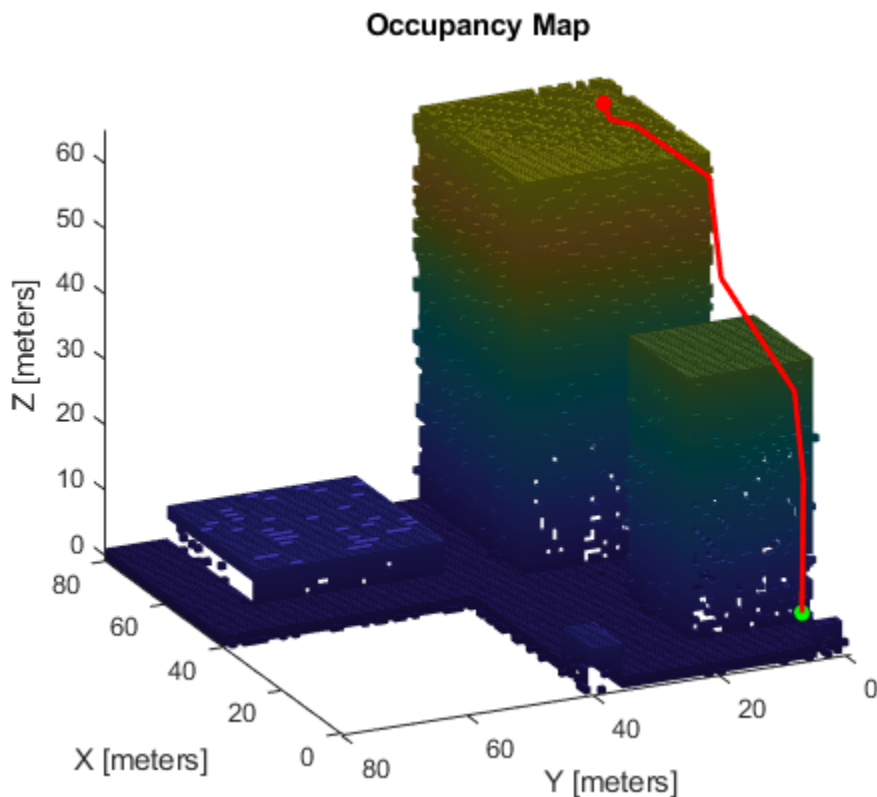
Plan Minimum Snap Trajectory for Quadrotor

This example shows how to plan a minimum snap trajectory (minimum control effort) for a multirotor unmanned aerial vehicle (UAV) from a start pose to a goal pose on a 3D map by using the optimized rapidly exploring random tree (RRT*) path planner.

In this example, you set up a 3D map, provide a start pose and goal pose, plan a path with RRT* using 3D straight line connections, and fit a minimum snap trajectory through the obtained waypoints.

Initial Setup

Configure the random number generator for repeatable result.

```
rng(100,"twister")
```

Load Map

Load the 3D occupancy map `uavMapCityBlock.mat`, which contains a set of pregenerated obstacles, into the workspace. The occupancy map is in an east-north-up (ENU) frame.

```
mapData = load("uavMapCityBlock.mat");
omap = mapData.omap;

% Consider unknown spaces to be unoccupied
omap.FreeThreshold = omap.OccupiedThreshold;
show(omap)
```

Set Start Pose and Goal Pose

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

```
startPose = [12 22 25 0.7 0.2 0 0.1];  % [x y z qw qx qy qz]
goalPose = [150 180 35 0.3 0 0.1 0.6];

% Plot the start and goal poses
hold on
scatter3(startPose(1),startPose(2),startPose(3),100,".r")
scatter3(goalPose(1),goalPose(2),goalPose(3),100,".g")
view([-31 63])
legend("","Start Position","Goal Position")
hold off
```

Occupancy Map
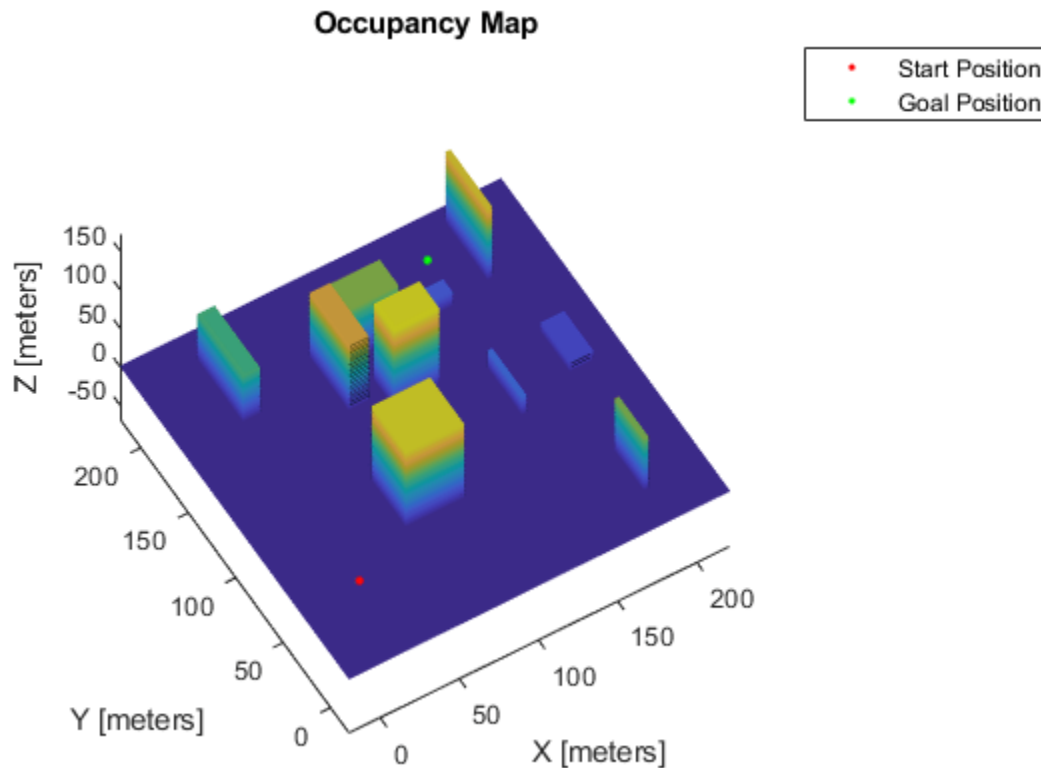
Plan Path with RRT* Using SE(3) State Space

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 with straight line connections using a `stateSpaceSE3` object. Use the `validatorOccupancyMap3D` object for collision checking between the multirotor UAV and the environment.

Define State Space Object

The `stateSpaceSE3` object defines the state space as `[x y z qw qx qy qz]`, where `[x y z]` specifies the position of the UAV in meters and `[qw qx qy qz]` specifies the orientation as a quaternion. Specify the position and orientation boundaries of the quadrotor as a 7-by-2 matrix. The orientation boundaries are optional, and can be set to `-Inf` and `Inf` if they are not applicable.

```
ss = stateSpaceSE3([-20 220;
                    -20 220;
                    -10 100;
                    inf inf;
                    inf inf;
                    inf inf;
                    inf inf]);
```

Define 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 locations 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 RRT* Path Planner

Create a `plannerRRTStar` object by specifying the state space and state validator as inputs. Set the
`MaxConnectionDistance`, `GoalBias`, `MaxIterations`, `ContinueAfterGoalReached`, and
`MaxNumTreeNodes` properties of the planner object to optimize the returned waypoints.

```
planner = plannerRRTStar(ss,sv);
planner.MaxConnectionDistance = 50;
planner.GoalBias = 0.8;
planner.MaxIterations = 1000;
planner.ContinueAfterGoalReached = true;
planner.MaxNumTreeNodes = 10000;
```

Execute Path Planning

Perform RRT* based path planning in 3D space to obtain waypoints. The planner finds a flight path
that is collision-free and suitable for the quadrotor. The solution info is helpful for tuning the planner.

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

Visualize Path

Check if the RRT* planner has found a path. If the planner has found a path, plot the waypoints.
Otherwise, terminate the script.

```
if (~solnInfo.IsPathFound)
    disp("No Path Found by the RRT, terminating example")
    return
end

% Plot map, start pose, and goal pose
show(omap)
hold on
scatter3(startPose(1),startPose(2),startPose(3),100,".r")
scatter3(goalPose(1),goalPose(2),goalPose(3),100,".g")

% Plot path computed by path planner
plot3(pthObj.States(:,1),pthObj.States(:,2),pthObj.States(:,3),"-g")
view([-31 63])
legend("","Start Position","Goal Postion","Planned Path")
hold off
```

Occupancy Map
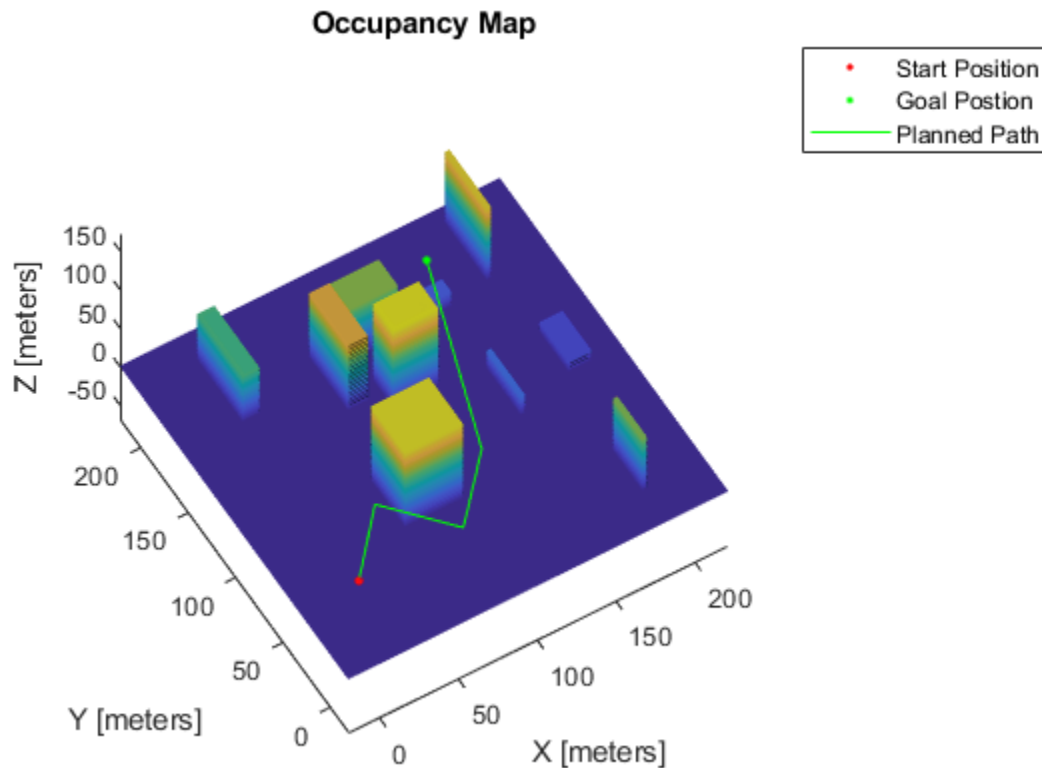
Generate Minimum Snap UAV Trajectory

The original planned path has some sharp corners while navigating toward the goal. Generate a smooth trajectory by fitting the obtained waypoints to the minimum snap trajectory using the `minsnappolytraj` function. For your initial estimate of the time required to reach each waypoint, assume the UAV moves at a constant speed.

To tune the trajectory and flight time, specify the `numSamples`, `TimeAllocation`, and `TimeWeight` arguments of the `minsnappolytraj` function.

```
waypoints = pthObj.States;
nWayPoints = pthObj.NumStates;

% Calculate the distance between waypoints
distance = zeros(1,nWayPoints);
for i = 2:nWayPoints
    distance(i) = norm(waypoints(i,1:3) - waypoints(i-1,1:3));
end

% Assume a UAV speed of 3 m/s and calculate time taken to reach each waypoint
UAVspeed = 3;
timepoints = cumsum(distance/UAVspeed);
nSamples = 100;

% Compute states along the trajectory
initialStates = minsnappolytraj(waypoints',timepoints,nSamples,MinSegmentTime=4,MaxSegmentTime=2(
```

Visualize Trajectory

Visualize the obtained minimum snap trajectory.

```
% Plot map, start pose, and goal pose
show(omap)
hold on
scatter3(startPose(1),startPose(2),startPose(3),30,".r")
scatter3(goalPose(1),goalPose(2),goalPose(3),30,".g")

% Plot the waypoints
plot3(pthObj.States(:,1),pthObj.States(:,2),pthObj.States(:,3),"-g")

% Plot the minimum snap trajectory
plot3(initialStates(:,1),initialStates(:,2),initialStates(:,3),"-y")
view([-31 63])
legend("","Start Position","Goal Postion","Planned Path","Initial Trajectory")
hold off
```
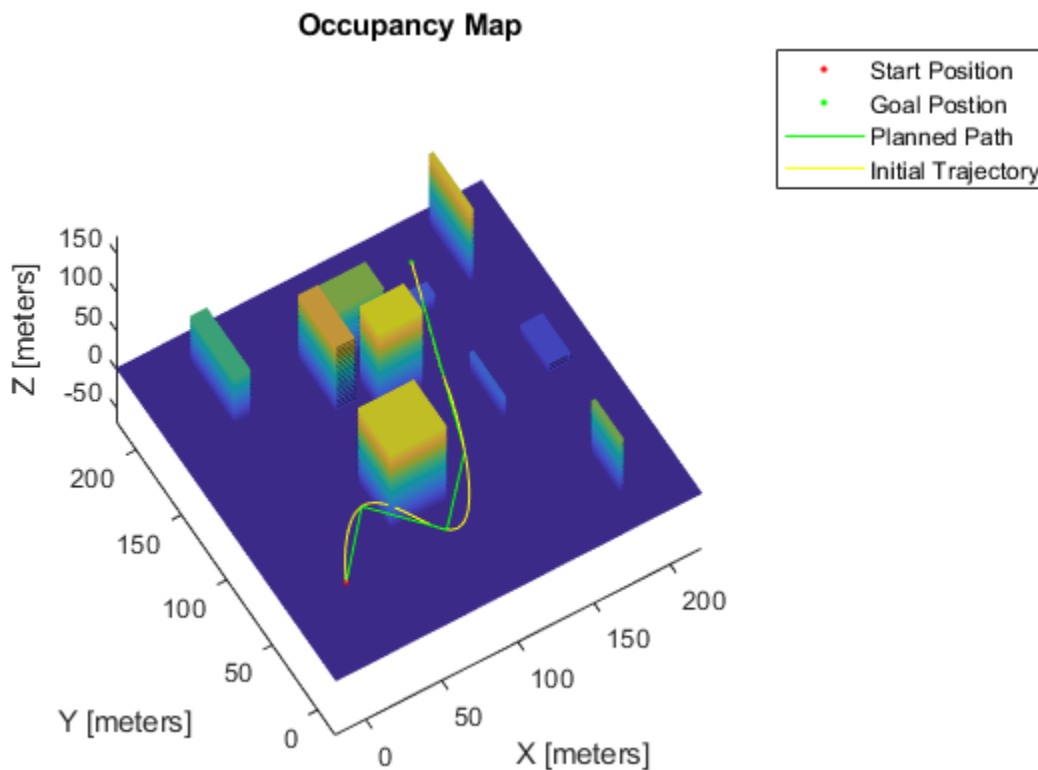

Generate Valid Minimum Snap Trajectory

Notice that the generated flight trajectory has some invalid states, which are not obstacle-free. You must modify the waypoints so that the generated trajectory is obstacle-free. To avoid invalid states, add intermediate waypoints on the segment between the pair of waypoints for which the trajectory is invalid. Insert waypoints iteratively until the entire trajectory is valid.

```
% Check if the trajectory is valid
states = initialStates;
```

```matlab
valid = all(isStateValid(sv,states));

while(~valid)
    % Check the validity of the states
    validity = isStateValid(sv,states);

    % Map the states to the corresponding waypoint segments
    segmentIndices = exampleHelperMapStatesToPathSegments(waypoints,states);

    % Get the segments for the invalid states
    % Use unique, because multiple states in the same segment might be invalid
    invalidSegments = unique(segmentIndices(~validity));

    % Add intermediate waypoints on the invalid segments
    for i = 1:size(invalidSegments)
        segment = invalidSegments(i);

        % Take the midpoint of the position to get the intermediate position
        midpoint(1:3) = (waypoints(segment,1:3) + waypoints(segment+1,1:3))/2;

        % Spherically interpolate the quaternions to get the intermediate quaternion
        midpoint(4:7) = slerp(quaternion(waypoints(segment,4:7)),quaternion(waypoints(segment+1,4
        waypoints = [waypoints(1:segment,:); midpoint; waypoints(segment+1:end,:)];

    end

    nWayPoints = size(waypoints,1);
    distance = zeros(1,nWayPoints);
    for i = 2:nWayPoints
        distance(i) = norm(waypoints(i,1:3) - waypoints(i-1,1:3));
    end

    % Calculate the time taken to reach each waypoint
    timepoints = cumsum(distance/UAVspeed);
    nSamples = 100;
    states = minsnappolytraj(waypoints',timepoints,nSamples,MinSegmentTime=2,MaxSegmentTime=20,T:

    % Check if the new trajectory is valid
    valid = all(isStateValid(sv,states));

end
```

Visualize Final Valid Trajectory

Visualize the final valid minimum snap trajectory.

```matlab
% Plot map, start and goal pose
show(omap)
hold on
scatter3(startPose(1),startPose(2),startPose(3),30,".r")
scatter3(goalPose(1),goalPose(2),goalPose(3),30,".g")

% Plot the waypoints
plot3(pthObj.States(:,1),pthObj.States(:,2),pthObj.States(:,3),"-g")

% Plot the initial trajectory
plot3(initialStates(:,1),initialStates(:,2),initialStates(:,3),"-y")
```

```
% Plot the final valid trajectory
plot3(states(:,1),states(:,2),states(:,3),"-c")
view([-31 63])
legend("","Start Position","Goal Postion","Planned Path","Initial Trajectory","Valid Trajectory")
hold off
```
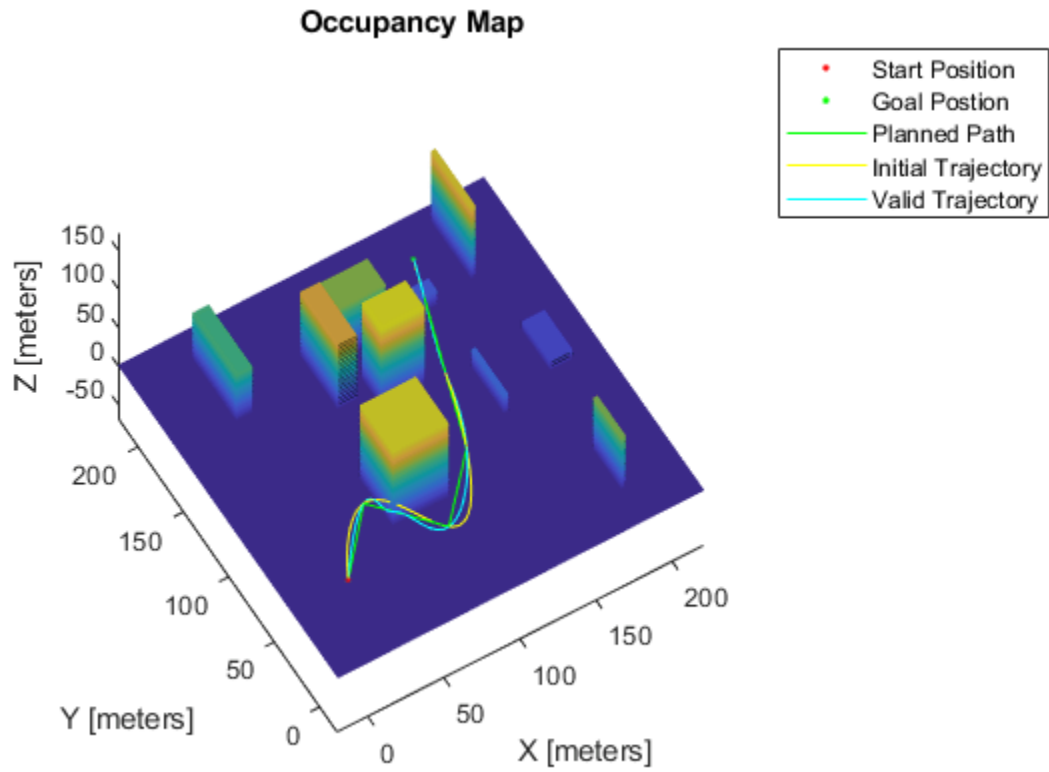
Tune UAV Parameters Using MAVLink Parameter Protocol

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 set up 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.

Set Up 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 set up 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.

Set Up 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);
```

Set Up 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);
```

Set Up 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_REQUET_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 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`.

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 UAVnode. 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.AutopilotTy
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 heartbeast 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.
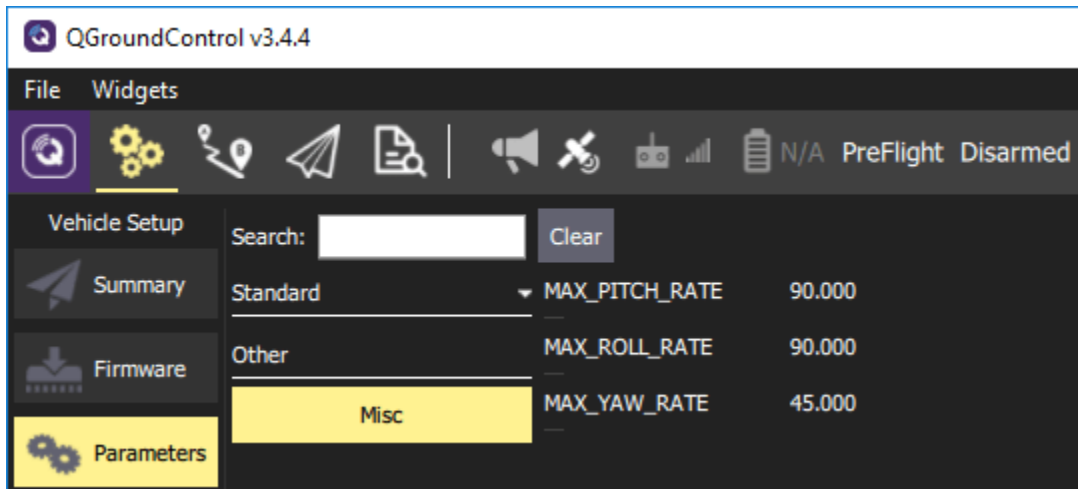
Note: Because we use a generic autopilot type, `"MAV_AUTOPILOT_GENERIC"`, QGroundControl does not recognize the connection as a known autopilot type. This does not affect the connection and the parameter values should still update as shown.

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
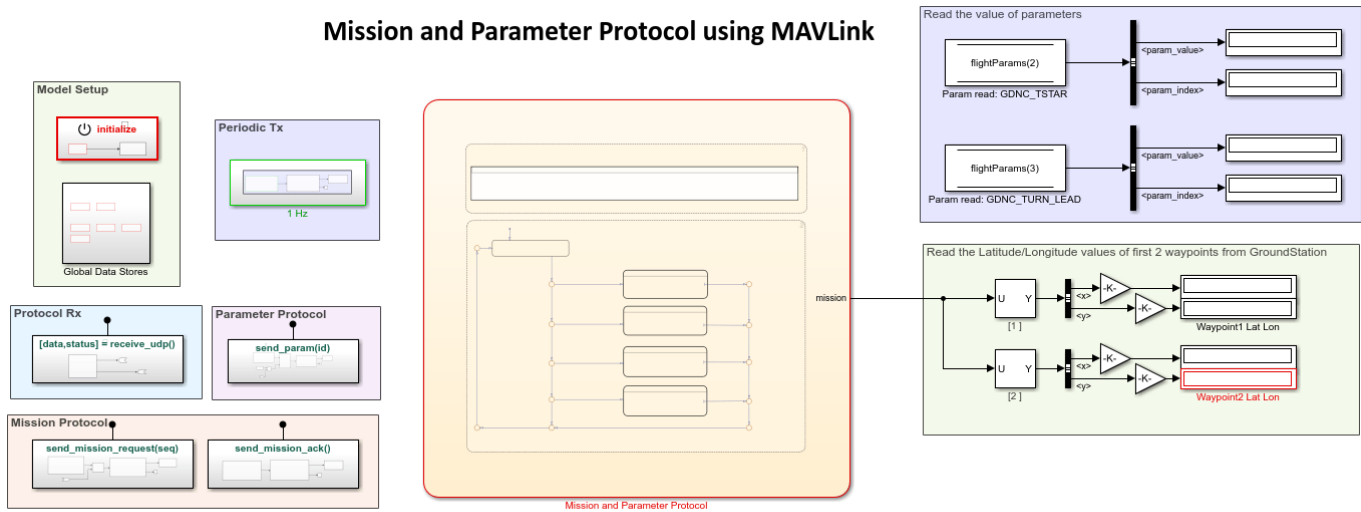
Copyright 2020 The MathWorks, Inc.

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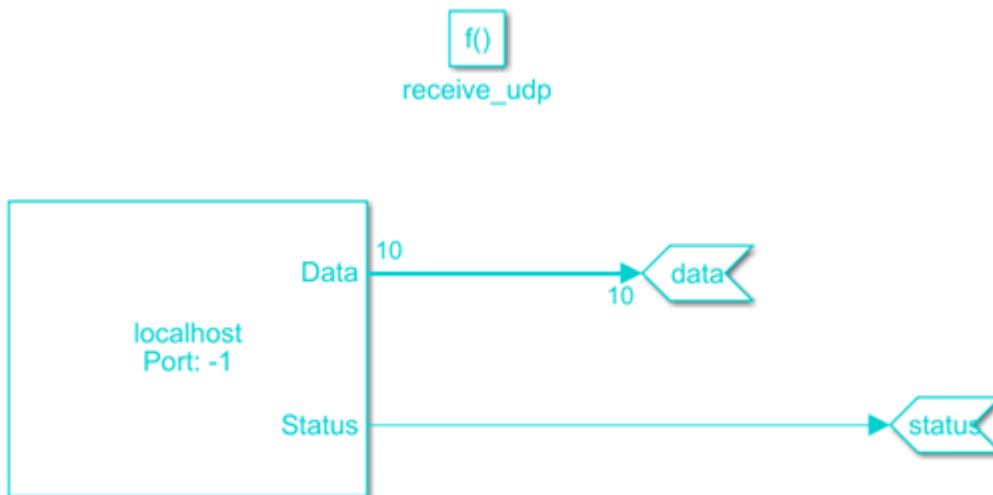
3. **Mission Protocol**: This area in the model consists of two subsystem blocks that send mission requests and mission acknowledgments to the QGC. These functions are called from the Stateflow chart that implements the mission microservice.

4. **Mission and Parameter Protocol**: The Stateflow chart that implements the mission and parameter logic in the model.
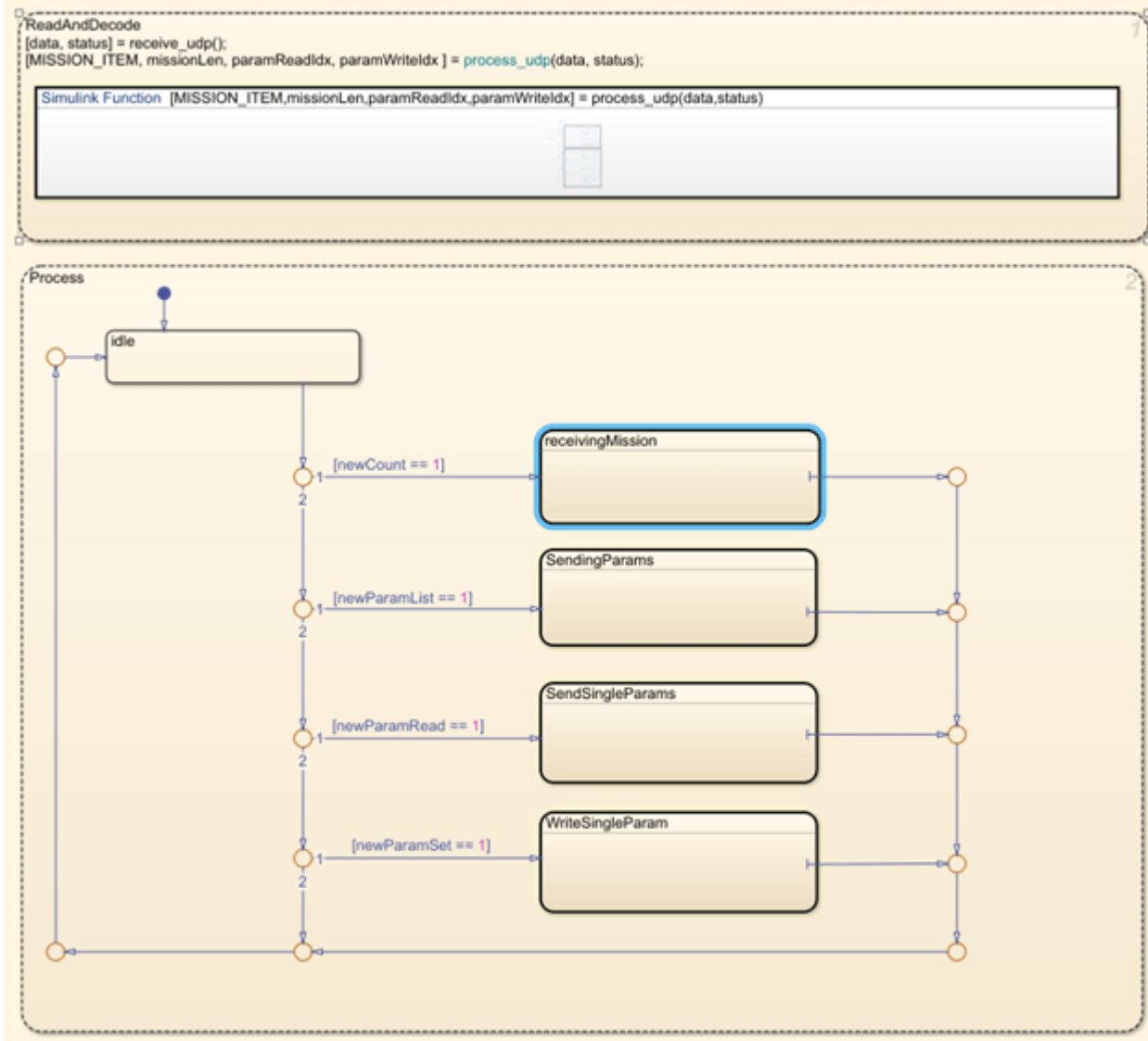
The received MAVLink data is deserialized in the `process_udp` Simulink function and then passed to the Stateflow logic that performs four tasks:

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.
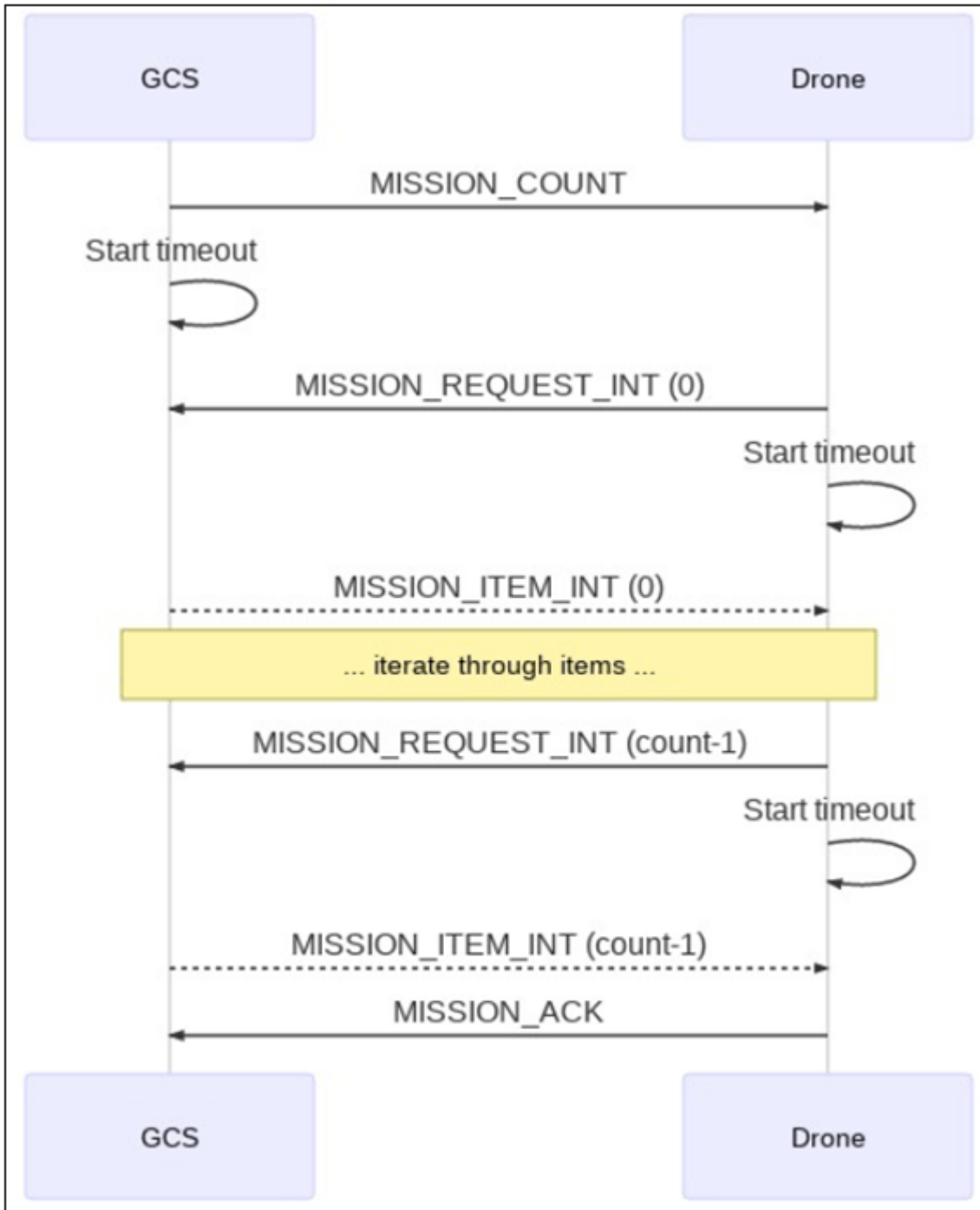
Image courtesy: https://mavlink.io/en/services/mission.html#uploading_mission

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.
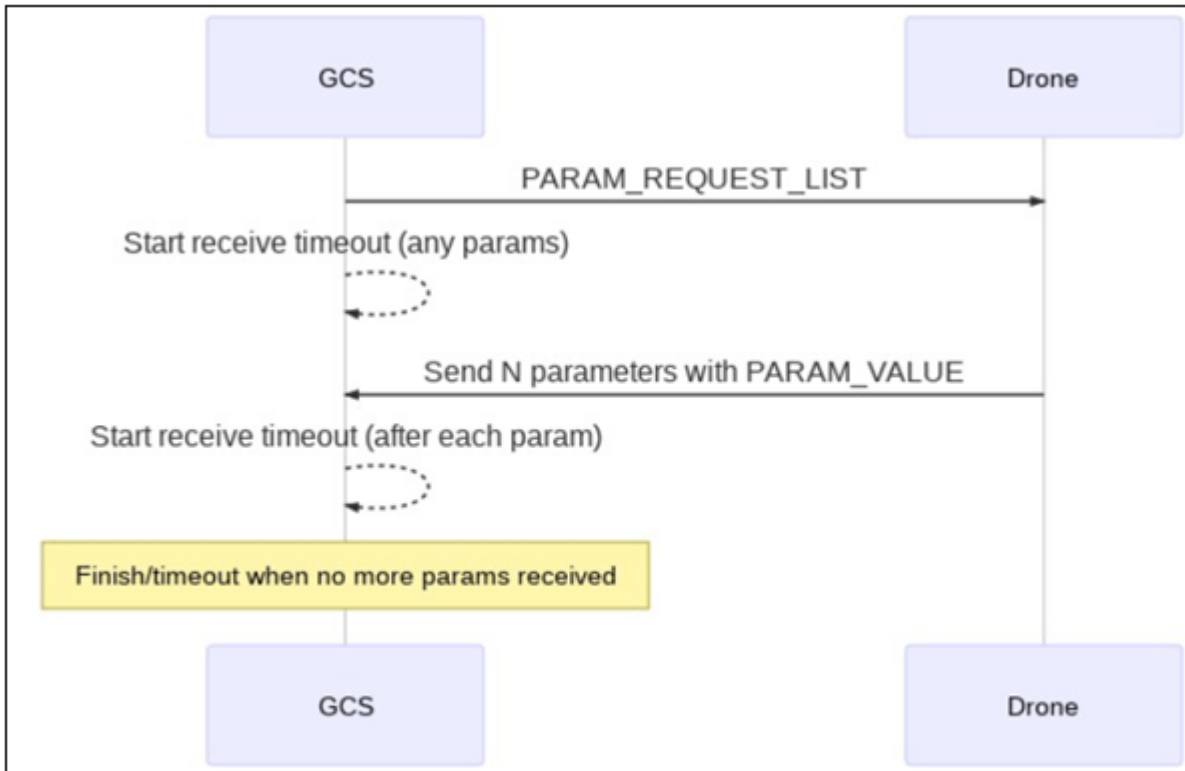
Image courtesy: https://mavlink.io/en/services/parameter.html#read_all

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
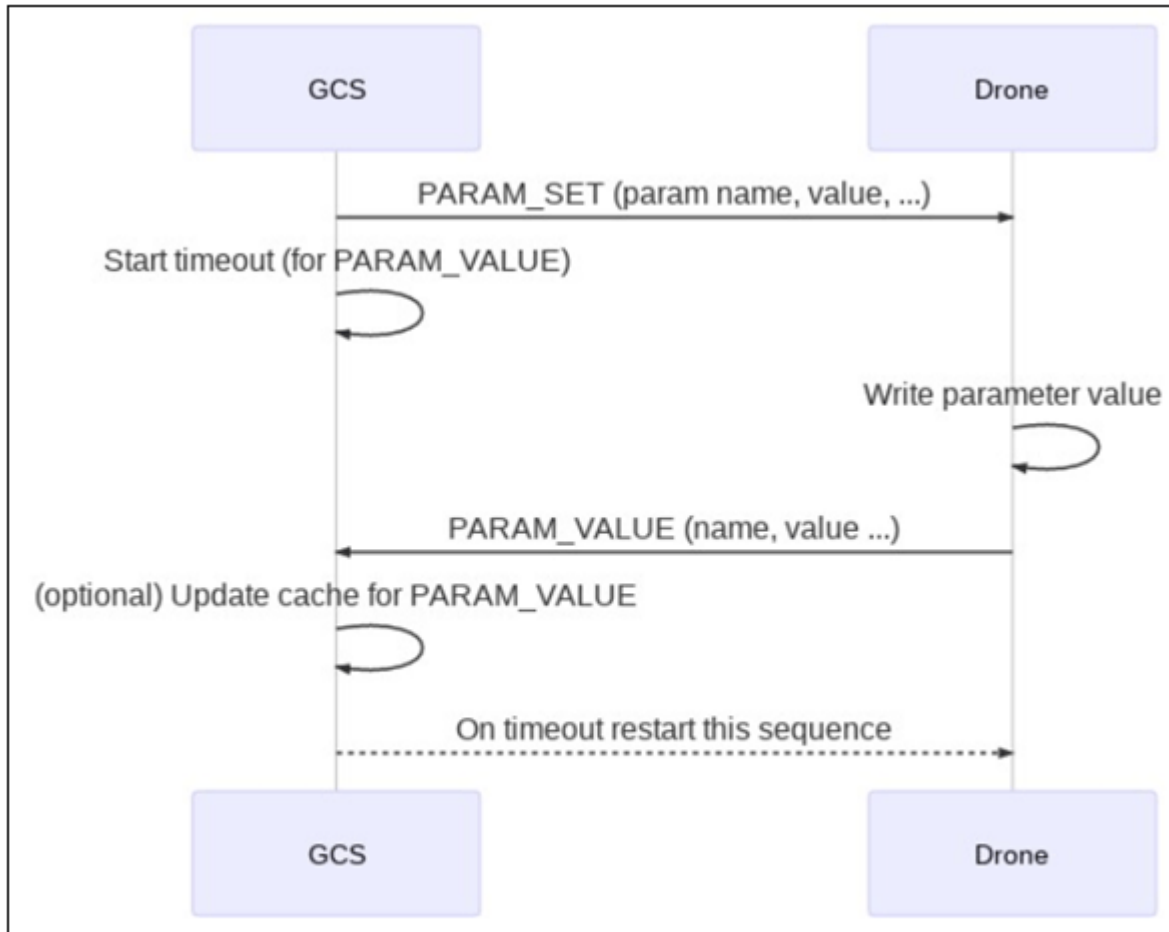
5. **Logic to read received waypoints and parameters**: Stateflow implements the two protocols and outputs the received waypoints and uploaded parameter values.
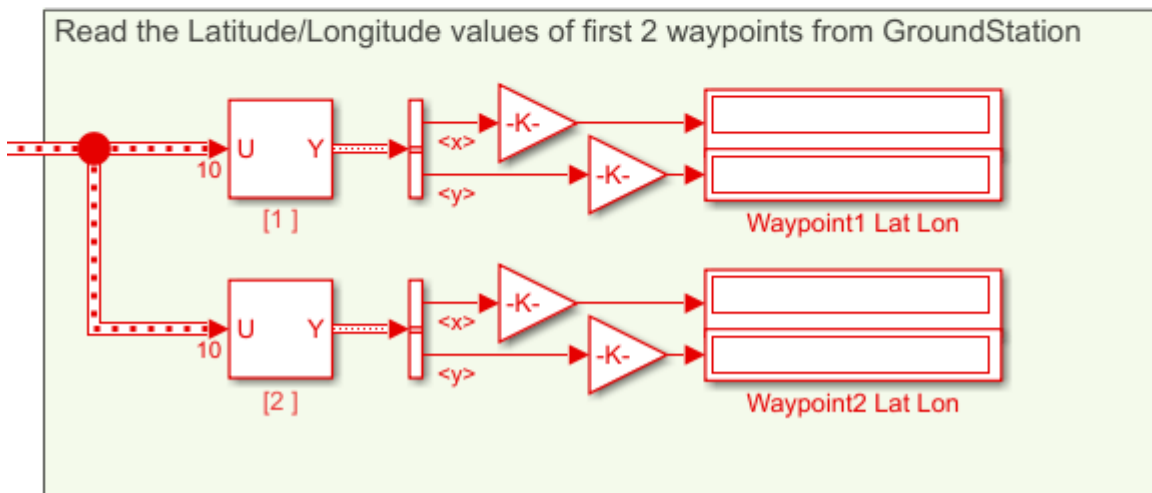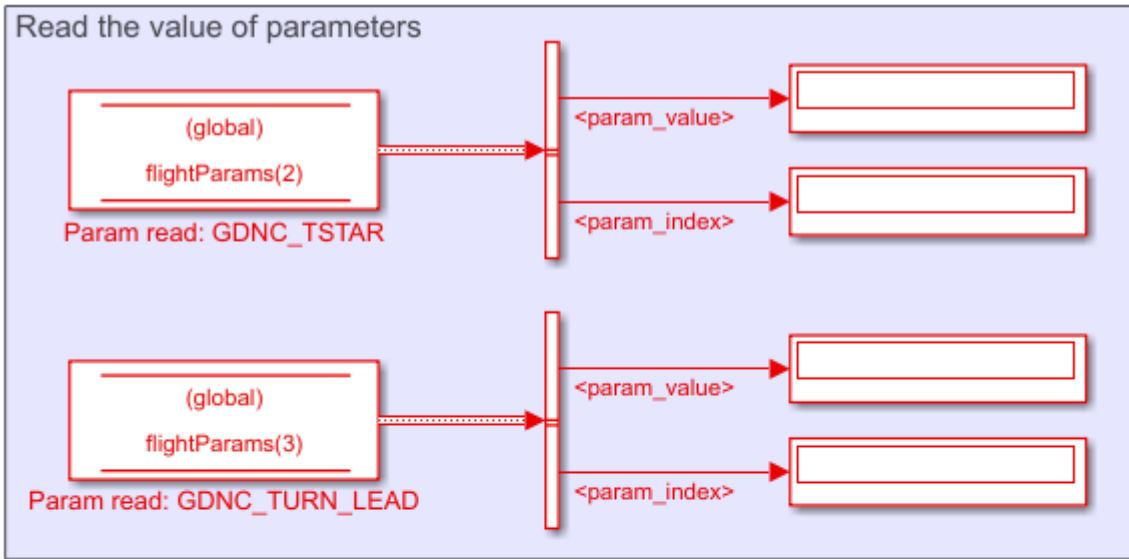
The next section explains how to upload a mission from the QGC to the drone.

