Radar Toolbox

User's Guide
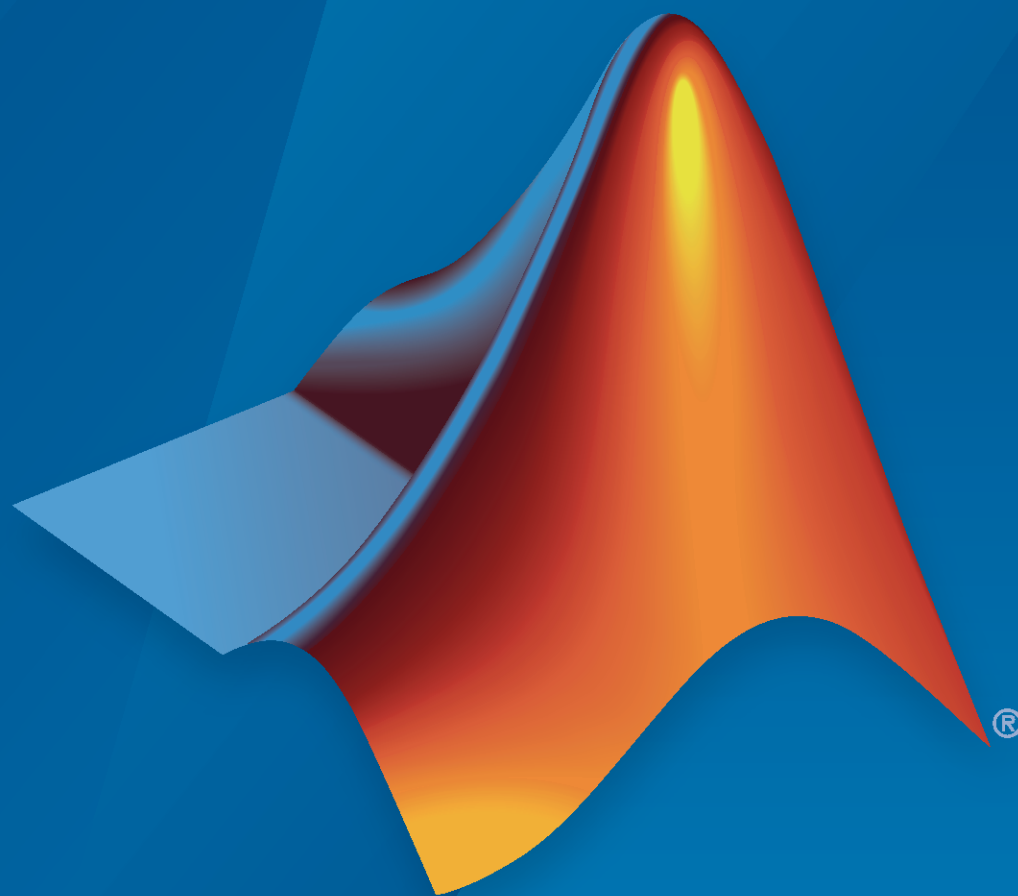
MATLAB&SIMULINK®

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

Simulink Examples

2

Clutter

3

1

Featured Examples

Radar Architecture: System Components and Requirements Allocation (Part 1)

This example is the first part of a two-part series on using Simulink® to design and test a radar system given a set of requirements. It starts by introducing a set of performance requirements that must be satisfied by the final design. A radar system architecture is then developed using Simulink System Composer™. The example then shows how to connect the radar requirements to the architecture and a corresponding design. Finally, it shows how to create a functioning model of a radar system by providing concrete implementations to the components of the architecture.

The second example in the series discusses testing the model and verification of the requirements. It shows how to use Simulink Test™ to set up test suites and run Monte Carlo simulations to verify the linked requirements. Part 2 also explores a scenario when the stated requirements have been revised. It shows how to trace the changes in the requirements to the corresponding components of the design and make modifications to the implementation and tests.

Performance Requirements

Radar system design typically begins with a set of requirements. The real-world radar systems must satisfy dozens or hundreds of requirements. In this example we consider an X-band radar system that must satisfy the following two performance requirements:

- R1: The radar must detect a Swerling 1 Case target with a radar cross section (RCS) of 1 m^2 at the range of 6000 m with a probability of detection of 0.9 and the probability of false alarm of 1e-6.
- R2: When returns are detected from two Swerling 1 Case targets separated in range by 70 m, with the same azimuth and elevation, the radar must resolve the two targets and generate two unique target reports 80 percent of the time.

Virtual Test Bed

As the first step, the example shows how to set up a virtual test bed for a radar system that will be used to implement and test the design. This test bed is useful for tracing the performance requirements to the individual components of the system, making iterative design changes, and testing and verifying the performance of the system. The example starts by creating a general top-level architecture model using System Composer. It then shows in more detail an architecture of a radar sensor component and the part of the test bed that simulates the environment and the radar targets.

Top-Level Architecture

The architecture model specifies only the conceptual components of the system, their interfaces, and links between them. The components of the architecture model are not required to have a concrete implementation. As will be shown further in this example, System Composer allows for defining specific Simulink behavior for some of the components while leaving other components specified only at the architecture level. A modular design like this is convenient and flexible since the behavior of the individual components can be modified or completely changed without the need to make any changes to other parts of the system.

In addition to the `Radar Sensor` component that models the actual radar sensor, the test bed also includes:

- `Power Substation` — Supplies power to the radar sensor.
- `Control Center` — Passes control commands to the radar sensor through `Communications Link` and receives the radar data back.
- `Targets and Environment` — Models the radar waveform propagation through the environment and the interaction of the waveform with the targets. `Radar Sensor` is connected to `Target and Environment` through a set of ports marked Tx, Rx, and `TargetsPos`. Tx and Rx links are used to pass the radar waveform to and from `Targets and Environment`. `TargetsPos` is used to pass the information about the targets positions to `Radar Sensor` in order to simulate the transmitted and received waveform in the directions of the targets.

Open the top-level architecture.

```
open_system('slexRadarArchitectureExample')
```
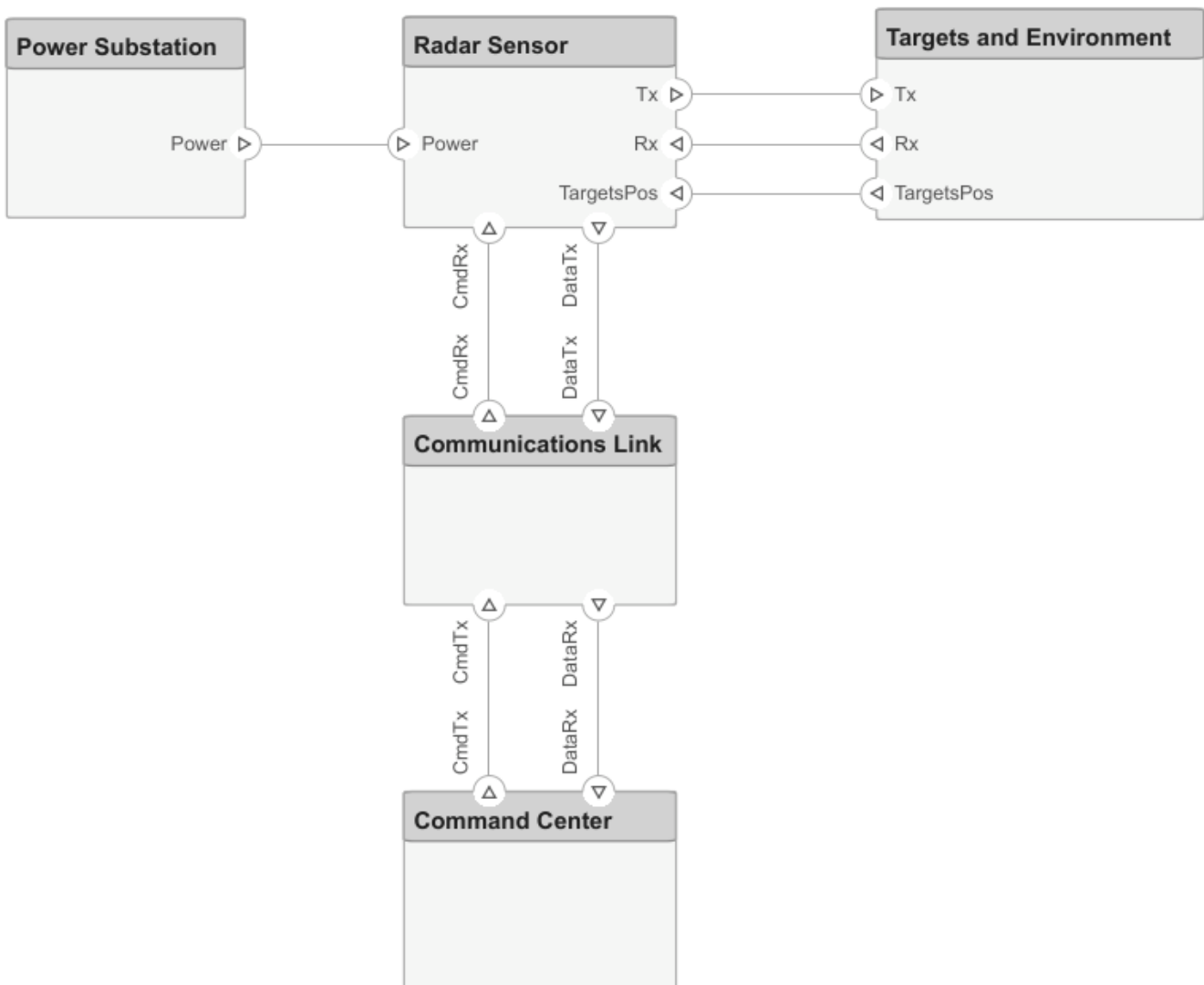
Radar Sensor

Each component in an architecture model can be further decomposed into subcomponents. As a next step, define architecture for a radar sensor. When `Radar Sensor` is decomposed, `Power`, `Tx`, `Rx`, `CmdRx`, and `DataTx` ports defined at the top level become available as the external ports. Open the `Radar Sensor` component.

```
open_system("slexRadarArchitectureExample/Radar Sensor");
```
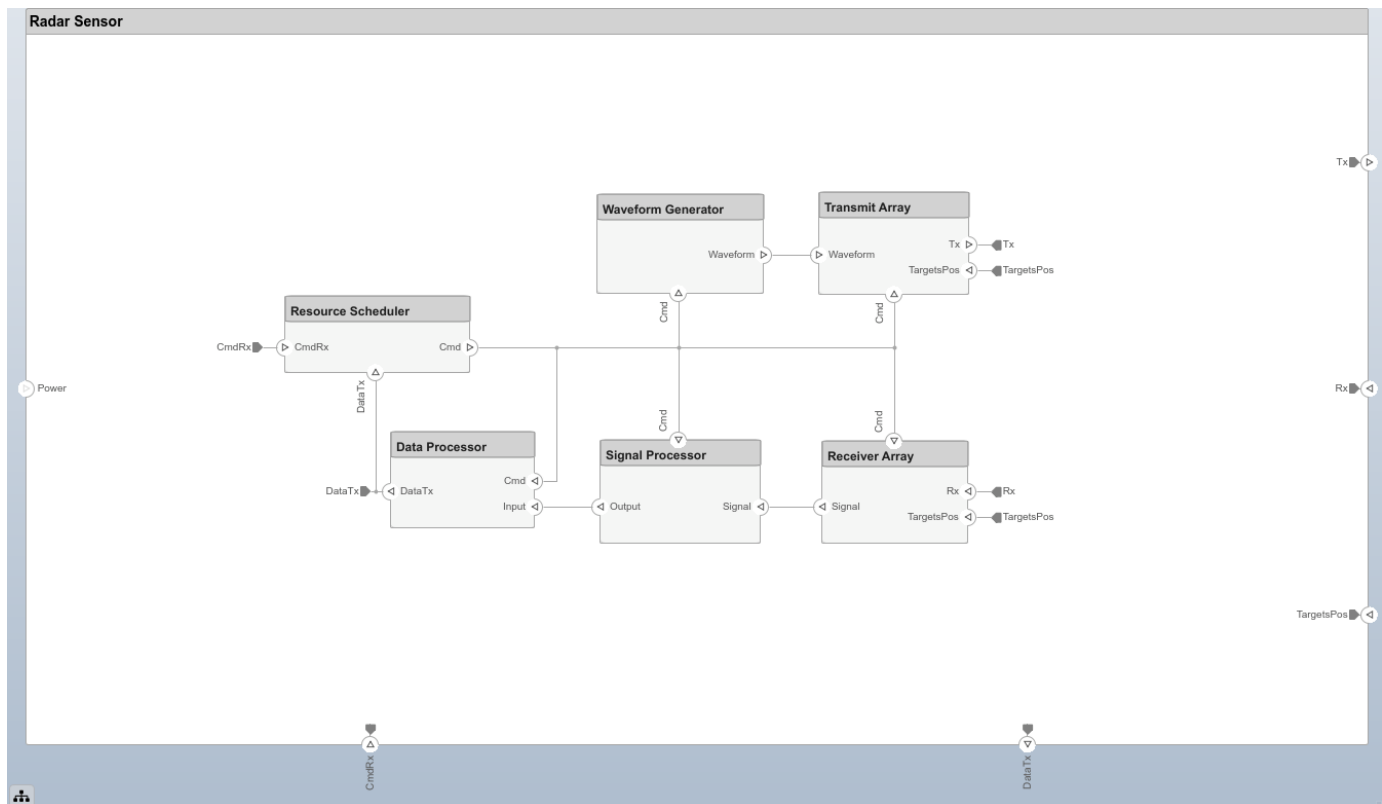

Define the following components to create an architecture model of a radar sensor:

- `Resource Scheduler` — Responsible for allocating the system resources within a dwell. It receives control commands from `Control Center` through the external CmdRx port. To indicate the flow of the control signals in the radar sensor architecture, `Resource Scheduler` is also linked to every component inside `Radar Sensor`.

- `Waveform Generator` — Produces samples of the radar waveform.

- `Transmit Array` — Passes the transmitted waveform to `Target and Environment` through the external `Tx` port.

- `Receiver Array` — Receives back the reflected waveform from `Target and Environment` through the external `Rx` port.

- `Signal Processor` — Performs beamforming, matched filtering, and pulse integration and passes the detections to `Data Processor`.

- `Data Processor` — Creates radar reports or radar tracks and passes them back to `Control Center`.

This architecture model of a radar sensor is very general. It does not make any assumptions about the type of a transmitted waveform, the shape or size of the antenna array, or the implementation of the signal and the data processing chains. The same architecture can be used to implement a large variety of different radar sensors. Further, this example implements only a subset of the listed components leaving out `Resource Scheduler` and `Data Processor`.

Targets and Environment

`Targets and Environment` can be decomposed into two subcomponents:

- `Targets` — Outputs positions and velocities of targets.
- `Propagation` — Models the propagation of the plane wave emitted by `Transmit Array` through the environment, reflection from the radar targets, and propagation back to `Receiver Array`.

Open `Targets and Environment` component.

```
open_system("slexRadarArchitectureExample/Targets and Environment")
```
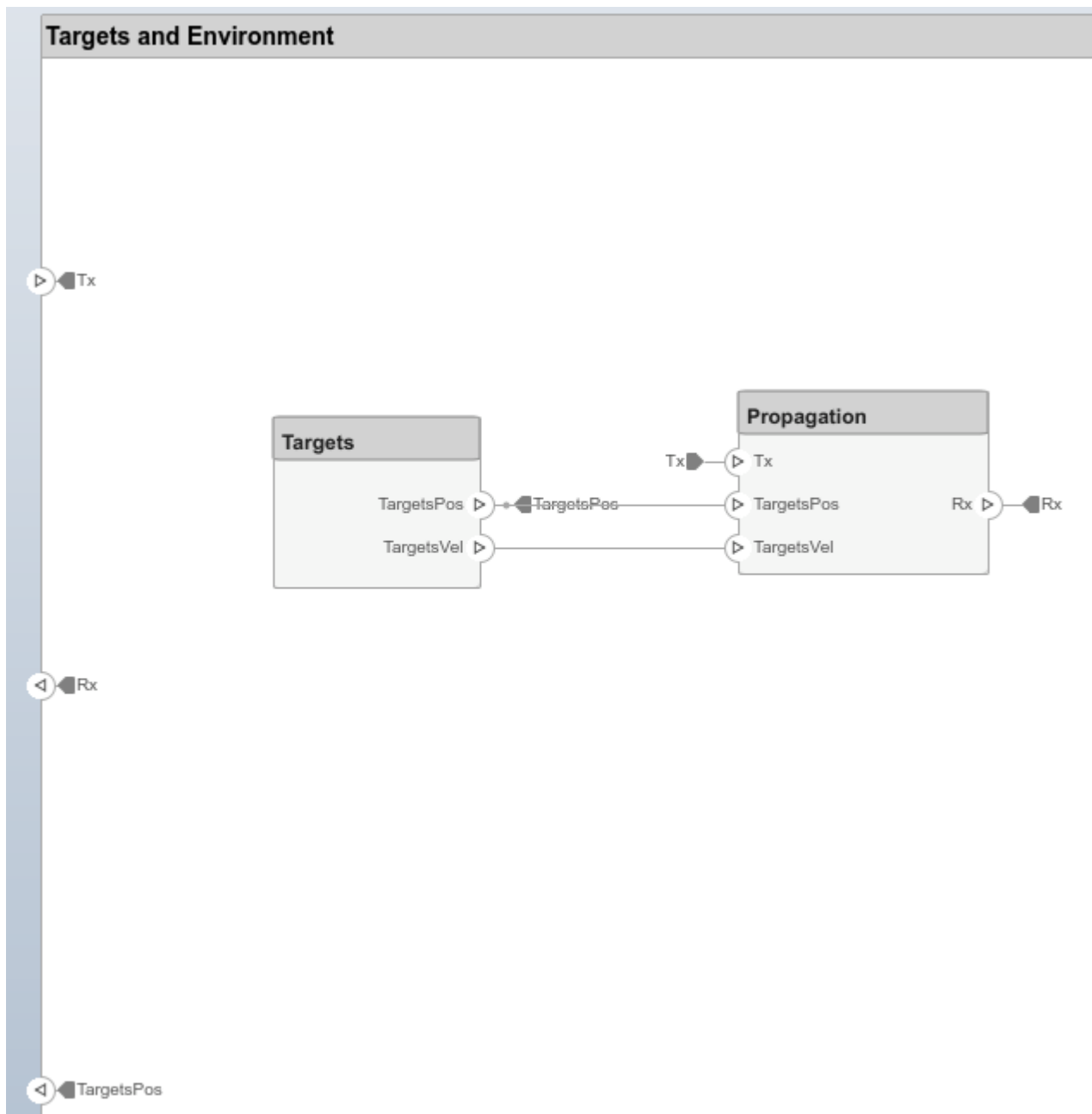
Requirements Traceability

Requirements Toolbox™ is a tool that provides a way to link the requirements to the components of the architecture responsible for implementing the corresponding functionality. When either the requirements or the model change, Requirements Toolbox provides a convenient way to trace the changes to the corresponding tests and verify that the performance and the requirements are always in agreement.

Launch Requirements Perspective app through the **Apps** tab. Then access Requirements Editor by navigating to the **Requirements** tab and selecting **Requirements Editor**. To create a new set of

requirements for the model, click on **New Requirement Set**. For this example, create a requirements set and add R1 and R2 to it. Open these requirements in Requirements Editor.

```
open('slreqRadarArchitectureExampleRequirements.slreqx')
```
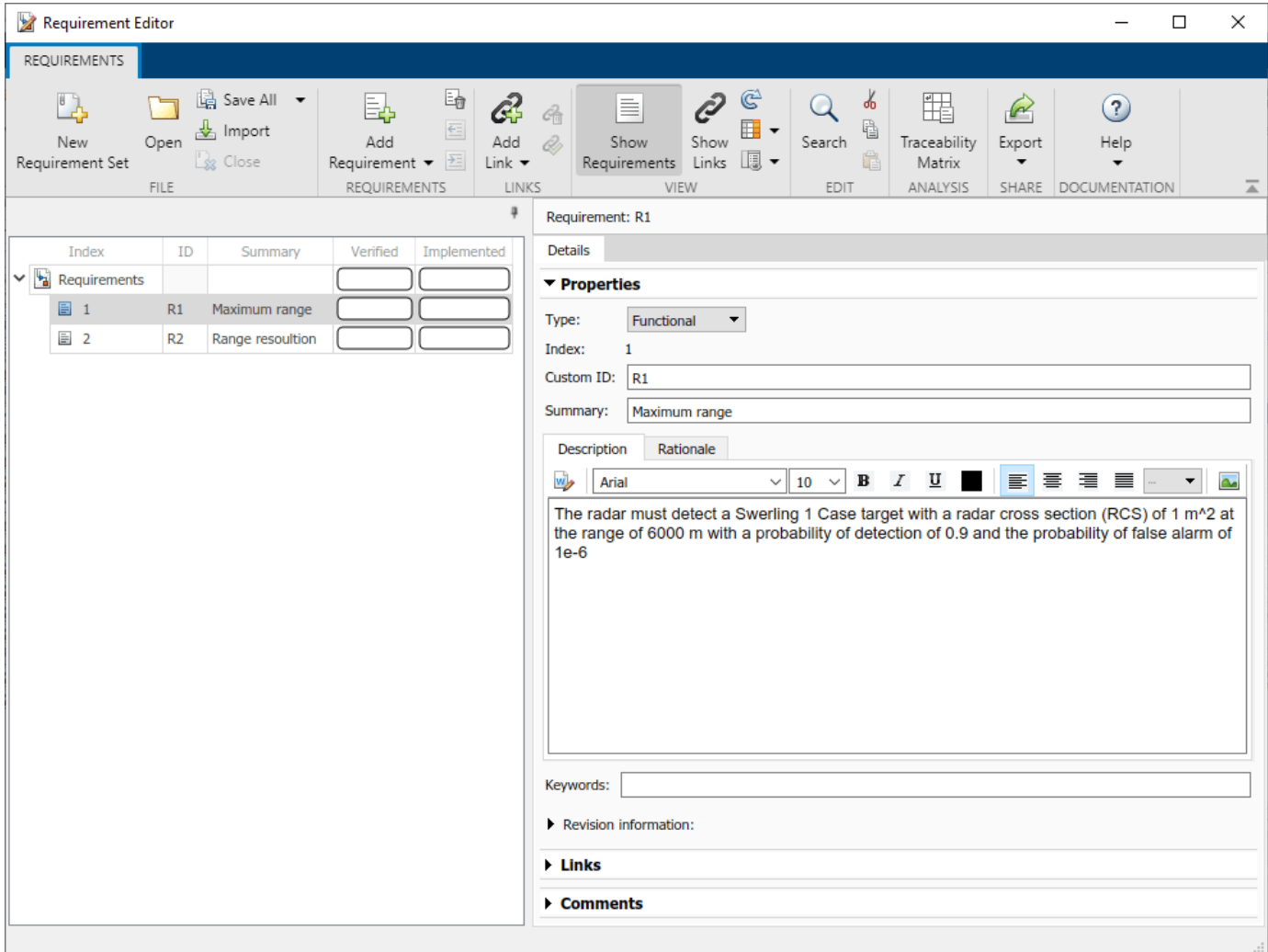

Requirements Editor lists the maximum range and the range resolution requirements. In the left panel it also shows the `Verified` and `Implemented` status for each requirement. At this moment, both requirements are not implemented and not verified. In order to change the `Implemented` status of a requirement, link it to a component of the architecture that implements the corresponding function. Link both requirements to `Waveform Generator` and `Signal Processor`. Requirements Perspective also shows the status of R1 and R2 in the bottom pane. After linking the requirements to the components, Requirements Perspective shows that the status of R1 and R2 has changed to `Implemented`. When a requirement is selected in Requirements Perspective, the components to which it is linked are highlighted with a purple frame. The linked components are also shown in the **Links** sections of the **Details** tab on the right.

Another convenient way to visualize the links between the requirements and the components of the architecture is the Traceability Matrix that can be generated by clicking on **Traceability Matrix** in the **Requirements** tab of Requirements Editor. It clearly shows which components are responsible for the implementation of each requirement.
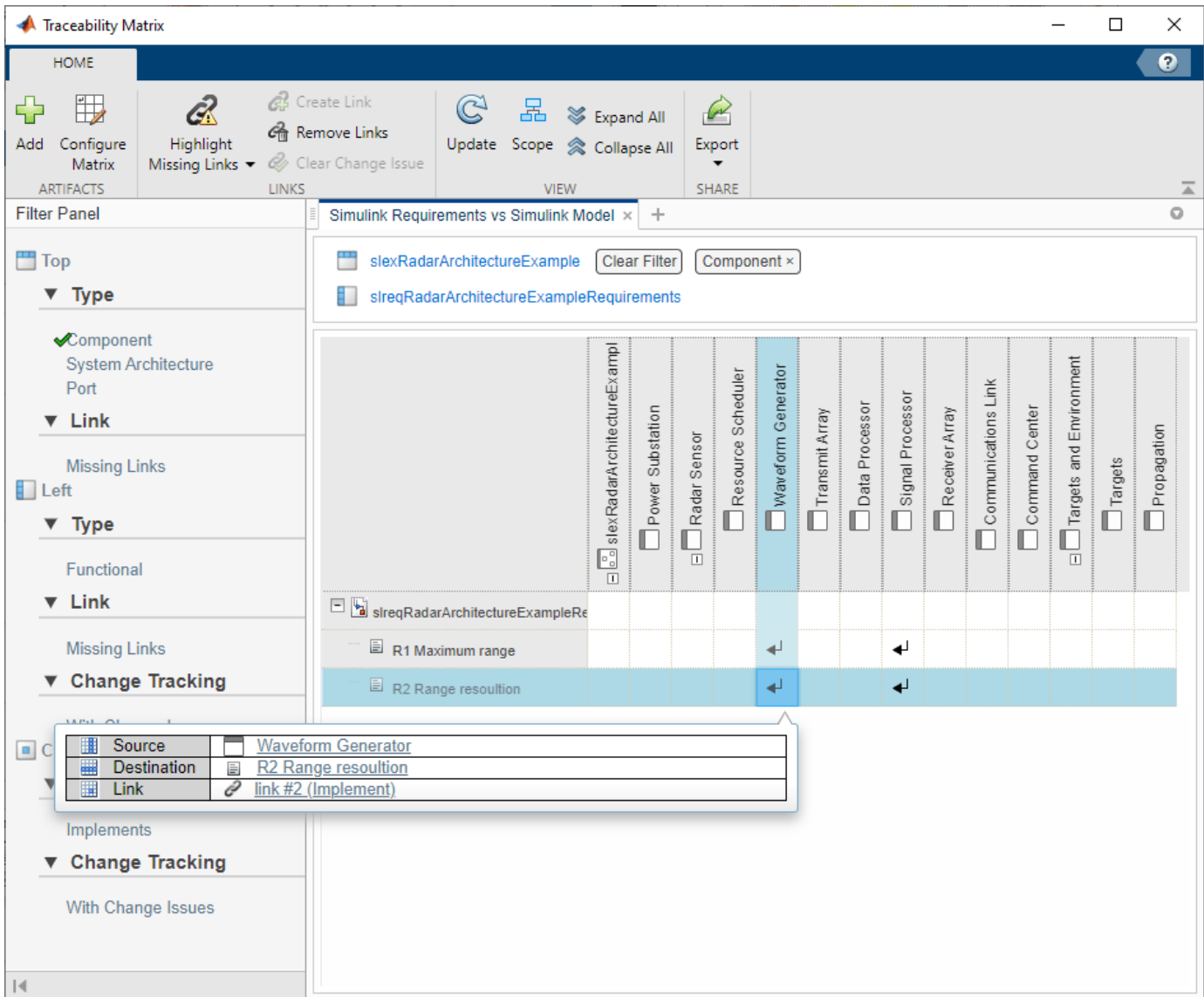
Component Implementation

To simulate a radar system, provide a concrete behavior to the components of the architecture model. System Composer allows for you to specify the behavior of some components in Simulink, while leaving the behavior of other components undefined. This provides a lot of flexibility to the design and simulation since you can build a functioning and testable model with some of the components modeled in detail while other components defined only at the abstract level. This example only specify the concrete behavior for the components of the radar sensor needed to implement generation, transmission, reception, and processing of the radar signal. It also provide a concrete implementation to `Targets and Environment`.

To specify the dimensions of signals within the model, the example assumes that the targets positions are specified by a three-row matrix, `tgtpos`, the targets velocities are specified by a three-row matrix, `tgtvel`, and the targets RCS are specified by a vector, `tgtrcs`.

System Parameters

To provide the Simulink behavior to the components of the radar sensor, first identify a set of radar design parameters that could satisfy the stated requirements. A set of parameters for a radar system that would satisfy R1 and R2 can be quickly found by performing a radar range equation analysis in the Radar Designer app. The app computes a variety of radar performance metrics and visualizes the detection performance of the radar system as a function of range. We use the `Metrics and Requirements` table to set the objective values of the maximum range and the range resolution requirements to the desired values specified in R1 and R2. Then we adjust the system parameters until the stoplight chart indicates that the performance of the system satisfies the objective requirement. The resulting set of the radar design parameters is:

- radar frequency — 10 GHz;
- peak power — 6000 W;
- pulse duration — 0.4 μs;
- pulse bandwidth — 2.5 MHz;
- pulse repetition frequency — 20 kHz;
- number of transmitted pulses — 10;
- antenna gain — 26 dB;
- noise figure — 0 dB;
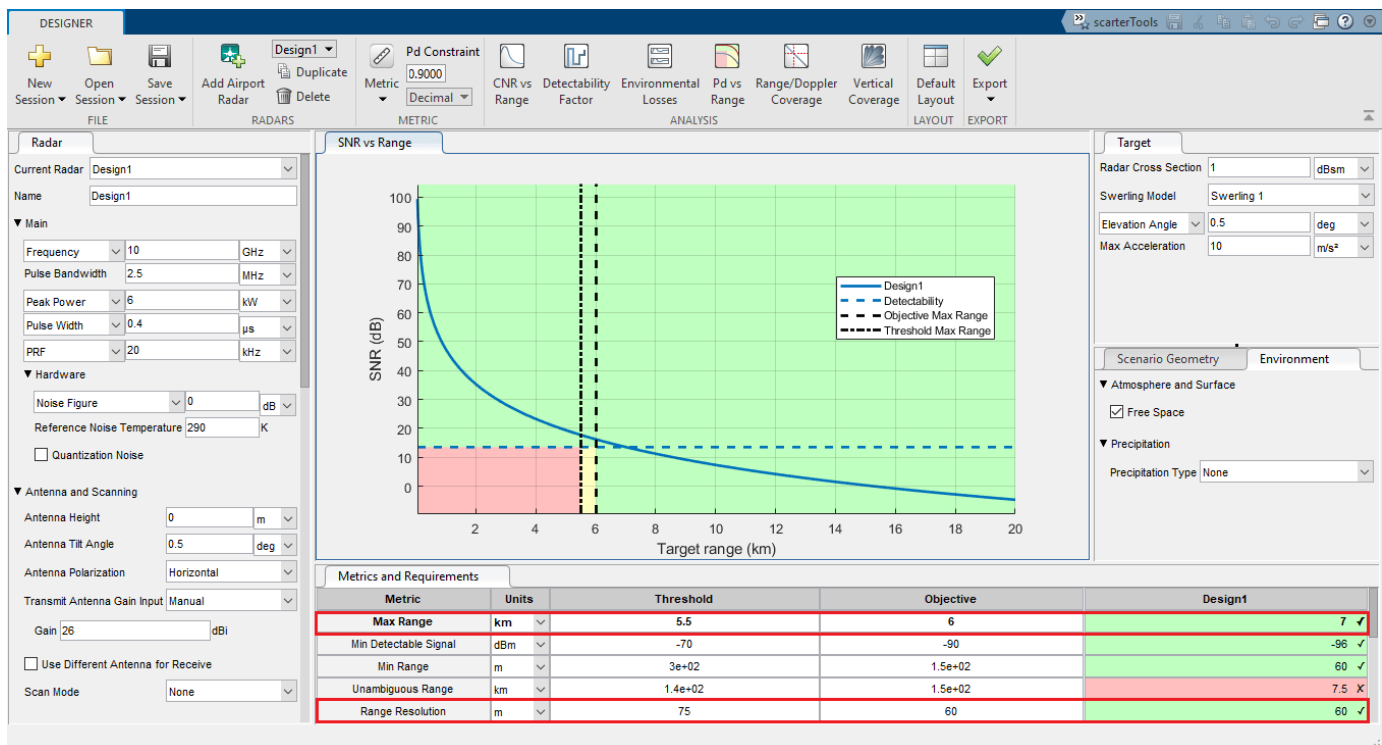
Open this design in Radar Designer app.

```
radarDesigner('RadarDesigner_RectangularWaveform.mat')
```

Waveform Generator

The analysis performed in the Radar Designer app assumes the time-bandwidth product to be equal to 1. This means that the transmitted waveform is an unmodulated rectangular pulse. Use the Pulse Waveform Analyzer app to confirm that the derived waveform parameters result in the desired performance and satisfy R1 and R2.

Start the Pulse Waveform Analyzer app with the waveform parameters defined in this example.

```
pulseWaveformAnalyzer('PulseWaveformAnalyzer_RectangularWaveform.mat')
```

The app shows that the range resolution and the unambiguous range agree well with the requirements.
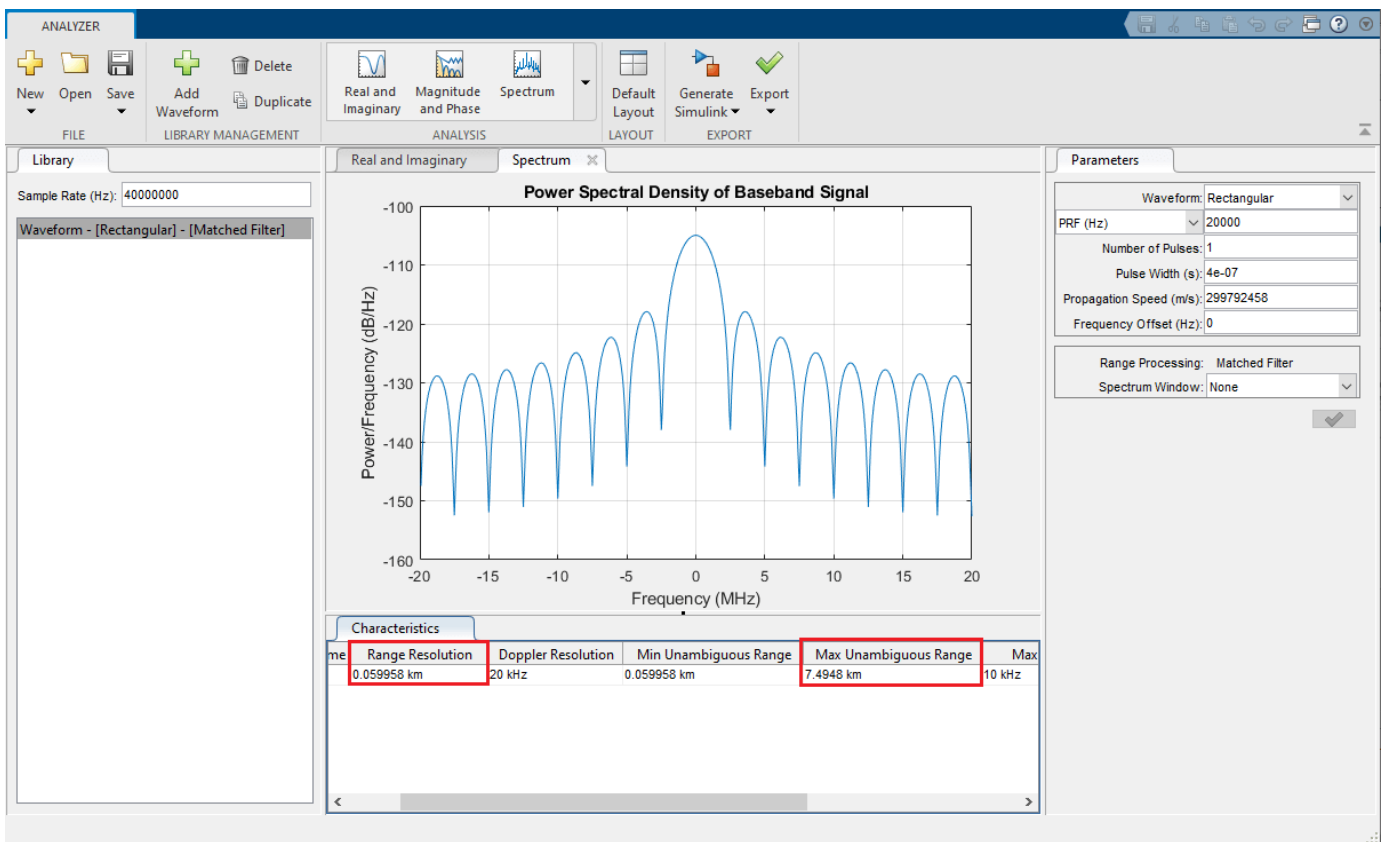
To implement this behavior in the radar model, the `Waveform Generator` component needs to contain only a single Simulink block generating a rectangular waveform. Connect the output of the `Rectangular Waveform` block to the external `Waveform` port linked to the `Transmit Array` component. Since this example does not consider the command signals, link `Cmd` input to a terminator.
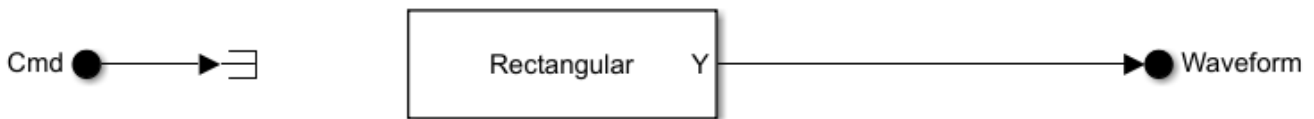
Set the `Output signal format` property of the block to `Pulses`. This means that every pulse repetition interval (PRI) of `1/prf` seconds, the block produces a column vector of `fs/prf` complex waveform samples.

Transmit Array

The `Transmit Array` component comprises the following Simulink blocks:

- `Transmitter` — Transmits the waveform generated by `Waveform Generator` with the specified peak power and transmit gain.
- `Range Angle Calculator` — Computes the directions towards the targets assuming the radar is placed on static platform located at the origin. The target directions are used as `Ang` input to `Narrowband Tx Array`.
- `Narrowband Tx Array` — Models an antenna array for transmitting narrowband signals. It outputs copies of the transmitted waveform radiated in the directions of the targets.
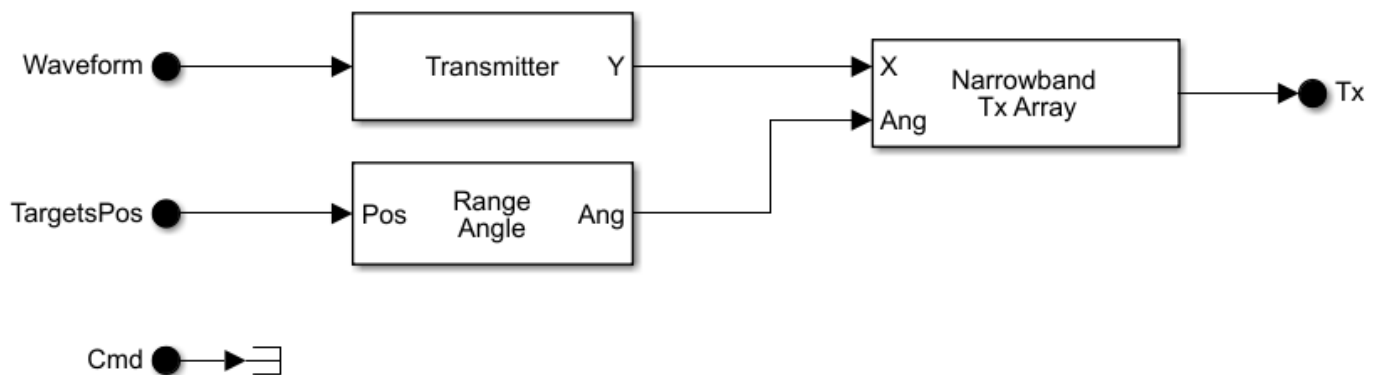
The radar range equation analysis identified that the transmit gain should be 26 dB. Set the `Gain` property of the `Transmitter` block to 20 dB and use an antenna array to get an additional gain of 6 dB. A phased array antenna with the desired properties can be designed using the Sensor Array Analyzer app. For this example, use a 4-element uniform linear array that has array gain of approximately 6 dB.

Open the array model in the Sensor Array Analyzer app.

```
sensorArrayAnalyzer('SensorArrayAnalyzer_ULA.mat')
```
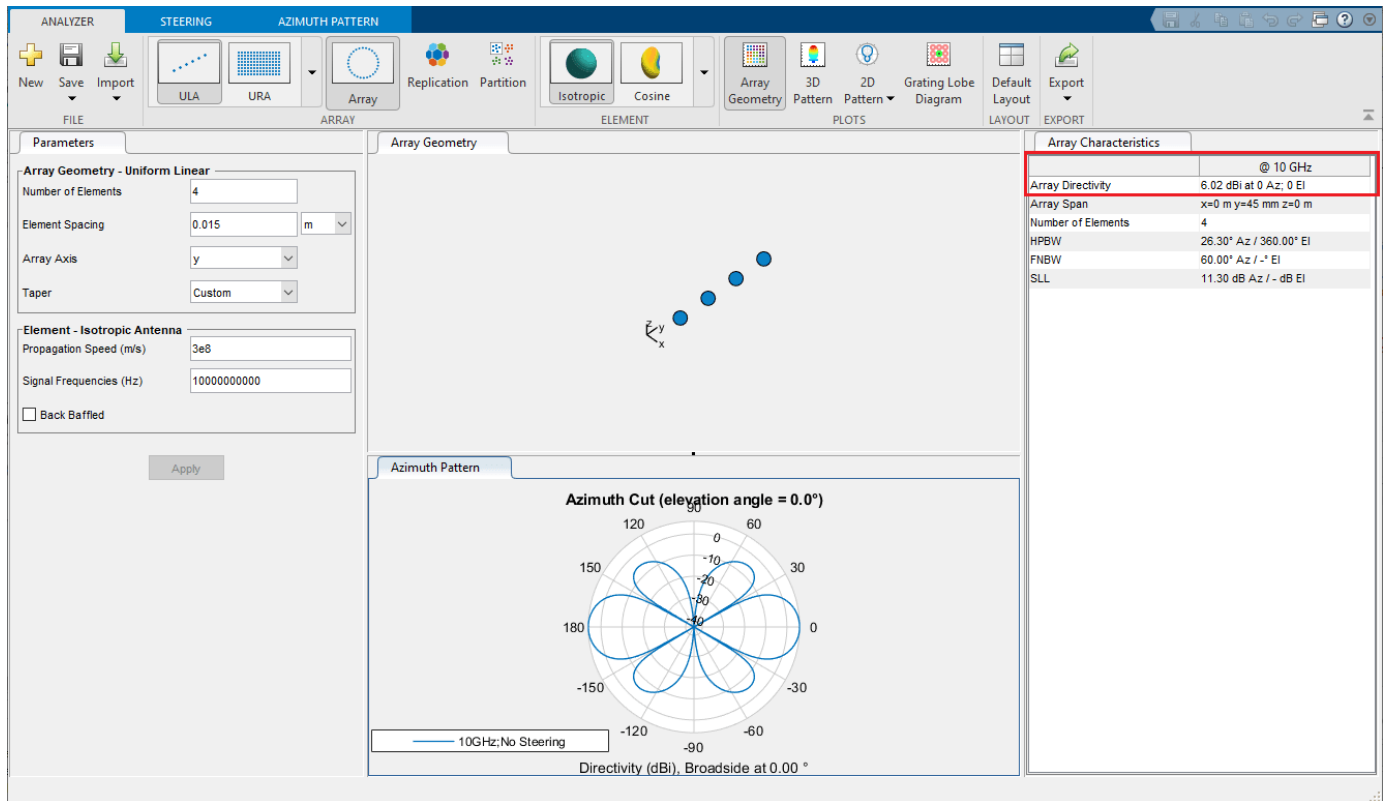
System Composer requires explicit specification of the dimensions, sample time, and complexity of the input signals. Set the dimensions of the `Waveform` input to `[fs/prf 1]`, the sample time to `1/ prf`, and the complexity to `'complex'`. The dimensions of `TargetsPos` input are set to `size(tgtpos)`, leaving the default setting for the corresponding sample time and complexity.
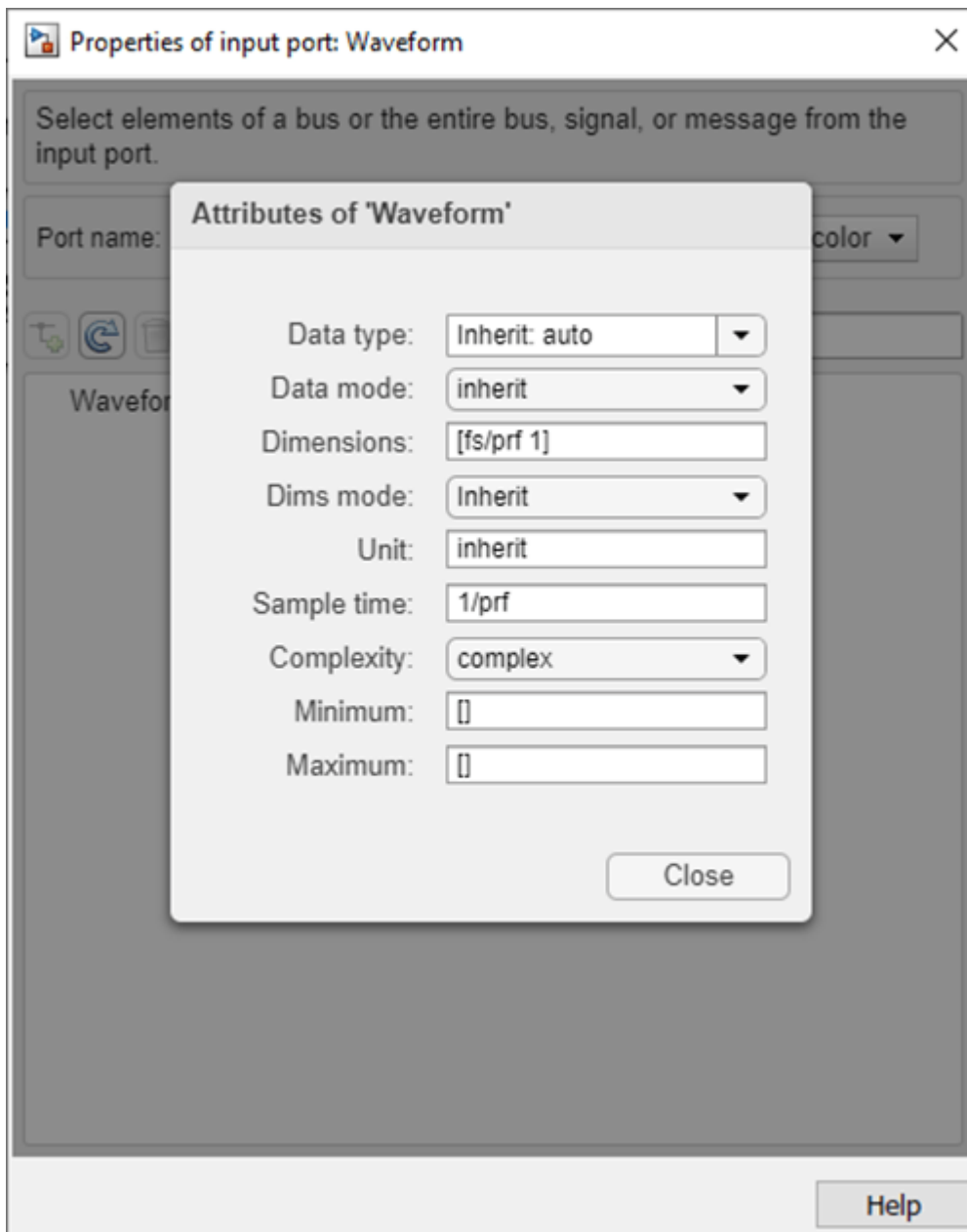
Receiver Array

- `Narrowband Rx Array` — Models the receive antenna array. It is configured using the same properties as the corresponding block in the `Transmit Array` component. At each array element the block combines the signals received from every target adding appropriate phase shifts given the targets directions computed by `Range Angle Calculator`. The output of the `Narrowband Rx Array` block is a `[fs/prf num_array_elements]` matrix.

- `Receiver Preamp` — Adds gain of 20 dB to the received signal.
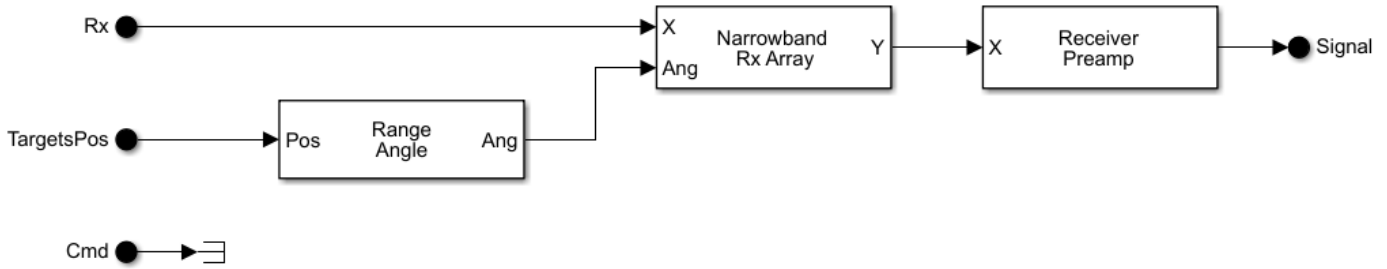
The `Rx` input is a matrix of received waveform samples with columns corresponding to `size(tgtpos,2)` targets. The dimensions of `Rx` must be set to [fs/prf `size(tgtpos,2)`], the sample time to `1/prf`, and the complexity to `'complex'`.

Signal Processor

`Signal Processor` implements a simple signal processing chain that consists of:

- `Phase Shift Beamformer` — Combines the received signals at each array element. This example sets the beamforming direction to the broadside.
- `Matched Filter` — Performs matched filtering to improve SNR. The coefficients of the matched filter are set to match the transmitted waveform.
- `Time Varying Gain` — Compensates for the free space propagation loss.
- `Noncoherent Integrator` — Integrates the magnitudes of the 10 received pulses to further improve SNR.
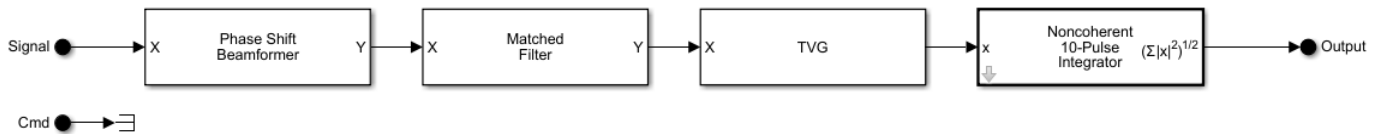
Set the dimensions of the `Signal` input to [`fs/prf num_array_elements`], the sample time to `1/prf`, and the complexity to `'complex'`.

Targets and Environment

The `Targets` component is implemented using a single `Platform` block.
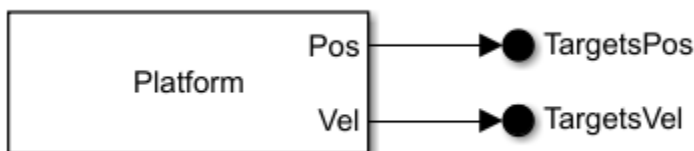
The `Propagation` component consists of:

- `Free Space Channel` — Models the two-way propagation path of the radar waveform. Set the origin position and velocity inputs of the `Free Space Channel` block to zero to indicate that the

radar is located at the origin and that it is not moving. Connect the destination position and velocity inputs to the targets positions and velocities through `TargetsPos` and `TargetVel` ports.

- `Radar Target` — Models the RCS and target fluctuation effects. Since this example considers slow fluctuating Swerling 1 Case targets, set the `Update` input to false. Also set the simulation stop time to `10/prf` indicating that a single simulation run constitutes a single coherent processing interval (CPI).
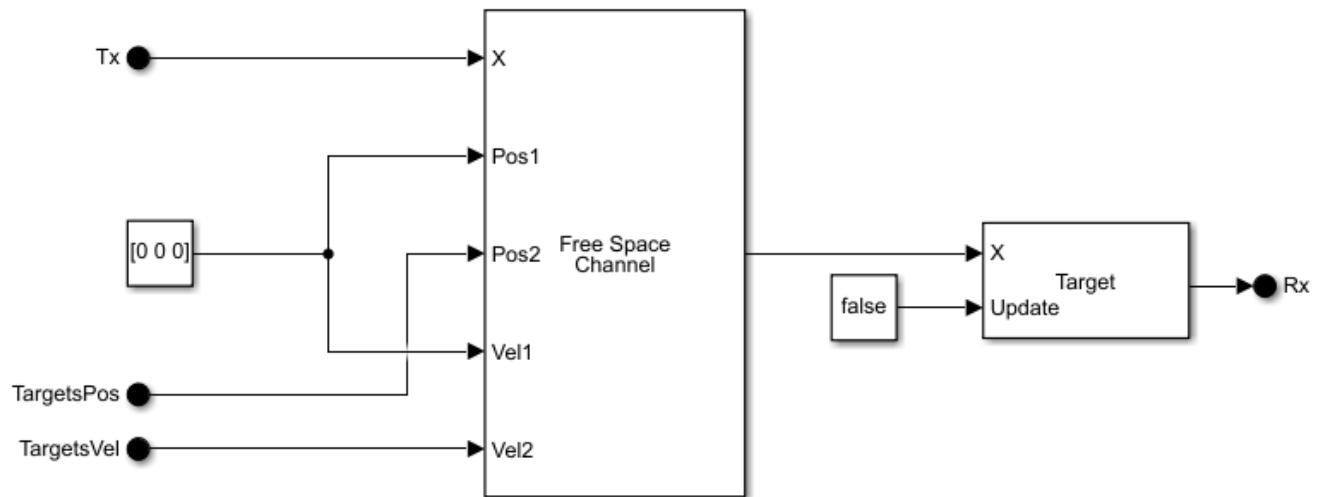
Set the dimensions of Tx input to `[fs/prf size(tgtpos,2)]`, the sample time to `1/prf`, and the complexity to `'complex'`.

Simulation Output

Specifying these blocks in Simulink is enough to obtain a model of a radar system that can produce radar detections. Prior to proceeding with testing the model and verifying the specific performance requirements, run the simulation and check whether it generates the results as expected. Consider three targets.

```
% Target positions
tgtpos = [[2024.66;0;0],[3518.63;0;0],[3845.04;0;0]];

% Target velocities
tgtvel = [[0;0;0],[0;0;0],[0;0;0]];

% Target RCS
tgtrcs = [1.0 1.0 1.0];
```

Adding the Simulation Data Inspector to log the output of the `Signal Processer` component and running a simulation results in the following range profile. As expected, there are three distinct peeks corresponding to the three targets in the simulation.

```
% Set the model parameters
helperslexRadarArchitectureParameters;

% Run the simulation
simOut = sim('slexRadarArchitectureExample');

data = simOut.logsout{1}.Values.Data;
```

```matlab
% Plot results
figure;
plot(range_gates, data(numel(range_gates)+1:end));
xlabel('Range (m)');
ylabel('Power (W)');
title('Signal Processor Output');

grid on;
```
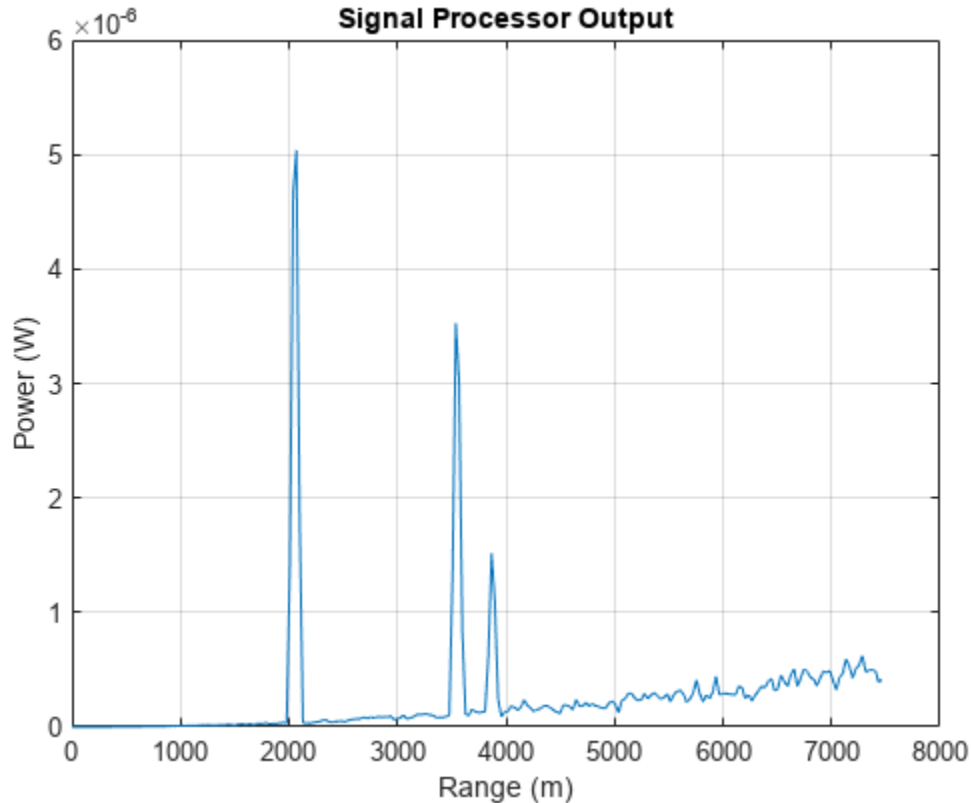

Summary

This example is the first part of a two-part series on how to design and verify a radar system in Simulink starting from a list of performance requirements. It shows how to build a radar system architecture using System Composer, which can be used as a virtual test bed for designing and testing radar system. Part 1 also shows how to link the performance requirements to the components of the architecture and how to implement the behavior of the components using Simulink to obtain a functioning and testable model.