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

1. Launch the QGC and navigate to the *Plan View*.
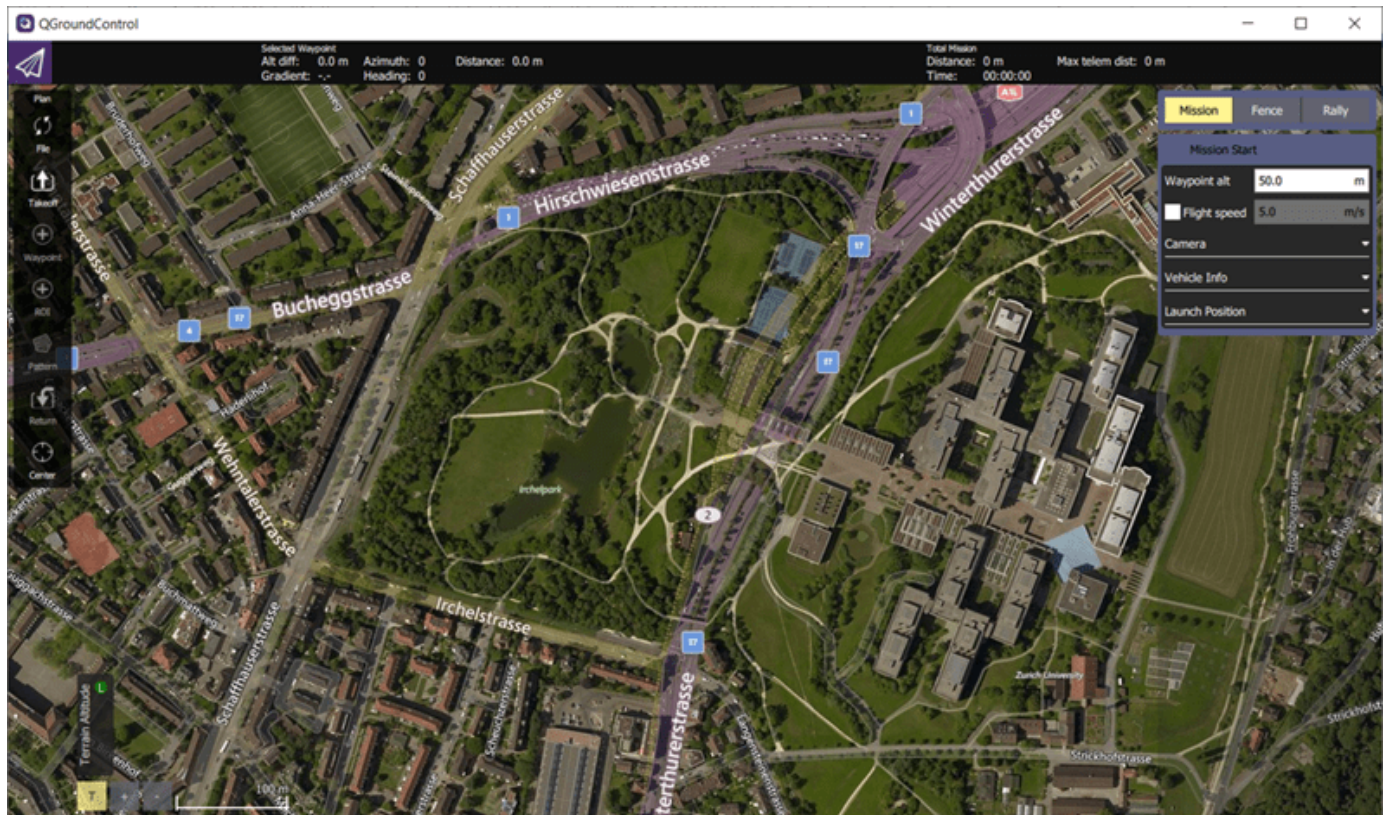
2. A preplanned mission, *MissionProtocol.plan*, is available with this example. Click **Open Model** at the top of this page to save the plan file to your computer. After you save the .plan file, launch QGC, and click **File > Open** to upload the plan to the QGC.
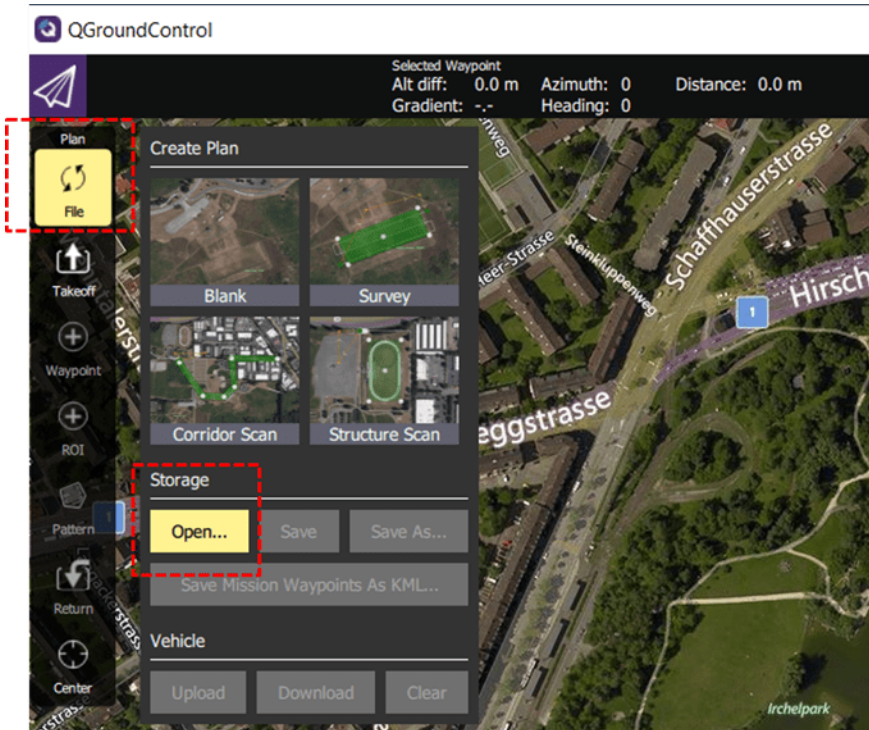
After you upload the plan, the mission is visible in QGC.
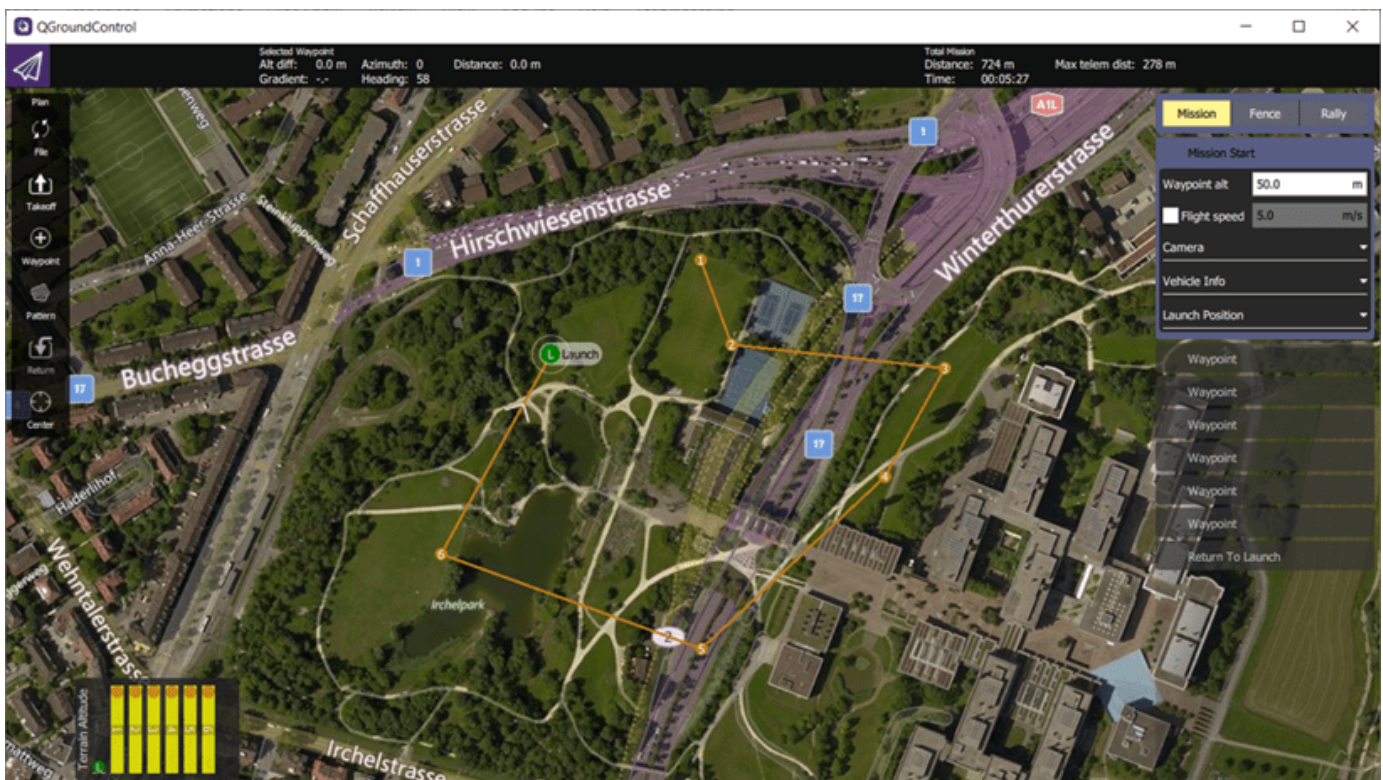
3. Run the Simulink model. The Simulink model sends HEARTBEAT message over MAVLink to QGC and thus establish connection with QGC.

4. Click **Upload** at the top right of QGC interface to upload the mission from QGroundControl.
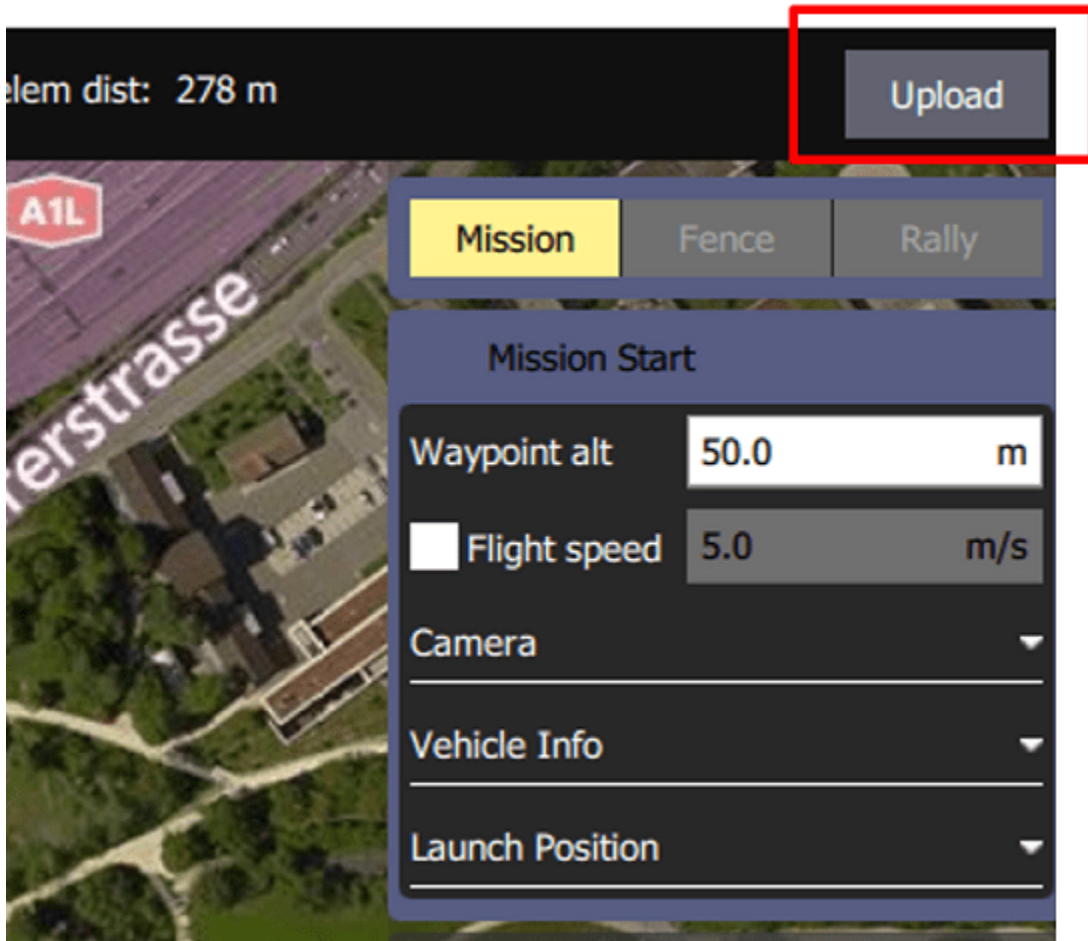
5. Observe that the latitude and longitude values from the first two waypoints of the uploaded mission are being displayed in Simulink.
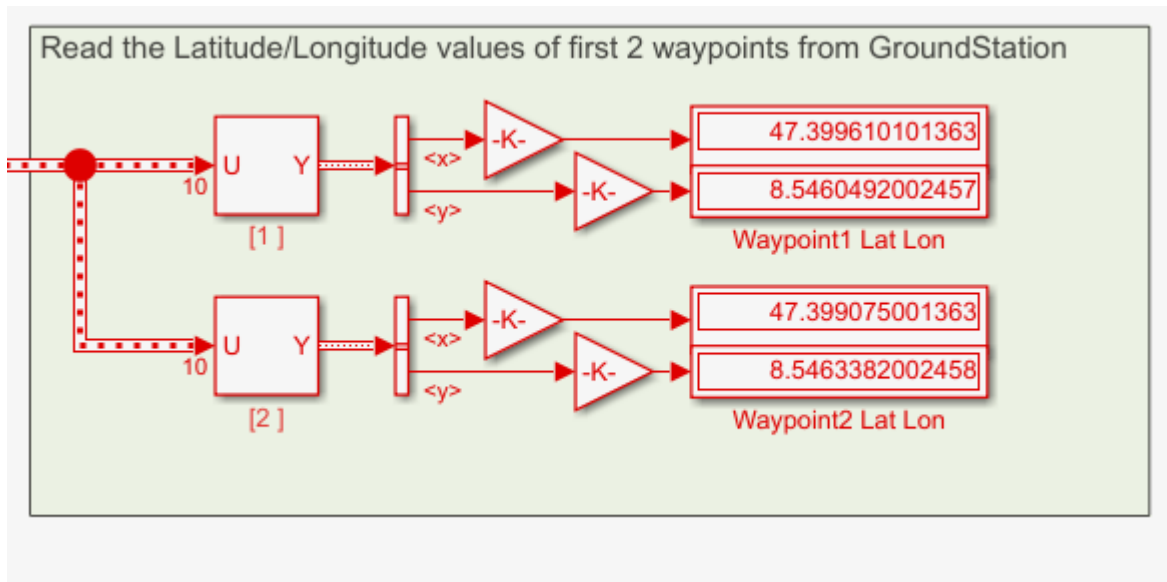
6. Change the waypoint1 and waypoint2 in the QGC by dragging the waypoints to a different location in the plan. Upload the modified mission by clicking **Upload Required**.
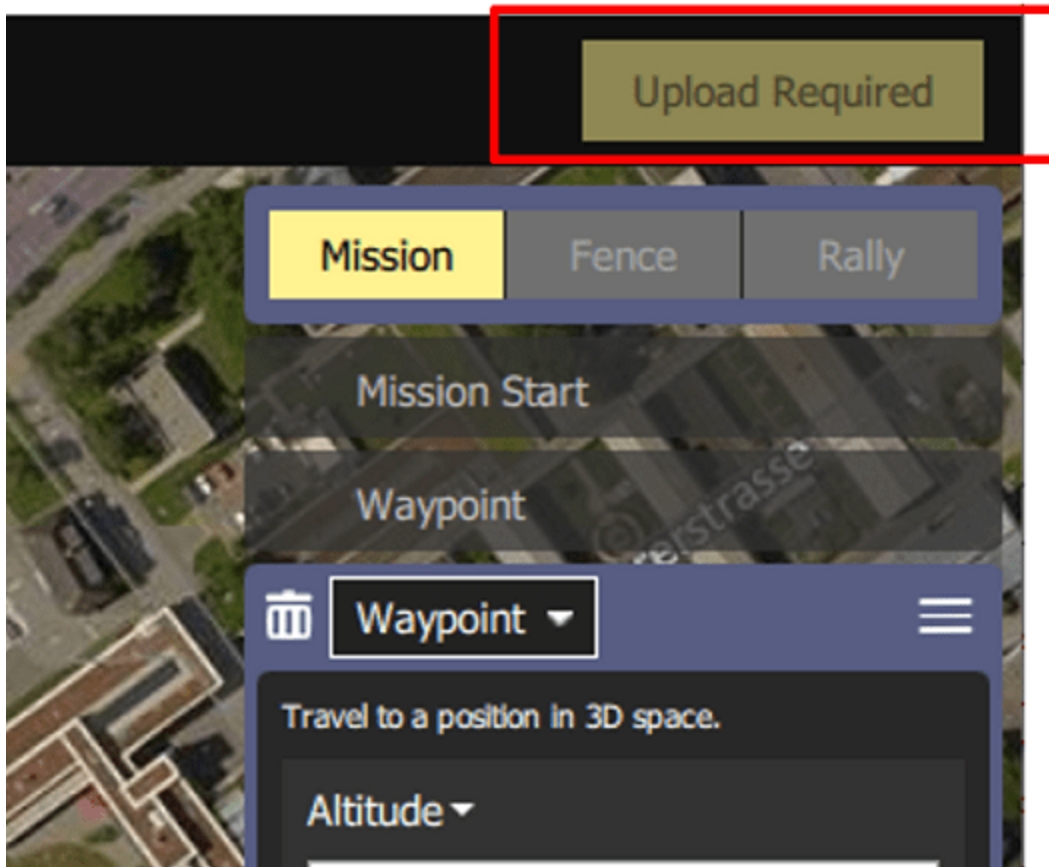
7. Observe the modified Latitude/Longitude values for waypoint 1 and 2 in Simulink.

Read the Latitude/Longitude values of first 2 waypoints from GroundStation
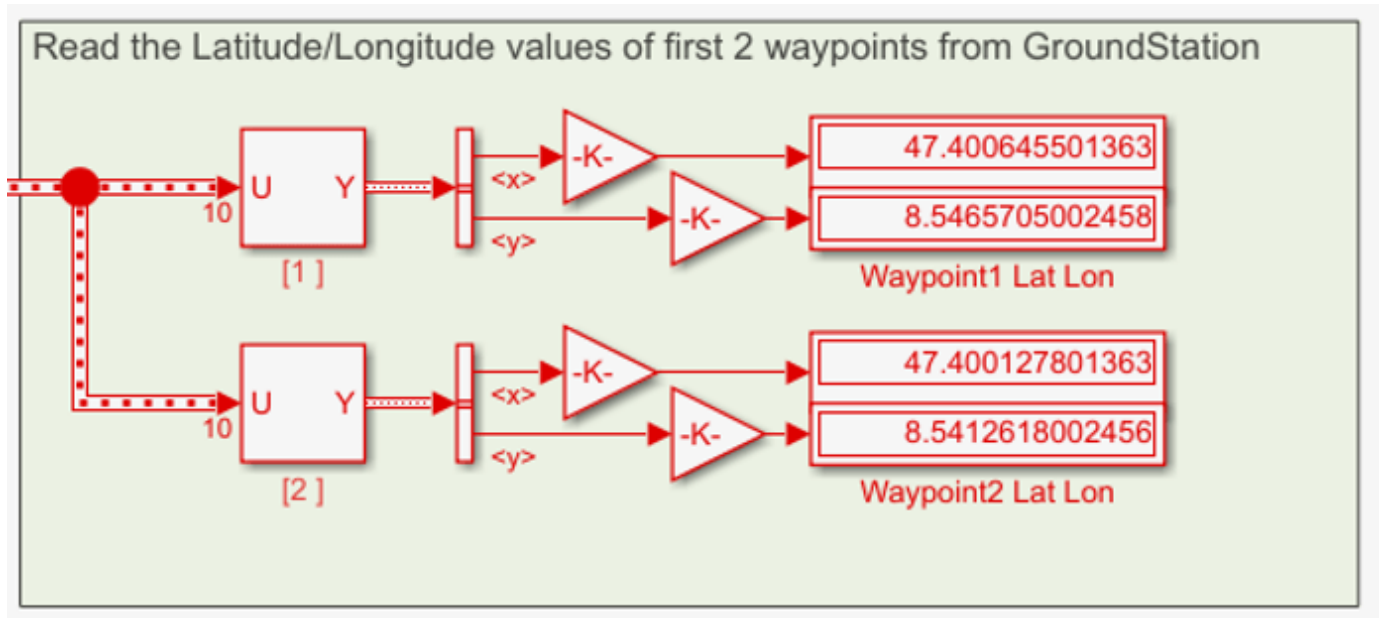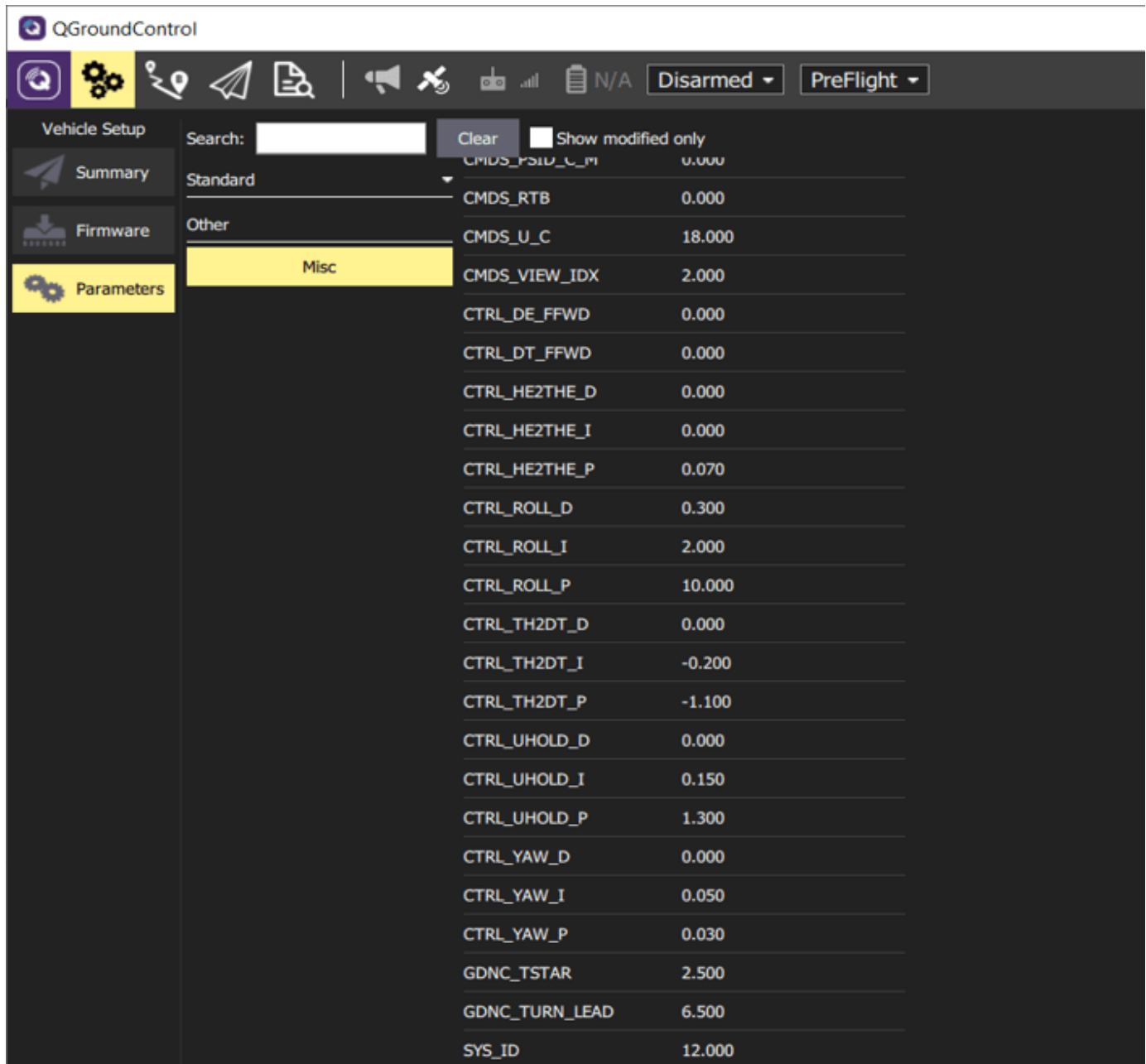
Modify Parameters in QGC and Send Them to Simulink

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.

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.

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.
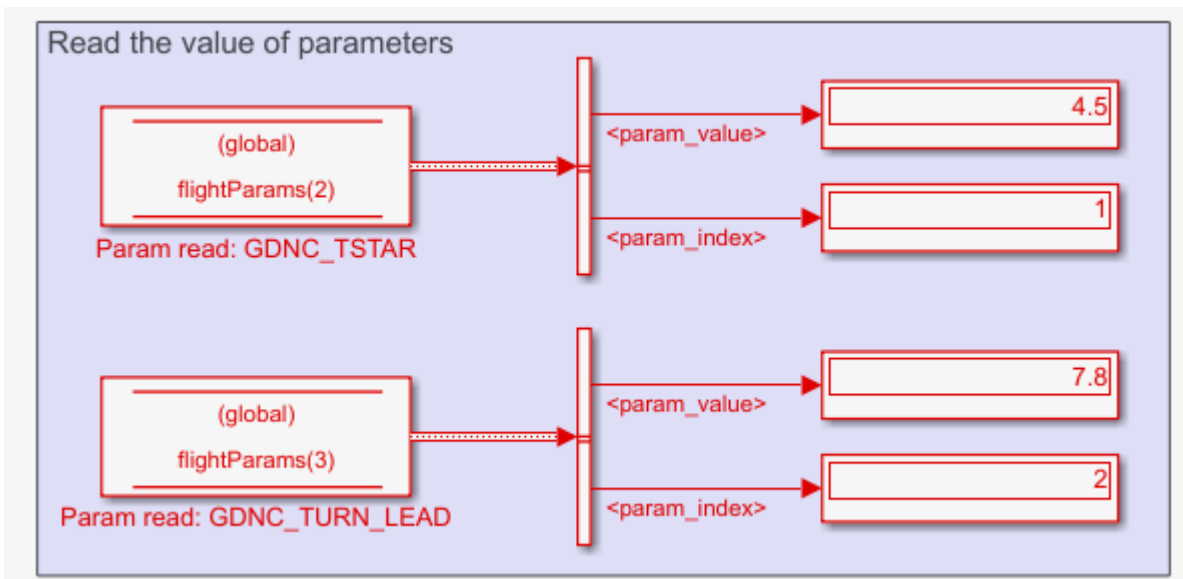
Other Things to try

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.

Onboard Computer Path Planning Interface for PX4 SITL Deployable on NVIDIA Jetson

This example demonstrates enabling and interfacing onboard computer path planning with PX4 Software-in-the-Loop (SITL).

Note: This example uses NVIDIA Jetson as the onboard computer.

In this example you:

- Enable onboard computer workflows with PX4 SITL.

- Implement a PX4 path planning interface in Simulink and deploy on NVIDIA Jetson.

- Establish MAVLink communication between the onboard computer and PX4 SITL.

- Run and complete a UAV mission with onboard computer assisted path planning.

The following diagram illustrates high level interface between multiple components used in this example.
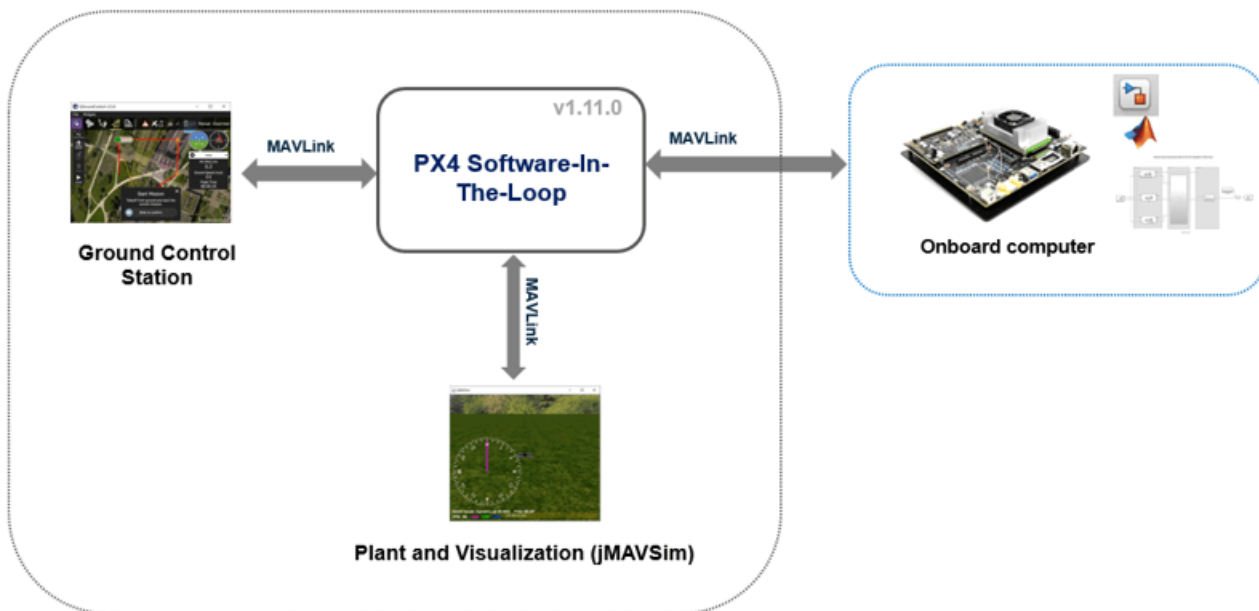
QGroundControl (QGC) is the ground control station software, which helps you to configure PX4 autopilot software. In this example, you use QGC to create, upload, and monitor a UAV mission. When PX4 SITL is in mission mode, it sends the mission waypoints to the onboard computer (NVIDIA Jetson), which is connected to the same network over MAVLink using PX4 path planning interface. The onboard computer uses the Waypoint Follower block from UAV Toolbox to generate trajectory to follow these waypoints. The updated trajectory waypoints output from the Waypoint Follower block are sent back to the PX4 SITL over MAVLink using the PX4 path planning interface.

Prerequisites

- If you are new to Simulink, watch the Simulink Quick Start video.

- If you are new to PX4, refer to PX4 Autopilot User Guide to understand the PX4 flight stack for UAVs.

This example uses:

- MATLAB®

- Simulink®

- UAV Toolbox™

- MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms

This example requires the following third-party software:

- PX4 flight stack for UAVs

- Cygwin Toolchain (For Windows Operating System)

- QGroundControl (QGC)

Install Required Applications

PX4 Source Code

For this example, use PX4 Autopilot Firmware v1.11.0.

Download PX4 Source Code from Github (For Ubuntu)

1. Open the bash terminal on Ubuntu 18.04.

2. Create a directory `mypx4` in the home folder.

3. Go to the newly created px4 directory, and run the following commands one-by-one. Wait for each command to execute before entering the next command.

- `git clone https://github.com/PX4/Firmware.git Firmware`

- `cd Firmware`

- `git checkout v1.11.0`

- `git submodule update --init --recursive`

4. Install the auxiliary packages required by running the ubuntu.sh script. Enter the following command (while in the Firmware folder) in the terminal.

- `bash ./Tools/setup/ubuntu.sh`

5. Restart the computer or log off and log in again after the previous process is complete.

Download PX4 Source Code from Github (For Windows)

1. The Cygwin toolchain is required to build the PX4 Firmware in Windows. Download version 0.8 of the PX4 Cygwin Toolchain MSI Installer, compatible with PX4 Firmware v1.11.0, available here.

2. Install the PX4 Cygwin Toolchain MSI Installer. At the last step of the PX4 Toolchain Setup wizard. select the option **Clone PX4 repository and Start Simulation**, and then click **Finish**. Doing so clones the latest master PX4 Firmware.
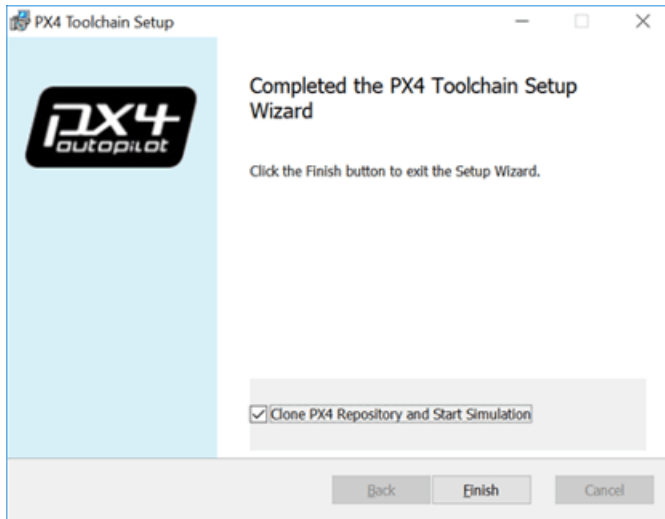
3. Wait for the firmware to finish cloning. After the firmware is cloned, Simulation starts in jMAVSim. Close the bash shell at this stage.

4. The PX4 firmware is cloned inside a folder named `home`, inside the Cygwin folder that you selected during installation, for example, `C:\px4_cygwin\home\`.

5. Navigate to the installed Cygwin Toolchain directory on your PC. For example, `C:\px4_cygwin`.

6. Launch the batch file `run-console.bat`. Doing so opens the Cygwin console.

7. From the Cygwin console, navigate to the PX4 directory.

- `cd home`

8. Go to the newly created PX4 directory, and run the following commands. Wait for each command to execute before entering the next command.

- `cd Firmware`

- `git checkout v1.11.0`

- `git submodule update --init --recursive`

QGroundControl

QGC provides full flight control and vehicle setup for PX4 powered vehicles. For QGC installation instructions, see Download and install QGC. This example uses QGC v3.5.6.

Step 1: Configure and Run the Model

To get started, follow these steps:

1. Open the example model by executing this command at the MATLAB command prompt.

```
open_system('OnboardComputer_PathPlanning_PX4_SITL_NVIDIA_Jetson.slx')
```

Onboard computer path planning interface for PX4 SITL deployable on NVIDIA Jetson

Copyright 2021 The MathWorks, Inc.
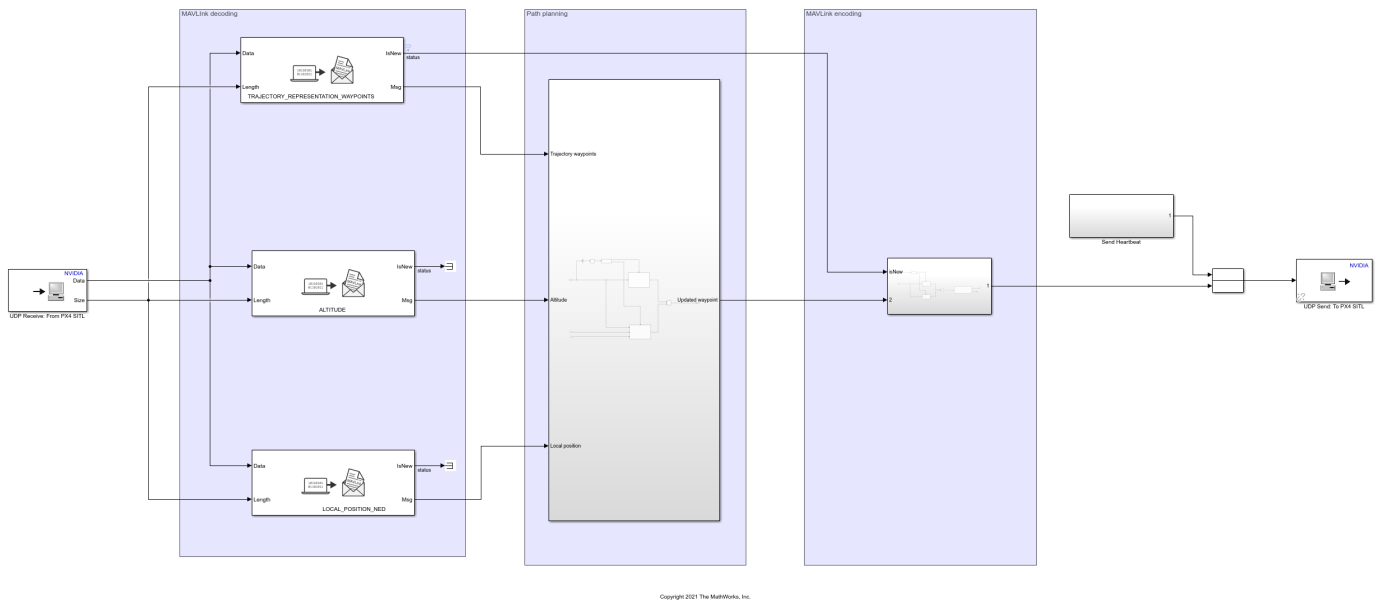
This model implements the PX4 path planning interface using MAVLink Serializer and MAVLink Deserializer blocks. Refer to PX4 Path Planning Interface for more details. The MAVLink messages that are exchanged as part of this interface are shown in the following diagram.
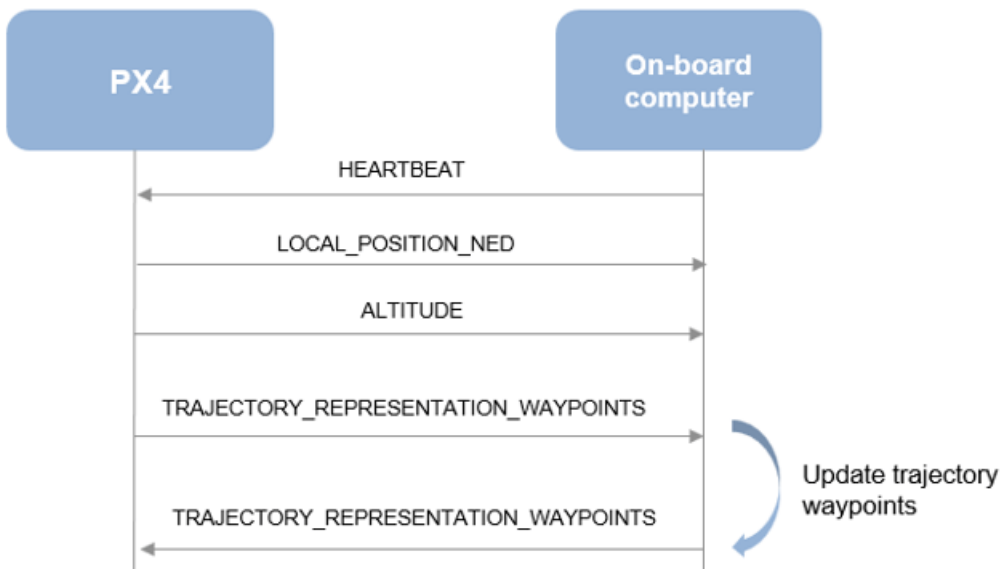
2. In **Target hardware resource** > **Board Parameters**, enter the IP address of the NVIDIA Jetson and your login credentials.

3. Find the IP address of the NVIDIA Jetson board connected. Ensure that the NVIDIA Jetson is connected to the same network as your host PC. In the MATLAB Command Window, run the `exampleHelperUpdateStartupScriptforOBC` script, attached to this example. The first input is the path to the PX4 source downloaded and second input is the IP address of the NVIDIA Jetson board. This script modifies the startup script of PX4 SITL to enable MAVLink connectivity to NVIDIA Jetson.

```
exampleHelperUpdateStartupScriptforOBC('C:\px4_cygwin\home\','172.19.XX.XX');
```

4. Update the MAVLink Dialect path of the MAVLink Serializer and MAVLink Deserializer blocks in the example to the MAVLink Dialect location in the PX4 source downloaded. A script is provided with the example to automate this. Run the script `exampleHelperUpdateMAVLinkDialectPath`, attached to this example to update the MAVLink Dialect path in the model. Input is the path to PX4 source downloaded.

```
exampleHelperUpdateMAVLinkDialectPath('C:\px4_cygwin\home\');
```
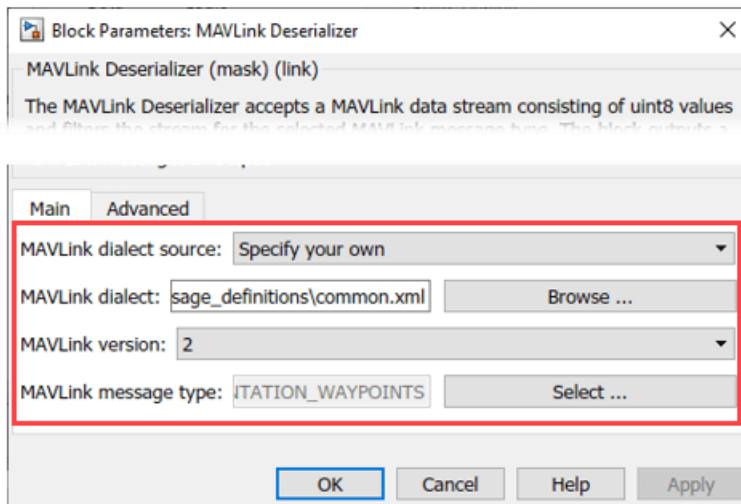
Note: The script provided with this example is specific to this example. If you are adding MAVLink Serializer and Deserializer blocks to this model or creating a new model, this script does not work. You can either update the script or update the block mask manually.

5. Open the block mask of the UDP Send block in the model. In the **Remote IP address** field, enter IP address of the host PC on which you are running PX4 SITL. You can also achieve this by defining a variable hostIP in the workspace. Ensure that the host PC and NVIDIA Jetson are connected to same network.

```
hostIP ='172.19.XX.XX';
```
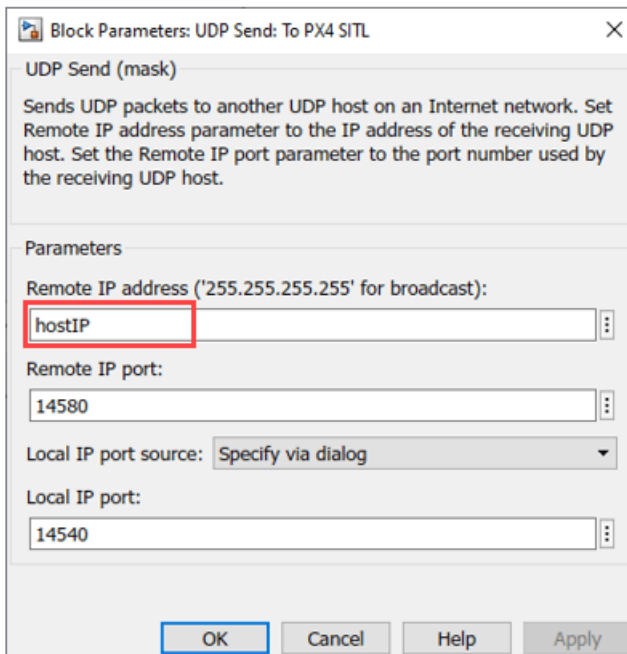

6. Run the example model in External mode by selecting the **Hardware** tab and then clicking **Monitor & Tune**. Ensure that the simulation is started.

Note: If you get "Build failed because the build file name(s) exceed the Windows limit of 260 characters. Build from a working directory with a shorter path." error, change the path to a different working directory and then run the example model in External mode.

Step 2: Launch PX4 SITL

Launch PX4 SITL with jMAVSim as the Flight Simulator.

On Windows, in Cygwin Console navigate to the PX4 directory and execute these commands:

On Ubuntu, navigate to the PX4 directory in terminal and run these commands:

- `cd Firmware`

- `make px4_sitl jmavsim`

PX4 SITL and jMAVSim launch.
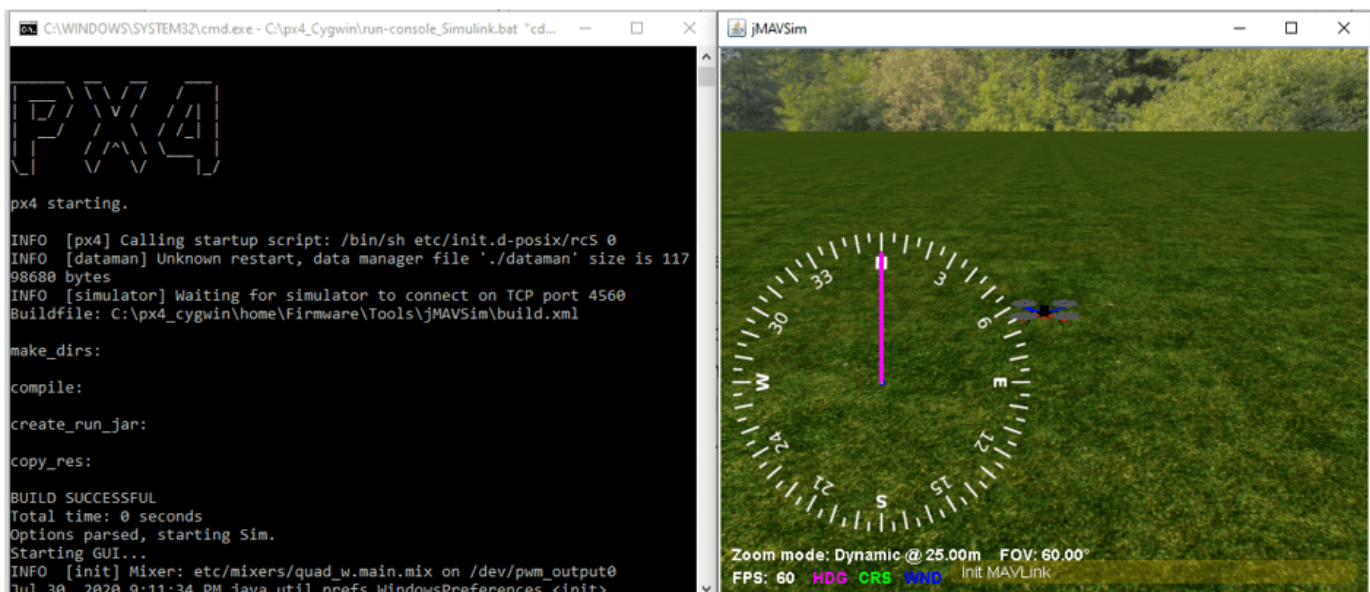
Step 3: Open QGC and Enable PX4 Path Planning Interface

1. Launch the QGC Application and wait till it connects to the PX4 SITL.

2. Set the PX4 parameter `COM_OBS_AVOID` to enable the PX4 path planning interface. Navigate to Parameters from the main menu and set the `COM_OBS_AVOID` parameter value to 1.
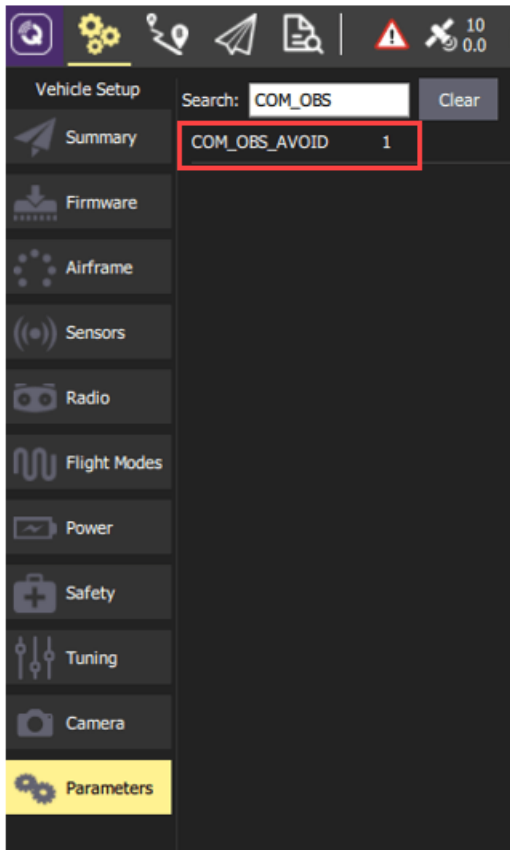
Step 4: Create, Upload, and Start Mission from QGC

You can create a mission or use the preplanned mission, `example_mission.plan`, which is available with this example. Perform these steps:

1. Launch the QGC and navigate to the Plan View.
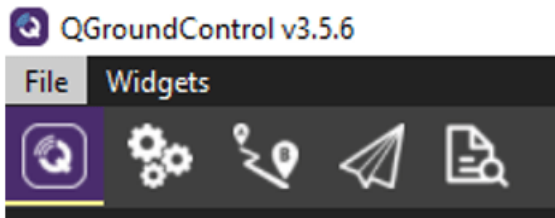
2. Create a mission or upload the preplanned mission, `example_mission.plan`.

- Create a mission: For information on creating a mission, see Plan View.