Part 2 of this example shows how to set up test suites to test the created radar design and how to verify that the stated performance requirements are satisfied.

Radar Architecture: Test Automation and Requirements Traceability (Part 2)

This example is the second part of a two-part series on how to design and test a radar system in Simulink® based on a set of performance requirements. It discusses testing of the model developed in Part 1 and verification of the initial requirements. It shows how to use Simulink Test™ for setting up test suites to verify requirements linked to the components of the system. The example also explores a scenario when the stated requirements have been revised leading to the changes in the design and tests.

Part 1 of this example starts with a set of performance requirements. It develops an architecture model of a radar system using Simulink System Composer™. This architecture model is employed as a virtual test bed for testing and verifying the radar system designs. Part 1 shows how to use Requirements Toolbox™ to link the requirements to the components of the architecture. It also shows how to implement the individual components of the architecture using Simulink.

Automated Testing

Prior to setting up the tests, load the model constructed in the Part 1 of the example.

```
open_system('slexRadarArchitectureExample')
```

Simulink Test Manager is a tool for creating tests suites for the model. To access Test Manager click on **Simulink Test** in the **Apps** tab, then navigate to **Tests** tab and click **Simulink Test Manager**. To get started with the tests, create a new test file for the model by clicking on **New Test File**. Then add two separate test suites, one for each requirement. Further configure the test suites by:

- Adding a description to each test suite to shortly describe what functionality is being tested.
- Linking the test suite to one or multiple requirements. The tests in the test suite must pass in order for the requirements to be verified.
- Adding callbacks for setup before and cleanup after the test run. This example requires a global variable in the base workspace in order to aggregate the results of multiple Monte Carlo runs within a single test suite.
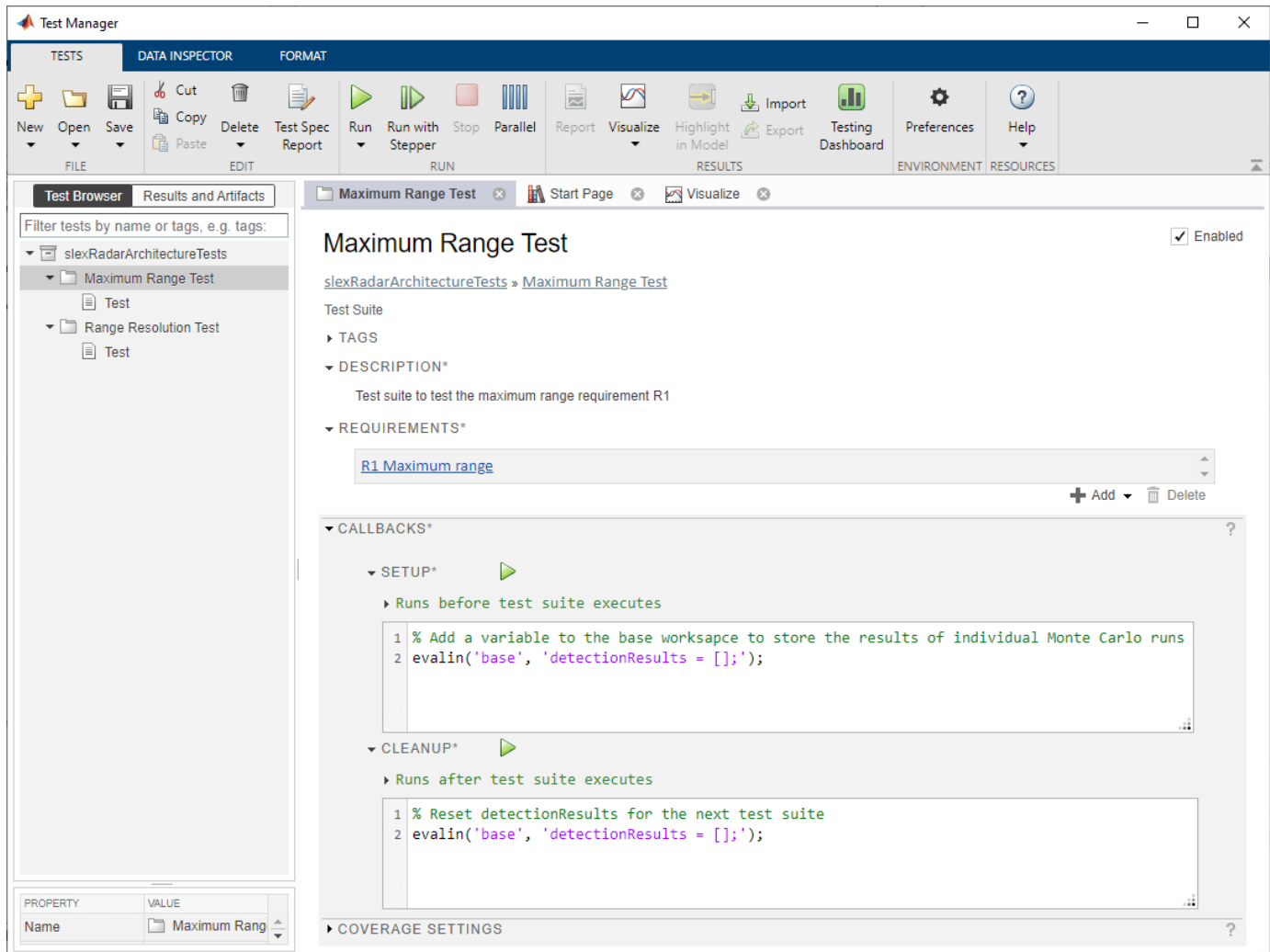
Next configure the tests within the test suites. The changes are made only in the System Under Test, Parameter Overrides, Iterations, and Custom Criteria sections.

- In the System Under Test section, set the **Model** field to the name of the model, which in this example is *slexRadarArchitectureExample*.
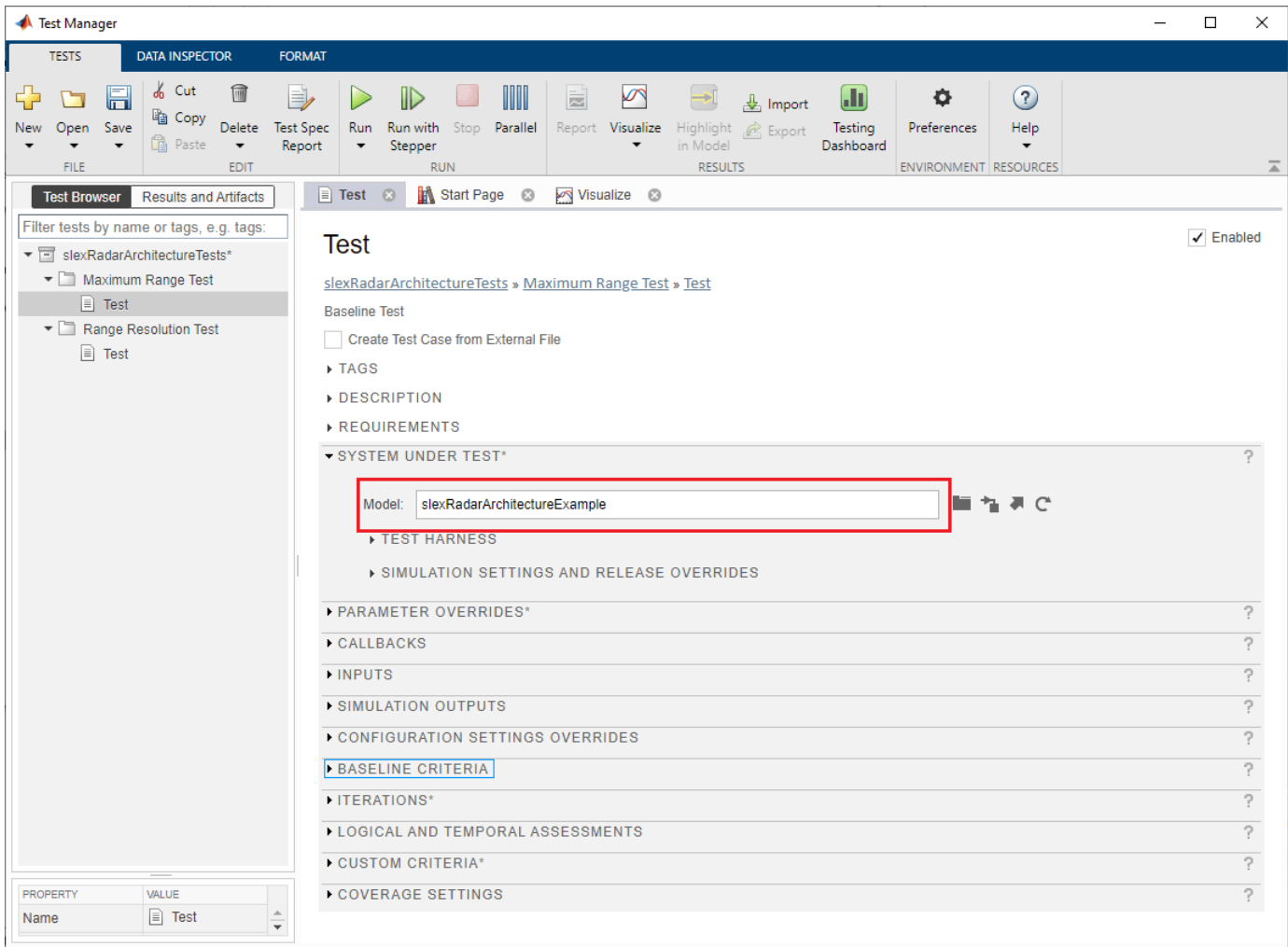
- The Parameter Overrides section is used to assign different values to the parameters in the base workspace during a test execution. Use this section to specify the targets parameters for the maximum range test and the range resolution test.

For the maximum range test, specify a single target with 1 m² radar cross section (RCS) at the range of 6000 m from the radar as stated in R1.

For the range resolution test, specify two targets with different RCS separated in range by 70 m as required by R2.

PARAMETER OVERRIDES*   ?

PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT	+
▾ ✓ Parameter Set 1				
✓ tgtpos	[2024.66 2094.31; 0 0;0 0]	base workspace	slexTargets/Motion Platform	
✓ tgtrcs	[1.32 1.2]	base workspace	slexPropagation/Radar Target	
✓ tgtvel	[0 0;0 0;0 0]	base workspace	slexTargets/Motion Platform	

➕ Add ▾   ⟳ Refresh   📤 Export   🗑 Delete

- Because of the random noise and the target fluctuation effects, it is possible to verify only the averaged radar system performance collected over multiple test runs. The Iterations section of the test can be used to configure the test to run multiple times to implement Monte Carlo simulations. This example adds a custom script to the Scripted Iterations subsection to set up Monte Carlo. The script performs only ten iterations. To robustly verify the performance of the system more iterations are required.

ITERATIONS*   ?

▸ TABLE ITERATIONS   ?

▾ SCRIPTED ITERATIONS*   ?

☐ Help on creating test iterations:
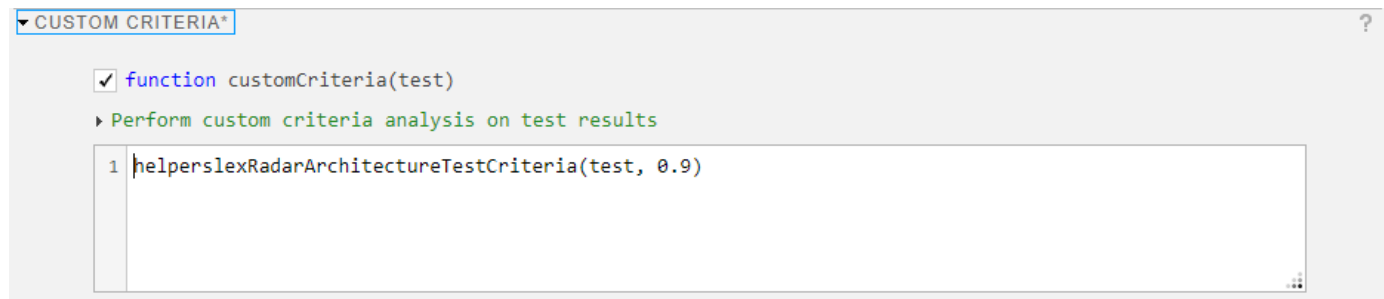
```
1  % Number of Monte Carlo runs
2  num_mc = 10;
3
4  % Create iterations for num_mc runs
5  for k = 1:num_mc
6      testItr = sltest.testmanager.TestIteration;
7      addIteration(sltest_testCase,testItr,sprintf('Iteration %d',k));
8  end
```

[Iteration Templates]   Generate an iteration script using templates

[Show Iterations]   Show the list of iterations that will execute

☐ Run test iterations in fast restart

- The Custom Criteria section allows you to specify a custom rule that verifies the test results at the end of each iteration. Configure it to run the `helperslexRadarArchitectureTestCriteria` helper function that processes results of each test iteration and stores them in the `detectionResults` variable in the base workspace. This function computes the number of detection threshold crossings. If this number is equal to the number of targets in the test, the system passes the test iteration, otherwise the iteration is declared as failed. In the last iteration, `helperslexRadarArchitectureTestCriteria` computes the total number of passed iterations. The second argument to this helper function is the percentage of the iterations that must pass for the entire test to pass. The maximum range test requires that at least 90% of all iterations pass. Since the range resolution test models two independent targets, it requires that at least 80% of all test iterations are successful.

▾ CUSTOM CRITERIA*   ?

    ✓ `function customCriteria(test)`

    ▸ Perform custom criteria analysis on test results

    1 `helperslexRadarArchitectureTestCriteria(test, 0.9)`

Open this test suite in Test Manager.

```
open('slexRadarArchitectureTests.mldatx')
```

After adding the tests and linking them to the requirements, the status of the requirements in the Requirements Editor indicates that the verification has been added but the tests have not yet been executed.
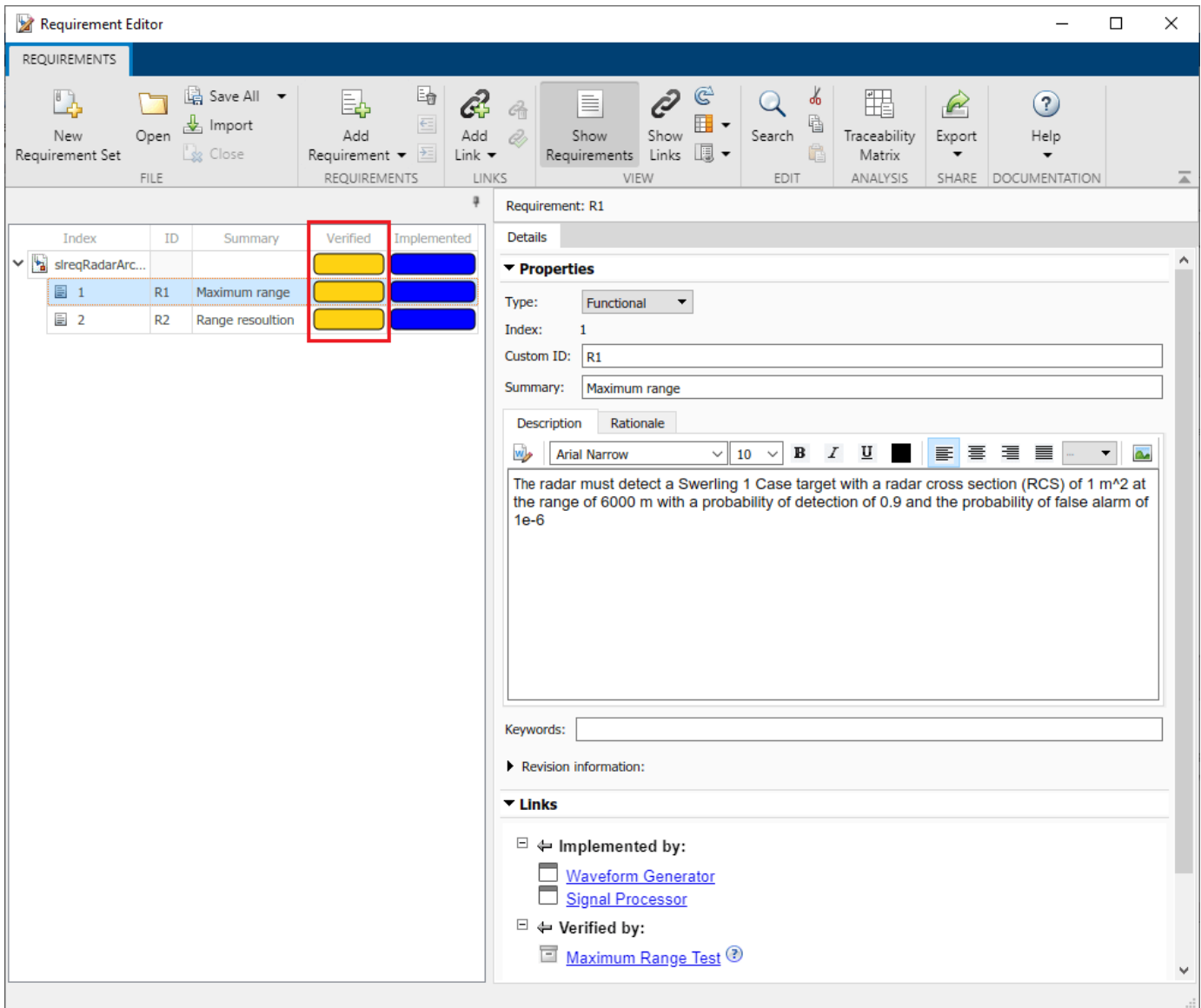
Now the tests can be launched. After running both test suites, inspect the results of each individual iteration using the Data Inspector. The custom criteria helper function also prints the status of each iteration to the Command Window.
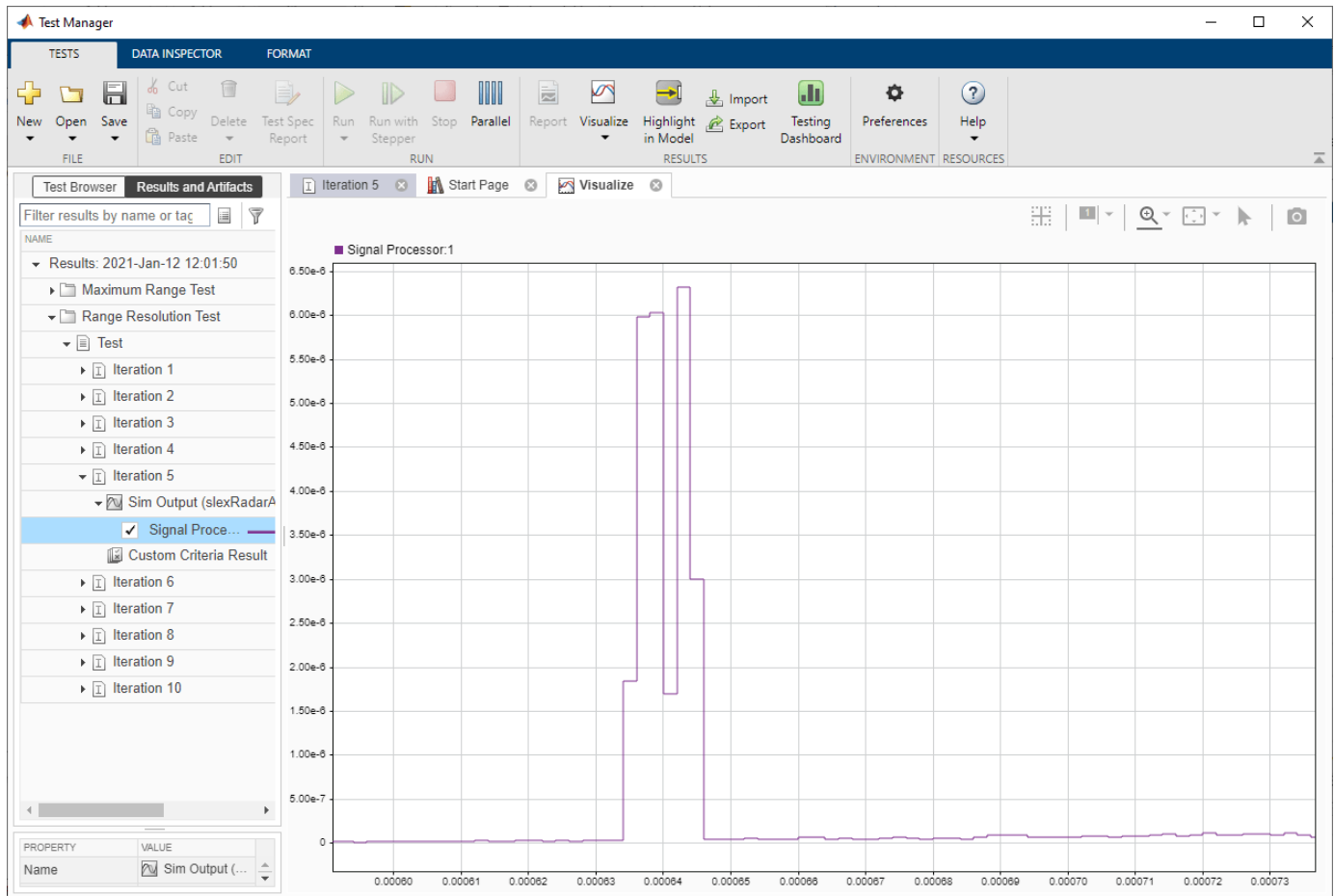
Since both tests passed, Requirements Editor now shows that both requirements have been implemented and verified.
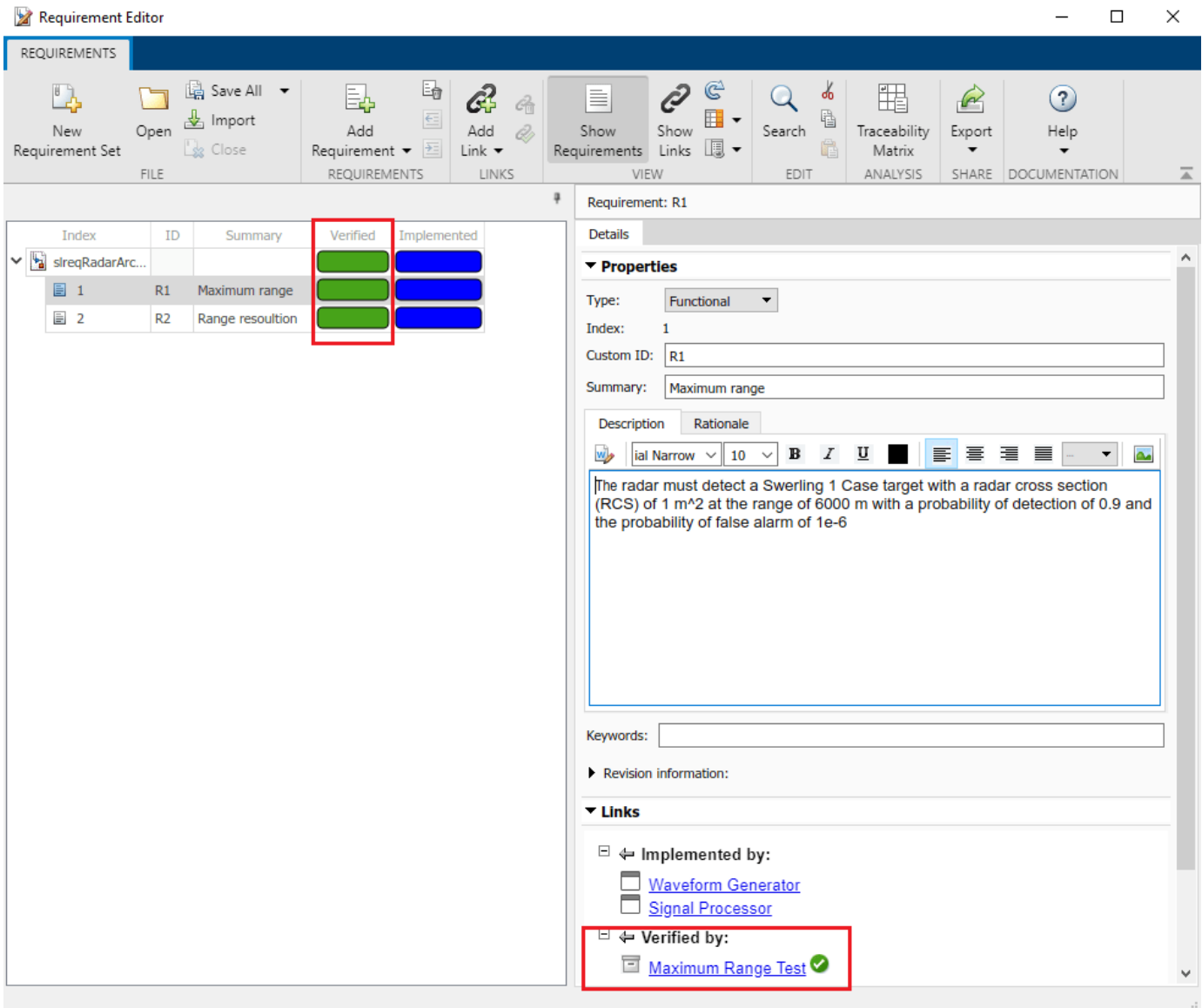
Revised Requirements

It is common that during a design process the initial requirements are revised and changed. This example assumes that the new maximum range requirement is 8000 m and the new range resolution requirement is 35 m. The updated requirements are:

- R1: The radar must detect a Swerling 1 Case target with a radar cross section (RCS) of 1 m^2 at the range of 8000 m with a probability of detection of 0.9 and the probability of false alarm of 1e-6.

- R2: When returns are detected from two Swerling 1 Case targets separated in range by 35 m, with the same azimuth, the radar must resolve the two targets and generate two unique target reports 80 percent of the time.

Making changes to requirements in Requirements Editor will generate change issues and highlight the Summary status of the corresponding requirement in red. The links to the components that

implement the changed requirement and to the tests that verify it are also highlighted. This way it is easy to identify which components of the design and which tests need to be updated in order to address the changes in the requirement and to test them.
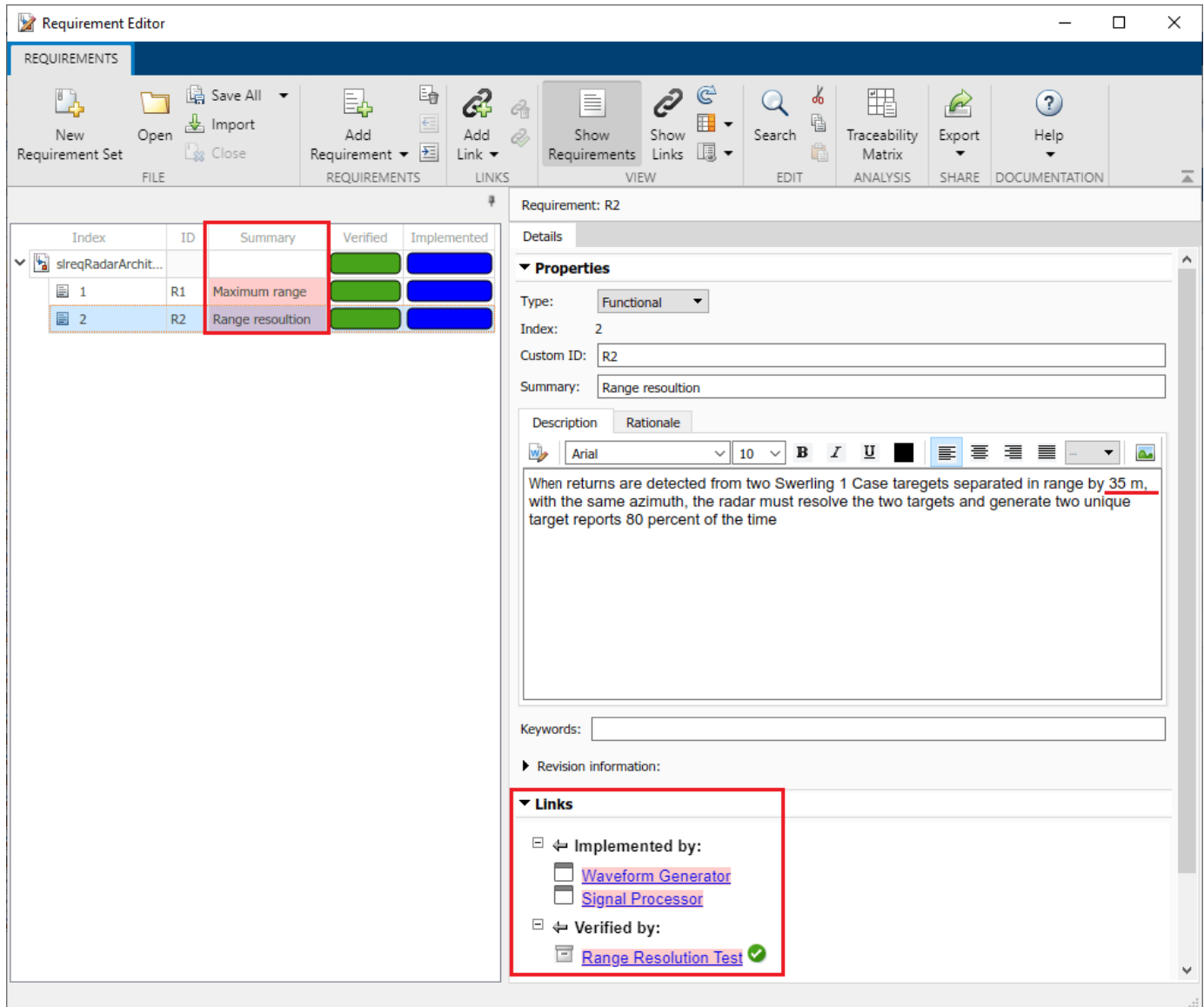
To monitor the changes in the requirements or in the implementations of the system components use the requirements Traceability Matrix.

Updated System Parameters

The new maximum range requirement is beyond the current unambiguous range of the system that equals 7494.8 m. To satisfy the new requirement, increase the unambiguous range. This can be accomplished by lowering the PRF. Setting the PRF to 16 kHz results in the unambiguous range of 9368.5 m, which is well beyond the required maximum range of 8000 m.

Since the current radar design transmits unmodulated rectangular pulses, the resolution limit of the system is determined by the pulse width. The current range resolution limit is 60 m. The new requirement of 35 m is almost two times lower. A rectangular pulse which satisfies this requirement would have to be twice as short, reducing the available power at the same range by half. The requirement analysis using the Radar Designer app shows that this system cannot reach the required detection performance at the maximum range of 8000 m. To achieve the required maximum range and range resolution, without increasing the peak transmitted power or the antenna gain, adopt a new waveform with the time-bandwidth product that is larger than 1. Setting the pulse width to 1 μs and the bandwidth to 5 MHz will provide the desired resolution.

Open this design in Radar Designer app.

```
radarDesigner('RadarDesigner_LFMWaveform.mat')
```
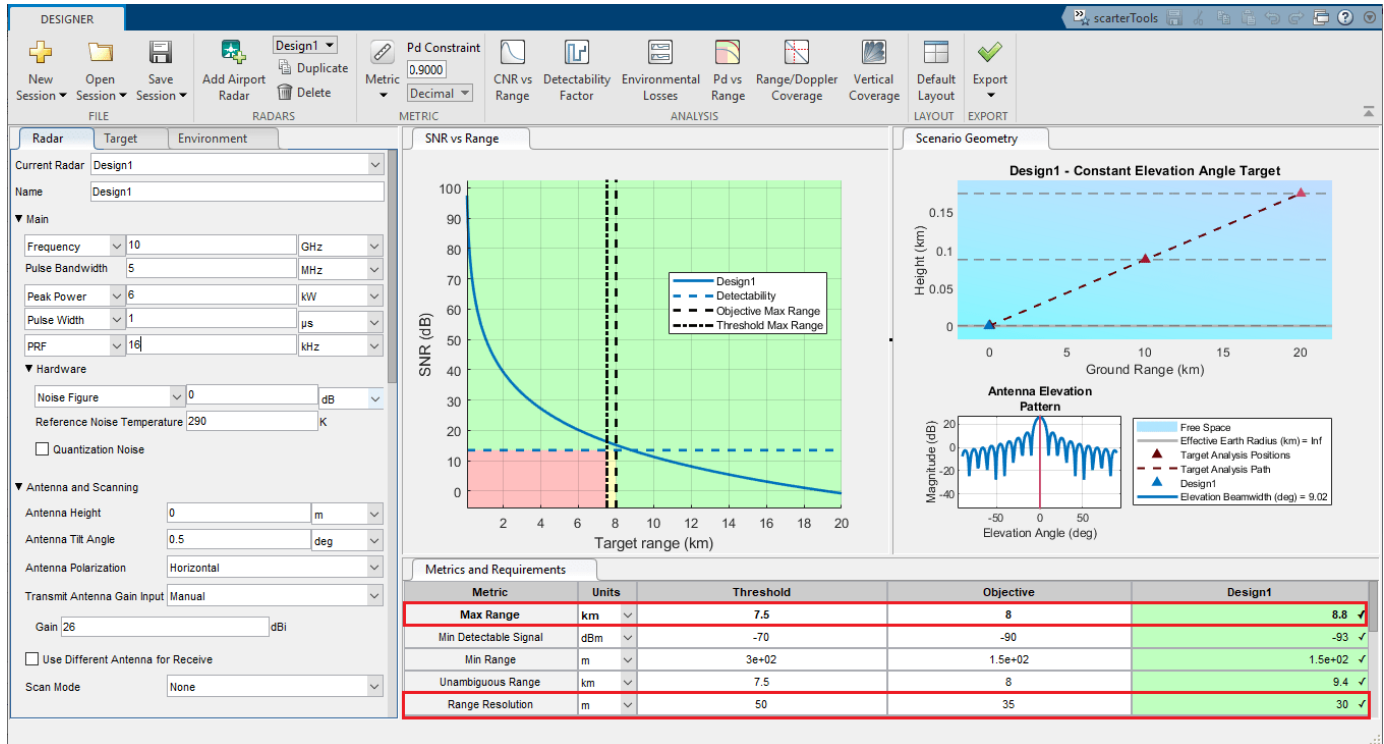

The Pulse Waveform Analyzer app can be used to select a radar waveform from several alternatives. This example uses the LFM waveform.

```
pulseWaveformAnalyzer('PulseWaveformAnalyzer_LFMWaveform.mat')
```
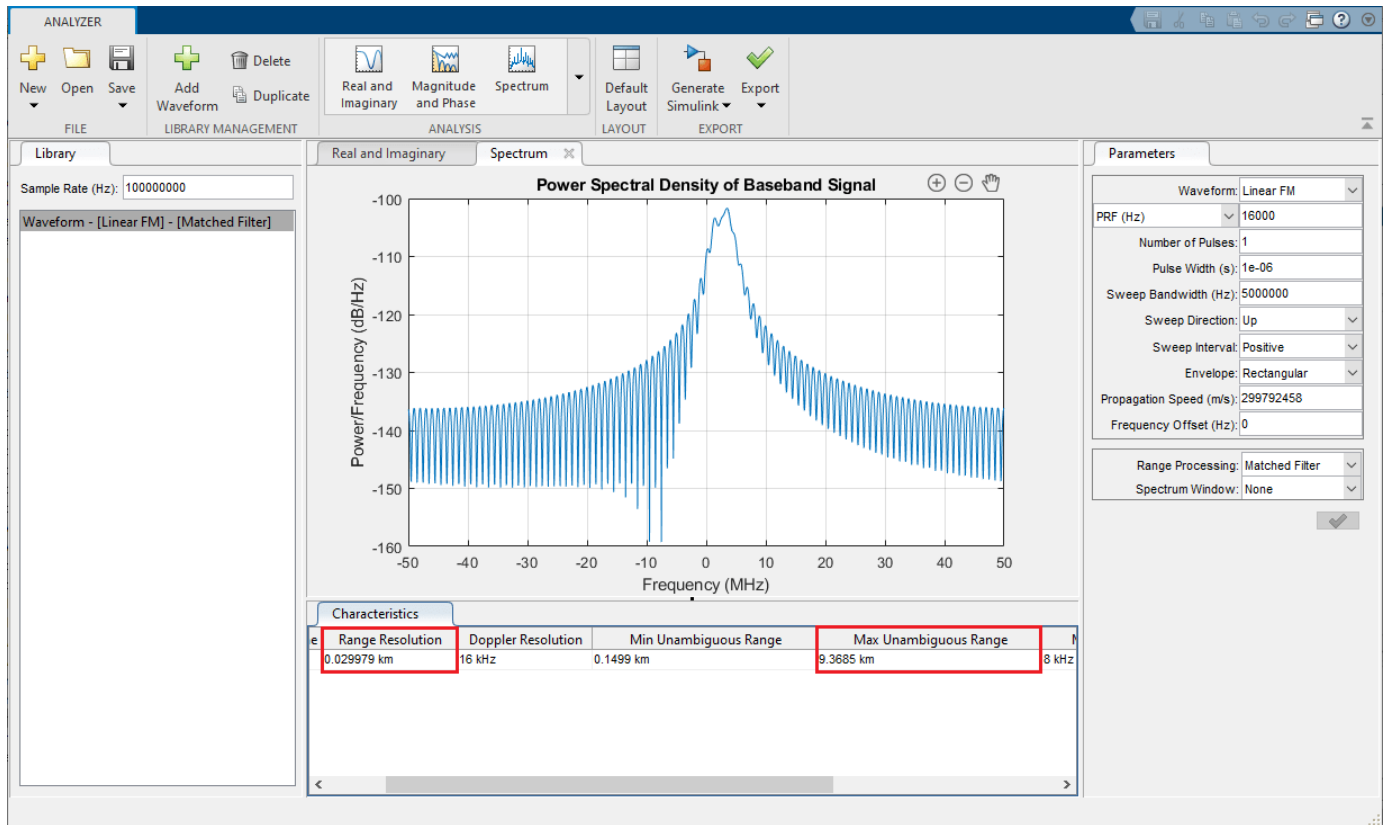
Revised Design

A convenient way to modify the behavior of a component of the system is to add an alternative design by creating a variant. This is done by right clicking on the component and selecting **Add Variant Choice**. Add a variant to `Waveform Generator` and add Simulink behavior to it to implement the LFM waveform generation.
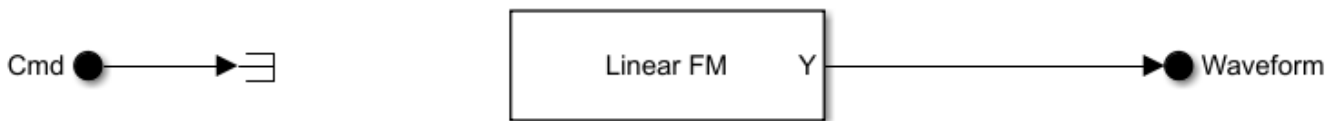
Configure the `Linear FM` block by setting the pulse width to the new value of 1 *μs*. Set the sweep bandwidth to 5 MHz and the PRF property to the updated PRF value of 16 kHz. Run the model with the LFM waveform.

```
% Set the model parameters
helperslexRadarArchitectureParameters;

% Update the model parameters to use the LFM waveform
helperslexRadarArchitectureParametersLFM;

simOut = sim('slexRadarArchitectureExample.slx');

data = simOut.logsout{1}.Values.Data;

figure;
plot(range_gates, data(numel(range_gates)+1:end));
```

```
xlabel('Range (m)');
ylabel('Power (W)');
title('Signal Processor Output');

grid on;
```
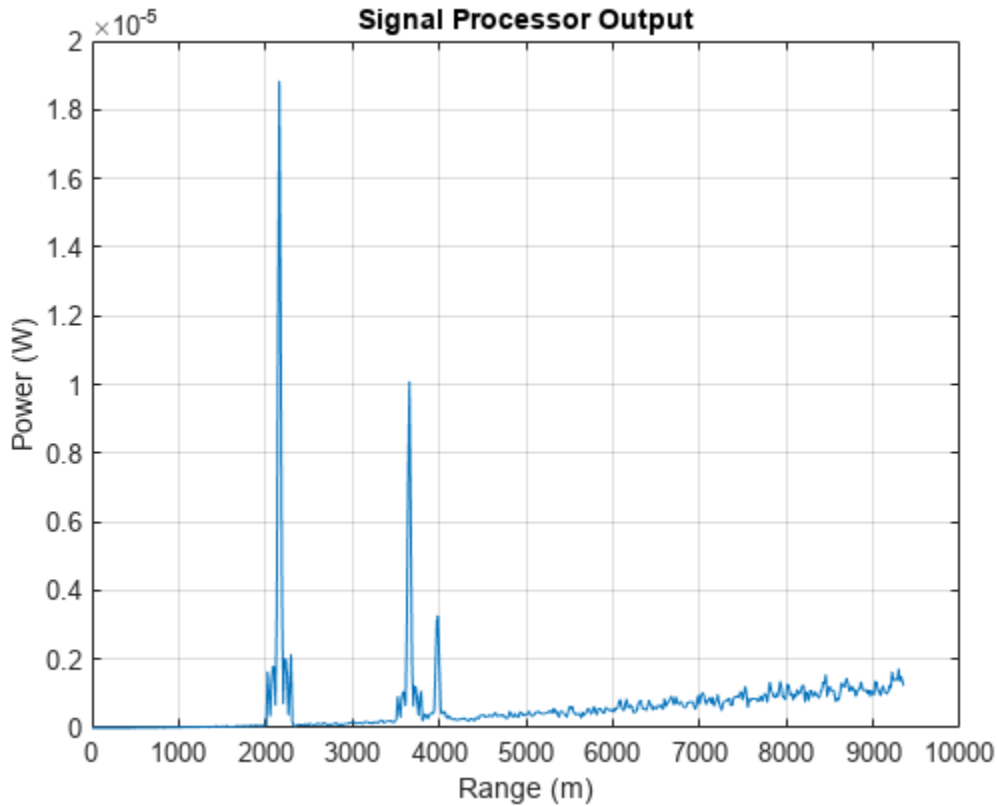

Updated Tests

Before verifying that the radar system with LFM can satisfy the updated requirements, make corresponding modifications to the tests by updating the targets positions.

- Set the target range in the maximum range test to 8000 m

- Change target ranges in the range resolution test so the targets are positioned 35 m from each other

After updating the tests, clear all change issues in Requirements Editor. Click **Show Links** in the **Requirements** tab, then select the links and click on **Clear All** button in **Change Information** section of the **Details** panel on the right. Launch the test when the issues are cleared. The new design will pass the updated tests and verify that the system satisfies the updated requirements confirming the predictions made by the Radar Designer app.

Summary

This example is the second part of a two-part series on how to design and test a radar system in Simulink based on a set of performance requirements. It shows how to use Simulink Test to test the model developed in Part 1, how to link the test to the requirements, and how to verify that the requirements are satisfied by running Monte Carlo simulations. The example also illustrates how to trace changes in the requirements to the corresponding components and how to create alternative designs by adding variants to the model. Part 1 of this example starts with the requirements that must be satisfied by the final design. It uses System Composer to develop an architecture model of a radar system that can serve as a virtual test bed. Part 1 also shows how to use Requirements Toolbox to link the requirements to the components and how to implement the individual components of the architecture using Simulink.

Benchmark Trajectories for Multi-Object Tracking

This example shows how to generate and visualize trajectories of multiple aircraft using the `radarScenario` and `waypointTrajectory` objects.

Introduction

The six aircraft trajectories modeled in this example are described in [1]. The aircraft fly in an arrangement intended to be received by a radar located at the origin.

Choice of Interpolant

Conceptually speaking, a trajectory is a curve through space which an object travels as a function of time. To define the curve, you may think of a curve through space that passes through a set of points called *waypoints* connected by an interpolating function called an *interpolant*. An interpolant allows you to define the path between waypoints via a continuous function. Common interpolants are polynomial based (for example, piecewise linear or cubic splines). For a rapidly changing trajectory, more waypoints are required to keep the interpolated curve as close to the true curve as possible; however, we can reduce the number of required points by choosing interpolants carefully.

Many motion models used in track filters consist of constant velocity, constant turn, or constant acceleration profiles. To accommodate these motion models, the interpolant used in the `waypointTrajectory` object is based on a piecewise clothoid spline (horizontally) and a cubic spline (vertically). The curvature of a clothoid spline varies linearly with respect to distance traveled; this lets you model straight and constant turns with ease, having one extra degree of freedom to transition smoothly between straight and curved segments. Similarly, objects in the air experience the effects of gravity, following a parabolic (quadratic) path. Having a cubic spline to model vertical elevation allows you to model the path with a similar extra degree of freedom.

Once the physical path through space of an object is known (and set), the speed of the object as a function of distance traveled is determined via cubic Hermite interpolation. This is useful for modeling trajectories of objects that accelerate through turns or straight segments.

The benchmark trajectories consist of straight, constant-g turns, and turns with acceleration.

Waypoint Construction

The following file contains tables of waypoints and velocities (in units of meters and meters per second) that can be used to reconstruct six aircraft trajectories. Load it into MATLAB® and examine the table containing the first trajectory.

```
load('radarBenchmarkTrajectoryTables.mat', 'trajTable');
trajTable{1}
```

```
ans =

  14x3 table

    Time          Waypoints                        Velocities
    _____    _____    _____

       0     72947     29474     -1258      -258.9    -129.69        0
      60     57413     21695     -1258      -258.9    -129.66        0
```

62	56905	21417	-1258	-245.3	-153.89	0
78.1	54591	17566	-1258	-20.635	-288.86	0
80	54573	17016	-1258	-2.8042	-289.59	0
83	54571	16147	-1258	-0.061	-289.56	0
110	54571	8329	-1258	0	-289.56	0
112.7	54634	7551.5	-1258	58.979	-283.56	0
120	55718	5785.5	-1258	226.41	-180.59	0
129	58170	5172.8	-1258	284.74	52.88	0
132	59004	5413.9	-1258	274.26	93.05	0
137.8	60592	5962.2	-1258	273.62	94.76	0
147.8	63328	6909.9	-1258	273.62	94.76	0
185	73508	10435	-1258	273.62	94.76	0

Scenario Generation

The table contains a set of waypoints and velocities that the aircraft passes through at the corresponding time.

To use the control points, you can create a scenario with six platforms and assign a trajectory to each.

```
scene = radarScenario('UpdateRate',10);

for n=1:6
    plat = platform(scene);
    traj = trajTable{n};
    plat.Trajectory = waypointTrajectory(traj.Waypoints, traj.Time, 'Velocities', traj.Velocities
end
```

Trajectory Visualization

Once you have the scenario and plotter set up, you can set up a `theaterPlot` object to create an animated view of the locations of the aircraft as time progresses.

```
helperPlot = helperBenchmarkPlotter(numel(scene.Platforms));

while advance(scene)
    % extract the pose of each of the six aircraft
    poses = platformPoses(scene);

    % update the plot
    update(helperPlot, poses, scene.SimulationTime);
end
```
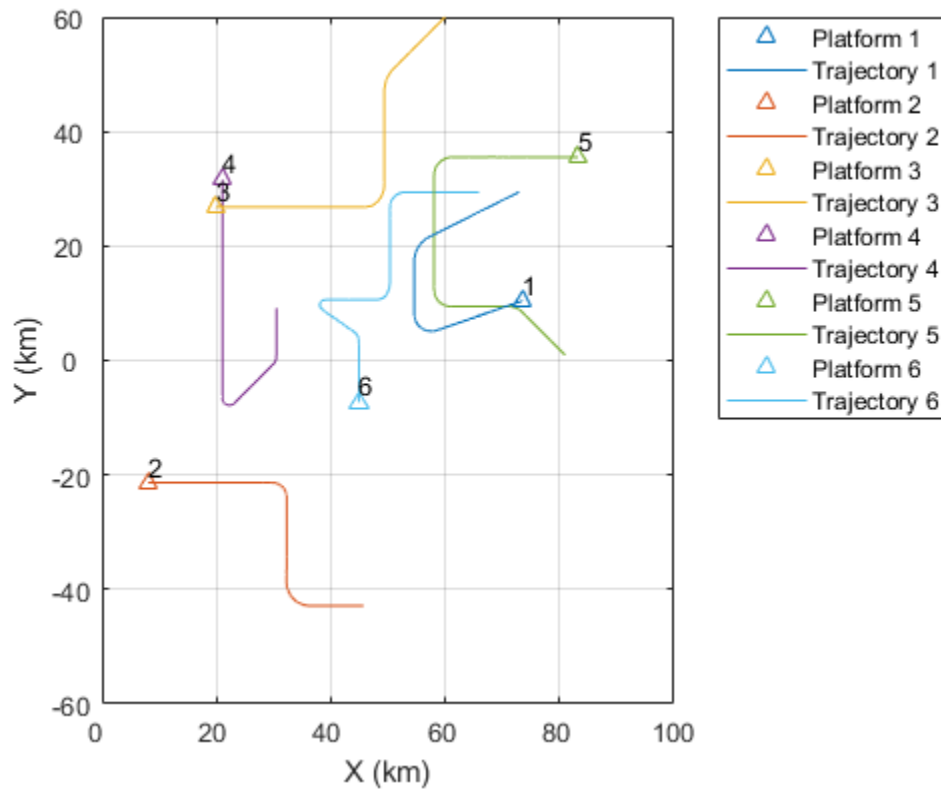
The trajectories plotted above are three-dimensional. You can rotate the plot so that the elevation of the trajectories is readily visible. You can use the `view` and `axis` commands to adjust the plot. Because the trajectories use a north-east-down (NED) coordinate system, elevation above ground has a negative z component.

```
view(60,10);
axis square
grid minor
set(gca,'ZDir','reverse');
```
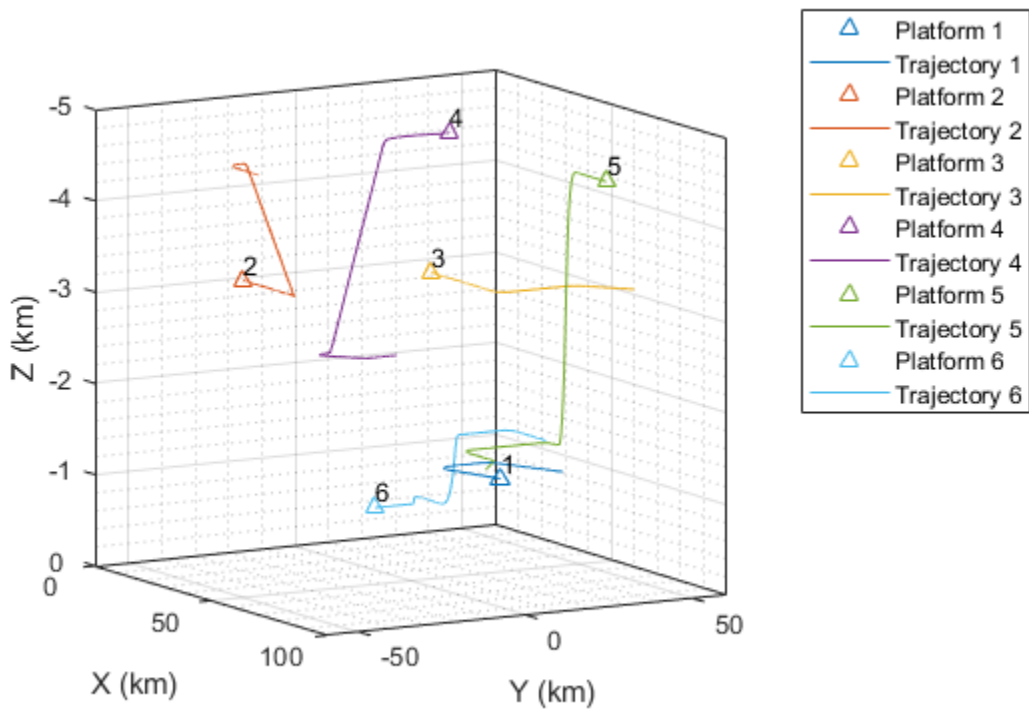
Trajectory 1

It can be instructive to view the control points used to generate the trajectories. The following figure shows the first trajectory, which is representative of a large aircraft.

The control points used to construct the path are plotted on the leftmost plot. Only a few waypoints are needed to mark the changes in curvature as the plane takes a constant turn.

The plots on the right show the altitude, magnitude of velocity (speed), and magnitude of acceleration, respectively. The speed stays nearly constant throughout despite the abrupt change in curvature. This is an advantage of using the clothoid interpolant.

```
[time, position, velocity, acceleration] = cumulativeHistory(helperPlot);
helperTrajectoryViewer(1, time, position, velocity, acceleration, trajTable);
```
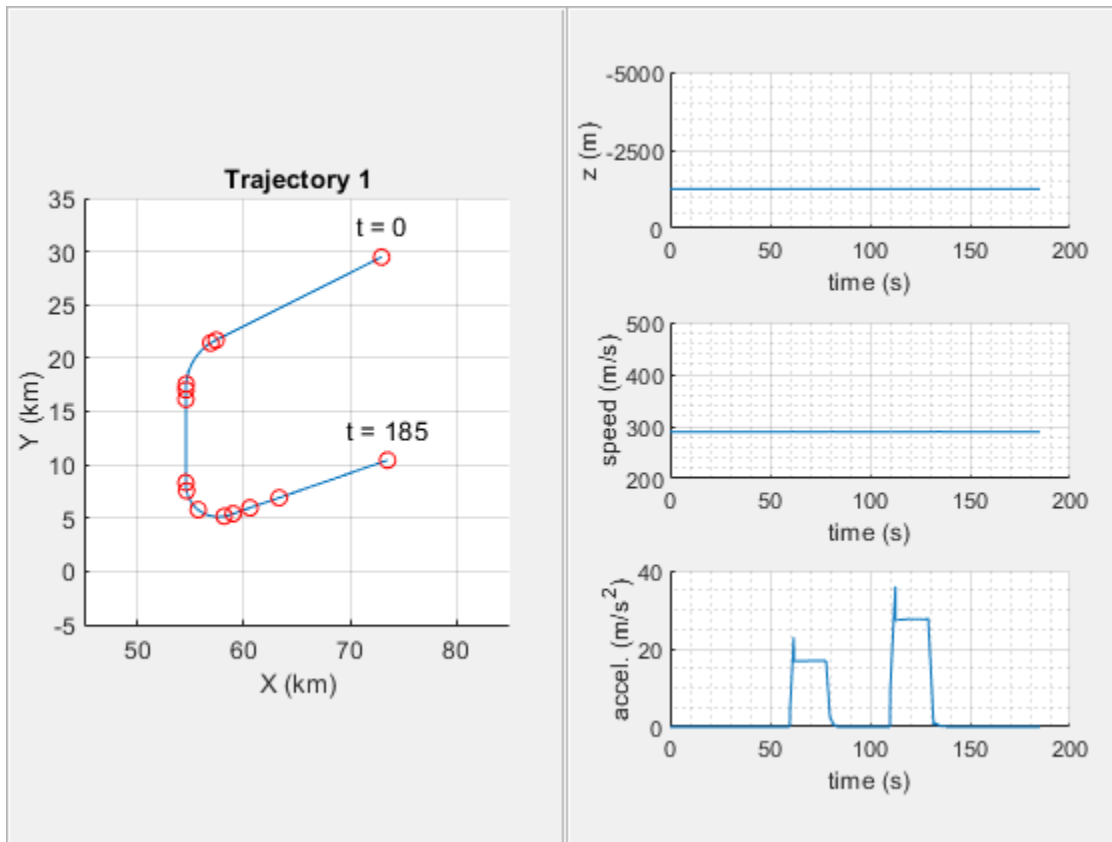
Trajectory 2

The second trajectory, shown below, represents the trajectory of a small maneuverable aircraft. It consists of two turns with several changes in acceleration immediately after the first turn and during the second turn. More waypoints are needed to adjust for these changes, but the rest of the trajectory requires fewer points.

```
helperTrajectoryViewer(2, time, position, velocity, acceleration, trajTable);
```
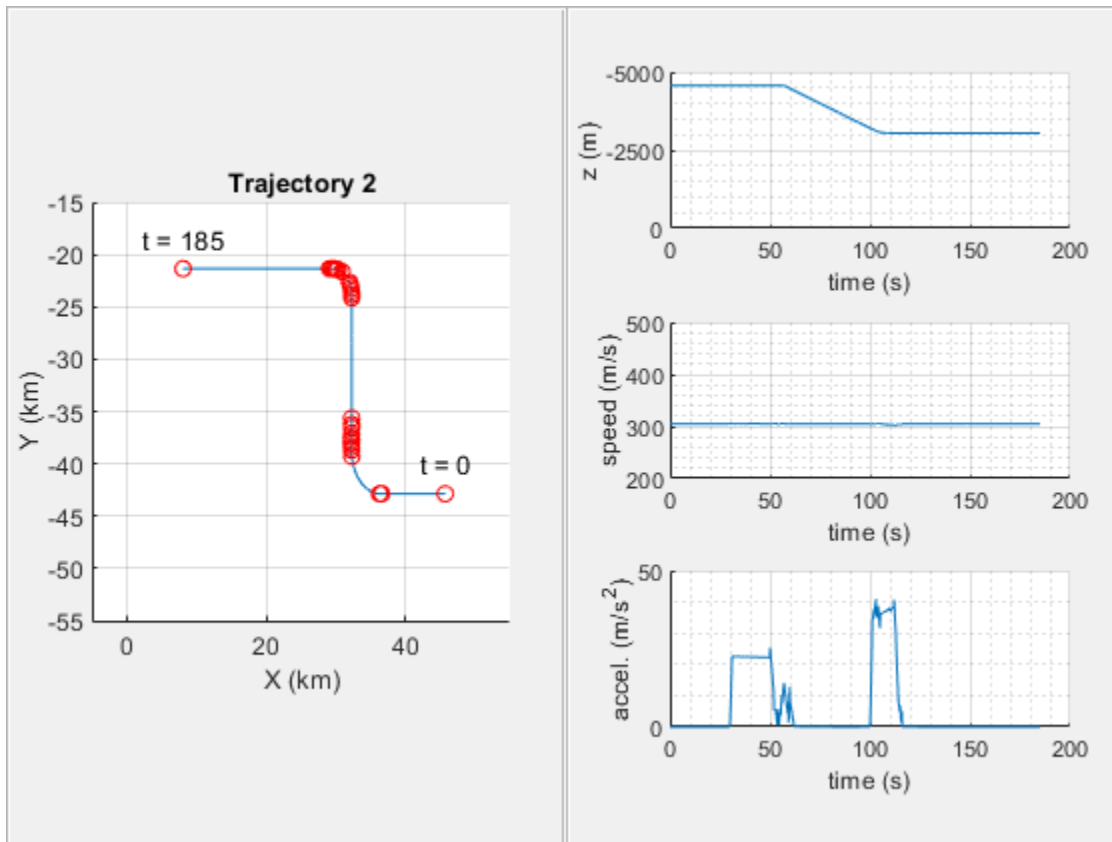
Trajectory 3

The third trajectory, shown below, is representative of a higher speed aircraft. It consists of two constant turns, where the aircraft decelerates midway throughout the second turn. You can see the control points that were used to mark the changes in velocity and acceleration in the x-y plot on the left.

```
helperTrajectoryViewer(3, time, position, velocity, acceleration, trajTable);
```
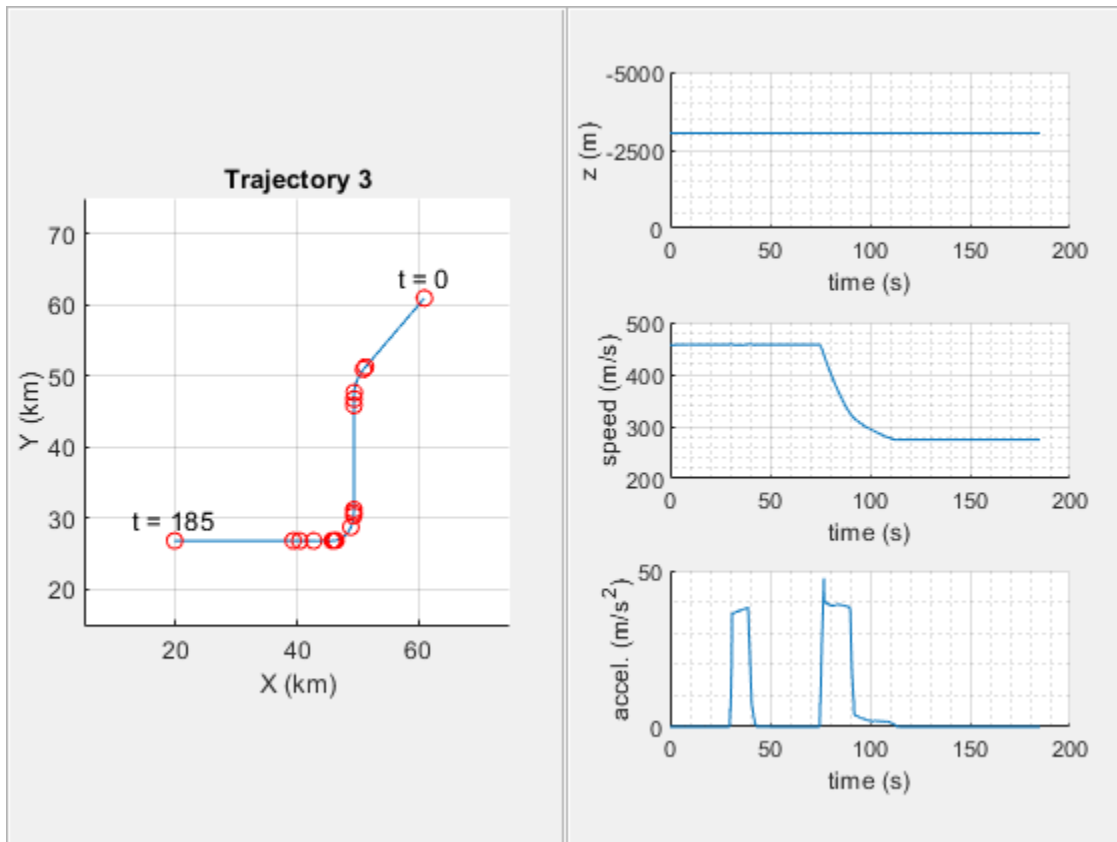
Trajectory 4

The fourth trajectory, also representative of a higher speed aircraft, is shown below. It consists of two turns, where the aircraft accelerates and climbs to a higher altitude.

```
helperTrajectoryViewer(4, time, position, velocity, acceleration, trajTable);
```
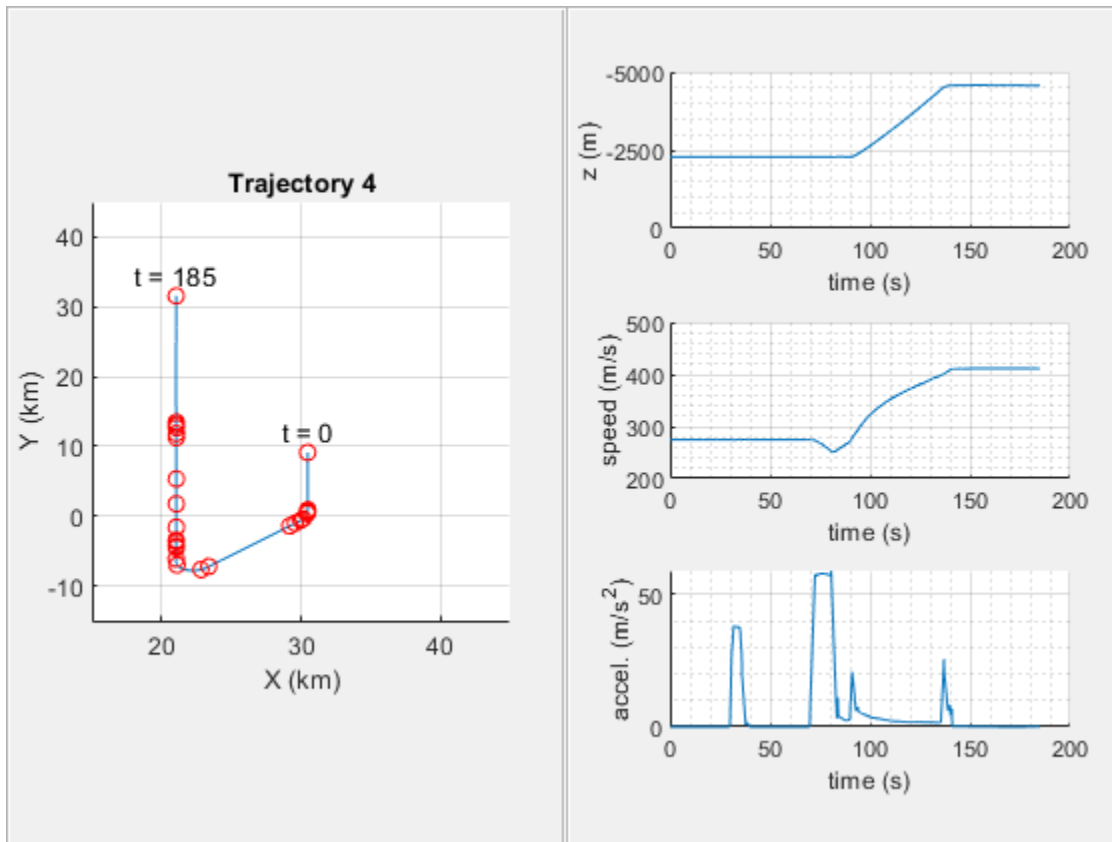
Trajectory 5

The fifth trajectory is representative of a maneuverable high-speed aircraft. It consists of three constant turns; however, it accelerates considerably throughout the duration of the flight. After the third turn the aircraft ascends to a level flight.

```
helperTrajectoryViewer(5, time, position, velocity, acceleration, trajTable);
```
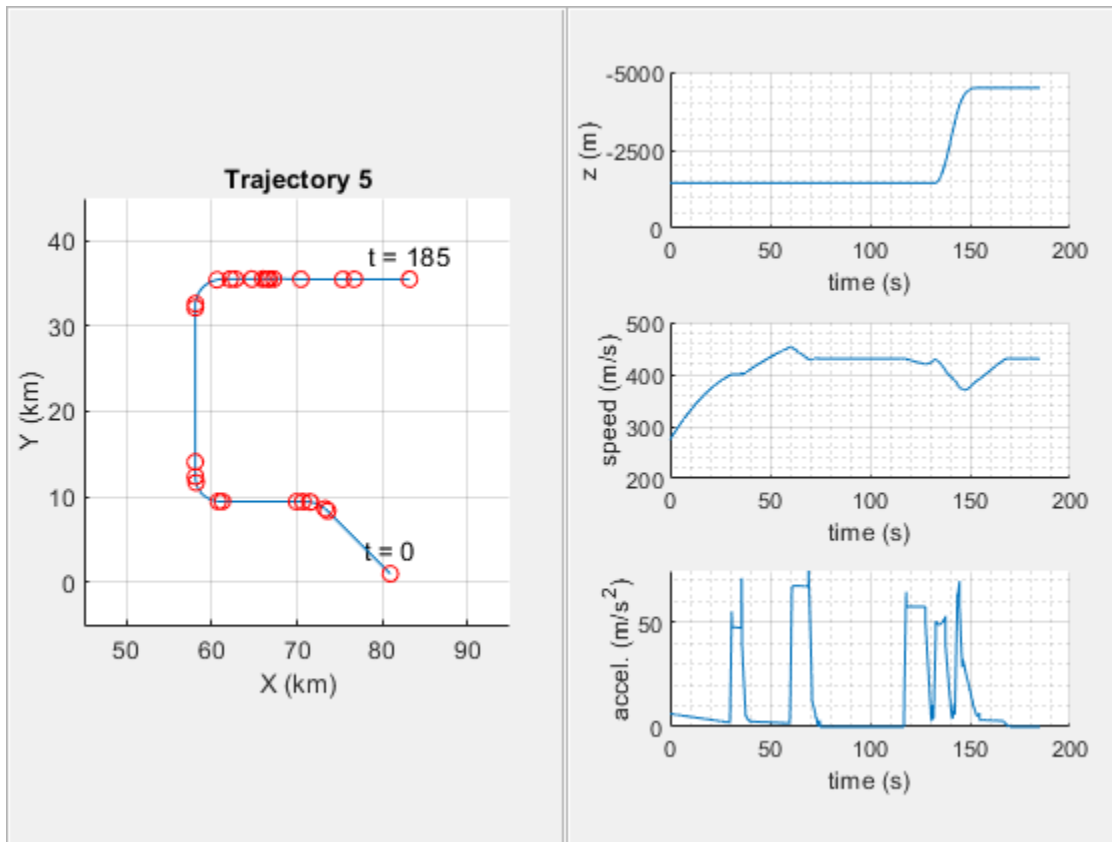
Trajectory 6

The sixth trajectory is also representative of a maneuverable high-speed aircraft. It consists of four turns. After the second turn, the aircraft decreases altitude and speed and enters the third turn. After the third turn, it accelerates rapidly and enters the fourth turn, continuing with straight and level flight.

```
helperTrajectoryViewer(6, time, position, velocity, acceleration, trajTable);
```
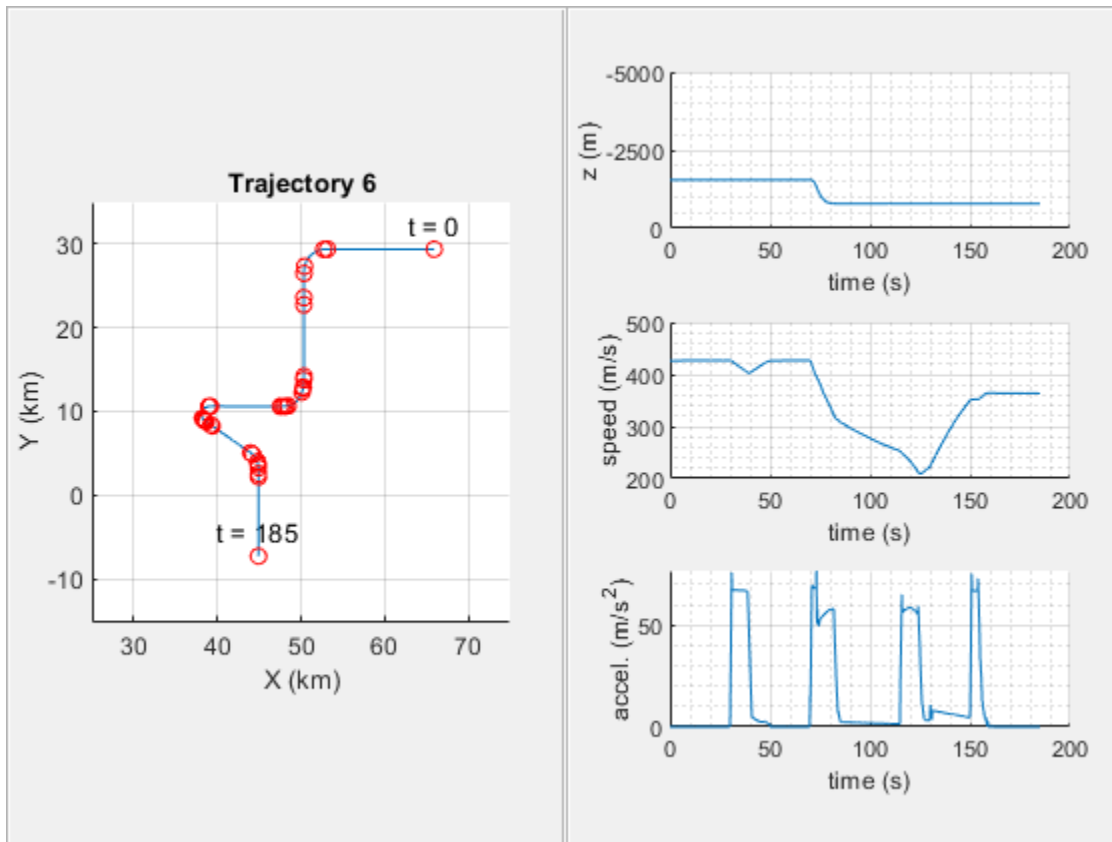
Summary

This example shows how to use `waypointTrajectory` and `radarScenario` objects to create a multi-object tracking scenario. In this example you learned the concepts behind the interpolant used inside `waypointTrajectory` and were shown how a scenario could be reproduced with a small number of waypoints.

Reference

1 Blair, W. D., et al. "Benchmark for Radar Allocation and Tracking in ECM." IEEE Transactions on Aerospace and Electronic Systems, vol. 34, no. 4, Oct. 1998, pp. 1097-114. DOI.org (Crossref), https://doi.org/10.1109/7.722694.

Simulate Radar Ghosts Due to Multipath Return

This example shows how to simulate ghost target detections and tracks due to multipath reflections, where signal energy is reflected off another target before returning to the radar. In this example you will simulate ghosts with both a statistical radar model and a transceiver model that generates IQ signals.

Motivation

Many highway scenarios involve not only other cars, but also barriers and guardrails. Consider the simple highway created using the Driving Scenario Designer (Automated Driving Toolbox) app. For more information on how to model barriers see the "Sensor Fusion Using Synthetic Radar and Vision Data" (Automated Driving Toolbox) example. Use the `helperSimpleHighwayScenarioDSD` function exported from the Driving Scenario Designer to get the highway scenario and a handle to the ego vehicle.

```
% Set random seed for reproducible results
rndState = rng('default');

% Create scenario using helper
[scenario, egoVehicle] = helperSimpleHighwayScenarioDSD();
```

To model the detections generated by a forward-looking automotive radar, use the `radarDataGenerator System object`™. Use a 77 GHz center frequency, which is typical for automotive radar. Generate detections up to 150 meters in range and with a radial speed up to 100 m/s.

```
% Automotive radar system parameters
freq = 77e9; % Hz
rgMax = 150; % m
spMax = 100; % m/s
rcs = 10;    % dBsm

azRes = 4;   % deg
rgRes = 2.5; % m
rrRes = 0.5; % m/s

% Create a forward-looking automotive radar
rdg = radarDataGenerator(1, 'No scanning', ...
    'UpdateRate', 10, ...
    'MountingLocation', [3.4 0 0.2], ...
    'CenterFrequency', freq, ...
    'HasRangeRate', true, ...
    'FieldOfView', [70 5], ...
    'RangeLimits', [0 rgMax], ...
    'RangeRateLimits', [-spMax spMax], ...
    'HasRangeAmbiguities',true, ...
    'MaxUnambiguousRange', rgMax, ...
    'HasRangeRateAmbiguities',true, ...
    'MaxUnambiguousRadialSpeed', spMax, ...
    'ReferenceRange', rgMax, ...
    'ReferenceRCS',rcs, ...
    'AzimuthResolution',azRes, ...
    'RangeResolution',rgRes, ...
    'RangeRateResolution',rrRes, ...
```

```
        'TargetReportFormat', 'Detections', ...
        'Profiles',actorProfiles(scenario));

% Create bird's eye plot and detection plotter function
[~,detPlotterFcn] = helperSetupBEP(egoVehicle,rdg);
title('Free Space (no multipath)');

% Generate raw detections
time = scenario.SimulationTime;
tposes = targetPoses(egoVehicle);
[dets,~,config] = rdg(tposes,time);

% Plot detections
detPlotterFcn(dets,config);
```
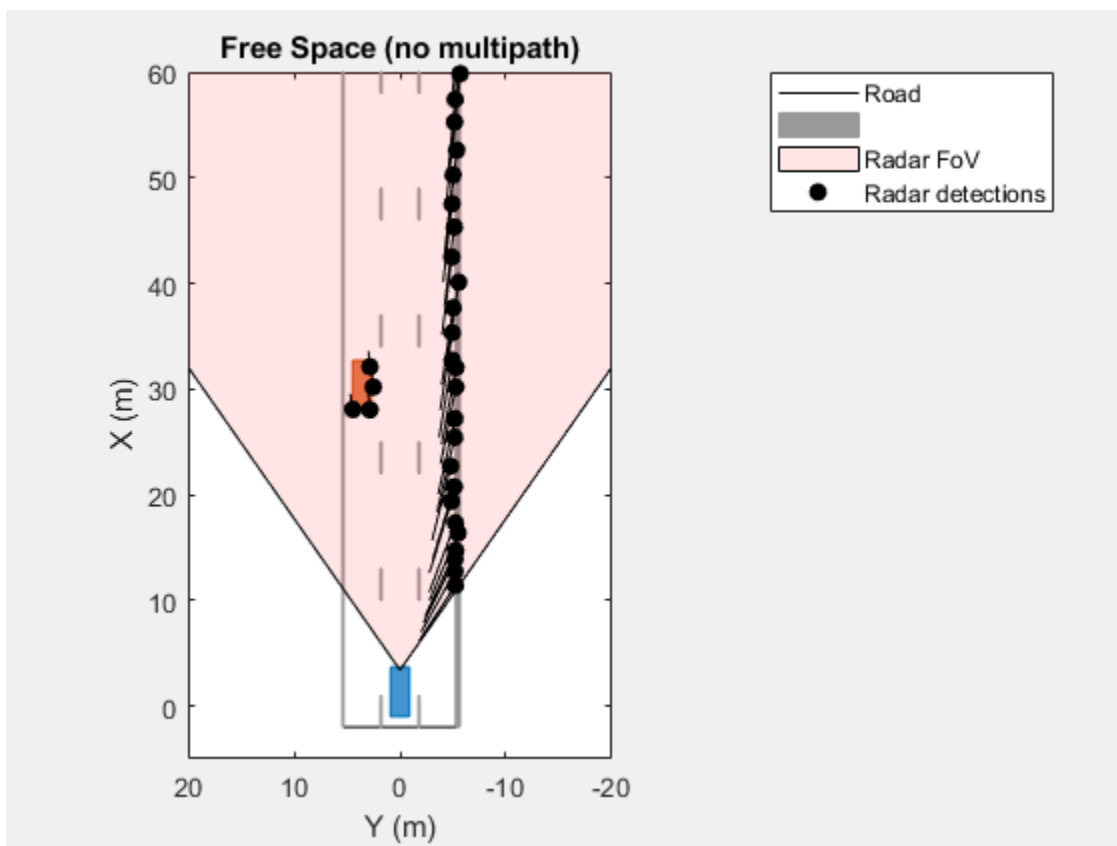

This figure shows the locations of the detections along the target vehicle as well as along the side of the barrier. However, detections are not always so well-behaved. One phenomenon that can pose considerable challenges to radar engineers is multipath. Multipath is when the signal not only propagates directly to the intended target and back to the radar but also includes additional reflections off objects in the environment.

Multipath Reflections

When a radar signal propagates to a target of interest it can arrive through various paths. In addition to the direct path from the radar to the target and then back to the radar, there are other possible propagation paths. The number of paths is unbounded, but with each reflection, the signal energy

decreases. Commonly, a propagation model considering three-bounce paths is used to model this phenomenon.

To understand the three-bounce model, first consider the simpler one-bounce and two-bounce paths, as shown in these figures.
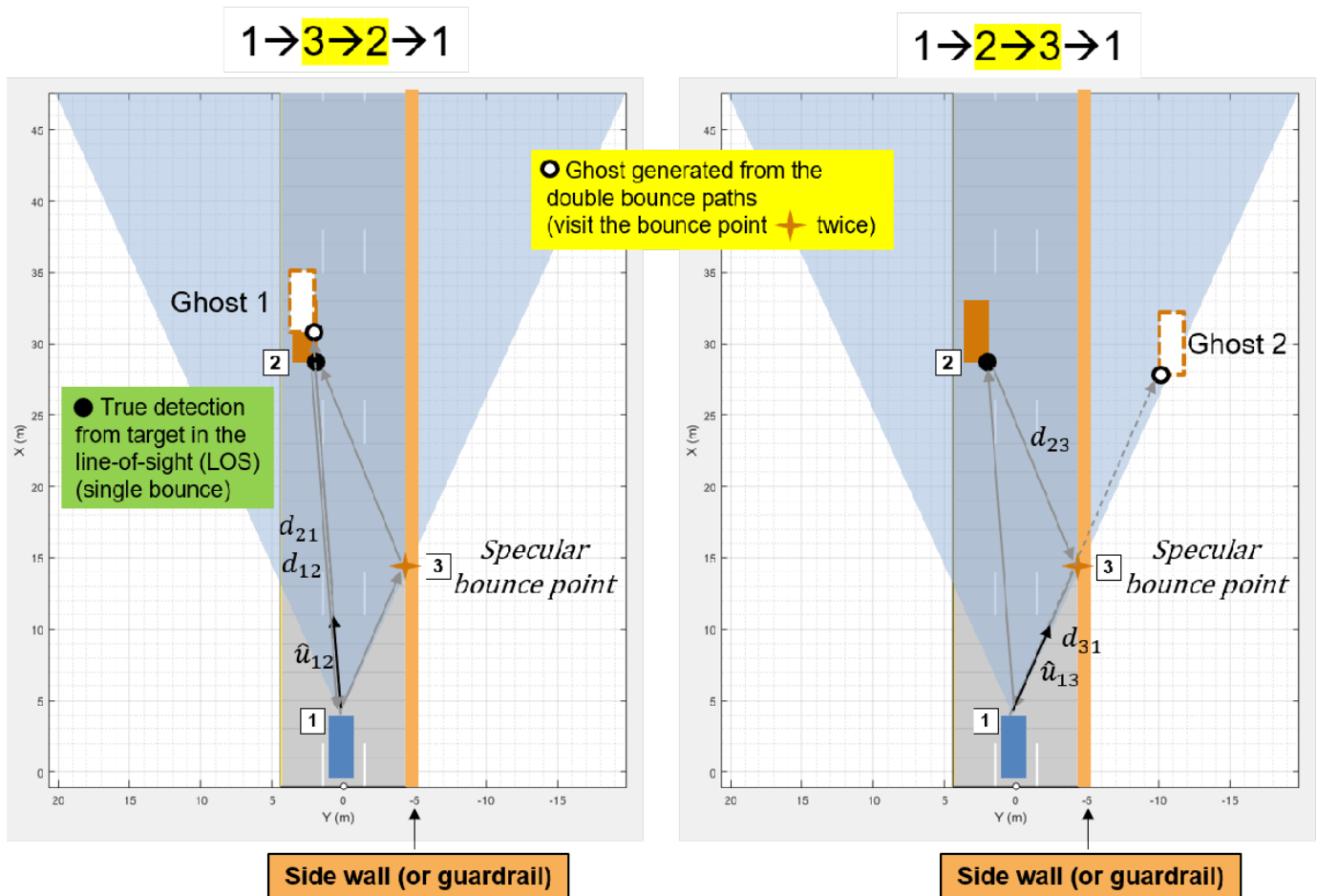
One-Bounce Path

The one-bounce path propagates from the radar (1) to the target (2) and then is reflected from the target (2) back to the radar. This is often referred to as the direct or line-of-sight path.

Two-Bounce Paths

In this case, there are two unique propagation paths that consist of two bounces.

The first two-bounce path propagates from the radar (1) to a reflecting surface (3), then to the target (2) before returning to the radar (1). Because the signal received at the radar arrives from the last bounce from the true target, it generates ghost detections along the same direction as the true target. Because the path length for this propagation is longer, it appears at a farther range than the true target detections.

The second two-bounce path propagates from the radar (1) to the target (2), then to the reflecting surface (3) before returning to the radar (1). In this case, the ghost detections appear on the other side of the reflecting surface as the radar receives the reflected signal in that direction.

Notice that the path length for both two-bounce paths is the same. As a result, the measured range and range rate for these paths will be the same as well.

Three-Bounce Path
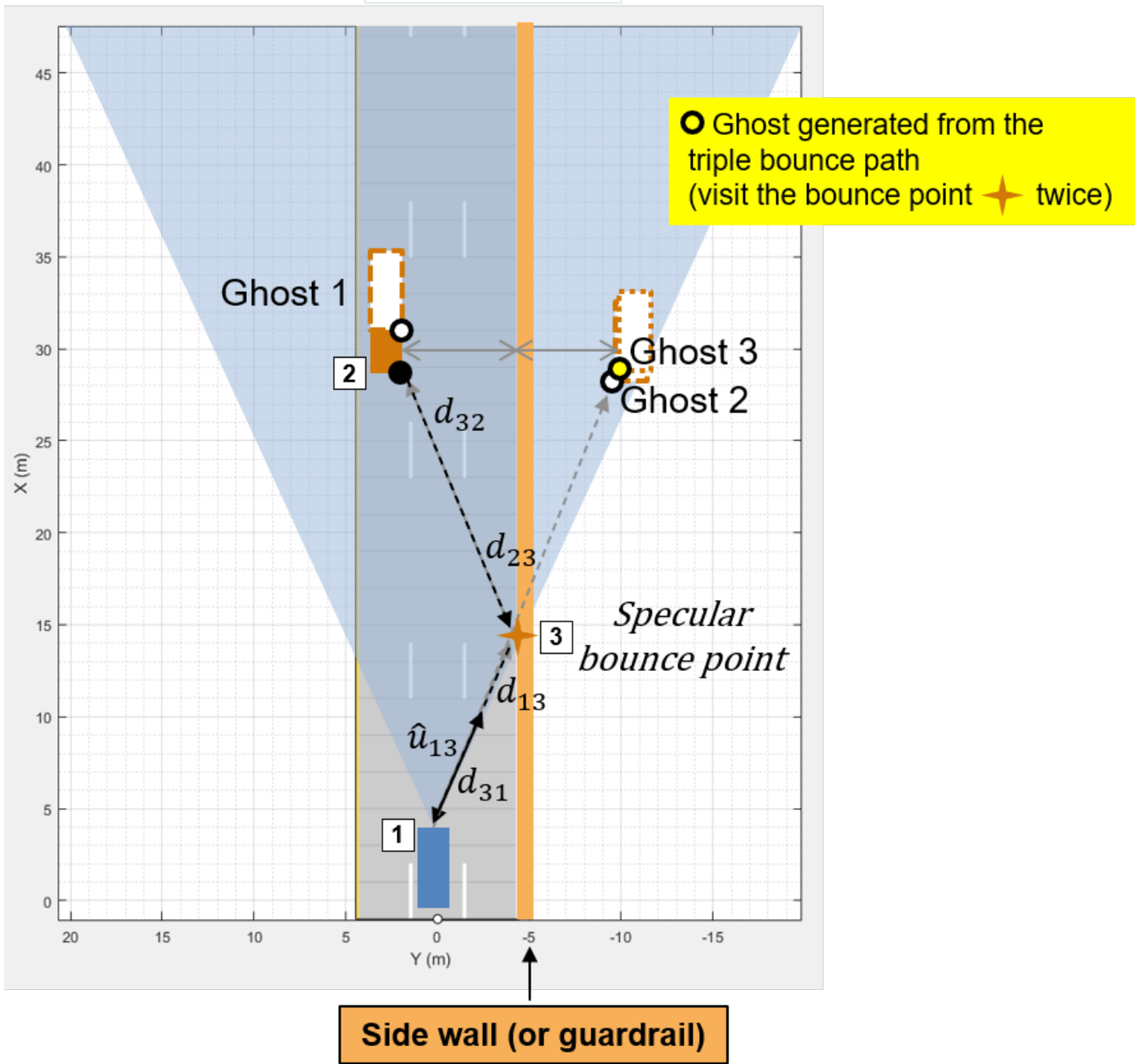
The three-bounce path reflects off the barrier twice. This path never propagates directly to the target or directly back to the radar. The three-bounce ghost detections appear on the other side of the reflecting surface as the radar receives the reflected signal in that direction. Additionally, it has the longest propagation path of the three-bounce paths and therefore has the longest measured range of

the three paths. This path corresponds to a mirror reflection of the true target on the other side of the barrier.

Model Ghost Target Detections

Set the `HasGhosts` property on the `radarDataGenerator` to model the detections generated from these three-bounce paths.

```
% Enable ghost target model
release(rdg);
rdg.HasGhosts = true;

% Generate raw detections
time = scenario.SimulationTime;
tposes = targetPoses(egoVehicle);
[dets,~,config] = rdg(tposes,time);

% Plot detections
detPlotterFcn(dets,config);
title('Simple Multipath Environment');
```
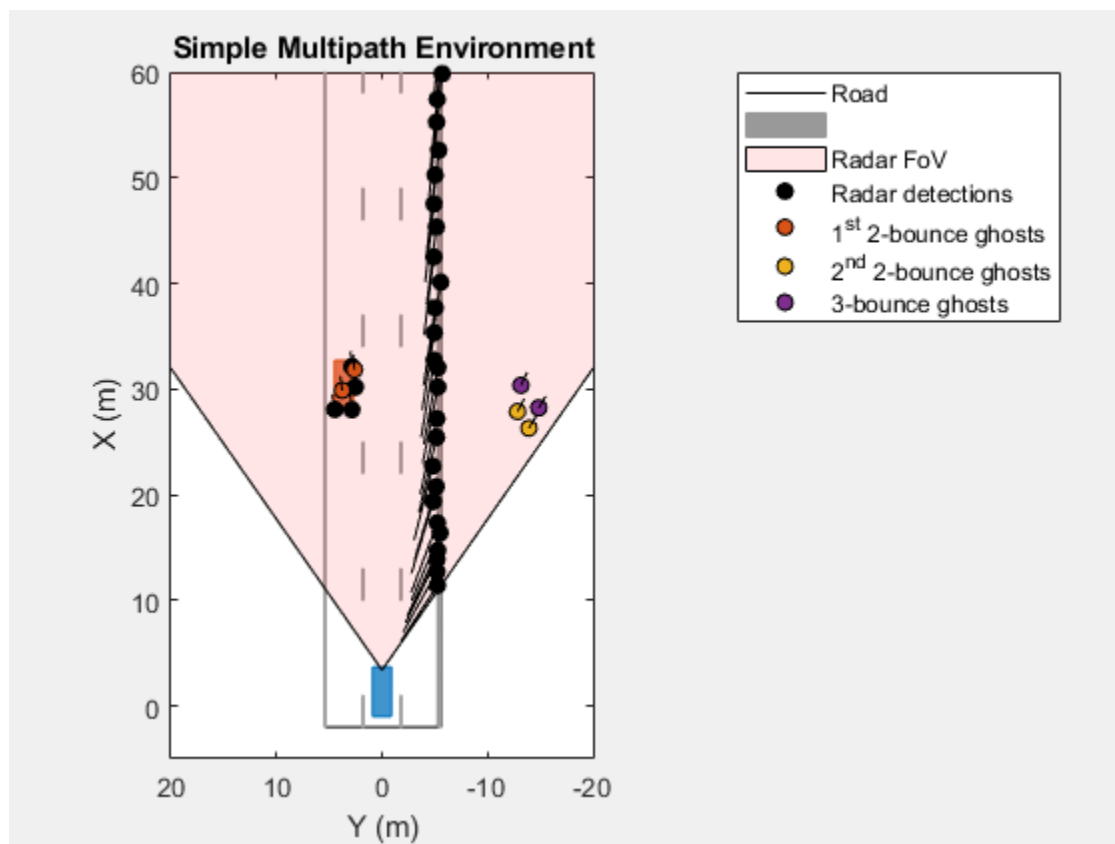

This figure reproduces the analysis of the three propagation paths. The first two-bounce ghosts lie in the direction of the target at a slightly longer range than the direct-path detections. The second two-bounce and three-bounce ghosts lie in the direction of the mirrored image of the target generated by the reflection from the barrier.

Ghost Tracks

Because the range and velocities of the ghost target detections are like the range and velocity of the true targets, they have kinematics that are consistent for a tracker that is configured to track the true target detections. This consistency between the kinematics of real and ghost targets results in tracks being generated for the ghost target on the other side of the barrier.

Set the `TargetReportFormat` property on `radarDataGenerator` to `Tracks` to model the tracks generated by a radar in the presence of multipath.

```matlab
% Output tracks instead of detections
release(rdg);
rdg.TargetReportFormat = 'Tracks';
rdg.ConfirmationThreshold = [2 3];
rdg.DeletionThreshold = [5 5];
FilterInitializationFcn = 'initcvekf'; % constant-velocity EKF

% Create a new bird's eye plot to plot the tracks
[bep,trkPlotterFcn] = helperSetupBEP(egoVehicle,rdg);
title('Simple Multipath Environment');

% Run simulation
restart(scenario);
scenario.StopTime = 7.5;
while advance(scenario)
    time = scenario.SimulationTime;
    tposes = targetPoses(egoVehicle);

    % Generate tracks
    [trks,~,config] = rdg(tposes,time);

    % Filter out tracks corresponding to static objects (e.g. barrier)
    dyntrks = helperKeepDynamicObjects(trks, egoVehicle);

    % Visualize dynamic tracks
    helperPlotScenario(bep,egoVehicle);
    trkPlotterFcn(dyntrks,config);
end
```
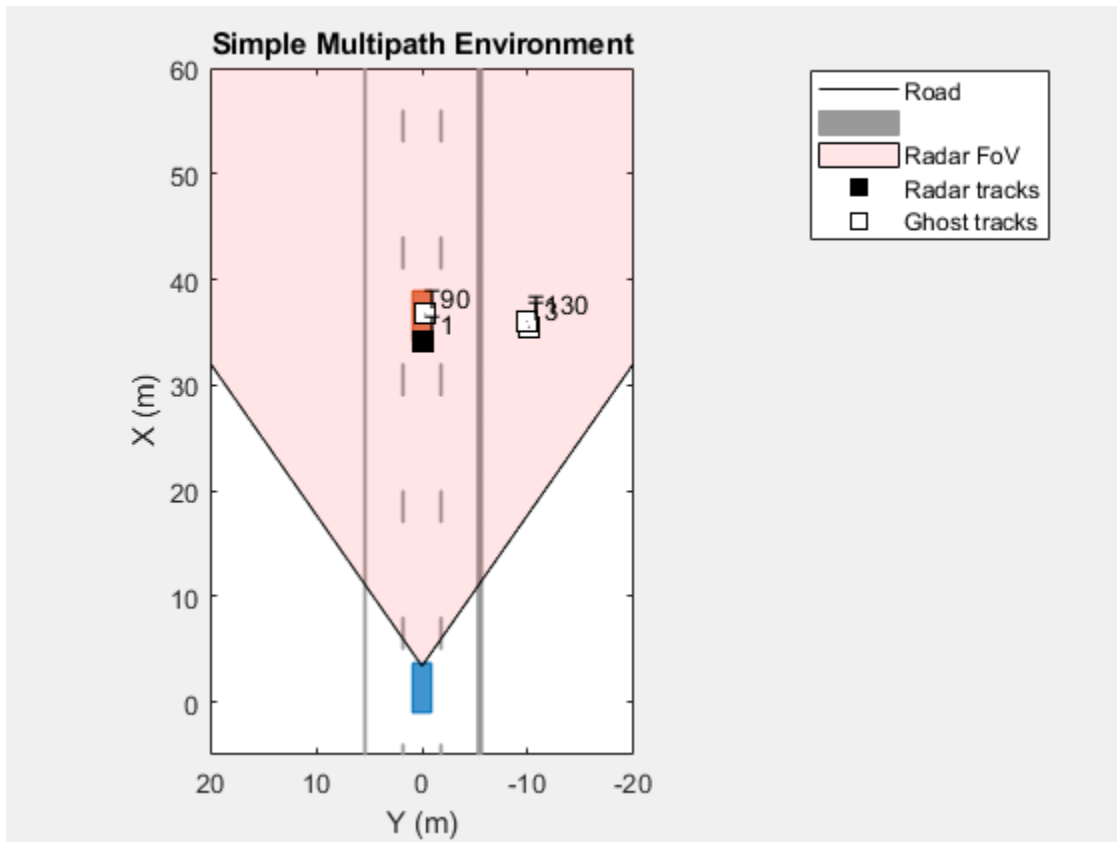
This figure shows the confirmed track positions using square markers. The tracks corresponding to static objects (for example a barrier) are not plotted. Notice that there are multiple tracks associated with the lead car. The tracks that overlay the lead car correspond to the true detection and the first two-bounce ghost. The tracks that lie off of the road on the other side of the guardrail correspond to the second two-bounce and three-bounce ghosts.

The track velocities are indicated by the length and direction of the vectors pointing away from the track position (these are small because they are relative to the ego vehicle). Ghost detections may fool a tracker because they have kinematics like the kinematics of the true targets. These ghost tracks can be problematic as they add an additional processing load to the tracker and can confuse control decisions using the target tracks.

Model IQ Signals

In the previous free-space and multipath simulations in this example, you used measurement-level radar models to generate detections and tracks. Now, use the `radarTransceiver` System object to generate time-domain IQ signals. Create an equivalent `radarTransceiver` directly from the `radarDataGenerator`.

The statistical radar has the following range and range-rate (Doppler) parameters which determine the constraints for the waveform used by the `radarTransceiver`.

```
rgMax = rdg.RangeLimits(2)        % m
```

rgMax = 150

```
spMax = rdg.RangeRateLimits(2)    % m/s
```

```
spMax = 100
```

Compute the pulse repetition frequency (PRF) that will satisfy the range rate for the radar.

```
lambda = freq2wavelen(rdg.CenterFrequency);
prf = 2*speed2dop(2*spMax,lambda);
```

Compute the number of pulses needed to satisfy the range-rate resolution requirement.

```
rrRes = rdg.RangeRateResolution
```

```
rrRes = 0.5000
```

```
dopRes = 2*speed2dop(rrRes,lambda);
numPulses = 2^nextpow2(prf/dopRes)
```

```
numPulses = 512
```

```
prf = dopRes*numPulses
```

```
prf = 1.3150e+05
```

Confirm that the unambiguous range that corresponds to this PRF is beyond the maximum range limit.

```
rgUmb = time2range(1/prf)
```

```
rgUmb = 1.1399e+03
```

Re-set some properties of the `radarDataGenerator` for better comparison to IQ results from the `radarTransceiver`.

```
release(rdg);

% Set the range and range-rate ambiguities according to desired PRF and
% number of pulses
rdg.MaxUnambiguousRange = rgUmb;
rdg.MaxUnambiguousRadialSpeed = spMax;

% Set the statistical radar to report clustered detections to compare to
% the IQ video from the radar transceiver.
rdg.TargetReportFormat = 'Clustered detections';
rdg.DetectionCoordinates = 'Body';
```

Construct the equivalent `radarTransceiver` directly from the `radarDataGenerator`. By default, the `radarTransceiver` uses a unit-gain rectangular antenna pattern with azimuth and elevation beamwidths equal to the azimuth and elevation resolutions of the `radarDataGenerator`. To simulate a radar with a transmit pattern that covers the desired field of view, first set the azimuth resolution of the `radarDataGenerator` equal to the field of view.

```
azRes = rdg.AzimuthResolution;
rdg.AzimuthResolution = rdg.FieldOfView(1);

% Construct the radar transceiver from the radar data generator
rtxrx = radarTransceiver(rdg)
```

```
rtxrx =
  radarTransceiver with properties:
```

```
                Waveform: [1x1 phased.RectangularWaveform]
             Transmitter: [1x1 phased.Transmitter]
         TransmitAntenna: [1x1 phased.Radiator]
          ReceiveAntenna: [1x1 phased.Collector]
                Receiver: [1x1 phased.ReceiverPreamp]
       MechanicalScanMode: 'None'
        ElectronicScanMode: 'None'
          MountingLocation: [3.4000 0 0.2000]
            MountingAngles: [0 0 0]
      NumRepetitionsSource: 'Property'
            NumRepetitions: 512
         RangeLimitsSource: 'Property'
               RangeLimits: [0 150]
           RangeOutputPort: false
            TimeOutputPort: false
```

```
% Restore the desired azimuth resolution for the statistical radar
rdg.AzimuthResolution = azRes;
```

The statistical radar emulates the use of a uniform linear array (ULA) to form multiple receive beams, as needed to achieve the specified azimuth resolution. The number or elements required is determined by the radar's azimuth resolution and wavelength, and can be found with the `beamwidth2ap` function.

```
numRxElmt = ceil(beamwidth2ap(rdg.AzimuthResolution,lambda,0.8859)/(lambda/2))
```

```
numRxElmt = 26
```

Attach a ULA to the receive antenna of the `radarTransceiver`.

```
elmt = rtxrx.ReceiveAntenna.Sensor;
rxarray = phased.ULA(numRxElmt,lambda/2,'Element',elmt);
rtxrx.ReceiveAntenna.Sensor = rxarray;
```

Generate IQ Samples

Use the `helper3BounceGhostPaths function` to compute the three-bounce paths for the target and sensor positions from the multipath scenario.

```
restart(scenario);
tposes = targetPoses(egoVehicle);
```

```
% Generate 3-bounce propagation paths for the targets in the scenario
paths = helper3BounceGhostPaths(tposes,rdg);
```

Use the `radarTransceiver` to generate the baseband sampled IQ data received by the radar.

```
time = scenario.SimulationTime; % Current simulation time
Xcube = rtxrx(paths,time);      % Generate IQ data for transceiver from the 3-bounce path model
```

Range and Doppler Processing

The received data cube has the three-dimensions: fast-time samples, receive antenna element, and slow-time samples.

```
size(Xcube)
```

```
ans = 1×3
```

```
    64    26   512
```

Use the `phased.RangeDopplerResponse` System object to perform range and Doppler processing along the first and third dimensions of the data cube.

```
rngdopproc = phased.RangeDopplerResponse( ...
    'RangeMethod','Matched filter', ...
    'DopplerOutput','Speed', ...
    'PropagationSpeed',rtxrx.ReceiveAntenna.PropagationSpeed, ...
    'OperatingFrequency',rtxrx.ReceiveAntenna.OperatingFrequency, ...
    'SampleRate',rtxrx.Receiver.SampleRate);
mfcoeff = getMatchedFilter(rtxrx.Waveform);
[Xrngdop,rggrid,rrgrid] = rngdopproc(Xcube,mfcoeff);
```

Beamforming

Use the `phased.PhaseShiftBeamformer` System object to form beams from the receive antenna array elements along the second dimension of the data cube.

```
azFov = rdg.FieldOfView(1);
anggrid = -azFov/2:azFov/2;
bmfwin = @(N)normmax(taylorwin(N,5,-60));
beamformer = phased.PhaseShiftBeamformer( ...
        'Direction',[anggrid;0*anggrid],...
        'SensorArray',rtxrx.ReceiveAntenna.Sensor, ...
        'OperatingFrequency',rtxrx.ReceiveAntenna.OperatingFrequency);
Xbfmrngdop = Xrngdop;
[Nr,Ne,Nd] = size(Xbfmrngdop);
Xbfmrngdop = permute(Xbfmrngdop,[1 3 2]); % Nr x Nd x Ne
Xbfmrngdop = reshape(Xbfmrngdop,[],Ne);
Xbfmrngdop = beamformer(Xbfmrngdop.*bmfwin(Ne)');
Xbfmrngdop = reshape(Xbfmrngdop,Nr,Nd,[]); % Nr x Nd x Nb
Xbfmrngdop = permute(Xbfmrngdop,[1 3 2]); % Nr x Nb x Nd
```

Use the `helperPlotBeamformedRangeDoppler` function to plot the range-angle map from the beamformed, range, and Doppler processed data cube.

```
helperPlotBeamformedRangeDoppler(Xbfmrngdop,rggrid,anggrid,rtxrx);
```
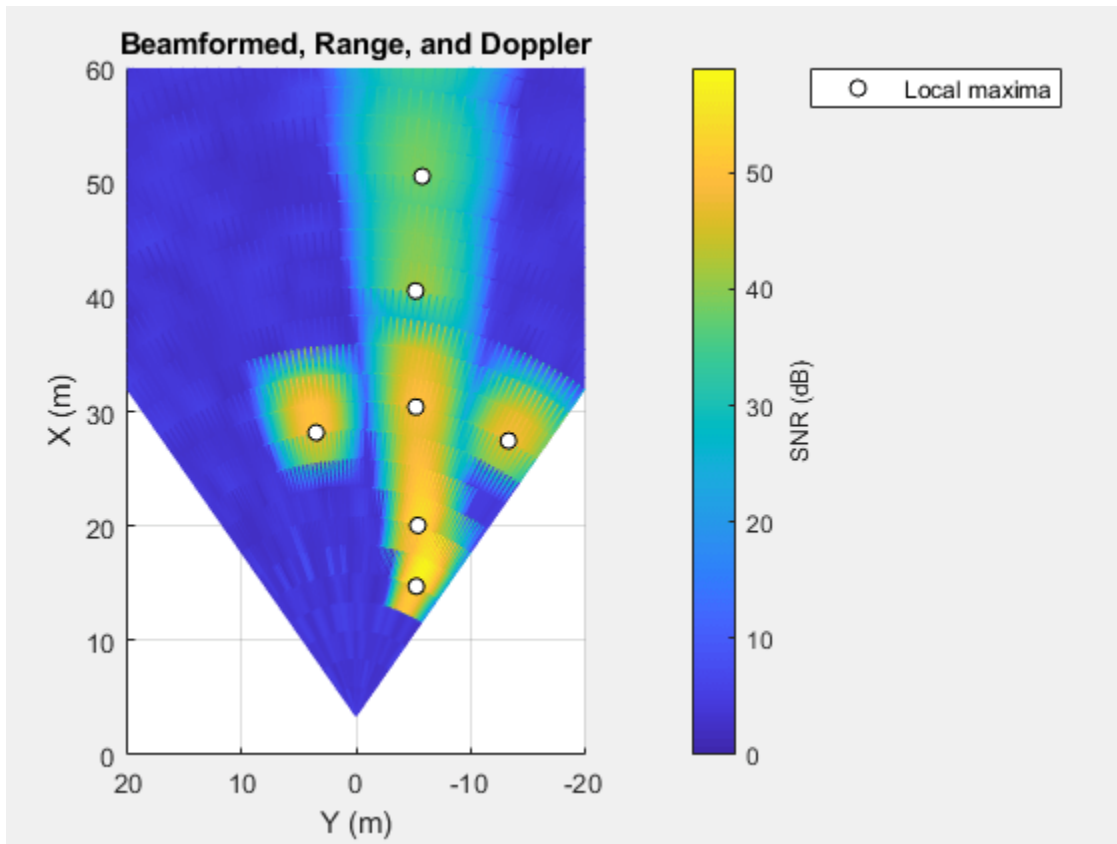
The local maxima of the received signals correspond to the location of the target vehicle, the guardrail, and the ghost image of the target vehicle on the other side of the guardrail. Show that measurement-level detections generated by `radarDataGenerator` are consistent with the peaks in the range-angle map generated by the equivalent `radarTransceiver`.

Use the `helperPlayStatAndIQMovie` function to compare the measurement-level detections and IQ processed video for the duration of this scenario.

```
helperPlayStatAndIQMovie(scenario,egoVehicle,rtxrx,rdg,rngdopproc,beamformer,bmfwin);
```
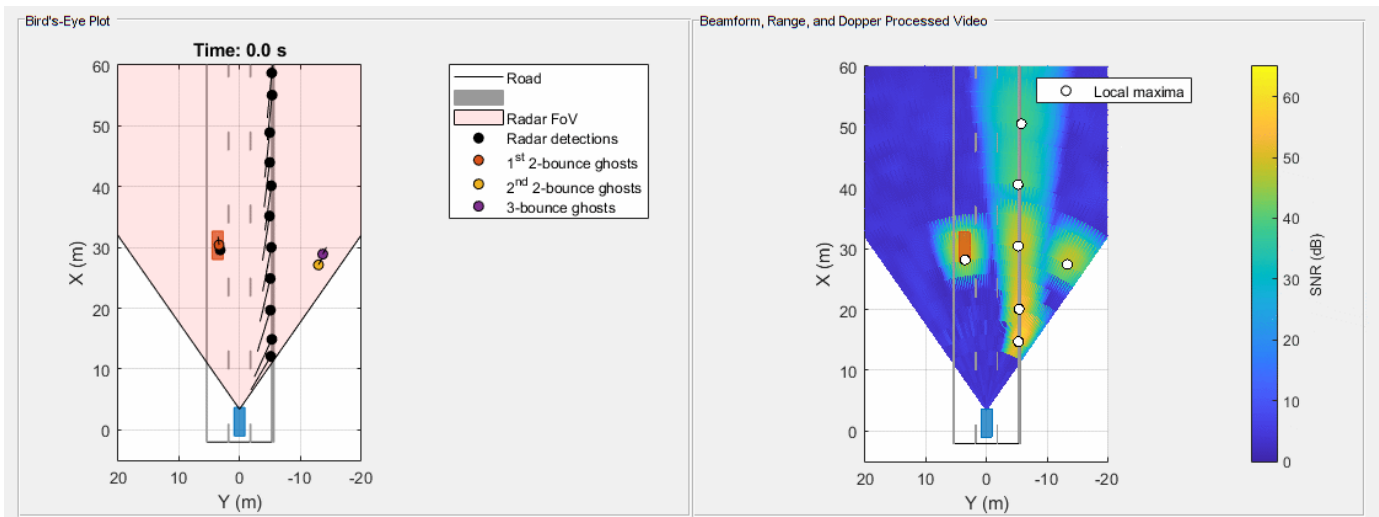
Multipath Ground Bounce

Multipath ghost detections can be used at times to see objects in the road that would otherwise not be detected by the radar due to occlusion. One example is the detection of an occluded vehicle due to multipath off of the road surface. Use the `helperGroundBounceScenarioDSD` function to create a scenario where a slower moving vehicle in the same lane as the ego vehicle is occluded by another vehicle directly in front of the radar.

```
[scenario, egoVehicle] = helperGroundBounceScenarioDSD;
ax3d = helperRadarChasePlot(egoVehicle);
```
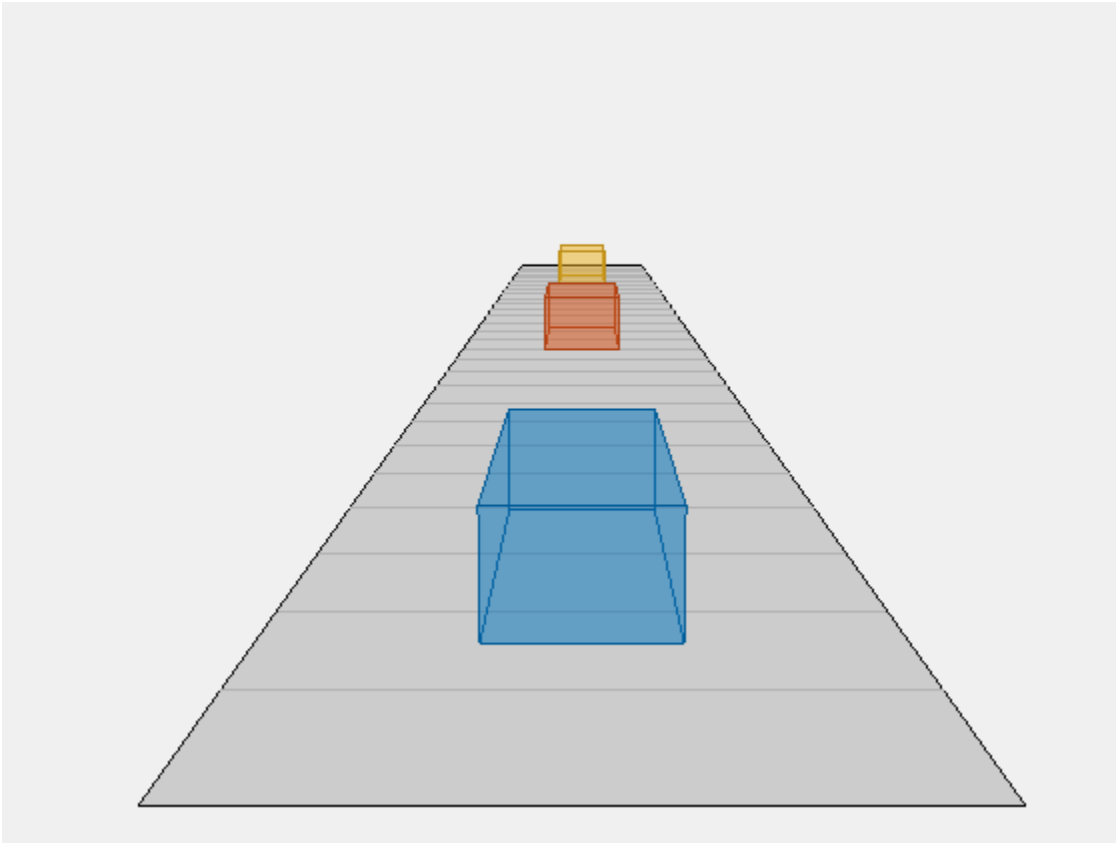
The yellow car is occluded by the red car. A line of sight does not exist between the blue ego car's forward looking-radar and the yellow car.

```
viewLoc = [scenario.Actors(2).Position(1)-10 -10];
chasePlot(egoVehicle,'ViewLocation',viewLoc,'ViewHeight',0,'ViewYaw',40,'Parent',ax3d);
```
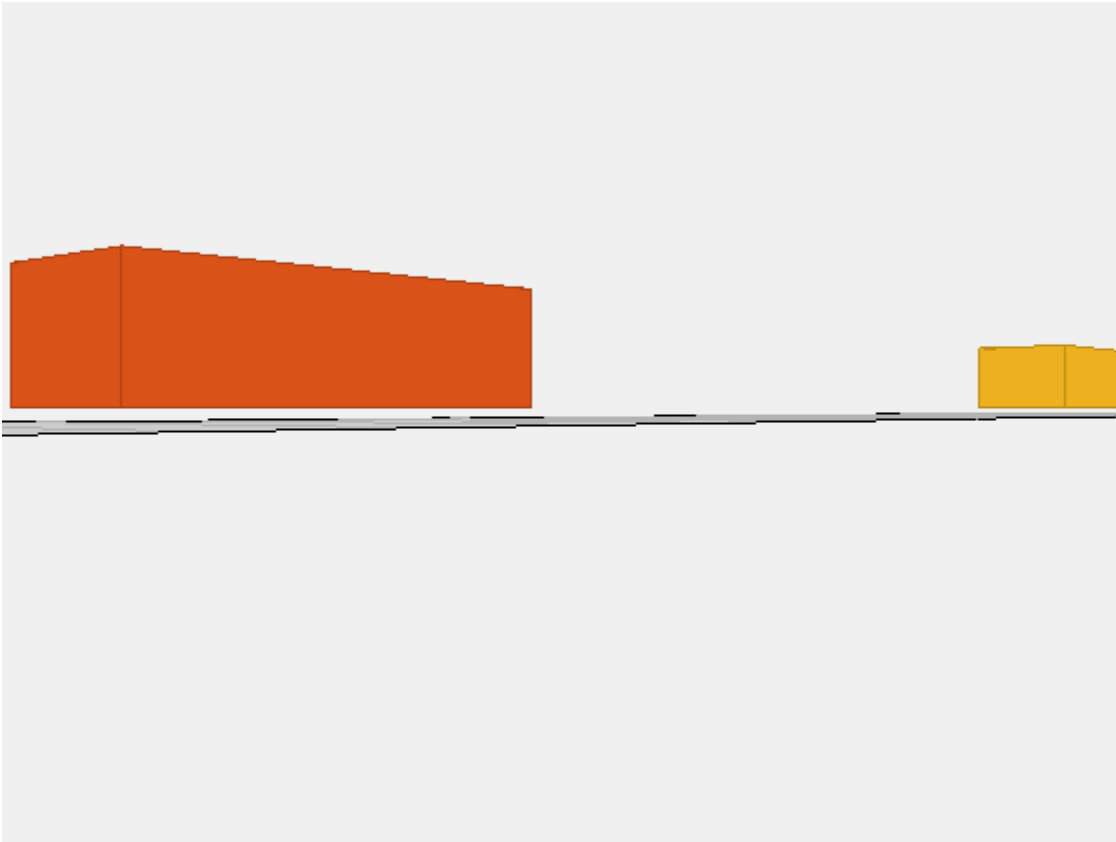
Multipath can pass through the space between the underside of a car and the surface of the road.