- To upload the preplanned mission, click **Open Model** at the top of this page and download the plan file (`example_mission.plan`) to your computer. In the QGC, navigate and select `example_mission.plan` file.
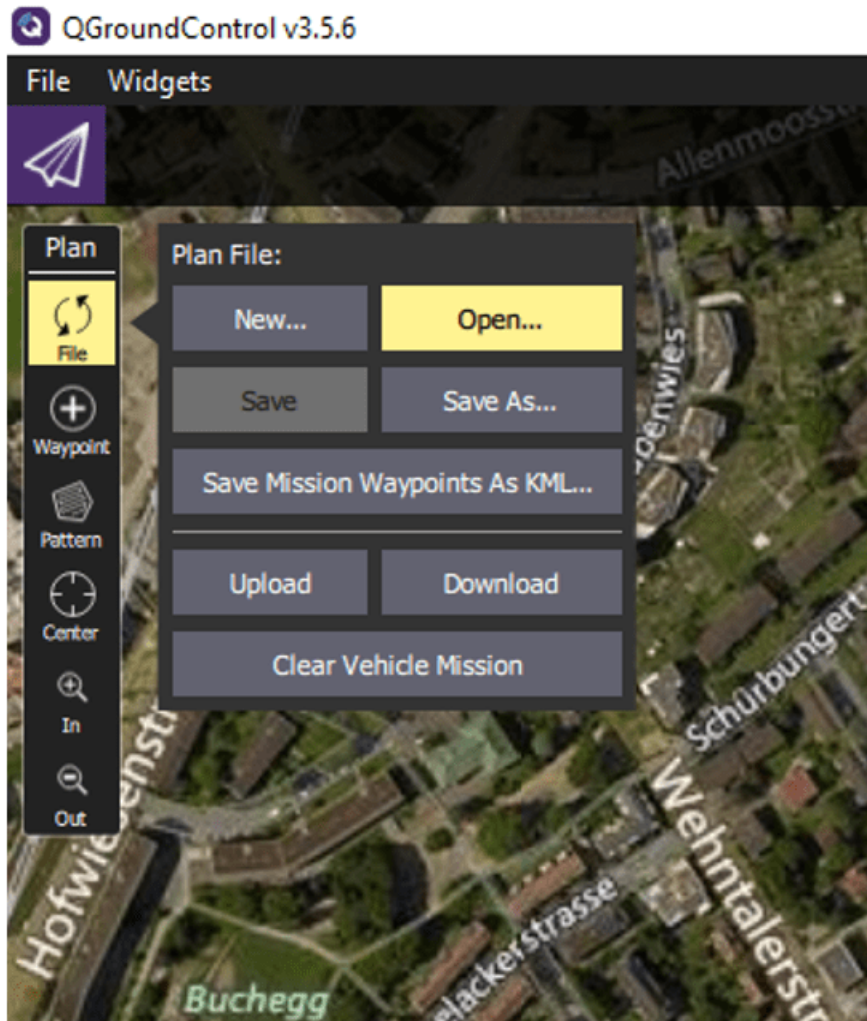
After you upload the plan, the mission is visible in QGC.
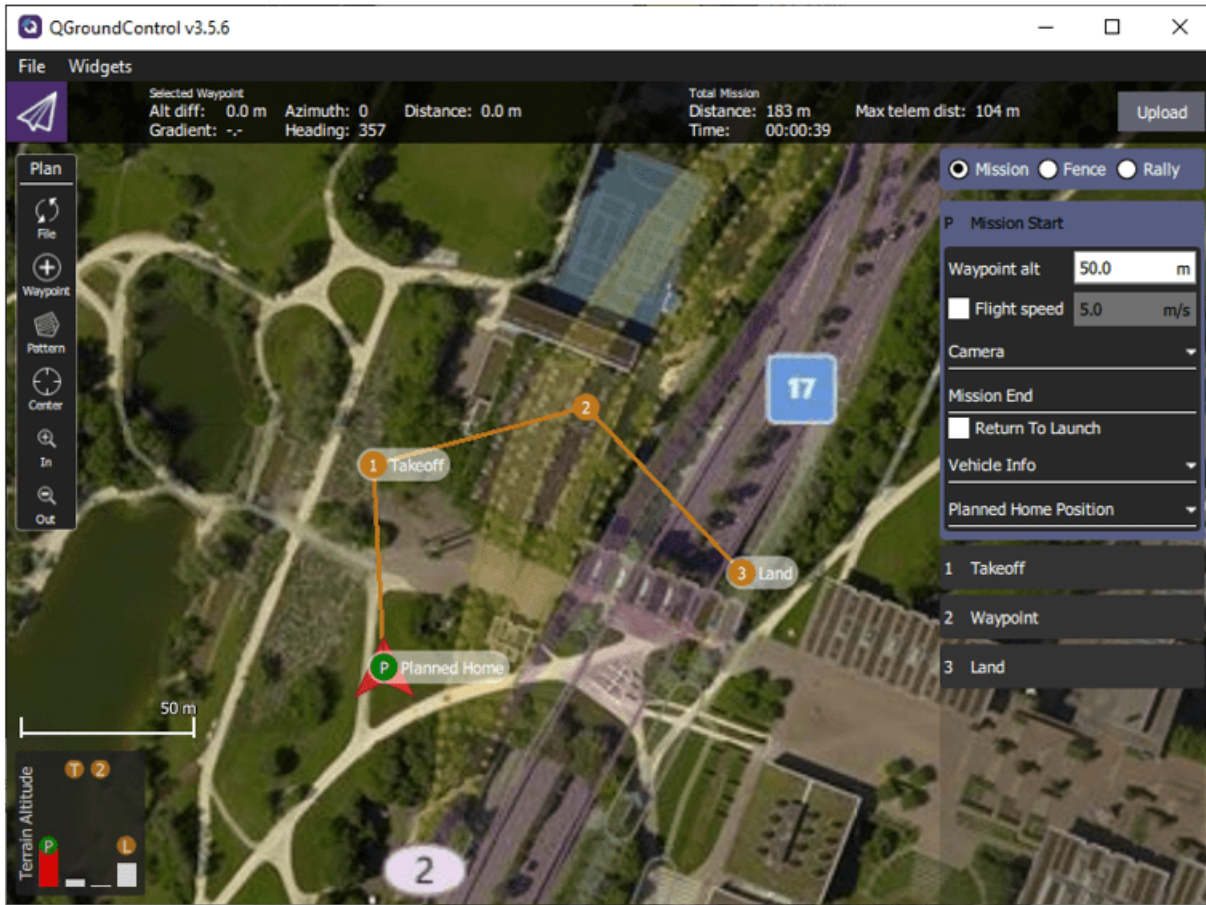
3. Click **Upload** in the QGC interface to upload the mission from QGroundControl.

4. Move to Fly View to see the uploaded mission.
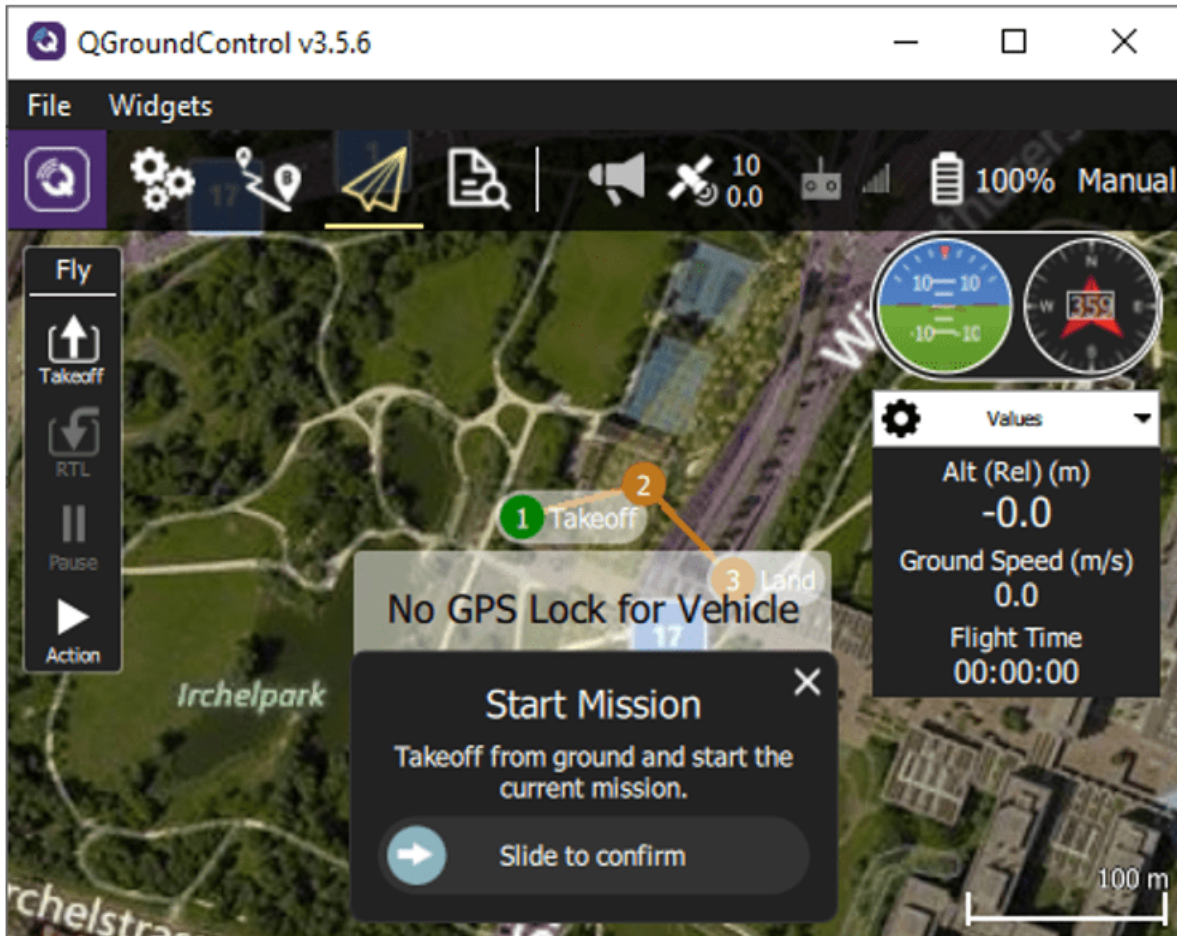
5. Ensure that GPS fusion is completed and the GPS location is updated in QGC.


```
INFO  [ecl/EKF] 2584000: reset velocity to zero
INFO  [ecl/EKF] 9884000: GPS checks passed
INFO  [ecl/EKF] 14424000: reset position to GPS
INFO  [ecl/EKF] 14424000: reset velocity to GPS
INFO  [ecl/EKF] 14424000: starting GPS fusion
```
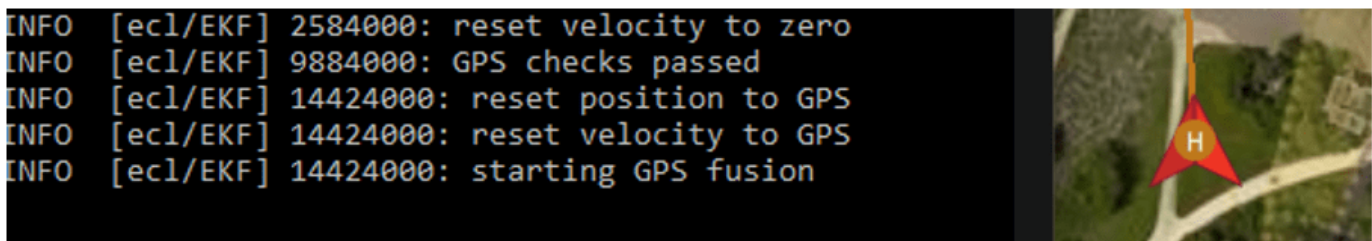
6. Start the mission from QGC
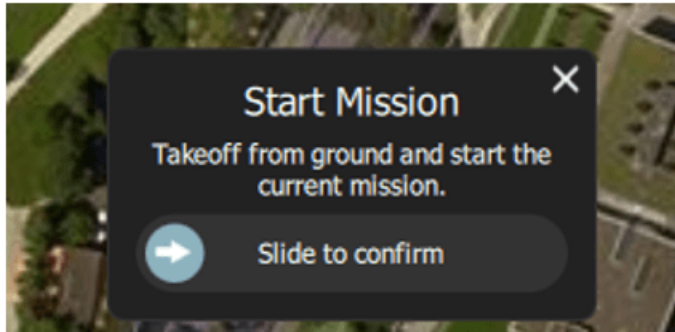
The drone takes off after the mission starts.

```
INFO  [commander] Takeoff detected
```
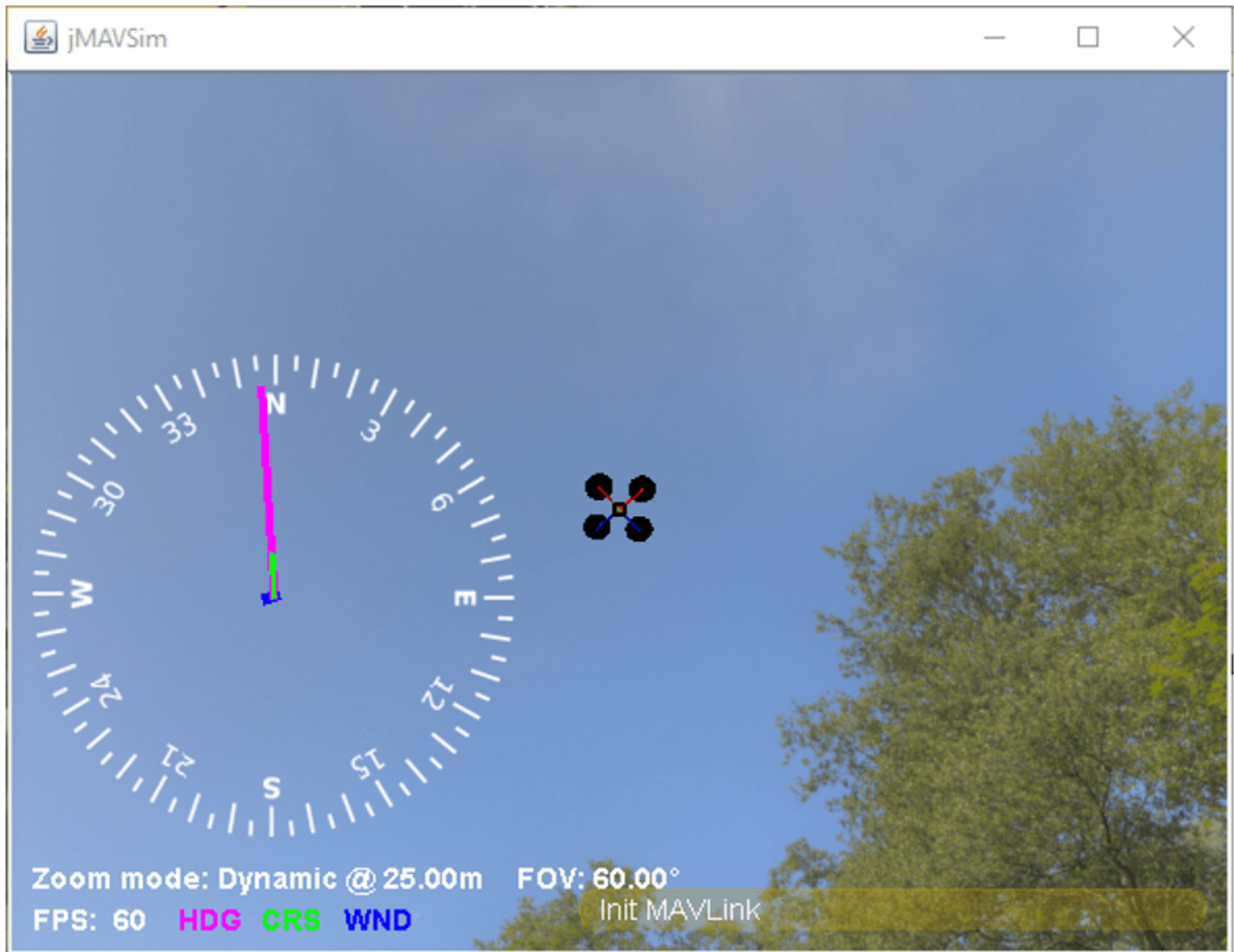
Observations After PX4 Autopilot Takes Off

In this example, the drone follows the mission. Through PX4 path planning interface, updated waypoints are sent to the onboard computer. On the onboard computer, Waypoint Follower block from UAV toolbox is used to generate trajectory from these waypoints. For more information, see Waypoint Follower. The generated trajectory is sent back to the PX4 SITL as a series of waypoints, one at a time.

Once the mission is complete, you can compare the planned path (yellow) and actual path (red) followed by the UAV.
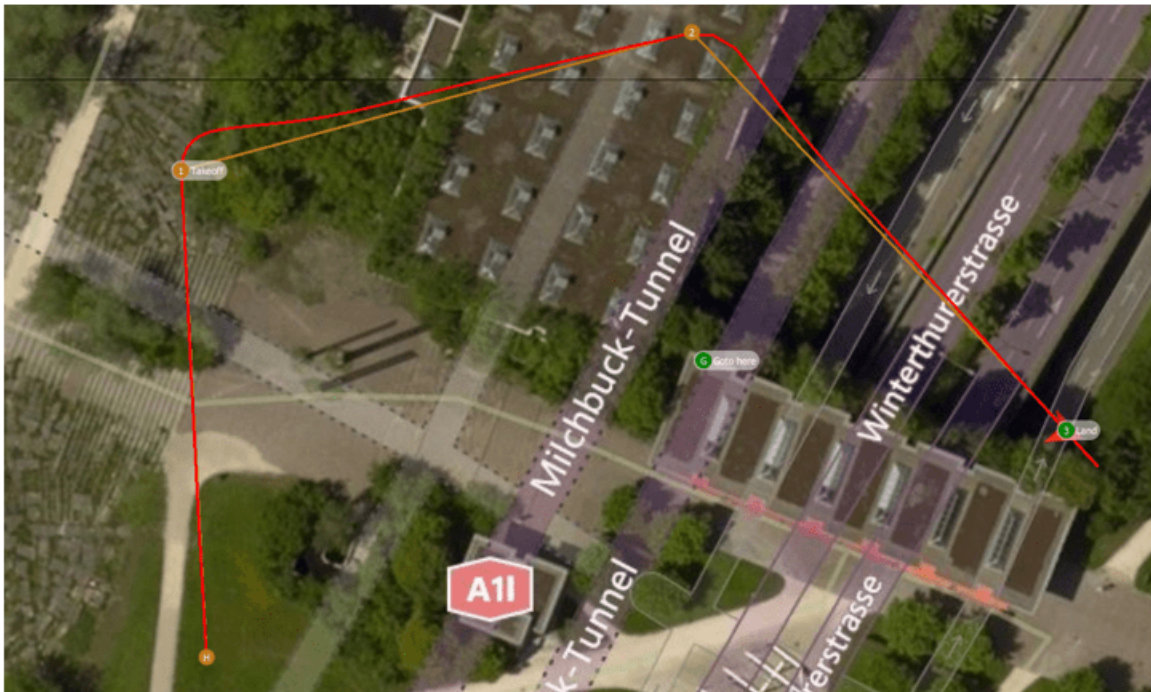
Troubleshooting

Description: The PX4 SITL is frozen a few minutes after the launch in Windows. QGC displays the error message "Communication Lost". This issue occurs when PX4 SITL is frozen and connectivity with QGC is lost a few minutes after launch.
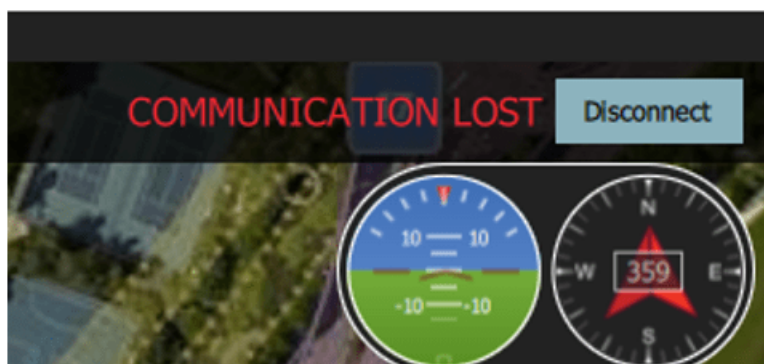
Action

- Shut down PX4 SITL by entering shutdown command in PX4 SITL window.

- Restart PX4 SITL using the `make px4_sitl jmavsim` command.

- Continue with starting the mission once GPS fusion is achieved.

Description: "Obstacle Avoidance system failed, loitering" warning when starting a mission.

This warning occurs because the communication between NVIDIA Jetson and PX4 SITL is not established.

Action: Ensure that host PC and NVIDIA Jetson are connected to same network. Try pinging NVIDIA jetson from the host PC and vice versa.

Description: "Connection to mission computer lost" warning during a mission. This warning occurs because the onboard computer is not accessible from PX4 SITL.

Action: Ensure that the monitor and tune simulation is running. If the simulation is stopped, run the model in external mode again.

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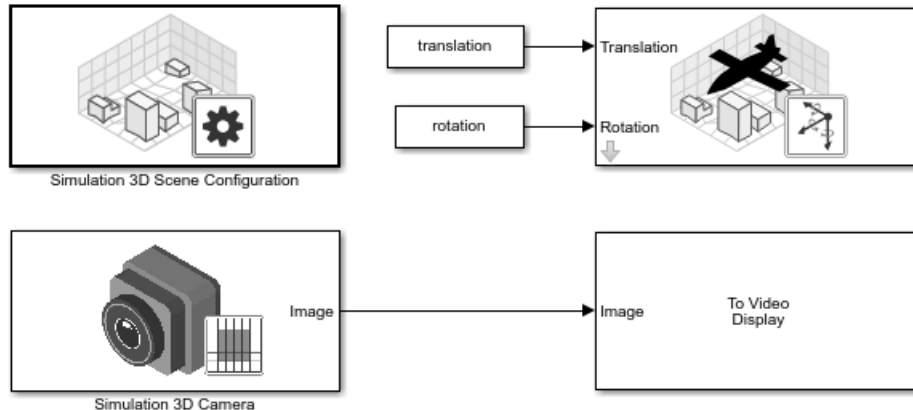
Simulink blocks related to the simulation environment can be found in the **UAV Toolbox** > **Simulation 3D** block library. These blocks provide the ability to:

- 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 2-27.

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. This visualization engine comes installed with the toolbox. When simulating in this environment, keep these requirements and limitations in mind.

Software Requirements

- Windows® 64-bit platform
- Visual Studio®
- Microsoft® DirectX® — If this software is not already installed on your machine and you try to simulate in the environment, the toolbox prompts you to install it. Once you install the software, you must restart the simulation.

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

MATLAB Release	Unreal Engine Version	Visual Studio Version
R2020b–R2021a	4.23	2019
R2021b	4.25	2019

Note Mac and Linux® platforms are not supported for Unreal Engine simulation.

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 Unreal Engine instances in the same MATLAB session
- Parallel simulations
- 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

More About

- "Scenario Simulation"

External Websites

- Unreal Engine 4 Documentation

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.
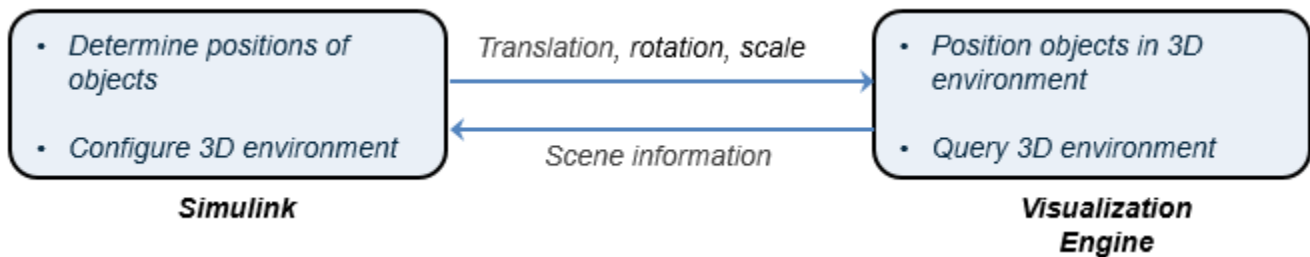
Block Execution Order

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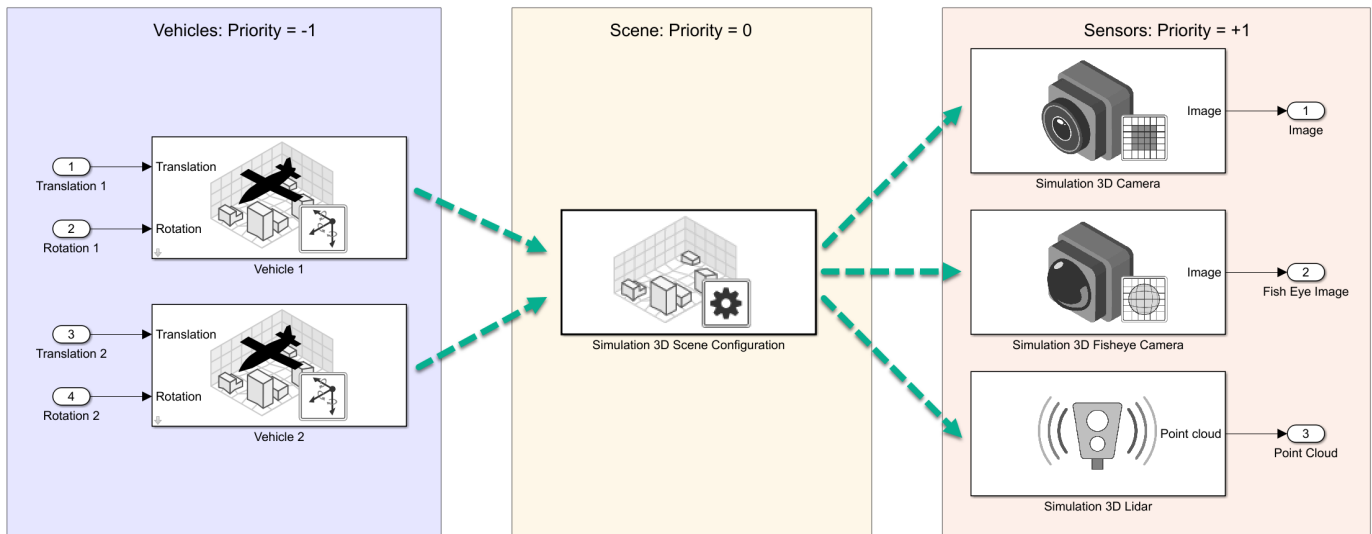
If your sensors are not detecting vehicles in the scene, it is possible that the Unreal Engine simulation blocks are executing out of order. Try updating the execution order and simulating again. For more details on execution order, see "Control and Display Execution Order" (Simulink).

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 *right-hand rule* 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 *right-handed* (RH) *Cartesian* coordinate systems: • Earth-fixed (inertial) • Vehicle	"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 *left-hand rule* 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.
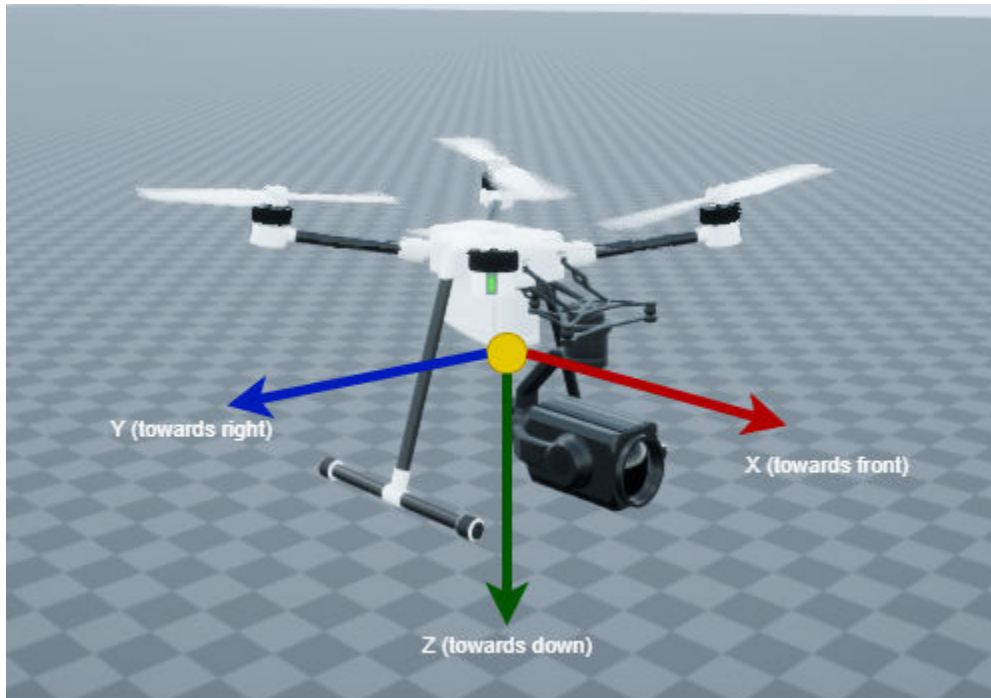
Rotational Degrees of Freedom

Rotations are defined by the Euler angles *P*, *Q*, *R* or Φ, Θ, Ψ. They are

- *P* or Φ: Roll about the *x*-axis
- *Q* or Θ: Pitch about the *y*-axis
- *R* or Ψ: Yaw about the *z*-axis
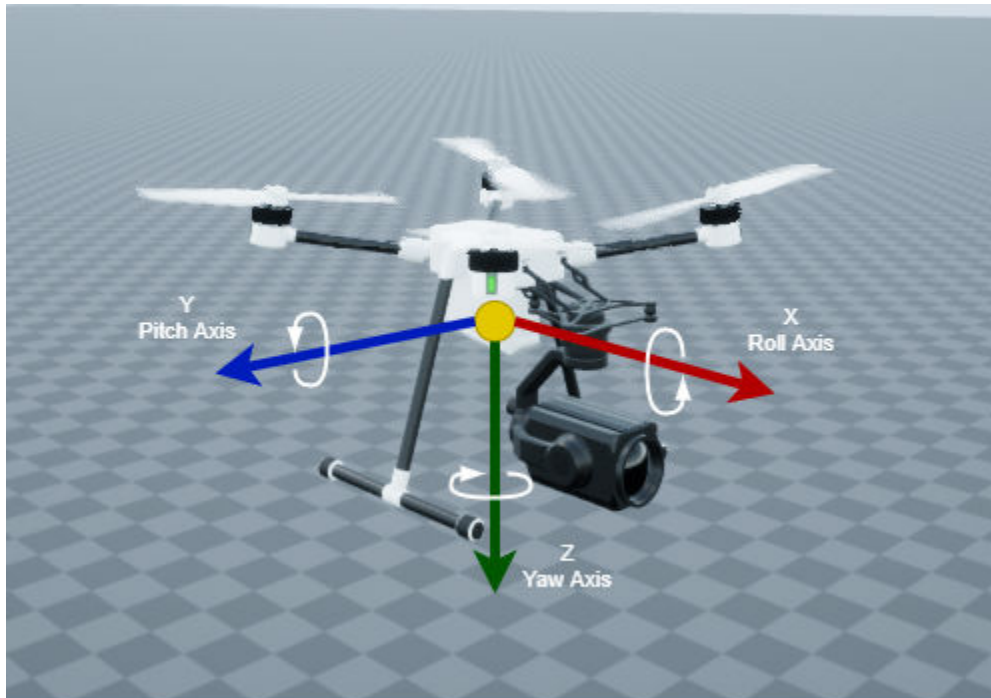
Unless otherwise specified, by default the software uses ZYX rotation order for Euler angles.

Unreal Engine World Coordinate System

The Unreal Engine environment uses a *left-hand rule* world coordinate system with axes that are fixed in the inertial reference frame.
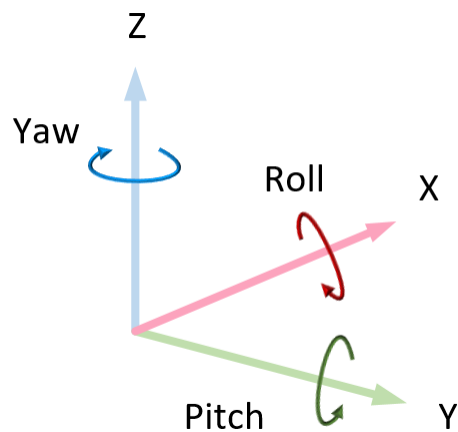
Axis	Description
X	Forward direction of the vehicle Roll — Right-handed rotation about X-axis

Axis	Description
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
Quadrotor | **Fixed Wing Aircraft**

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	• Camera with lens that is based on the ideal pinhole camera. See "What Is Camera Calibration?" (Computer Vision Toolbox) • Includes parameters for image size, focal length, distortion, and skew • Includes options to output ground truth for depth estimation and semantic segmentation	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 2-18
		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 2-18
		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 2-18
Simulation 3D Fisheye Camera	• Fisheye camera that can be described using the Scaramuzza camera model. See "Fisheye Calibration Basics" (Computer Vision Toolbox) • Includes parameters for distortion center, image size, and mapping coefficients	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	• Scanning lidar sensor model • Includes parameters for detection range, resolution, and fields of view	Display point cloud data by using `pcplayer` within a MATLAB Function block. Sample visualization:	"UAV Package Delivery" on page 1-65

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.
- UAV – 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")
```
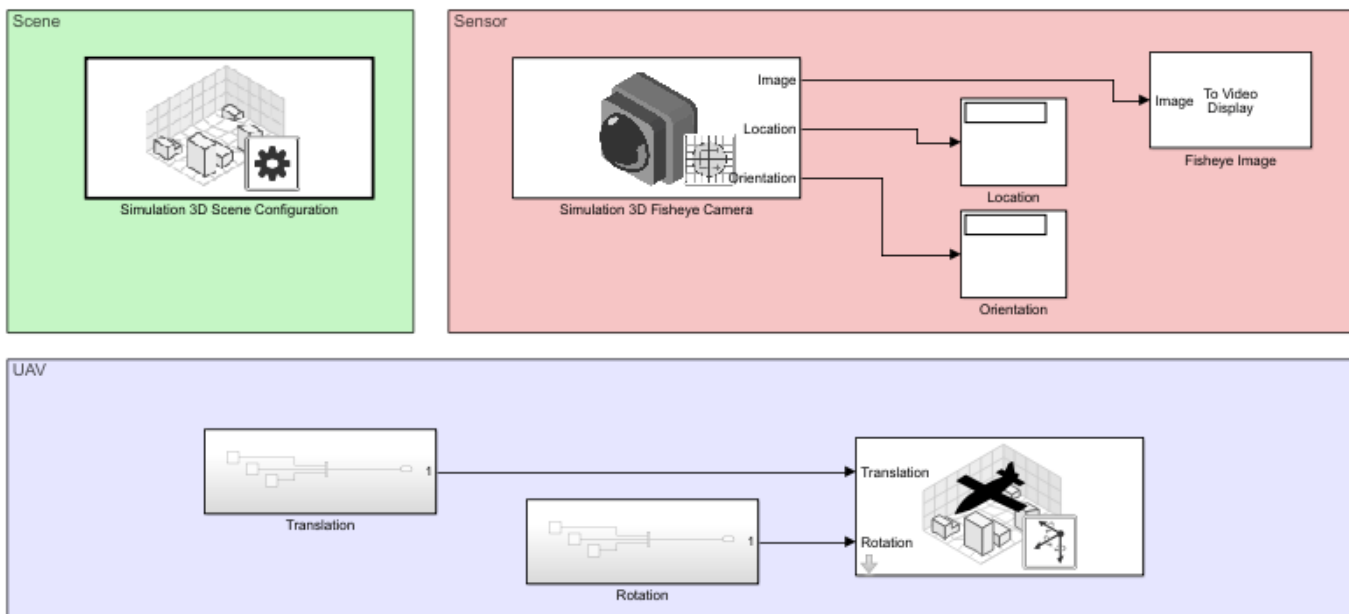

Inspect Scene

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')
```
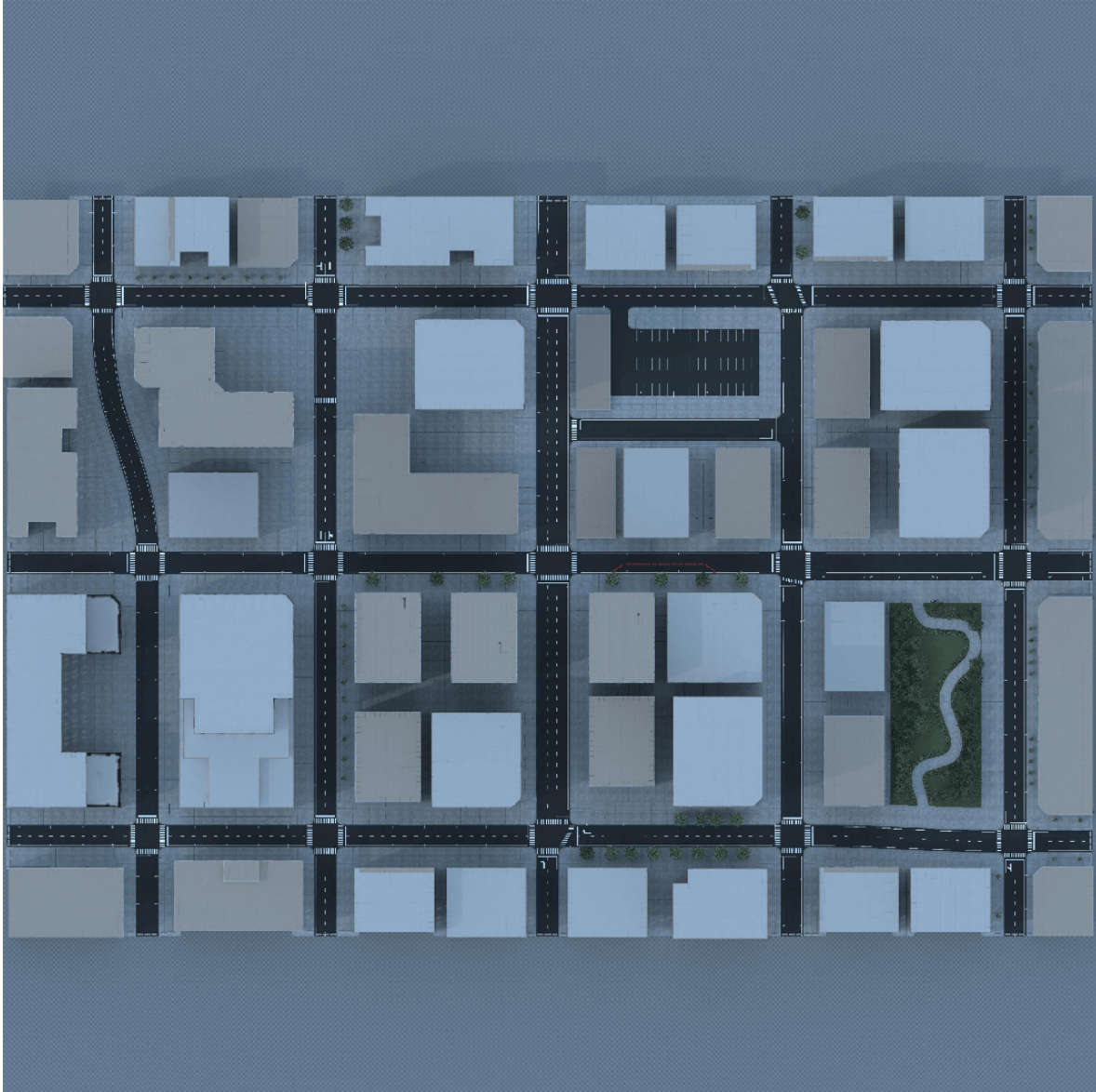

The Scene view parameter of this block determines the view from which the Unreal Engine window displays the scene. In this block, Scene view is set to the root of the scene (the scene origin), select root. You can also change the scene view to the quadrotor UAV.

Inspect Vehicle

The Simulation 3D UAV Vehicle block models the quadcopter, named `Quadrotor1`, in the scenario. During simulation, the quadrotor flys one complete circle with a radius of 5m and elevation of 1.5m around the center of the scene. The yaw angle of the quadrotor viewpoint oscillates from left to right

in the direction of travel. To create more realistic trajectories, you can obtain waypoints from a scene and specify these waypoints as inputs to the Simulation 3D UAV Vehicle block.

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 forward along the X-axis of the center of the ego vehicle by 0.1 meters.

- The **Parameters** tab contains the intrinsic camera parameters of a fisheye camera. These parameters are set to their default values except the mapping coefficient, where the second coefficient is set to -0.0005 to model lens distortion.

- 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. This image shows a sample snapshot from the Video Viewer block stream.
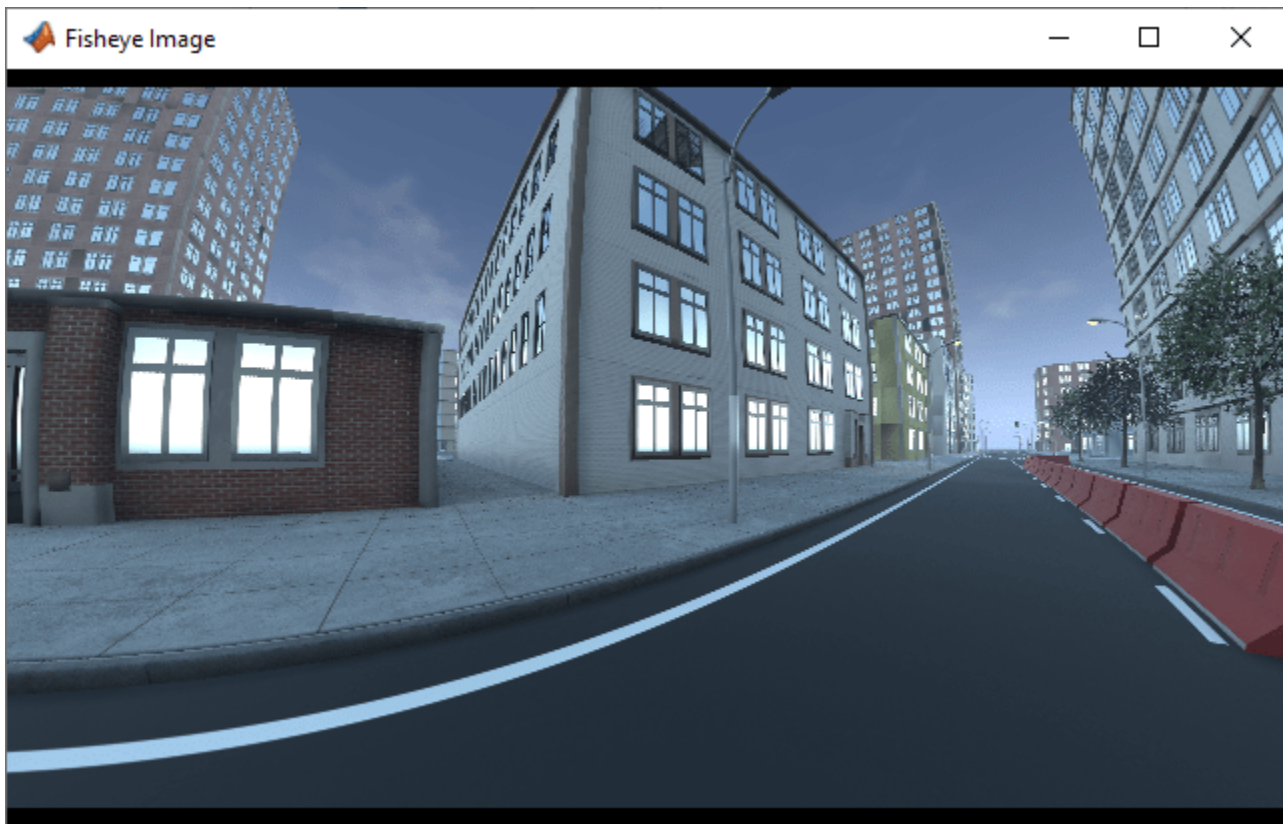
Simulate Model

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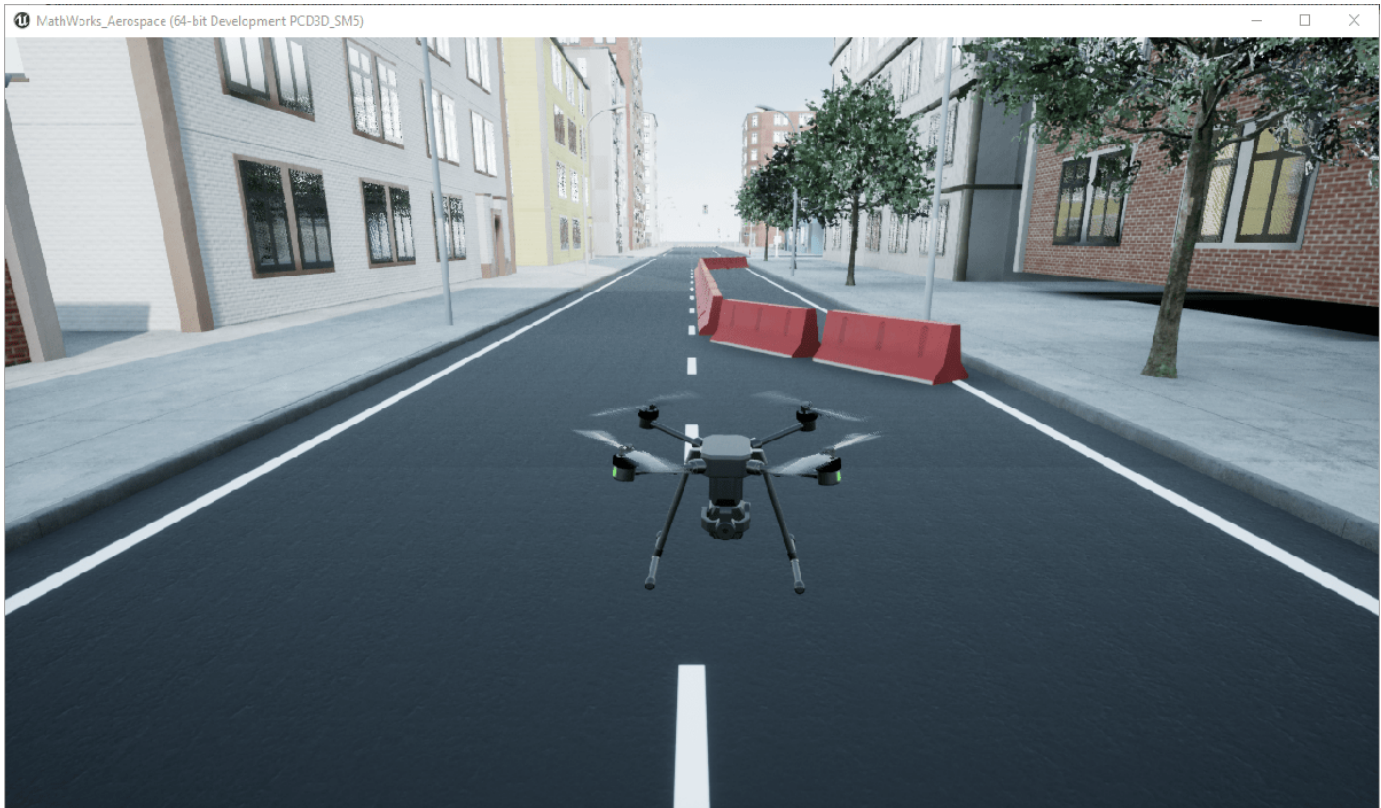
To change the view of the scene during simulation, use the numbers 1–9 on the numeric keypad. For a bird's-eye view of the scene, press 0.

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 Scene Configuration | Simulation 3D Camera | 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.
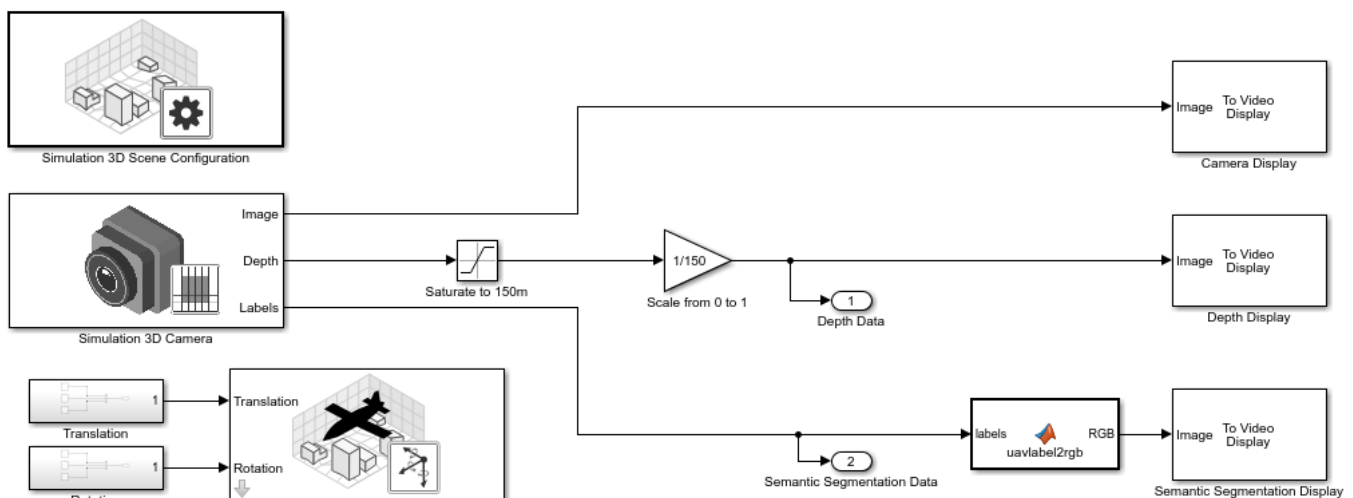
Depth Visualization

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` (Image Processing Toolbox) 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.
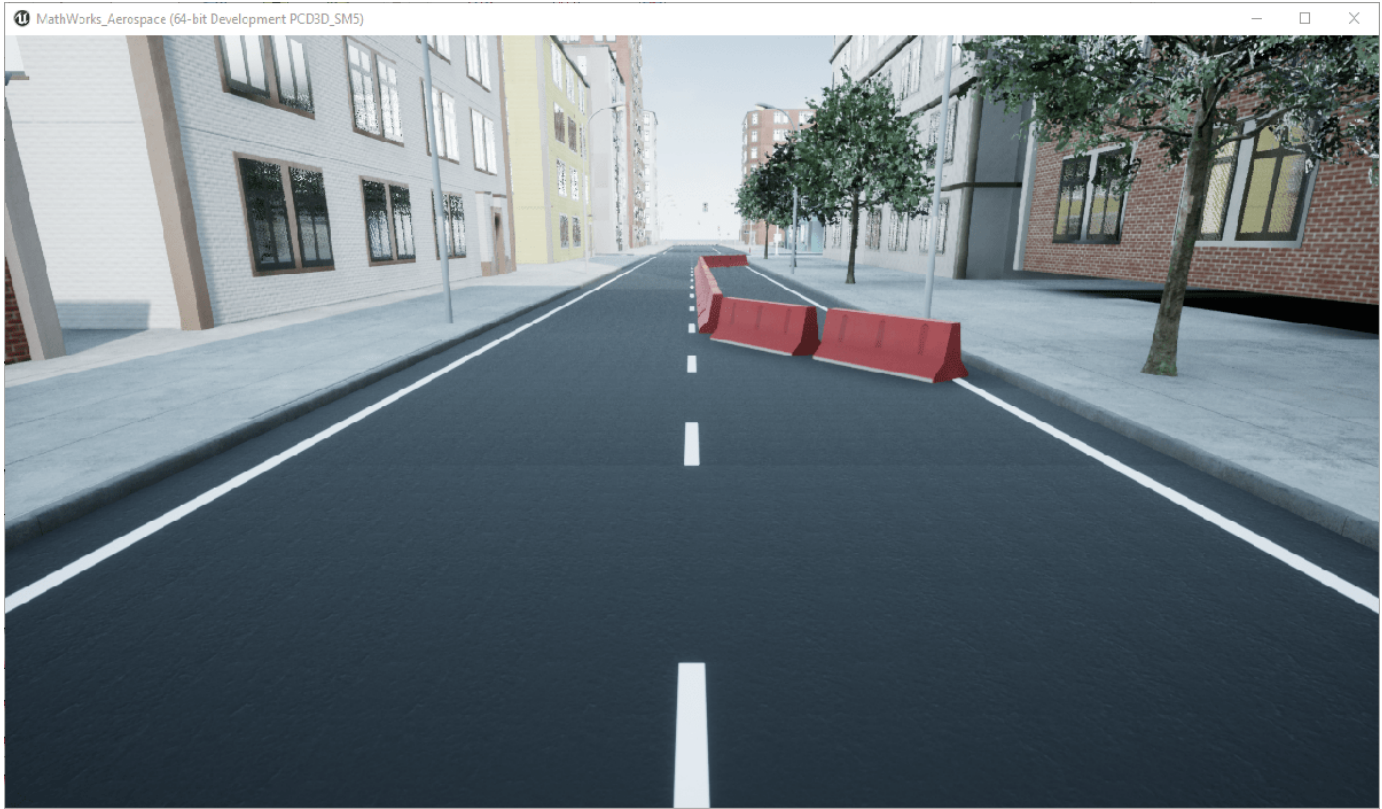
The Camera Display, Depth Display, and Semantic Segmentation Display blocks display the outputs from the camera sensor.
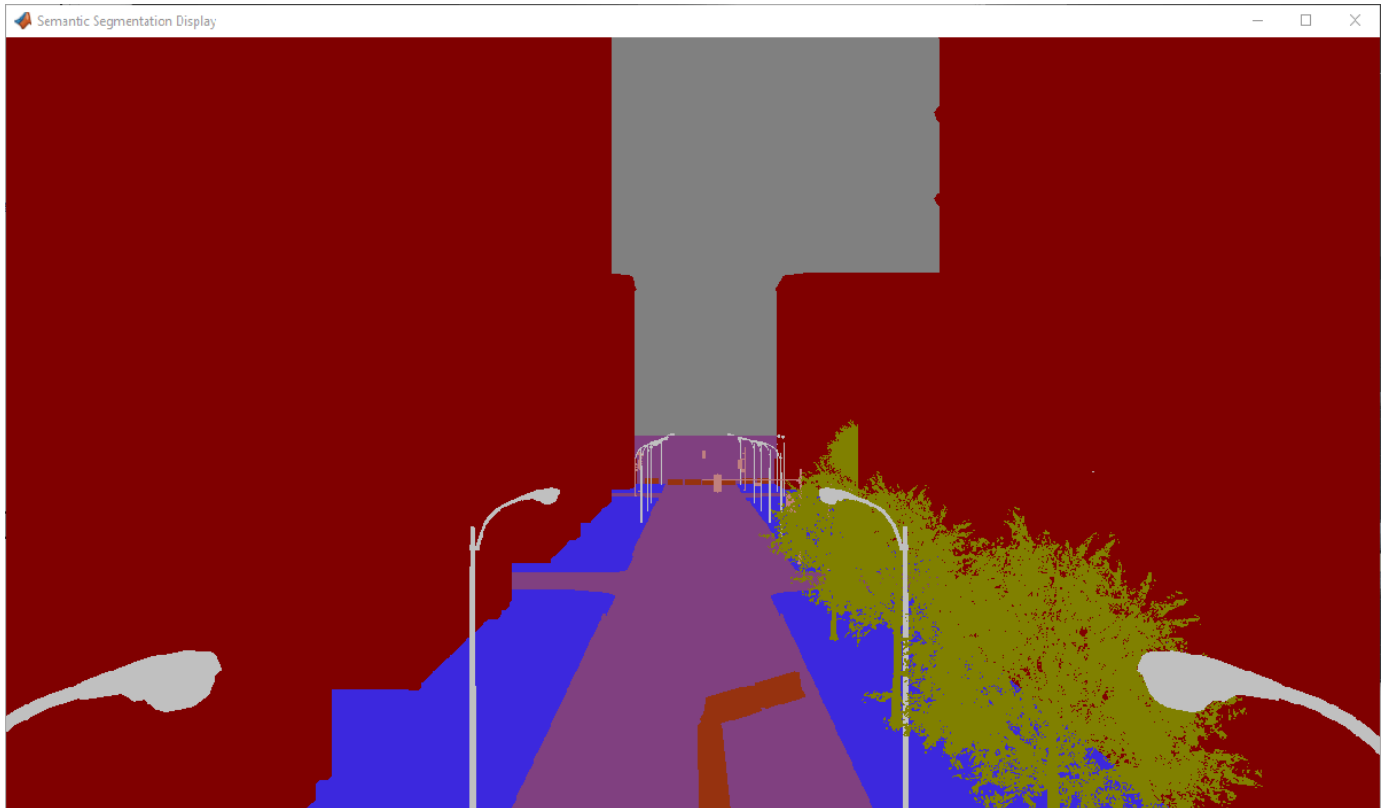
To change the visualization range of the output depth data, try updating the values in the Saturation and Gain blocks.

To change the semantic segmentation colors, try modifying the color values defined in the `sim3dColormap` function. Alternatively, in the uav`label2rgb` MATLAB Function block, try replacing the input colormap with your own colormap or a predefined colormap. See `colormap`.

See Also

Simulation 3D Scene Configuration | Simulation 3D Camera | Simulation 3D UAV Vehicle

Stream Camera, Depth and Semantic Segmentation Data from Unreal Engine to NVIDIA Jetson

This example shows how to stream simulated camera, depth, and semantic segmentation label data from an Unreal Engine® scene to NVIDIA® Jetson hardware using the **Video Send** block in Simulink®. It then shows how to visualize incoming data streams on a monitor connected to the Jetson platform, by deploying separate models for each incoming data stream. The deployed models contain the **Network Video Receive** and **SDL Video Display** blocks from the MATLAB® Coder™ Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms.

Send Data Streams from Simulink

Open the model `StreamFromUnrealToJetsonExampleModel`.

```
open_system("StreamFromUnrealToJetsonExampleModel.slx")
```

This model simulates a US City Block scene in Unreal Engine, in which a UAV follows a trajectory defined by the position and orientation values defined in the model workspace. The UAV has an onboard camera. The camera is modeled by Simulation 3D Camera block that outputs the RGB image, depth map and semantic segmentation map of labels that correspond to the objects in the scene. The frame rate of the camera is configured to be 30 frames per second (fps) as defined by the `Sample time` parameter in the Simulation 3D Scene Configuration block. The three **Video Send** blocks stream corresponding data to the Jetson hardware. Change the IP address and port number values on these blocks to values that correspond to your Jetson hardware and network setup. Note that the `Max frame time (ms)` parameter in all three blocks is configured to be 34 milliseconds based on the input frame rate of 30 fps. All three blocks are configured to stream to different ports on the same remote IP address.
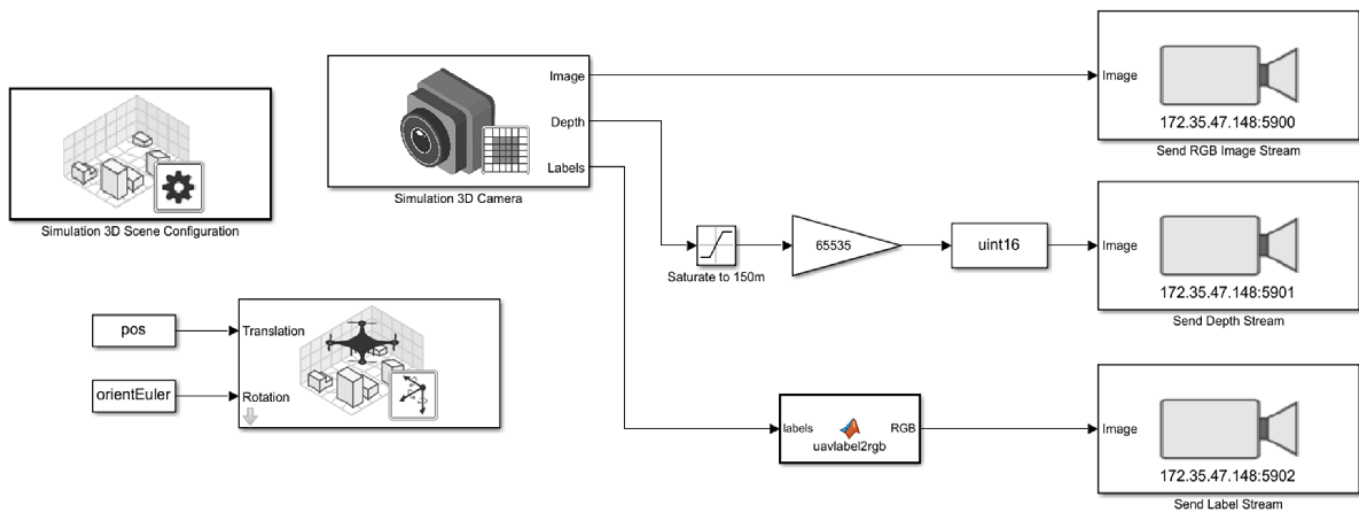
Copyright 2021 The MathWorks, Inc.

Inspect the block parameter configuration for the three data streams. The depth map is converted to `uint16` data type using lossy rescaling and is configured to stream as 16-bit grayscale image with `VP9` compression format. Additionally, note that the `segLabels2RGBImage` MATLAB Function block converts the segmentation label map to stream as a matrix of RGB values analogous to the camera image stream, with `VP8` compression format. For more information about the colormap for label

visualization, see "Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation" on page 2-18.

Run `StreamFromUnrealToJetsonExampleModel` to start streaming data to the Jetson.

`sim(`"`StreamFromUnrealToJetsonExampleModel.slx`"`);`

Receive and Visualize Data Streams on Jetson

To visualize the data now streaming to Jetson, you must deploy three separate models individually to the Jetson hardware. This is because a Simulink model can only contain one **SDL Video Display** block and there are three separate data streams. First, open the model, `VideoStreamReceiveOnJetsonExampleModel`.

`open_system(`"`VideoStreamReceiveOnJetsonExampleModel.slx`"`)`

Note that the port number and compression parameters in the **Network Video Receive** block are same as those of the corresponding **Video Send** block streaming camera images. The sample time can be any value that is less than or equal to the sample time of the data stream. On the **Hardware** tab in the toolstrip, select **Hardware Settings**. Under **Hardware board settings > Target hardware resources**, populate the `Device Address`, `Username` and `Password` fields with values corresponding to your Jetson board. Then, under **Deploy** section of the **Hardware** tab**,** select **Build, Deploy & Start** to generate code for the model and deploy it to the Jetson. The SDL Video Display window opens up and visualizes the RGB image stream on the monitor connected to the Jetson.

To restart streaming after the simulation is complete, re-run `StreamFromUnrealToJetsonExampleModel`. Then, open and deploy the model,

`DepthStreamReceiveOnJetsonExampleModel,` to visualize the depth map stream on the monitor connected to the Jetson.

```
sim("StreamFromUnrealToJetsonExampleModel.slx");
open_system("DepthStreamReceiveOnJetsonExampleModel.slx")
```


To restart streaming after the simulation is complete, re-run `StreamFromUnrealToJetsonExampleModel`. Then, open and deploy the model, `LabelStreamReceiveOnJetsonExampleModel`, to visualize the semantic segmentation label map stream on the monitor connected to the Jetson.

```
sim("StreamFromUnrealToJetsonExampleModel.slx");
open_system("LabelStreamReceiveOnJetsonExampleModel.slx")
```
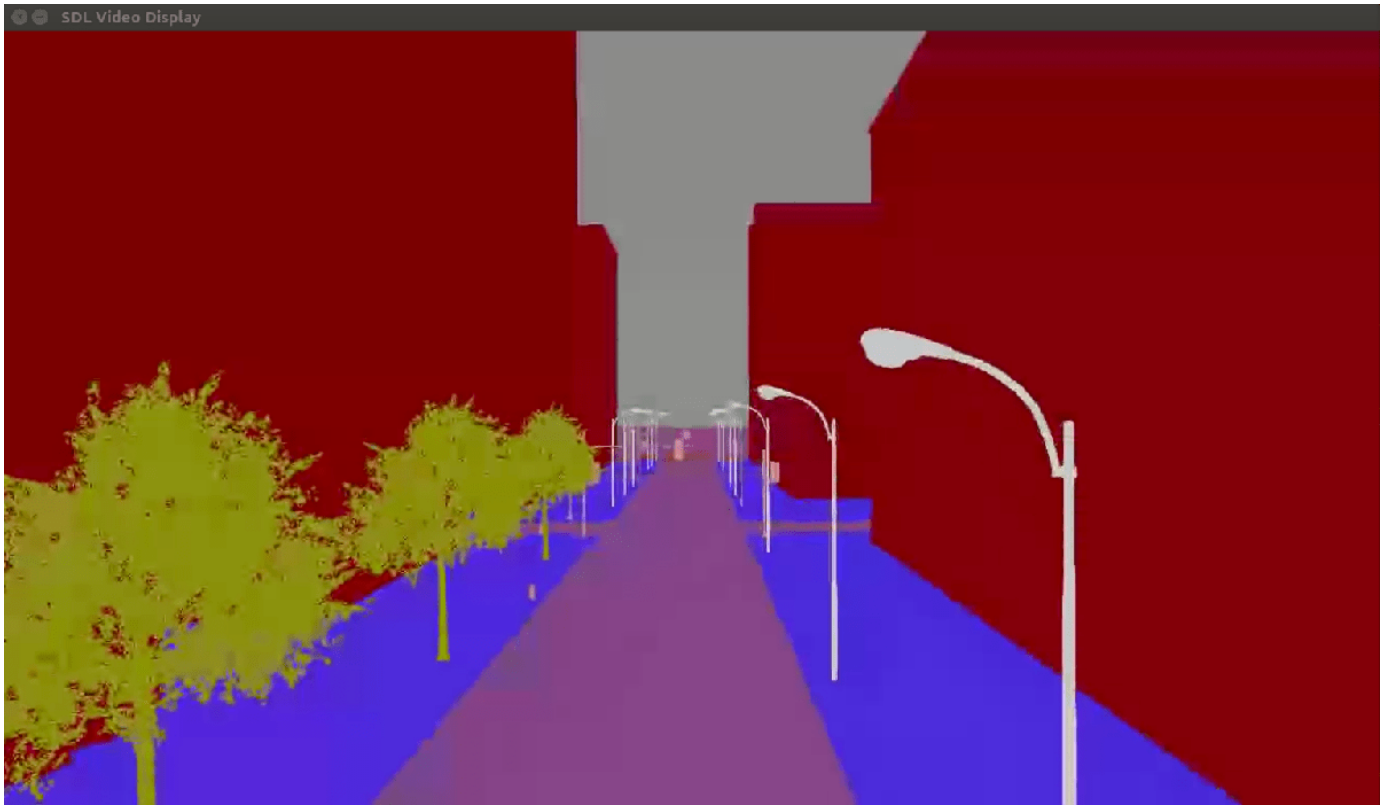
Customize Unreal Engine Scenes for UAVs

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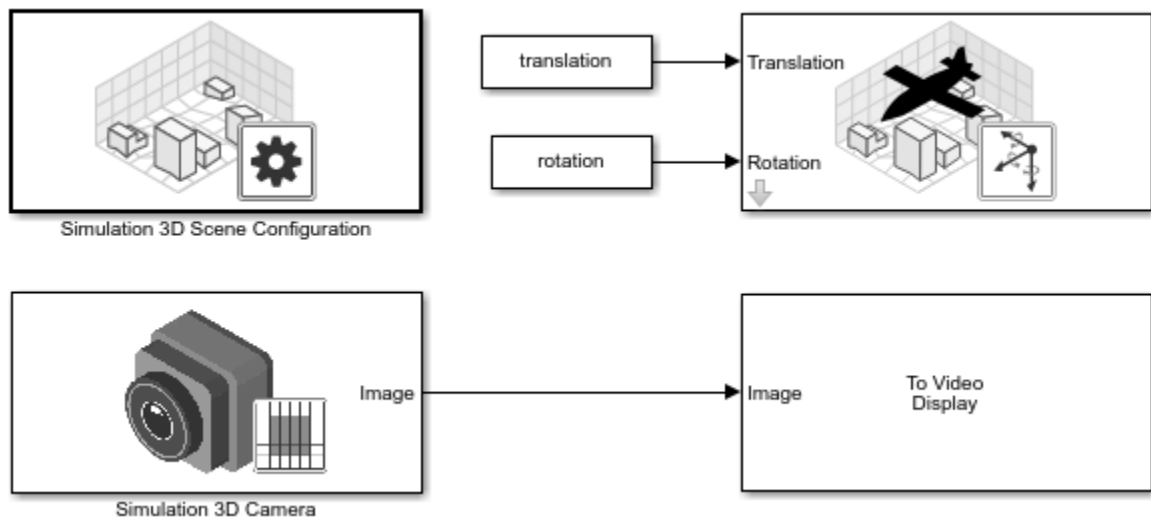
To customize Unreal Engine scenes for UAV flight simulations, follow these steps:

1   "Install Support Package for Customizing Scenes" on page 2-28
2   "Migrate Projects Developed Using Prior Support Packages" on page 2-31
3   "Customize Unreal Engine Scenes Using Simulink and Unreal Editor" on page 2-32
4   "Package Custom Scenes into Executable" on page 2-38

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.

Note These installation instructions apply to **R**2021**b**. If you are using a previous release, see the documentation for Other Releases.

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.

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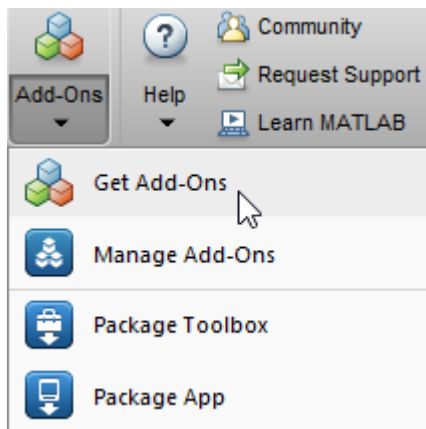
2   In the Add-On Explorer window, search for the UAV Toolbox Interface for Unreal Engine Projects support package. Click **Install**.

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 (`AutoVrtlEnv.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 plugins, `MathWorkSimulation` and `MathworksUAVContent`. These plugins establish 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 `AutoVrtlEnv` 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","shared","sim3dprojects","spkg");
localFolder = "C:\Local";
```

2   Copy the `AutoVrtlEnv` project from the support package folder to the local destination folder.

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

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

Copy Plugin to Unreal Editor

Copy the `MathWorksSimulation` and `MathworksUAVContent` plugin folders 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_4.25";
```

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

```
mwSimPluginName = "MathWorksSimulation.uplugin";
mwSimPluginFolder = fullfile(supportPackageFolder,"plugins","mw_simulation","MathWorksSimulat
mwUAVPluginName = "MathworksUAVContent.uplugin";
mwUAVPluginFolder = fullfile(supportPackageFolder,"plugins","mw_uav","MathworksUAVContent");

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,"MathworksUAVContent"));
        disp("Successfully copied MathworksUAVContent plugin to UE4 engine plugins!")
    end
```

After you install and set up the support package, you can begin customizing scenes. If you want to use a project developed using a prior release of the UAV Toolbox Interface for Unreal Engine Projects support package, you must migrate the project to make it compatible with the currently supported Unreal Editor version, see "Migrate Projects Developed Using Prior Support Packages" on page 2-31. Otherwise, see "Customize Unreal Engine Scenes Using Simulink and Unreal Editor" on page 2-32.

See Also

Migrate Projects Developed Using Prior Support Packages

After you install the UAV Toolbox Interface for Unreal Engine Projects support package as described in "Install Support Package for Customizing Scenes" on page 2-28, you may need to migrate your project. If your Simulink model uses an Unreal Engine executable or project developed using a prior release of the support package, you must migrate the project to make it compatible with Unreal Editor 4.25. Follow these steps:

1   Open Unreal Engine 4.25. For example, navigate to `C:\Program Files\Epic Games \UE_4.25\Engine\Binaries\Win64` and open `UE4Editor.exe`.

2   Use the Unreal Project Browser to open the project that you want to migrate.

3   Follow the prompts to open a copy of the project. The editor creates a new project folder in the same location as the original, appended with 4.25. Close the editor.

4   In a file explorer, remove the space in the migrated project folder name. For example, rename `MyProject 4.25` to `MyProject4.25`.

5   Use MATLAB to open the migrated project in Unreal Editor 4.25. For example, if you have a migrated project saved to the `C:/Local` folder, use this MATLAB code:

```
path = fullfile('C:','Local','MyProject4.25','MyProject.uproject');
editor = sim3d.Editor(path);
open(editor);
```

Note The support package may includes changes in the implementation of some actors. Therefore, if the original project contains actors that are placed in the scene, some of them might not fully migrate to Unreal Editor 4.25. To check, examine the Output Log.
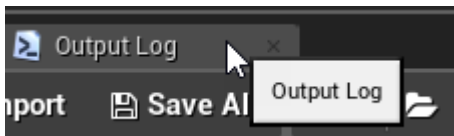
The log might contain error messages. For more information, see the Unreal Engine 4 Documentation or contact MathWorks Technical Support.

6   Optionally, after you migrate the project, you can use the project to create an Unreal Engine executable. See "Package Custom Scenes into Executable" on page 2-38.

After you migrate the project, you can create custom scenes. See "Customize Unreal Engine Scenes Using Simulink and Unreal Editor" on page 2-32.

See Also
Simulation 3D Scene Configuration

More About
- "Customize Unreal Engine Scenes for UAVs" on page 2-27

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 2-28, 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.

To use a project that you developed using a prior release of the support package, first migrate the project to be compatible with the currently supported Unreal Engine version. See "Migrate Projects Developed Using Prior Support Packages" on page 2-31.

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 1-65 example.
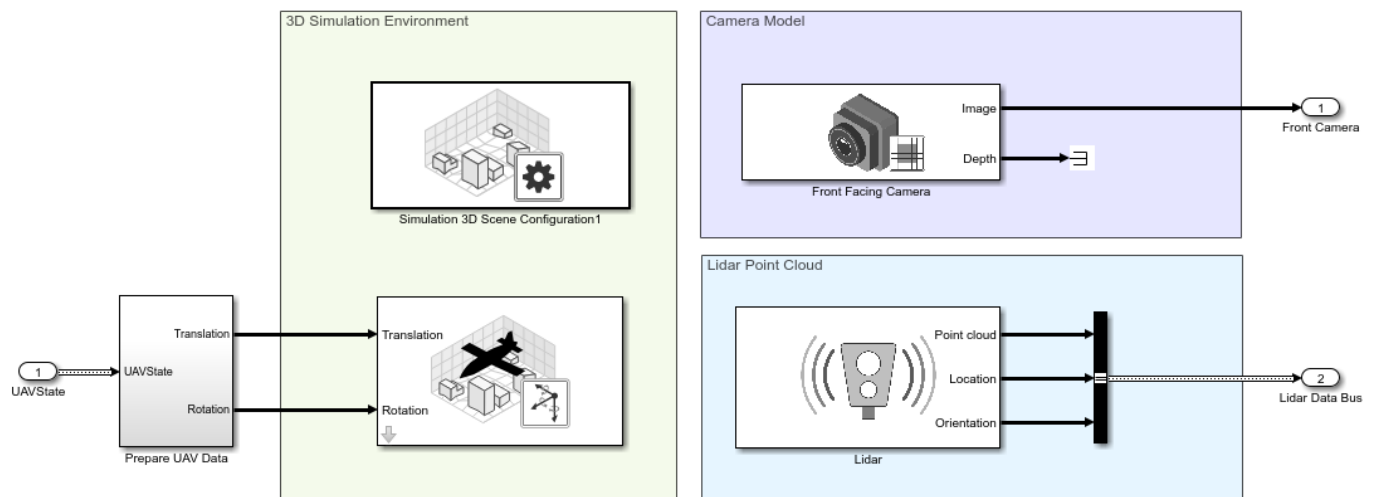
2   In the Simulation 3D Scene Configuration block of this model, set the **Scene source** parameter to `Unreal Editor`.

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 `AutoVrtlEnv` project that comes installed with the UAV Toolbox Interface for Unreal Engine Projects support package.

`C:\Local\AutoVrtlEnv\AutoVrtlEnv.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 `AutoVrtlEnv` 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** and **Mathworks UAV Content** plugins are installed. Make sure both plugins are enabled by verifying that the **Enabled** box is checked for both. These plugins are the `MathWorksSimulation.uplugin` and `MathworksUAVContent.uplugin` files that you copied into your Unreal Editor installation in "Install Support Package for Customizing Scenes" on page 2-28. After enabling the plugins, you may have to restart the Unreal Editor. Click **Restart Now** if prompted.

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. To use the lighting that comes installed with `AutoVrtlEnv`, see Use `AutoVrtlEnv` Project Lighting in Custom Scene on page 2-35.

Reparent Actor Blueprint

Note If you are using a scene from the `AutoVrtlEnv` 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 2-29 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 `AutoVrtlEnv.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 `AutoVrtlEnv` 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 `AutoVrtlEnv` 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 `AutoVrtlEnv` 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 `AutoVrtlEnv` 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 2-41.

Use AutoVrtlEnv Project Lighting in Custom Scene

To use the lighting that comes installed with the `AutoVrtlEnv` project in UAV Toolbox, follow these steps.

1   On the **World Settings** tab, clear **Force no precomputed lighting**.
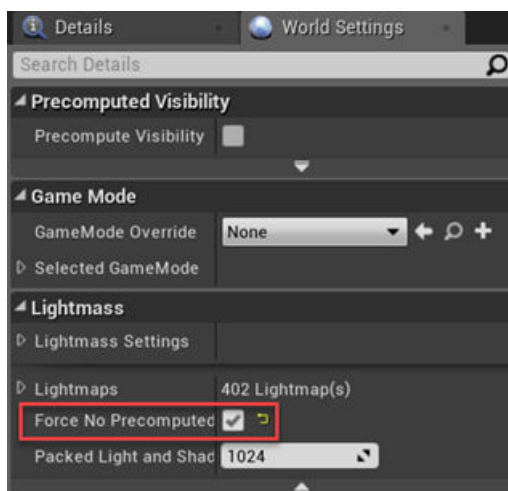
2   Under **Build**, select **Lighting Quality > Production** to rebuild the maps with production quality. Rebuilding large maps can take time.
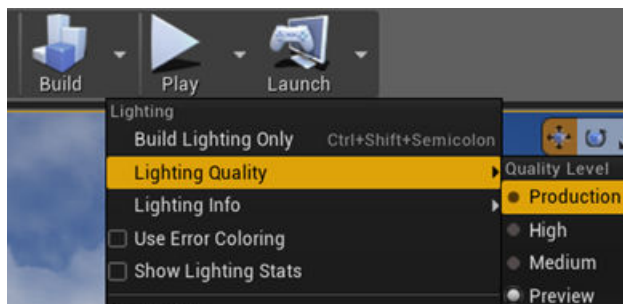
Run Simulation

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.

Key	Camera View
1	Back left
2	Back
3	Back right
4	Left
5	Internal
6	Right
7	Front left
8	Front
9	Front right
0	Overhead

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 2-38.

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 2-32.

2. Ensure the plugin content is visible in the Content Browser. Under **View Options**, check the **Show Engine Content** and **Show Plugin Content** check boxes.
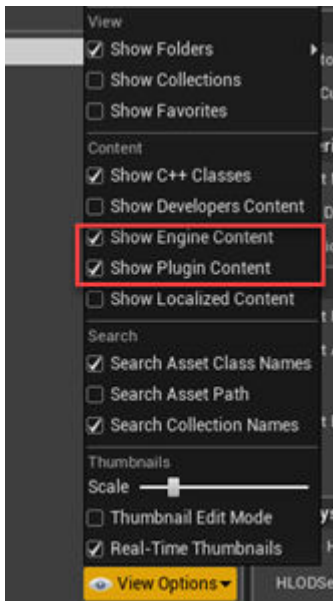
3. In the Unreal Editor toolbar, select **Settings > Project Settings** to open the Project Settings window.

4. In the left pane, in the **Project** section, click **Packaging**.

5. 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.
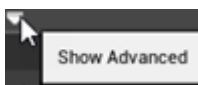
Packaging Option	Enable or Disable
Use Pak File	Enable
Cook everything in the project content directory (ignore list of maps below)	Disable
Cook only maps (this only affects cookall)	Enable
Create compressed cooked packages	Enable
Exclude editor content while cooking	Enable

6   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.

7   Specify the required asset directories to include in the executable. These directories are located in the `MathWorksSimulation` plugin.

Under **Additional Asset Directories to Cook**, click the **Adds Element** button ⊞ to add elements and specify these directories:

- `/MathWorksSimulation/Characters`
- `/MathWorksSimulation/VehiclesCommon`
- `/MathWorksSimulation/Vehicles`
- `/MathWorksSimulation/Weather`

To include the `MathworksUAVContent` plugin assets, also add that entire directory:

- `/MathWorksUAVContent`

8   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**.

9   (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 2-41.

10   Close the **Project Settings** window.

11   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 `AutoVrtlEnv` 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\shared\sim3dprojects\spkg\AutoVrtlE`

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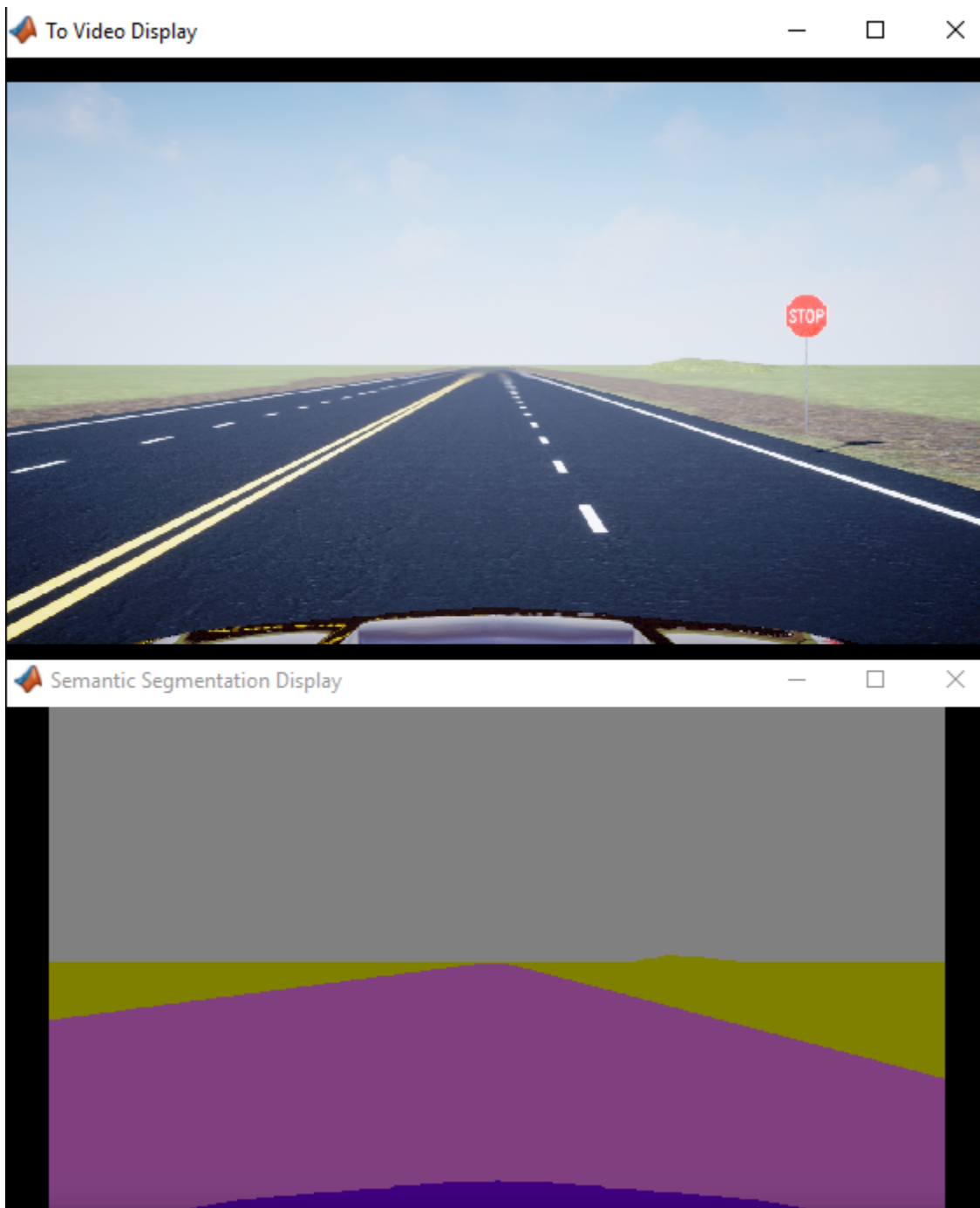
To apply a stencil ID label to a scene element, follow these steps:

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 2-32.

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 [Show Advanced] 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	*Not used*
3	Other
4	*Not used*
5	Pole
6	*Not used*
7	Road
8	Sidewalk
9	Vegetation
10	Vehicle
11	*Not used*
12	Generic traffic sign
13	Stop sign
14	Yield sign
15	Speed limit sign
16	Weight limit sign
17-18	*Not used*
19	Left and right arrow warning sign
20	Left chevron warning sign
21	Right chevron warning sign
22	*Not used*
23	Right one-way sign
24	*Not used*

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

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 [🔷 Lit] 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 2-18.

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.
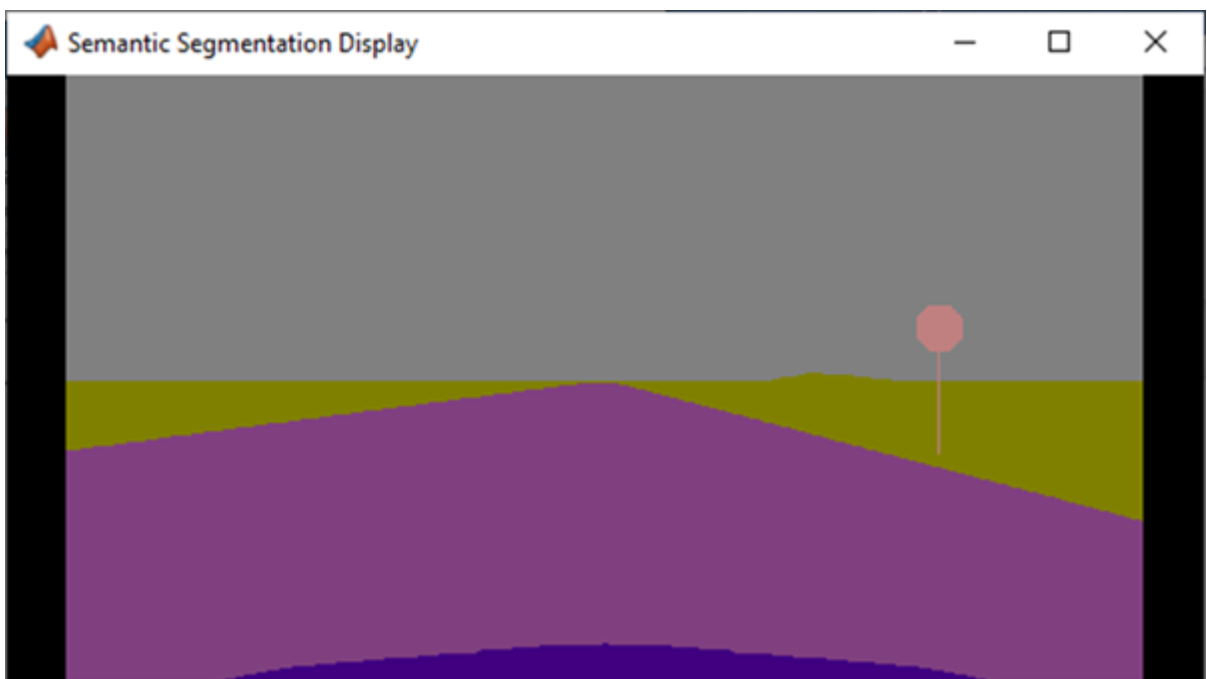
See Also

Simulation 3D Scene Configuration | Simulation 3D Camera | Simulation 3D UAV Vehicle

More About

- "Depth and Semantic Segmentation Visualization Using Unreal Engine Simulation" on page 2-18
- "Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)

Stereo Visual SLAM for UAV Navigation in 3D Simulation

Visual SLAM is the process of calculating the position and orientation of a camera with respect to its surroundings while simultaneously mapping the environment. Developing a visual SLAM algorithm and evaluating its performance in varying conditions is a challenging task. One of the biggest challenges is generating the ground truth of the camera sensor, especially in outdoor environments. The use of simulation enables testing under a variety of scenarios and camera configurations while providing precise ground truth.

This example demonstrates the use of Unreal Engine® simulation to develop a visual SLAM algorithm for a UAV equipped with a stereo camera in a city block scenario. For more information about the implementation of the visual SLAM pipeline for a stereo camera [1] on page 2-0 , see the "Stereo Visual Simultaneous Localization and Mapping" (Computer Vision Toolbox) example.

Set Up Simulation Environment

First, set up a scenario in the simulation environment that can be used to test the visual SLAM algorithm. Use a scene depicting a typical city block with a UAV as the vehicle under test.

Next, select a trajectory for the UAV to follow in the scene. You can follow the "Select Waypoints for Unreal Engine Simulation" (Automated Driving Toolbox) example to interactively select a sequence of waypoints and then use the `helperSelectSceneWaypoints` function to generate a reference trajectory for the UAV. This example uses a recorded reference trajectory as shown below:

```
% Load reference path
data       = load('uavStereoSLAMData.mat');

pos        = data.pos;         % Position
orientEuler = data.orientEuler;  % Orientation
```
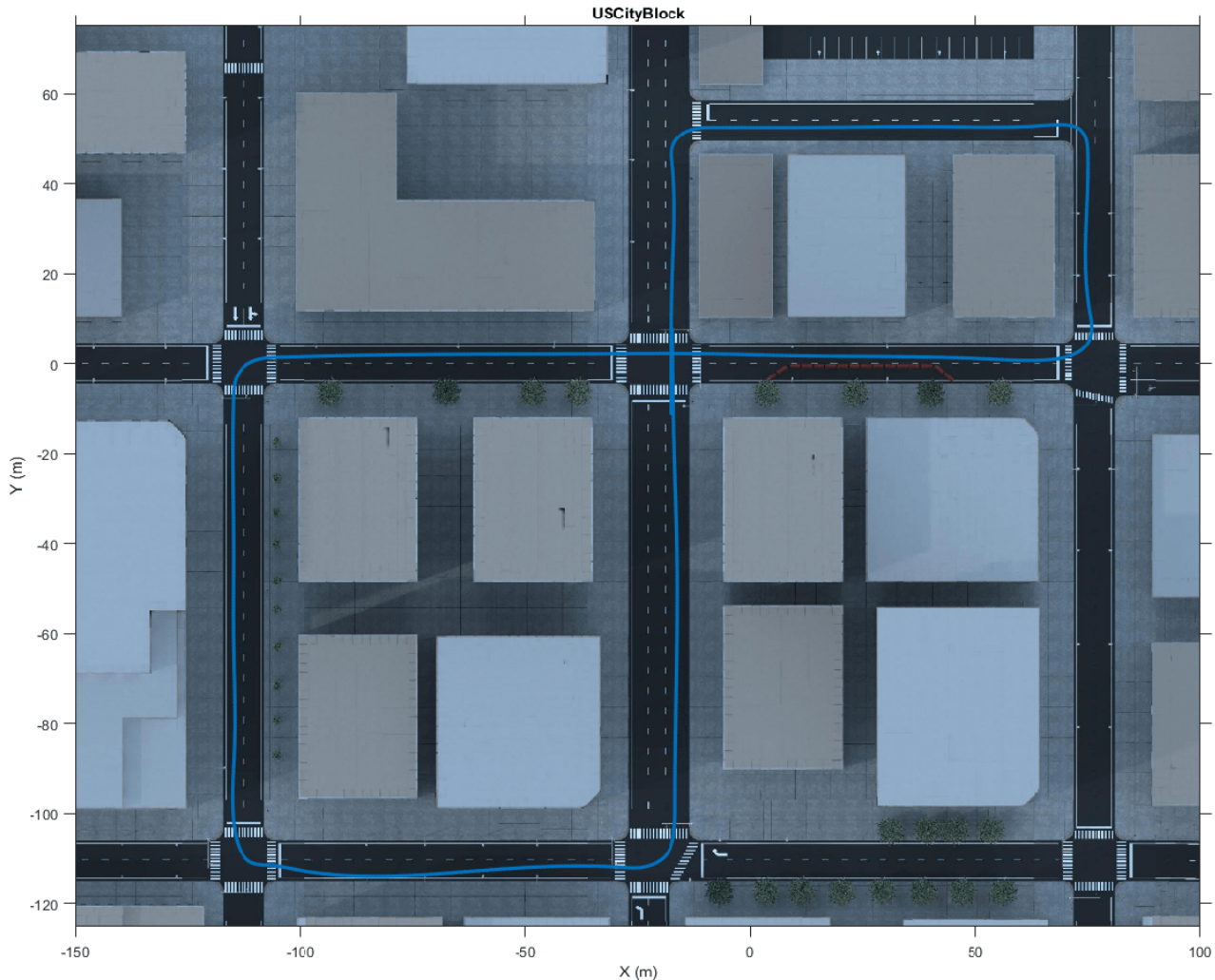
The UAVVisual`SLAMIn3DSimulation` Simulink® model is configured with the US City Block scene using the Simulation 3D Scene Configuration block. The model places a UAV on the scene using the Simulation 3D UAV Vehicle block. A stereo camera consisting of two Simulation 3D Camera blocks is attached to the UAV. In the dialog box of the Simulation 3D Camera block, use the **Mounting** tab to adjust the placement of the camera. Use the **Parameters** tab to configure properties of the camera to simulate different cameras. To estimate the intrinsics of the stereo camera that you want to simulate, use the "Using the Stereo Camera Calibrator App" (Computer Vision Toolbox) app.

```
% Stereo camera parameters
focalLength    = [1109, 1109]; % In pixels
principalPoint = [640, 360];   % In pixels [x, y]
```

```
imageSize       = [720, 1280];  % In pixels [mrows, ncols]
baseline        = 0.5;          % In meters

% Open the model
modelName = 'UAVVisualSLAMIn3DSimulation';
open_system(modelName);
```
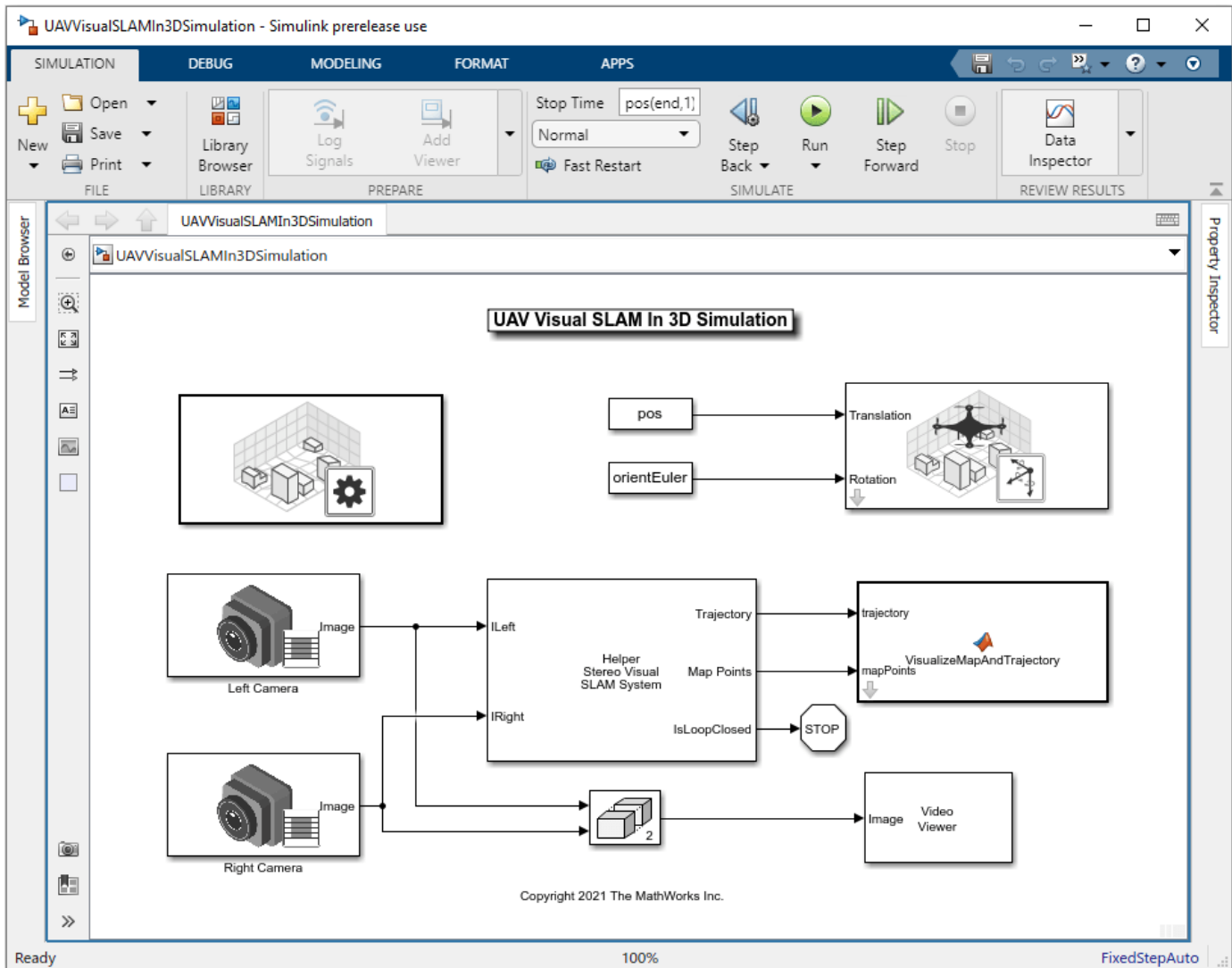

Implement the Stereo Visual SLAM Algorithm

The Helper Stereo Visual SLAM System block implements the stereo visual SLAM pipeline, consisting of the following steps:

- **Map Initialization**: The pipeline starts by initializing the map of 3-D points from a pair of images generated from the stereo camera using the disparity map. The left image is stored as the first key frame.