Reuse the `radarDataGenerator` to generate ghost target detections due to multipath between the vehicles and the road surface. Use the `helperRoadProfiles` and `helperRoadPoses` functions to include the road surface in the list of targets modeled in the scenario to enable multipath between the road surface and the vehicles.

```
release(rdg);
rdg.RangeRateResolution = 0.5;
rdg.FieldOfView(2) = 10;
rdg.TargetReportFormat = 'Detections';

tprofiles = actorProfiles(scenario);
rdprofiles = helperRoadProfiles(scenario);
rdg.Profiles = [tprofiles;rdprofiles];

% Create bird's eye plot and detection plotter function
[bep,detPlotterFcn] = helperSetupBEP(egoVehicle,rdg);
[ax3d,detChasePlotterFcn] = helperRadarChasePlot(egoVehicle,rdg);
camup(ax3d,[0 0 1]);
pos = egoVehicle.Position+[-5 -5 0];
campos(ax3d,pos);
camtarget(ax3d,[15 0 0]);

% Generate clustered detections
time = scenario.SimulationTime;
tposes = targetPoses(egoVehicle);
```

```
rdposes = helperRoadPoses(egoVehicle);
poses = [tposes rdposes];

[dets,~,config] = rdg(poses,time);

% Plot detections
dyndets = helperKeepDynamicObjects(dets,egoVehicle);
detPlotterFcn(dyndets,config);
```
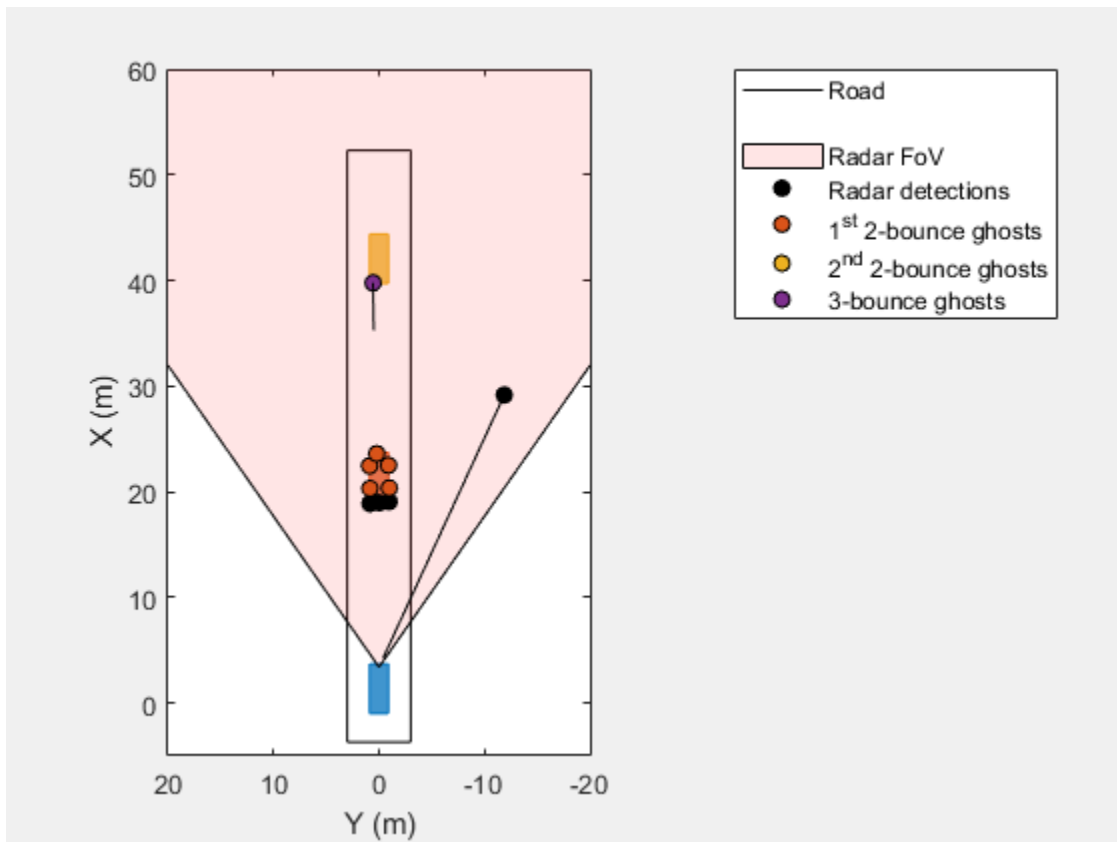

The detection from the occluded car is possible due to the three-bounce path that exists between the road surface and the underside of the red car.

```
% Find the 3-bounce detection from the occluded car
i3 = find(cellfun(@(d)d.ObjectAttributes{1}.BouncePathIndex,dyndets)==3);
det3 = dyndets{i3};

% Plot the 3-bounce path between the radar and the occluded car
iBncTgt = find([poses.ActorID]==det3.ObjectAttributes{1}.BounceTargetIndex);
iTgt = find([poses.ActorID]==det3.ObjectAttributes{1}.TargetIndex);
pos = [rdg.MountingLocation;poses(iBncTgt).Position;poses(iTgt).Position]+egoVehicle.Position;
hold(ax3d,'on');
plot3(ax3d,pos(:,1),pos(:,2),pos(:,3),'r-*','LineWidth',2);
campos([-6 -15 2]); camtarget([17 0 0]);
```
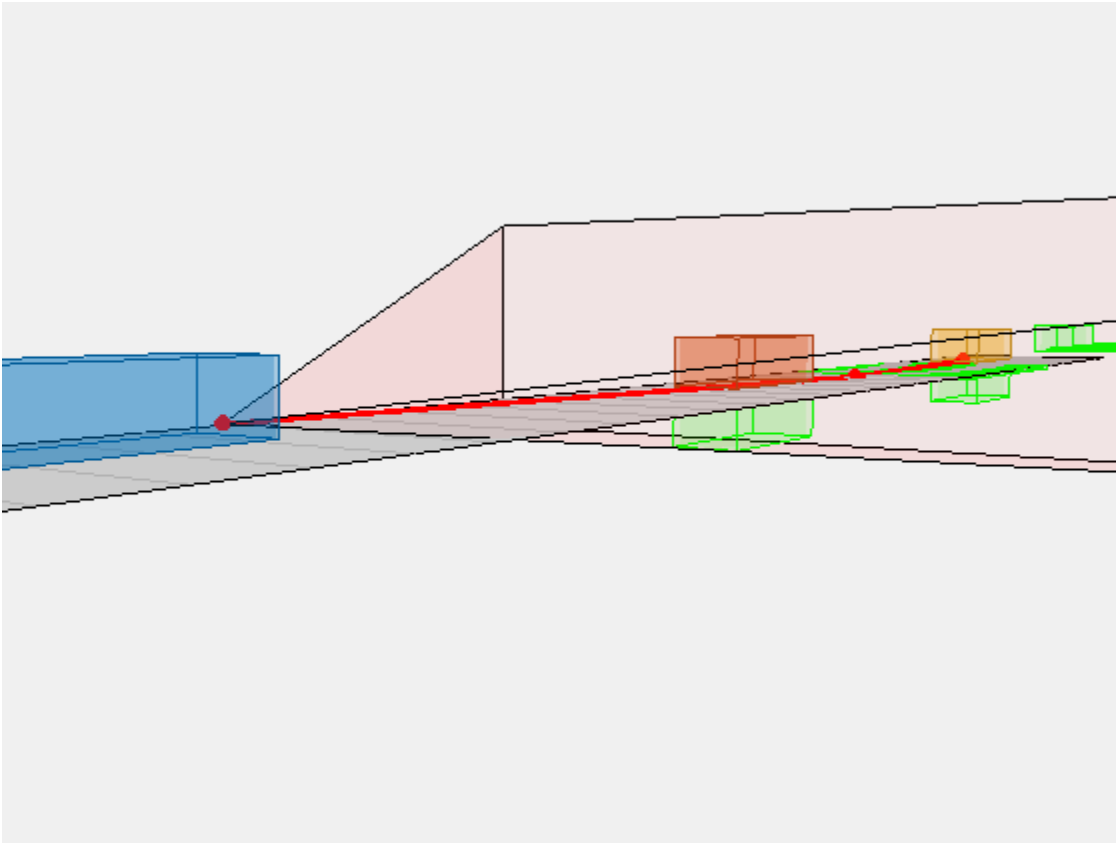
This figure shows the three-bounce path as the red line. Observe that a bounce path between the radar and the occluded yellow car exists by passing below the underside of the red car.

```
% Show bounce path arriving at the occluded yellow car
campos([55 -10 3]); camtarget([35 0 0]);
```
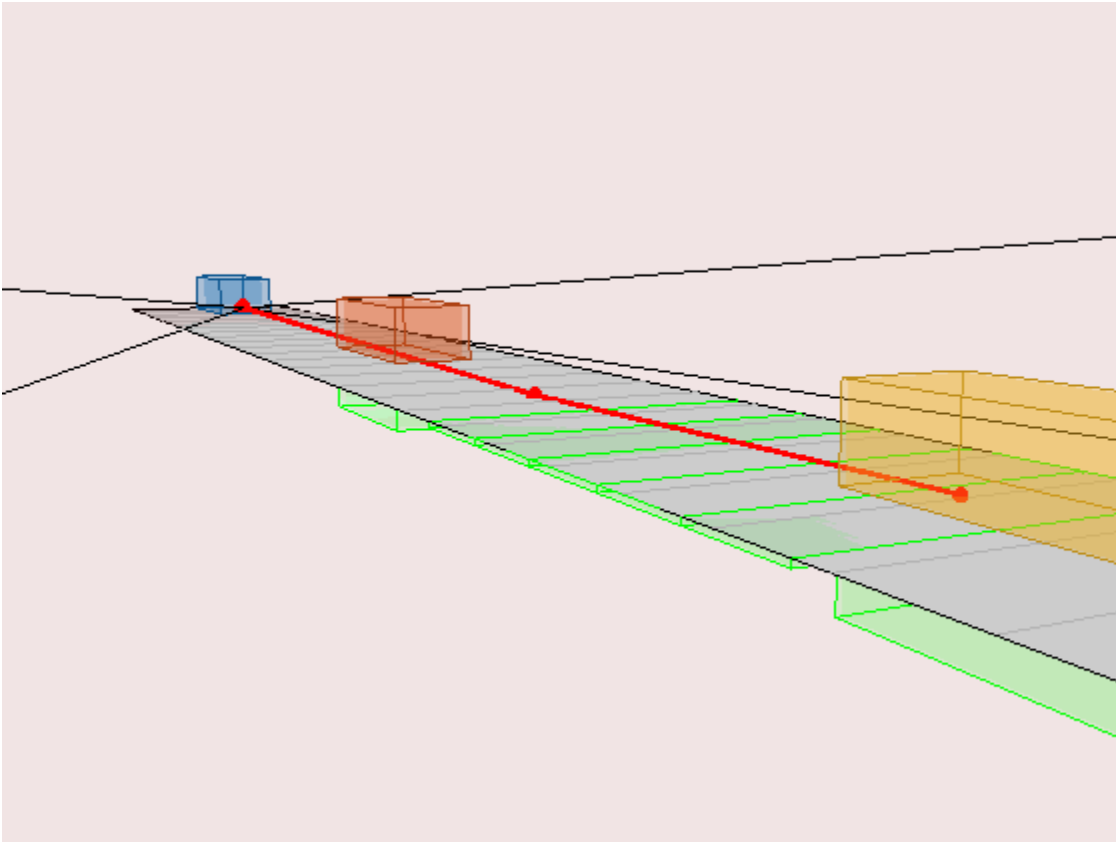
This figure shows the three-bounce path arriving at the occluded yellow car after bouncing off of the road surface.

Summary

In this example, you learned how ghost target detections arise from multiple reflections that can occur between the radar and a target. An automotive radar scenario was used to highlight a common case where ghost targets are generated by a guardrail in the field of view of the radar. As a result, there are four unique bounce paths which can produce these ghost detections. The kinematics of the ghost target detections are like the detections of true targets, and as a result, these ghost targets can create ghost tracks which can add additional processing load to a tracker and may confuse control algorithms using these tracks. The `radarTransceiver` can be used to generate higher-fidelity IQ data that is appropriate as input to detection and tracking algorithms.

```
% Restore random state
rng(rndState);
```

Supporting Functions

helperKeepDynamicObjects

```
function dynrpts = helperKeepDynamicObjects(rpts,egoVehicle)
% Filter out target reports corresponding to static objects (e.g. guardrail)
%
% This is a helper function and may be removed or modified in a future
% release.
```

```matlab
dynrpts = rpts;
if ~isempty(rpts)
    if iscell(rpts)
        vel = cell2mat(cellfun(@(d)d.Measurement(4:end),rpts(:)','UniformOutput',false));
    else
        vel = cell2mat(arrayfun(@(t)t.State(2:2:end),rpts(:)','UniformOutput',false));
    end
    vel = sign(vel(1,:)).*sqrt(sum(abs(vel(1:2,:)).^2,1));
    egoVel = sign(egoVehicle.Velocity(1))*norm(egoVehicle.Velocity(1:2));
    gndvel = vel+egoVel; % detection speed relative to ground
    isStatic = gndvel > -4 & ... greater than 4 m/s departing and,
        gndvel < 8; % less than 8 m/s closing speed
    dynrpts = rpts(~isStatic);
end
end
```

normmax

```matlab
function y = normmax(x)
if all(abs(x(:))==0)
    y = ones(size(x),'like',x);
else
    y = x(:)/max(abs(x(:)));
end
end
```

Highway Vehicle Tracking with Multipath Radar Reflections

This example shows the challenges associated with tracking vehicles on a highway in the presence of multipath radar reflections. It also shows a ghost filtering approach used with an extended object tracker to simultaneously filter ghost detections and track objects.

Introduction

Automotive radar sensors are robust against adverse environment conditions encountered during driving, such as fog, snow, rain, and strong sunlight. Automotive radar sensors have this advantage because they operate at substantially large wavelengths compared to visible-wavelength sensors, such as lidar and camera. As a side effect of using large wavelengths, surfaces around the radar sensor act like mirrors and produce undesired detections due to multipath propagation. These detections are often referred to as *ghost detections* because they seem to originate from regions where no target exists. This example shows you the impact of these multipath reflections on designing and configuring an object tracking strategy using radar detections. For more details regarding multipath phenomenon and simulation of ghost detections, see the "Simulate Radar Ghosts Due to Multipath Return" on page 1-45 example.

In this example, you simulate the multipath detections from radar sensors in an urban highway driving scenario. The highway is simulated with a barrier on both sides of the road. The scenario consists of an ego vehicle and four other vehicles driving on the highway. The ego vehicle is equipped with four radar sensors providing 360-degree coverage. This image shows the configuration of the radar sensors and detections from one scan of the sensors. The red regions represent the field of view of the radar sensors and the black dots represent the detections.
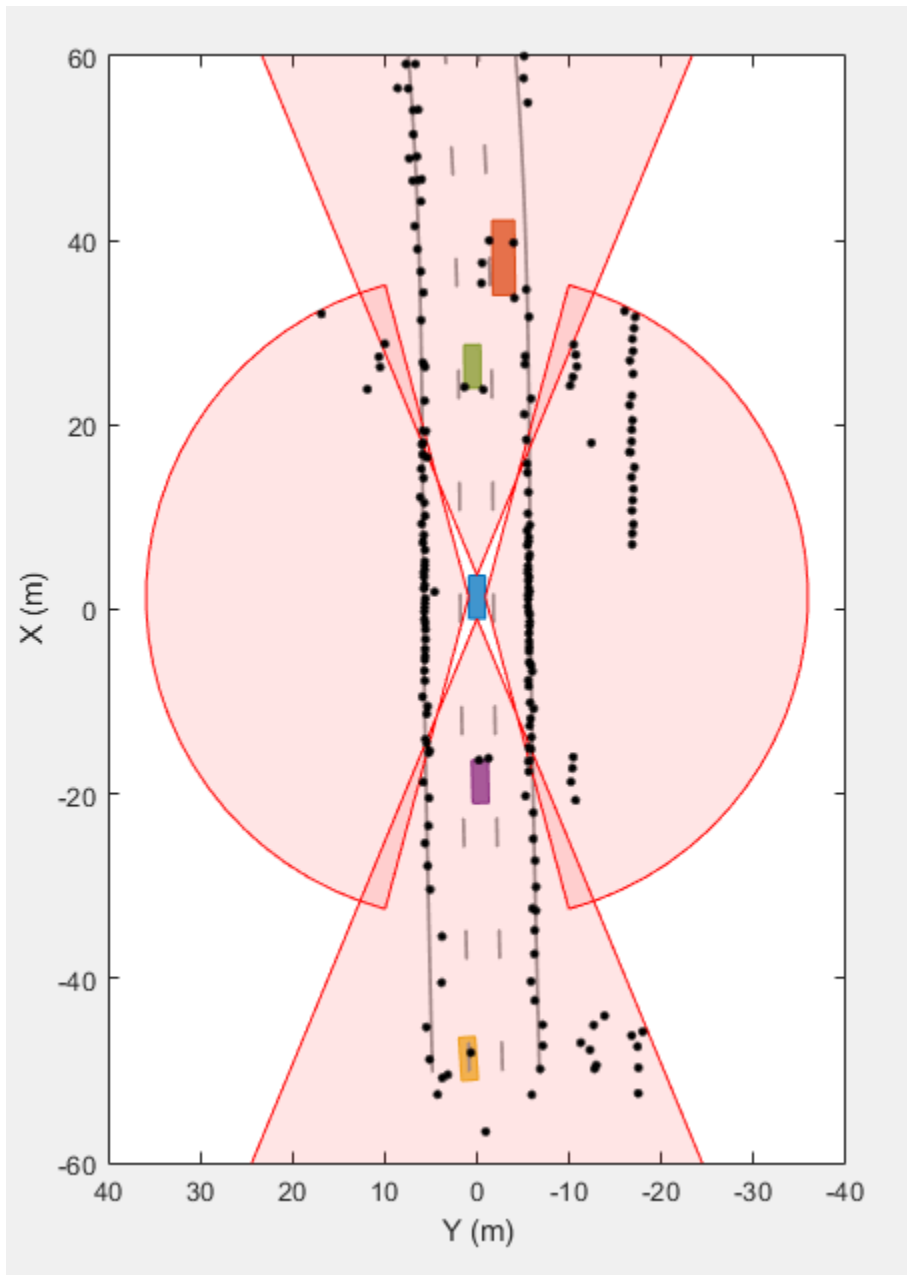
The radar sensors report detections from the vehicles and from the barriers that are on both sides of the highway. The radars also report detections that do not seem to originate from any real object in the scenario. These are ghost detections due to multipath propagation of radar signals. Object trackers assume all detections originate from real objects or uniformly distributed random clutter in the field of view. Contrary to this assumption, the ghost detections are typically more persistent than clutter and behave like detections from real targets. Due to this reason, an object tracking algorithm is very likely to generate false tracks from these detections. It is important to filter out these detections before processing the radar scan with a tracker.

Generate Sensor Data

The scenario used in this example is created using the `drivingScenario` class. You use the `radarDataGenerator` System object™ to simulate radar returns from a direct path and from reflections in the scenario. The `HasGhosts` property of the sensor is specified as `true` to simulate multipath reflections. The creation of the scenario and the sensor models is wrapped in the helper function `helperCreateMultipathDrivingScenario`, which is attached with this example. The data set obtained by sensor simulation is recorded in a MAT file that contains returns from the radar and the corresponding sensor configurations. To record the data for a different scenario or sensor configuration, you can use the following command:

```
 helperRecordData(scenario, egoVehicle, sensors, fName);
```

```
% Create the scenario
[scenario, egoVehicle, sensors] = helperCreateMultipathDrivingScenario;
```

```
% Load the recorded data
load('MultiPathRadarScenarioRecording.mat','detectionLog','configurationLog');
```

Radar Processing Chain: Radar Detections to Track List

In this section, you set up an integrated algorithm to simultaneously filter radar detections and track extended objects. The block diagram illustrates the radar processing chain used in this example.
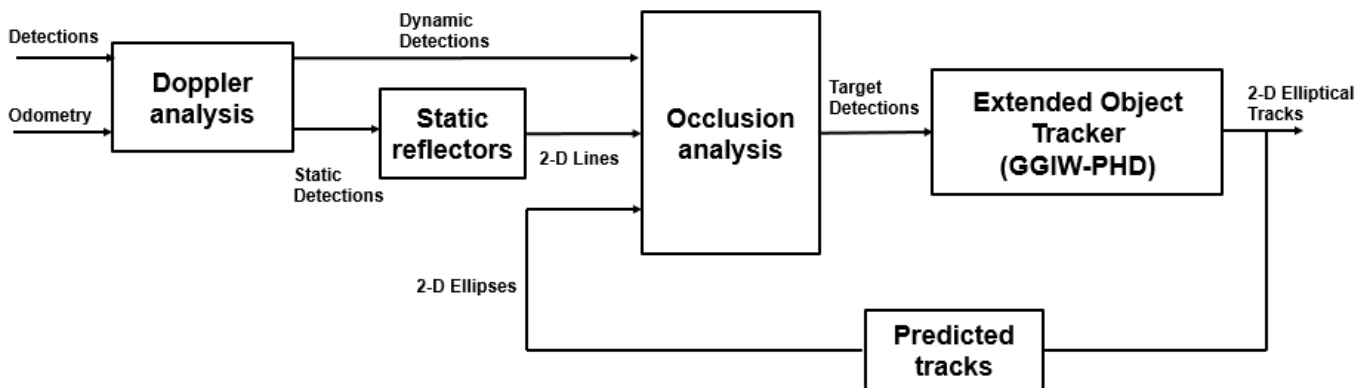
Next, you learn about each of these steps and the corresponding helper functions.

Doppler analysis

The radar sensors report the measured relative radial velocity of the reflected signals. In this step, you utilize the measured radial velocity of the detections to determine if the target is static or dynamic [1]. In the previous radar scan, a large percentage of the radar detections originate from the static environment around the ego vehicle. Therefore, classifying each detection as static or dynamic greatly helps to improve the understanding of the scene. You use the helper function `helperClassifyStaticDynamic` to classify each detection as static or dynamic.

Static reflectors

The static environment is also typically responsible for a large percentage of the ghost reflections reported by the radar sensors. After segmenting the data set and finding static detections, you process them to find 2-D line segments in the coordinate frame of the ego vehicle. First, you use the

DBSCAN algorithm to cluster the static detections into different clusters around the ego vehicle. Second, you fit a 2-D line segment on each cluster. These fitted line segments define possible reflection surfaces for signals propagating back to the radar. You use the helper function `helperFindStaticReflectors` to find these 2-D line segments from static detections.

Occlusion analysis

Reflection from surfaces produce detections from the radar sensor that seem to originate behind the reflector. After segmenting the dynamic detections from the radar, you use simple occlusion analysis to determine if the radar detection is occluded behind a possible reflector. Because signals can be reflected by static or dynamic objects, you perform occlusion analysis in two steps. First, dynamic detections are checked against occlusion with the 2-D line segments representing the static reflectors. You use the helper function `helperClassifyGhostsUsingReflectors` to classify if the detection is occluded by a static reflector. Second, the algorithm uses information about predicted tracks from an extended tracking algorithm to check for occlusion against dynamic objects in the scene. The algorithm uses only confirmed tracks from the tracker to prevent overfiltering of the radar detections in the presence of tentative or false tracks. You use the helper function `helperClassifyGhostsUsingTracks` to classify if the detection is occluded by a dynamic object.

This entire algorithm for processing radar detections and classifying them is then wrapped into a larger helper function, `helperClassifyRadarDetections`, which classifies and segments the detection list into four main categories:

1 Target detections – These detections are classified to originate from real dynamic targets in the scene.

2 Environment detections – These detections are classified to originate from the static environment.

3 Ghost (Static) – These detections are classified to originate from dynamic targets but reflected via the static environment.

4 Ghost (Dynamic) – These detections are classified to originate from dynamic targets but reflected via other dynamic objects.

Setup GGIW-PHD Extended Object Tracker

The target detections are processed with an extended object tracker. In this example, you use a gamma Gaussian inverse Wishart probability hypothesis density (GGIW-PHD) extended object tracker. The GGIW-PHD tracker models the target with an elliptical shape and the measurement model assumes that the detections are uniformly distributed inside the extent of the target. This model allows a target to accept two-bounce ghost detections, which have a higher probability of being misclassified as a real target. These two-bounce ghost detections also report a Doppler measurement that is inconsistent with the actual motion of the target. When these ghost detections and real target detections from the same object are estimated to belong to the same partition of detections, the incorrect Doppler information can potentially cause the track estimate to diverge.

To reduce this problem, the tracker processes the range-rate measurements with higher measurement noise variance to account for this imperfection in the target measurement model. The tracker also uses a combination of a high assignment threshold and low merging threshold. A high assignment threshold allows the tracker to reduce the generation of new components from ghost targets detections, which are misclassified as target detections. A low merging threshold enables the tracker to discard corrected components (hypothesis) of a track, which might have diverged due to correction with ghost detections.

You set up the tracker using the `trackerPHD` System object™. For more details about extended object trackers, refer to the "Extended Object Tracking of Highway Vehicles with Radar and Camera" (Automated Driving Toolbox) example.

```
% Configuration of the sensors from the recording to set up the tracker
[~, sensorConfigurations] = helperAssembleData(detectionLog{1},configurationLog{1});

% Configure the tracker to use the GGIW-PHD filter with constant turn-rate motion model
for i = 1:numel(sensorConfigurations)
    sensorConfigurations{i}.FilterInitializationFcn = @helperInitGGIWFilter;
    sensorConfigurations{i}.SensorTransformFcn = @ctmeas;
end

% Create the tracker using trackerPHD with Name-value pairs
tracker = trackerPHD(SensorConfigurations = sensorConfigurations,...
    PartitioningFcn = @(x)helperMultipathExamplePartitionFcn(x,2,5),...
    AssignmentThreshold = 450,...
    ExtractionThreshold = 0.8,...
    ConfirmationThreshold = 0.85,...
    MergingThreshold = 25,...
    DeletionThreshold = 1e-3,...
    BirthRate = 1e-4,...
    HasSensorConfigurationsInput = true...
    );
```

Run Scenario and Track Objects

Next, you advance the scenario, use the recorded measurements from the sensors, and process them using the previously described algorithm. You analyze the performance of the tracking algorithm by using the generalized optimal subpattern assignment (GOSPA) metric. You also analyze the performance of the classification filtering algorithm by estimating a confusion matrix between true and estimated classification of radar detections. You obtain the true classification information about the detections using the `helperTrueClassificationInfo` helper function.

```
% Create trackGOSPAMetric object to calculate GOSPA metric
gospaMetric = trackGOSPAMetric(Distance = 'custom', ...
    DistanceFcn = @helperGOSPADistance, ...
    CutoffDistance = 35);

% Create display for visualization of results
display = helperMultiPathTrackingDisplay;

% Predicted track list for ghost filtering
predictedTracks = objectTrack.empty(0,1);

% Confusion matrix
confMat = zeros(5,5,numel(detectionLog));

% GOSPA metric
gospa = zeros(4,numel(detectionLog));

% Ground truth
groundTruth = scenario.Actors(2:end);

for i = 1:numel(detectionLog)
    % Advance scene for visualization of ground truth
    advance(scenario);
```

```
% Current time
time = scenario.SimulationTime;

% Detections and sensor configurations
[detections, configurations] = helperAssembleData(detectionLog{i},configurationLog{i});

% Predict confirmed tracks to current time for classifying ghosts
if isLocked(tracker)
    predictedTracks = predictTracksToTime(tracker,'confirmed',time);
end

% Classify radar detections as targets, ghosts, or static environment
[targets, ghostStatic, ghostDynamic, static, reflectors, classificationInfo] = helperClassify

% Pass detections from target and sensor configurations to the tracker
confirmedTracks = tracker(targets, configurations, time);

% Visualize the results
display(egoVehicle, sensors, targets, confirmedTracks, ghostStatic, ghostDynamic, static, re

% Calculate GOSPA metric
[gospa(1, i),~,~,gospa(2,i),gospa(3,i),gospa(4,i)] = gospaMetric(confirmedTracks, groundTrut

% Get true classification information and generate confusion matrix
trueClassificationInfo = helperTrueClassificationInfo(detections);
confMat(:,:,i) = helperConfusionMatrix(trueClassificationInfo, classificationInfo);
end
```

Results

Animation and snapshot analysis

The animation that follows shows the result of the radar data processing chain. The black ellipses around vehicles represent estimated tracks. The radar detections are visualized with four different colors depending on their predicted classification from the algorithm. The black dots in the visualization represent static radar target detections. Notice that these detections are overlapped by black lines, which represent the static reflectors found using the DBSCAN algorithm. The maroon markers represent the detections processed by the extended object tracker, while the green and blue markers represent radar detections classified as reflections via static and dynamic objects, respectively. Notice that the tracker is able to maintain a track on all four vehicles during the scenario.
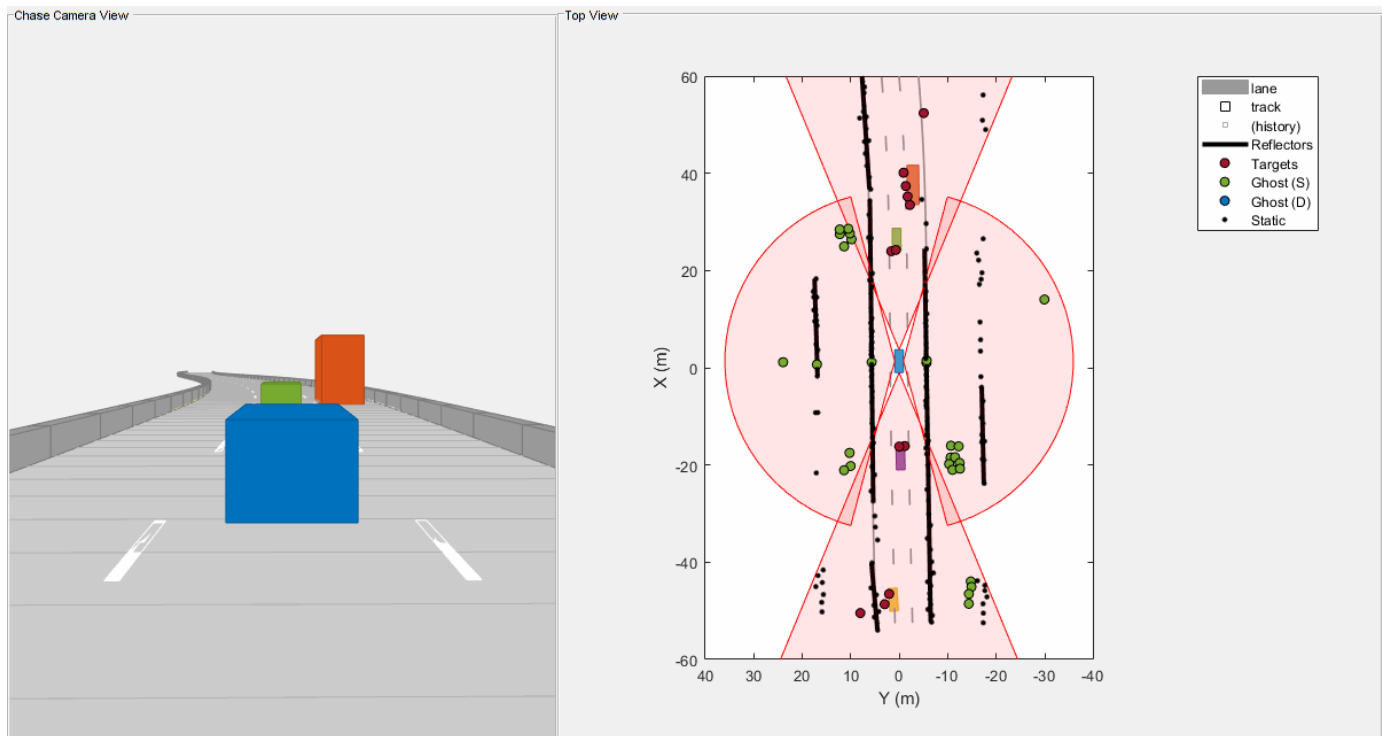
Next, you analyze the performance of the algorithm using different snapshots captured during the simulation. The snapshot below is captured at 3 seconds and shows the situation in front of the ego vehicle. At this time, the ego vehicle is approaching the slow-moving truck, and the left radar sensor observes reflections of these objects via the left barrier. These detections appear as mirrored detections of these objects in the barrier. Notice that the black line estimated as a 2-D reflector is in the line of sight of these detections. Therefore, the algorithm is able to correctly classify these detections as ghost targets reflected off static objects.

```
f = showSnaps(display,1:2,1);
if ~isempty(f)
    ax = findall(f,'Type','Axes','Tag','birdsEyePlotAxes');
    ax.XLim = [-10 30];
    ax.YLim = [-10 20];
end
```
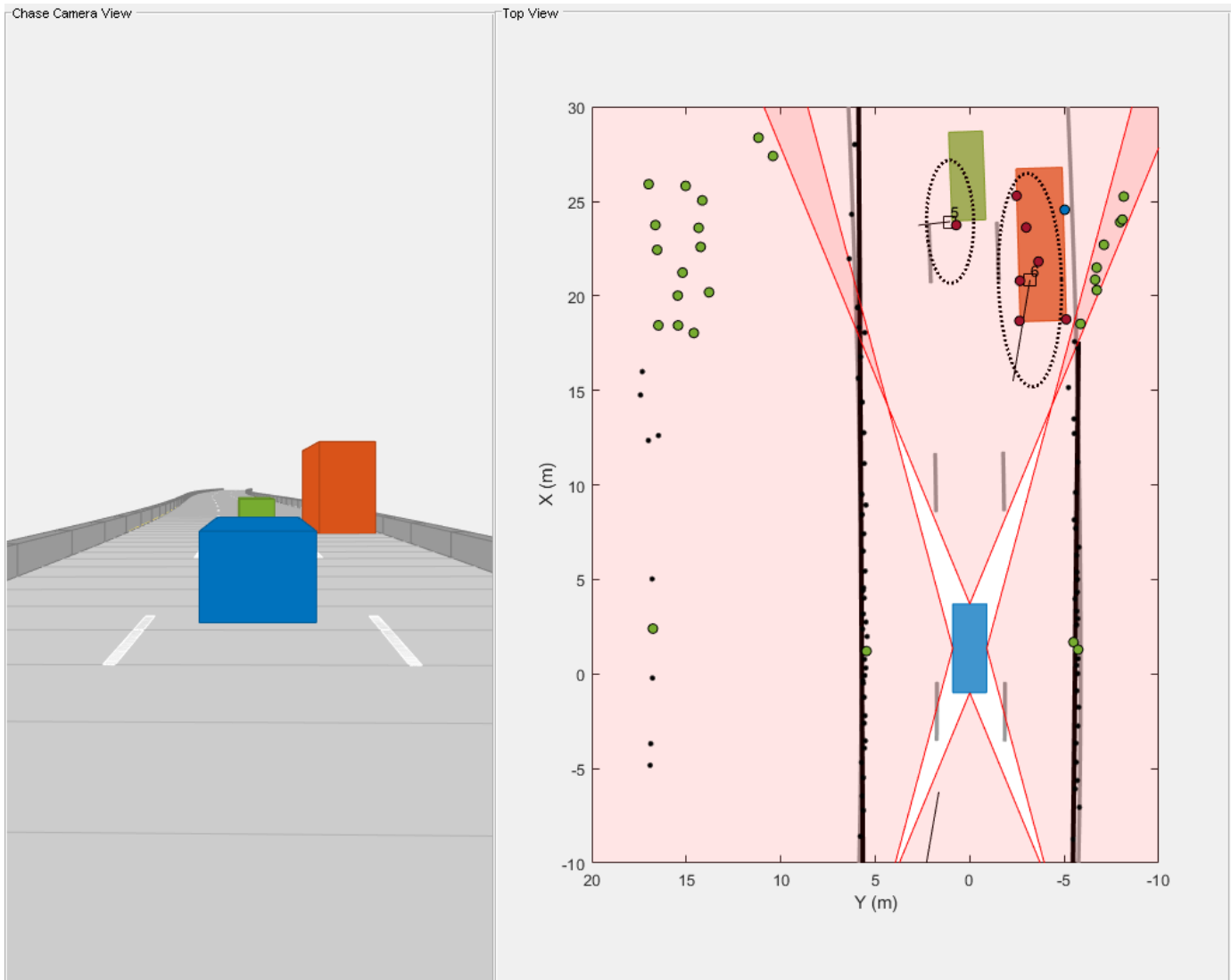
Next, analyze the performance of the algorithm using the snapshot captured at 4.3 seconds. At this time, the ego vehicle is even closer to the truck and the truck is approximately halfway between the green vehicle and the ego vehicle. During these situations, the left side of the truck acts as a strong reflector and generates ghost detections. The detections on the right half of the green vehicle are from two-bounce detections off of the green vehicle as the signal travels back to the sensor after reflecting off the truck. The algorithm is able to classify these detections as ghost detections generated from dynamic object reflections because the estimated extent of the truck is in the direct line of sight of these detections.

```
f = showSnaps(display,1:2,2);
if ~isempty(f)
    ax = findall(f,'Type','Axes','Tag','birdsEyePlotAxes');
    ax.XLim = [-10 30];
    ax.YLim = [-10 20];
end
```
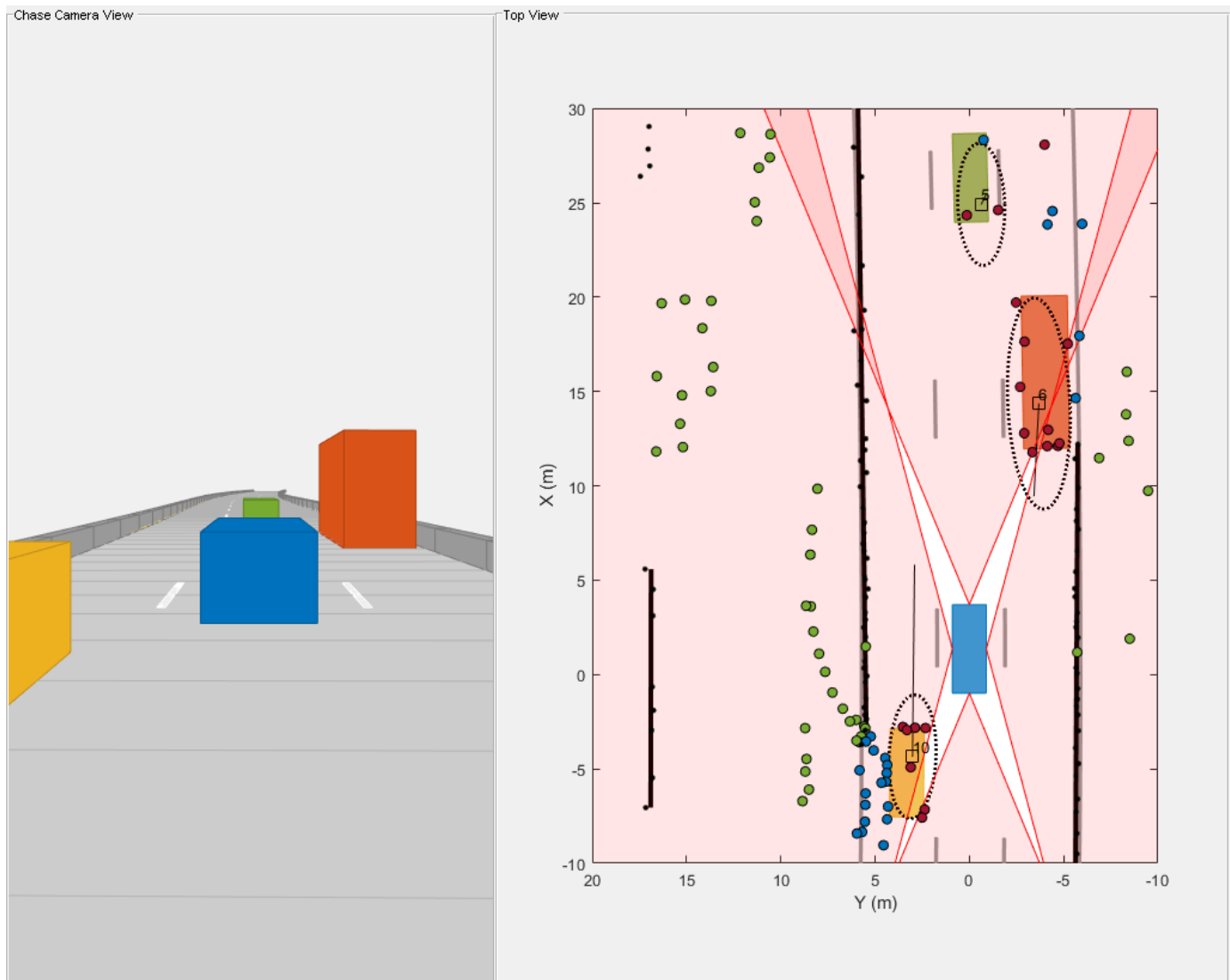
Also notice the passing vehicle denoted by the yellow car on the left of the ego vehicle. The detections, which seem to originate from the nonvisible surface of the yellow vehicle, are two-bounce detections of the barriers, reflected via the front face of the passing vehicle. These ghost detections are misclassified as target detections because they seem to originate from inside the estimated extent of the vehicle. At the same location, the detections that lie beyond the barrier are also two-bounce detections of the front face when the signal is reflected from the barrier and returns to the sensor. Since these detections lie beyond the extent of the track and the track is in the direct line of sight, they are classified as ghost detections from reflections off dynamic objects.

Performance analysis

Quantitatively assess the performance of the tracking algorithm by using the GOSPA metric and its associated components. A lower value of the metric denotes better tracking performance. In the figure below, the *Missed-target* component of the metric remains zero after a few steps in the beginning, representing establishment delay of the tracker as well as an occluded target. The zero value of the component shows that no targets were missed by the tracker. The *False-tracks* component of the metric increased around for 1 second around 85th time step. This denotes a false

track confirmed by the tracker for a short duration from ghost detections incorrectly classified as a real target.

```
figure;
plot(gospa','LineWidth',2);
legend('GOSPA','Localization GOSPA','Missed-target GOSPA','False-tracks GOSPA');
```
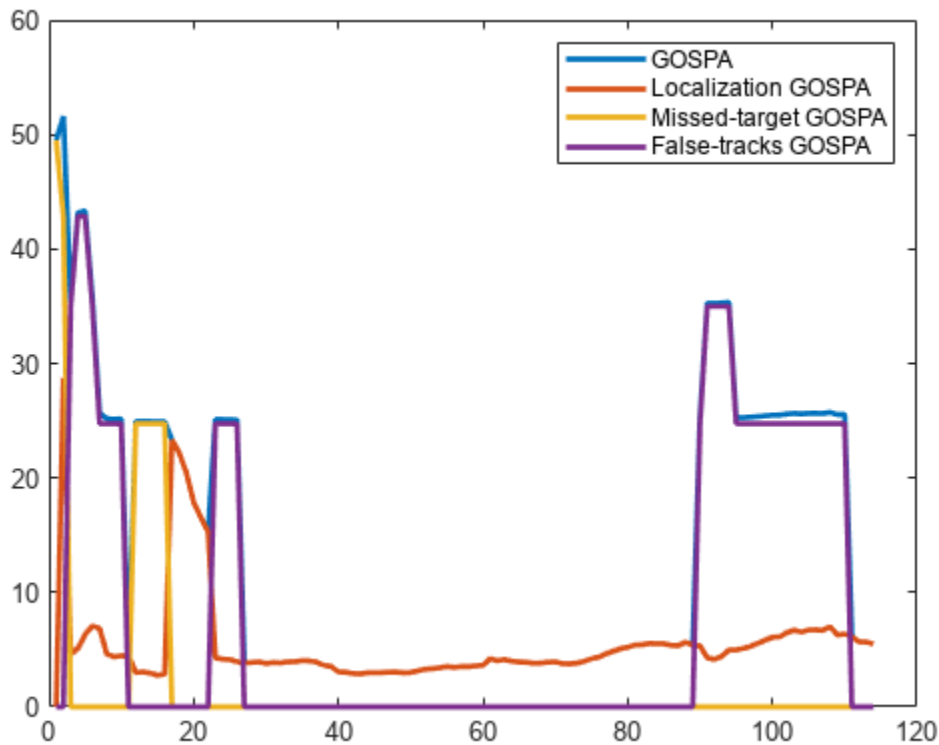

Similar to the tracking algorithm, you also quantitatively analyze the performance of the radar detection classification algorithm by using a confusion matrix [2]. The rows shown in the table denote the true classification information of the radar detections and the columns represent the predicted classification information. For example, the second element of the first row defines the percentage of target detections predicted as ghosts from static object reflections.

More than 90% of the target detections are classified correctly. However, a small percentage of the target detections are misclassified as ghosts from dynamic reflections. Also, approximately 3% of ghosts from static object reflections and 20% of ghosts from dynamic object reflections are misclassified as targets and sent to the tracker for processing. A common situation when this occurs in this example is when the detections from two-bounce reflections lie inside the estimated extent of the vehicle. Further, the classification algorithm used in this example is not designed to find false alarms or clutter in the scene. Therefore, the fifth column of the confusion matrix is zero. Due to spatial distribution of the false alarms inside the field of view, the majority of false alarm detections are either classified as reflections from static objects or dynamic objects.

```
% Accumulate confusion matrix over all steps
confusionMatrix = sum(confMat,3);
numElements = sum(confusionMatrix,2);
```

```
numElemsTable = array2table(numElements,'VariableNames',{'Number of Detections'},'RowNames',{'Tai
disp('True Information');disp(numElemsTable);
```

```
True Information
              Number of Detections
              _____

    Targets             1969
    Ghost (S)           3155
    Ghost (D)            849
    Environment        27083
    Clutter              138
```

```
% Calculate percentages
percentMatrix = confusionMatrix./numElements*100;
```

```
percentMatrixTable = array2table(round(percentMatrix,2),'RowNames',{'Targets','Ghost (S)','Ghost
    "VariableNames",{'Targets','Ghost (S)','Ghost (D)', 'Environment','Clutter'});
```

```
disp('True vs Predicted Confusion Matrix (%)');disp(percentMatrixTable);
```

```
True vs Predicted Confusion Matrix (%)
                  Targets     Ghost (S)     Ghost (D)     Environment     Clutter

                  _____     _____     _____     _____     _____

    Targets        90.76        0.56          8.28           0.41            0
    Ghost (S)        3.2        83.77         12.68           0.35            0
    Ghost (D)      18.26         0.24         81.51              0            0
    Environment     1.05         2.71          4.06          92.17            0
    Clutter        21.74        62.32         15.22           0.72            0
```

Summary

In this example, you simulated radar detections due to multipath propagation in an urban highway driving scenario. You configured a data processing algorithm to simultaneously filter ghost detections and track vehicles on the highway. You also analyzed the performance of the tracking algorithm and the classification algorithm using the GOSPA metric and confusion matrix.

References

[1] Prophet, Robert, et al. "Instantaneous Ghost Detection Identification in Automotive Scenarios." *2019 IEEE Radar Conference (RadarConf)*. IEEE, 2019.

[2] Kraus, Florian, et al. "Using machine learning to detect ghost images in automotive radar." *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2020.

Track-to-Track Fusion for Automotive Safety Applications

This example shows how to fuse tracks from two vehicles to provide a more comprehensive estimate of the environment than can be seen by each vehicle. The example demonstrates the use of a track-level fuser and the object track data format. In this example, you use the driving scenario and vision detection generator from Automated Driving Toolbox™, the radar data generator from the Radar Toolbox™, and the tracking and track fusion models from Sensor Fusion and Tracking Toolbox™.

Motivation

Automotive safety applications rely on the fusion of data from different sensor systems mounted on the vehicle. Individual vehicles fuse sensor detections either by using a centralized tracker or by taking a more decentralized approach and fusing tracks produced by individual sensors. In addition to intravehicle data fusion, the fusion of data from multiple vehicles provides added benefits, which include better coverage, situational awareness, and safety [1] on page 1-85. This intervehicle sensor fusion approach takes advantage of the variety of sensors and provides better coverage to each vehicle, because it uses data updated by sensors on other vehicles in the area. Governments and vehicle manufacturers have long recognized the need to share information between vehicles in order to increase automotive safety. For example, V2X protocols and cellular communication links are being developed.

While sensor fusion across multiple vehicles is beneficial, most vehicles are required to meet certain safety requirements even if only internal sensors are available. Therefore, the vehicle is likely to be equipped with a tracker, a track fuser, or both. These tracking algorithms provide situational awareness at the single vehicle level. As a result, the assumption made in this example is that vehicles share situational awareness by broadcasting tracks and performing track-to-track fusion.

This example demonstrates the benefit of fusing tracks from two vehicles to enhance situational awareness and safety. This example does not simulate the communications systems. Instead, the example assumes that a communications system provides the bandwidth required to transmit tracks between the two vehicles.

Track-to-Track Architecture

The following block diagram depicts the main functions in the two vehicles, where:

- Vehicle 1 has two sensors, each providing detections to a local vehicle tracker. The tracker uses the detections from the local sensors to track objects and outputs these local tracks to the vehicle track fuser.
- Vehicle 2 has a single sensor, which is a tracking radar. The tracking radar outputs tracks and serves as the local tracker for vehicle 2. The tracks from the tracking radar are inputs to the vehicle track fuser on vehicle 2.
- The track fuser on each vehicle fuses the local vehicle tracks with the tracks received from the other vehicle's track fuser. After each update, the track fuser on each vehicle broadcasts its fused tracks, which feed into the next update of the track fuser on the other vehicle.
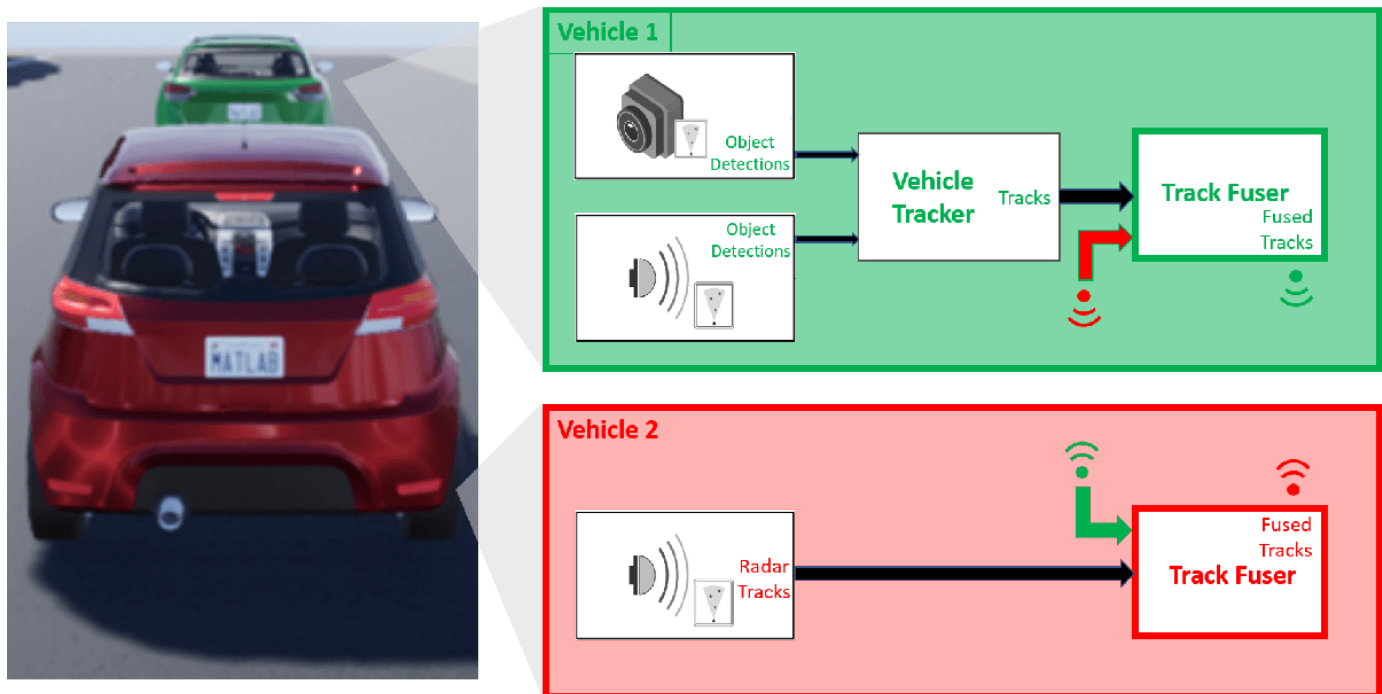
In this example, you use a `trackerJPDA` (Sensor Fusion and Tracking Toolbox) object to define the vehicle 1 tracker.

```
% Create the tracker for vehicle 1
v1Tracker = trackerJPDA('TrackerIndex',1, 'DeletionThreshold', [4 4], 'AssignmentThreshold', [100
posSelector = [1 0 0 0 0 0; 0 0 1 0 0 0];
```

In this architecture, the fused tracks from one vehicle update the fused tracks on the other vehicle. These fused tracks are then broadcast back to the first vehicle. To avoid rumor propagation, be careful how tracks from another vehicle update the track fuser.

Consider the following rumor propagation example: at some update step, vehicle 1 tracks an object using its internal sensors. Vehicle 1 then fuses the object track and transmits it to vehicle 2, which now fuses the track with its own tracks and becomes aware of the object. Up to this point, this is exactly the goal of track-to-track fusion: to enhance the situational awareness of vehicle 2 with information from vehicle 1. Since vehicle 2 now knows about the object, it starts broadcasting the track as well, perhaps for the benefit of another vehicle (not shown in the example).