- **Tracking**: Once a map is initialized, for each new stereo pair, the pose of the camera is estimated by matching features in the left image to features in the last key frame. The estimated camera pose is refined by tracking the local map.

2-49

- **Local Mapping**: If the current left image is identified as a key frame, new 3-D map points are computed from the disparity of the stereo pair. At this stage, bundle adjustment is used to minimize reprojection errors by adjusting the camera pose and 3-D points.
- **Loop Closure**: Loops are detected for each key frame by comparing it against all previous key frames using the bag-of-features approach. Once a loop closure is detected, the pose graph is optimized to refine the camera poses of all the key frames.

For the implementation details of the algorithm, see the "Stereo Visual Simultaneous Localization and Mapping" (Computer Vision Toolbox) example.

Run Stereo Visual SLAM Simulation

Simulate the model and visualize the results. The **Video Viewer** block displays the stereo image output. The **Point Cloud Player** displays the reconstructed 3-D map with the estimated camera trajectory.

```
if ~ispc
    error("Unreal Engine Simulation is supported only on Microsoft" + char(174) + " Windows" + ch
end

% Run simulation
sim(modelName);
```
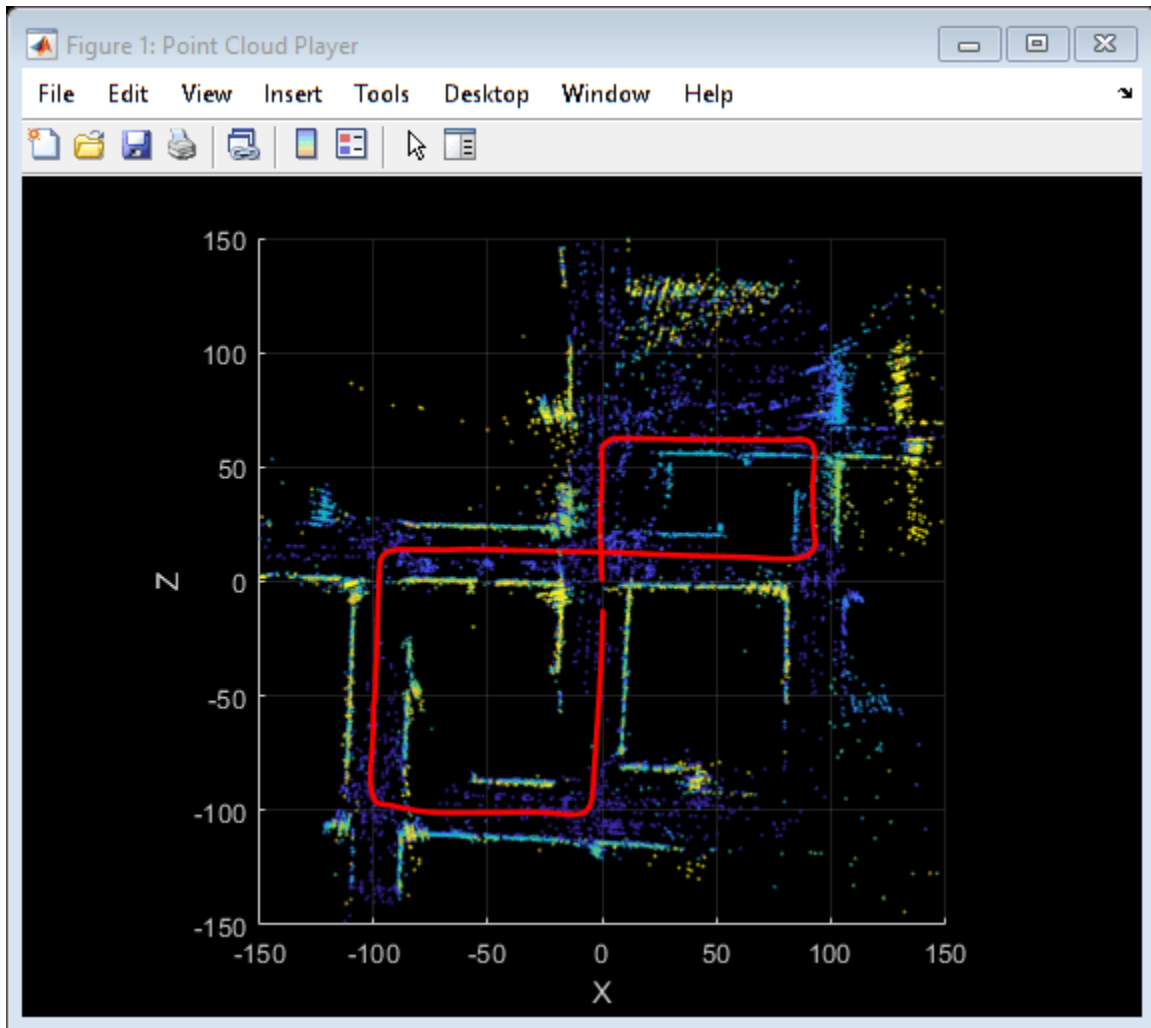
```
Loop edge added between keyframe: 5 and 356
Loop edge added between keyframe: 3 and 356
Loop edge added between keyframe: 4 and 358
Loop edge added between keyframe: 5 and 358
Loop edge added between keyframe: 6 and 358
```

Close the model.

```
close_system(modelName);
```

References

[1] Mur-Artal, Raul, and Juan D. Tardós. "ORB-SLAM2: An open-source SLAM system for monocular, stereo, and RGB-D cameras." *IEEE Transactions on Robotics* 33, no. 5 (2017): 1255-1262.

Prepare Custom UAV Vehicle Mesh for the Unreal Editor

This example shows you how to create a vehicle mesh that is compatible with the project in the UAV Toolbox Interface for Unreal Engine Projects support package. You can specify the mesh in the Simulation 3D UAV Vehicle block to visualize the vehicle in the Unreal Editor when you run a simulation.

Before you start, install the UAV Toolbox Interface for Unreal Engine Projects support package. See "Install Support Package for Customizing Scenes" on page 2-28.

To create a compatible custom vehicle mesh, follow these workflow steps.

Step	Description
"Set Up Bone Hierarchy" on page 2-53	In a 3D creation environment, set up the vehicle mesh bone hierarchy and specify part names.
"Assign Materials" on page 2-55	Optionally, assign materials to the vehicle parts.
"Export Mesh and Armature" on page 2-59	Export the vehicle mesh and armature in the .fbx file format.
"Import Mesh to Unreal Editor" on page 2-61	Import the vehicle mesh into the Unreal Editor.
"Set Block Parameters" on page 2-62	Set up the Simulation 3D UAV Vehicle block parameters.

Note To create the mesh, this example uses the 3D creation software Blender® Version 2.80.

Set Up Bone Hierarchy

1 Import a vehicle mesh into a 3D modeling tool, such as Blender.
2 To ensure that this mesh is compatible with the animation components in the UAV Toolbox Interface for Unreal Engine Projects support package, use this naming convention for the vehicle parts in the mesh.

Vehicle Part	Name
UAV body	UAV_Body
UAV motor	UAV_Motor$n$
UAV rotor blade	UAV_Rotor$n$
Camera pivot point	UAV_CameraPivot
Camera arm	UAV_CameraArm

3 Set the UAV body object, UAV_Body, as the parent of the other UAV objects.
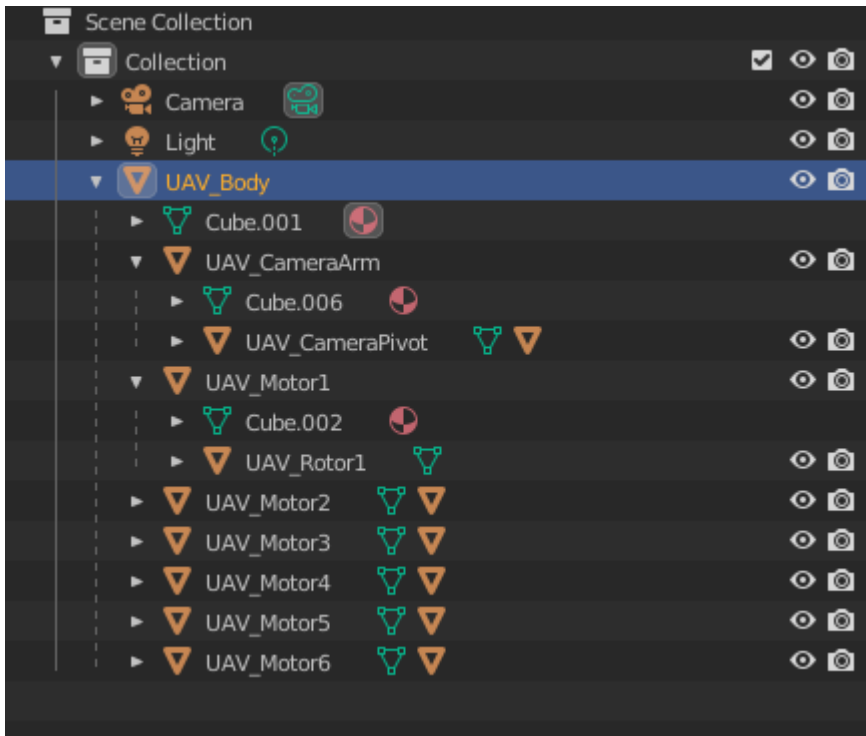
4 Set each `UAV_Motor`*n* objects as the parent of the corresponding `UAV_Rotor`*n* object.
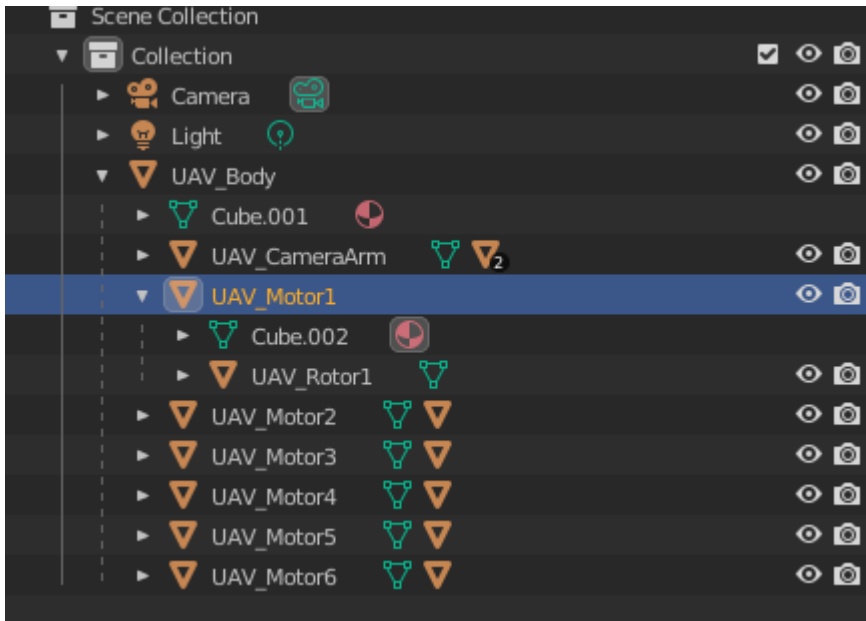
5 Set the `UAV_CameraArm` object as the parent of the `UAV_CameraPivot` object.
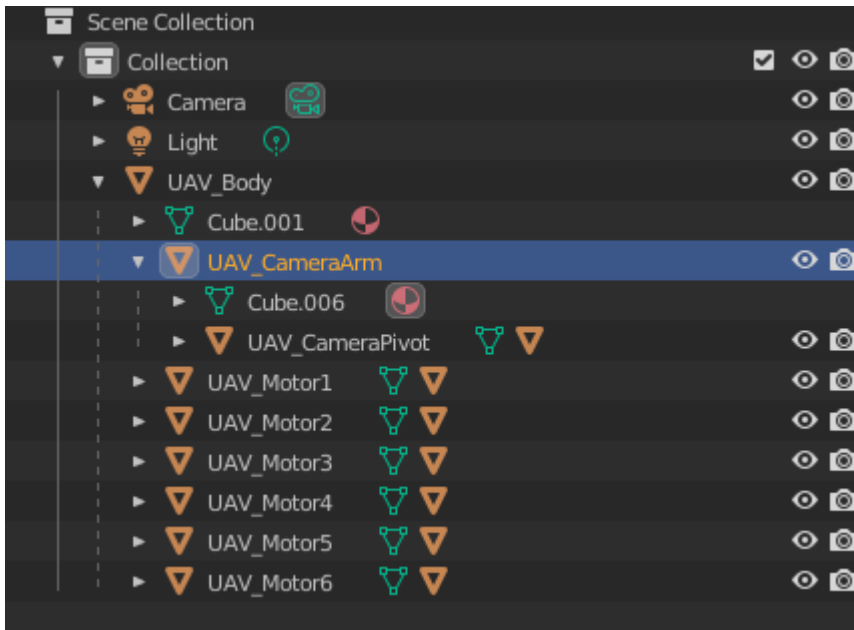
Assign Materials

You can optionally assign material slots to the vehicle parts. In this example, the mesh uses one material for the body, one for the six motors, and one for the six rotors.
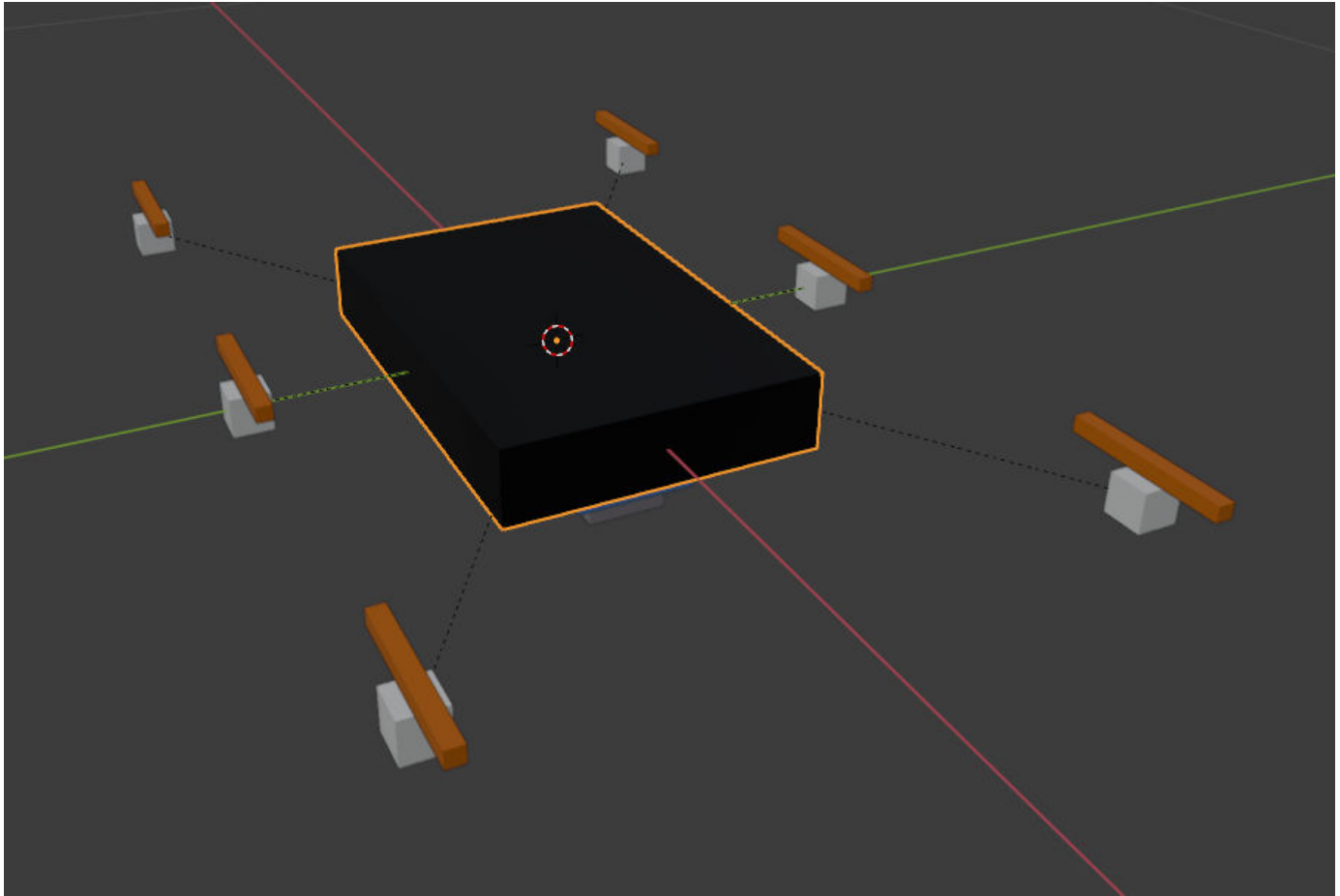
1. Create and assign material slots to the vehicle chassis. Confirm that the first vehicle slot corresponds to the UAV_Body object. For example, this image shows the hierarchy in Blender.
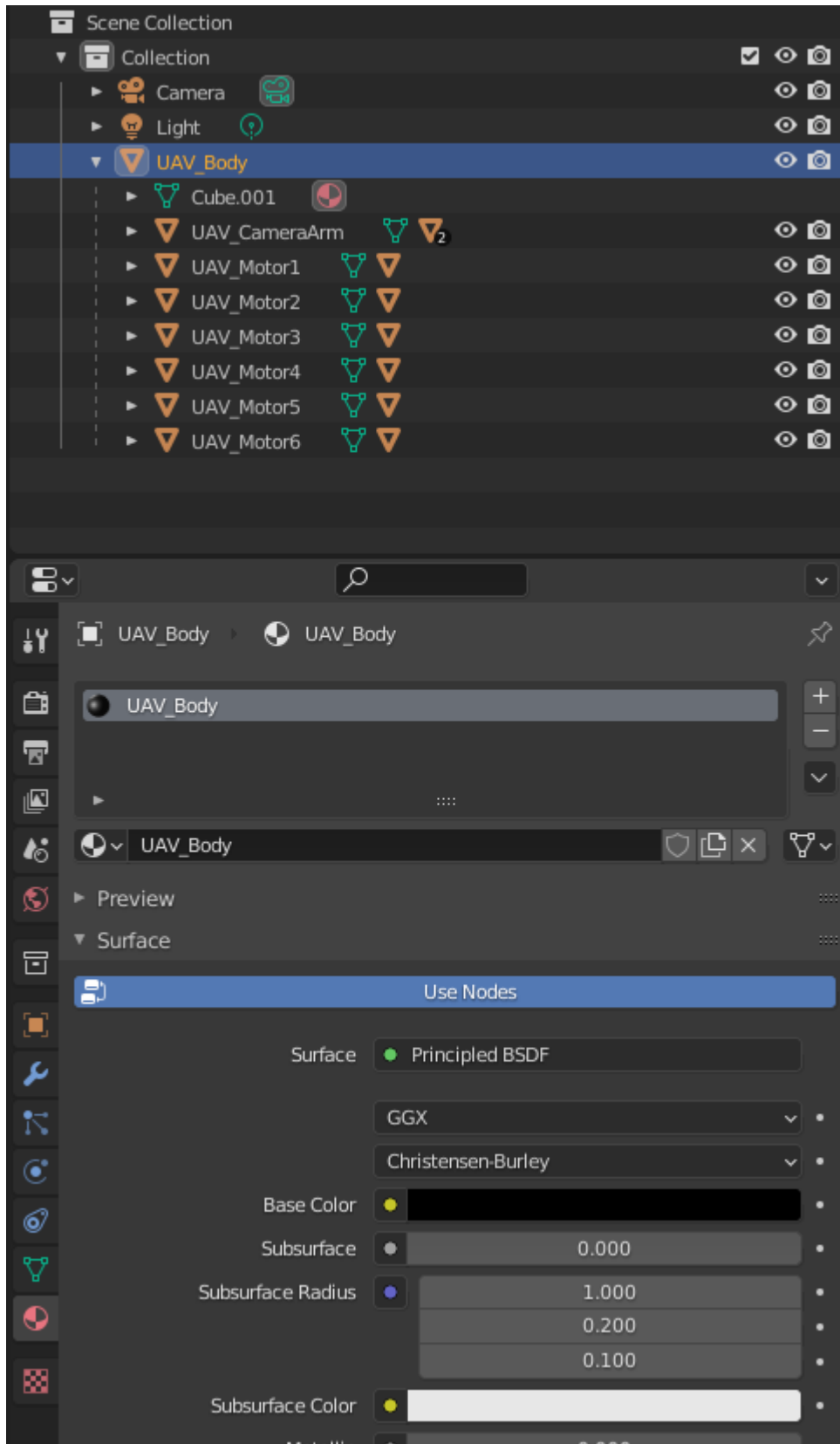
2 Create and assign material slots to the motors.
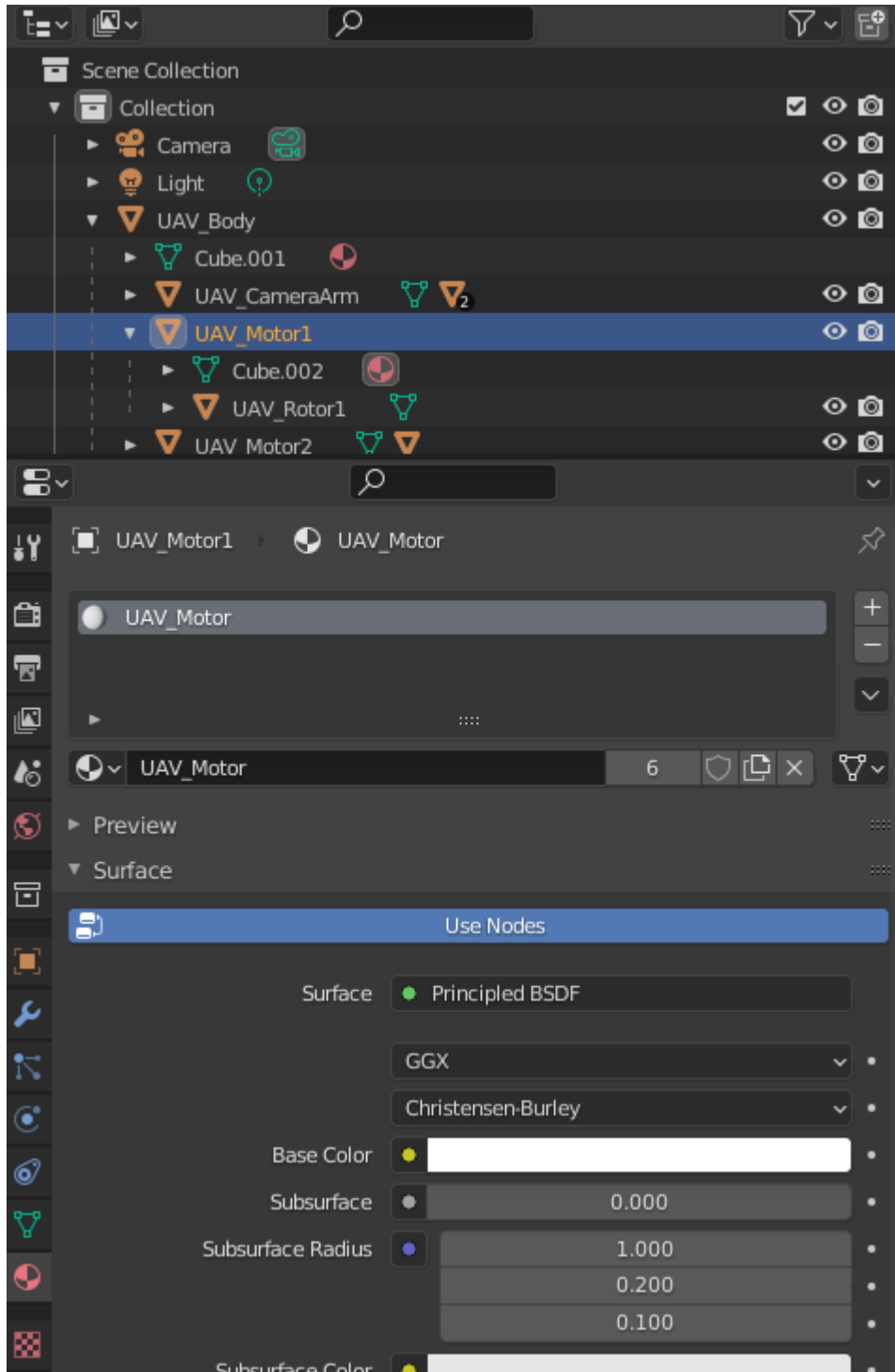
3 Create and assign material slots to the rotors.
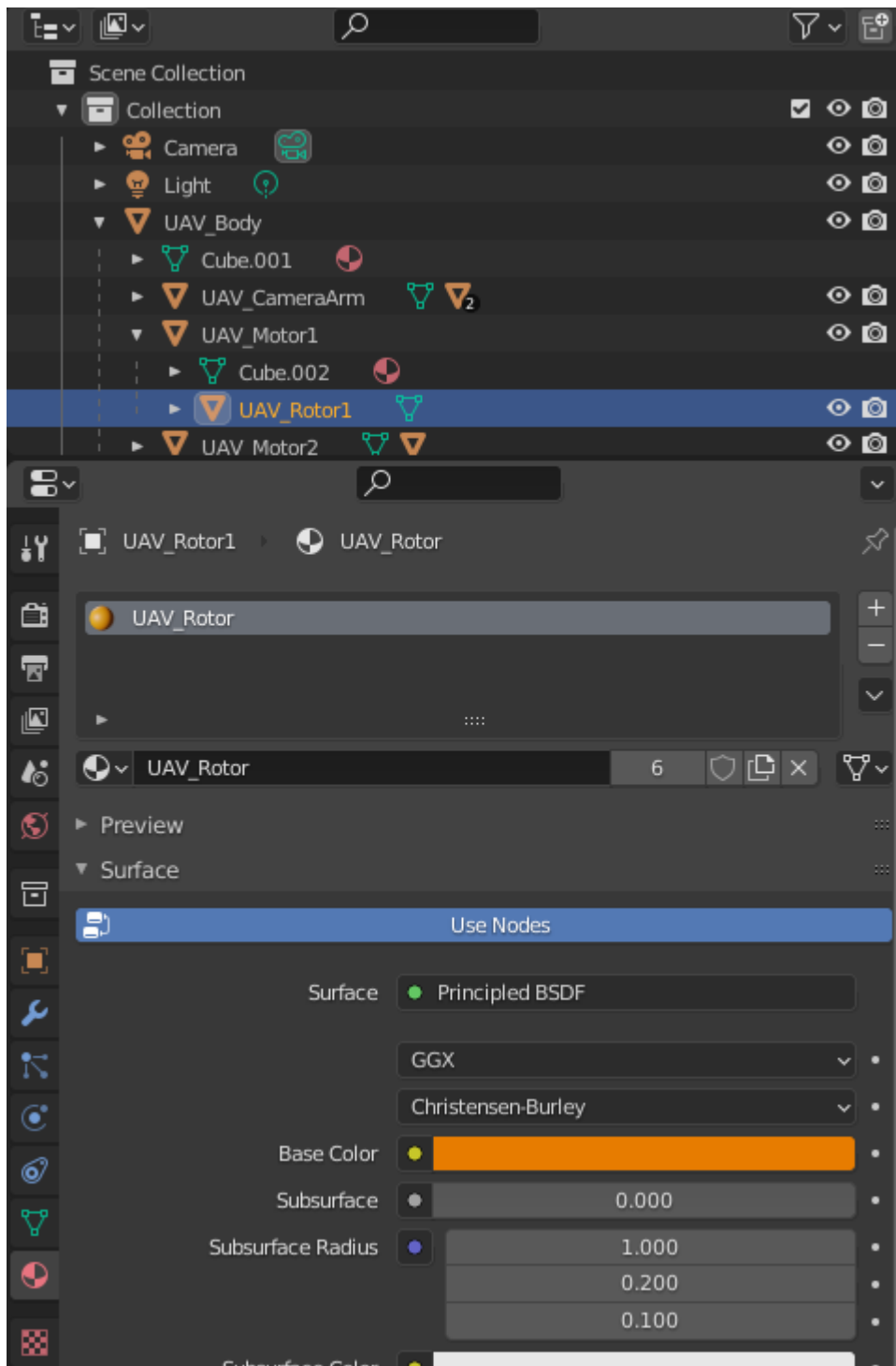
Export Mesh and Armature

Export the mesh and armature in the .fbx file format. For example, in Blender:

1   On the **Object Types** pane, select **Armature** and **Mesh**.
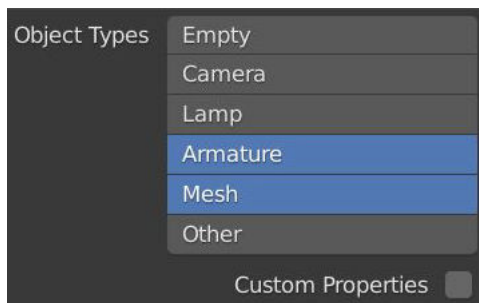
2   On the **Transform** pane, set:

- **Scale** to `1.00`
- **Apply Scalings** to `All Local`
- **Forward** to `X Forward`
- **Up** to `Z Up`

Select **Apply Unit**.
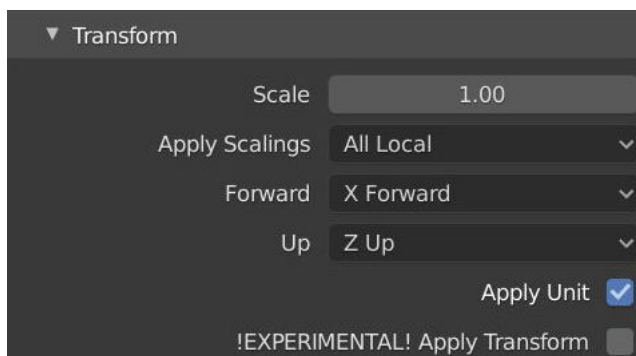
3   On the **Geometry** pane:

- Set **Smoothing** to `Face`
- Select **Apply Modifiers**
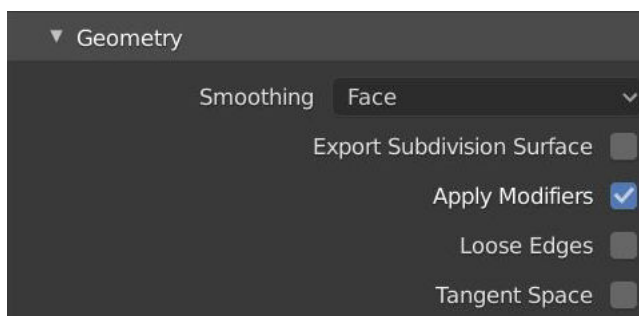
4   On the **Armature** pane, set:

- **Primary Bone Axis** to `X Axis`
- **Secondary Bone Axis** to `Z Axis`

Select **Export FBX**.
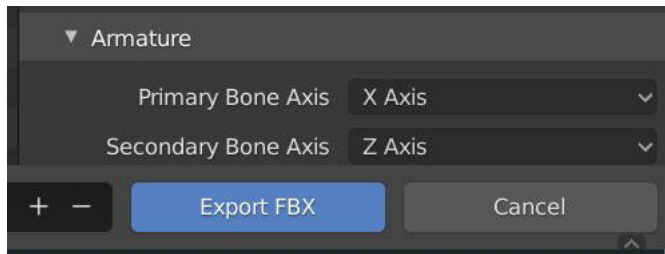
Import Mesh to Unreal Editor

1 Open the Unreal Engine `AutoVrtlEnv.uproject` project in the Unreal Editor.

2 In the editor, import the FBX® file as a skeletal mesh. Assign the `UAV_Skeleton` asset to the **Skeleton** parameter.
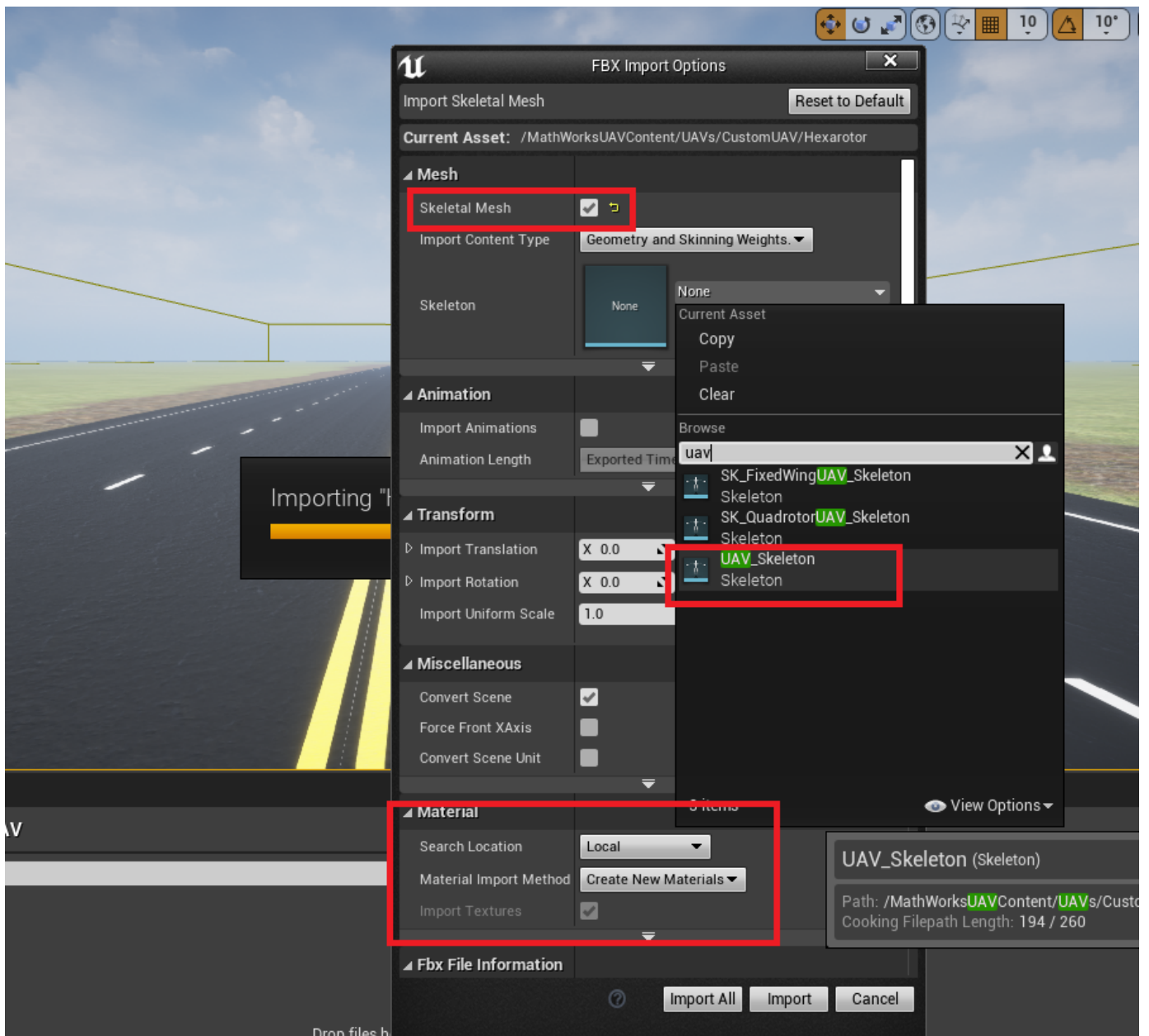
Set Block Parameters

In your Simulink model, set these Simulation 3D UAV Vehicle block parameters:

- **Type** to `Custom`.
- **Path** to the path in the Unreal Engine project that contains the imported mesh.

See Also

Simulation 3D Scene Configuration | Simulation 3D UAV Vehicle

More About

- "Unreal Engine Simulation Environment Requirements and Limitations" on page 2-5
- "Coordinate Systems for Unreal Engine Simulation in UAV Toolbox" on page 2-9

External Websites

- Blender

3D Data Processing – User's Guide

Choose a 3-D Coordinate System

Coordinate systems represent position on the Earth using coordinates. The toolbox provides functions to transform coordinates between geodetic, east-north-up (ENU), and north-east-down (NED).

Global system like geodetic system describe the position of an object using a triplet of coordinates. Local systems such as ENU and NED systems require two triplets of coordinates: one triplet describes the location of the origin, and the other triplet describes the location of the object with respect to the origin.

When you work with 3-D coordinate systems, you must specify an ellipsoid model that approximates the shape of the Earth. All of the sample coordinates on this page use the World Geodetic System of 1984 (WGS84).

Geodetic Coordinates

A geodetic system uses the coordinates [`lat lon alt`] to represent position relative to a reference ellipsoid.

- *lat*, the latitude, originates at the equator. More specifically, the latitude of a point is the angle a normal to the ellipsoid at that point makes with the equatorial plane, which contains the center and equator of the ellipsoid. An angle of latitude is within the range [−90°, 90°]. Positive latitudes correspond to north and negative latitudes correspond to south.
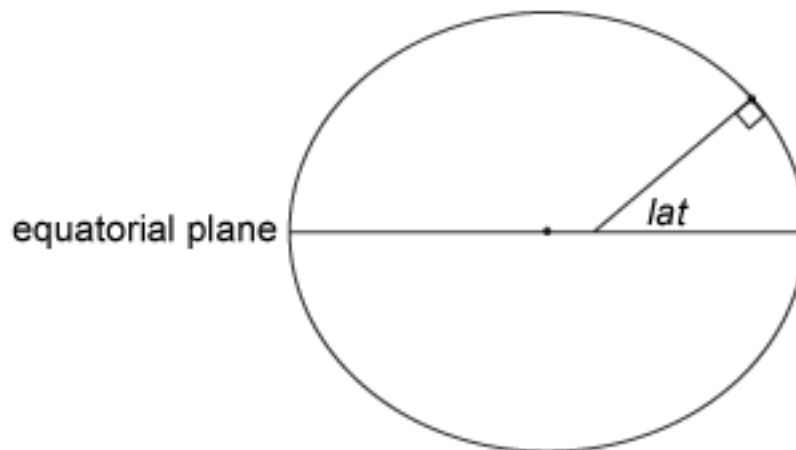
- *lon*, the longitude, originates at the prime meridian. More specifically, the longitude of a point is the angle that a plane containing the ellipsoid center and the meridian containing that point makes with the plane containing the ellipsoid center and prime meridian. Positive longitudes are measured in a counterclockwise direction from a vantage point above the North Pole. Typically, longitude is within the range [−180°, 180°] or [0°, 360°].
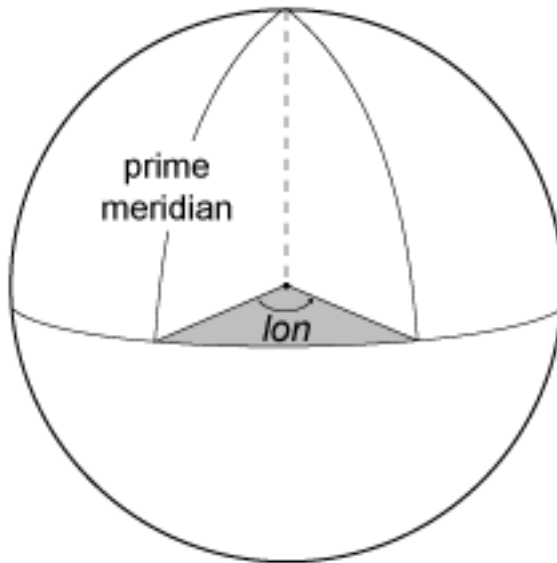
- *alt*, the ellipsoidal height, is measured along a normal of the reference spheroid.
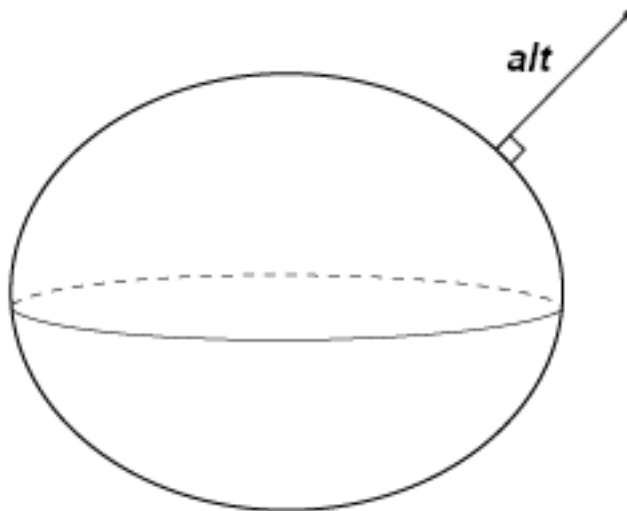
East-North-Up Coordinates

An east-north-up (ENU) system uses the Cartesian coordinates [*xEast yNorth zUp*] to represent position relative to a local origin. The local origin is described by the geodetic coordinates [*lat0 lon0 alt0*]. Note that the origin does not necessarily lie on the surface of the ellipsoid.

- The positive *xEast*-axis points east along the parallel of latitude containing *lat0*.
- The positive *yNorth*-axis points north along the meridian of longitude containing *lon0*.
- The positive *zUp*-axis points upward along the ellipsoid normal.
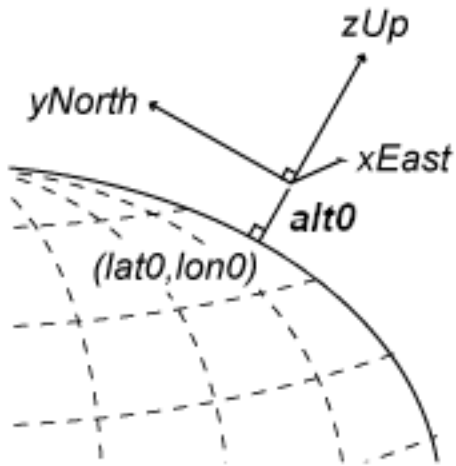
North-East-Down Coordinates

A north-east-down (NED) system uses the Cartesian coordinates [*xNorth yEast zDown*] to represent position relative to a local origin. The local origin is described by the geodetic coordinates [*lat0 lon0 alt0*]. Typically, the local origin of an NED system is above the surface of the Earth.

- The positive *xNorth*-axis points north along the meridian of longitude containing *lon0*.
- The positive *yEast*-axis points east along the parallel of latitude containing *lat0*.
- The positive *zDown*-axis points downward along the ellipsoid normal.
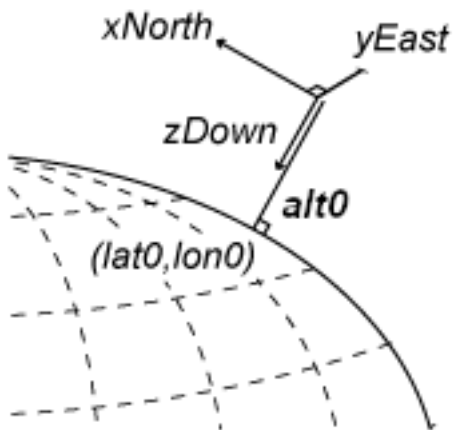
An NED coordinate system is commonly used to specify location relative to a moving aircraft. In this application, the origin and axes of an NED system change continuously. Note that the coordinates are not fixed to the frame of the aircraft.

Tips

If you are transforming coordinates between ENU and NED systems with the same origin, then you do not need to specify a reference ellipsoid or the coordinates of the origin.

See Also

enu2lla | lla2enu | lla2ned | ned2lla

References

[1] Guowei, C., B.M. Cheh, and T. H. Lee. *Unmanned Rotorcraft Systems*. London: Springer-Verlag London Limited: 2011.

[2] Van Sickle, J. *Basic GIS Coordinates*. Boca Raton, FL: CRC Press LLC, 2004.

Simulink Block Examples

Generate Course and Yaw Commands for Orbit Following in Simulink®

This example shows how to use the **UAV Orbit Follower** block to generate course and yaw commands for orbiting a location of interest with a UAV.

Open the model. Click **Open Live Script** to get a copy of the Simulink® model. This model illustrates the inputs and the outputs of the block.

You must specify the current UAV pose as an `[x;y;z;course]` column vector. Typically, the pose is gathered from telemetry data from the UAV.

Also, specify the orbit center location in *xyz*-coordinates, orbit radius, turn direction, and lookahead distance on the path. The lookahead distance is important for tuning the path tracking. Higher values smooth the path, but lower values can improve tracking of the actual orbit.

```
open_system("uav_orbit_follower_ex1.slx")
```
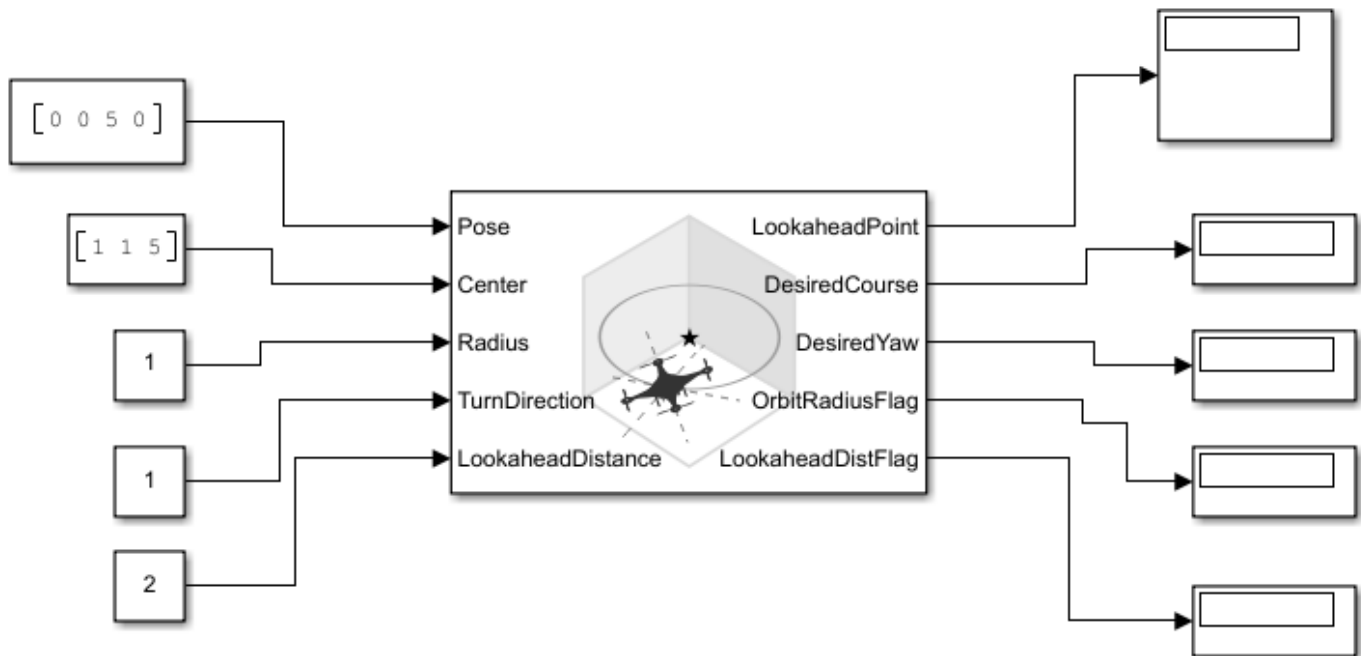

Run the model to get the desired course and yaw for following the orbit. These outputs can be used to generate commands for a UAV.

```
sim("uav_orbit_follower_ex1.slx");
```
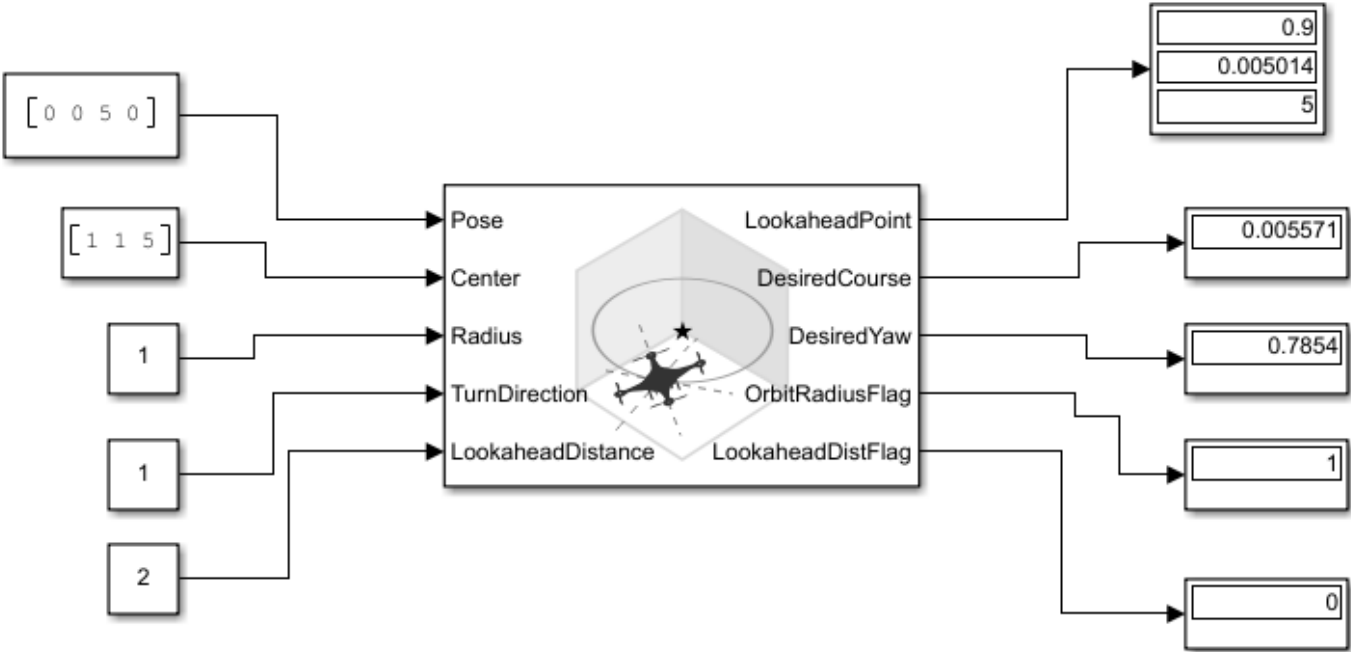
Close the model with:

```
close_system("uav_orbit_follower_ex1.slx", 0);
```

UAV Obstacle Avoidance in Simulink

This model implements waypoint following along with obstacle avoidance on a UAV in a simulated scenario. The model takes a set of waypoints and uses the 3D VFH+ algorithm to provide an obstacle-free path.

Create UAV Scenario with Custom Lidar Sensor and Obstacles

Create Scenario

Create a UAV scenario and set its local origin.

```
Scenario = uavScenario("UpdateRate",100,"ReferenceLocation",[0 0 0]);
```

Add a marker to indicate the start pose of the UAV.

```
addMesh(Scenario,"cylinder",{[0 0 1] [0 .01]},[0 1 0]);
```

Define UAV Platform

Specify the initial position and orientation of the UAV in the north-east-down (NED) frame.

```
InitialPosition = [0 0 -7];
InitialOrientation = [0 0 0];
```

Create a UAV platform in the scenario.

```
platUAV = uavPlatform("UAV",Scenario, ...
                      "ReferenceFrame","NED", ...
                      "InitialPosition",InitialPosition, ...
                      "InitialOrientation",eul2quat(InitialOrientation));
```

Add a quadrotor mesh for visualization.

```
updateMesh(platUAV,"quadrotor",{1.2},[0 0 1],eul2tform([0 0 pi]));
```

Create and Mount Sensor Model

Specify the lidar resolution.

```
AzimuthResolution = 0.5;
ElevationResolution = 2;
```

Specify the lidar range.

```
MaxRange = 7;
AzimuthLimits = [-179 179];
ElevationLimits = [-15 15];
```

Create a statistical sensor model to generate point clouds for the lidar sensor.

```
LidarModel = uavLidarPointCloudGenerator("UpdateRate",10, ...
                                         "MaxRange",MaxRange, ...
                                         "RangeAccuracy",3, ...
                                         "AzimuthResolution",AzimuthResolution, ...
                                         "ElevationResolution",ElevationResolution, ...
                                         "AzimuthLimits",AzimuthLimits, ...
                                         "ElevationLimits",ElevationLimits, ...
                                         "HasOrganizedOutput",true);
```

Create a lidar sensor and mount the sensor on the quadrotor.

```
uavSensor("Lidar",platUAV,LidarModel, ...
          "MountingLocation",[0 0 -0.4], ...
          "MountingAngles",[0 0 180]);
```

Preview the scenario using the `show3D` function.

```
show3D(Scenario);
```
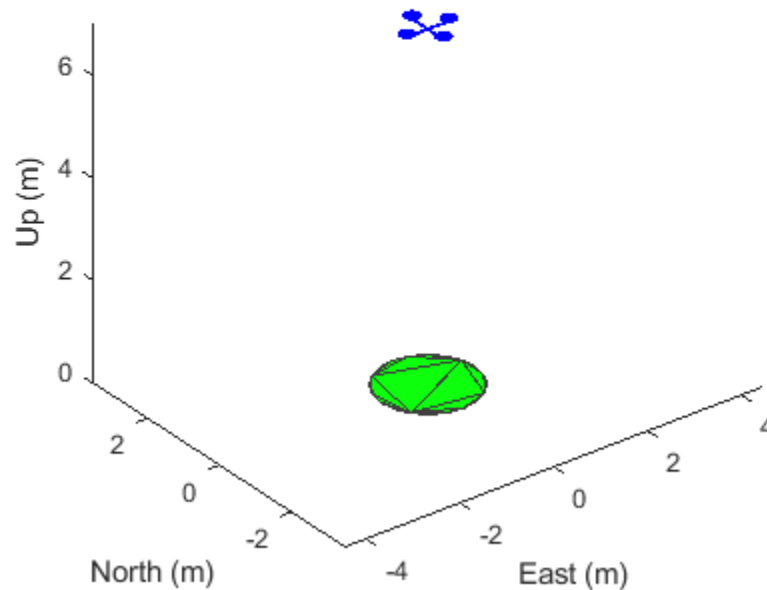

Add Obstacles to Scenario

Add cuboid obstacles, representing buildings, to the scenario.

```
ObstaclePositions = [10 0; 20 10; 10 20]; % Locations of the obstacles
ObstacleHeight = 15;                       % Height of the obstacles
ObstaclesWidth = 3;                        % Width of the obstacles

for i = 1:size(ObstaclePositions,1)
    addMesh(Scenario,"polygon", ...
        {[ObstaclePositions(i,1)-ObstaclesWidth/2 ObstaclePositions(i,2)-ObstaclesWidth/2; ...
        ObstaclePositions(i,1)+ObstaclesWidth/2 ObstaclePositions(i,2)-ObstaclesWidth/2; ...
        ObstaclePositions(i,1)+ObstaclesWidth/2 ObstaclePositions(i,2)+ObstaclesWidth/2; ...
        ObstaclePositions(i,1)-ObstaclesWidth/2 ObstaclePositions(i,2)+ObstaclesWidth/2], ...
        [0 ObstacleHeight]},0.651*ones(1,3));
end
show3D(Scenario);
legend("Start Position","Obstacles")
```
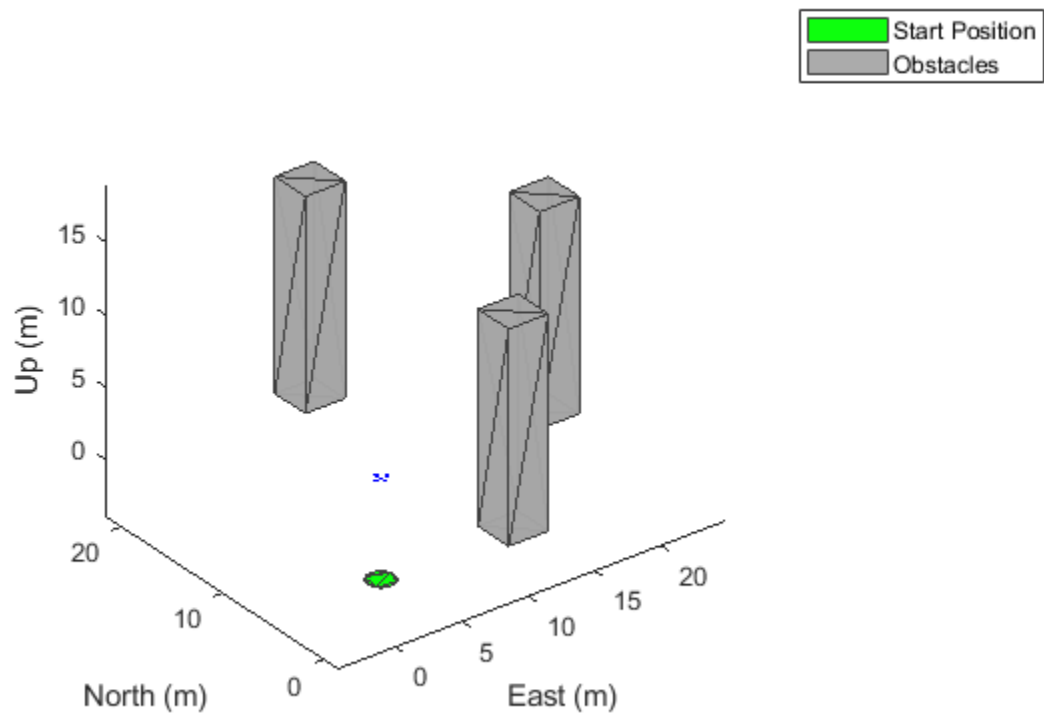
Model Overview

The model consists of these main components:

- `UAV scenario` — Configures the scenario and visualizes the trajectory.
- `Waypoint following and obstacle avoidance` — Implements waypoint following with obstacle avoidance.
- `Controller and plant` — Position controller for the UAV.
- `Control Panel` — Use this panel to enable or disable obstacle avoidance, as well as alter the lookahead distance for obstacle avoidance.

Open the model.

```
open_system("ObstacleAvoidanceDemo.slx");
```
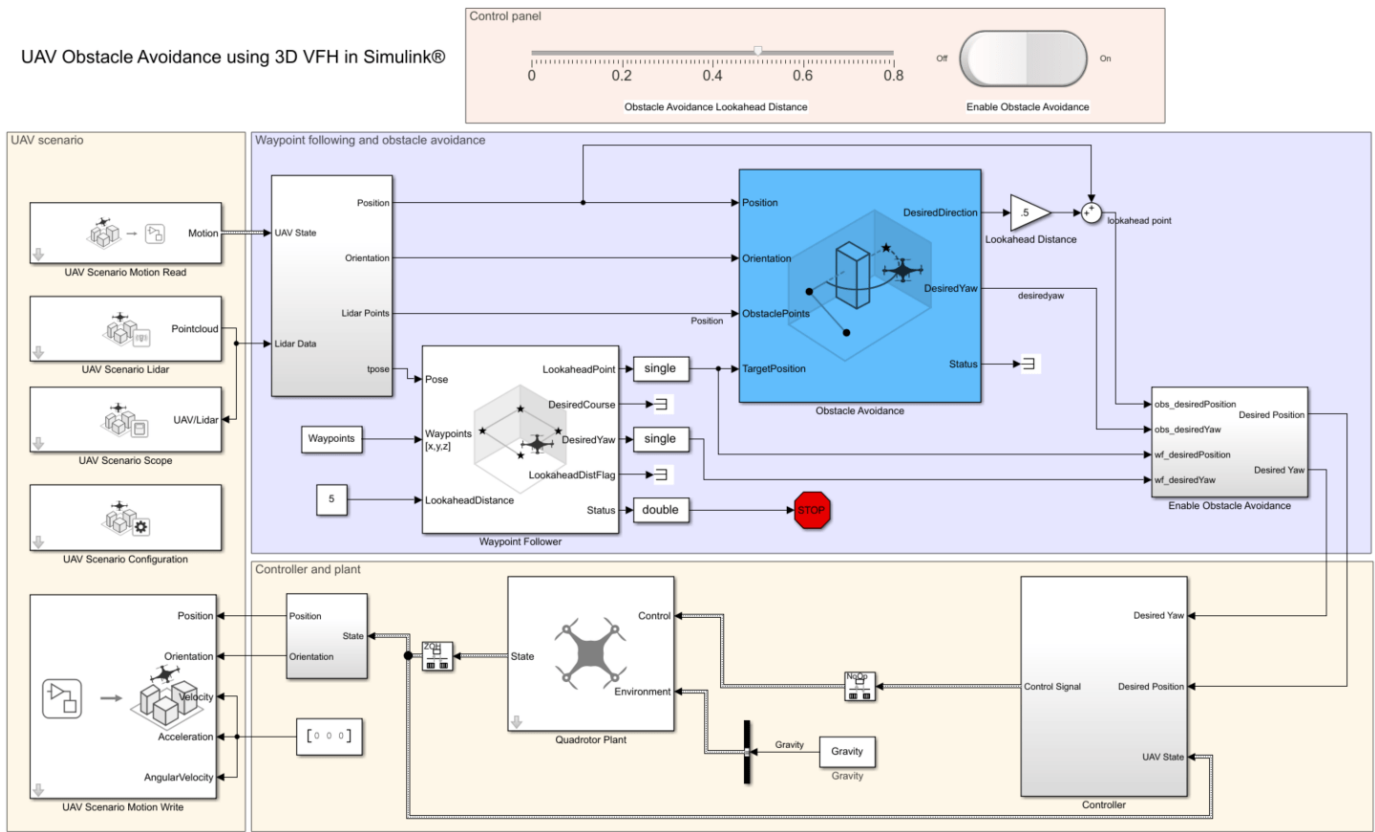
UAV Scenario

The scenario blocks configure the scenario and visualize the obstacles, trajectory, and the lidar point cloud data.
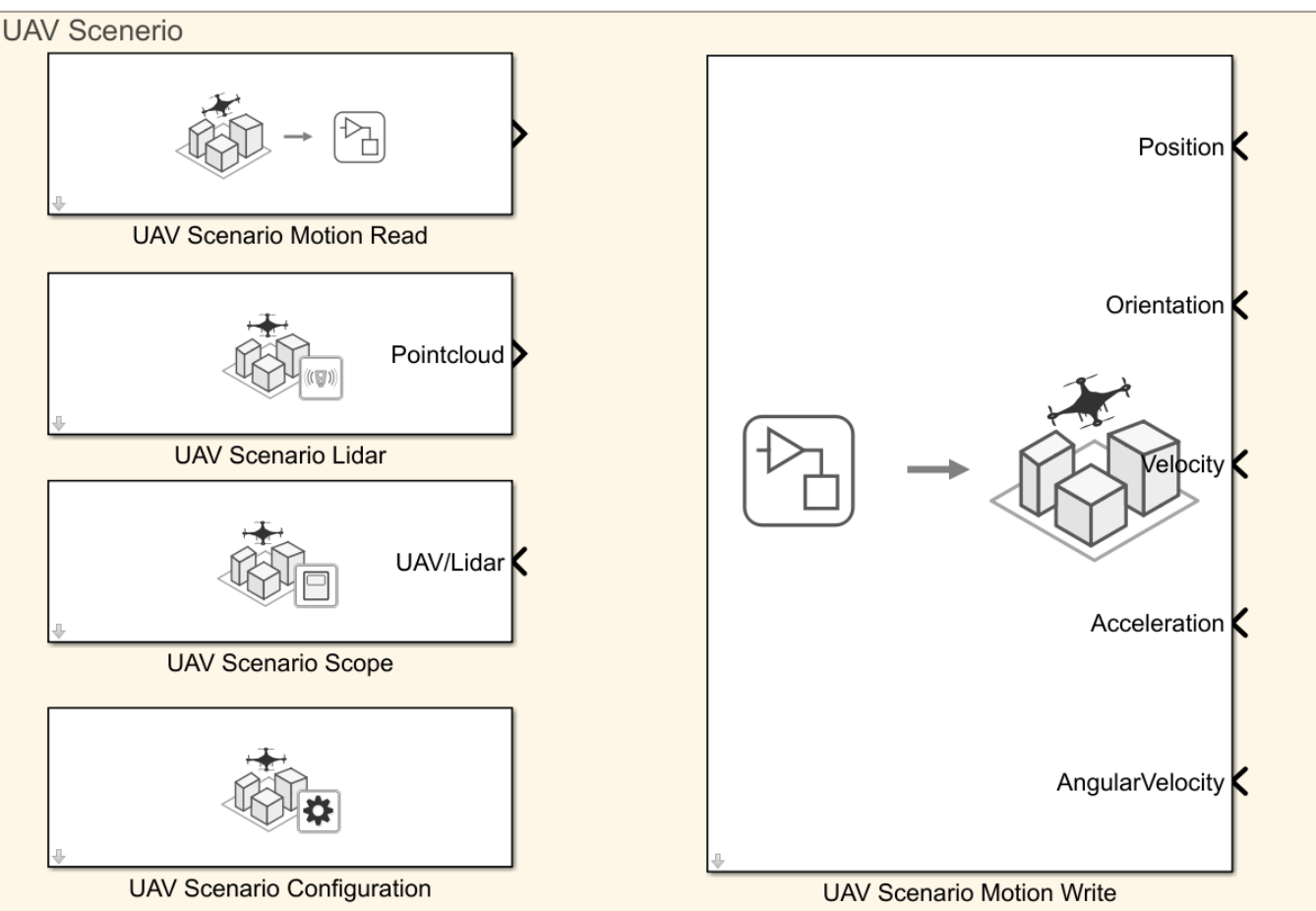
This subsystem contains these blocks:

- UAV Scenario Configuration — Configures the scenario blocks to use the generated scenario for simulation.
- UAV Scenario Motion Read — Reads the current UAV state from the scenario.
- UAV Scenario Lidar — Reads the point cloud data from the scenario.
- UAV Scenario Motion Write — Updates the new UAV state.
- UAV Scenario Scope — Visualizes the UAV trajectory and lidar point cloud data.

Waypoint Following and Obstacle Avoidance

The `Waypoint following and obstacle avoidance` subsystem finds the obstacle-free desired position and the desired yaw according to the current UAV state and point cloud data.
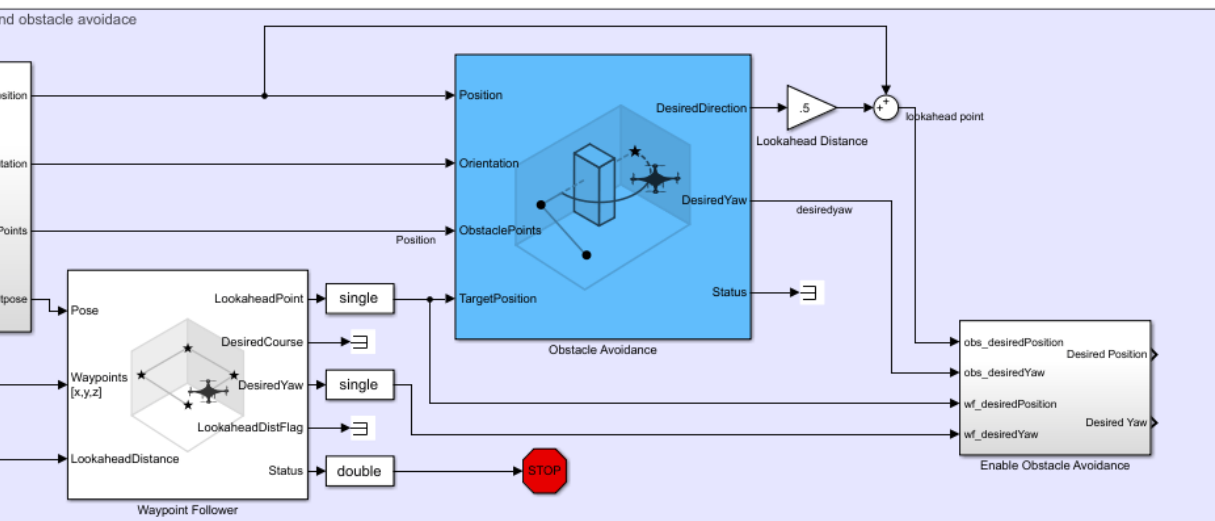
This subsystem includes these blocks and subsystems:

- Waypoint Follower — Computes a lookahead point for the UAV in the direction of the next waypoint.

- Obstacle Avoidance — Uses the 3D VFH+ algorithm to calculate the obstacle-free direction and yaw for a collision-free flight, and updates the lookahead point computed by the Waypoint Follower block.

- Conversion — This subsystem controls the frequency at which obstacle avoidance is used during the flight, and other data type and transform conversions.

- Lookahead Distance — Constant block, the value of which is multiplied by the unit vector in the desired direction, and then added to the current UAV position to compute the desired position.

- Enable Obstacle Avoidance — This subsystem enables or disables obstacle avoidance.

- Waypoints — The set of waypoints through which the UAV is expected to fly.

Specify the waypoints for the UAV.

```
Waypoints = [InitialPosition; 0 20 -7; 20 20 -7; 20 0 -7];
```

Add markers to indicate the waypoints.

```
for i = 2:size(Waypoints,1)
    addMesh(Scenario,"cylinder",{[Waypoints(i,2) Waypoints(i,1) 1] [0 0.1]},[1 0 0]);
end
show3D(Scenario);
hold on
plot3([InitialPosition(1,2); Waypoints(:,2)],[InitialPosition(1,2); Waypoints(:,1)],[-InitialPos:
legend(["Start Position","Obstacles","","","Waypoints","","","Direct Path"])
```
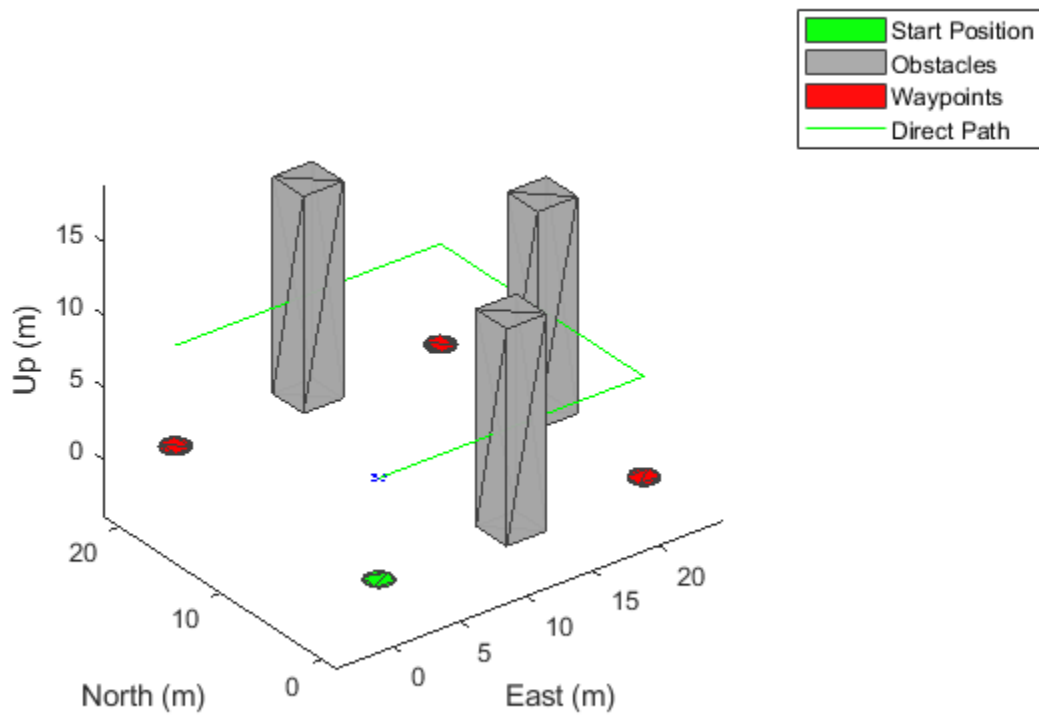
Controller and plant

The `Controller and plant` subsystem generates the control commands and updates the UAV state based on the lookahead point.
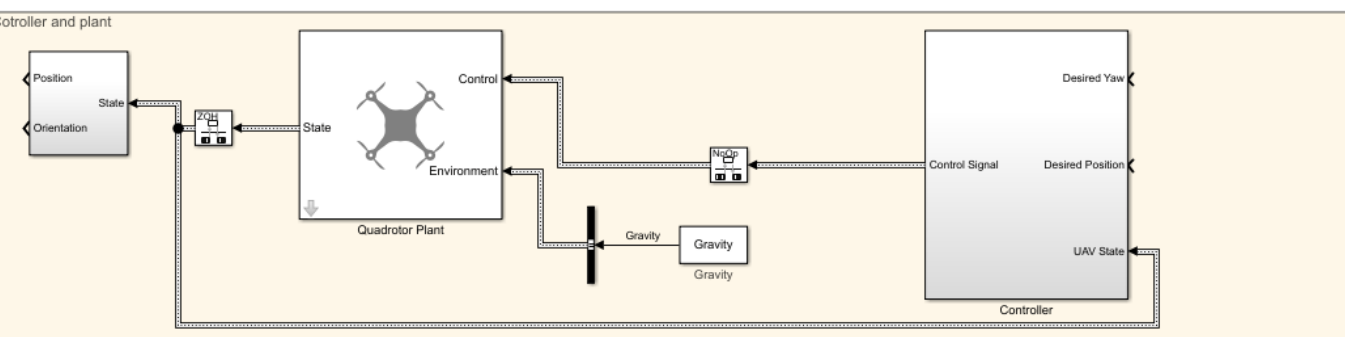
This subsystem includes these blocks:

- `Controller` — This subsystem computes the control commands (roll, pitch, yaw, and thrust) to move the UAV towards the desired position. It uses multiple PID loops to implement position control.

- `Quadrotor Plant` — This Guidance Model block updates the UAV state using the control commands.

- Conversion — This subsystem extracts the position and orientation from the UAV state, and performs data and coordinate transforms for visualization.

Specify the controller parameters. These parameters are based on a hit-and-trial approach, and can be tuned for a smoother flight.

```
% Proportional Gains
Px = 6;
Py = 6;
Pz = 6.5;

% Derivative Gains
Dx = 1.5;
Dy = 1.5;
Dz = 2.5;

% Integral Gains
Ix = 0;
Iy = 0;
Iz = 0;

% Filter Coefficients
Nx = 10;
Ny = 10;
Nz = 14.4947065605712;
```

Specify gravity, drone mass, and sample time for the controller and plant blocks.

```
UAVSampleTime = 0.001;
Gravity = 9.81;
DroneMass = 0.1;
```

Control Panel

The switch enables or disables the updates to the lookahead point from the Obstacle Avoidance block.

The slider updates the lookahead distance used to compute the lookahead point.
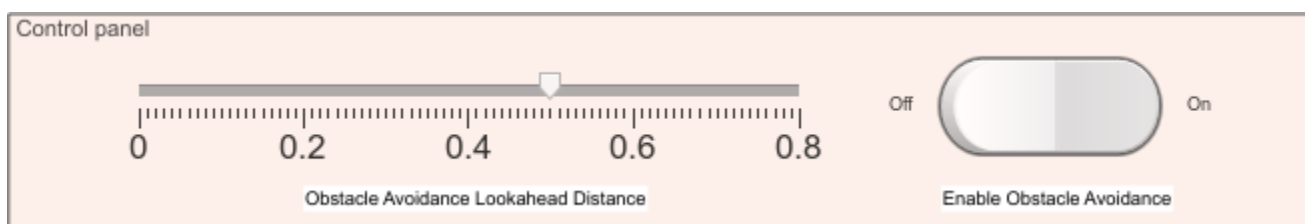
- At greater lookahead distances, UAV flight is faster, but has greater risk of colliding with an obstacle.

- At lower values, the flight is slower, but has a lower risk of colliding with an obstacle.

Simulate Model

Configure and run the model, and observe the motion of the UAV.

- The UAV flies through the waypoints while avoiding obstacles, and then the simulation stops.

- Alter the lookahead distance to change the UAV speed.
- Change the parameters of the Obstacle Avoidance block and note the change in the flight path.

```
out = sim("ObstacleAvoidanceDemo.slx");
```
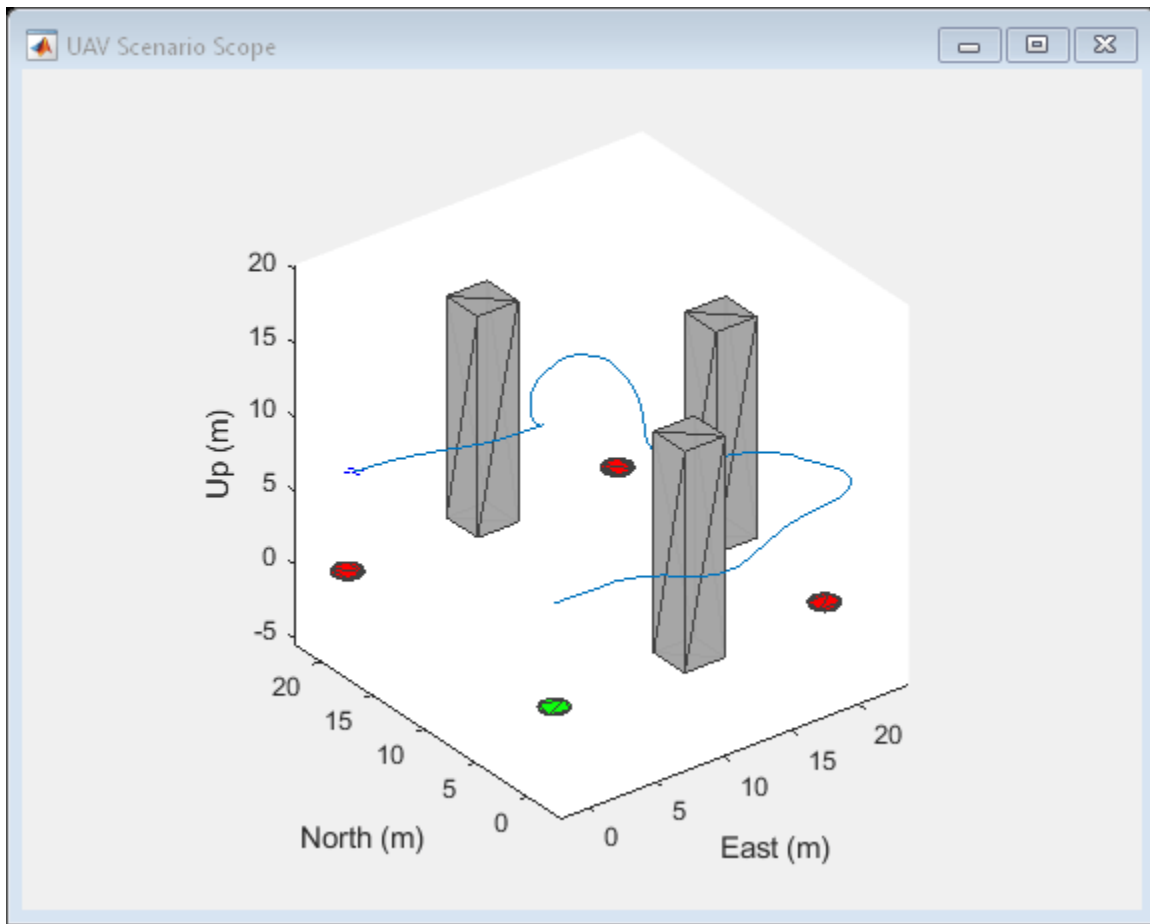

Visualize Obstacle-Free UAV trajectory

Plot the actual UAV trajectory and the waypoints to show the effect of obstacle avoidance on the UAV flight.

```
hold on
points = squeeze(out.trajectoryPoints(1,:,:))';
plot3(points(:,2),points(:,1),-points(:,3),"-r");
legend(["Start Position","Obstacles","","","Waypoints","","","Direct Path","UAV Trajectory"])
```
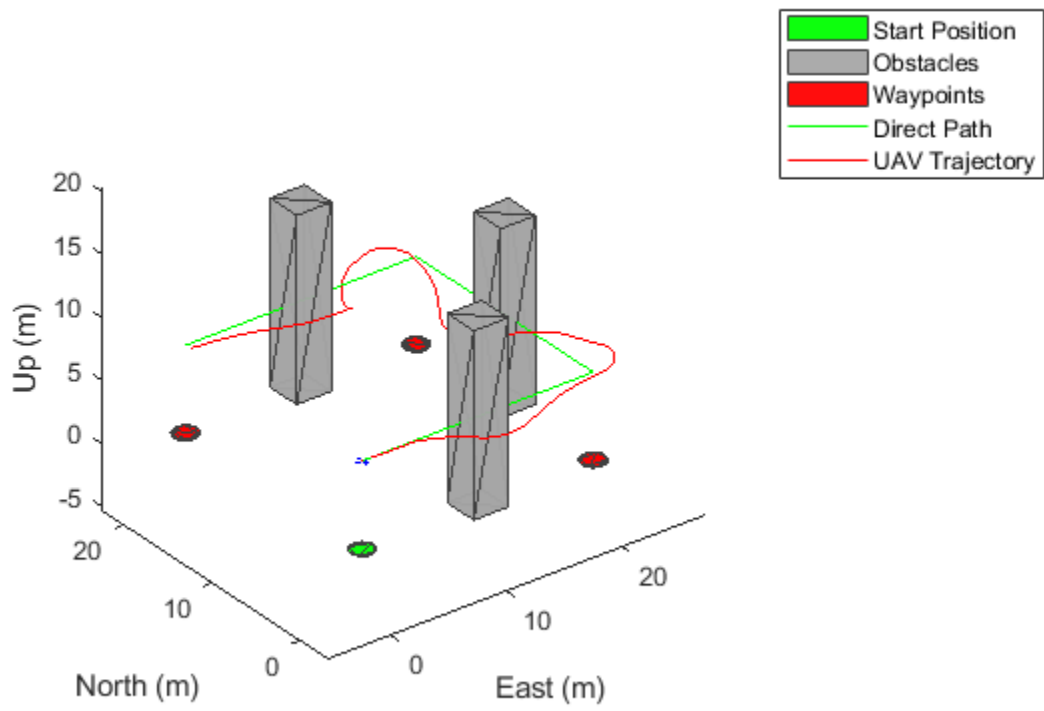
Add GPS Sensor Noise to Multirotor Guidance Model

This example shows how to use a `gpsSensor` Block to add sensor noise to the position and velocity output of a guidance model in Simulink®.

Example Model

Open the Simulink model.

```
open_system("uavGPSModel.slx");
```
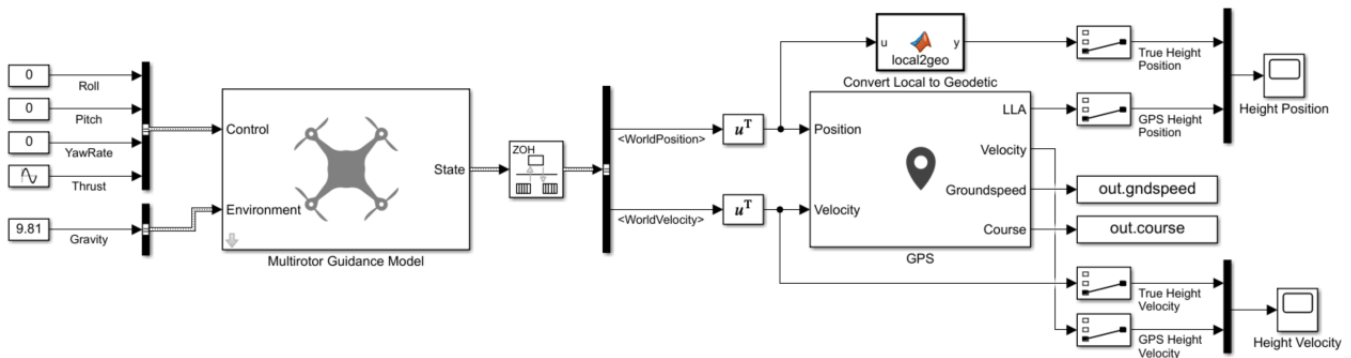

© Copyright 2021 The MathWorks, Inc.

The model uses a Guidance Model block to simulate a multirotor UAV platform. The GPS block adds noise to the state of the UAV.

Constant values for the roll, pitch, and yaw rate, and a varying value created by a sine wave for the thrust are inputs to **Control** port of the Guidance Model. The roll, pitch and yaw rate are set to 0 so the varying thrust input changes only the height. When setting the controls for the block, check that the gain values listed in the block are appropriate. The only **Environment** input for a multirotor UAV is gravity, specified as 9.81. Bus Creator (Simulink) blocks combine the control inputs and environment input into their respective busses, with the bus names specified in the **Input/Output Bus Names** parameter of the Guidance Model.

Because the GPS block uses discrete states and has separate inputs for Position and Velocity, the model uses a Rate Transition block to convert the continuous signal from the Guidance Model block into a bus containing two discrete signals. It extracts the `WorldPosition` and `WorldVelocity` signals from the bus and transposes them before using them as input to the GPS block. Scope blocks display the true vertical position and vertical velocity of the UAV alongside its GPS-affected vertical position and vertical velocity readings.
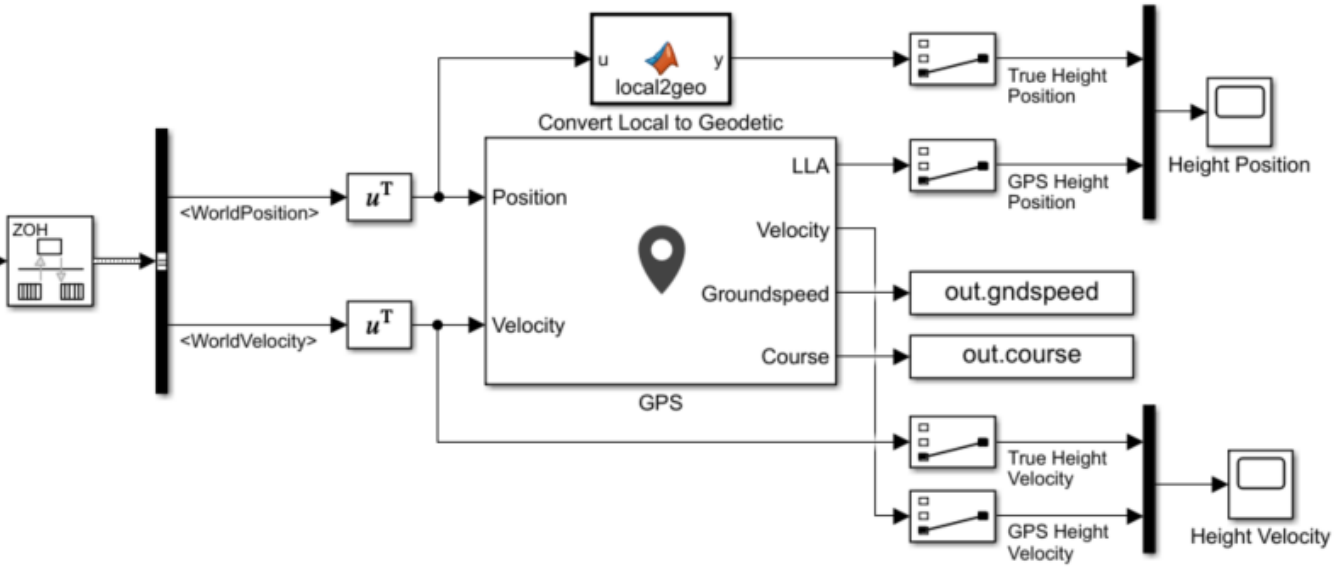
Run the Model

Run the model. The `Height Position` and `Height Velocity` scopes show the effect the GPS block has on the original values of the position and velocity signals, retrospectively.
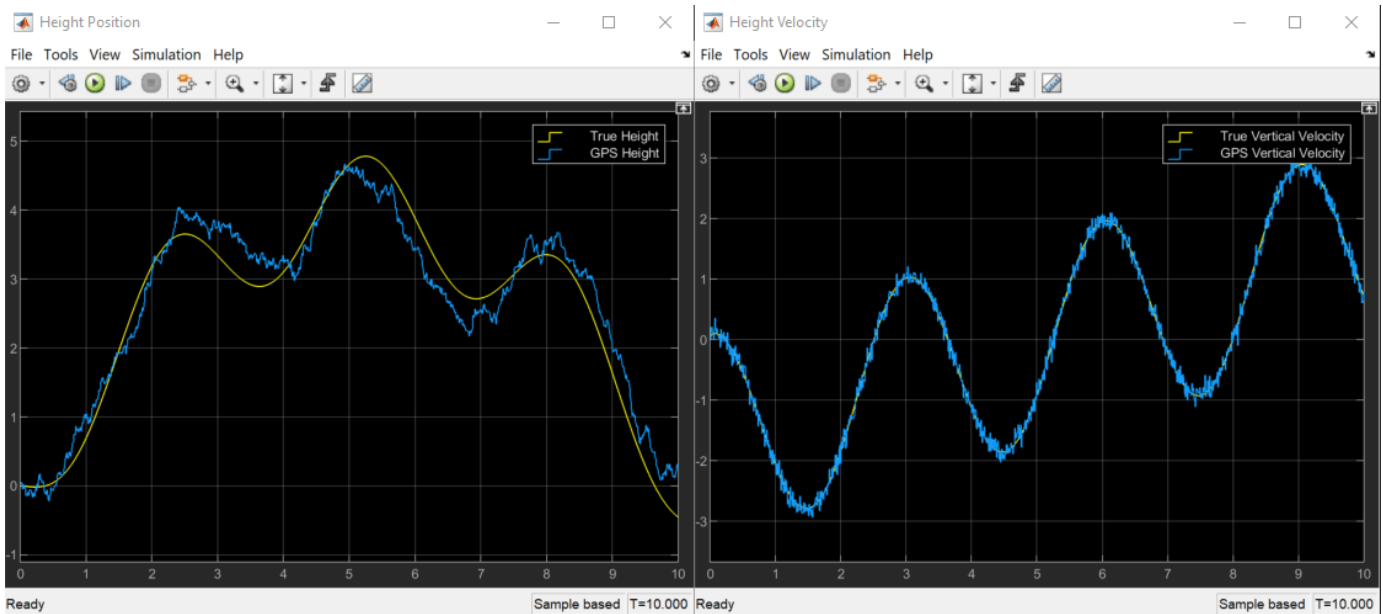
Simulate GPS Sensor Noise

This example shows how to use the GPS block to add GPS sensor noise to position and velocity inputs in Simulink®.

Examine Model

Open the Simulink model.

```
open_system("GPSNoiseModel.slx");
```
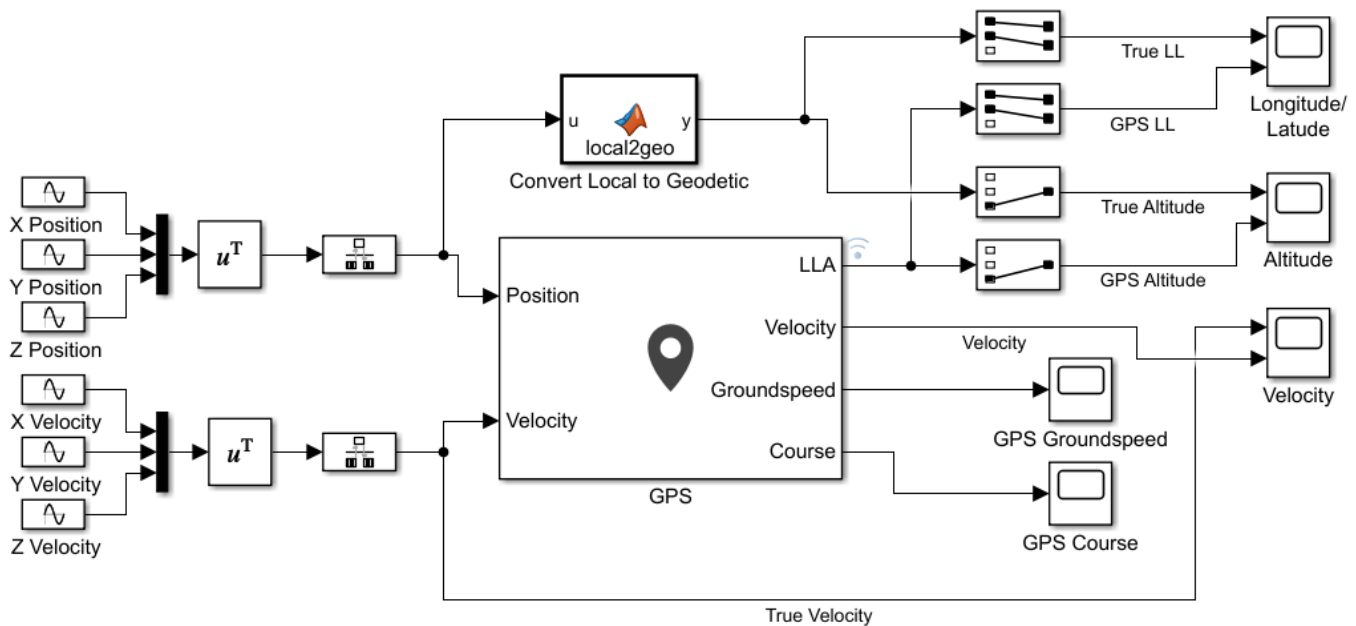

© Copyright 2021 The MathWorks, Inc.

This model generates the X, Y, Z values, for both position and velocity, as individual sine waves and combines them using Mux blocks. Because the GPS block requires discrete signals, the combined position and velocity pass through Rate Transition blocks to the inputs to the **Position** and **Velocity** ports of the GPS block. The GPS block has default parameter settings except for the **Vertical position accuracy**, which is set to `1.5` due to the scale of the position and velocity.

Compare the outputs of the GPS block against the true signal values using Scope blocks. To do this for position, the local position coordinates will need to be converted to LLA coordinates. Use `ned2lla` function in the MATLAB Function block to convert the NED coordinates to LLA coordinates.

A MATLAB Function block uses the `ned2lla` function to convert the local position coordinates of the true signal values to geodetic coordinates. The model then plots the outputs of the GPS block against the true signal values.