However, vehicle 1 now receives track information from vehicle 2 about the object that only vehicle 1 actually tracks. So, the track fuser on vehicle 1 must be aware that the tracks of this object it gets from vehicle 2 do not actually contain any new information updated by an independent source. To make the distinction between tracks that contain new information and tracks that just repeat information, you must define vehicle 2 as an *external source* to the track fuser on vehicle 1. Similarly, vehicle 1 must be defined as an external source to the track fuser on vehicle 2. Furthermore, you need to define only tracks that are updated by a track fuser based on information from an internal source as *self-reported*. By doing so, the track fuser in each vehicle ignores updates from tracks that bounce back and forth between the track fusers without any new information in them.

The local tracker of each vehicle tracks objects relative to the vehicle reference frame, called the ego frame. The track-to-track fusion is done at the scenario frame, which is the global-level frame. The helper `egoToScenario` function transforms tracks from the ego frame to the scenario frame.

Similarly, the function `scenarioToEgo` transforms tracks from the scenario frame to any of the ego frames. Both transformations rely on the `StateParameters` property of the `objectTrack` (Sensor Fusion and Tracking Toolbox) objects. When the `trackFuser` object calculates the distance of a central track in the scenario frame to a local track in any frame, it uses the `StateParameters` of the local track to perform the coordinate transformation.

To achieve the previously described `trackFuser` definitions, define the following sources as a `fuserSourceConfiguration` (Sensor Fusion and Tracking Toolbox) object.

```
% Define sources for each vehicle
v1TrackerConfiguration = fuserSourceConfiguration('SourceIndex',1,'IsInternalSource',true, ...
    "CentralToLocalTransformFcn", @scenarioToEgo, 'LocalToCentralTransformFcn', @egoToScenario);
v2FuserConfiguration = fuserSourceConfiguration('SourceIndex',4,'IsInternalSource',false);
v1Sources = {v1TrackerConfiguration; v2FuserConfiguration};
v2TrackerConfiguration = fuserSourceConfiguration('SourceIndex',2,'IsInternalSource',true, ...
    "CentralToLocalTransformFcn", @scenarioToEgo, 'LocalToCentralTransformFcn', @egoToScenario);
v1FuserConfiguration = fuserSourceConfiguration('SourceIndex',3,'IsInternalSource',false);
v2Sources = {v2TrackerConfiguration; v1FuserConfiguration};
```

You can now define each vehicle track fuser as a `trackFuser` (Sensor Fusion and Tracking Toolbox) object.

```
stateParams = struct('Frame','Rectangular','Position',[0 0 0],'Velocity',[0 0 0]);
v1Fuser = trackFuser('FuserIndex',3,...
    'AssignmentThreshold', [100 inf], ...
    'MaxNumSources',2,'SourceConfigurations',v1Sources,...
    'StateFusion','Intersection','DeletionThreshold',[3 3],...
    'StateParameters',stateParams);
v2Fuser = trackFuser('FuserIndex',4,...
    'AssignmentThreshold', [100 inf], ...
    'MaxNumSources',2,'SourceConfigurations',v2Sources,'StateFusion',...
    'Intersection','DeletionThreshold',[3 3],...
    'StateParameters',stateParams);
```

```
% Initialize the following variables
fusedTracks1 = objectTrack.empty(0,1);
fusedTracks2 = objectTrack.empty(0,1);
wasFuser1Updated = false;
wasFuser2Updated = false;
```

Define Scenario

The following scenario shows two vehicles driving down a street. Vehicle 1 is the lead vehicle and is equipped with two forward-looking sensors: a short-range radar sensor and a vision sensor. Vehicle 2, driving 10 meters behind vehicle 1, is equipped with a long-range radar. The right side of the street contains parked vehicles. A pedestrian stands between the vehicles. This pedestrian is shown as a dot at about X = 60 meters.

Due to the short distance between vehicle 2 and vehicle 1, most of the vehicle 2 radar sensor coverage is occluded by vehicle 1. As a result, most of the tracks that the track fuser on vehicle 2 maintains are first initialized by tracks broadcast from vehicle 1.

```
% Create the drivingScenario object and the two vehicles
[scenario, vehicle1, vehicle2] = createDrivingScenario;
```

```
% Create all the sensors
[sensors, numSensors, attachedVehicle] = createSensors(scenario);
```

```
% Create display
[f,plotters] = createV2VDisplay(scenario, sensors, attachedVehicle);
```

The following chase plot is seen from the point of view of the second vehicle. An arrow indicates the position of the pedestrian that is almost entirely occluded by the parked vehicles and the first vehicle.
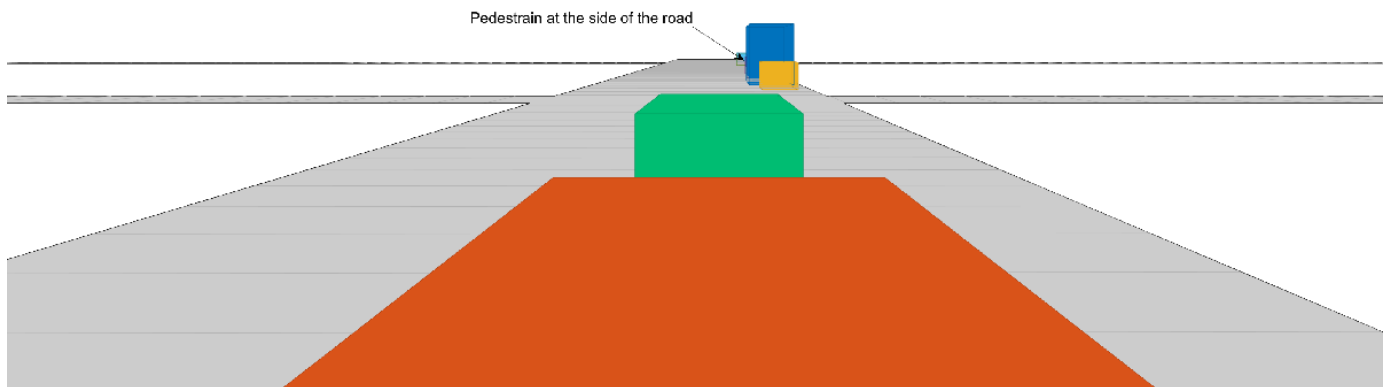

```
% Define each vehicle as a combination of an actor, sensors, a tracker, and plotters
v1=struct('Actor',{vehicle1},'Sensors',{sensors(attachedVehicle==1)},'Tracker',{v1Tracker},'DetP
v2=struct('Actor',{vehicle2},'Sensors',{sensors(attachedVehicle==2)},'Tracker',{{}},'DetPlotter'
```

Run Simulation

The following code runs the simulation.

```
running = true;

% For repeatable results, set the random number seed
s = rng;
rng(2019)
snaptimes = [0.5, 2.8, 4.4, 6.3, inf];
snaps = cell(numel(snaptimes,1));
i = 1;
f.Visible = 'on';
while running && ishghandle(f)
    time  = scenario.SimulationTime;

    % Detect and track at the vehicle level
    [tracks1,wasTracker1Updated] = detectAndTrack(v1,time,posSelector);
    [tracks2,wasTracker2Updated] = detectAndTrack(v2,time,posSelector);

    % Keep the tracks from the previous fuser update
    oldFusedTracks1 = fusedTracks1;
    oldFusedTracks2 = fusedTracks2;
```

```matlab
    % Update the fusers
    if wasTracker1Updated || wasFuser2Updated
        tracksToFuse1 = [tracks1;oldFusedTracks2];
        if isLocked(v1Fuser) || ~isempty(tracksToFuse1)
            [fusedTracks1,~,~,info1] = v1Fuser(tracksToFuse1,time);
            wasFuser1Updated = true;
            pos = getTrackPositions(fusedTracks1,posSelector);
            ids = string([fusedTracks1.TrackID]');
            plotTrack(plotters.veh1FusePlotter,pos,ids);
        else
            wasFuser1Updated = false;
            fusedTracks1 = objectTrack.empty(0,1);
        end
    else
        wasFuser1Updated = false;
        fusedTracks1 = objectTrack.empty(0,1);
    end
    if wasTracker2Updated || wasFuser1Updated
        tracksToFuse2 = [tracks2;oldFusedTracks1];
        if isLocked(v2Fuser) || ~isempty(tracksToFuse2)
            [fusedTracks2,~,~,info2] = v2Fuser(tracksToFuse2,time);
            wasFuser2Updated = true;
            pos = getTrackPositions(fusedTracks2,posSelector);
            ids = string([fusedTracks2.TrackID]');
            plotTrack(plotters.veh2FusePlotter,pos,ids);
        else
            wasFuser2Updated = false;
            fusedTracks2 = objectTrack.empty(0,1);
        end
    else
        wasFuser2Updated = false;
        fusedTracks2 = objectTrack.empty(0,1);
    end

    % Update the display
    updateV2VDisplay(plotters, scenario, sensors, attachedVehicle)

    % Advance the scenario one time step and exit the loop if the scenario is complete
    running = advance(scenario);

    % Capture an image of the frame at specified times
    if time >= snaptimes(i)
        snaps{i} = takesnap(f);
        i = i + 1;
    end
end
```
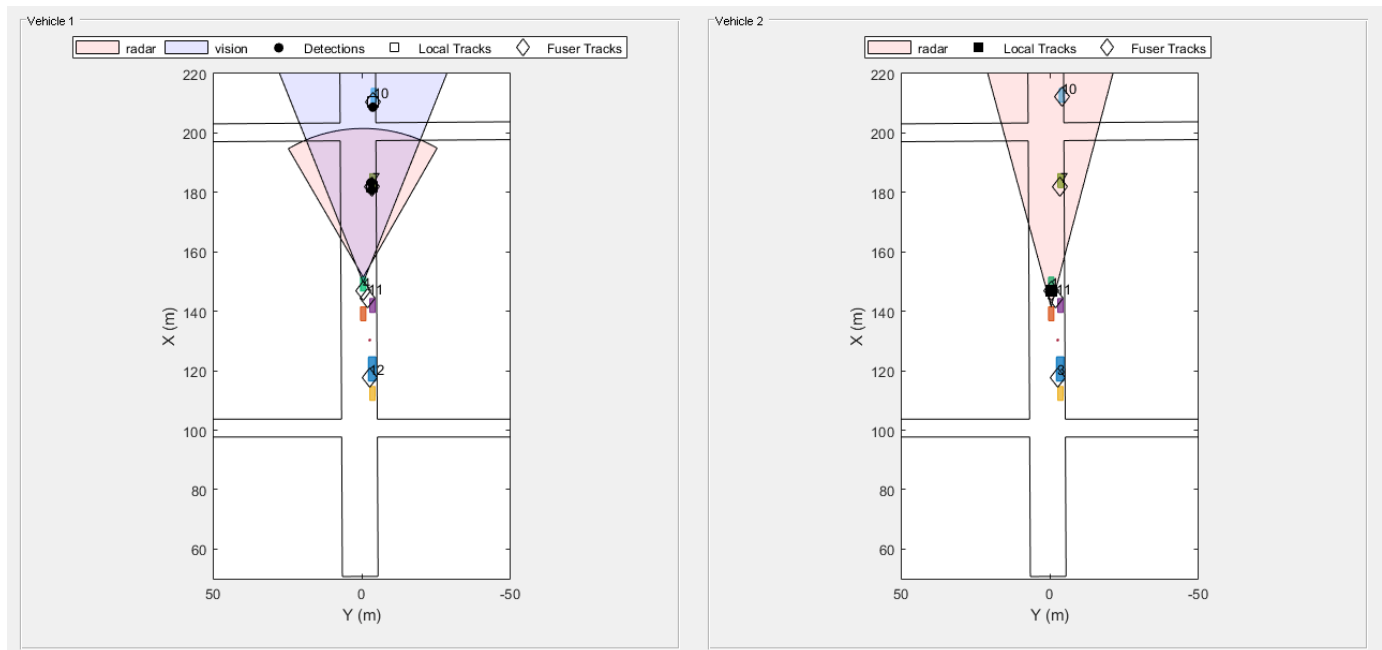
The figure shows the scene and tracking results at the end of the scenario. Subsequent sections of this example analyze the tracking results at key times.

Analyze Tracking at Beginning of Simulation

When the simulation begins, vehicle 1 detects the vehicles parked on the right side of the street. Then, vehicle 1 tracker confirms the tracks associated with the parked vehicles. At this time, the only object detected and tracked by vehicle 2 tracker is vehicle 1, which is immediately in front of it. Once the vehicle 1 track fuser confirms the tracks, it broadcasts them, and the vehicle 2 track fuser fuses them. As a result, vehicle 2 becomes aware of the parked vehicles before it can detect them on its own.

```
showsnap(snaps,1)
```
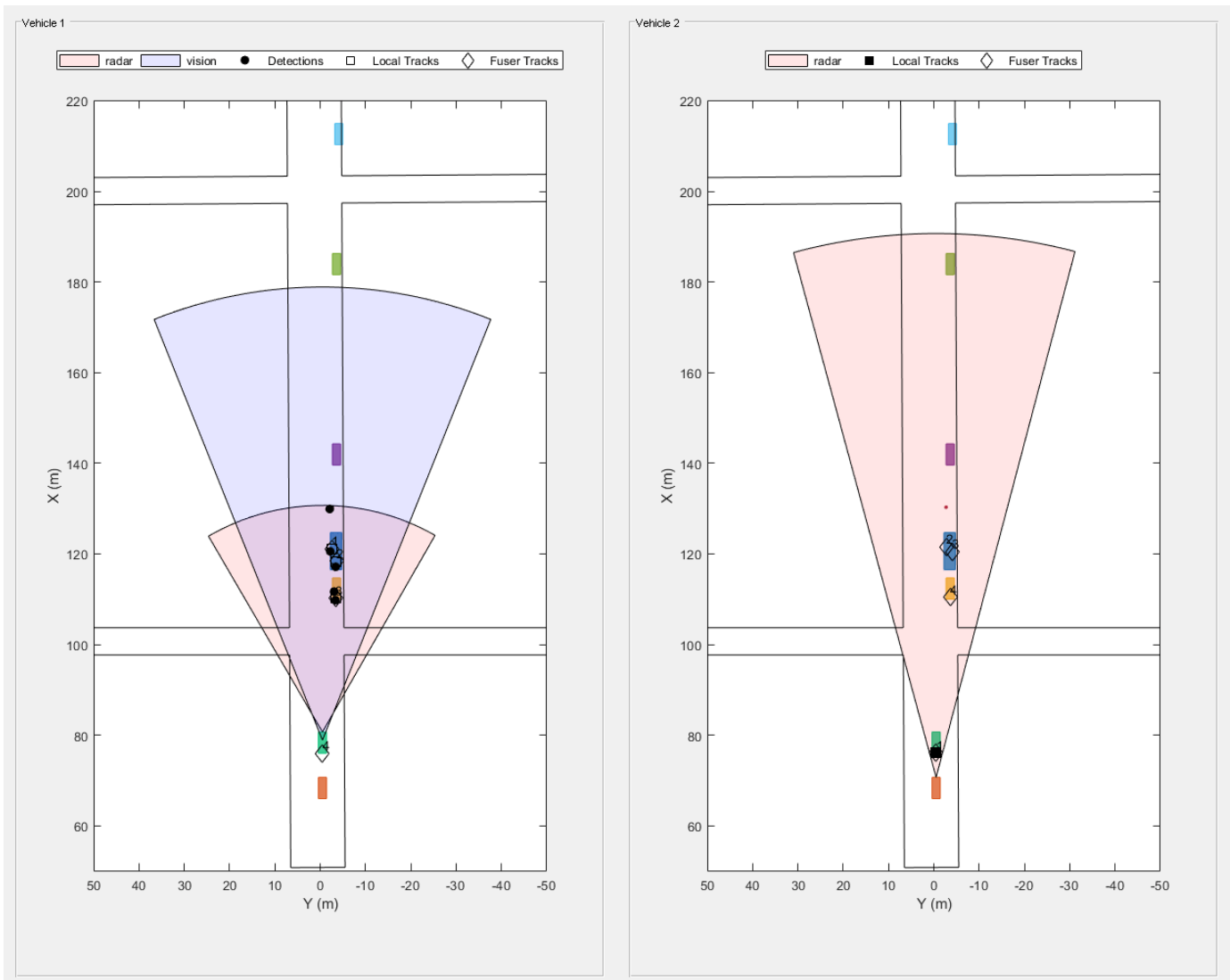
Analyze Tracking of Pedestrian at Side of Street

As the simulation continues, vehicle 2 is able to detect and track the vehicles parked at the side as well and fuses them with the tracks coming from vehicle 1. Vehicle 2 is able to detect and track the pedestrian about 4 seconds into the simulation, and vehicle 2 fuses the track associated with the pedestrian around 4.4 seconds into the simulation (see snapshot 2). However, it takes vehicle 2 about two seconds before it can detect and track the pedestrian by its own sensors (see snapshot 3). Detecting a pedestrian in the street two seconds earlier can markedly improve safety.

```
showsnap(snaps,2)
```
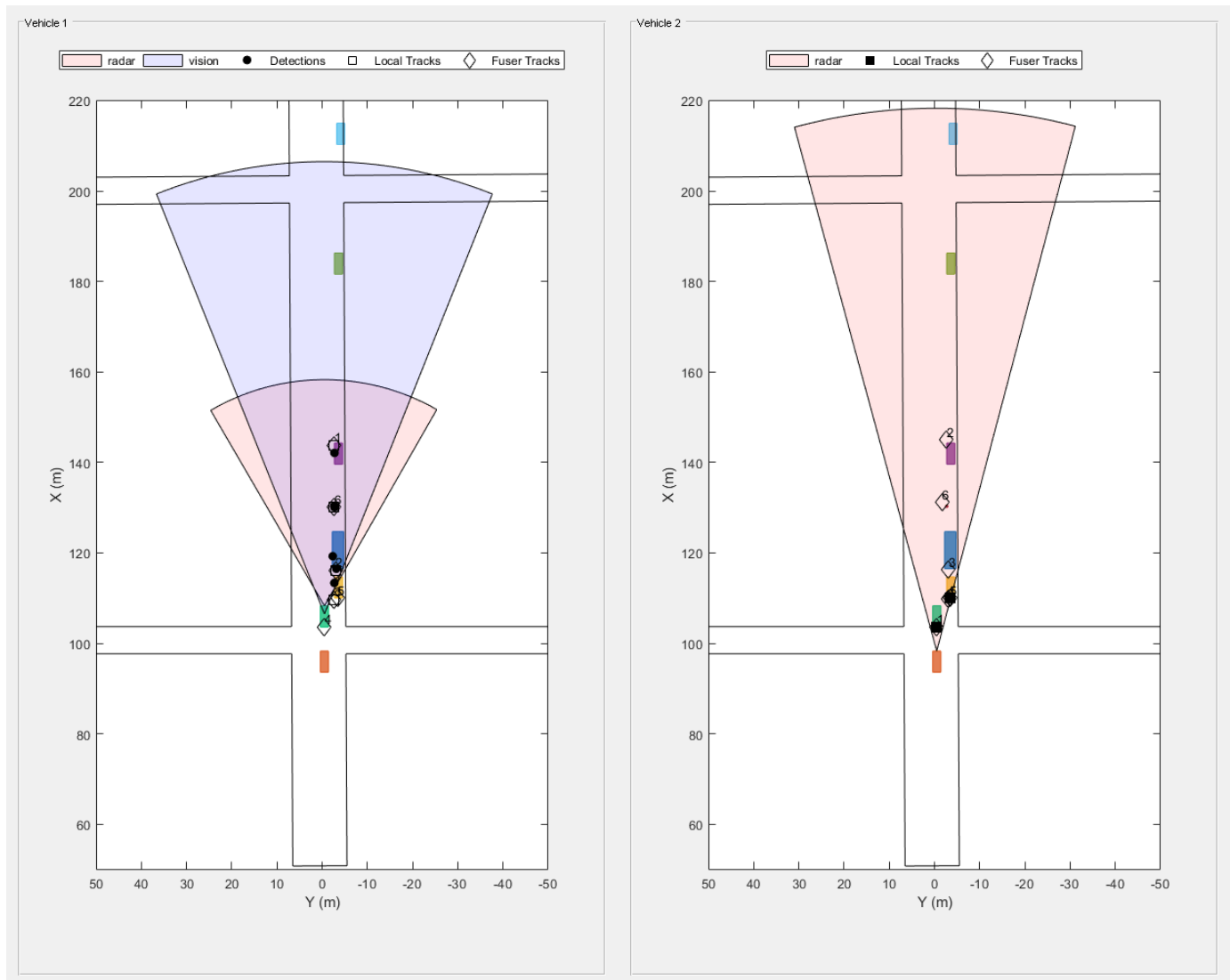
```
showsnap(snaps,3)
```
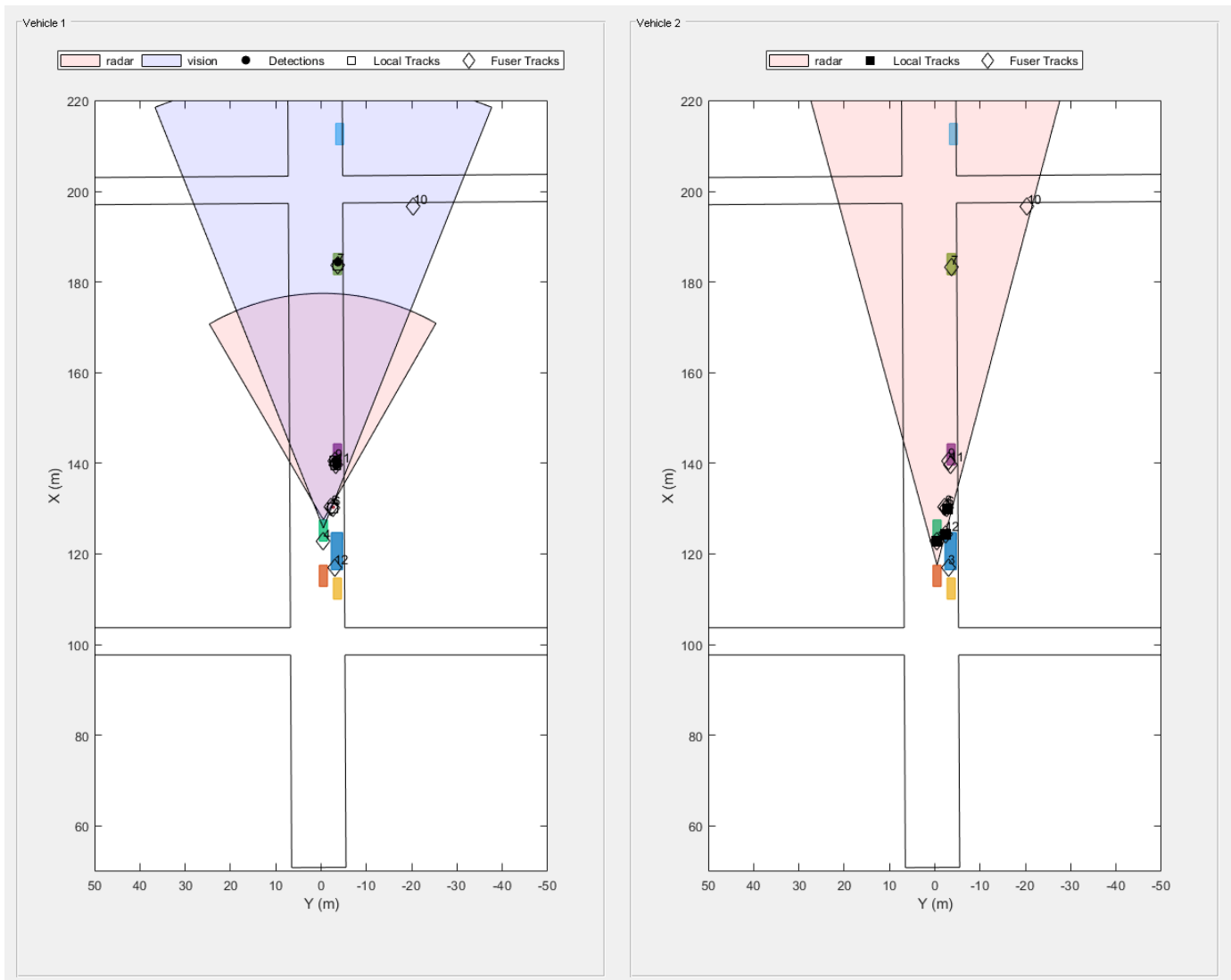
Avoiding Rumor Propagation

When the two vehicles communicate tracks to each other, there is a risk that they will continue communicating information about objects that they do not detect anymore just by repeating what the other vehicle communicated. This situation is called rumor propagation.

As the vehicles pass the objects, and these objects go out of their field of view, the fused tracks associated with these objects are dropped by both trackers (see snapshot 4). Dropping the tracks demonstrates that the fused tracks broadcast back and forth between the two vehicles are not used to propagate rumors.

```
showsnap(snaps,4)
```
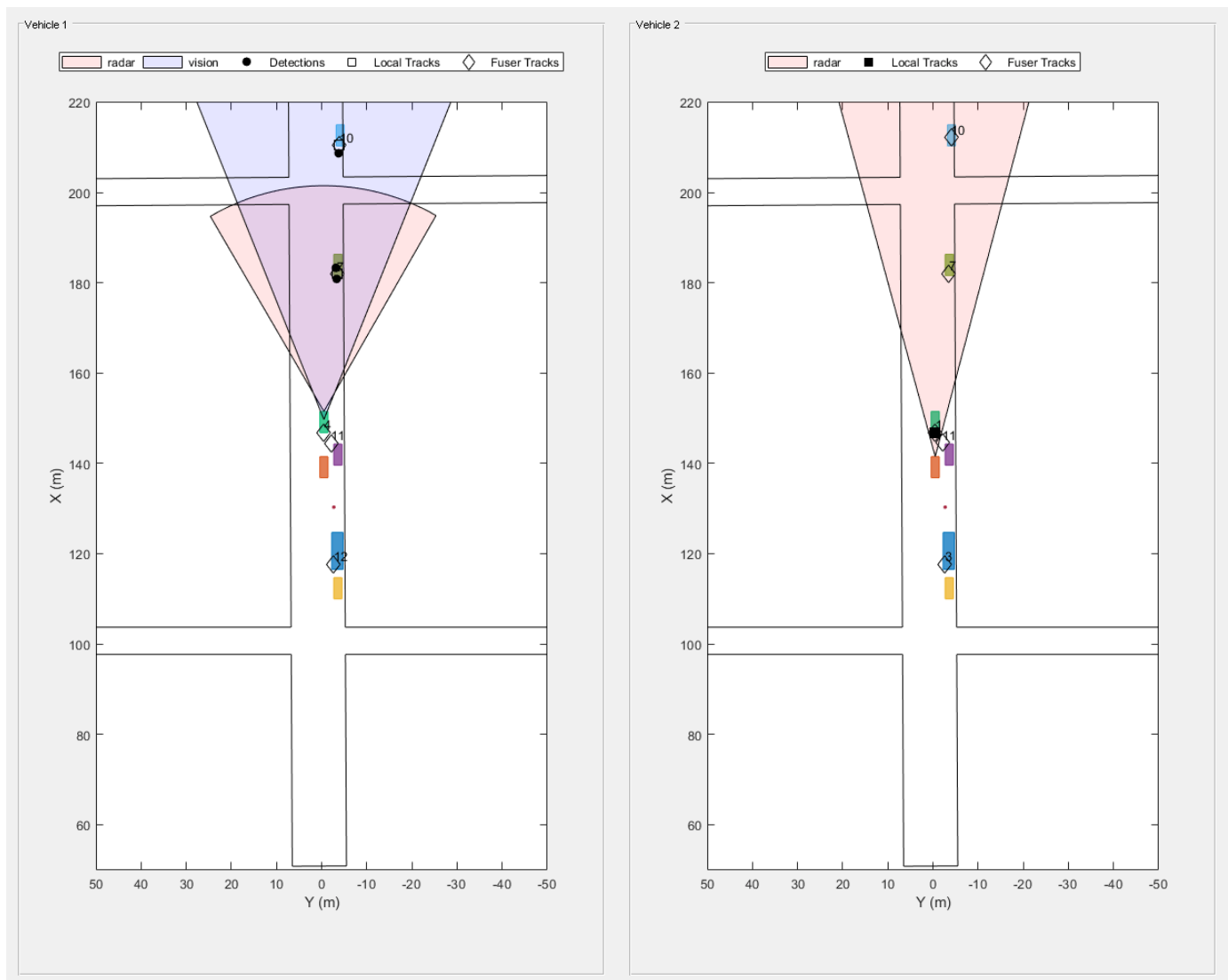
```
% Restart the driving scenario to return the actors to their initial positions
restart(scenario);

% Release all the sensor objects so they can be used again
for sensorIndex = 1:numSensors
    release(sensors{sensorIndex});
end

% Return the random seed to its previous value
rng(s)
```

Summary

In this example, you saw how track-to-track fusion can enhance the situational awareness and increase the safety in automotive applications. You saw how to set up a `trackFuser` to perform track-to-track fusion and how to define sources as either internal or external by using the

fuserSourceConfiguration object. By doing so, you avoid rumor propagation and keep only the fused tracks that are really observed by each vehicle to be maintained.

References

[1] Duraisamy, B., T. Schwarz, and C. Wohler. "Track Level Fusion Algorithms for Automotive Safety Applications." In *2013 International Conference on Signal Processing , Image Processing & Pattern Recognition,* 179–84, 2013. https://doi.org/10.1109/ICSIPR.2013.6497983.

Supporting Functions

createDrivingScenario

Create a driving scenario defined in the **Driving Scenario Designer** app.

```matlab
function [scenario, egoVehicle, secondVehicle] = createDrivingScenario
% Construct a drivingScenario object
scenario = drivingScenario('SampleTime', 0.1);

% Add all road segments
roadCenters = [50.8 0.5 0; 253.4 1.5 0];
roadWidth = 12;
road(scenario, roadCenters, roadWidth);

roadCenters = [100.7 -100.6 0; 100.7 103.7 0];
road(scenario, roadCenters);

roadCenters = [201.1 -99.2 0; 199.7 99.5 0];
road(scenario, roadCenters);

% Add the ego vehicle
egoVehicle = vehicle(scenario, 'ClassID', 1, 'Position', [65.1 -0.9 0], 'PlotColor', [0 0.7410 0
waypoints = [71 -0.5 0; 148.7 -0.5 0];
speed = 12;
trajectory(egoVehicle, waypoints, speed);

% Add the second vehicle
secondVehicle = vehicle(scenario, 'ClassID', 1, 'Position', [55.1 -0.9 0]);
waypoints = [61 -0.5 0; 138.7 -0.5 0];
speed = 12;
trajectory(secondVehicle, waypoints, speed);

% Add the parked cars
vehicle(scenario, 'ClassID', 1, 'Position', [111.0 -3.6 0]);
vehicle(scenario, 'ClassID', 1, 'Position', [140.6 -3.6 0]);
vehicle(scenario, 'ClassID', 1, 'Position', [182.6 -3.6 0]);
vehicle(scenario, 'ClassID', 1, 'Position', [211.3 -4.1 0]);

% Add pedestrian
actor(scenario, 'ClassID', 4, 'Length', 0.5, 'Width', 0.5, ...
    'Height', 1.7, 'Position', [130.3 -2.7 0], 'RCSPattern', [-8 -8;-8 -8]);

% Add parked truck
vehicle(scenario, 'ClassID', 2, 'Length', 8.2, 'Width', 2.5, ...
    'Height', 3.5, 'Position', [117.5 -3.5 0]);
end
```

createSensors

Create the sensors used in the scenario and list their attachments to vehicles.

```
function [sensors, numSensors, attachedVehicle] = createSensors(scenario)
% createSensors Returns all sensor objects to generate detections
% Units used in createSensors and createDrivingScenario
% Distance/Position - meters
% Speed             - meters/second
% Angles            - degrees
% RCS Pattern       - dBsm

% Assign into each sensor the physical and radar profiles for all actors
profiles = actorProfiles(scenario);

% Vehicle 1 radar reports clustered detections
sensors{1} = radarDataGenerator('No scanning', 'SensorIndex', 1, 'UpdateRate', 10, ...
    'MountingLocation', [3.7 0 0.2], 'RangeLimits', [0 50], 'FieldOfView', [60 5], ...
    'RangeResolution', 2.5, 'AzimuthResolution', 4, ...
    'Profiles', profiles, 'HasOcclusion', true, 'HasFalseAlarms', false, ...
    'TargetReportFormat', 'Clustered detections');

% Vehicle 2 radar reports tracks
sensors{2} = radarDataGenerator('No scanning', 'SensorIndex', 2, 'UpdateRate', 10, ...
    'MountingLocation', [3.7 0 0.2], 'RangeLimits', [0 120], 'FieldOfView', [30 5], ...
    'RangeResolution', 2.5, 'AzimuthResolution', 4, ...
    'Profiles', profiles, 'HasOcclusion', true, 'HasFalseAlarms', false, ...
    'TargetReportFormat', 'Tracks', 'DeletionThreshold', [3 3]);

% Vehicle 1 vision sensor reports detections
sensors{3} = visionDetectionGenerator('SensorIndex', 3, ...
    'MaxRange', 100, 'SensorLocation', [1.9 0], 'DetectorOutput', 'Objects only', ...
    'ActorProfiles', profiles);
attachedVehicle = [1;2;1];
numSensors = numel(sensors);
end
```

scenarioToEgo

Perform coordinate transformation from scenario to ego coordinates.

`trackInScenario` has `StateParameters` defined to transform it from scenario coordinates to ego coordinates.

The state uses the constant velocity model [x;vx;y;vy;z;vz].

```
function trackInEgo = scenarioToEgo(trackInScenario)
egoPosInScenario = trackInScenario.StateParameters.OriginPosition;
egoVelInScenario = trackInScenario.StateParameters.OriginVelocity;
stateInScenario = trackInScenario.State;
stateShift = [egoPosInScenario(1);egoVelInScenario(1);egoPosInScenario(2);egoVelInScenario(2);ego
stateInEgo = stateInScenario - stateShift;
trackInEgo = objectTrack('UpdateTime',trackInScenario.UpdateTime,'State',stateInEgo,'StateCovaria
end
```

egoToScenario

Perform coordinate transformation from ego to scenario coordinates.

`trackInEgo` has `StateParameters` defined to transform it from ego coordinates to scenario coordinates.

The state uses the constant velocity model [x;vx;y;vy;z;vz].

```
function trackInScenario = egoToScenario(trackInEgo)
egoPosInScenario = trackInEgo.StateParameters.OriginPosition;
egoVelInScenario = trackInEgo.StateParameters.OriginVelocity;
stateInScenario = trackInEgo.State;
stateShift = [egoPosInScenario(1);egoVelInScenario(1);egoPosInScenario(2);egoVelInScenario(2);ego
stateInEgo = stateInScenario + stateShift;
trackInScenario = objectTrack('UpdateTime',trackInEgo.UpdateTime,'State',stateInEgo,'StateCovaria
end
```

detectAndTrack

This function is used for collecting all the detections from the sensors in one vehicle and updating the tracker with them.

The agent is a structure that contains the actor information and the sensors, tracker, and plotter to plot detections and vehicle tracks.

```
function [tracks,wasTrackerUpdated] = detectAndTrack(agent,time,posSelector)
% Create detections from the vehicle
poses = targetPoses(agent.Actor);
[detections,isValid] = vehicleDetections(agent.Actor.Position,agent.Sensors,poses,time,agent.DetP

% Update the tracker to get tracks from sensors that reported detections
if isValid
    agent.Tracker.StateParameters = struct(...
        'Frame','Rectangular', ...
        'OriginPosition', agent.Actor.Position, ...
        'OriginVelocity', agent.Actor.Velocity);
    tracks = agent.Tracker(detections,time);
    tracksInScenario = tracks;
    for i = 1:numel(tracks)
        tracksInScenario(i) = egoToScenario(tracks(i));
    end
    pos = getTrackPositions(tracksInScenario,posSelector);
    plotTrack(agent.TrkPlotter,pos)
    wasTrackerUpdated = true;
else
    tracks = objectTrack.empty(0,1);
    wasTrackerUpdated = false;
end

% Get additional tracks from tracking sensors
[sensorTracks,wasSensorTrackerUpdated] = vehicleTracks(agent.Actor,agent.Sensors,poses,time,agent
tracks = vertcat(tracks,sensorTracks);
wasTrackerUpdated = wasTrackerUpdated || wasSensorTrackerUpdated;
end
```

vehicleDetections

Collect the detections from all the sensors attached to this vehicle that return detections.

```
function [objectDetections,isValid] = vehicleDetections(position, sensors, poses, time, plotter)
numSensors = numel(sensors);
```

```matlab
objectDetections = {};
isValidTime = false(1, numSensors);

% Generate detections for each sensor
for sensorIndex = 1:numSensors
    sensor = sensors{sensorIndex};
    if isa(sensor, 'visionDetectionGenerator') || ~strcmpi(sensor.TargetReportFormat,'Tracks')
        [objectDets, ~, sensorConfig] = sensor(poses, time);
        if islogical(sensorConfig)
            isValidTime(sensorIndex) = sensorConfig;
        else
            isValidTime(sensorIndex) = sensorConfig.IsValidTime;
        end
        objectDets = cellfun(@(d) setAtt(d), objectDets, 'UniformOutput', false);
        numObjects = numel(objectDets);
        objectDetections = [objectDetections; objectDets(1:numObjects)]; %#ok<AGROW>
    end
end
isValid = any(isValidTime);

% Plot detections
if numel(objectDetections)>0
    detPos = cellfun(@(d)d.Measurement(1:2), objectDetections, 'UniformOutput', false);
    detPos = cell2mat(detPos')' + position(1:2);
    plotDetection(plotter, detPos);
end
end

function d = setAtt(d)
% Set the attributes to be a structure
d.ObjectAttributes = struct;
% Keep only the position measurement and remove velocity
if numel(d.Measurement)==6
    d.Measurement = d.Measurement(1:3);
    d.MeasurementNoise = d.MeasurementNoise(1:3,1:3);
    d.MeasurementParameters{1}.HasVelocity = false;
end
end
```

vehicleTracks

Collect all the tracks from sensors that report tracks on the vehicle.

```matlab
function [tracks,wasTrackerUpdated] = vehicleTracks(actor, sensors, poses, time, plotter)
% Create detections from the vehicle
numSensors = numel(sensors);
tracks = objectTrack.empty;
isValidTime = false(1, numSensors);

% Generate detections for each sensor
for sensorIndex = 1:numSensors
    sensor = sensors{sensorIndex};
    if isa(sensor, 'radarDataGenerator') && strcmpi(sensor.TargetReportFormat,'Tracks')
        [sensorTracks, ~, sensorConfig] = sensor(poses, time);
        if islogical(sensorConfig)
            isValidTime(sensorIndex) = sensorConfig;
        else
            isValidTime(sensorIndex) = sensorConfig.IsValidTime;
```

```matlab
        end
        numObjects = numel(sensorTracks);
        tracks = [tracks; sensorTracks(1:numObjects)]; %#ok<AGROW>
    end
end
wasTrackerUpdated = any(isValidTime);

if ~wasTrackerUpdated % No vehicle tracking sensor udpated
    return
end

% Add vehicle position and velocity to track state parameters
for i = 1:numel(tracks)
    tracks(i).StateParameters.OriginPosition = tracks(i).StateParameters.OriginPosition + actor.P
    tracks(i).StateParameters.OriginVelocity = tracks(i).StateParameters.OriginVelocity + actor.V
end

% Plot tracks
if numel(tracks)>0
    trPos = arrayfun(@(t)t.State([1,3]), tracks, 'UniformOutput', false);
    trPos = cell2mat(trPos')' + actor.Position(1:2);
    plotTrack(plotter, trPos);
end
end
```

Track-to-Track Fusion for Automotive Safety Applications in Simulink

This example shows how to perform track-to-track fusion in Simulink® with Sensor Fusion and Tracking Toolbox™. In the context of autonomous driving, the example illustrates how to build a decentralized tracking architecture using a Track-To-Track Fuser block. In the example, each vehicle performs tracking independently as well as fuses tracking information received from other vehicles. This example closely follows the "Track-to-Track Fusion for Automotive Safety Applications" (Sensor Fusion and Tracking Toolbox) MATLAB® example.

Introduction

Automotive safety applications largely rely on the situational awareness of the vehicle. A better situational awareness provides the basis to a successful decision-making for different situations. To achieve this, vehicles can benefit from intervehicle data fusion. This example illustrates the workflow in Simulink for fusing data from two vehicles to enhance situational awareness of the vehicle.

Setup and Overview of the Model
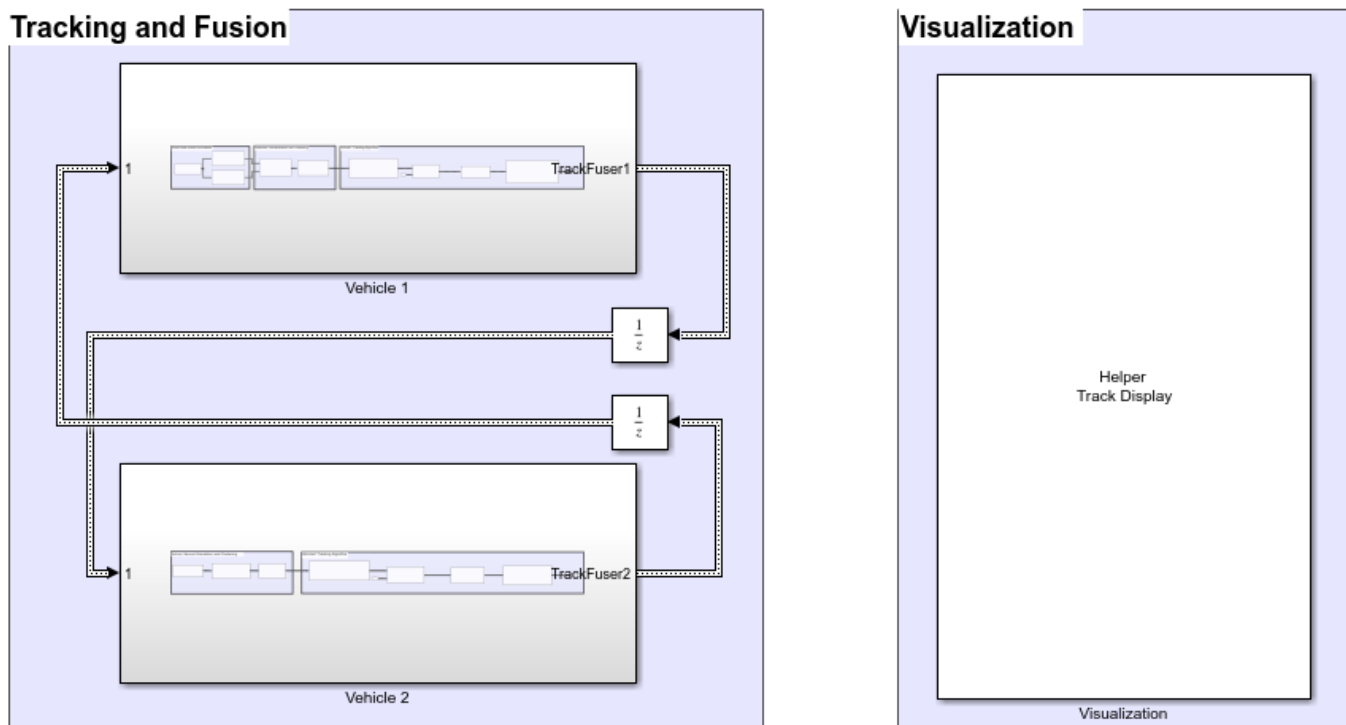
Prior to running this example, the `drivingScenario` object was used to create the same scenario defined in "Track-to-Track Fusion for Automotive Safety Applications" (Sensor Fusion and Tracking Toolbox). The roads and actors from this scenario were then saved to the scenario object file `TrackToTrackFusionScenario.mat`.

Tracking and Fusion

In the Tracking and Fusion section of the model there are two subsystems that implement the target tracking and fusion capabilities of Vehicle 1 and Vehicle 2 in this scenario.

Vehicle 1 Subsystem
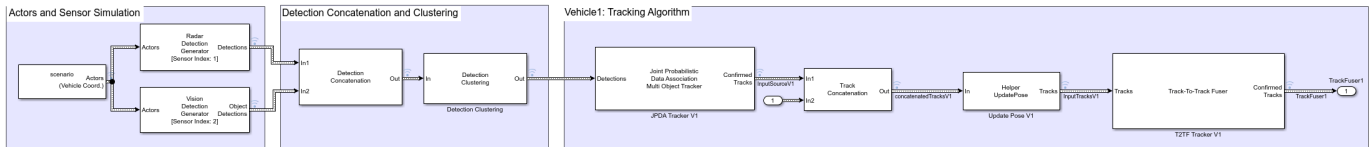
This subsystem includes the Scenario Reader (Automated Driving Toolbox) block that reads the actor pose data from the saved file. The block converts the actor poses from the world coordinates of the scenario into ego vehicle coordinates. The actor poses are streamed on a bus generated by the block. The actor poses are used by the Sensor Simulation subsystem, which generates radar and vision detections. These detections are then passed to the JPDA Tracker V1 block, which processes the detections to generate a list of tracks. The tracks are then passed into a Track Concatenation1 block, which concatenates these input tracks. The first input to the Track Concatenation1 block is the local tracks from the JPDA tracker and the second input is the tracks received from the track fuser of the other vehicle. To transform local tracks to central tracks, the track fuser needs the parameter information about the local tracks. However, this information is not available from the direct outputs of the JPDA tracker. Therefore, a helper Update Pose block is used to supply this information by reading the data from the v1Pose.mat file. The updated tracks are then broadcasted to T2TF Tracker V1 block as an input. Finally, the Track-To-Track Fuser (Sensor Fusion and Tracking Toolbox) T2TF Tracker V1 block fuse the local vehicle tracks with the tracks received from the track fuser of the other vehicle. After each update, the track fuser on each vehicle broadcasts its fused tracks to be fed into the update of the track fuser of the other vehicle in the next time stamp.

Vehicle 2 Subsystem
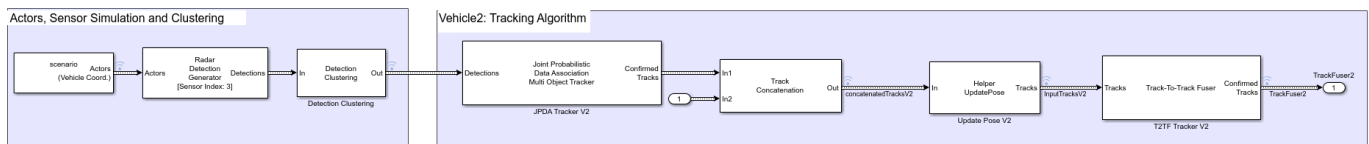
Vehicle 2 subsystem follows a similar setup as the Vehicle 1 subsystem.

Visualization

The Visualization block is implemented using the MATLAB System block and is defined using the HelperTrackDisplay block. The block uses RunTimeObject parameters Out, Confirmed Tracks, Tracks and Confirmed Tracks of Detection Clustering, JPDA Tracker V1, Update Pose V1, T2TF Tracker V1 blocks respectively for vehicle 1 and RunTimeObject parameters Out, Confirmed Tracks, Tracks and Confirmed Tracks of Detection Clustering, JPDA Tracker V2, Update Pose V2, T2TF Tracker V2 blocks respectively for vehicle 2 to display their outputs. See "Access Block Data During Simulation" (Simulink) for further information on how to access block outputs during simulation.

Results

After running the model, you can visualize the results. This animation shows the results for this simulation.

The visualization includes two panels. The left panel shows the detections, local tracks, and fused tracks that vehicle 1 generated during the simulation and represents the situational awareness of vehicle 1. The right panel shows the situational awareness of vehicle 2.

The recorded detections are represented by black circles. The local and fused tracks from vehicle 1 are represented by a square and a diamond, respectively. The local and fused tracks from vehicle 2 represented by a solid black square and a diamond. At the start of simulation, vehicle 1 detects vehicles parked on the right side of the street, and tracks associated with the parked vehicles are confirmed. Currently, vehicle 2 only detects vehicle 1 which is immediately in front of it. As the simulation continues, the confirmed tracks from vehicle 1 are broadcast to the fuser on vehicle 2. After fusing the tracks, vehicle 2 becomes aware of the objects prior to detecting these objects on its own. Similarly, vehicle 2 tracks are broadcast to vehicle 1. Vehicle 1 fuses these tracks and becomes aware of the objects prior to detecting them on its own.

In particular, you observe that the pedestrian standing between the blue and purple cars on the right side of the street is detected and tracked by vehicle 1. Vehicle 2 first becomes aware of the pedestrian by fusing the track from Vehicle 1 at around 0.8 seconds. It takes vehicle 2 roughly 3 seconds before it starts detecting the pedestrian using its own sensor. The ability to track a pedestrian based on inputs from vehicle 1 allows vehicle 2 to extend its situational awareness and to mitigate the risk of accident.
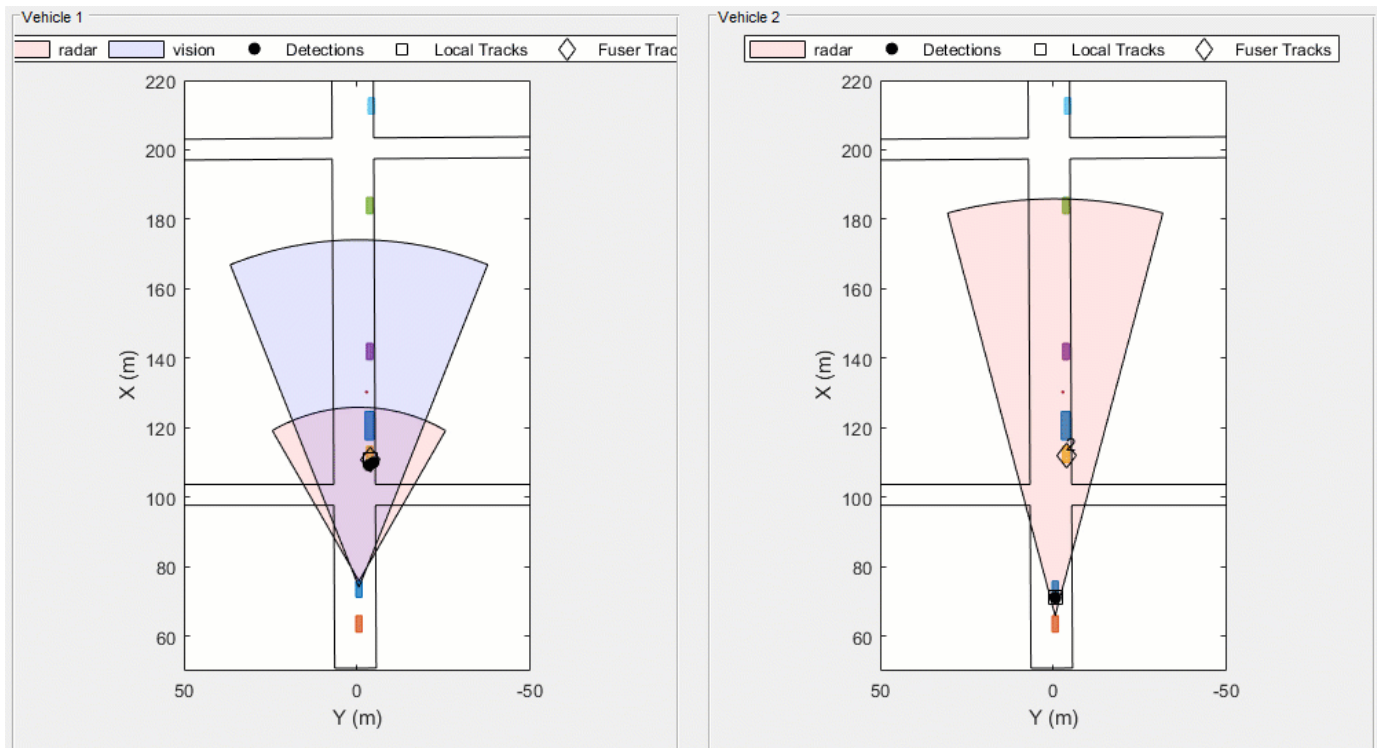
Summary

This example showed how to perform track-to-track fusion in Simulink. You learned how to perform tracking using a decentralized tracking architecture, where each vehicle is responsible for maintaining its own local tracks, fuse tracks from other vehicles, and communicate the tracks to the other vehicle. You also use a JPDA tracker block to generate the local tracks.

Convert Detections to objectDetection Format

These examples show how to convert actual detections in the native format of the sensor into `objectDetection` objects. `objectDetection` is the standard input format for most tracking filters and trackers in the toolbox. The six examples progressively show how to set up `objectDetection` with varied tracking scenarios.

- Example 1 on page 1-93 configures the detection in a stationary rectangular frame.
- Example 2 on page 1-94 configures the detection in a moving rectangular frame.
- Example 3 on page 1-96 configures the detection in a moving spherical frame.
- Example 4 on page 1-98 shows how to express detections obtained by consecutive rotations.
- Example 5 on page 1-99 shows how to configure 3-D detections.
- Example 6 on page 1-101 shows how to configure classified detections.

An `objectDetection` report must contain the basic detection information: `Time` and `Measurement`. It can also contain other key properties, including `MeasurementNoise`, `SensorIndex`, `ObjectClassID`, `ObjectClassParameters`, `ObjectAttributes`, and `MeasurementParameters`. Setting up `MeasurementParameters` correctly so that a filter or tracker can interpret the measurement is crucial in creating `objectDetection`. The first example shows the basic setup of an `objectDetection`. Examples 2 through 5 focus on how to correctly set up `MeasurementParameters`. The last example shows how to set up `ObjectClassParameters`.

Example 1: Convert Detections in Stationary Rectangular Frame

Consider a 2-D tracking scenario with a stationary tower and a truck. The tower located at the origin of the scenario frame is equipped with a radar sensor. At $t = 0$ seconds, the truck at the position of (10,20,0) meters is traveling in the positive X direction at a speed of 5 m/s.
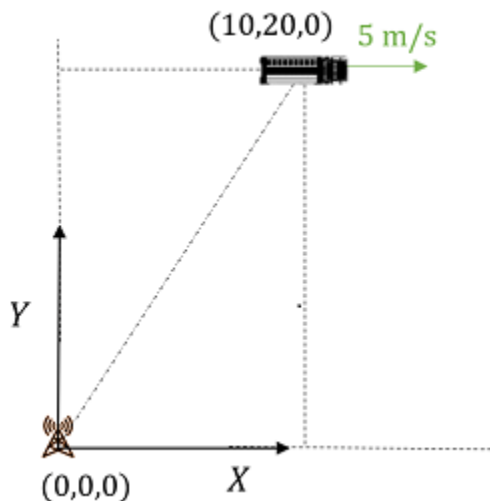
The radar sensor outputs 3-D position and velocity measurements in the scenario frame, so the measurement can be written as follows:

```
measurement1 = [10;20;0;5;0;0]; % [x;y;z;vx;vy;vz]
```

You can specify additional properties such as `MeasurmentNoise`, `SensorIndex`, `ObjectClassID`, and `ObjectAttributes` for the `objectDetection` object. For example, assuming the standard deviation of the position and velocity measurement noise is 10 m and 1 m/s, respectively, you can define the measurement error covariance matrix as:

```
measurementNoise1 = diag([10*ones(3,1);ones(3,1)]);
```

Create an `objectDetection` using these values.

```
time1 = 0; % detection time
detect1 = objectDetection(time1,measurement1,'MeasurementNoise',measurementNoise1)

detect1 =
  objectDetection with properties:

                   Time: 0
            Measurement: [6x1 double]
       MeasurementNoise: [6x6 double]
            SensorIndex: 1
          ObjectClassID: 0
    ObjectClassParameters: []
    MeasurementParameters: {}
        ObjectAttributes: {}
```

Example 2: Convert Detections in Moving Rectangular Frame

Consider a 2-D tracking scenario with an ego car and a truck. At $t = 0$ seconds, the car is located at $(20,10,0)$ meters with respect to the scenario frame. The car is moving with a speed of 5 m/s in the Y direction of the scenario frame. The local (forward) frame of the ego car, $\{x,y\}$, rotates from the scenario frame by an angle of 90 degrees. As in the previous example, a truck at the position of $(10,20,0)$ meters is traveling in the positive X direction at a speed of 5 m/s.
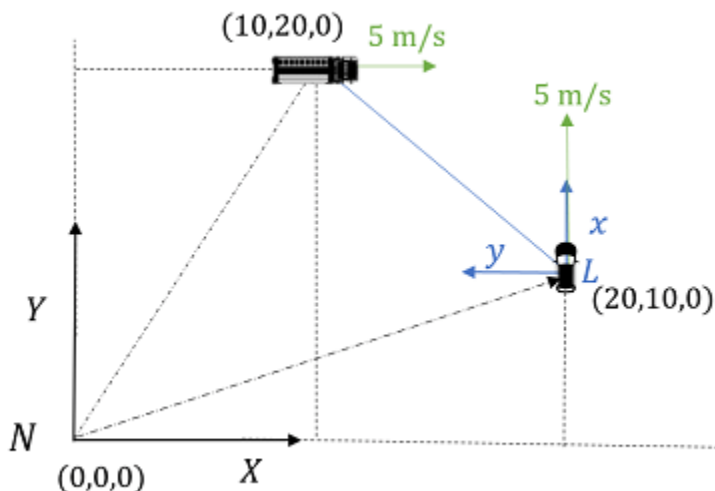
Meanwhile, the ego car is observing the truck in its own local frame, $\{x,y\}$. In practice, you can obtain the measurement directly from the sensor system of the ego car. From the figure, the measurements of the truck are [10; 10; 0 -5; -5; 0] with respect to the $\{x,y\}$ frame in the order of [x;y;z;vx;vy;vz].

```
measurement2 = [10; 10; 0; -5;  -5; 0]; % [x;y;z;vx;vy;vz]
```

To specify the object detection, you need to specify the coordinate transformation from the scenario rectangular frame {*X*,*Y*} to the local rectangular frame {*x*,*y*}. You can use the `MeasurementParameters` property of `objectDetection` to specify these transformation parameters. In the transformation, the scenario frame is the *parent* frame, and the ego car local frame is the *child* frame.