Run Model

Run the model. The output scopes show the effect of the noise from the GPS sensor on the original and velocity outputs.
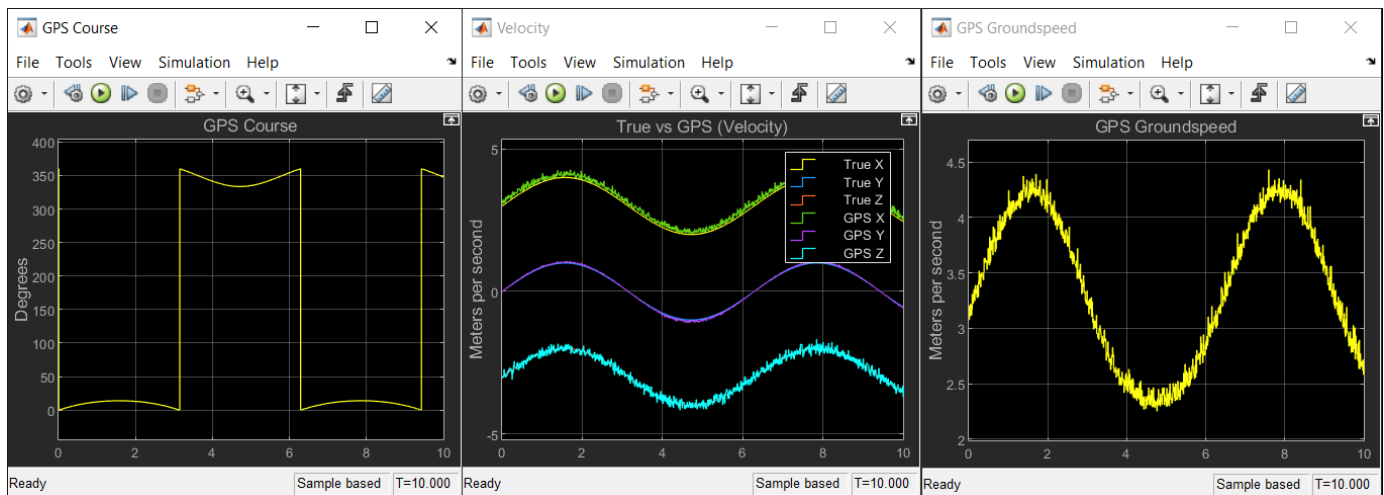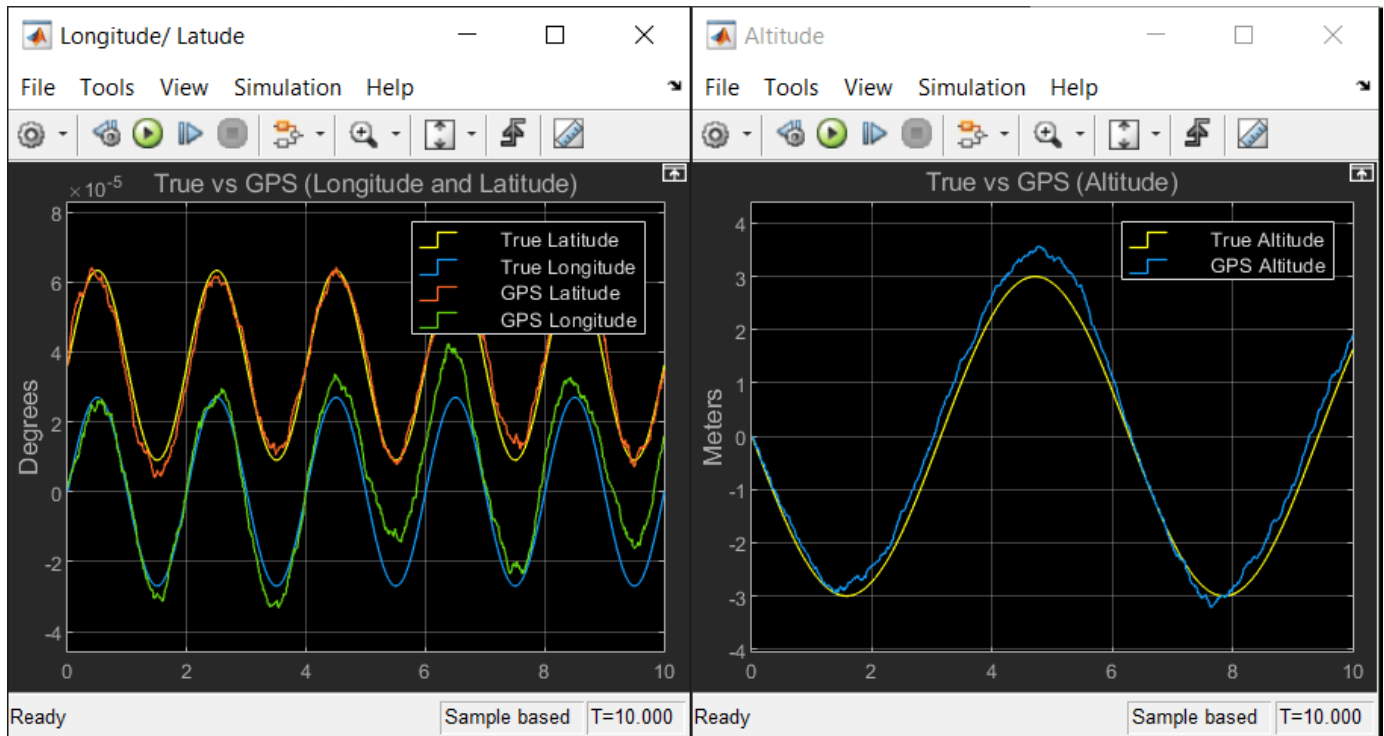
See Also

Functions
ned2lla

Blocks
GPS | Rate Transition

Simulate UAV Scenario Using Scenario Blocks

This example shows how to use the UAV scenario blocks to simulate a scenario in Simulink®.

Create Scenario

Initialize your UAV scenario with meshes and `uavPlatform` objects for the UAV scenario blocks to use.

```
% Initialize the scenario
scene = uavScenario(UpdateRate=100,ReferenceLocation=[0 0 0]);

%Create a ground for visualization
addMesh(scene,"polygon",{[-15 -15; 15 -15; 15 15; -15 15] [-0.5 0]},[0.3 0.3 0.3]);

% Add cylinder meshes to scan with lidar sensor
addMesh(scene,"cylinder",{[-5 5 2] [0 12]},[0 1 0]);
addMesh(scene,"cylinder",{[5 5 2] [0 12]},[0 1 0]);
addMesh(scene,"cylinder",{[5 -5 2] [0 12]},[0 1 0]);
```

Create one UAV platform to be controlled and another to be stationary.

```
% Platform/UAV initial position and orientation
initpos = [0 0 -5]; % NED Frame
initori = [0 0 0];

% Add UAV Platforms to the Scenario and scale them for easier visualization
platform = uavPlatform("platformUAV",scene,ReferenceFrame="NED", ...
    InitialPosition=initpos,InitialOrientation=eul2quat(initori));
platform2 = uavPlatform("platformUAV2",scene,ReferenceFrame="NED", ...
    InitialPosition=[0 7 -11],InitialOrientation=eul2quat(initori));

updateMesh(platform,"quadrotor",{2},[0 0 0],eul2tform([0 0 pi]));
updateMesh(platform2,"quadrotor",{2},[0 0 0],eul2tform([0 0 pi]));
```

Create a `uavSensor` with a lidar sensor model, for use in the Simulink simulation, by using the `uavLidarPointCloudGenerator,` and attach it to the first UAV platform. Specify the parameters of the lidar in the UAV Scenario Lidar block, because the model does not load them from the object.

```
LidarModel = uavLidarPointCloudGenerator;
uavSensor("Lidar",platform,LidarModel,MountingLocation=[0 0 1],MountingAngles=[0 0 180]);
```

Configure the Scenario

```
open_system("UAVScenarioModel.slx")
```
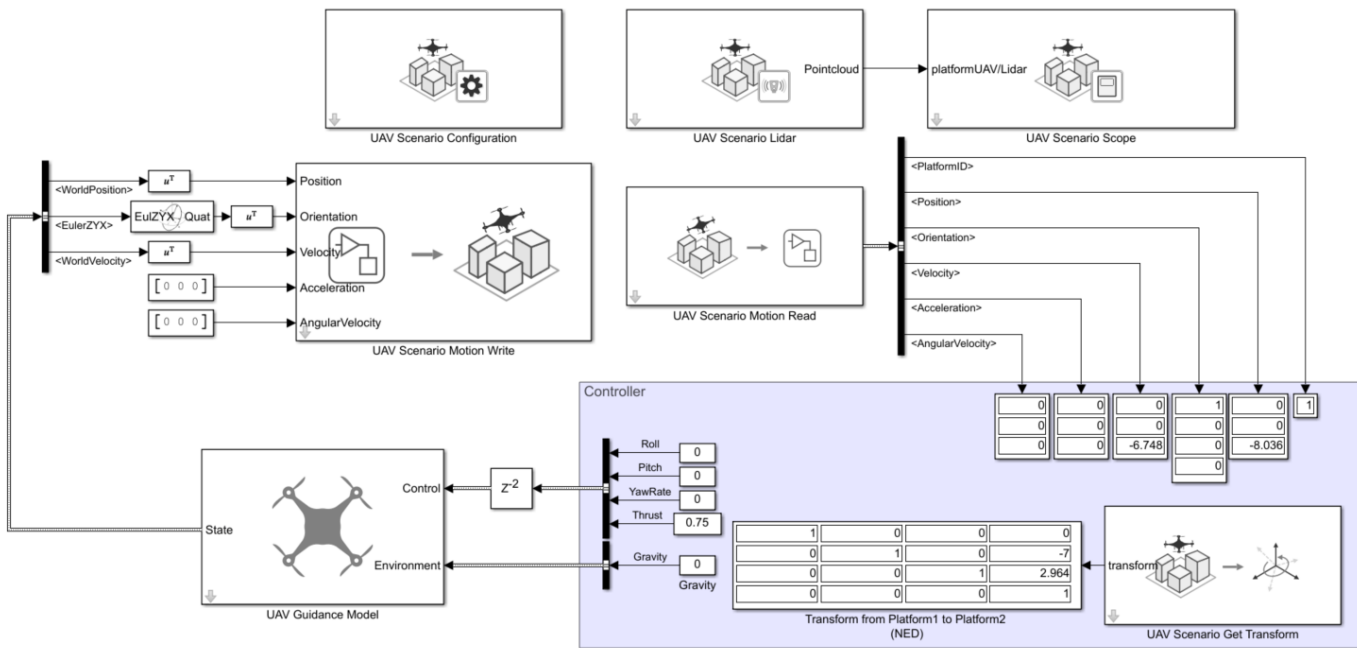
©Copyright 2021 The MathWorks, Inc.

The UAV Scenario Configuration block is essential for simulating the UAV scenario in Simulink, and must execute first. Place it in the model and set the **MATLAB or model workspace variable name** to the name of your scenario. Whenever you make changes to the scenario scene in MATLAB, open this block and click **Refresh** to reflect those changes in Simulink. Change the sample time and the bus and signal names as necessary.

Block Parameters: UAV Scenario Configuration   ✕

UAV Scenario Configuration (mask) (link)

Import a uavScenario object and simulate the scenario.

You must have this block in models that have UAV Scenario sensor and motion blocks to test perception, control and planning algorithms with data from uavScenario environment.

Sample time must be a positive value. This block only supports discrete sample time.

Main	Scenario Bus and Signal Names

MATLAB or model workspace variable name: scene ⋮ Refresh

Sample time: 0.1 ⋮

OK   Cancel   Help   Apply

Write to Motion Bus

The UAV Scenario Motion Write block updates the motion bus of the first platform UAV. This updates the position of the UAV in the figure from the UAV Scenario Scope block. Open the UAV Scenario Motion Write block parameters and click **Select** to configure the block to update `platformUAV`. This is the platform UAV that we will be updating with this block.
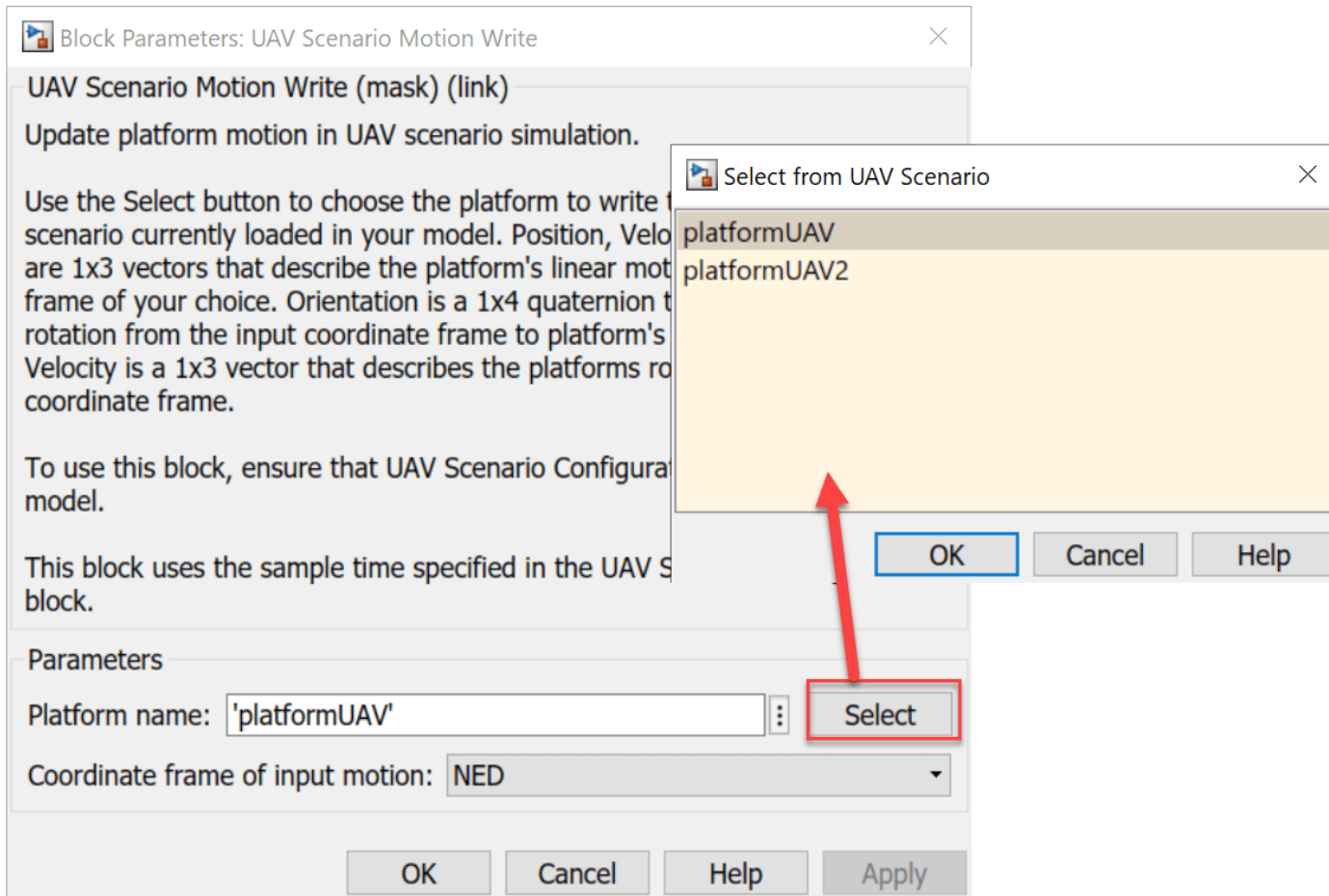
This example uses a multirotor guidance model with constant inputs to determine the next position to write to the motion bus. However, you can use any kind of controller to update the motion bus of the platform if you can extract inputs to use in the UAV Scenario Motion Write block.

UAV Scenario Motion Write

UAV Guidance Model
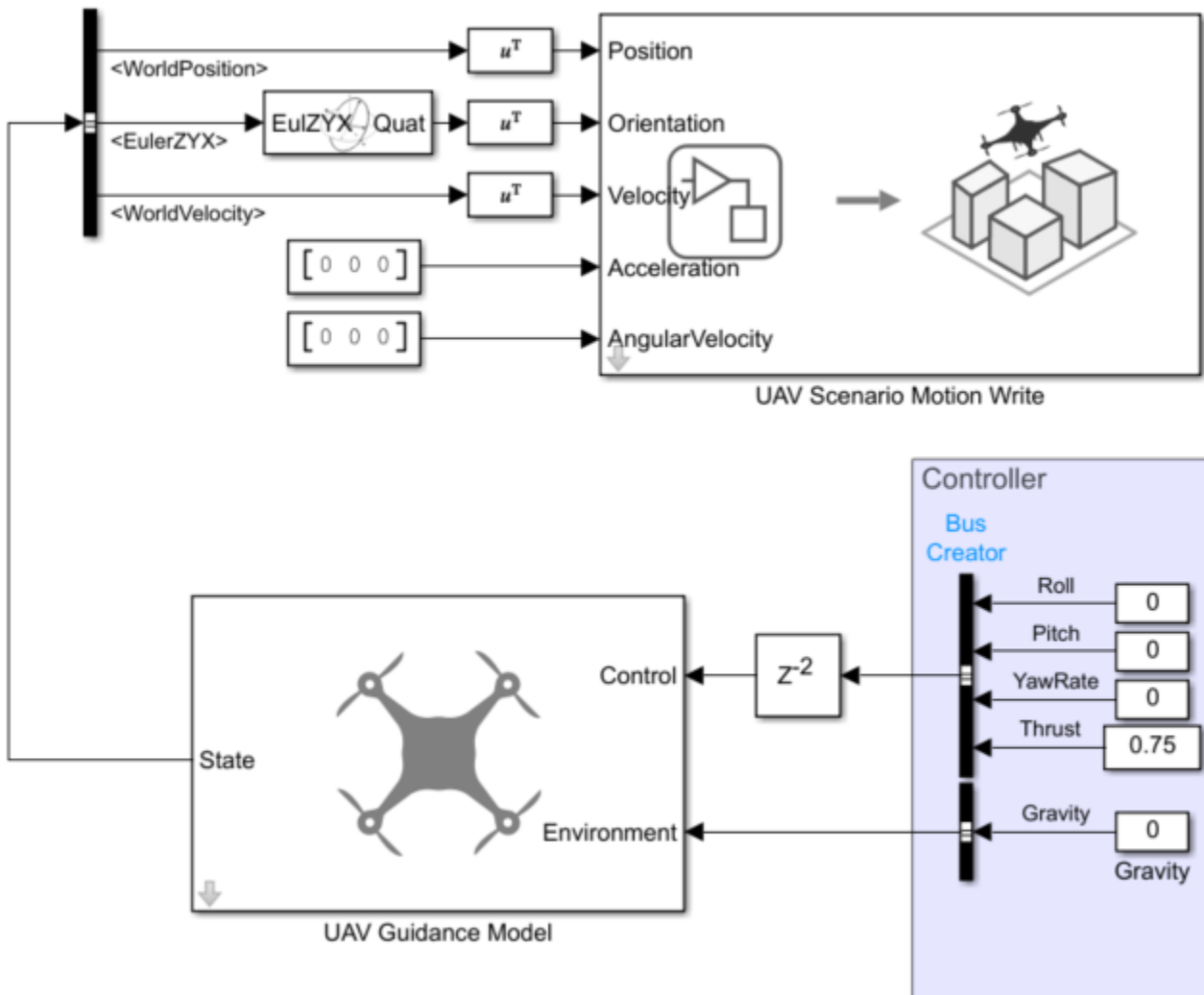
Controller

Read from Motion Bus

After the UAV Scenario Motion Write block writes to the motion bus, the UAV Scenario Motion Read block reads the data from the motion bus. A controller can use this data to determine the inputs to your plant. This example does not use a controller, instead reading the outputs into Display blocks. Open UAV Scenario Motion Write block parameters and click **Select** to again choose `platformUAV`.
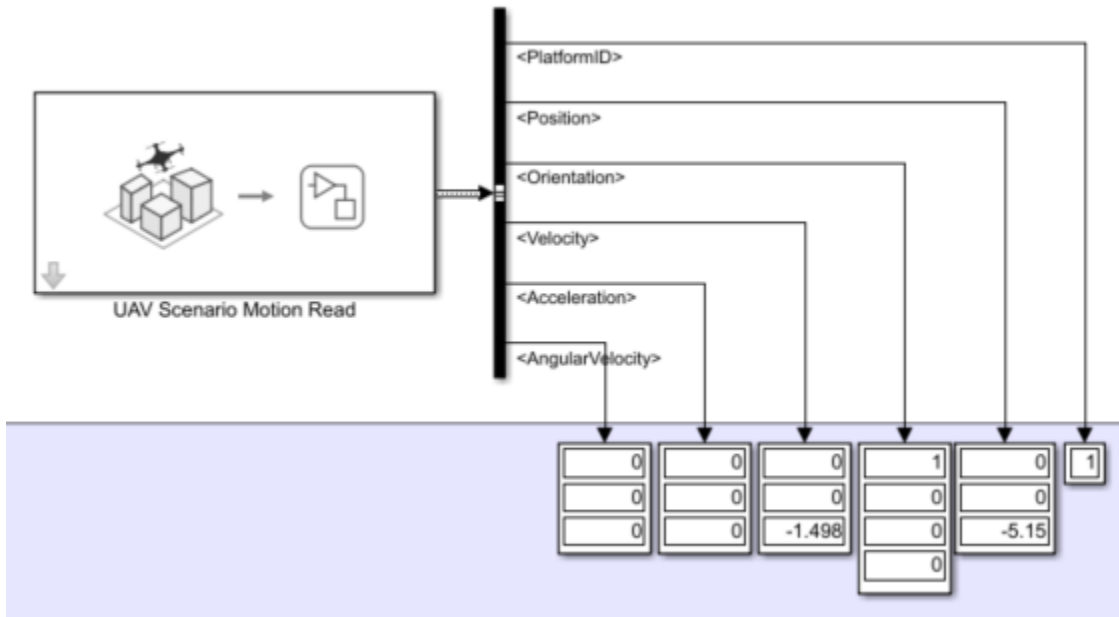
Get Transform

The UAV Scenario Get Transform block gets the transformation matrix between the two UAVs. Open the UAV Scenario Get Transform block parameters and set the **Source Frame** parameter to `platformUAV` and the **Target Frame** parameter to `platformUAV2`.
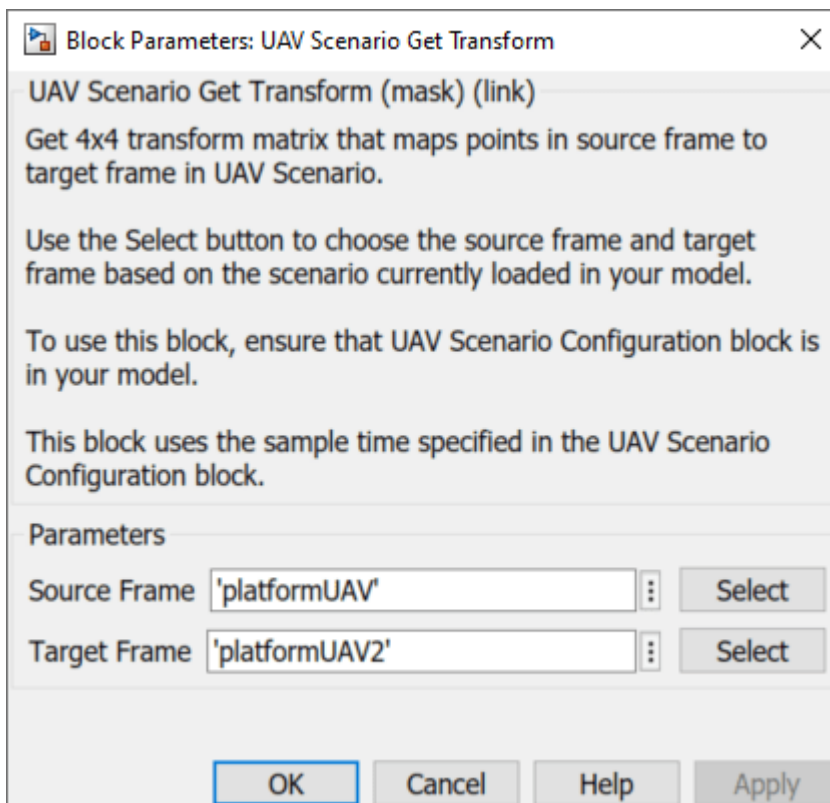
In this example, a Display block shows the transfomation matrix. However a controller can have many applications for the transformation data.
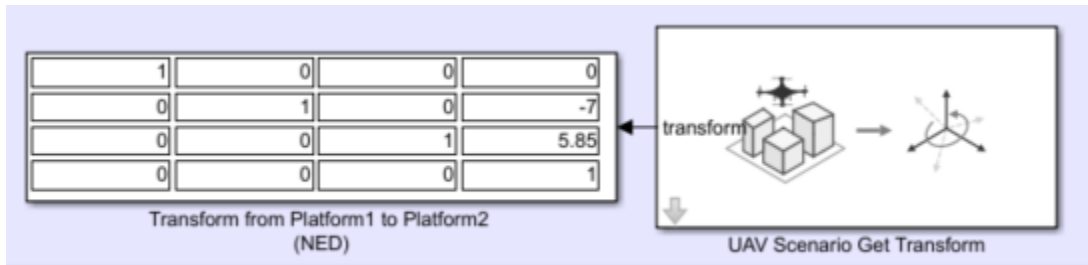
View and Simulate Lidar

The UAV Scenario Lidar block enables the use of the created lidar sensor in the scenario. Open the UAV Scenario Lidar block parameters and specify the **Sensor name** parameter as `platformUAV/Lidar` by clicking **Select.** You can edit the settings of the lidar by specifying parameters in the block.

To visualize the lidar readings and the motion of the UAV, connect the UAV Scenario Lidar block to a UAV Scenario Scope block. You must create a port by enabling the visualization toggle for the sensors you want to connect. If you recently added a lidar sensor, but do not see it in the table, open the UAV Scenario Configuration block parameters and click **Refresh,** and open the UAV Scenario Lidar block parameters and click **Refresh sensor table**.
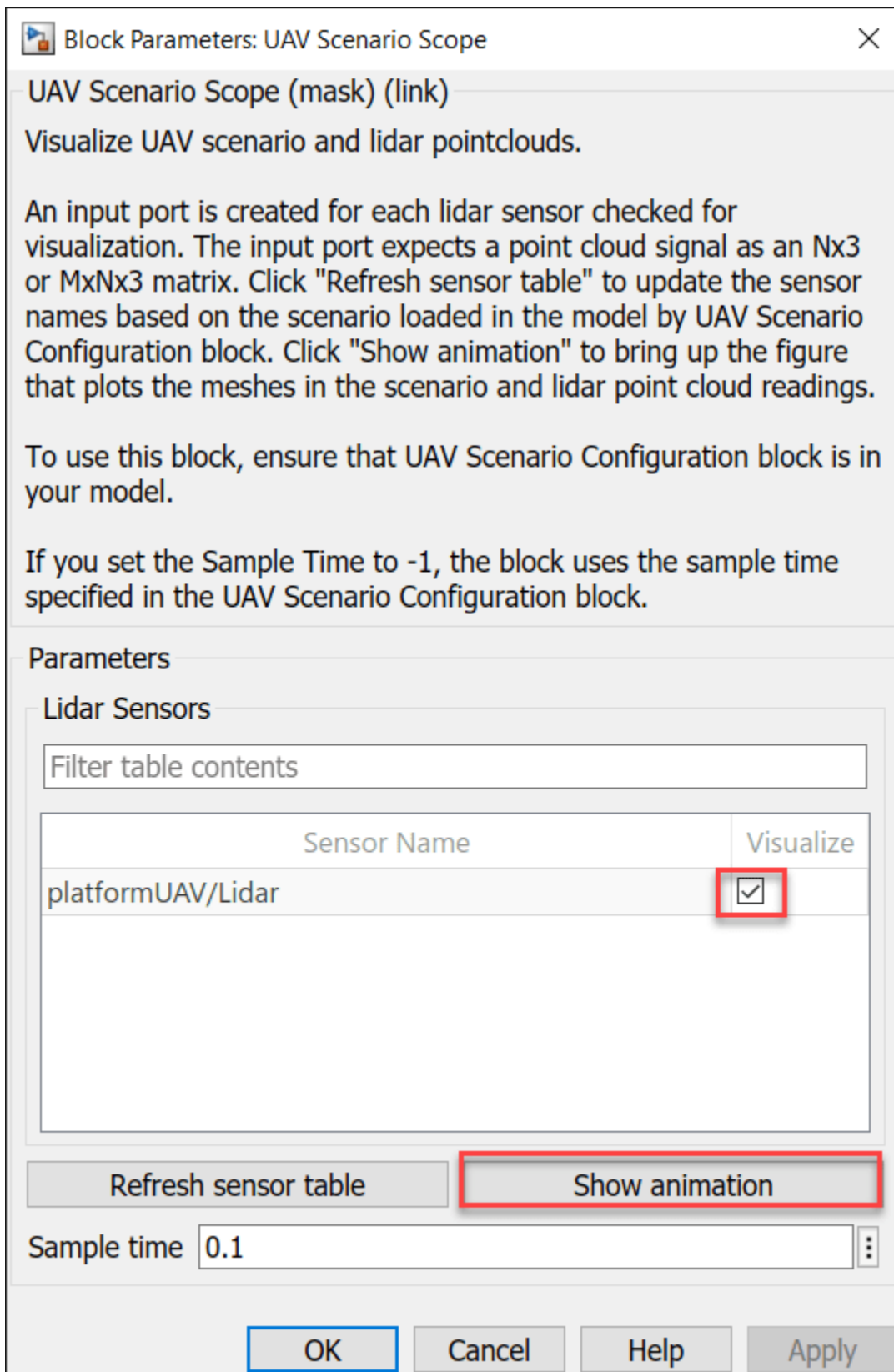
Enabling visualization of the sensor enables a port on the UAV Scenario Scope block with the name of the platform and its sensor.
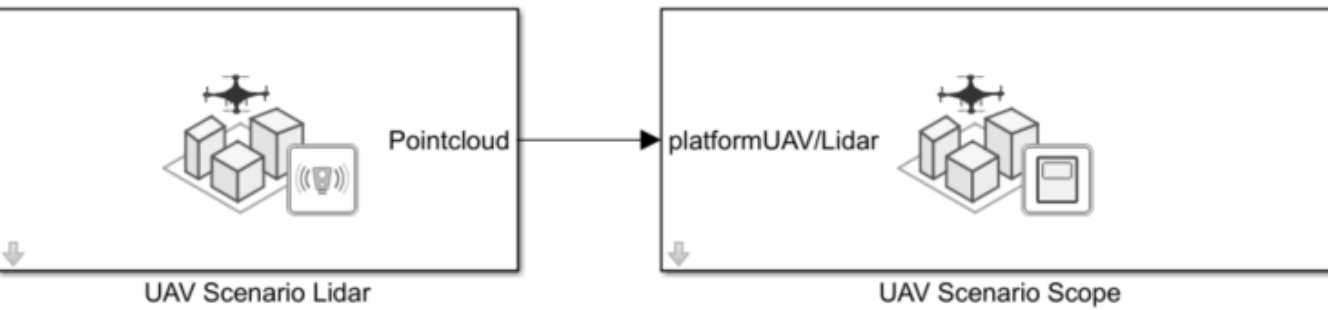
Select **Show animation** in the UAV Scenario Scope block to view the scenario. Run the model to see the animation.
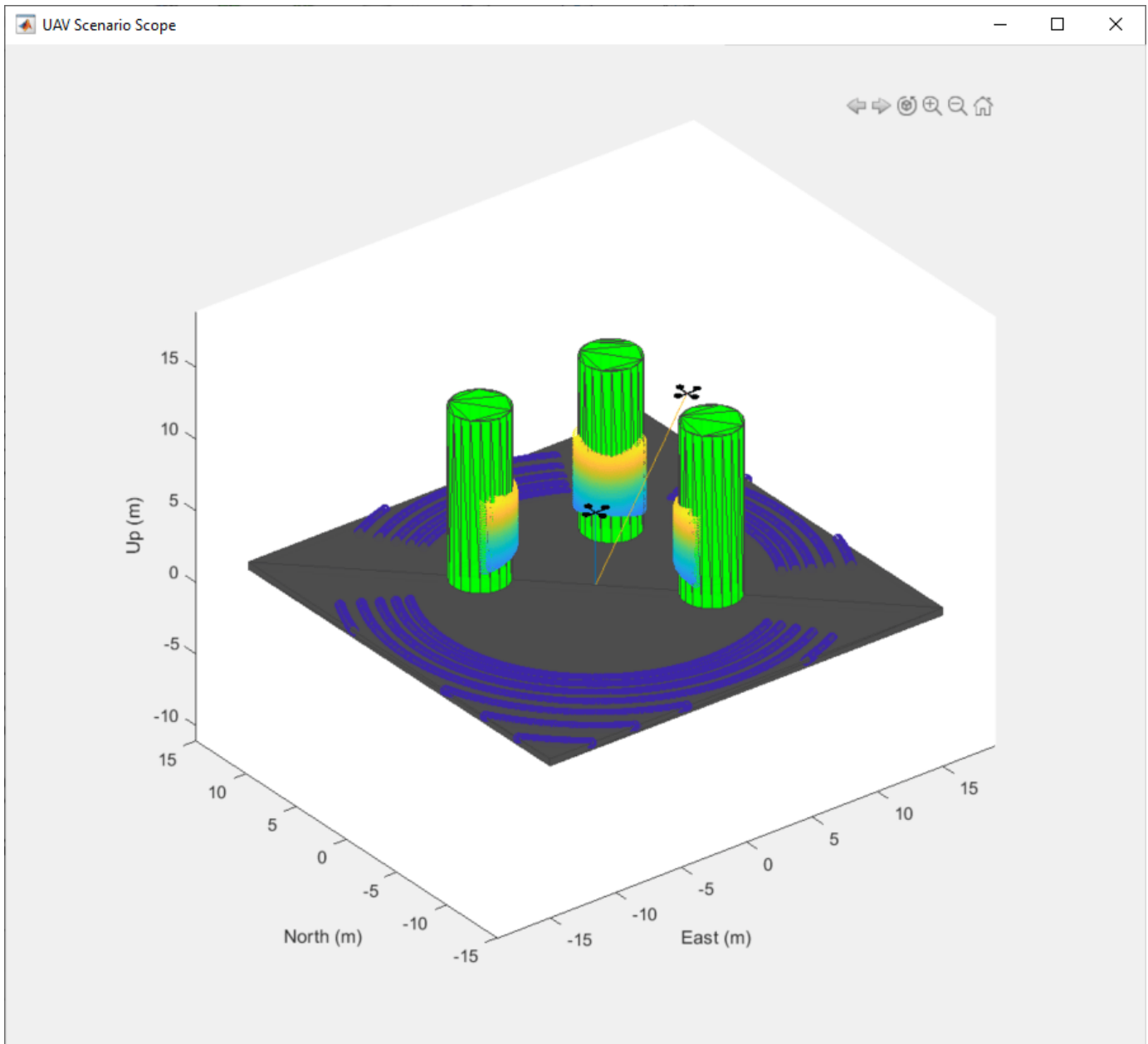
Simulate INS Block

In this example, you simulate an INS block by using the pose information of a vehicle undertaking a left-turn trajectory.

Load Vehicle Trajectory Data

First, you load the trajectory information of the vehicle to the workspace.

```
load leftTurnTrajectory.mat
```

You notice that seven new variables appear in MATLAB workspace.

- `dt` — The time step size of 0.4 seconds.
- `t` — The total time span of 7.88 seconds.
- `vehPos`, `vehVel`, `vehAcc`, `vehOrient`, `vehAngVel` — The history of position, velocity, acceleration, orientation, and angular velocity, each specified as a 198-by-3 matrix, where 198 is the total number of steps.

Open Simulink Model

Next, you open the Simulink model.

```
open simulateINS.slx
```
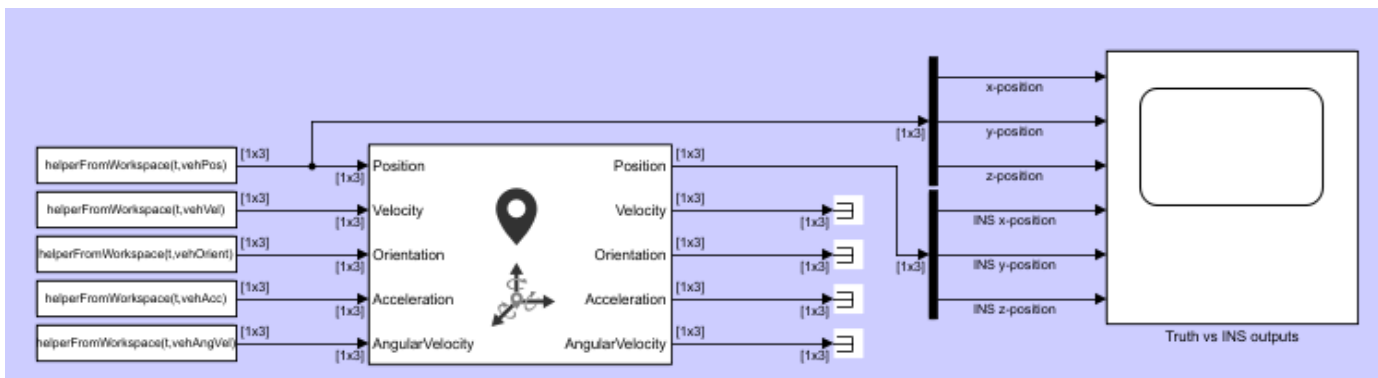

The model contains three parts: the data importing part, the INS block, and the scope block to compare the true positions with the INS outputs.

The data importing part imports the vehicle trajectory data into Simulink using the From Workspace (Simulink) block. You use a helper function `helperFromWorkspace`, attached in the example folder, to convert the trajectory data into a structure format required by the From Workspace block.

Run the Model

Run the Simulink model.

```
resulsts = sim('simulateINS');
```

Click on the scope block and see the results. The INS block position outputs closely follow the truth with the addition of noise.
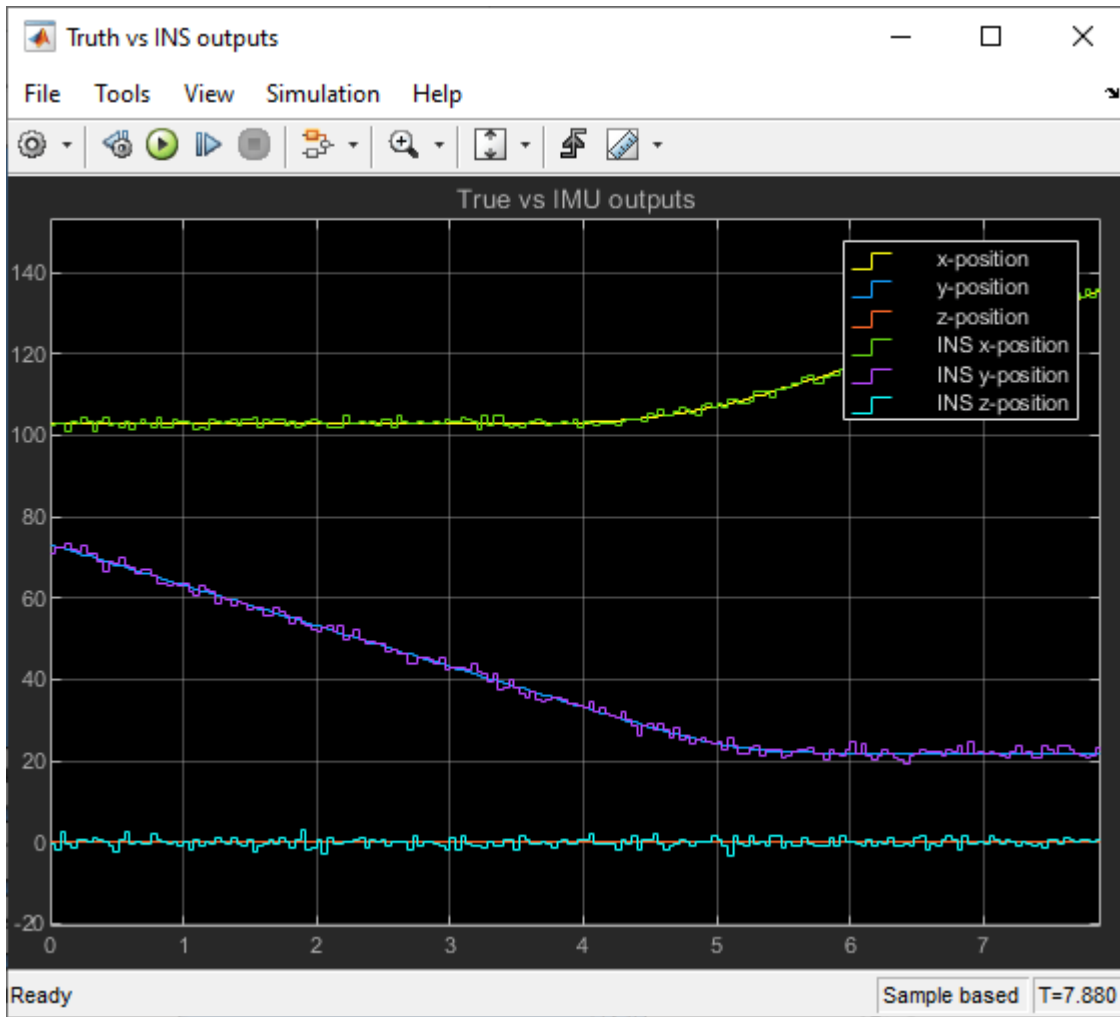
Lidar and Radar Fusion in an Urban Air Mobility Scenario

In this example you learn how to use multi-object trackers to track various Unmanned Air Vehicles (UAVs) in an urban environment. You create a scene using the `uavScenario` object based on building and terrain data available online. You use lidar and radar sensor models to generate synthetic sensor data. Finally, you use various tracking algorithms to estimate the state of all UAVs in the scene.

UAVs are designed for a wide range of operations. Many applications are set in urban environments, such as drone package delivery, air taxis, and power line inspection. The safety of these operations becomes critical as the number of applications grows, making controlling the urban airspace a challenge.

Create an urban air mobility scenario

In this example, you use the terrain and building data of Boulder, CO. The Digital Terrain Elevation Data (DTED) file is downloaded from the "SRTM Void Filled" data set available from the U.S. Geological Survey. The building data, saved in `building.mat`, is downloaded from OpenStreetMap available online. You create a UAV scenario using this data.

```
dtedfile = "n39_w106_3arc_v2.dt1";
buildingfile = "building.mat";
scene = createScenario(dtedfile,buildingfile);
```

Next, you add a few UAVs to the scenario.

To model a package delivery operation, you define a trajectory leaving from the roof of a building and flying to another building. The trajectory is composed of three legs. The quadrotor takes off vertically, then flies toward the next delivery destination, and finally lands vertically on the roof.

```
waypointsA = [1895 90 20; 1915 108 35; 1900 115 20];
timeA = [0 25 50];
trajA = waypointTrajectory(waypointsA, "TimeOfArrival", timeA, "ReferenceFrame", "ENU", "AutoBan
uavA = uavPlatform("UAV", scene, "Trajectory", trajA, "ReferenceFrame", "ENU");
updateMesh(uavA, "quadrotor", {5}, [0 1 1], eye(4));
```

You add another UAV to model an air taxi flying by. Its trajectory is linear, slightly descending. You use the `fixedwing` geometry to model a larger UAV that are suitable to transporting people.

```
waypointsB = [1940 120 50; 1800 50 20];
timeB = [0 41];
trajB = waypointTrajectory(waypointsB, "TimeOfArrival", timeB, "ReferenceFrame", "ENU", "AutoBan
uavB = uavPlatform("UAV2", scene, "Trajectory", trajB, "ReferenceFrame", "ENU");
updateMesh(uavB, "fixedwing", {10},  [0 1 1], eye(4));
```

Then you add a quadrotor with a trajectory following the street path. This could represent a UAV inspecting power grid lines for maintenance purposes.

```
waypointsC = [1950 60 35; 1900 60 35; 1890 80 35];
timeC = linspace(0,41,size(waypointsC,1));
trajC = waypointTrajectory(waypointsC, "TimeOfArrival", timeC, "ReferenceFrame", "ENU", "AutoBan
uavC = uavPlatform("UAV3", scene, "Trajectory", trajC, "ReferenceFrame", "ENU");
updateMesh(uavC, "quadrotor", {5}, [0 1 1], eye(4));
```

Finally, you add the ego UAV, a UAV responsible for surveilling the scene and tracking different moving platforms.

```
waypointsD = [1900 140 65; 1910 100 65];
timeD = [0 60];
trajD = waypointTrajectory(waypointsD, "TimeOfArrival", timeD, ...
    "ReferenceFrame", "ENU", "AutoBank", true, "AutoPitch", true);
egoUAV = uavPlatform("EgoVehicle", scene, "Trajectory", trajD, "ReferenceFrame", "ENU");
updateMesh(egoUAV, "quadrotor", {5}, [0 0 1], eye(4));
```

Define UAV sensor suite

Mount sensors on the ego vehicle. You use a lidar puck that is commonly used in automotive [1]. It is a small sensor that can be attached on a quadrotor. You use the following specification for the lidar puck:

- Range resolution: 3 cm

- Maximum range: 100 m

- 360 degrees azimuth span with 0.2° resolution

- 30 degrees elevation span with 2° resolution

- Update rate: 10 Hz

- Mount with a 90° tilt to look down

```
% Mount a lidar on the quadrotor
lidarOrient = [90 90 0];
lidarSensor = uavLidarPointCloudGenerator("MaxRange",100, ...
    "RangeAccuracy", 0.03, ...
    "ElevationLimits", [-15 15], ...
    "ElevationResolution", 2, ...
    "AzimuthLimits", [-180 180], ...
    "AzimuthResolution", 0.2, ...
    "UpdateRate", 10, ...
    "HasOrganizedOutput", false);
lidar = uavSensor("Lidar", egoUAV, lidarSensor, "MountingLocation", [0 0 -3], "MountingAngles",l:
```

Next you add a radar using the `radarDataGenerator` System object from the Radar Toolbox. To add this sensor to the UAV platform, you need to define a custom adaptor class. The details are shown in the "Simulate Radar Sensor Mounted On UAV" on page 1-95 example. In this example, you use the `helperRadarAdaptor` class. This class uses the mesh geometry of targets to define cuboid dimensions for the radar model. The mesh is also used to derive a simple RCS signature for each target. Inspired from the Echodyne EchoFlight UAV radar [2], the radar configuration is selected as:

- Frequency: 24.45 - 24.65 GHz

- Field of view: 120° azimuth 80° elevation

- Resolution: 2 deg in azimuth, 6° in elevation

- Full scan rate: 1 Hz

- Sensitivity: 0 dBsm at 200 m

Additionally, you configure the radar to output multiple detections per object. Though the radar can output tracks representing point targets, you want to estimate the extent of the target, which is not available with the default track output. Therefore, you set the `TargetReportFormat` property to `Detections` so that the radar report crude detections directly.

```
% Mount a radar on the quadrotor.
radarSensor = radarDataGenerator("no scanning","SensorIndex",1,...
    "FieldOfView",[120 80],...
```

```
    "UpdateRate", 1,...
    'MountingAngles',[0 30 0],...
    "HasElevation", true,...
    "ElevationResolution", 6,...
    "AzimuthResolution", 2, ...
    "RangeResolution", 4, ...
    "RangeLimits", [0 200],...
    'ReferenceRange',200,...
    'CenterFrequency',24.55e9,...
    'Bandwidth',200e6,...
    "TargetReportFormat","Detections",...
    "DetectionCoordinates","Sensor rectangular",...
    "HasFalseAlarms",false,...
    "FalseAlarmRate", 1e-7);

radarcov = coverageConfig(radarSensor);
radar = uavSensor("Radar",egoUAV,helperRadarAdaptor(radarSensor));
```

Define tracking system

Lidar point cloud processing

Lidar sensors return point clouds. To fuse the lidar output, the point cloud must be clustered to extract object detections. Segment out the terrain using the `segmentGroundSMRF` function from Lidar Toolbox. The remaining point cloud is clustered, and a simple threshold is applied to each cluster mean elevation to filter out building detections. Fit each cluster with a cuboid to extract a bounding box detection. The helper class `helperLidarDetector` available in this example has the implementation details.