- The `Frame` property sets the child frame type to `'rectangular'`(in this example) or `'spherical'`.
- The `OriginPosition` property sets the position of the origin of the child frame with respect to the parent frame.
- The `OriginVelocity` property sets the velocity of the origin of the child frame with respect to the parent frame.

```
MP2 = struct();

MP2.Frame = 'rectangular';
MP2.OriginPosition =[20; 10; 0];
MP2.OriginVelocity = [0; 5; 0];
```

Specify rotation using the rotation matrix converted from Euler angles. Set `IsParentToChild` to true to indicate rotation from the parent frame to the child frame.

```
rotAngle2 = [90 0 0]; % [yaw,pitch,row]
rotQuat2 = quaternion(rotAngle2,'Eulerd','ZYX','frame');
rotMatrix2 = rotmat(rotQuat2,'frame');
MP2.Orientation = rotMatrix2;
MP2.IsParentToChild = true;
```

Specify measurements.

- Set `HasElevation` and `HasAzimuth` both to `false`, since the child frame is rectangular.
- Set `HasRange` to `true` to enable position measurement.
- Set `HasVelocity` to `true` to enable velocity measurement.

```
MP2.HasElevation = false;
MP2.HasAzimuth = false;
MP2.HasRange = true;
MP2.HasVelocity = true;
```

Create the `objectDetection` object and specify the `MeasurementParameters` property.

```
time2 = 0;
detection2 = objectDetection(time2,measurement2,'MeasurementParameters',MP2)

detection2 =
  objectDetection with properties:

                    Time: 0
             Measurement: [6x1 double]
        MeasurementNoise: [6x6 double]
             SensorIndex: 1
           ObjectClassID: 0
    ObjectClassParameters: []
    MeasurementParameters: [1x1 struct]
```

```
        ObjectAttributes: {}
```

To verify the object detection, you can use the `cvmeas` (Sensor Fusion and Tracking Toolbox) measurement function to regenerate the measurement. The `cvmeas` function can take the actual state of the target and measurement parameters as input. The state input of `cvmeas` is in the order of [x;vx;y;vy;z;vz]. As shown in the following output, the results agree with `measurement2`.

```
state2 =[10;5;20;0;0;0]; % State of truck in [x;vx;y;vy;z;vz]
cvmeas2 = cvmeas(state2,MP2) % Measurement in [x;y;z;vx;vy;vz]
```

```
cvmeas2 = 6×1

    10.0000
    10.0000
         0
    -5.0000
    -5.0000
         0
```

Example 3: Convert Detections in Moving Spherical Frame

Consider the previous tracking scenario, only now the measurement is obtained by a scanning radar with a spherical output frame. The boresight direction of the radar is aligned with the *Y* direction (same as *x* direction) at *t* = 0 seconds.
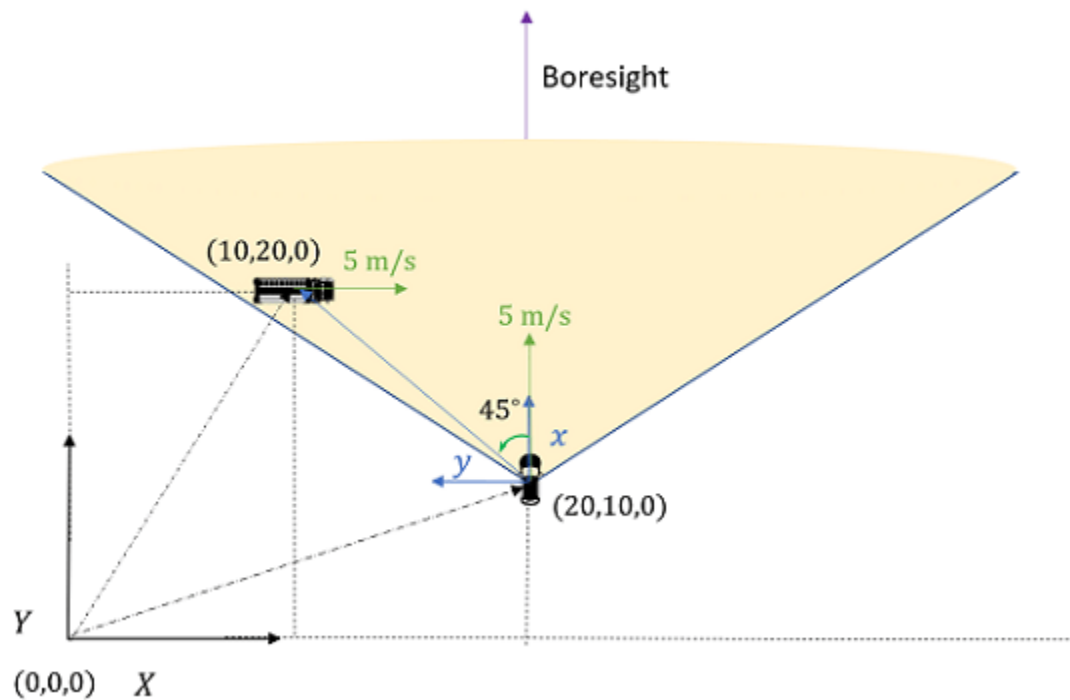
Since the relative velocity between the truck and the car is in the line-of-sight direction, the measurement, which is in the order of [azimuth; elevation; range; range-rate], can be obtained as follows:

```
measurement3 =[45; 0; 10/sind(45); -5/sind(45)]; % [az;el;rng;rr]. Units in degrees.
```

Specify the measurement parameters.

```
MP3 = struct();

MP3.Frame = 'spherical'; % The child frame is spherical.
MP3.OriginPosition = [20; 10; 0];
MP3.OriginVelocity = [0; 5; 0];

% Specify rotation.
rotAngle3 = [90 0 0];
rotQuat3 = quaternion(rotAngle3,'Eulerd','ZYX','frame');
rotMatrix3 = rotmat(rotQuat3,'frame');
MP3.Orientation = rotMatrix3;
MP3.IsParentToChild = true;
```

Set `HasElevation` and `HasAzimuth` to `true` to output azimuth and elevation angles in the spherical child frame. Set `HasRange` and `HasVelocity` both to `true` to output range and range-rate, respectively.

```
MP3.HasElevation = true;
MP3.HasAzimuth = true;
MP3.HasRange = true;
MP3.HasVelocity = true;
```

Create the `objectDetection` object.

```
time3 = 0;
detection3 = objectDetection(time3,measurement3,'MeasurementParameters',MP3)

detection3 =
  objectDetection with properties:

                     Time: 0
              Measurement: [4x1 double]
         MeasurementNoise: [4x4 double]
              SensorIndex: 1
            ObjectClassID: 0
    ObjectClassParameters: []
    MeasurementParameters: [1x1 struct]
         ObjectAttributes: {}
```

Verify the results using `cvmeas`. The results agree with `measurement3`.

```
state3 = [10;5;20;0;0;0]; % [x;vx;y;vy;z;vz]
cvmeas3 = cvmeas(state3,MP3) % [az;el;rng;rr]

cvmeas3 = 4×1

   45.0000
         0
   14.1421
   -7.0711
```

Example 4: Convert Detections Between Three Frames

Consider the previous tracking scenario, only now the boresight direction of the radar rotates 45 degrees from the *x* direction of the car's local frame.
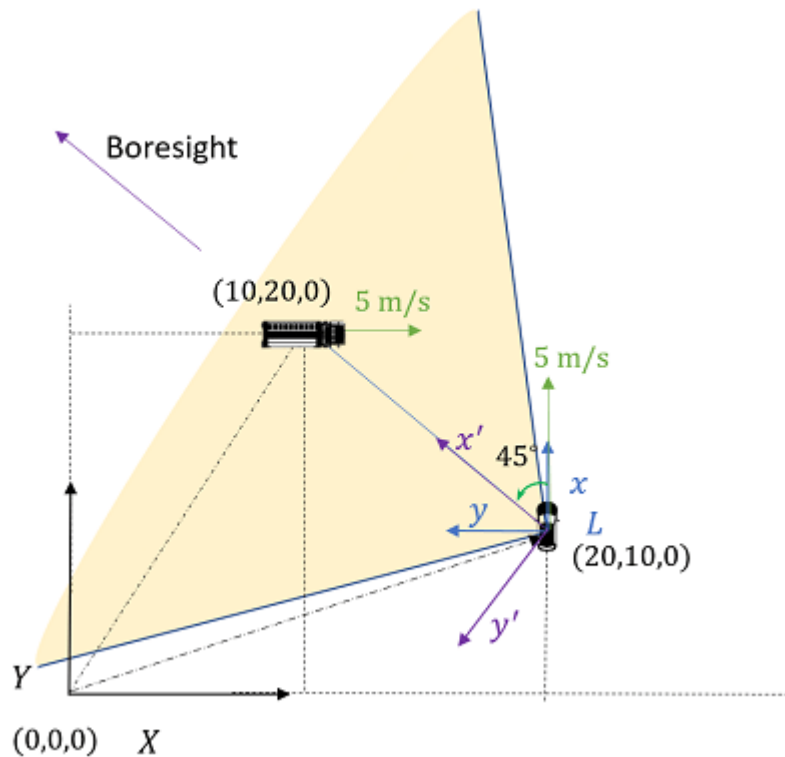
The new measurements, expressed in the new spherical frame {x',y'}, are:

```
measurement4 = [0; 0; 10/sind(45); -5/sind(45)]; % [az;el;rng;rr]
```

For the measurement parameters, you can specify the rotation as a 135-degree rotation from the scenario frame to the new spherical frame. Alternately, you can specify it as two consecutive rotations: rectangular {X,Y} to rectangular {x,y} and rectangular {x,y} to spherical {x',y'}. To illustrate the multiple frame transformation feature supported by the `MeasurementParameters` property, this example uses the latter approach.

The first set of measurement parameters is exactly the same as `MP2` used in Example 2 on page 1-94. `MP2` accounts for the rotation from the rectangular {X,Y} to the rectangular {x,y}. For the second set of measurement parameters, `MP4`, you need to specify only a 45-degree rotation from the rectangular {x,y} to the spherical {x',y'}.

```
MP4 = struct();

MP4.Frame = 'spherical';
MP4.OriginPosition =[0; 0; 0]; % Colocated positions.
MP4.OriginVelocity = [0; 0; 0]; % Same origin velocities.

% Specify rotation.
rotAngle4 = [45 0 0];
rotQuat4 = quaternion(rotAngle4,'Eulerd','ZYX','frame');
```

```
rotMatrix4 = rotmat(rotQuat4,'frame');
MP4.Orientation = rotMatrix4;
MP4.IsParentToChild = true;

% Specify outputs in the spherical child frame.
MP4.HasElevation = true;
MP4.HasAzimuth = true;
MP4.HasRange = true;
MP4.HasVelocity = true;
```

Create the combined `MeasurementParameters` input, `MPc`.

```
MPc =[MP4 MP2];
```

Note that the sequence of measurement parameters matters here. The specified `MPc` represents first performing coordinate transformation corresponding to MP2 and then performing coordinate transformation corresponding to MP4.

Next, create the `objectDetection` object.

```
time4 = 0;
detection4 = objectDetection(time4,measurement4,'MeasurementParameters',MPc)

detection4 =
  objectDetection with properties:

                     Time: 0
              Measurement: [4x1 double]
         MeasurementNoise: [4x4 double]
              SensorIndex: 1
            ObjectClassID: 0
    ObjectClassParameters: []
    MeasurementParameters: [1x2 struct]
         ObjectAttributes: {}
```

Verify the results using `cvmeas`. The result agrees with `measurement4`.

```
state4 = [10;5;20;0;0;0]; % [x;vx;y;vy;z;vz]
cvmeas4 = cvmeas(state4,MPc) % [az;el;rr;rrate]

cvmeas4 = 4×1

    0.0000
         0
   14.1421
   -7.0711
```

Example 5: Convert 3D Detections

Consider an unmanned aerial vehicle (UAV) monitoring a region. At t = 0 seconds, the UAV is at the position of (5,5,-1) km with respect to the global north-east-down (NED) frame. The velocity of the UAV is (-50,-100,5) m/s. The orientation of the UAV body frame {x,y,z} with respect to the global NED frame is given as (-120,2,2) degrees in yaw, pitch, and roll. At the same time, a car at the position of (1,1,0) km is moving east with a speed of 30 m/s. The UAV measures the car using a radar system aligned with its own body axis.
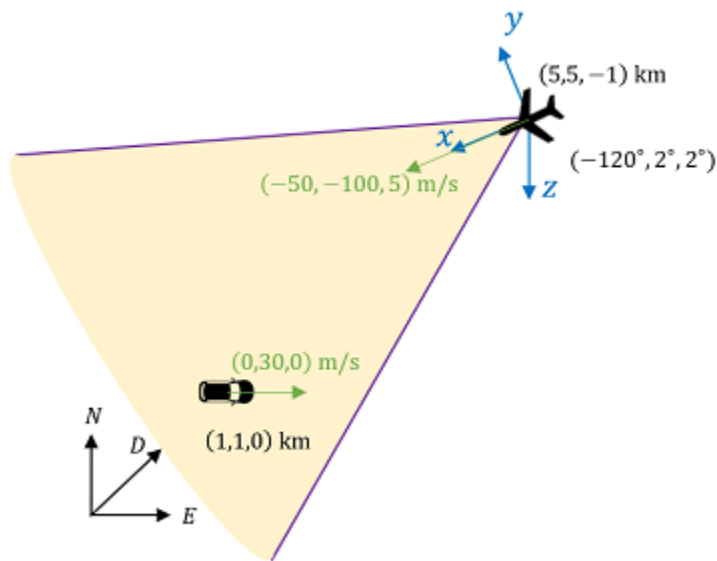
Based on this information, specify the kinematic parameters for the measurement transformation.

Specify the frame type, origin position, and origin velocity of the UAV body frame.

```
MP5 = struct();
```

```
MP5.Frame = 'spherical';
MP5.OriginPosition = [5000; 5000; -1000];
MP5.OriginVelocity = [-50; -100; 5];
```

Specify the rotation from the NED frame to the UAV body frame.

```
Rot_angle5 = [-120 2 2]; % [yaw,pitch,roll]
Rot_quat5 = quaternion(Rot_angle5,'Eulerd','ZYX','frame');
Rot_matrix5 = rotmat(Rot_quat5,'frame');
MP5.Orientation = Rot_matrix5;
MP5.IsParentToChild = true;
```

Specify the output measurements in a spherical frame.

```
MP5.HasElevation = true;
MP5.HasAzimuth = true;
MP5.HasRange = true;
MP5.HasVelocity = true;
```

You can obtain the measurement directly from the radar system on the UAV. Here, you use the `cvmeas` function to obtain the measurement in the order of [azimuth;elevation;range;range-rate].

```
car_state5 = [1000;0;1000;30;0;0]; % [x;vx;y;vy;z;vz].
measurement5 = cvmeas(car_state5,MP5);
meas_az5 = measurement5(1)
```

```
meas_az5 = -14.6825
```

```
meas_el5 = measurement5(2)
```

```
meas_el5 = 12.4704

meas_rng5 = measurement5(3)

meas_rng5 = 5.7446e+03

meas_rr5 = measurement5(4)

meas_rr5 = -126.2063
```

The elevation angle is defined as an angle from the xy-plane to the z direction. That is why the elevation angle is positive for a target on the ground relative to the UAV. This convention is used throughout the toolbox.

The measurement noise for azimuth, elevation, range, and range-rate is [1,1,20,2], respectively. Also, the index of the radar is 2, and the radar can classify the detected object as 1 for the type of 'car'.

```
index5 = 2;
covariance5 = diag([1;1;20;2]);
classID5 = 1;
```

Create an objectDetection object for the detection.

```
time5 = 0;
detection = objectDetection(time5,measurement5,'SensorIndex',index5,...
            'MeasurementNoise',covariance5,'ObjectClassID',classID5,'MeasurementParameters',MP5)

detection =
  objectDetection with properties:

                     Time: 0
              Measurement: [4x1 double]
         MeasurementNoise: [4x4 double]
              SensorIndex: 2
            ObjectClassID: 1
      ObjectClassParameters: []
    MeasurementParameters: [1x1 struct]
         ObjectAttributes: {}
```

Example 6: Classified Detections

Consider a vision tracking scenario where camera frames feed into an object detector. In such cases, detections often provide a classification of the object. Consider an object detector that output bounding box detections and classifies objects into the following classes {'Car', 'Pedestrian', 'Bicycle'}. The statistics of the detector are captured by its confusion matrix C. Create a detection with bounding box measurement, 'Pedestrian' classification, and confusion matrix C as defined below.

```
C = [0.9 0.05 0.05; ...
     0.05 0.9 0.05; ...
     0.05 0.05 0.9];
ClassID = 2; % Pedestrian
ClassParams = struct('ConfusionMatrix', C);

boundingbox = [250 140 300 400]; % bounding box in top left width height coordinates
detection = objectDetection(0, boundingbox, ObjectClassID=ClassID, ObjectClassParameters=struct(
```

1-101

```
detection =
  objectDetection with properties:

                     Time: 0
              Measurement: [250 140 300 400]
         MeasurementNoise: [4x4 double]
              SensorIndex: 1
            ObjectClassID: 2
      ObjectClassParameters: [1x1 struct]
     MeasurementParameters: {}
          ObjectAttributes: {}
```

Automotive Adaptive Cruise Control Using FMCW and MFSK Technology

This example shows how to model an automotive radar in Simulink® that includes adaptive cruise control (ACC), which is an important function of an advanced driver assistance system (ADAS). The example explores scenarios with a single target and multiple targets. It shows how frequency-modulated continuous-wave (FMCW) and multiple frequency-shift keying (MFSK) waveforms can be processed to estimate the range and speed of surrounding vehicles.

Available Example Implementations

This example includes four Simulink models:

- FMCW Radar Range Estimation: slexFMCWExample.slx
- FMCW Radar Range and Speed Estimation of Multiple Targets: slexFMCWMultiTargetsExample.slx
- MFSK Radar Range and Speed Estimation of Multiple Targets: slexMFSKMultiTargetsExample.slx
- FMCW Radar Range, Speed, and Angle Estimation of Multiple Targets: slexFMCWMultiTargetsDOAExample.slx

FMCW Radar Range Estimation

The following model shows an end-to-end FMCW radar system. The system setup is similar to the MATLAB® "Automotive Adaptive Cruise Control Using FMCW Technology" on page 1-148 example. The only difference between this model and the aforementioned example is that this model has an FMCW waveform sweep that is symmetric around the carrier frequency.

FMCW Radar Range Estimation

Radar Copyright 2014-2022 The MathWorks Inc.   *Channel and Target*
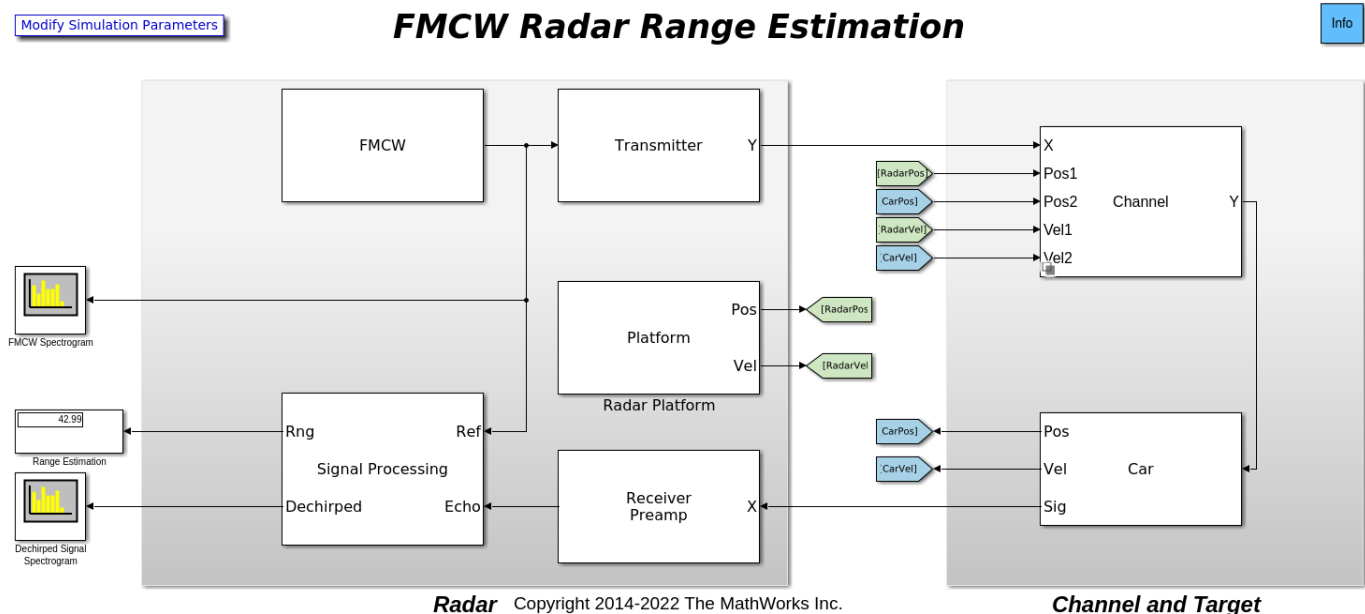
The figure shows the signal flow in the model. The Simulink blocks that make up the model are divided into two major sections, the **Radar** section and the **Channel and Target** section. The shaded

block on the left represents the radar system. In this section, the FMCW signal is generated and transmitted. This section also contains the receiver that captures the radar echo and performs a series of operations, such as dechirping and pulse integration, to estimate the target range. The shaded block on the right models the propagation of the signal through space and its reflection from the car. The output of the system, the estimated range in meters, is shown in the display block on the left.

Radar

The radar system consists of a co-located transmitter and receiver mounted on a vehicle moving along a straight road. It contains the signal processing components needed to extract the information from the returned target echo.

* `FMCW` - Creates an FMCW signal. The FMCW waveform is a common choice in automotive radar, because it provides a way to estimate the range using a continuous wave (CW) radar. The distance is proportional to the frequency offset between the transmitted signal and the received echo. The signal sweeps a bandwidth of 150 MHz.
* `Transmitter` - Transmits the waveform. The operating frequency of the transmitter is 77 GHz.
* `Receiver Preamp` - Receives the target echo and adds the receiver noise.
* `Radar Platform` - Simulates the radar vehicle trajectory.
* `Signal Processing` - Processes the received signal and estimates the range of the target vehicle.

Within the **Radar**, the target echo goes through several signal processing steps before the target range can be estimated. The signal processing subsystem consists of two high-level processing stages.

* Stage 1: The first stage dechirps the received signal by multiplying it with the transmitted signal. This operation produces a beat frequency between the target echo and the transmitted signal. The target range is proportional to the beat frequency. This operation also reduces the bandwidth required to process the signal. Next, 64 sweeps are buffered to form a datacube. The datacube dimensions are fast-time versus slow-time. This datacube is then passed to a `Matrix Sum` block, where the slow-time samples are integrated to boost the signal-to-noise ratio. The data is then passed to the `Range Response` block, which performs an FFT operation to convert the beat frequency to range. Radar signal processing lends itself well to parallelization, so the radar data is then partitioned in range into 5 parts prior to further processing.
* Stage 2: The second stage consists of 5 parallel processing chains for the detection and estimation of the target.
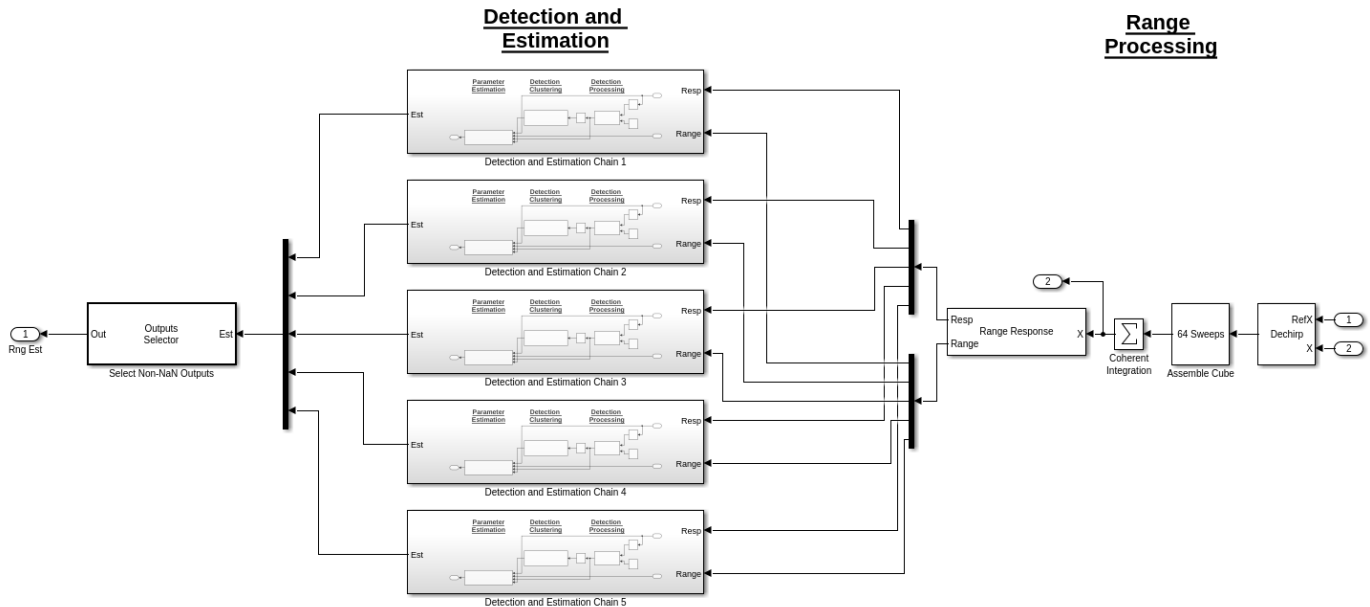
Within Stage 2, each `Detection and Estimation Chain` block consists of 3 processing steps.

- Detection Processing: The radar data is first passed to a 1-dimensional cell-averaging (CA) constant false alarm rate (CFAR) detector that operates in the range dimension. This block identifies detections or hits.

- Detection Clustering: The detections are then passed to the next step where they are aggregated into clusters using the Density-Based Spatial Clustering of Applications with Noise algorithm in the `DBSCAN Clusterer` block. The clustering block clusters the detections in range using the detections identified by the `CA CFAR` block.

- Parameter Estimation: After detections and clusters are identified, the last step is the `Range Estimator` block. This step estimates the range of the detected targets in the radar data.
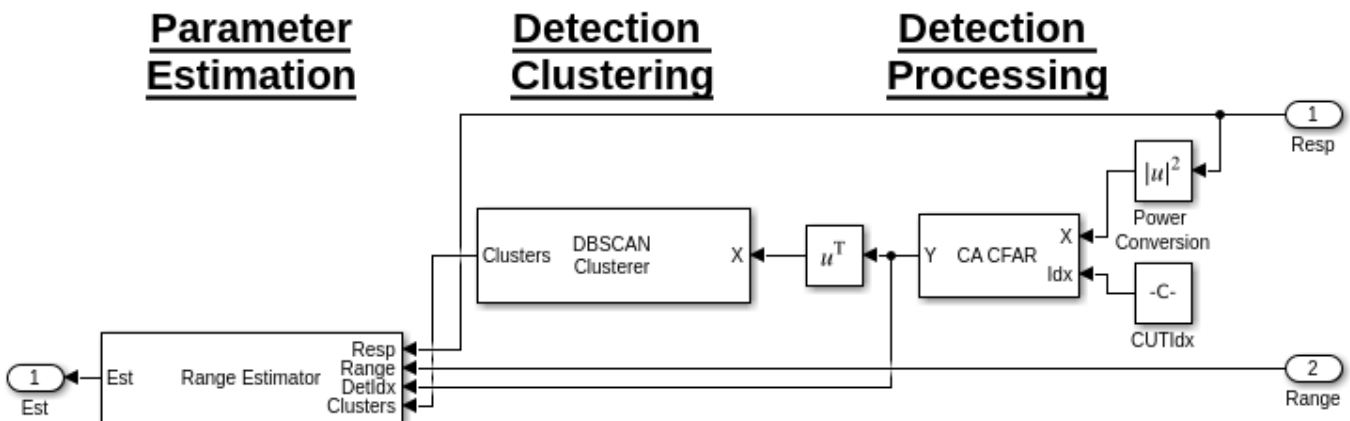
Channel and Target

The **Channel and Target** part of the model simulates the signal propagation and reflection off the target vehicle.

- Channel - Simulates the signal propagation between the radar vehicle and the target vehicle. The channel can be set as either a line-of-sight free space channel or a two-ray channel where the signal arrives at the receiver via both the direct path and the reflected path off the ground. The default choice is a free space channel.
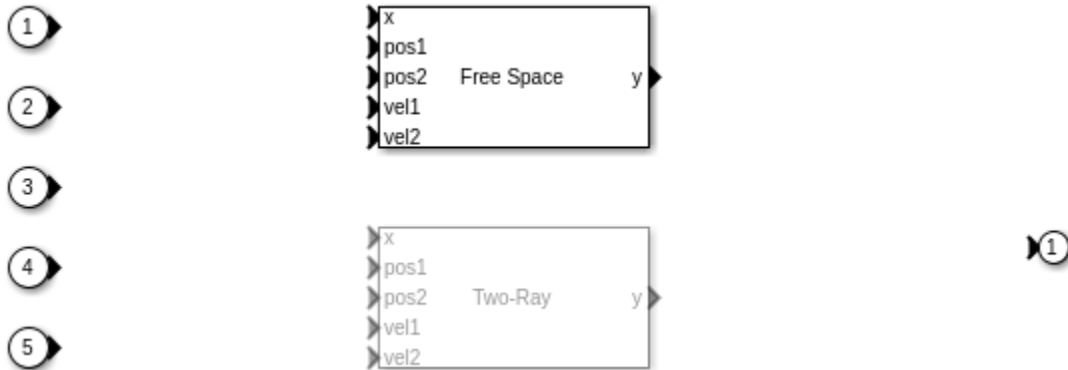
- Car - Reflects the incident signal and simulates the target vehicle trajectory. The subsystem, shown below, consist of two parts: a target model to simulate the echo and a platform model to simulate the dynamics of the target vehicle.
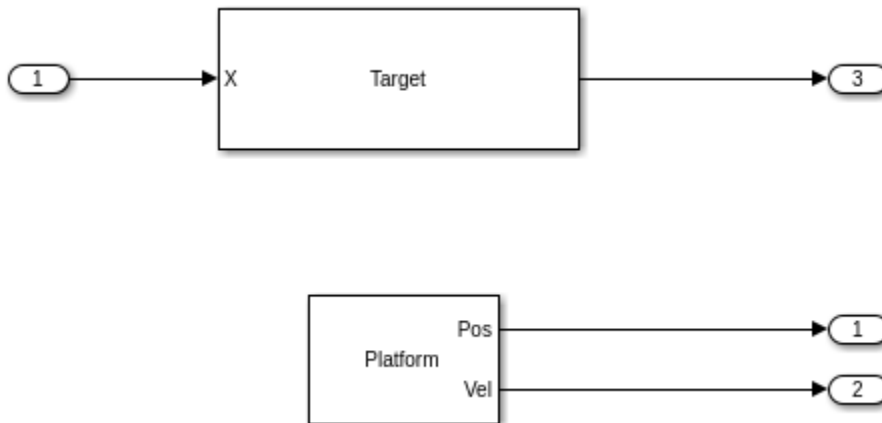
In the Car subsystem, the target vehicle is modeled as a point target with a specified radar cross section. The radar cross section is used to measure how much power can be reflected from a target.

In this model's scenario, the radar vehicle starts at the origin, traveling at 100 km/h (27.8 m/s), while the target vehicle starts at 43 meters in front of the radar vehicle, traveling at 96 km/h (26.7 m/s). The positions and velocities of both the radar and the target vehicles are used in the propagation channel to calculate the delay, Doppler, and signal loss.

Exploring the Model

Several dialog parameters of the model are calculated by the helper function helperslexFMCWParam. To open the function from the model, click on Modify Simulation Parameters block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the

structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

The spectrogram of the FMCW signal below shows that the signal linearly sweeps a span of 150 MHz approximately every 7 microseconds. This waveform provides a resolution of approximately 1 meter.
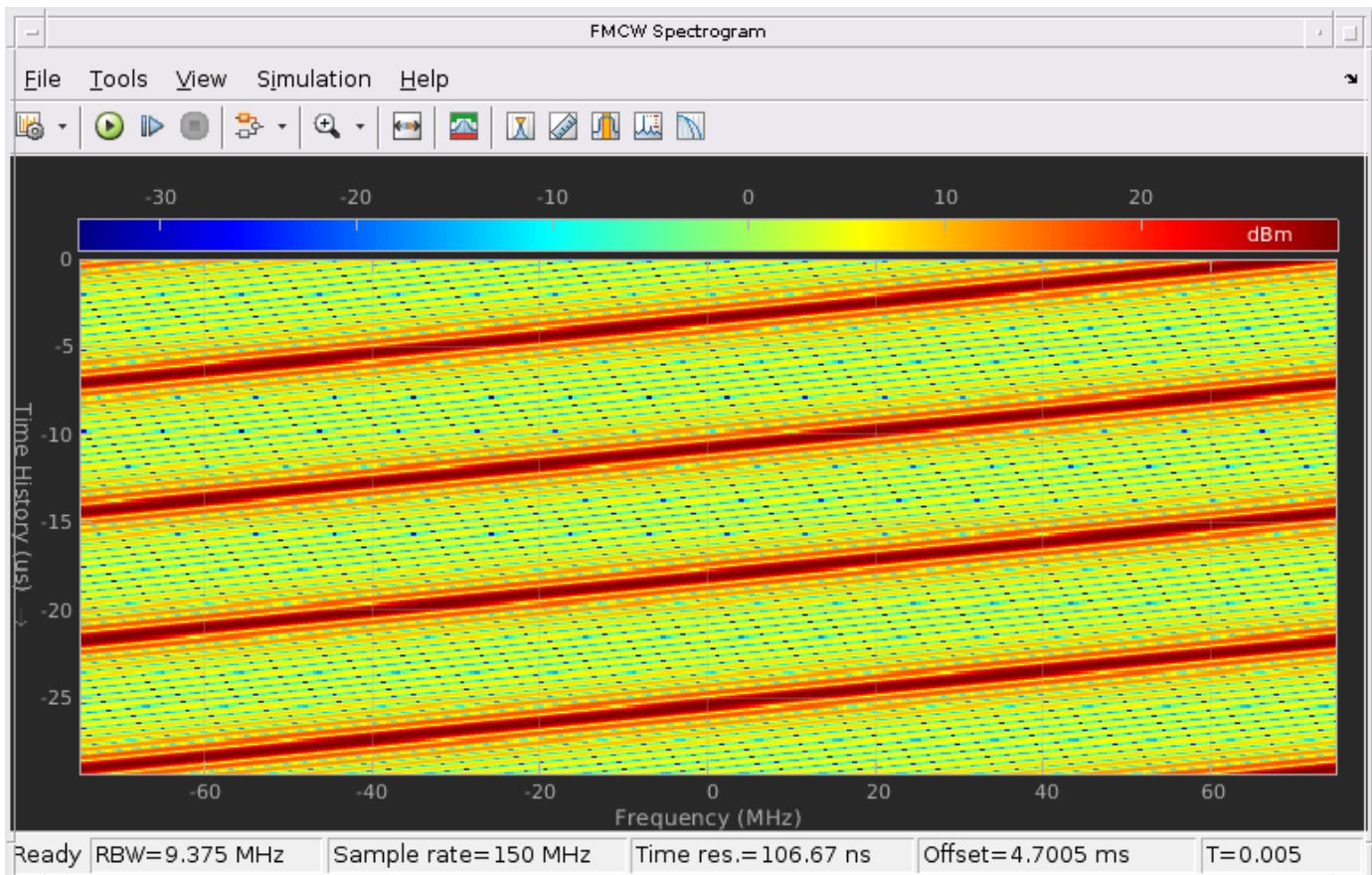
The spectrum of the dechirped signal is shown below. The figure indicates that the beat frequency introduced by the target is approximately 100 kHz. Note that after dechirp, the signal has only a single frequency component. The resulting range estimate calculated from this beat frequency, as displayed in the overall model above, is well within the 1 meter range resolution.
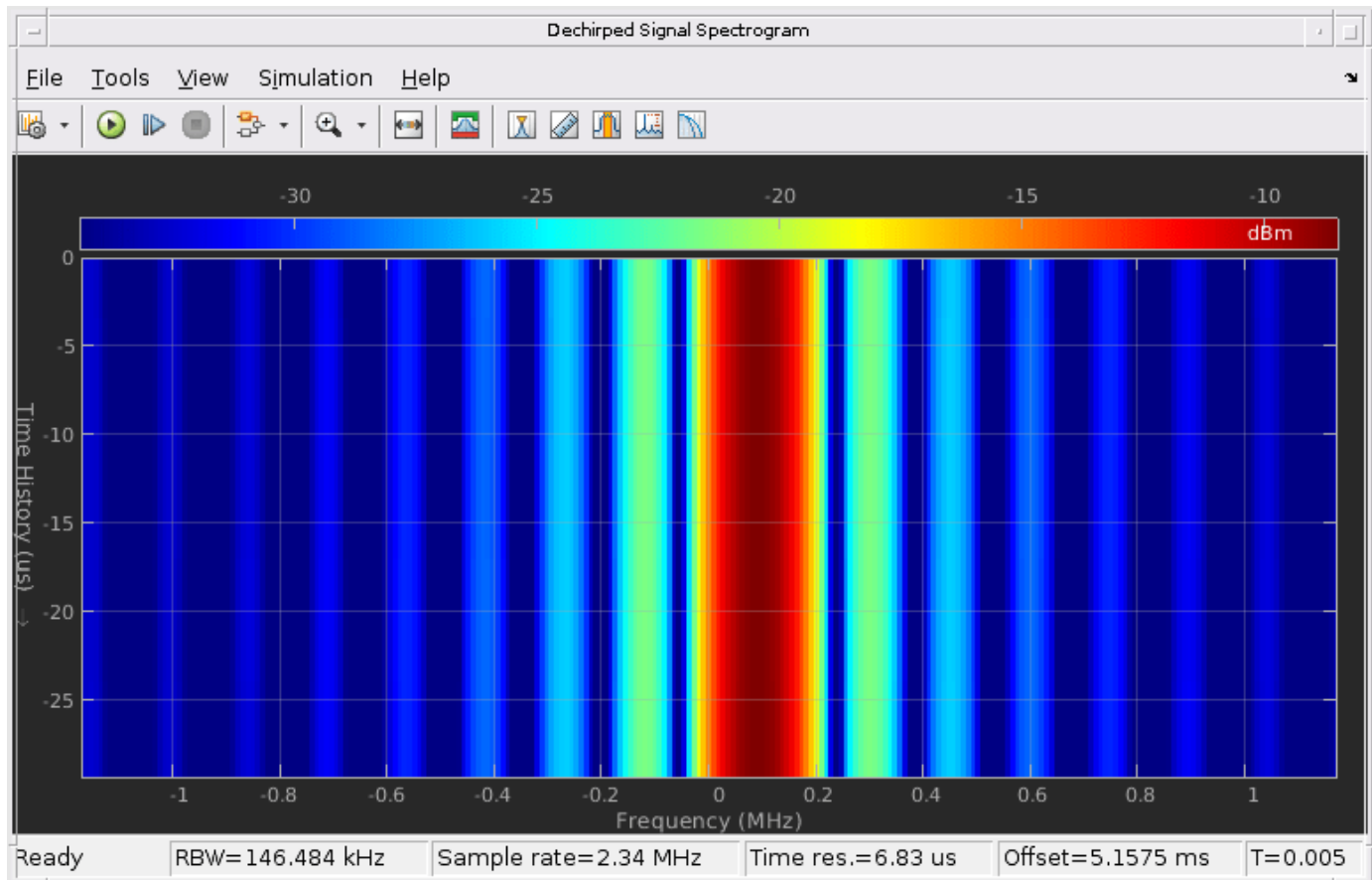
However, this result is obtained with the free space propagation channel. In reality, the propagation between vehicles often involves multiple paths between the transmitter and the receiver. Therefore, signals from different paths may add either constructively or destructively at the receiver. The following section sets the propagation to a two-ray channel, which is the simplest multipath channel.
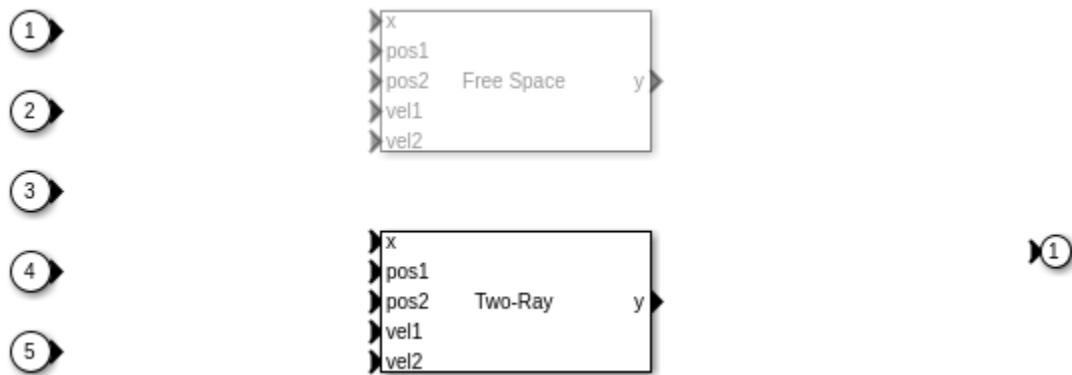
Run the simulation and observe the spectrum of the dechirped signal.
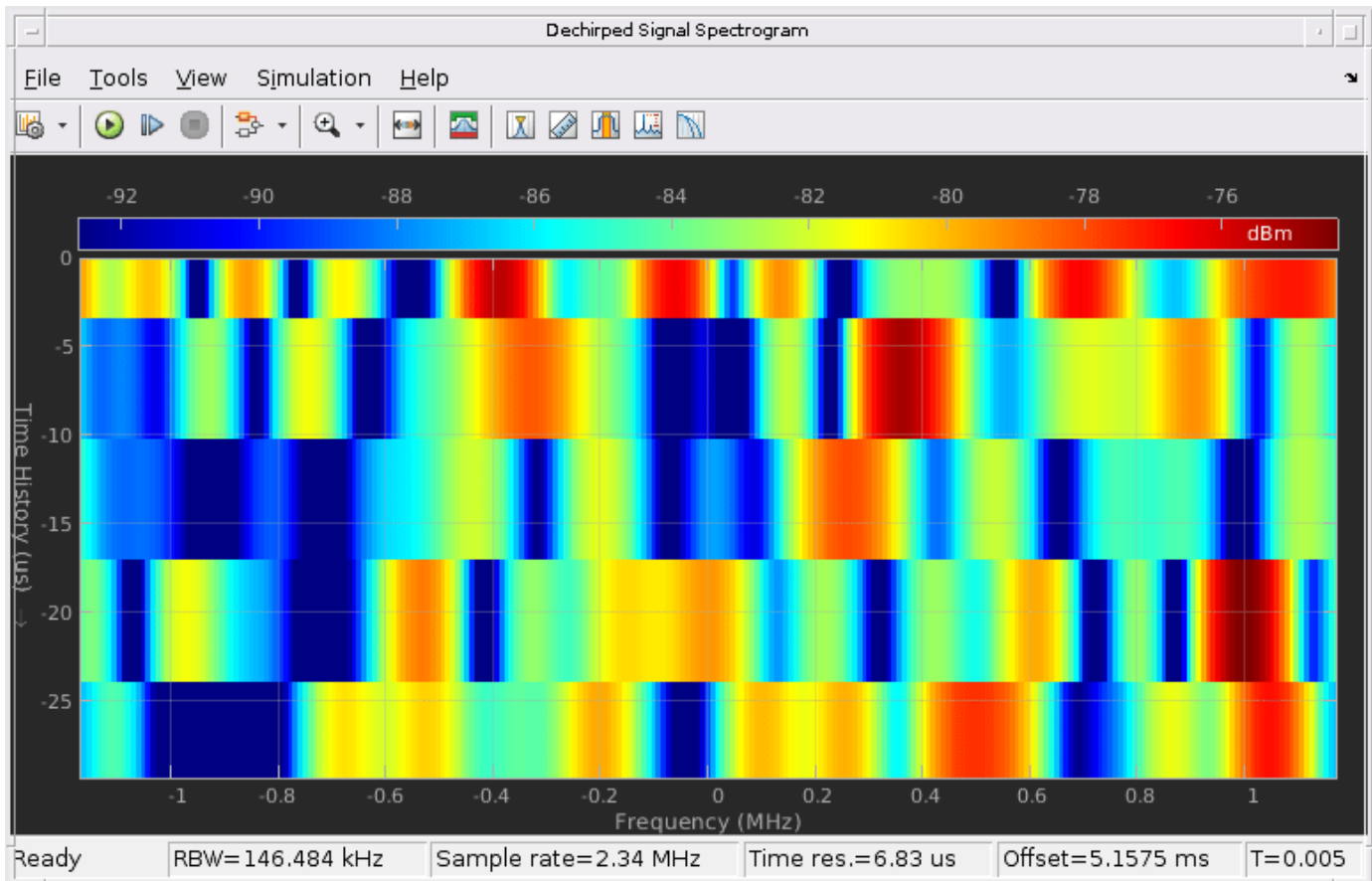
Note that there is no longer a dominant beat frequency, because at this range, the signal from the direct path and the reflected path combine destructively, thereby canceling each other out. This can also be seen from the estimated range, which no longer matches the ground truth.

FMCW Radar Range and Speed Estimation of Multiple Targets

The example model below shows a similar end-to-end FMCW radar system that simulates 2 targets. This example estimates both the range and the speed of the detected targets.

Modify Simulation Parameters *FMCW Radar Multiple Targets Range and Speed Estimation* **Info**

Radar   Copyright 2014-2022 The MathWorks Inc.   *Channel and Target*
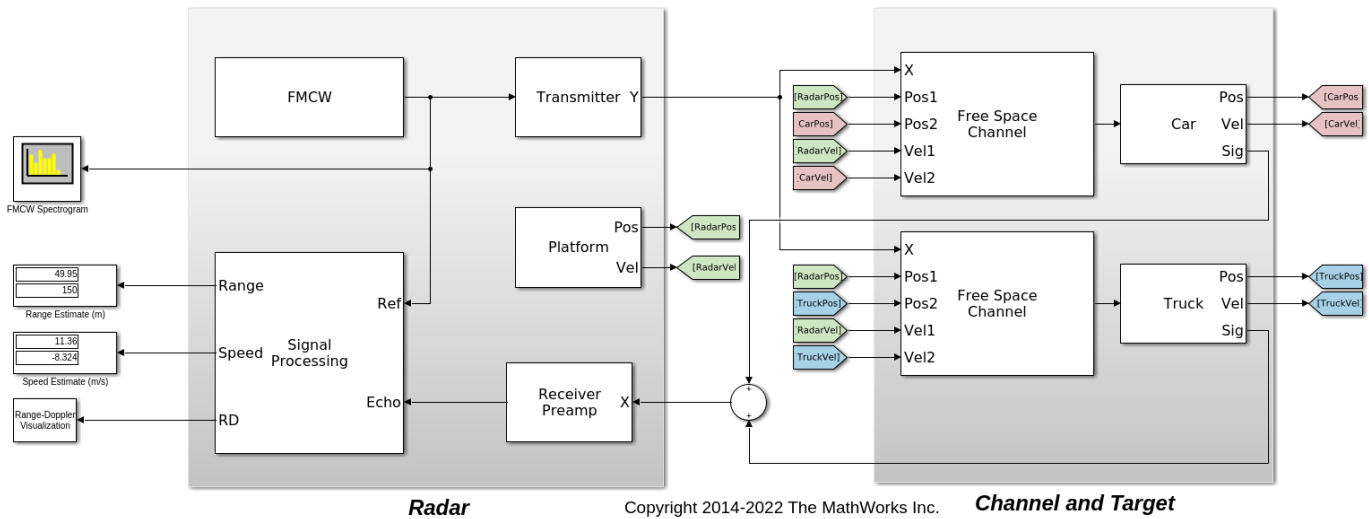
The model is essentially the same as the previous example with 4 primary differences. This model:

- contains two targets,
- uses range-Doppler joint processing, which occurs in the `Range-Doppler Response` block,
- processes only a subset of the data in range rather than the whole datacube in multiple chains, and
- performs detection using a 2-dimensional CA CFAR.

Radar

This model uses range-Doppler joint processing in the signal processing subsystem. Joint processing in the range-Doppler domain makes it possible to estimate the Doppler across multiple sweeps and then to use that information to resolve the range-Doppler coupling, resulting in better range estimates.

The signal processing subsystem is shown in detail below.
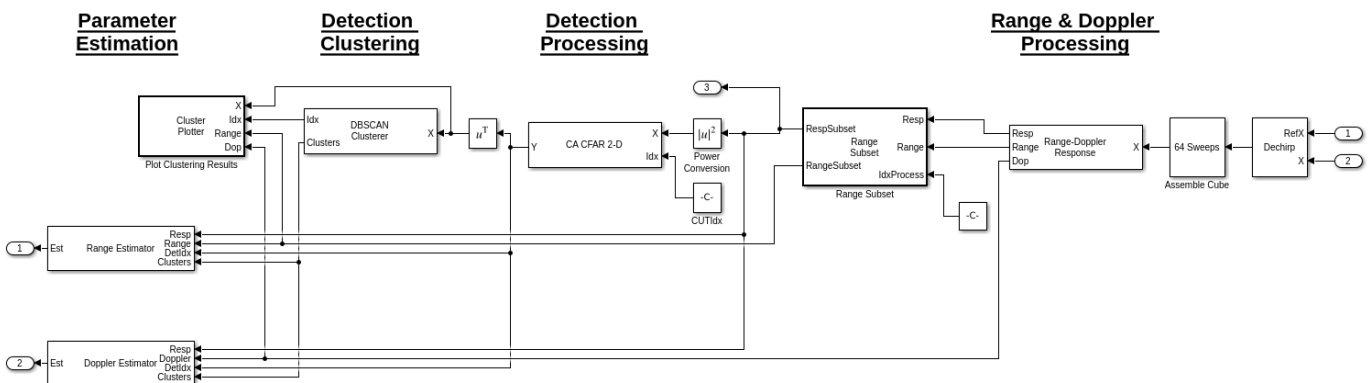
The stages that make up the signal processing subsystem are similar to the prior example. Each stage performs the following actions.

- Stage 1: The first stage again performs dechirping and assembly of a datacube with 64 sweeps. The datacube is then passed to the `Range-Doppler Response` block to compute the range-Doppler map of the input signal. The datacube is then passed to the `Range Subset` block, which obtains a subset of the datacube that will undergo further processing.
- Stage 2: The second stage is where the detection processing occurs. The detector in this example is the `CA CFAR 2-D` block that operates in both the range and Doppler dimensions.
- Stage 3: Clustering occurs in the `DBSCAN Clusterer` block using both the range and Doppler dimensions. Clustering results are then displayed by the `Plot Clusters` block.
- Stage 4: The fourth and final stage estimates the range and speed of the targets from the range-Doppler map using the `Range Estimator` and `Doppler Estimator` blocks, respectively.

As mentioned in the beginning of the example, FMCW radar uses a frequency shift to derive the range of the target. However, the motion of the target can also introduce a frequency shift due to the Doppler effect. Therefore, the beat frequency has both range and speed information coupled. Processing range and Doppler at the same time lets us remove this ambiguity. As long as the sweep is fast enough so that the target remains in the same range gate for several sweeps, the Doppler can be calculated across multiple sweeps and then be used to correct the initial range estimates.

Channel and Target

There are now two target vehicles in the scene, labeled as Car and Truck, and each vehicle has an associated propagation channel. The Car starts 50 meters in front of the radar vehicle and travels at a speed of 60 km/h (16.7 m/s). The Truck starts at 150 meters in front of the radar vehicle and travels at a speed of 130 km/h (36.1 m/s).

Exploring the Model

Several dialog parameters of the model are calculated by the helper function helperslexFMCWMultiTargetsParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

The FMCW signal shown below is the same as in the previous model.
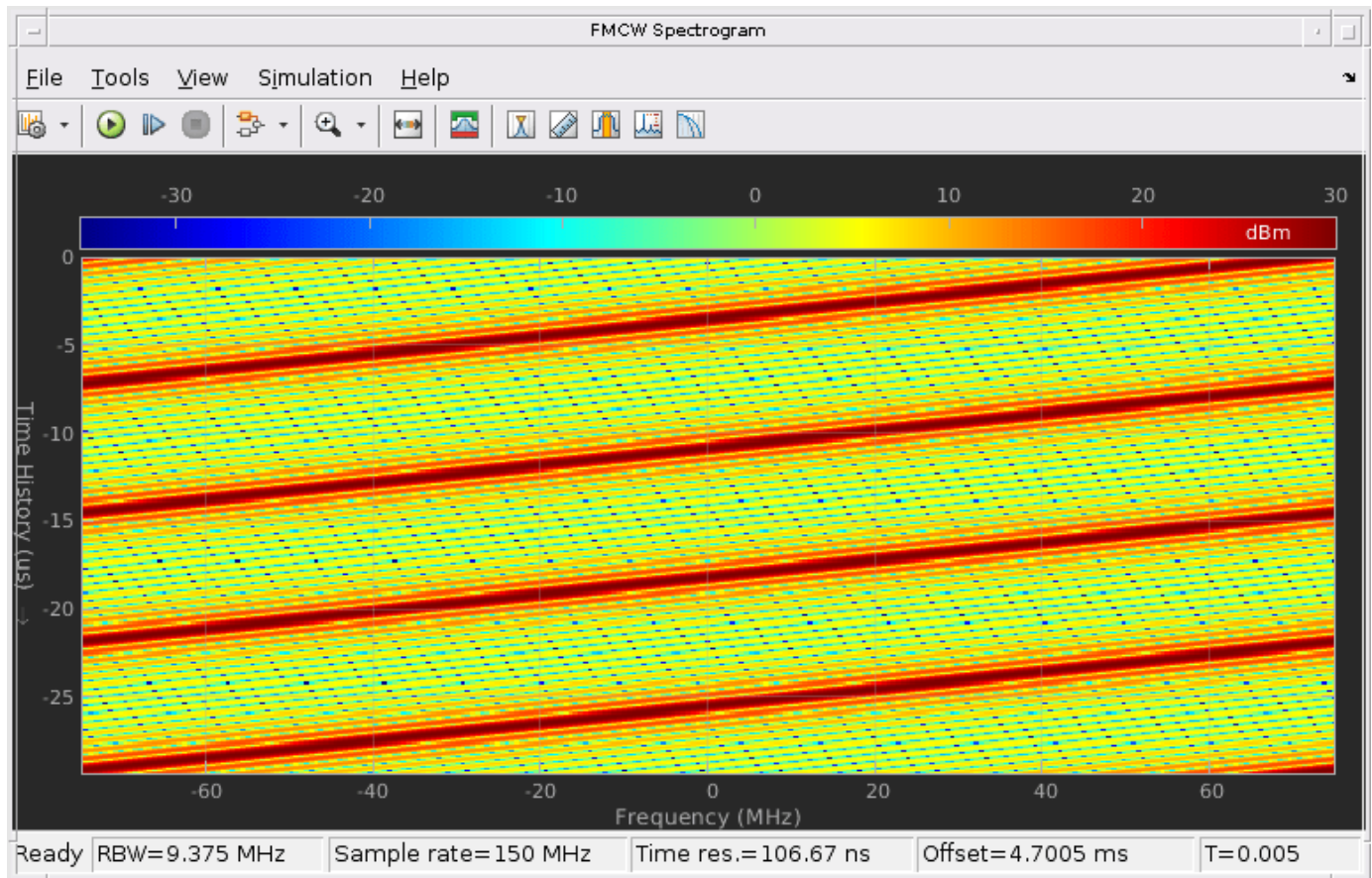
The two targets can be visualized in the range-Doppler map below.
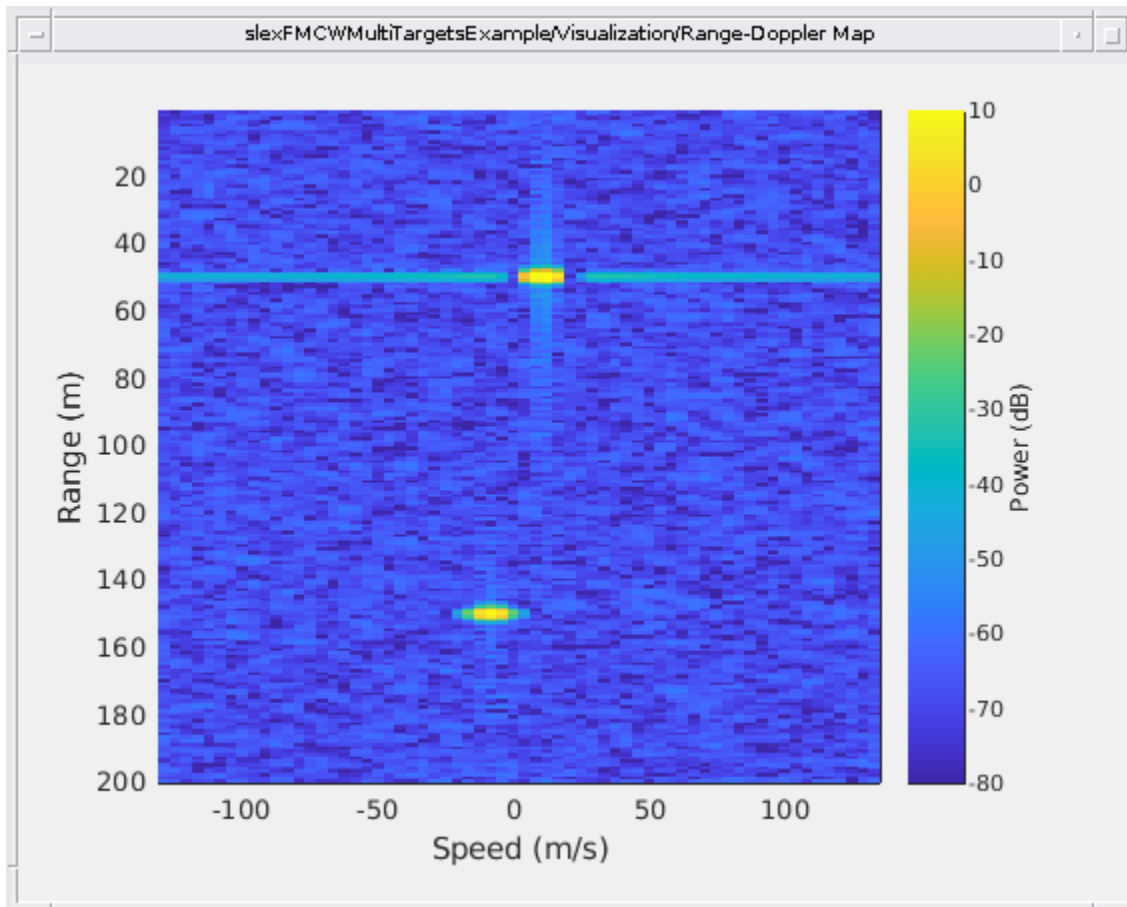
The map correctly shows two targets: one at 50 meters and one at 150 meters. Because the radar can only measure the relative speed, the expected speed values for these two vehicles are 11.1 m/s and -8.3 m/s, respectively, where the negative sign indicates that the Truck is moving away from the radar vehicle. The exact speed estimates may be difficult to infer from the range-Doppler map, but the estimated ranges and speeds are shown numerically in the display blocks in the model on the left. As can be seen, the speed estimates match the expected values well.

MFSK Radar Range and Speed Estimation of Multiple Targets

To be able to do joint range and speed estimation using the above approach, the sweep needs to be fairly fast to ensure the vehicle is approximately stationary during the sweep. This often translates to higher hardware cost. MFSK is a new waveform designed specifically for automotive radar so that it can achieve simultaneous range and speed estimation with longer sweeps.

The example below shows how to use MFSK waveform to perform the range and speed estimation. The scene setup is the same as the previous model.

MFSK Radar Range and Speed Estimation of Multiple Targets

Radar　　Copyright 2014-2022 The MathWorks Inc.　　**Channel and Target**
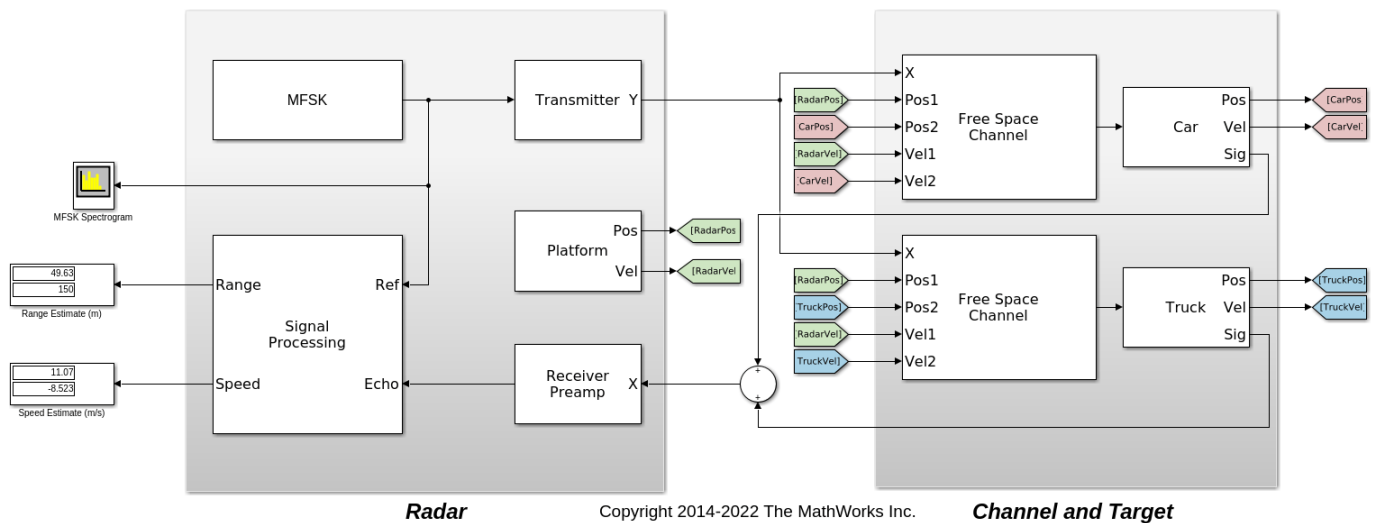
The primary differences between this model and the previous are in the waveform block and the signal processing subsystem. The MFSK waveform essentially consists of two FMCW sweeps with a fixed frequency offset. The sweep in this case happens at discrete steps. From the parameters of the MFSK waveform block, the sweep time can be computed as the product of the step time and the number of steps per sweep. In this example, the sweep time is slightly over 2 ms, which is several orders larger than the 7 microseconds for the FMCW used in the previous model. For more information on the MFSK waveform, see the "Simultaneous Range and Speed Estimation Using MFSK Waveform" example.
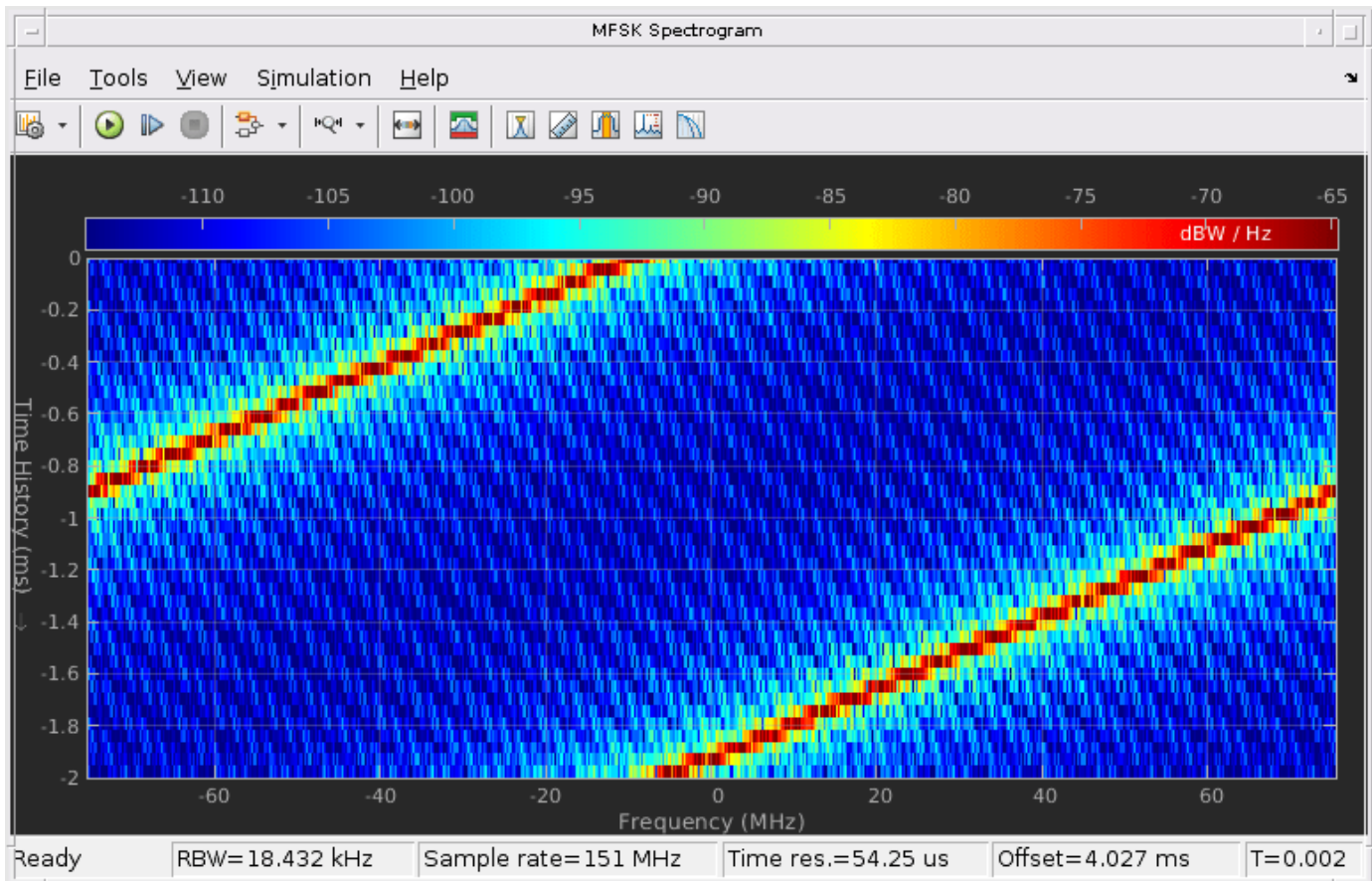
The signal processing subsystem describes how the signal gets processed for the MFSK waveform. The signal is first sampled at the end of each step and then converted to the frequency domain via an FFT. A 1-dimensional CA CFAR detector is used to identify the peaks, which correspond to targets, in the spectrum. Then the frequency at each peak location and the phase difference between the two sweeps are used to estimate the range and speed of the target vehicles.
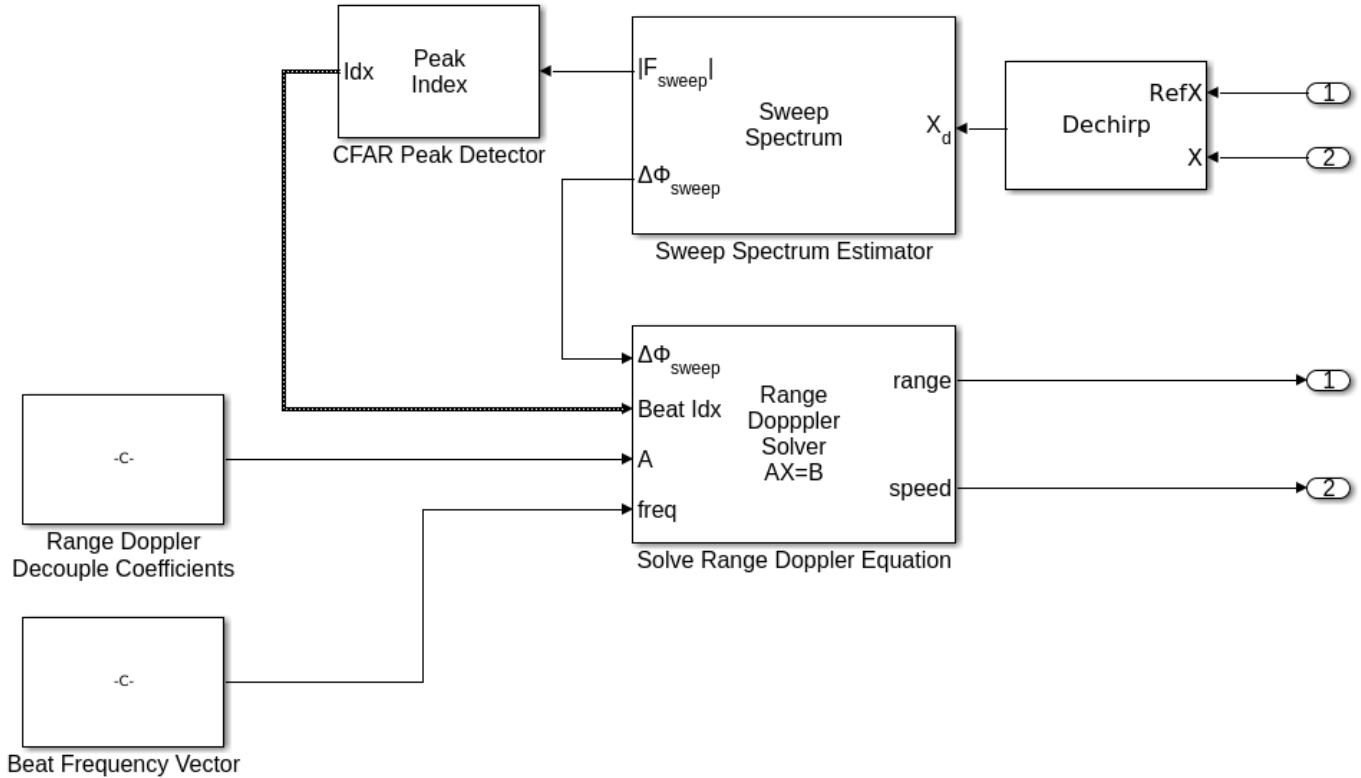
Exploring the Model

Several dialog parameters of the model are calculated by the helper function helperslexMFSKMultiTargetsParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

The estimated results are shown in the model, matching the results obtained from the previous model.

FMCW Radar Range, Speed, and Angle Estimation of Multiple Targets

One can improve the angular resolution of the radar by using an array of antennas. This example shows how to resolve three target vehicles traveling in separate lanes ahead of a vehicle carrying an antenna array.

FMCW Radar Range, Speed, and Angle Estimation
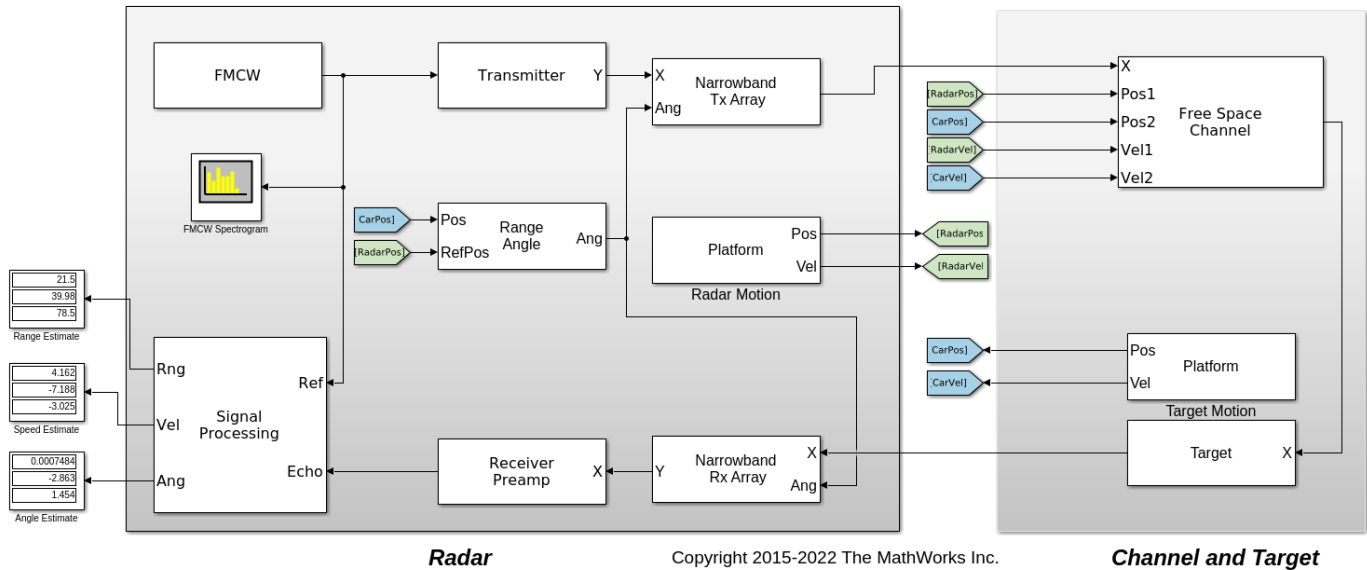
In this scenario, the radar is traveling in the center lane of a highway at 100 km/h (27.8 m/s). The first target vehicle is traveling 20 meters ahead in the same lane as the radar at 85 km/h (23.6 m/s). The second target vehicle is traveling at 125 km/h (34.7 m/s) in the right lane and is 40 meters ahead. The third target vehicle is traveling at 110 km/h (30.6 m/s) in the left lane and is 80 meters ahead. The antenna array of the radar vehicle is a 4-element uniform linear array (ULA).

The origin of the scenario coordinate system is at the radar vehicle. The ground truth range, speed, and angle of the target vehicles with respect to the radar are

```
              Range (m)     Speed (m/s)      Angle (deg)
    -------------------------------------------------------
    Car 1        20             4.2               0
    Car 2        40.05         -6.9              -2.9
    Car 3        80.03         -2.8               1.4
```

The signal processing subsystem now includes direction of arrival estimation in addition to the range and Doppler processing.
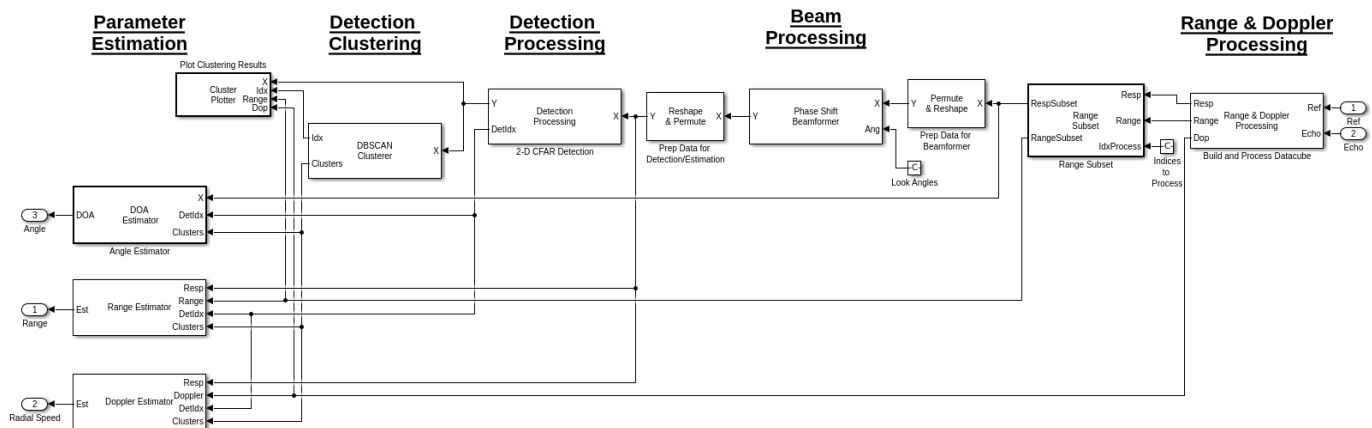
The processing is very similar to the previously discussed FMCW Multiple Target model. However, in this model, there are 5 stages instead of 4.

- Stage 1: Similar to the previously discussed FMCW Multiple Target model, this stage performs dechirping, datacube formation, and range-Doppler processing. The datacube is then passed to the `Range Subset` block, thereby obtaining the subset of the datacube that will undergo further processing.

- Stage 2: The second stage is the `Phase Shift Beamformer` block where beamforming occurs based on the specified look angles that are defined in the parameter helper function helperslexFMCWMultiTargetsDOAParam.

- Stage 3: The third stage is where the detection processing occurs. The detector in this example is again the `CA CFAR 2-D` block that operates in both the range and Doppler dimensions.

- Stage 4: Clustering occurs in the `DBSCAN Clusterer` block using the range, Doppler, and angle dimensions. Clustering results are then displayed by the `Plot Clusters` block.

- Stage 5: The fourth and final stage estimates the range and speed of the targets from the range-Doppler map using the `Range Estimator` and `Doppler Estimator` blocks, respectively. In addition, direction of arrival (DOA) estimation is performed using a custom block that features an implementation of the Phased Array System Toolbox™ Root MUSIC Estimator.

Exploring the Model

Several dialog parameters of the model are calculated by the helper function helperslexFMCWMultiTargetsDOAParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

The estimated results are shown in the model and match the expected values well.

Summary

The first model shows how to use an FMCW radar to estimate the range of a target vehicle. The information derived from the echo, such as the distance to the target vehicle, are necessary inputs to a complete automotive ACC system.

The example also discusses how to perform combined range-Doppler processing to derive both range and speed information of target vehicles. However, it is worth noting that when the sweep time is long, the system capability for estimating the speed is degraded, and it is possible that the joint processing can no longer provide accurate compensation for range-Doppler coupling. More discussion on this topic can be found in the MATLAB "Automotive Adaptive Cruise Control Using FMCW Technology" on page 1-148 example.

The following model shows how to perform the same range and speed estimation using an MFSK waveform. This waveform can achieve the joint range and speed estimation with longer sweeps, thus reducing the hardware requirements.

The last model is an FMCW radar featuring an antenna array that performs range, speed, and angle estimation.

Increasing Angular Resolution with Virtual Arrays

This example introduces how forming a virtual array in MIMO radars can help increase angular resolution. It shows how to simulate a coherent MIMO radar signal processing chain using Phased Array System Toolbox™.

Introduction

There are two categories of multiple input multiple output (MIMO) radars. Multistatic radars form the first category. They are often referred to as statistical MIMO radars. Coherent MIMO radars form the second category and are the focus of this example. A benefit of coherent MIMO radar signal processing is the ability to increase the angular resolution of the physical antenna array by forming a virtual array.

Virtual Array

A virtual array can be created by quasi-monostatic MIMO radars, where the transmit and receive arrays are closely located. To better understand the virtual array concept, first look at the two-way pattern of a conventional phased array radar. The two-way pattern of a phased array radar is the product of its transmit array pattern and receive array pattern. For example, consider a 77 GHz millimeter wave radar with a 2-element transmit array and a 4-element receive array.

```
fc = 77e9;
c = 3e8;
lambda = c/fc;
Nt = 2;
Nr = 4;
```

If both arrays have half-wavelength spacing, which are sometimes referred to as full arrays, then the two-way pattern is close to the receive array pattern.

```
dt = lambda/2;
dr = lambda/2;

txarray = phased.ULA(Nt,dt);
rxarray = phased.ULA(Nr,dr);

ang = -90:90;

pattx = pattern(txarray,fc,ang,0,'Type','powerdb');
patrx = pattern(rxarray,fc,ang,0,'Type','powerdb');
pat2way = pattx+patrx;

helperPlotMultipledBPattern(ang,[pat2way pattx patrx],[-30 0],...
    {'Two-way Pattern','Tx Pattern','Rx Pattern'},...
    'Patterns of full/full arrays - 2Tx, 4Rx',...
    {'-','--','-.'});
```
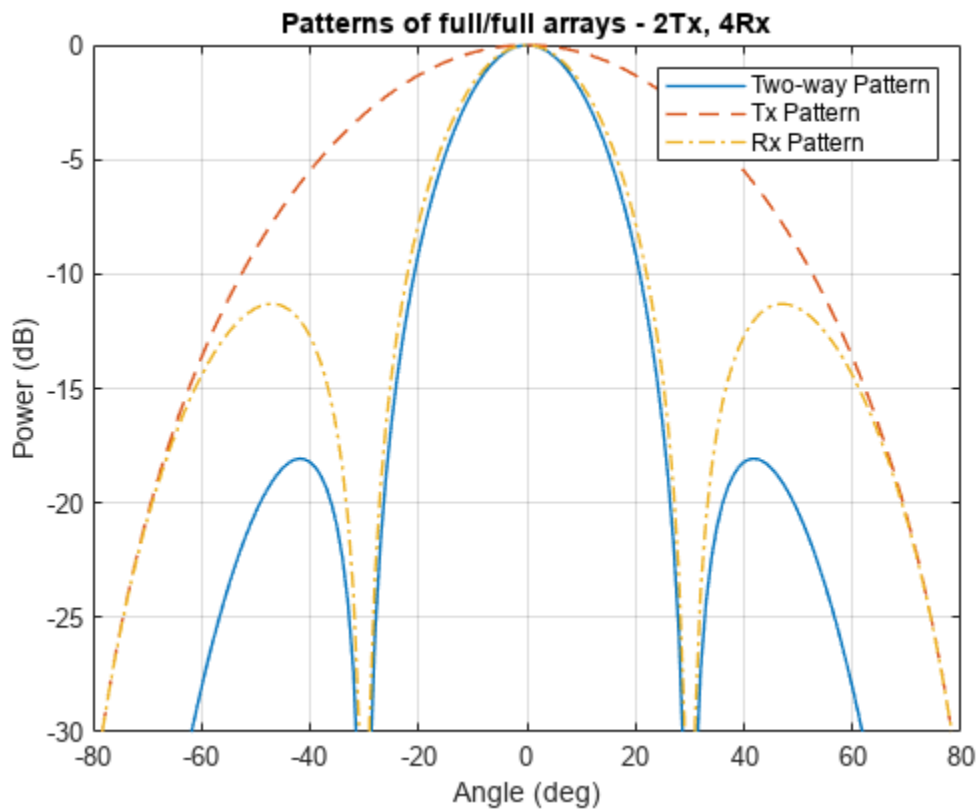
If the full transmit array is replaced with a thin array, meaning the element spacing is wider than half wavelength, then the two-way pattern has a narrower beamwidth. Notice that even though the thin transmit array has grating lobes, those grating lobes are not present in the two-way pattern.

```
dt = Nr*lambda/2;
txarray = phased.ULA(Nt,dt);
pattx = pattern(txarray,fc,ang,0,'Type','powerdb');
pat2way = pattx+patrx;
helperPlotMultipledBPattern(ang,[pat2way pattx patrx],[-30 0],...
    {'Two-way Pattern','Tx Pattern','Rx Pattern'},...
    'Patterns of thin/full arrays - 2Tx, 4Rx',...
    {'-','--','-.'});
```

Patterns of thin/full arrays - 2Tx, 4Rx
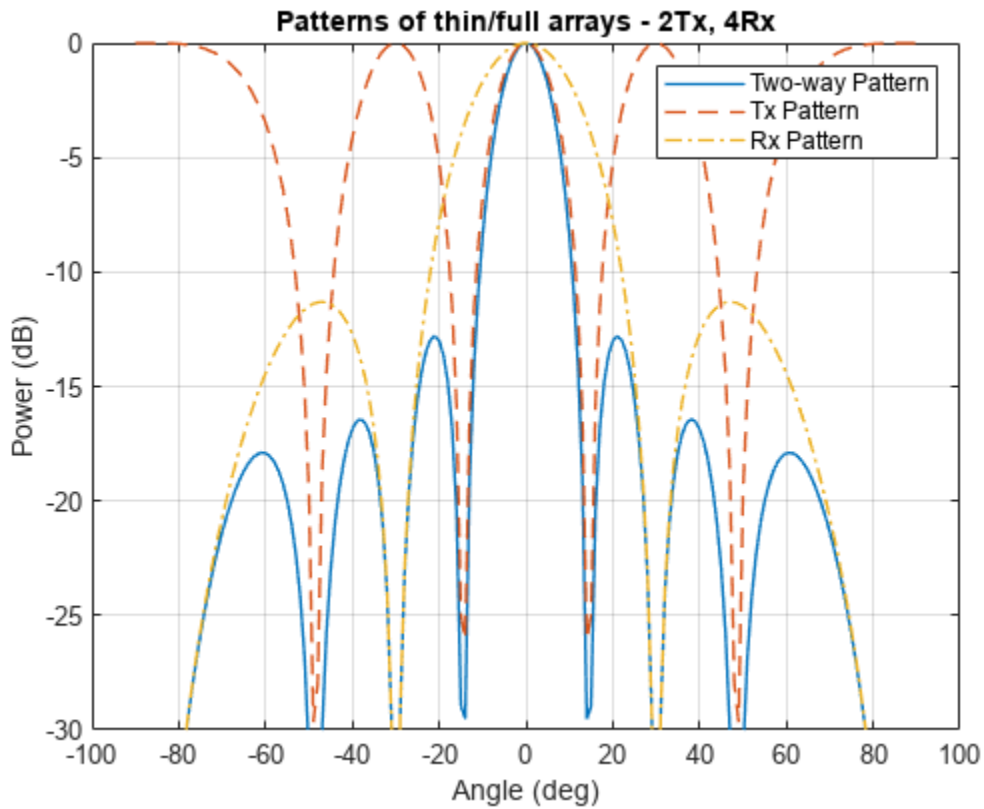
The two-way pattern of this system corresponds to the pattern of a virtual receive array with 2 x 4 = 8 elements. Thus, by carefully choosing the geometry of the transmit and the receive arrays, we can increase the angular resolution of the system without adding more antennas to the arrays.

TX Array 2λ spacing   RX Array 0.5λ spacing   Virtual Array 0.5λ spacing

```
varray = phased.ULA(Nt*Nr,dr);
patv = pattern(varray,fc,ang,0,'Type','powerdb');
helperPlotMultipledBPattern(ang,[pat2way patv],[-30 0],...
    {'Two-way Pattern','Virtual Array Pattern'},...
    'Patterns of thin/full arrays and virtual array',...
    {'-','--'},[1 2]);
```

Patterns of thin/full arrays and virtual array
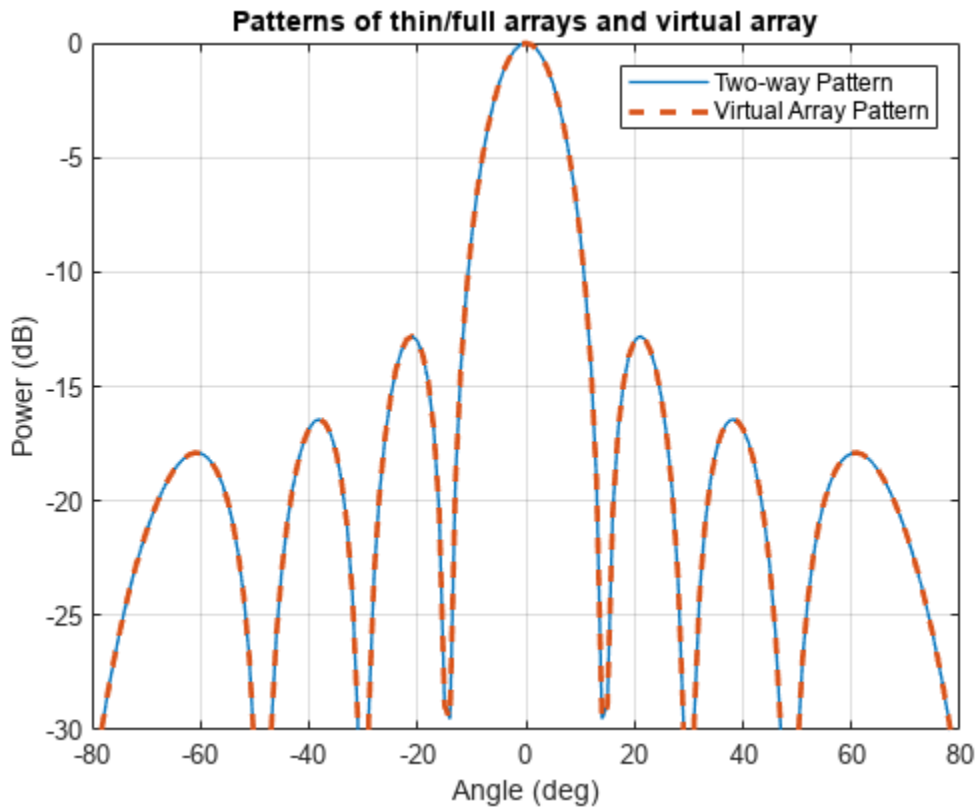
Virtual Array in MIMO Radars

In a coherent MIMO radar system, each antenna of the transmit array transmits an orthogonal waveform. Because of this orthogonality, it is possible to recover the transmitted signals at the receive array. The measurements at the physical receive array corresponding to each orthogonal waveform can then be stacked to form the measurements of the virtual array.
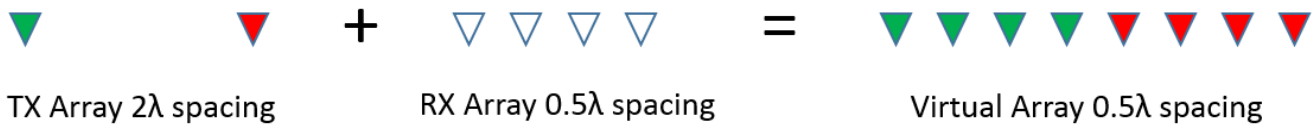
Note that since each element in the transmit array radiates independently, there is no transmit beamforming, so the transmit pattern is broad and covers a large field of view (FOV). This allows the simultaneous illumination of all targets in the FOV. The receive array can then generate multiple beams to process all target echoes. Compared to conventional phased array radars that need successive scans to cover the entire FOV, this is another advantage of MIMO radars for applications that require fast reaction time.

TDM-MIMO Radar Simulation

Time division multiplexing (TDM) is one way to achieve orthogonality among transmit channels. The remainder of this example shows how to model and simulate a TDM-MIMO frequency-modulated

continuous wave (FMCW) automotive radar system. The waveform characteristics are adapted from the "Automotive Adaptive Cruise Control Using FMCW Technology" on page 1-148 example.

```
waveform = helperDesignFMCWWaveform(c,lambda);
fs = waveform.SampleRate;
```

Imagine that there are two cars in the FOV with a separation of 20 degrees. As seen in the earlier array pattern plots of this example, the 3dB beamwidth of a 4-element receive array is around 30 degrees so conventional processing would not be able to separate the two targets in the angular domain. The radar sensor parameters are as follows:

```
transmitter = phased.Transmitter('PeakPower',0.001,'Gain',36);
receiver = phased.ReceiverPreamp('Gain',40,'NoiseFigure',4.5,'SampleRate',fs);

txradiator = phased.Radiator('Sensor',txarray,'OperatingFrequency',fc,...
    'PropagationSpeed',c,'WeightsInputPort',true);

rxcollector = phased.Collector('Sensor',rxarray,'OperatingFrequency',fc,...
    'PropagationSpeed',c);
```

Define the position and motion of the ego vehicle and the two cars in the FOV.

```
radar_speed = 100*1000/3600;      % Ego vehicle speed 100 km/h
radarmotion = phased.Platform('InitialPosition',[0;0;0.5],'Velocity',[radar_speed;0;0]);

car_dist = [40 50];                % Distance between sensor and cars (meters)
car_speed = [-80 96]*1000/3600;    % km/h -> m/s
car_az = [-10 10];
car_rcs = [20 40];
car_pos = [car_dist.*cosd(car_az);car_dist.*sind(car_az);0.5 0.5];

cars = phased.RadarTarget('MeanRCS',car_rcs,'PropagationSpeed',c,'OperatingFrequency',fc);
carmotion = phased.Platform('InitialPosition',car_pos,'Velocity',[car_speed;0 0;0 0]);
```

The propagation model is assumed to be free space.

```
channel = phased.FreeSpace('PropagationSpeed',c,...
    'OperatingFrequency',fc,'SampleRate',fs,'TwoWayPropagation',true);
```

The raw data cube received by the physical array of the TDM MIMO radar can then be simulated as follows:

```
rng(2017);
Nsweep = 64;
Dn = 2;       % Decimation factor
fs = fs/Dn;
xr = complex(zeros(fs*waveform.SweepTime,Nr,Nsweep));

w0 = [0;1];   % weights to enable/disable radiating elements

for m = 1:Nsweep
    % Update radar and target positions
    [radar_pos,radar_vel] = radarmotion(waveform.SweepTime);
    [tgt_pos,tgt_vel] = carmotion(waveform.SweepTime);
    [~,tgt_ang] = rangeangle(tgt_pos,radar_pos);

    % Transmit FMCW waveform
    sig = waveform();
```

```
        txsig = transmitter(sig);

        % Toggle transmit element
        w0 = 1-w0;
        txsig = txradiator(txsig,tgt_ang,w0);

        % Propagate the signal and reflect off the target
        txsig = channel(txsig,radar_pos,tgt_pos,radar_vel,tgt_vel);
        txsig = cars(txsig);

        % Dechirp the received radar return
        rxsig = rxcollector(txsig,tgt_ang);
        rxsig = receiver(rxsig);
        dechirpsig = dechirp(rxsig,sig);

        % Decimate the return to reduce computation requirements
        for n = size(xr,2):-1:1
            xr(:,n,m) = decimate(dechirpsig(:,n),Dn,'FIR');
        end
    end
end
```

Virtual Array Processing for TDM-MIMO Radar

The data cube received by the physical array must be processed to form the virtual array data cube. For the TDM-MIMO radar system used in this example, the measurements corresponding to the two transmit antenna elements can be recovered from two consecutive sweeps by taking every other page of the data cube.

```
xr1 = xr(:,:,1:2:end);
xr2 = xr(:,:,2:2:end);
```

Now the data cube in `xr1` contains the return corresponding to the first transmit antenna element, and the data cube in `xr2` contains the return corresponding to the second transmit antenna element. Hence, the data cube from the virtual array can be formed as:

```
xrv = cat(2,xr1,xr2);
```

Next, perform range-Doppler processing on the virtual data cube. Because the range-Doppler processing is linear, the phase information is preserved. Therefore, the resulting response can be used later to perform further spatial processing on the virtual aperture.

```
nfft_r = 2^nextpow2(size(xrv,1));
nfft_d = 2^nextpow2(size(xrv,3));

rngdop = phased.RangeDopplerResponse('PropagationSpeed',c,...
    'DopplerOutput','Speed','OperatingFrequency',fc,'SampleRate',fs,...
    'RangeMethod','FFT','PRFSource','Property',...
    'RangeWindow','Hann','PRF',1/(Nt*waveform.SweepTime),...
    'SweepSlope',waveform.SweepBandwidth/waveform.SweepTime,...
    'RangeFFTLengthSource','Property','RangeFFTLength',nfft_r,...
    'DopplerFFTLengthSource','Property','DopplerFFTLength',nfft_d,...
    'DopplerWindow','Hann');

[resp,r,sp] = rngdop(xrv);
```

The resulting `resp` is a data cube containing the range-Doppler response for each element in the virtual array. As an illustration, the range-Doppler map for the first element in the virtual array is shown.

```
plotResponse(rngdop,squeeze(xrv(:,1,:)));
```
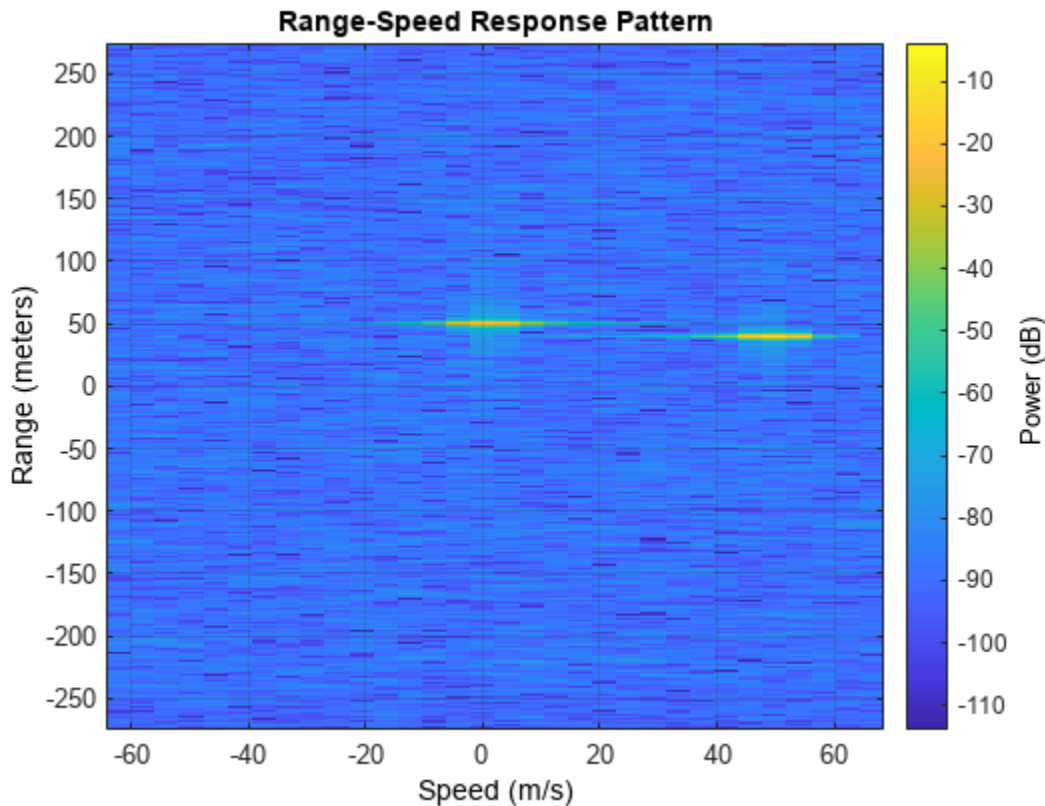
Range-Speed Response Pattern

The detection can be performed on the range-Doppler map from each pair of transmit and receive element to identify the targets in scene. In this example, a simple threshold-based detection is performed on the map obtained between the first transmit element and the first receive element, which corresponds to the measurement at the first element in the virtual array. Based on the range-Doppler map shown in the previous figure, the threshold is set at 10 dB below the maximum peak.

```
respmap = squeeze(mag2db(abs(resp(:,1,:))));
ridx = helperRDDetection(respmap,-10);
```

Based on the detected range of the targets, the corresponding range cuts can be extracted from the virtual array data cube to perform further spatial processing. To verify that the virtual array provides a higher resolution compared to the physical array, the code below extracts the range cuts for both targets and combines them into a single data matrix. The beamscan algorithm is then performed over these virtual array measurements to estimate the directions of the targets.

```
xv = squeeze(sum(resp(ridx,:,:),1))';

doa = phased.BeamscanEstimator('SensorArray',varray,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'DOAOutputPort',true,'NumSignals',2,'ScanAngles',ang);
[Pdoav,target_az_est] = doa(xv);

fprintf('target_az_est = [%s]\n',num2str(target_az_est));

target_az_est = [-6  10]
```

The two targets are successfully separated. The actual angles for the two cars are -10 and 10 degrees.

The next figure compares the spatial spectrums from the virtual and the physical receive array.

```
doarx = phased.BeamscanEstimator('SensorArray',rxarray,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'DOAOutputPort',true,'ScanAngles',ang);
Pdoarx = doarx(xr);

helperPlotMultipledBPattern(ang,mag2db(abs([Pdoav Pdoarx])),[-30 0],...
    {'Virtual Array','Physical Array'},...
    'Spatial spectrum for virtual array and physical array',{'-','--'});
```
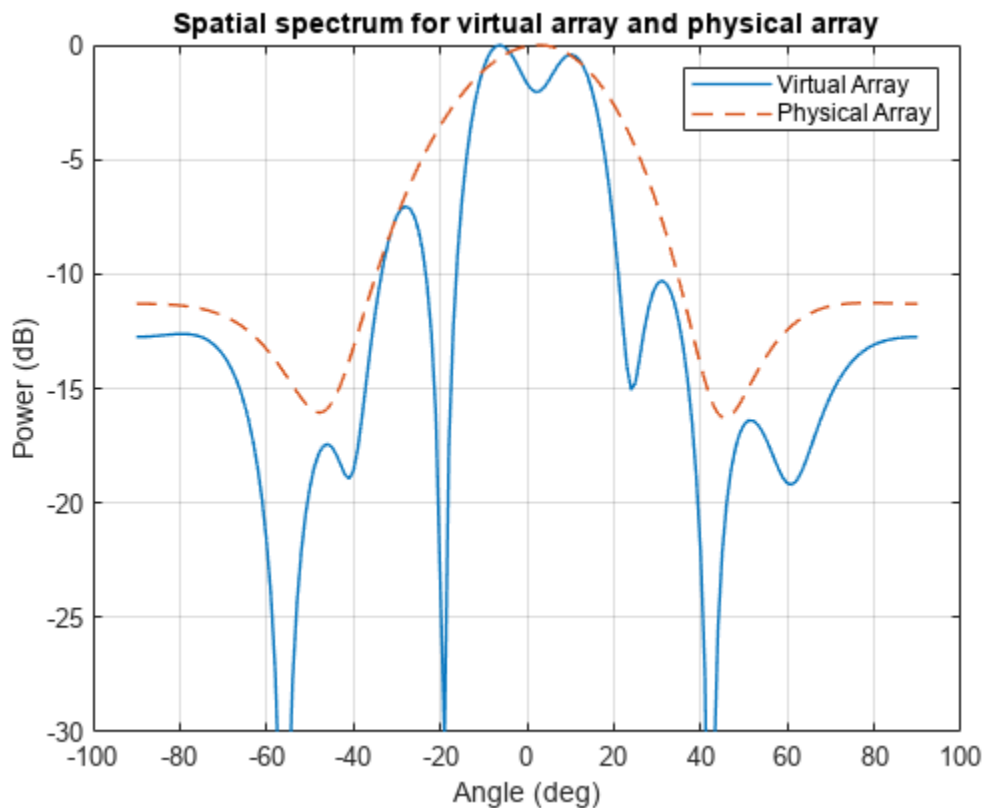

In this example, the detection is performed on the range-Doppler map without spatial processing of the virtual array data cube. This works because the SNR is high. If the SNR is low, it is also possible to process the virtual array blindly across the entire range-Doppler map to maximize the SNR before the detection.

Phase-coded MIMO Radars

Although a TDM-MIMO radar's processing chain is relatively simple, it uses only one transmit antenna at a time. Therefore, it does not take advantage of the full capacity of the transmit array. To improve the efficiency, there are other orthogonal waveforms that can be used in a MIMO radar.

For instance, a MIMO radars can also adopt a phase-coded MIMO waveform. In this case, each radiating element can transmit a uniquely coded waveform, and the receiver can then have a matched filter bank corresponding to each of those phase coded waveform. The signals can then be

recovered and processed to form the virtual array. Using the same system configuration as in previous sections, one phase-coded scheme to achieve orthogonality is to have one element always transmit the same FMCW waveform, while the second transmit element reverses the phase of the FMCW waveform for each sweep. This way both transmit elements are active in all sweeps. For the first sweep, the two elements transmit the same waveform; and for the second sweep, the two elements transmit the waveform with opposite phase, and so on. This is essentially coding the consecutive sweeps from different elements with a Hadamard code. It is similar to the Alamouti codes used in MIMO communication systems.

The raw data cube received by the physical array of the Phase-Coded MIMO radar can then be simulated as follows:

```
rng(2018);
xr = complex(zeros(fs*waveform.SweepTime,Nr,Nsweep));

w =  hadamard(2); % weights to enable/disable radiating elements
for m = 1:Nsweep
    % Update radar and target positions
    [radar_pos,radar_vel] = radarmotion(waveform.SweepTime);
    [tgt_pos,tgt_vel] = carmotion(waveform.SweepTime);
    [~,tgt_ang] = rangeangle(tgt_pos,radar_pos);

    % Transmit FMCW waveform
    sig = waveform();
    txsig = transmitter(sig);

    % Toggle transmit element
    if mod(m,2) == 1
        w0 = w(:,1);
    else
        w0 = w(:,2);
    end

    txsig = txradiator(txsig,tgt_ang,w0);

    % Propagate the signal and reflect off the target
    txsig = channel(txsig,radar_pos,tgt_pos,radar_vel,tgt_vel);
    txsig = cars(txsig);

    % Dechirp the received radar return
    rxsig = rxcollector(txsig,tgt_ang);
    rxsig = receiver(rxsig);
    dechirpsig = dechirp(rxsig,sig);

    % Decimate the return to reduce computation requirements
    for n = size(xr,2):-1:1
        xr(:,n,m) = decimate(dechirpsig(:,n),Dn,'FIR');
    end
end
```

Virtual Array Processing for Phase-coded MIMO Radar

For the Phase-Coded MIMO radar system used in this example, the measurements corresponding to the two sets of phase code can be recovered from two consecutive sweeps by taking every other page of the data cube.

```
Nvsweep = Nsweep/2;
xr1 = xr(:,:,1:2:end);
xr2 = xr(:,:,2:2:end);
```

Next, perform range-Doppler processing on the two data cubes separately.

```
nfft_r = 2^nextpow2(size(xr1,1));
nfft_d = 2^nextpow2(size(xr1,3));
```

```
rngdop = phased.RangeDopplerResponse('PropagationSpeed',c,...
    'DopplerOutput','Speed','OperatingFrequency',fc,'SampleRate',fs,...
    'RangeMethod','FFT','PRFSource','Property',...
    'RangeWindow','Hann','PRF',1/(Nt*waveform.SweepTime),...
    'SweepSlope',waveform.SweepBandwidth/waveform.SweepTime,...
    'RangeFFTLengthSource','Property','RangeFFTLength',nfft_r,...
    'DopplerFFTLengthSource','Property','DopplerFFTLength',nfft_d,...
    'DopplerWindow','Hann');
```

As an illustration, the range-Doppler map of Phase-Coded MIMO radar is same as that of TDM MIMO radar.

```
plotResponse(rngdop,squeeze(xr1(:,1,:)));
```
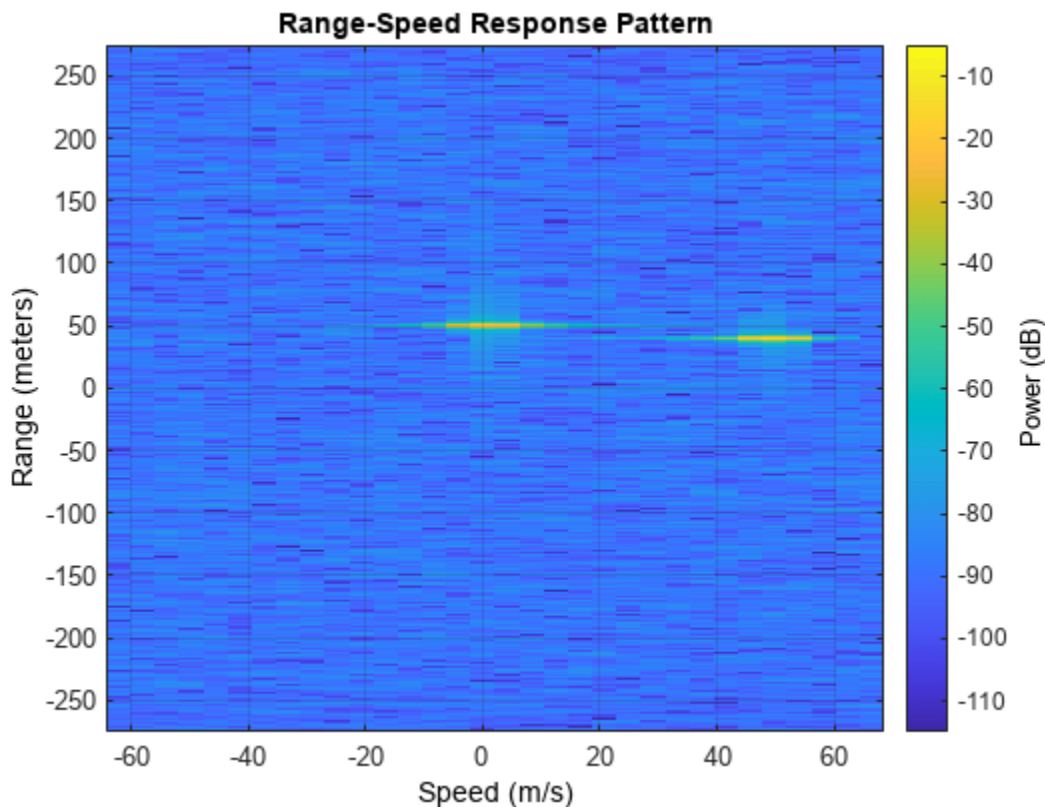

For a phase-coded MIMO radar, the virtual array can no longer be formed by simply cascading the signal from the two adjacent pulses together. Instead, before castcading, the phase code embedded in pulses has to be corrected. This example uses a Hadamard code based approach, thus the process to remove the embedded phases is referred to as Hadamard decode process. After the Hadamard

decode processing, range-Doppler responses of different transmitters are cascaded. The resulting `respvr` is a data cube containing the range-Doppler response for physical arrays from different transmit antenna elements.

```
resp_DDM(:,:,:,1) = rngdop(xr1);
[resp_DDM(:,:,:,2),r,sp] = rngdop(xr2);
respvr(:,:,:,1) = helperHadamardDecode(resp_DDM,w(:,1));
respvr(:,:,:,2) = helperHadamardDecode(resp_DDM,w(:,2));
% for i = 1:2
%     H = w(:,i);
%     respvr(:,:,:,i) = hadamard_decode(resp_DDM,H);
% end
```

Hence, the data cube from the virtual array can be formed as:

```
resp_vr = cat(2,respvr(:,:,:,1),respvr(:,:,:,2));
```

The detection can be performed on the range-Doppler map from each pair of transmit and receive element to identify the targets in scene. In this example, a simple threshold-based detection is performed on the map obtained between the first transmit element and the first receive element, which corresponds to the measurement at the first element in the virtual array. Based on the range-Doppler map shown in the previous figure, the threshold is set at 10 dB below the maximum peak.

```
respmap = squeeze(mag2db(abs(resp_vr(:,1,:))));
ridx = helperRDDetection(respmap,-10);
```

Based on the detected range of the targets, the corresponding range cuts can be extracted from the virtual array data cube to perform further spatial processing. To verify that the virtual array provides a higher resolution compared to the physical array, the code below extracts the range cuts for both targets and combines them into a single data matrix. The beamscan algorithm is then performed over these virtual array measurements to estimate the directions of the targets.

```
xv = squeeze(sum(resp_vr(ridx,:,:),1))';

doa = phased.BeamscanEstimator('SensorArray',varray,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'DOAOutputPort',true,'NumSignals',2,'ScanAngles',ang);
[Pdoav,target_az_est] = doa(xv);

fprintf('target_az_est = [%s]\n',num2str(target_az_est));

target_az_est = [10 -11]
```

The two targets are successfully separated. The actual angles for the two cars are -10 and 10 degrees.