Lidar cuboid detections are formatted using the `objectDetection` object. The measurement state for these detections is $[x, y, z, L, W, H, q_0, q_1, q_2, q_3]$, where:

- x, y, z are the cuboid center coordinates along the East, North, and Up axes of the scenario, respectively.
- L, W, H are the length, width, and height of the cuboid, respectively.
- $q = q_0 + q_1 . i + q_2 . j + q_3 . k$ is the quaternion defining the orientation of the cuboid with respect to the ENU axes.

```
lidarDetector = helperLidarDetector(scene)

lidarDetector =
  helperLidarDetector with properties:

           MaxWindowRadius: 3
            GridResolution: 1.5000
    SegmentationMinDistance: 5
    MinDetectionsPerCluster: 2
         MinZDistanceCluster: 20
           EgoVehicleRadius: 10
```

Lidar tracker

You use a point target tracker, `trackerJPDA`, to track the lidar bounding box detections. A point tracker assumes that each UAV can generate at most one detection per sensor scan. This assumption is valid because you have clustered the point cloud into cuboids. To setup a tracker, you need to

define the motion model and the measurement model. In this example, you model the dynamics of UAVs using an augmented constant velocity model. The constant velocity model is sufficient to track trajectories consisting of straight flight legs or slowly varying segments. Moreover, assume the orientation of the UAV is constant and assume the dimensions of the UAVs are constant. As a result, the track state and state transition equations are $X = \begin{bmatrix} x, v_x, y, v_y, z, v_z, L, W, H, q_0, q_1, q_2, q_3 \end{bmatrix}$ and

$$
X_{k+1} = \begin{bmatrix} \begin{bmatrix} 1 & t_s & 0 & . & . & 0 \\ 0 & 1 & 0 & . & & . \\ . & . & 1 & t_s & . & . \\ . & & . & 1 & 0 & 0 \\ . & & & . & 1 & t_s \\ 0 & . & . & . & 0 & 1 \end{bmatrix} & 0_3 & 0_3 \\ 0_3 & I_3 & 0_{3x4} \\ 0_3 & 0_{4x3} & I_4 \end{bmatrix} X_k + Q_k
$$

Here, v_x, v_y, v_z are the cuboid velocity vector coordinates along the scenario ENU axes. You track orientation using a quaternion because of the discontinuity of Euler angles when using tracking filters. t_s, the time interval between updates k and k+1, is equal to 0.1 seconds. Lastly, Q_k is the additive process noise that captures the modeling inaccuracy.

The inner transition matrix corresponds to the constant velocity model. You define an augmented state version of `constvel` and `cvmeas` to account for the additional constant states. The details are implemented in the supporting functions `initLidarFilter`, `augmentedConstvel`, `augmentedConstvelJac`, `augmentedCVmeas`, and `augmentedCVMeasJac` in the end of the example.

```
lidarJPDA = trackerJPDA('TrackerIndex',2,...
    'AssignmentThreshold',[70 150],...
    'ClutterDensity',1e-16,...
    'DetectionProbability',0.99,...
    'DeletionThreshold',[10 10],... Delete lidar track if missed for 1 second
    'ConfirmationThreshold',[4 5],...
    'FilterInitializationFcn',@initLidarFilter)

lidarJPDA =
  trackerJPDA with properties:

               TrackerIndex: 2
      FilterInitializationFcn: @initLidarFilter
               MaxNumEvents: Inf
          EventGenerationFcn: 'jpdaEvents'
               MaxNumTracks: 100
           MaxNumDetections: Inf
              MaxNumSensors: 20
              TimeTolerance: 1.0000e-05

              OOSMHandling: 'Terminate'

          AssignmentThreshold: [70 150]
       InitializationThreshold: 0
          DetectionProbability: 0.9900
               ClutterDensity: 1.0000e-16
```

```
                TrackLogic: 'History'
     ConfirmationThreshold: [4 5]
         DeletionThreshold: [10 10]
          HitMissThreshold: 0.2000

         HasCostMatrixInput: false
 HasDetectableTrackIDsInput: false
            StateParameters: [1×1 struct]

                  NumTracks: 0
         NumConfirmedTracks: 0
```

Radar tracker

In this example, you assume the radar returns are preprocessed such that only returns from moving objects are preserved, that is no returns from the ground or the buildings. The radar measurement state is $[x, v_x, y, v_y, z, v_z]$. The radar resolution is fine enough to generate multiple returns per UAV target and its detections should not be fed directly to a point target tracker. There are two possible approaches to track with the high-resolution radar detections. One way, you can cluster the detections and augment the state with dimensions and orientation constants as done previously with the lidar cuboids. On the other way, you can feed the detections to an extended target tracker adopted in this example by using a GGIW-PHD tracker. This tracker estimates the extent of each target using an inverse Wishart distribution, whose expectation is a 3-by-3 positive definite matrix, representing the extent of a target as a 3D ellipse. This second approach is preferable because there aren't too many detections per object and clustering is less accurate than extended-target tracking

To create a GGIW-PHD tracker, you first define the tracking sensor configuration for each sensor reporting to the tracker. In this case, you only need to define the configuration for one radar. When the radar mounting platform is moving, you need to update this configuration with the current radar pose before each tracker step. Next, you define a filter initialization function based on the sensor configuration. Finally, you construct a `trackerPHD` object and increase the partitioning threshold to capture the dimensions of objects tracked in this example. The implementation details are shown at the end of the example in the supporting function `createRadarTracker`.

```
radarPHD = createRadarTracker(radarSensor, egoUAV)

radarPHD =
  trackerPHD with properties:

                  TrackerIndex: 1
          SensorConfigurations: {[1×1 trackingSensorConfiguration]}
                PartitioningFcn: @(dets)partitionDetections(dets,threshold(1),threshold(2),'Dist
                  MaxNumSensors: 20
                   MaxNumTracks: 1000

           AssignmentThreshold: 50
                     BirthRate: 1.0000e-03
                     DeathRate: 1.0000e-06

           ExtractionThreshold: 0.8000
         ConfirmationThreshold: 0.9900
             DeletionThreshold: 0.1000
              MergingThreshold: 50
            LabelingThresholds: [1.0100 0.0100 0]
```

```
               StateParameters: [1×1 struct]
    HasSensorConfigurationsInput: true
                      NumTracks: 0
             NumConfirmedTracks: 0
```

Track fusion

The final step in creating the tracking system is to define a track fuser object to fuse lidar tracks and radar tracks. You use the 13-dimensional state of lidar tracks as the fused state definition.

```
radarConfig = fuserSourceConfiguration('SourceIndex',1,...
    'IsInitializingCentralTracks',true);

lidarConfig = fuserSourceConfiguration('SourceIndex',2,...
    'IsInitializingCentralTracks',true);

fuser = trackFuser('SourceConfigurations',{radarConfig,lidarConfig},...
    'ProcessNoise',blkdiag(2*eye(6),1*eye(3),0.2*eye(4)),...
    'HasAdditiveProcessNoise',true,...
    'AssignmentThreshold',200,...
    'ConfirmationThreshold',[4 5],...
    'DeletionThreshold',[5 5],...
    'StateFusion','Cross',...
    'StateTransitionFcn',@augmentedConstvel,...
    'StateTransitionJacobianFcn',@augmentedConstvelJac);
```

Visualization

You use a helper class to visualize the scenario. This helper class utilizes the `uavScenario` visualization capabilities and the `theaterPlot` plotter to represent detection and track information.

The display is divided into 5 tiles, showing respectively, the overall 3D scene, three chase cameras for three UAVs, and the legend.

```
viewer = helperUAVDisplay(scene);
```
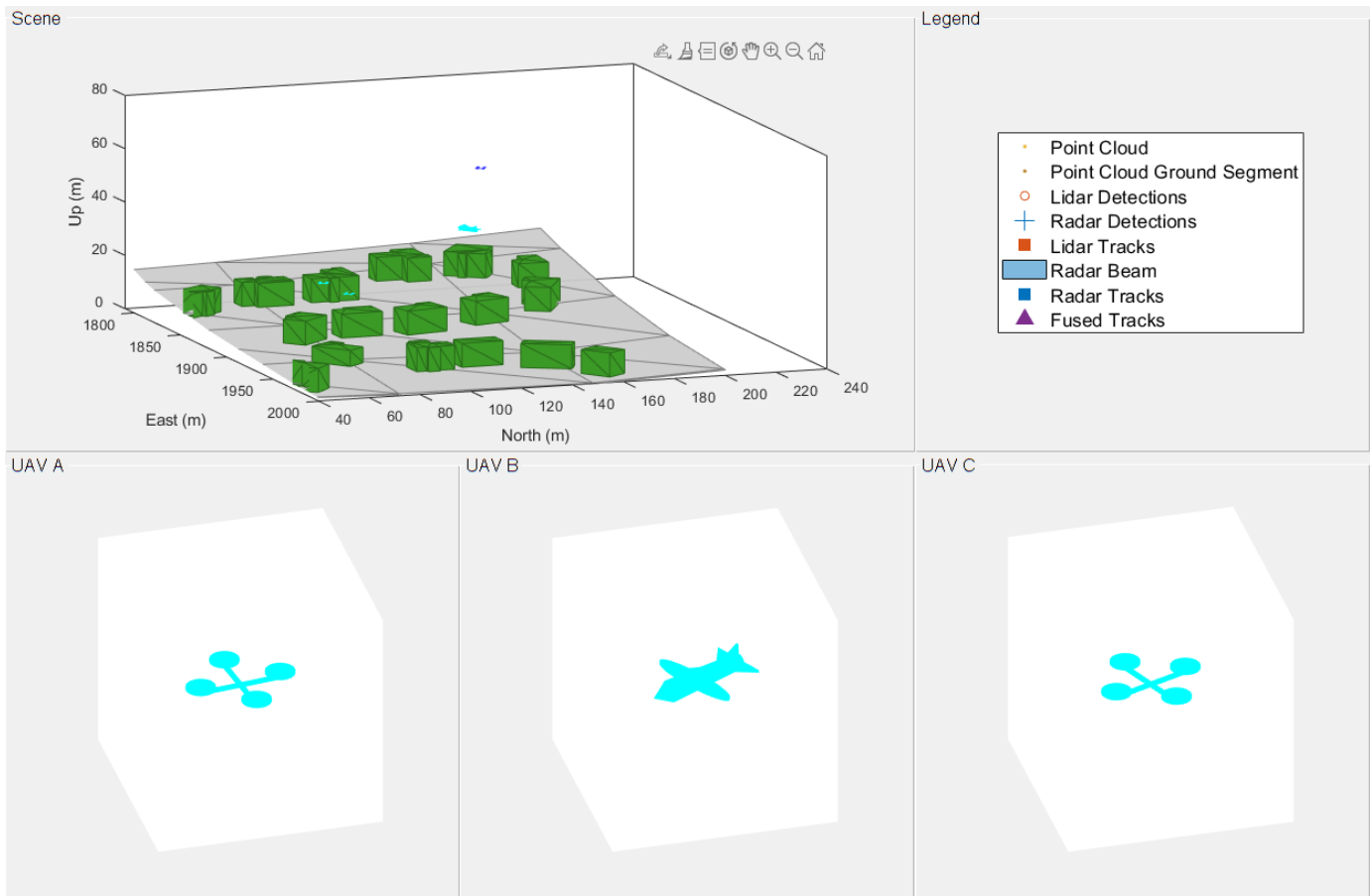
Simulate the scenario

You run the scenario and visualize the results of the tracking system. The true pose of each target as well as the radar, lidar, and fused tracks are saved for off-line metric analysis.

```
setup(scene);
s = rng;
rng(2021);

numSteps = scene.StopTime*scene.UpdateRate;
truthlog = cell(1,numSteps);
radarlog = cell(1,numSteps);
lidarlog = cell(1,numSteps);
fusedlog = cell(1,numSteps);
logCount = 0;

while advance(scene)
    time = scene.CurrentTime;
    % Update sensor readings and read data.
    updateSensors(scene);
    egoPose = read(egoUAV);

    % Track with radar
    [radardets, radarTracks, inforadar] = updateRadarTracker(radar,radarPHD, egoPose, time);
```

```
% Track with lidar
[lidardets, lidarTracks, nonGroundCloud, groundCloud] = updateLidarTracker(lidar,lidarDetecto

% Fuse lidar and radar tracks
rectRadarTracks = formatPHDTracks(radarTracks);
if isLocked(fuser) || ~isempty(radarTracks) || ~isempty(lidarTracks)
    [fusedTracks,~,allfused,info] = fuser([lidarTracks;rectRadarTracks],time);
else
    fusedTracks = objectTrack.empty;
end

% Save log
logCount = logCount + 1;
lidarlog{logCount} = lidarTracks;
radarlog{logCount} = rectRadarTracks;
fusedlog{logCount} = fusedTracks;
truthlog{logCount} = logTargetTruth(scene.Platforms(1:3));

% Update figure
viewer(radarcov, nonGroundCloud, groundCloud, lidardets, radardets, lidarTracks, radarTracks,
end
```
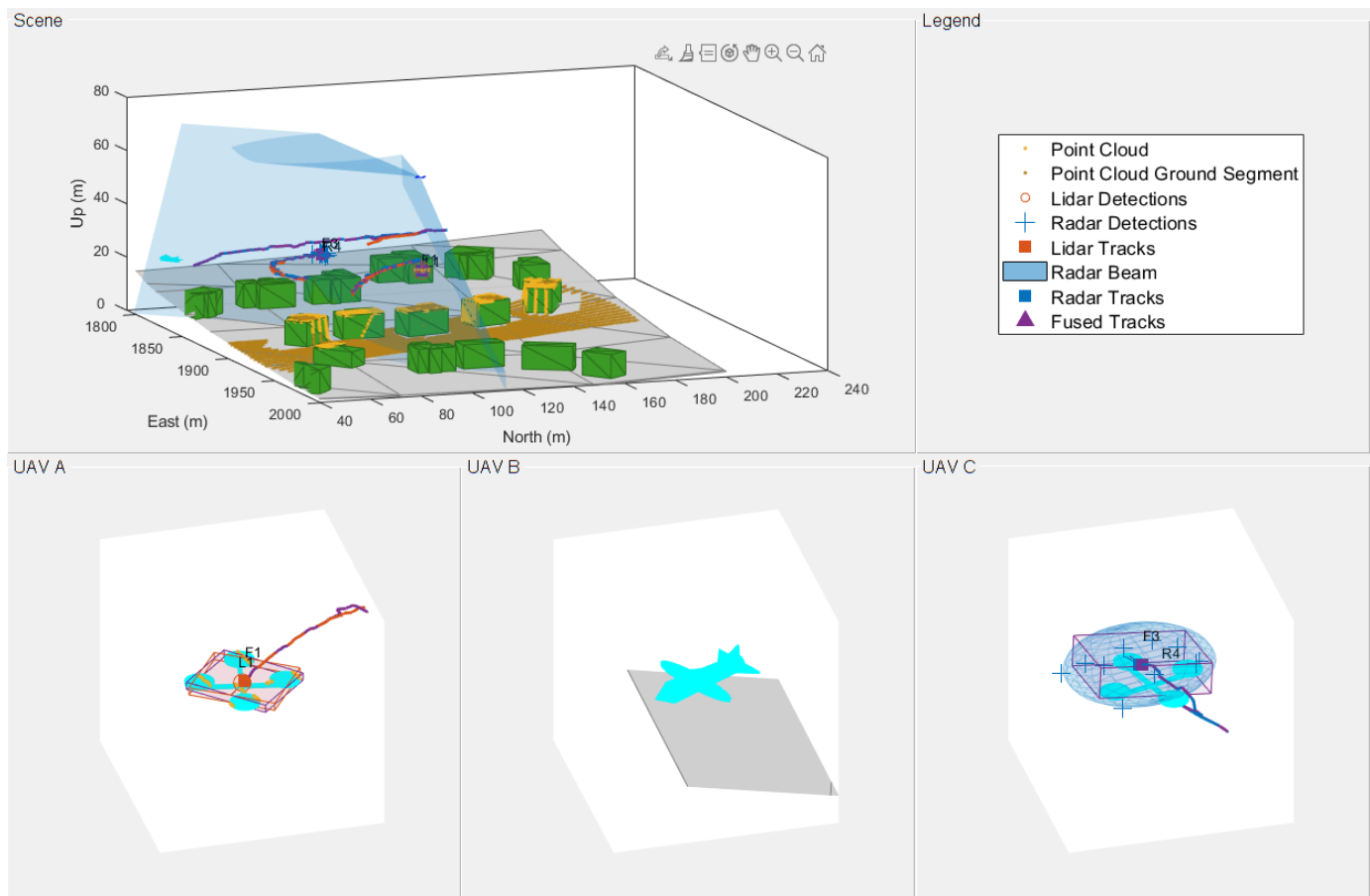

Based on the visualization results, you perform an initial qualitative assessment of the tracking performance. The display at the end of the scenario shows that all three UAVs were well tracked by the ego. With the current sensor suite configuration, lidar tracks were only established partially due

to the limited coverage of the lidar sensor. The wider field of view of the radar allowed establishing radar tracks more consistently in this scenario.
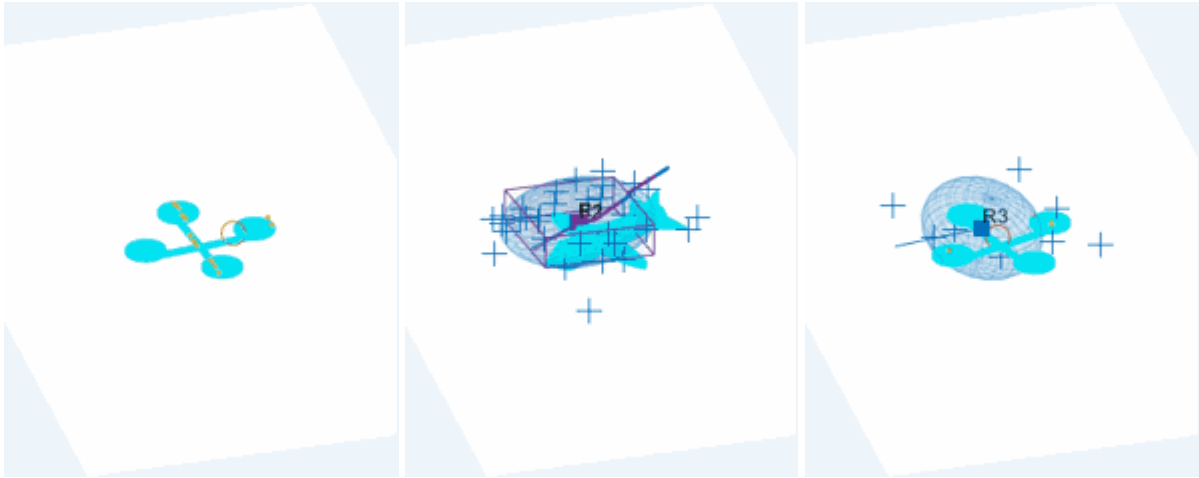
The three animated GIFs above show parts of the chase views. You can see that the quality of lidar tracks (orange box) is affected by the geometry of the scenario. UAV A (left) is illuminated by the lidar (shown in yellow) almost directly from above. This enables the tracker to capture the full extent of the drone over time. However, UAV C (right) is partially seen by the radar which leads to underestimating the size of the drone. Also, the estimated centroid periodically oscillates around the true drone center. The larger fixed-wing UAV (middle) generates many lidar points. Thus, the tracker can detect and track the full extent of the target once it has completely entered the field of view of the lidar. In all three cases, the radar, shown in blue, provides more accurate information of the target extent. As a result, the fused track box (in purple) is more closely capturing the extent of each UAV. However, the radar returns are less accurate in position. Radar tracks show more position bias and poorer orientation estimate.

Tracking metrics

In this section you analyze the performance of the tracking system using the GOSPA tracking metric. You first define the distance function which quantifies the error between track and truth using a scalar value. A lower GOSPA value means an overall better performance.

```
gospaR = trackGOSPAMetric('Distance','custom','DistanceFcn',@metricDistance);
gospaL = clone(gospaR);
gospaF = clone(gospaR);

gospaRadar = zeros(1,numSteps);
gospaLidar = zeros(1,numSteps);
gospaFused = zeros(1,numSteps);

for i=1:numSteps
    truth = truthlog{i};
    gospaRadar(i) = gospaR(radarlog{i},truth);
    gospaLidar(i) = gospaL(lidarlog{i},truth);
    gospaFused(i) = gospaF(fusedlog{i},truth);
end

figure
plot(gospaRadar,'Color',viewer.RadarColor,'LineWidth',2);
hold on
```

```
grid on
plot(gospaLidar,'Color',viewer.LidarColor,'LineWidth',2);
plot(gospaFused,'Color',viewer.FusedColor,'LineWidth',2);
legend('Radar','Lidar','Lidar + Radar');
xlabel('Steps')
ylabel('GOSPA')
```
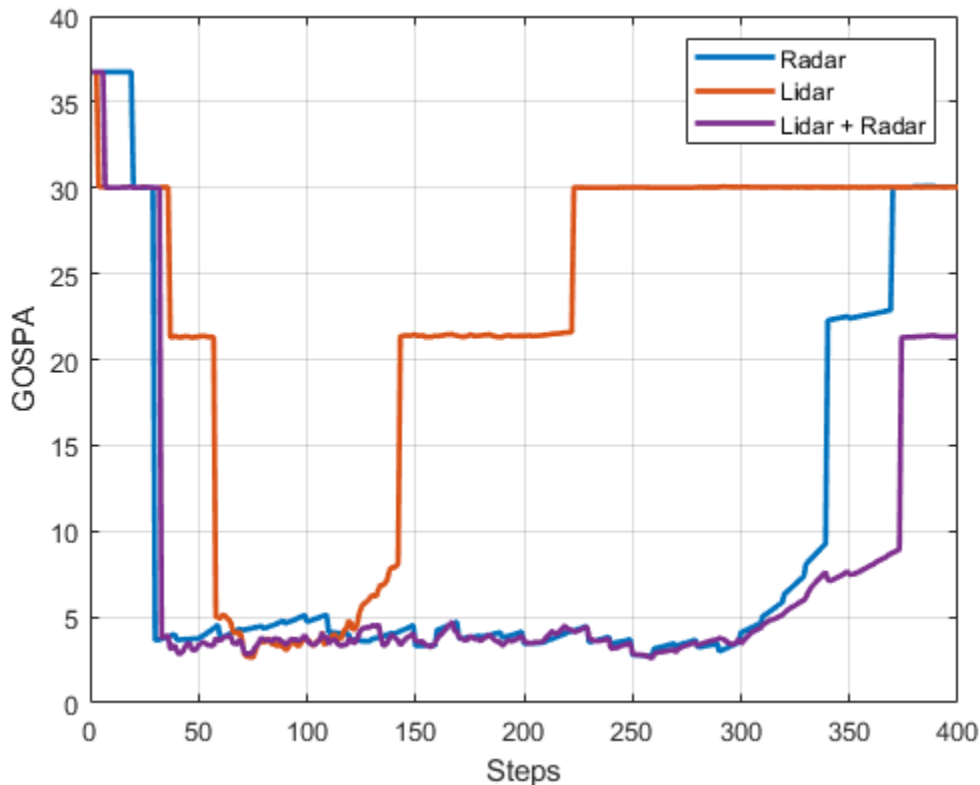

You analyze the overall system performance. Each tracker is penalized for not tracking any of the UAVs even if the target UAV is outside of the sensor coverage. This shows improved performance when fusing lidar and radar due to the added surveillance area. This is particularly noticeable at the end of the simulation where two targets are tracked, one by radar and the other by lidar, but both are tracked by the fuser. Additionally, you can see that the fused GOSPA is below the minimum of lidar and radar GOSPA, showing the fused track has better quality than each individual track.

```
% clean up
removeCustomTerrain("southboulder");
rng(s);
```

Summary

In this example, you have learned how to model a UAV-borne lidar and radar tracking system and tested it on an urban air mobility scenario. You used the `uavScenario` object to create a realistic urban environment with terrain and buildings. You generated synthetic sensor data to test a complete tracking system chain, involving point cloud processing, point target and extended target tracking, and track fusion.

Supporting Functions

createScenario creates the `uavScenario` using the OpenStreetMap terrain and building mesh data

```
function scene = createScenario(dtedfile,buildingfile)

try
    addCustomTerrain("southboulder",dtedfile);
catch
    % custom terrain was already added.
end
load(buildingfile,'buildings');

minHeight = 1.6925e+03;
latlonCenter = [39.9786 -105.2882 minHeight];
scene = uavScenario("UpdateRate",10,"StopTime",40,...
    "ReferenceLocation",latlonCenter);

% Add terrain mesh
sceneXLim = [1800 2000];
sceneYLim = [0 200];
scene.addMesh("terrain", {"southboulder", sceneXLim, sceneYLim},[0 0 0]);

% Add buildings
for idx = 1:numel(buildings)-1
    v = buildings{idx}.Vertices;
    v(:,3) = v(:,3) - minHeight;
    rangeVMin = min(v);
    rangeVMax = max(v);
    if rangeVMin(1) > sceneXLim(1) && rangeVMax(1) < sceneXLim(2) &&...
            rangeVMin(2) > sceneYLim(1) && rangeVMax(2) < sceneYLim(2)
        scene.addMesh("custom", {v, buildings{idx}.Faces},[0 0 0]);
    end
end

end
```

createRadarTracker creates the trackerPHD tracker to fuse radar detections.

```
function tracker = createRadarTracker(radar, egoUAV)

% Create sensor configuration for trackerPHD
fov = radar.FieldOfView;
sensorLimits = [-fov(1)/2 fov(1)/2; -fov(2)/2 fov(2)/2; 0 inf];
sensorResolution = [radar.AzimuthResolution;radar.ElevationResolution; radar.RangeResolution];
Kc = radar.FalseAlarmRate/(radar.AzimuthResolution*radar.RangeResolution*radar.ElevationResolutio
Pd = radar.DetectionProbability;

sensorPos = radar.MountingLocation(:);
sensorOrient = rotmat(quaternion(radar.MountingAngles, 'eulerd', 'ZYX', 'frame'),'frame');

% Specify frame info of radar with respect to UAV
sensorTransformParameters(1) = struct('Frame','Spherical',...
    'OriginPosition', sensorPos,...
    'OriginVelocity', zeros(3,1),...% Sensor does not move relative to ego
    'Orientation', sensorOrient,...
    'IsParentToChild',true,...% Frame rotation is supplied as orientation
```

```
        'HasElevation',true,...
        'HasVelocity',false);

    % Specify frame info of UAV with respect to scene
    egoPose = read(egoUAV);
    sensorTransformParameters(2) = struct('Frame','Rectangular',...
        'OriginPosition', egoPose(1:3),...
        'OriginVelocity', egoPose(4:6),...
        'Orientation', rotmat(quaternion(egoPose(10:13)),'Frame'),...
        'IsParentToChild',true,...
        'HasElevation',true,...
        'HasVelocity',false);

    radarPHDconfig = trackingSensorConfiguration(radar.SensorIndex,...
        'IsValidTime', true,...
        'SensorLimits',sensorLimits,...
        'SensorResolution', sensorResolution,...
        'DetectionProbability',Pd,...
        'ClutterDensity', Kc,...
        'SensorTransformFcn',@cvmeas,...
        'SensorTransformParameters', sensorTransformParameters);

    radarPHDconfig.FilterInitializationFcn = @initRadarFilter;

    radarPHDconfig.MinDetectionProbability = 0.4;

    % Threshold for partitioning
    threshold = [3 16];
    tracker = trackerPHD('TrackerIndex',1,...
        'HasSensorConfigurationsInput',true,...
        'SensorConfigurations',{radarPHDconfig},...
        'BirthRate',1e-3,...
        'AssignmentThreshold',50,...% Minimum negative log-likelihood of a detection cell to add bir
        'ExtractionThreshold',0.80,...% Weight threshold of a filter component to be declared a track
        'ConfirmationThreshold',0.99,...% Weight threshold of a filter component to be declared a co
        'MergingThreshold',50,...% Threshold to merge components
        'DeletionThreshold',0.1,...% Threshold to delete components
        'LabelingThresholds',[1.01 0.01 0],...% This translates to no track-splitting. Read LabelingT
        'PartitioningFcn',@(dets) partitionDetections(dets, threshold(1),threshold(2),'Distance','eu
end
```

initRadarfilter implements the GGIW-PHD filter used by the `trackerPHD` object. This filter is used during a tracker update to 1) initialize new birth components in the density and 2) initialize new component from detection partitions.

```
function phd = initRadarFilter (detectionPartition)

if nargin == 0

    % Process noise
    sigP = 0.2;
    sigV = 1;
    Q = diag([sigP, sigV, sigP, sigV, sigP, sigV].^2);

    phd = ggiwphd(zeros(6,0),repmat(eye(6),[1 1 0]),...
        'ScaleMatrices',zeros(3,3,0),...
        'MaxNumComponents',1000,...
        'ProcessNoise',Q,...
```

```matlab
        'HasAdditiveProcessNoise',true,...
        'MeasurementFcn', @cvmeas,...
        'MeasurementJacobianFcn', @cvmeasjac,...
        'PositionIndex', [1 3 5],...
        'ExtentRotationFcn', @(x,dT)eye(3,class(x)),...
        'HasAdditiveMeasurementNoise', true,...
        'StateTransitionFcn', @constvel,...
        'StateTransitionJacobianFcn', @constveljac);

else %nargin == 1
    % ------------------
    % 1) Configure Gaussian mixture
    % 2) Configure Inverse Wishart mixture
    % 3) Configure Gamma mixture
    % ----------------

    %% 1) Configure Gaussian mixture
    meanDetection = detectionPartition{1};
    n = numel(detectionPartition);

    % Collect all measurements and measurement noises.
    allDets = [detectionPartition{:}];
    zAll = horzcat(allDets.Measurement);
    RAll = cat(3,allDets.MeasurementNoise);

    % Specify mean noise and measurement
    z = mean(zAll,2);
    R = mean(RAll,3);
    meanDetection.Measurement = z;
    meanDetection.MeasurementNoise = R;

    % Parse mean detection for position and velocity covariance.
    [posMeas,velMeas,posCov] = matlabshared.tracking.internal.fusion.parseDetectionForInitFcn(mea

    % Create a constant velocity state and covariance
    states = zeros(6,1);
    covariances = zeros(6,6);
    states(1:2:end) = posMeas;
    states(2:2:end) = velMeas;
    covariances(1:2:end,1:2:end) = posCov;
    covariances(2:2:end,2:2:end) = 10*eye(3);

    % process noise
    sigP = 0.2;
    sigV = 1;
    Q = diag([sigP, sigV, sigP, sigV, sigP, sigV].^2);

    %% 2) Configure Inverse Wishart mixture parameters
    % The extent is set to the spread of the measurements in positional-space.
    e = zAll - z;
    Z = e*e'/n + R;
    dof = 150;
    % Measurement Jacobian
    p = detectionPartition{1}.MeasurementParameters;
    H = cvmeasjac(states,p);

    Bk = H(:,1:2:end);
    Bk2 = eye(3)/Bk;
```

```matlab
        V = (dof-4)*Bk2*Z*Bk2';

        % Configure Gamma mixture parameters such that the standard deviation
        % of the number of detections is n/4
        alpha = 16; % shape
        beta = 16/n; % rate

        phd = ggiwphd(...
            ... Gaussian parameters
            states,covariances,...
            'HasAdditiveMeasurementNoise' ,true,...
            'ProcessNoise',Q,...
            'HasAdditiveProcessNoise',true,...
            'MeasurementFcn', @cvmeas,...
            'MeasurementJacobianFcn' , @cvmeasjac,...
            'StateTransitionFcn', @constvel,...
            'StateTransitionJacobianFcn' , @constveljac,...
            'PositionIndex'   ,[1 3 5],...
            'ExtentRotationFcn' , @(x,dT) eye(3),...
            ... Inverse Wishart parameters
            'DegreesOfFreedom',dof,...
            'ScaleMatrices',V,...
            'TemporalDecay',150,...
            ... Gamma parameters
            'Shapes',alpha,'Rates',beta,...
            'GammaForgettingFactors',1.05);
    end

end
```

formatPHDTracks formats the elliptical GGIW-PHD tracks into rectangular augmented state tracks for track fusion. **convertExtendedTrack** returns state and state covariance of augmented rectangular state. The Inverse Wishart random matrix eigen values are used to derive rectangular box dimensions. The eigen vectors provide with the orientation quaternion. In this example, you use an arbitrary covariance for radar track dimension and orientation, which is often sufficient for tracking.

```matlab
function tracksout = formatPHDTracks(tracksin)
% Convert track struct from ggiwphd to objectTrack with state definition
% [x y z vx vy vz L W H q0 q1 q2 q3]
N = numel(tracksin);
tracksout = repmat(objectTrack,N,1);
for i=1:N
    tracksout(i) = objectTrack(tracksin(i));
    [state, statecov] = convertExtendedTrack(tracksin(i));
    tracksout(i).State = state;
    tracksout(i).StateCovariance = statecov;
end
end

function [state, statecov] = convertExtendedTrack(track)
% Augment the state with the extent information

extent = track.Extent;
[V,D] = eig(extent);
% Choose L > W > H. Use 1.5 sigma as the dimension
[dims, idx] = sort(1.5*sqrt(diag(D)),'descend');
```

```
V = V(:,idx);
q = quaternion(V,'rotmat','frame');
q = q./norm(q);
[q1, q2, q3, q4] = parts(q);
state = [track.State; dims(:); q1 ; q2 ; q3 ; q4 ];
statecov = blkdiag(track.StateCovariance, 4*eye(3), 4*eye(4));

end
```

updateRadarTracker updates the radar tracking chain. The function first reads the current radar returns. Then the radar returns are passed to the GGIW-PHD tracker after updating its sensor configuration with the current pose of the ego drone.

```
function [radardets, radarTracks, inforadar] = updateRadarTracker(radar,radarPHD, egoPose, time)
[~,~,radardets, ~, ~] = read(radar); % isUpdated and time outputs are not compatible with this wo
inforadar = [];
if mod(time,1) ~= 0
    radardets = {};
end
if mod(time,1) == 0 && (isLocked(radarPHD) || ~isempty(radardets))
    % Update radar sensor configuration for the tracker
    configs = radarPHD.SensorConfigurations;
    configs{1}.SensorTransformParameters(2).OriginPosition = egoPose(1:3);
    configs{1}.SensorTransformParameters(2).OriginVelocity = egoPose(4:6);
    configs{1}.SensorTransformParameters(2).Orientation = rotmat(quaternion(egoPose(10:13)),'fram
    [radarTracks,~,~,inforadar] = radarPHD(radardets,configs,time);
elseif isLocked(radarPHD)
    radarTracks = predictTracksToTime(radarPHD,'confirmed', time);
    radarTracks = arrayfun(@(x) setfield(x,'UpdateTime',time), radarTracks);
else
    radarTracks = objectTrack.empty;
end
end
```

updateLidarTracker updates the lidar tracking chain. The function first reads the current point cloud output from the lidar sensor. Then the point cloud is processed to extract object detections. Finally, these detections are passed to the point target tracker.

```
function [lidardets, lidarTracks,nonGroundCloud, groundCloud] = updateLidarTracker(lidar, lidarDe
[~, time, ptCloud] = read(lidar);
% lidar is always updated
[lidardets,nonGroundCloud, groundCloud] = lidarDetector(egoPose, ptCloud,time);
if isLocked(lidarJPDA) || ~isempty(lidardets)
    lidarTracks = lidarJPDA(lidardets,time);
else
    lidarTracks = objectTrack.empty;
end
end
```

initLidarFilter initializes filter for the lidar tracker. The initial track state is derived from the detection position measurement. Velocity is set to 0 with a large covariance to allow future detections to be associated to the track. Augmented state motion model, measurement functions, and jacobians are also defined below.

```
function ekf = initLidarFilter(detection)

% Lidar measurement: [x y z L W H q0 q1 q2 q3]
meas = detection.Measurement;
```

```
initState = [meas(1);0;meas(2);0;meas(3);0; meas(4:6);meas(7:10) ];
initStateCovariance = blkdiag(100*eye(6), 100*eye(3), eye(4));

% Process noise standard deviations
sigP = 1;
sigV = 2;
sigD = 0.5; % Dimensions are constant but partially observed
sigQ = 0.5;

Q = diag([sigP, sigV, sigP, sigV, sigP, sigV, sigD, sigD, sigD, sigQ, sigQ, sigQ, sigQ].^2);

ekf = trackingEKF('State',initState,...
    'StateCovariance',initStateCovariance,...
    'ProcessNoise',Q,...
    'StateTransitionFcn',@augmentedConstvel,...
    'StateTransitionJacobianFcn',@augmentedConstvelJac,...
    'MeasurementFcn',@augmentedCVmeas,...
    'MeasurementJacobianFcn',@augmentedCVmeasJac);
end

function stateOut = augmentedConstvel(state, dt)
% Augmented state for constant velocity
stateOut = constvel(state(1:6,:),dt);
stateOut = vertcat(stateOut,state(7:end,:));
% Normalize quaternion in the prediction stage
idx = 10:13;
qparts = stateOut(idx);
n = sqrt(sum(qparts.^2));
qparts = qparts./n;
if qparts(1) < 0
    stateOut(idx) = -qparts;
else
    stateOut(idx) = qparts;
end
end

function jacobian = augmentedConstvelJac(state,varargin)
jacobian = constveljac(state(1:6,:),varargin{:});
jacobian = blkdiag(jacobian, eye(7));
end

function measurements = augmentedCVmeas(state)
measurements = cvmeas(state(1:6,:));
measurements = [measurements; state(7:9,:); state(10:13,:)];
end

function jacobian = augmentedCVmeasJac(state,varargin)
jacobian = cvmeasjac(state(1:6,:),varargin{:});
jacobian = blkdiag(jacobian, eye(7));
end
```

logTargetTruth logs true pose and dimensions throughout the simulation for performance analysis.

```
function logEntry = logTargetTruth(targets)
n = numel(targets);
targetPoses = repmat(struct('Position',[],'Velocity',[],'Dimension',[],'Orientation',[]),1,n);
uavDimensions = [5 5 0.3 ; 9.8 8.8 2.8; 5 5 0.3];
```

```matlab
for i=1:n
    pose = read(targets(i));
    targetPoses(i).Position = pose(1:3);
    targetPoses(i).Velocity = pose(4:6);
    targetPoses(i).Dimension = uavDimensions(i,:);
    targetPoses(i).Orientation = pose(10:13);
end
logEntry = targetPoses;
end
```

metricDistance defines a custom distance for GOSPA. This distance incorporates errors in position, velocity, dimension, and orientation of the tracks.

```matlab
function out = metricDistance(track,truth)
positionIdx = [1 3 5];
velIdx = [2 4 6];
dimIdx = 7:9;
qIdx = 10:13;

trackpos = track.State(positionIdx);
trackvel = track.State(velIdx);
trackdim = track.State(dimIdx);
trackq = quaternion(track.State(qIdx)');

truepos = truth.Position;
truevel = truth.Velocity;
truedim = truth.Dimension;
trueq = quaternion(truth.Orientation);

errpos = truepos(:) - trackpos(:);
errvel = truevel(:) - trackvel(:);
errdim = truedim(:) - trackdim(:);

% Weights expressed as inverse of the desired accuracy
posw = 1/0.2; %m^-1
velw = 1/2; % (m/s) ^-1
dimw = 1/4; % m^-1
orw = 1/20; % deg^-1

distPos = sqrt(errpos'*errpos);
distVel = sqrt(errvel'*errvel);
distdim = sqrt(errdim'*errdim);
distq = rad2deg(dist(trackq, trueq));

out = (distPos * posw + distVel * velw + distdim * dimw + distq * orw)/(posw + velw + dimw + orw)
end
```

References

1   Velodyne Lidar puck: https://velodynelidar.com/products/puck/

2   Echodyne UAV radar: https://www.echodyne.com/defense/uav-radar/

Avoid Moving Obstacles Based on Radar Detections

This example demonstrates how to avoid collision with moving obstacles based on the velocity obstacle concept [1].The `uavScenario` used in this example is based on the ""Simulate Radar Sensor Mounted On UAV" on page 1-95" example, which shows how to generate track detections of moving UAVs close to the ego vehicle.

Setup testing scenario

The testing scenario has two UAVs, of which one ego UAV carries a radar sensor and the other UAV acts as a moving obstacle. The radar sensor on the ego vehicle generates target tracks containing target position and velocity information based on detections. The ego UAV can execute avoidance maneuver based on detection information.

```
rng(0) % For repeatable results.

% Create a scenario that runs for 10 seconds.
s = uavScenario("StopTime",30,"HistoryBufferSize",200);

% Create a fixed-wing target that moves from [30 0 0] to [20 10 0].
target = uavPlatform("Target",s,"Trajectory",waypointTrajectory([30 0 0; 0 30 0],"TimeOfArrival"
updateMesh(target,"fixedwing",{1},[1 0 0],eul2tform([0 0 pi]));

% Create a quadrotor as the ego vehicle.
egoVelocity = [1 1 0];
egoMultirotor = uavPlatform("EgoVehicle",s,"InitialVelocity", egoVelocity);
updateMesh(egoMultirotor,"quadrotor",{1},[0 1 0],eul2tform([0 0 pi]));

% Create a radar sensor and set up its properties.
radarSensor = radarDataGenerator("no scanning","SensorIndex",1,"UpdateRate",10,...
    "FieldOfView",[120 80],...
    "HasElevation", true,...
    "ElevationResolution", 3,...
    "AzimuthResolution", 1, ...
    "RangeResolution", 10, ... meters
    "RangeRateResolution",3,...
    "RangeLimits", [0 750],...
    "TargetReportFormat","Tracks",...
    "TrackCoordinates",'Scenario',...
    "HasINS", true,...
    "HasFalseAlarms",true,...
    "FalseAlarmRate",1e-5,...
    "HasRangeRate",true,...
    "FalseAlarmRate", 1e-7);

% Mount the radar sensor on the ego vehicle. ExampleHelperUAVRadar inherits
% from the uav.SensorAdaptor class.
radar = uavSensor("Radar",egoMultirotor,ExampleHelperUAVRadar(radarSensor),"MountingAngles", [0 0
```

Simulate the scenario without obstacle avoidance

In the scenario, if you let the ego UAV keep flying along its initial direction, it will eventually collide with the fixed wing UAV as shown in the simulation.

```
% Set up the 3D view of the scenario.
[ax, plotFrames] = show3D(s);
% Represent the ego UAV as a green marker.
```

```
plot3(0,0,0,"Marker","diamond","MarkerFaceColor","green","Parent",plotFrames.EgoVehicle.BodyFrame
% Represent the other UAV as a red marker.
plot3(0,0,0,"Marker","diamond","MarkerFaceColor","red","Parent",plotFrames.Target.BodyFrame);
xlim([-5,35]);
ylim([-5,35]);

% Start simulation.
setup(s);
while advance(s)
    % Move ego UAV along its velocity vector.
    motion = read(egoMultirotor);
    motion(1:2) = motion(1:2) + motion(4:5)/s.UpdateRate;
    move(egoMultirotor, motion);

    show3D(s,"FastUpdate", true,"Parent",ax);
    pause(0.02);
end
```
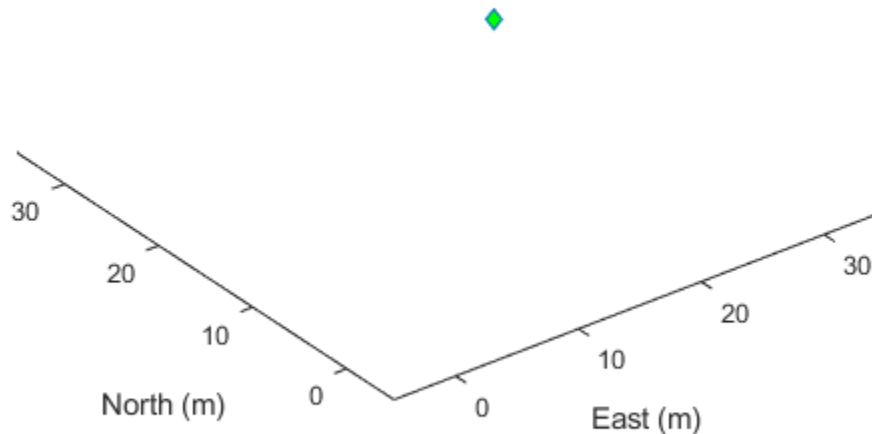

Configure obstacle avoidance behavior

To avoid such collisions, you need to detect the potential collision using radar sensor readings and generate avoidance maneuver using a velocity obstacle approach. First you setup the avoidance parameter that controls the safety radius of the avoidance algorithm and the time horizon at which the avoidance algorithm will start working.

```
% Set up safety radius
radiusUAV = 0.5;  % UAV radius (m)
```

```
radiusObs = 2;  % Obstacle radius (m)
safetyDist = 2;  % Collision safety radius (m) - same for all obstacles
effectiveRadius = radiusUAV+radiusObs+safetyDist;

% Time Horizon (time until collision to perform avoidance maneuver)
Th = 10;  % sec
```

Simulate scenario and test obstacle avoidance behavior

In this section, you simulate the UAV scenario to test the obstacle avoidance algorithm. The
avoidance algorithm controls the motion of the ego multirotor according to the desired velocity
output for collision avoidance. The avoidance algorithm tries to find a collision free direction based on
the velocities of both the ego vehicle and the target vehicle, and then direct the ego vehicle towards
the collision free direction while maintaining the same velocity magnitude.

```
% Set up the 3D view of the scenario.
[ax,plotFrames] = show3D(s);
% Represent the ego UAV as a green marker.
plot3(0,0,0,"Marker","diamond","MarkerFaceColor","green","Parent",plotFrames.EgoVehicle.BodyFrame
% Represent the other UAV as a red marker.
plot3(0,0,0,"Marker","diamond","MarkerFaceColor","red","Parent",plotFrames.Target.BodyFrame);
xlim([-5,35]);
ylim([-5,35]);

% Start simulation.
restart(s);
setup(s);
desVel = egoVelocity(1:2);
while advance(s)

    % Move the ego UAV along its velocity vector.
    motion = read(egoMultirotor);
    motion(1:2) = motion(1:2) + desVel/s.UpdateRate;
    motion(4:5) = desVel;
    move(egoMultirotor, motion);

    % Update sensor readings and read data.
    updateSensors(s);

    % Obtain detections from the radar.
    [isUpdated,time,confTracks,numTracks,config] = read(radar);

    % Perform obstacle avoidance maneuver by adjusting ego vehicle's
    % velocity
    desVel = egoVelocity(1:2);
    if numTracks > 0
        % Detect imminent collision using
        % exampleHelperDetectImminentCollision function.
        obstacleStates = [confTracks.State];
        [isOnCollisionPath, VOFrontAngle, VOBackAngle, VOMinAngle, VOMaxAngle] ...
            = exampleHelperDetectImminentCollision(motion(1:2)', motion(4:5)', obstacleStates([1
            obstacleStates([2,4],:),effectiveRadius,Th);

        % Find obstacles that are on imminent collision path.
        collisionObsIdx = find(isOnCollisionPath);
        if any(isOnCollisionPath)
            % Compute the velocity required to avoid collision using the
            % exampleHelperCVHeuristic2D function.
```

```
            desVel = exampleHelperCVHeuristic2D(norm(egoVelocity), ...
                motion(1:2)', ...
                motion(4:5)', ...
                obstacleStates([1,3],collisionObsIdx), ...
                obstacleStates([2,4],collisionObsIdx), ...
                effectiveRadius, ...
                VOFrontAngle(collisionObsIdx), ...
                VOBackAngle(collisionObsIdx), ...
                VOMinAngle(collisionObsIdx), ...
                VOMaxAngle(collisionObsIdx), Th);
            desVel = desVel';
        end
    end

    show3D(s,"FastUpdate", true,"Parent",ax);
    pause(0.02);
end
```
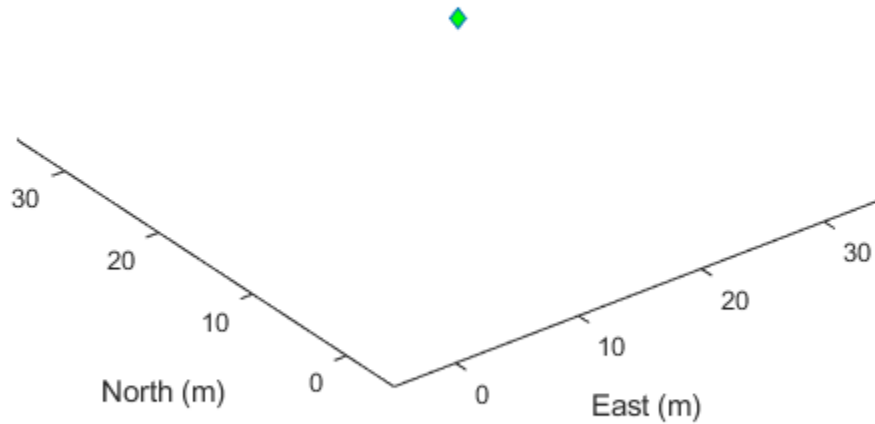

Reference

[1] Fiorini P, Shiller Z. Motion Planning in Dynamic Environments Using Velocity Obstacles. The International Journal of Robotics Research. 1998;17(7):760-772. doi:10.1177/027836499801700706