The next figure compares the spatial spectrums from the virtual and the physical receive array.

```
doarx = phased.BeamscanEstimator('SensorArray',rxarray,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'DOAOutputPort',true,'ScanAngles',ang);
Pdoarx = doarx(xr);

helperPlotMultipledBPattern(ang,mag2db(abs([Pdoav Pdoarx])),[-30 0],...
    {'Virtual Array','Physical Array'},...
    'Spatial spectrum for virtual array and physical array',{'-','--'});
```

Spatial spectrum for virtual array and physical array
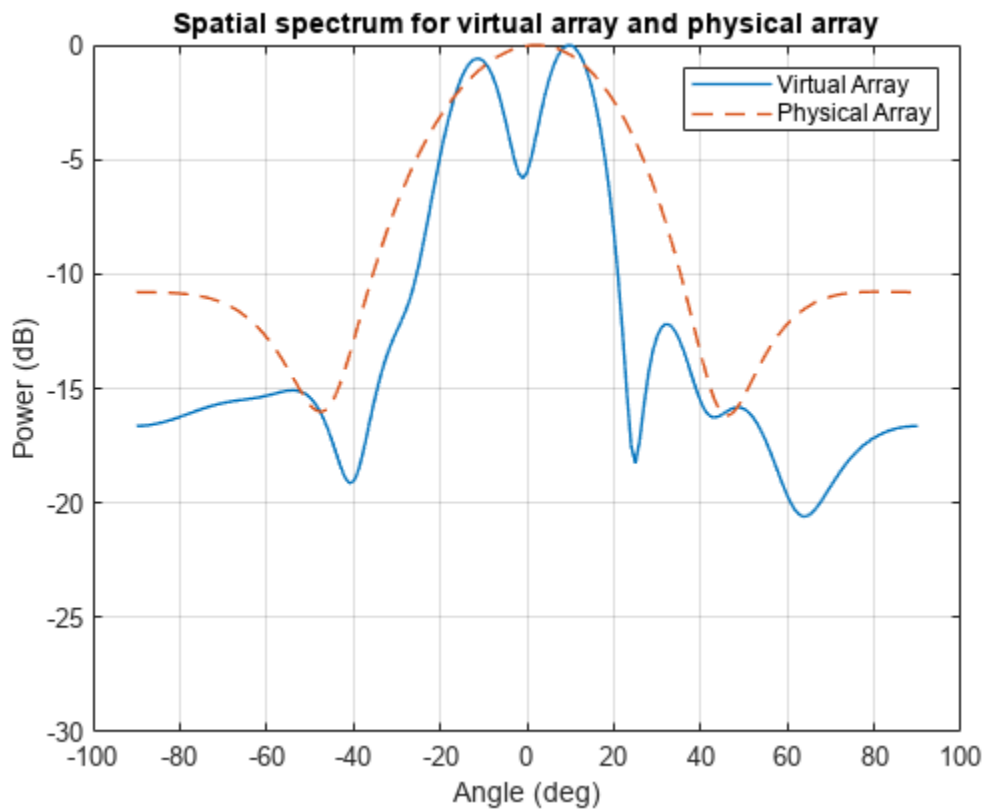
Summary

In this example, we gave a brief introduction to coherent MIMO radar and the virtual array concept. We simulated the return of a MIMO radar with a 2-element transmit array and a 4-element receive array and performed direction of arrival estimation of the simulated echos of two closely spaced targets using an 8-element virtual array.

References

[1] Frank Robey, et al. *MIMO Radar Theory and Experimental Results*, Conference Record of the Thirty Eighth Asilomar Conference on Signals, Systems and Computers, California, pp. 300-304, 2004.

[2] Eli Brookner, *MIMO Radars and Their Conventional Equivalents*, IEEE Radar Conference, 2015.

[3] Sandeep Rao, *MIMO Radar*, Texas Instruments Application Report SWRA554, May 2017.

[4] Jian Li and Peter Stoica, *MIMO Radar Signal Processing*, John Wiley & Sons, 2009.

Patch Antenna Array for FMCW Radar

This example shows how to model a 77 GHz antenna array for frequency-modulated continuous-wave (FMCW) radar applications. The presence of antennas and antenna arrays in and around vehicles has become commonplace with the introduction of wireless collision detection, collision avoidance, and lane departure warning systems. The two frequency bands considered for such systems are centered around 24 GHz and 77 GHz, respectively. In this example, we will investigate the microstrip patch antenna as a phased array radiator. The dielectric substrate is air.

This example requires the Antenna Toolbox™.

Antenna Array Design

The FMCW antenna array is intended for a forward radar system designed to look for and prevent a collision. Therefore, a cosine antenna pattern is an appropriate choice for the initial design since it does not radiate any energy backwards. Assume that the radar system operates at 77 GHz with a 700 MHz bandwidth.

```
fc = 77e9;
fmin = 73e9;
fmax = 80e9;
vp = physconst('lightspeed');
lambda = vp/fc;

cosineantenna = phased.CosineAntennaElement;
cosineantenna.FrequencyRange = [fmin fmax];
pattern(cosineantenna,fc)
```

3D Directivity Pattern
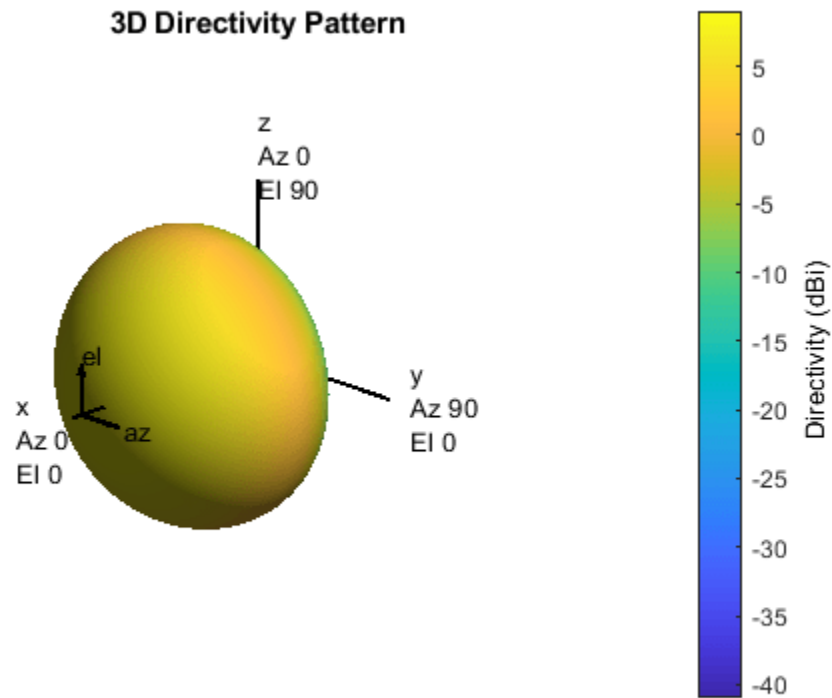
The array itself needs to be mounted on or around the front bumper. The array configuration we investigate is a 2 X 4 rectangular array, similar to what is mentioned in [1]. Such a design has bigger aperture along azimuth direction thus providing better azimuth resolution.

```
Nrow = 2;
Ncol = 4;
fmcwCosineArray = phased.URA;
fmcwCosineArray.Element = cosineantenna;
fmcwCosineArray.Size = [Nrow Ncol];
fmcwCosineArray.ElementSpacing = [0.5*lambda 0.5*lambda];
pattern(fmcwCosineArray,fc)
```

3D Directivity Pattern
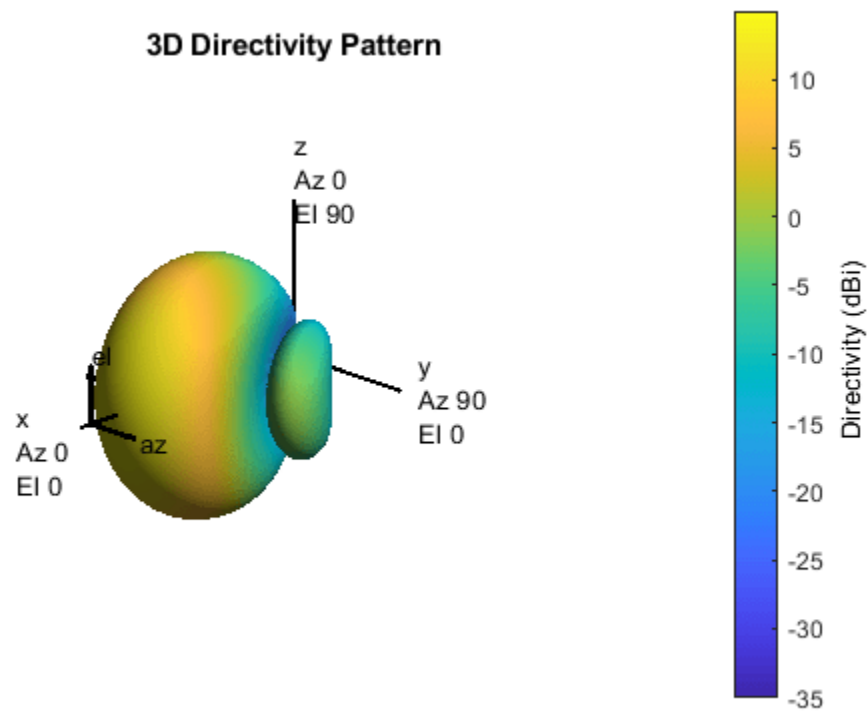
Design Realistic Patch Antenna

The Antenna Toolbox has several antenna elements that could provide hemispherical coverage and resemble a pattern of cosine shape. Choose a patch antenna element with typical radiator dimensions. The patch length is approximately half-wavelength at 77 GHz and the width is 1.5 times the length to improving the bandwidth. The ground plane is λ on each side and the feed offset from center in the direction of the patch length is about a quarter of the length.

```
patchElement = design(patchMicrostrip,fc);
```

Because the default patch antenna geometry has its maximum radiation directed towards zenith, rotate the patch antenna by 90 degrees about the y-axis so that the maximum is along the x-axis.

```
patchElement.Tilt = 90;
patchElement.TiltAxis = [0 1 0];
```

Isolated Patch Antenna 3D Pattern and Resonance

Plot the pattern of the patch antenna at 77 GHz. The patch is a medium gain antenna with the peak directivity around 10 dBi.

```
myFigure = gcf;
myFigure.Color = 'w';
pattern(patchElement,fc)
```
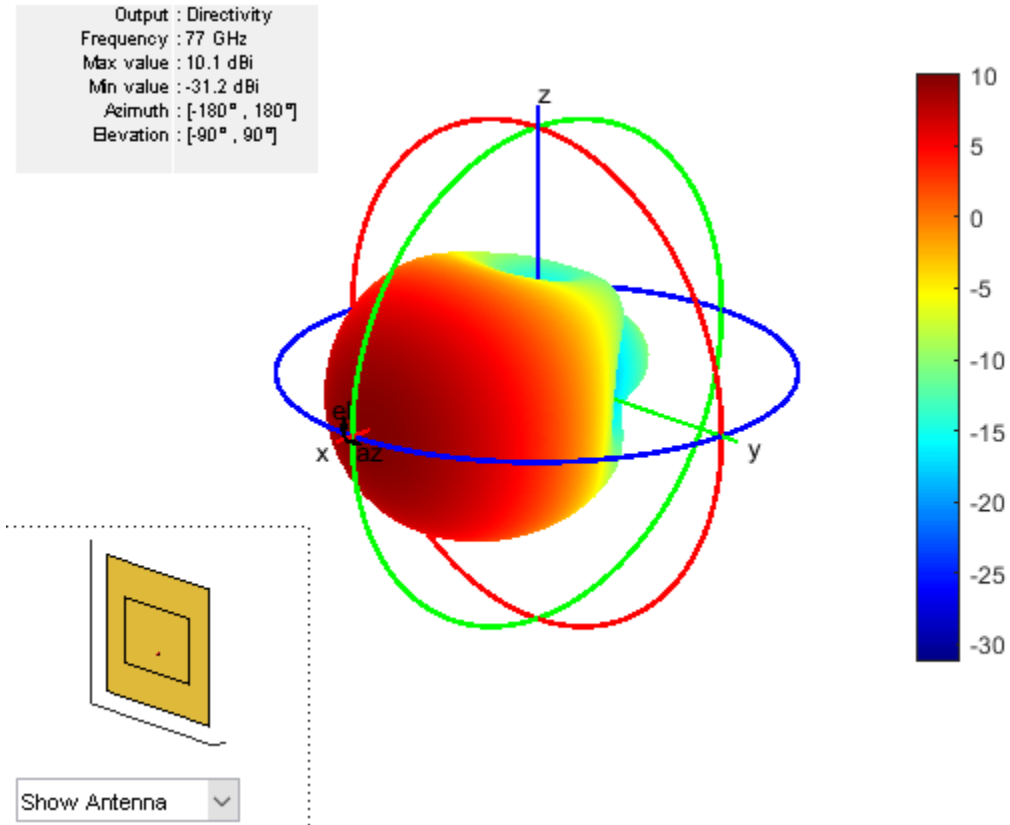
The patch is radiating in the correct mode with a pattern maximum at 0 degrees azimuth and 0 degrees elevation. Since the initial dimensions are approximations, it is important to verify the input impedance behavior.

```
Numfreqs = 21;
freqsweep = unique([linspace(fmin,fmax,Numfreqs) fc]);
impedance(patchElement,freqsweep);
```
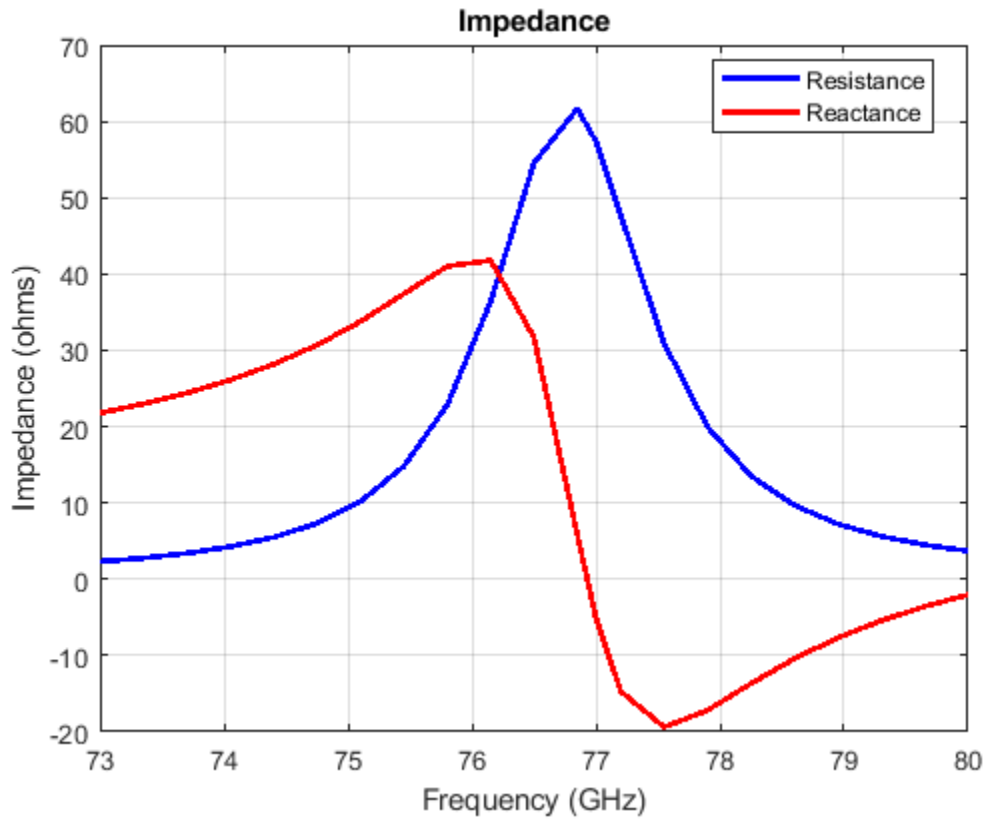
According to the figure, the patch antenna has its first resonance (parallel resonance) at 77 GHz.

Next is to check the reflection coefficient of the patch antenna to confirm a good impedance match. It is typical to consider the value $S_{11} = -10dB$ as a threshold value for determining the antenna bandwidth.

```
s = sparameters(patchElement,freqsweep);
rfplot(s,'m-.')
hold on
line(freqsweep/1e9,ones(1,numel(freqsweep))*-10,'LineWidth',1.5)
hold off
```
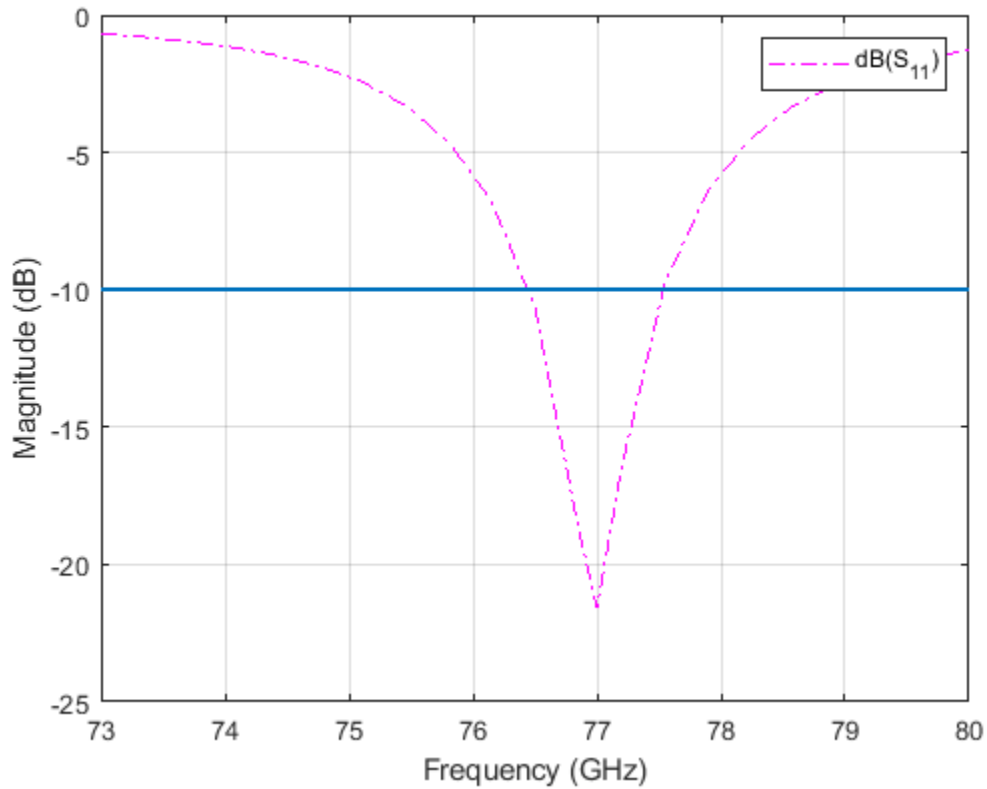
The deep minimum at 77 GHz indicates a good match to 50. The antenna bandwidth is slightly greater than 1 GHz. Thus, the frequency band is from 76.5 GHz to 77.5 GHz.

Finally, check if the pattern at the edge frequencies of the band meets the design. This is a good indication whether the pattern behaves the same across the band. The patterns at 76.5 GHz and 77.6 GHz are shown below.

```
pattern(patchElement,76.5e9)
```
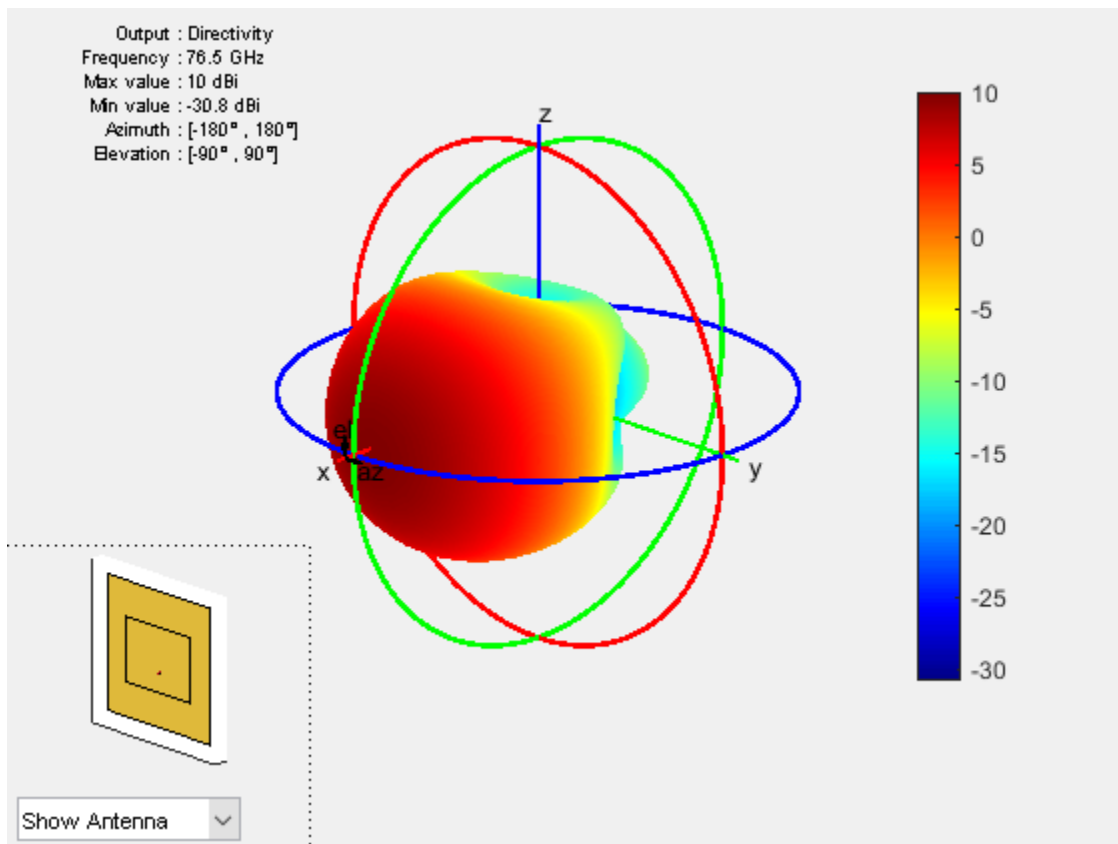
```
pattern(patchElement,77.6e9)
```
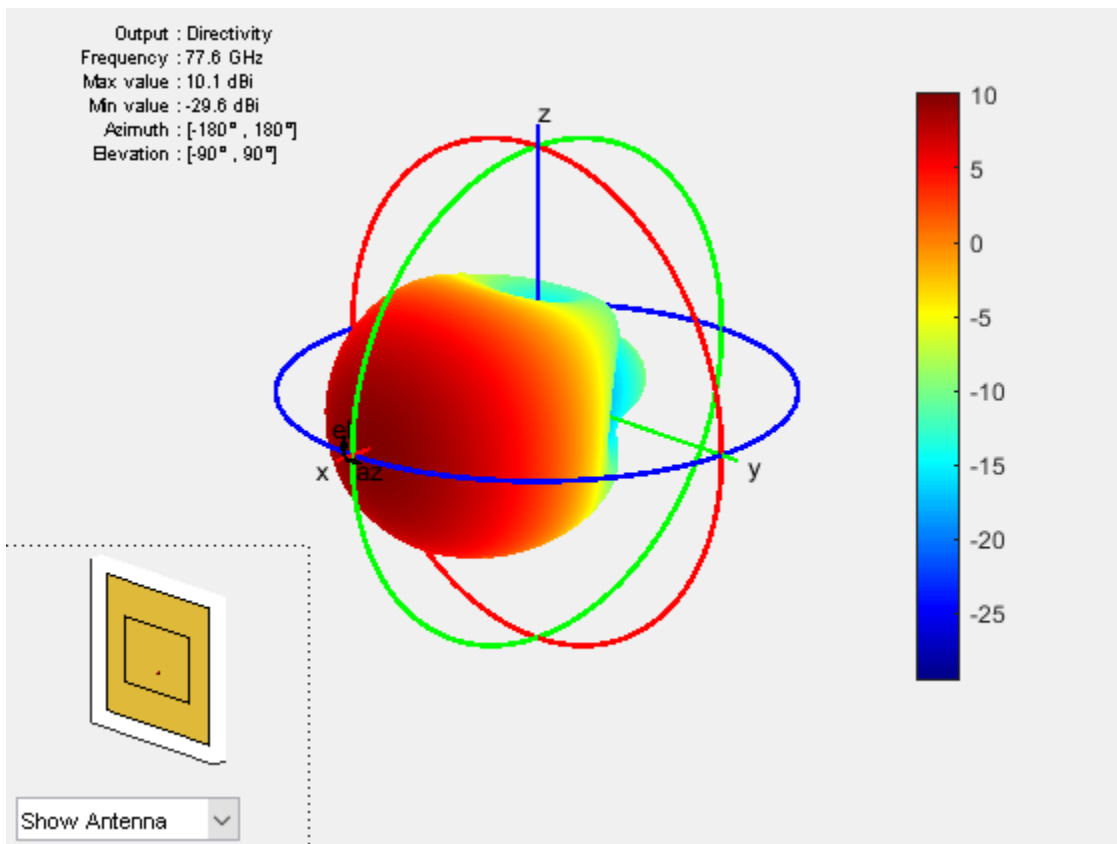
Create Array from Isolated Radiators and Plot Pattern

Next, create a uniform rectangular array (URA) with the patch antenna. The spacing is chosen to be $\lambda/2$, where λ is the wavelength at the upper frequency of the band (77.6 GHz).

```
fc2 = 77.6e9;
lambda_fc2 = vp/77.6e9;
fmcwPatchArray = phased.URA;
fmcwPatchArray.Element = patchElement;
fmcwPatchArray.Size = [Nrow Ncol];
fmcwPatchArray.ElementSpacing = [0.5*lambda_fc2 0.5*lambda_fc2];
```

The following figure shows the pattern of the resulting patch antenna array. The pattern is computed using a 5 degree separation in both azimuth and elevation.

```
az = -180:5:180;
el = -90:5:90;
clf
pattern(fmcwPatchArray,fc,az,el)
```

3D Directivity Pattern
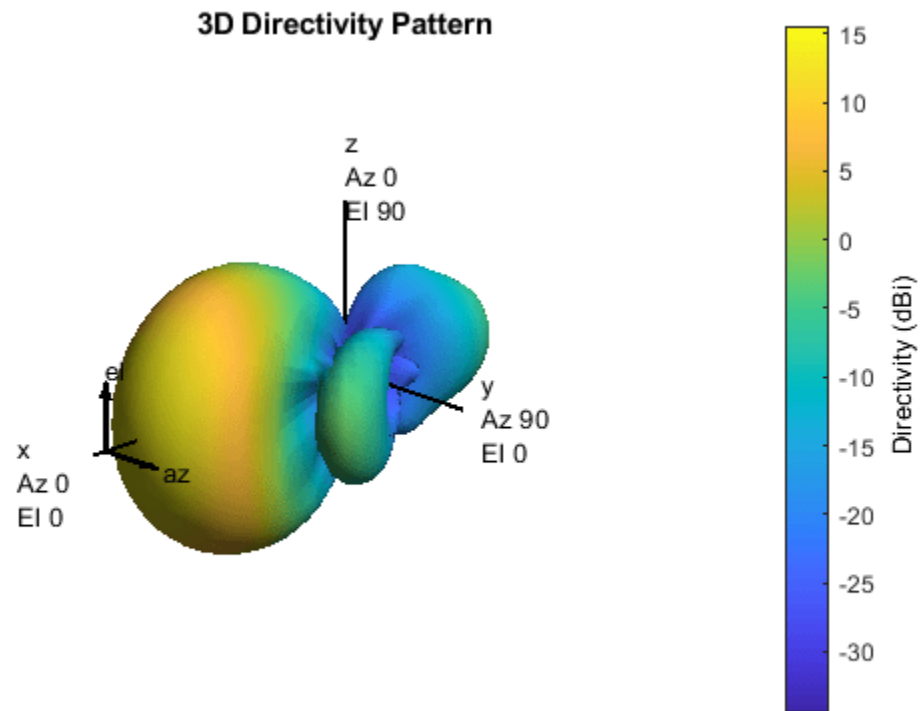
Plots below compare the pattern variation in orthogonal planes for the patch antenna array and the cosine element array. Note that both arrays ignore mutual coupling effect.

First, plot the patterns along the azimuth direction.

```
patternAzimuth(fmcwPatchArray,fc)
hold on
patternAzimuth(fmcwCosineArray,fc)
p = polarpattern('gco');
p.LegendLabels = {'Patch','Cosine'};
```

Azimuth Cut (elevation angle = 0.0°)

Directivity (dBi), Broadside at 0.00 °
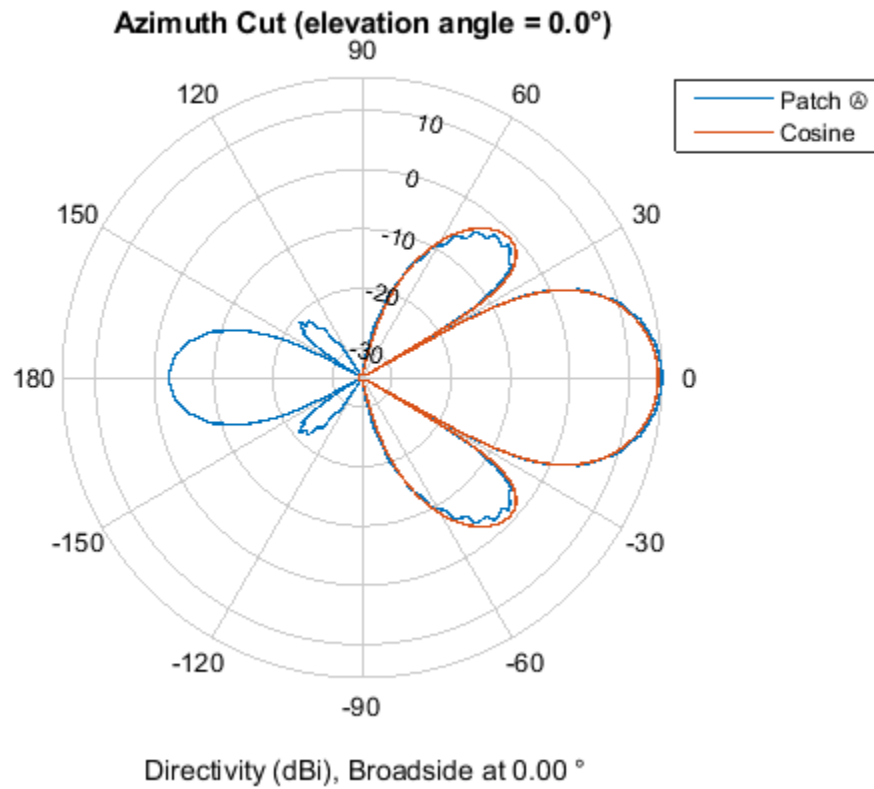
Then, plot the patterns along the elevation direction.

```
clf
patternElevation(fmcwPatchArray,fc)
hold on
patternElevation(fmcwCosineArray,fc)
p = polarpattern('gco');
p.LegendLabels = {'Patch','Cosine'};
```

Elevation Cut (azimuth angle = 0.0°)

Directivity (dBi), Broadside at 0.00 °
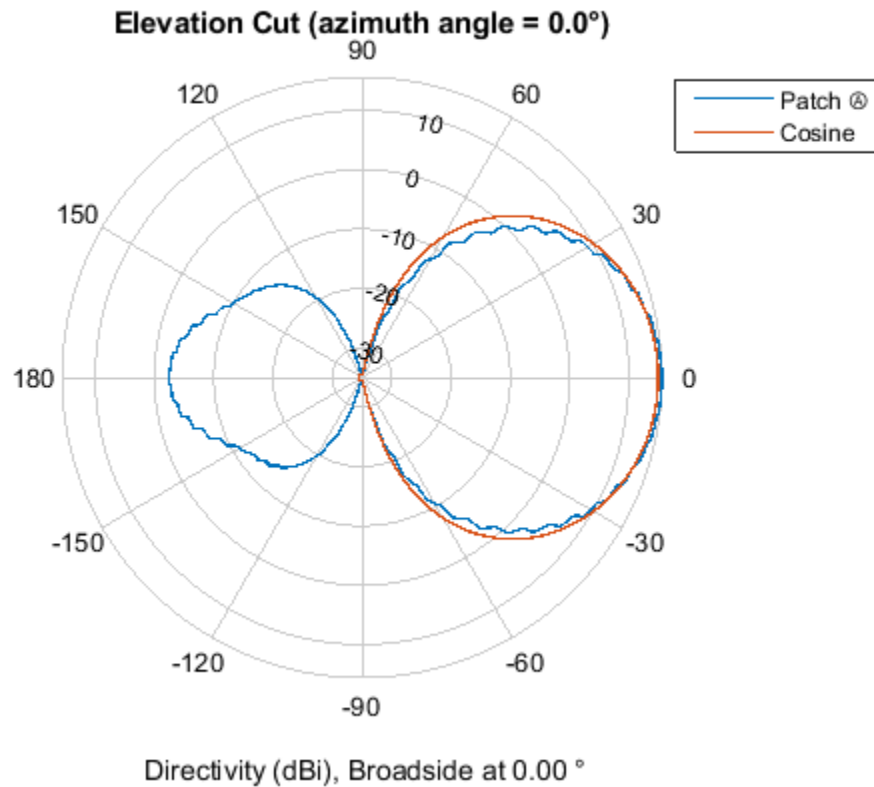
The figures show that both arrays have similar pattern behavior around the main beam in the elevation plane (azimuth = 0 deg). The patch-element array has a significant backlobe as compared to the cosine-element array.

Conclusions

This example starts the design of an antenna array for FMCW radar with an ideal cosine antenna and then uses a patch antenna to form the real array. The example compares the patterns from the two arrays to show the design tradeoff. From the comparison, it can be seen that using the isolated patch element is a useful first step in understanding the effect that a realistic antenna element will have on the array pattern.

However, analysis of realistic arrays must also consider mutual coupling effect. Since this is a small array, the individual element patterns in the array environment could be distorted significantly. As a result it is not possible to replace the isolated element pattern with an embedded element pattern, as shown in the "Modeling Mutual Coupling in Large Arrays Using Embedded Element Pattern" example. A full-wave analysis must be performed to understand the effect of mutual coupling on the overall array performance.

Reference

[1] R. Kulke, et al. 24 GHz Radar Sensor Integrates Patch Antennas, EMPC 2005 `http://empire.de/main/Empire/pdf/publications/2005/26-doc-empc2005.pdf`

1-141

Simultaneous Range and Speed Estimation Using MFSK Waveform

This example compares triangle sweep frequency-modulated continuous (FMCW) and multiple frequency-shift keying (MFSK) waveforms used for simultaneous range and speed estimation for multiple targets. The MFSK waveform is specifically designed for automotive radar systems used in advanced driver assistance systems (ADAS). It is particularly appealing in multitarget scenarios because it does not introduce ghost targets.

Triangle Sweep FMCW Waveform

In the "Automotive Adaptive Cruise Control Using FMCW Technology" on page 1-148 example an automotive radar system is designed to perform range estimation for an automatic cruise control system. In the latter part of that example, a triangle sweep FMCW waveform is used to estimate the range and speed of the target vehicle simultaneously.

Although the triangle sweep FMCW waveform elegantly solves the range-Doppler coupling issue for a single target, its processing becomes complicated in multitarget situations. The next section shows how a triangle sweep FMCW waveform behaves when two targets are present.

The scene includes a car 50 m away from the radar, traveling at 96 km/h along the same direction as the radar, and a truck at 55 m away, traveling at 70 km/h in the opposite direction. The radar itself is traveling at 60 km/h.

```
rng(2015);
[fmcwwaveform,target,tgtmotion,channel,transmitter,receiver, ...
    sensormotion,c,fc,lambda,fs,maxbeatfreq] = helperMFSKSystemSetup;
```

Next, simulate the radar echo from the two vehicles. The FMCW waveform has a sweep bandwidth of 150 MHz so the range resolution is 1 meter. Each up or down sweep takes 1 millisecond, so each triangle sweep takes 2 milliseconds. Note that only one triangle sweep is needed to perform the joint range and speed estimation.

```
Nsweep = 2;
xr = helperFMCWSimulate(Nsweep,fmcwwaveform,sensormotion,tgtmotion, ...
    transmitter,channel,target,receiver);
```

Although the system needs a 150 MHz bandwidth, the maximum beat frequency is much less. This means that at the processing side, one can decimate the signal to a lower frequency to ease the hardware requirements. The beat frequencies are then estimated using the decimated signal.

```
dfactor = ceil(fs/maxbeatfreq)/2;
fs_d = fs/dfactor;
fbu_rng = rootmusic(decimate(xr(:,1),dfactor),2,fs_d);
fbd_rng = rootmusic(decimate(xr(:,2),dfactor),2,fs_d);
```

Now there are two beat frequencies from the up sweep and two beat frequencies from the down sweeps. Since any pair of beat frequencies from an up sweep and a down sweep can define a target, there are four possible combinations of range and Doppler estimates, yet only two of them are associated with the real targets.

```
sweep_slope = fmcwwaveform.SweepBandwidth/fmcwwaveform.SweepTime;
rng_est = beat2range([fbu_rng fbd_rng;fbu_rng flipud(fbd_rng)], ...
    sweep_slope,c)
```

```
rng_est = 4×1

    49.9802
    54.9406
    64.2998
    40.6210
```

The remaining two are what is often referred to as ghost targets. The relationship between real targets and ghost targets can be better explained using time-frequency representation.
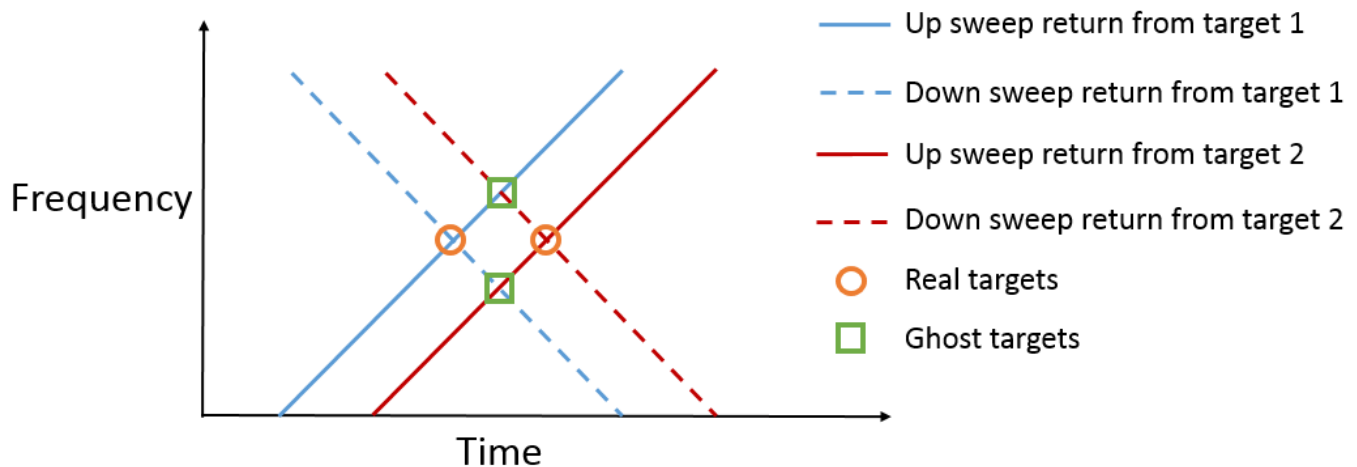
As shown in the figure, each intersection of an up sweep return and a down sweep return indicates a possible target. So it is critical to distinguish between the true targets and the ghost targets. To solve this ambiguity, one can transmit additional FMCW signals with different sweep slopes. Since only the true targets will occupy the same intersection in the time-frequency domain, the ambiguity is resolved. However, this approach significantly increases the processing complexity as well as the processing time needed to obtain the valid estimates.

MFSK Waveform

Multiple frequency shift keying (MFSK) waveform [1] on page 1-147 is designed for the automotive radar to achieve simultaneous range and Doppler estimation under the multiple targets situation without falling into the trap of ghost targets. Its time-frequency representation is shown in the following figure.
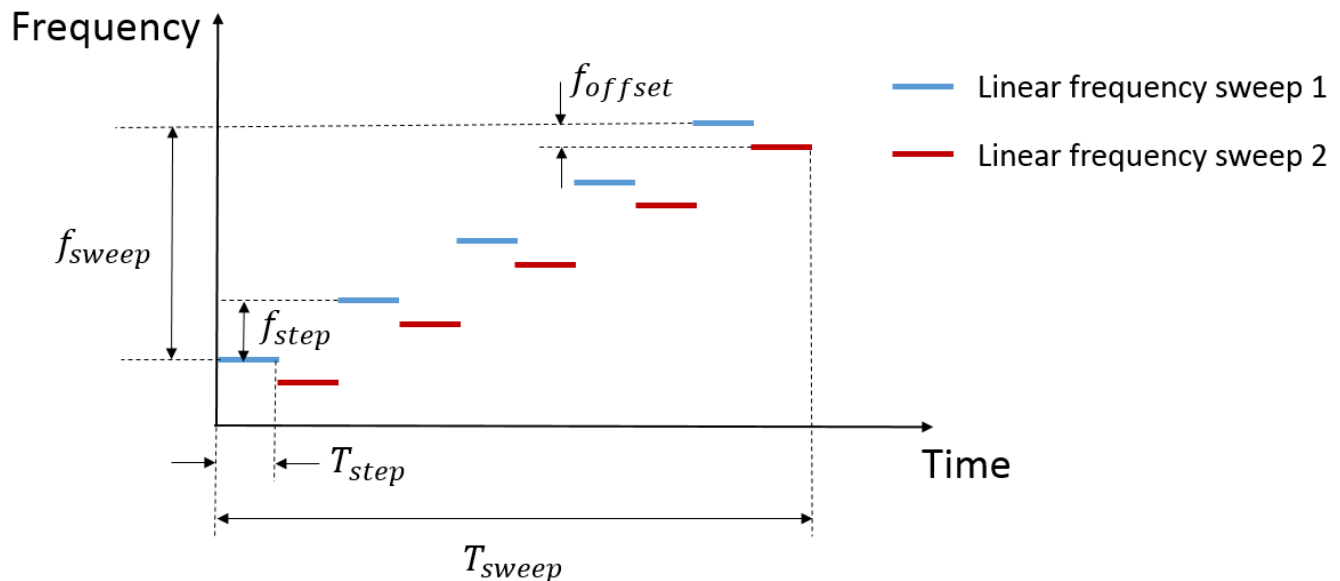
The figure indicates that the MFSK waveform is a combination of two linear FMCW waveforms with a fixed frequency offset. Unlike the regular FMCW waveforms, MFSK sweeps the entire bandwidth at discrete steps. Within each step, a single frequency continuous wave signal is transmitted. Because there are two tones within each step, it can be considered as a frequency shift keying (FSK) waveform. Thus, there is one set of range and Doppler relation from FMCW waveform and another set of range and Doppler relation from FSK. Combining two sets of relations together can help resolve the coupling between range and Doppler regardless of the number of targets present in the scene.

The following sections simulate the previous example, but use an MFSK waveform instead.

End-to-end Radar System Simulation Using MFSK Waveform

First, parameterize the MFSK waveform to satisfy the system requirement specified in [1] on page 1-147. Because the range resolution is 1 m, the sweep bandwidth is set at 150 MHz. In addition, the frequency offset is set at —294 kHz as specified in [1] on page 1-147. Each step lasts about 2 microseconds and the entire sweep has 1024 steps. Thus, each FMCW sweep takes 512 steps and the total sweep time is a little over 2 ms. Note that the sweep time is comparable to the FMCW signal used in previous sections.

```
mfskwaveform = phased.MFSKWaveform( ...
    'SampleRate',151e6, ...
    'SweepBandwidth',150e6, ...
    'StepTime',2e-6, ...
    'StepsPerSweep',1024, ...
    'FrequencyOffset',-294e3, ...
    'OutputFormat','Sweeps', ...
    'NumSweeps',1);
```

The figure below shows the spectrogram of the waveform. It is zoomed into a small interval to better reveal the time-frequency characteristics of the waveform.

```
numsamp_step = round(mfskwaveform.SampleRate*mfskwaveform.StepTime);
sig_display = mfskwaveform();
```

```
spectrogram(sig_display(1:8192),kaiser(3*numsamp_step,100), ...
    ceil(2*numsamp_step),linspace(0,4e6,2048),mfskwaveform.SampleRate, ...
    'yaxis','reassigned','minthreshold',-60)
```
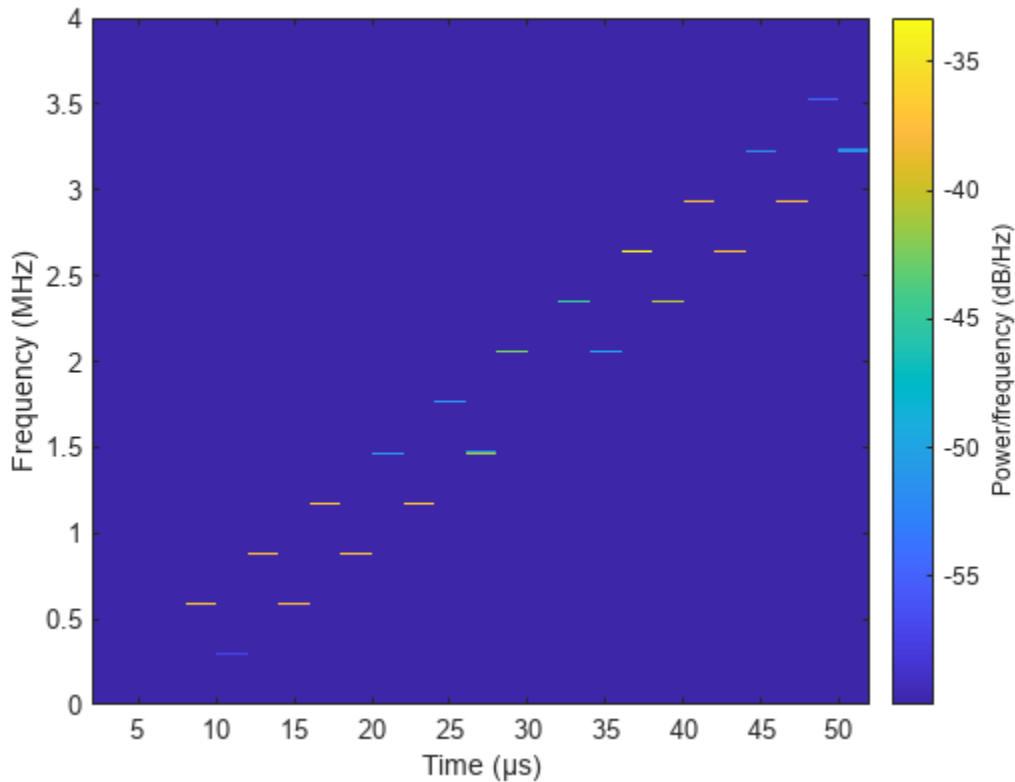

Next, simulate the return of the system. Again, only one sweep is needed to estimate the range and Doppler.

```
Nsweep = 1;
release(channel);
channel.SampleRate = mfskwaveform.SampleRate;
release(receiver);
receiver.SampleRate = mfskwaveform.SampleRate;
xr = helperFMCWSimulate(Nsweep,mfskwaveform,sensormotion,tgtmotion, ...
    transmitter,channel,target,receiver);
```

The subsequent processing samples the return echo at the end of each step and group the sampled signals into two sequences corresponding to two sweeps. Note that the sampling frequency of the resulting sequence is now proportional to the time at each step, which is much less compared the original sample rate.

```
x_dechirp = reshape(xr(numsamp_step:numsamp_step:end),2,[]).';
fs_dechirp = 1/(2*mfskwaveform.StepTime);
```

As in the case of FMCW signals, the MFSK waveform is processed in the frequency domain. The following figures show the frequency spectrums of the received echoes corresponding to the two sweeps.

```
xf_dechirp = fft(x_dechirp);
num_xf_samp = size(xf_dechirp,1);
beatfreq_vec = (0:num_xf_samp-1).'/num_xf_samp*fs_dechirp;
clf
subplot(211),plot(beatfreq_vec/1e3,abs(xf_dechirp(:,1)))
grid on
ylabel('Magnitude')
title('Frequency spectrum for sweep 1')
subplot(212),plot(beatfreq_vec/1e3,abs(xf_dechirp(:,2)))
grid on
ylabel('Magnitude')
title('Frequency spectrum for sweep 2')
xlabel('Frequency (kHz)')
```
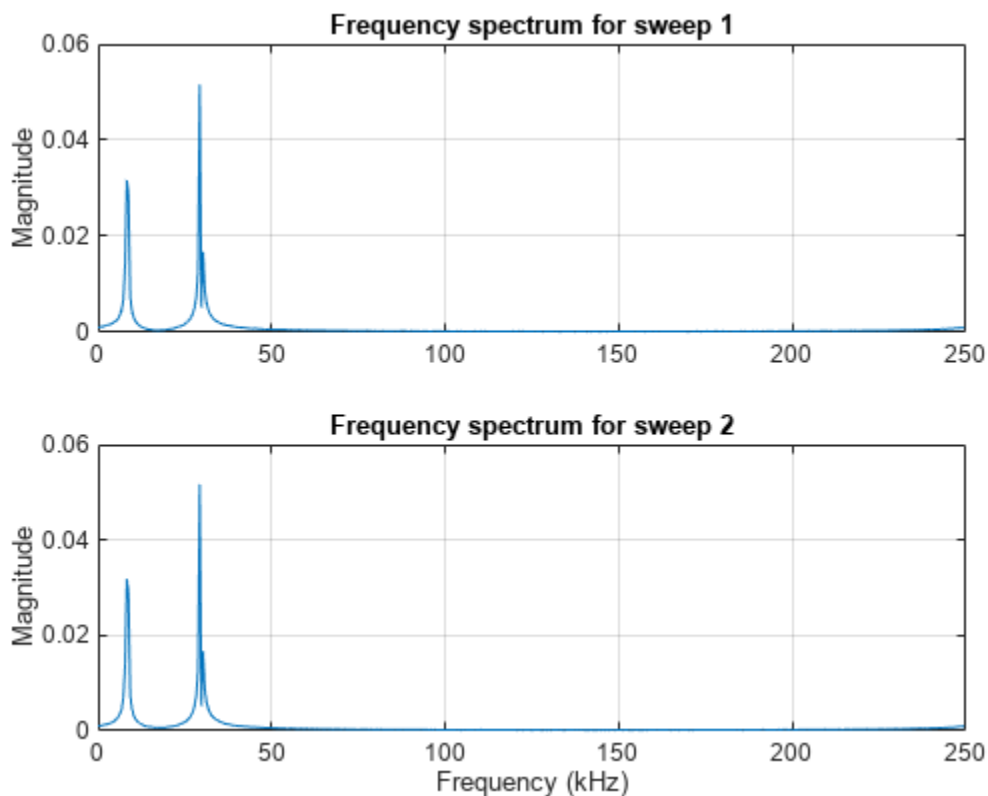
Note that there are two peaks in each frequency spectrum indicating two targets. In addition, the peaks are at the same locations in both returns so there are no ghost targets.

To detect the peaks, one can use a CFAR detector. Once detected, the beat frequencies as well as the phase differences between two spectra are computed at the peak locations.

```
cfar = phased.CFARDetector('ProbabilityFalseAlarm',1e-2, ...
    'NumTrainingCells',8);
peakidx = cfar(abs(xf_dechirp(:,1)),1:num_xf_samp);
Fbeat = beatfreq_vec(peakidx);
phi = angle(xf_dechirp(peakidx,2))-angle(xf_dechirp(peakidx,1));
```

Finally, the beat frequencies and phase differences are used to estimate the range and speed. Depending on how you construct the phase difference, the equations are slightly different. For the

approach shown in this example, it can be shown that the range and speed satisfy the following relation:

$$f_b = -\frac{2v}{\lambda} + \frac{2\beta R}{c}$$

$$\Delta\phi = -\frac{4\pi T_s v}{\lambda} + \frac{4\pi f_{offset} R}{c}$$

where f_b is the beat frequency, $\Delta\phi$ is the phase difference, λ is the wavelength, c is the propagation speed, T_s is the step time, f_{offset} is the frequency offset, β is the sweep slope, R is the range, and v is the speed. Based on the equation, the range and speed are estimated as follows.

```
sweep_slope = mfskwaveform.SweepBandwidth/ ...
    (mfskwaveform.StepsPerSweep*mfskwaveform.StepTime);
temp = ...
    [-2/lambda 2*sweep_slope/c;-2/lambda*mfskwaveform.StepTime 2*mfskwaveform.FrequencyOffset/c]\
    [Fbeat phi/(2*pi)].';
r_est = temp(2,:)

r_est = 1×2

   54.8564   49.6452


v_est = temp(1,:)

v_est = 1×2

   36.0089   -9.8495
```

The estimated range and speed match the true range and speed values very well.

- Car: r = 50 m, v = -10 m/s
- Truck: r = 55 m, v = 36 m/s

Summary

This example shows two simultaneous range and speed estimation approaches, using either a triangle sweep FMCW waveform or an MFSK waveform. This example shows that the MFSK waveform has an advantage over the FMCW waveform when multiple targets are present because it does not introduce ghost targets during the processing.

References

[1] Rohling, H. and M. Meinecke. *Waveform Design Principle for Automotive Radar Systems*, Proceedings of CIE International Conference on Radar, 2001.

Automotive Adaptive Cruise Control Using FMCW Technology

This example shows how to model an automotive adaptive cruise control system using the frequency modulated continuous wave (FMCW) technique. This example performs range and Doppler estimation of a moving vehicle. Unlike pulsed radar systems that are commonly seen in the defense industry, automotive radar systems often adopt FMCW technology. Compared to pulsed radars, FMCW radars are smaller, use less power, and are much cheaper to manufacture. As a consequence, FMCW radars can only monitor a much smaller distance.

FMCW Waveform

Consider an automotive long range radar (LRR) used for adaptive cruise control (ACC). This kind of radar usually occupies the band around 77 GHz, as indicated in [1]. The radar system constantly estimates the distance between the vehicle it is mounted on and the vehicle in front of it, and alerts the driver when the two become too close. This figure shows a sketch of ACC.

A popular waveform used in ACC system is FMCW. The principle of range measurement using the FMCW technique can be illustrated using the following figure.
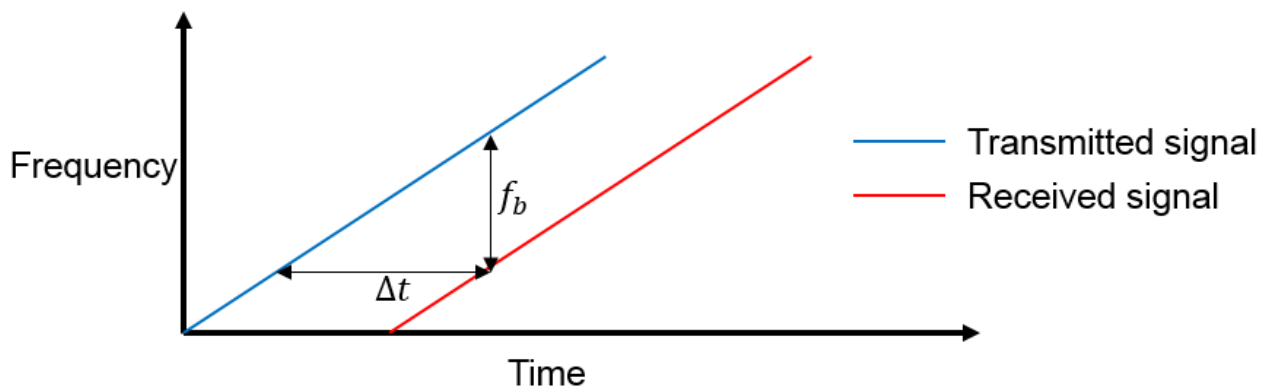
The received signal is a time-delayed copy of the transmitted signal where the delay, Δt, is related to the range. Because the signal is always sweeping through a frequency band, at any moment during the sweep, the frequency difference, f_b, is a constant between the transmitted signal and the received signal. f_b is usually called the beat frequency. Because the sweep is linear, you can derive the time delay from the beat frequency and then translate the delay to the range.

In an ACC setup, the maximum range the radar needs to monitor is around 200 m and the system needs to be able to distinguish two targets that are 1 meter apart. From these requirements, you can compute the waveform parameters.

```
fc = 77e9;
c = 3e8;
lambda = c/fc;
```

The sweep time can be computed based on the time needed for the signal to travel the unambiguous maximum range. In general, for an FMCW radar system, the sweep time should be at least five to six times the round trip time. This example uses a factor of 5.5.

```
range_max = 200;
tm = 5.5*range2time(range_max,c);
```

The sweep bandwidth can be determined according to the range resolution and the sweep slope is calculated using both sweep bandwidth and sweep time.

```
range_res = 1;
bw = rangeres2bw(range_res,c);
sweep_slope = bw/tm;
```

Because an FMCW signal often occupies a large bandwidth, setting the sample rate blindly to twice the bandwidth often stresses the capability of A/D converter hardware. To address this issue, you can often choose a lower sample rate. Consider two things here:

1 For a complex sampled signal, the sample rate can be set to the same as the bandwidth.
2 FMCW radars estimate the target range using the beat frequency embedded in the dechirped signal. The maximum beat frequency the radar needs to detect is the sum of the beat frequency corresponding to the maximum range and the maximum Doppler frequency. Hence, the sample rate only needs to be twice the maximum beat frequency.

In this example, the beat frequency corresponding to the maximum range is as follows.

```
fr_max = range2beat(range_max,sweep_slope,c);
```

Because the maximum speed of a traveling car is about 230 km/h. Hence the maximum Doppler shift and the maximum beat frequency can be computed as

```
v_max = 230*1000/3600;
fd_max = speed2dop(2*v_max,lambda);
fb_max = fr_max+fd_max;
```

This example adopts a sample rate of the larger of twice the maximum beat frequency and the bandwidth.

```
fs = max(2*fb_max,bw);
```

The following table summarizes the radar parameters.

```
System parameters             Value
----------------------------------
Operating frequency (GHz)     77
Maximum target range (m)      200
Range resolution (m)          1
Maximum target speed (km/h)   230
Sweep time (microseconds)     7.33
```

```
Sweep bandwidth (MHz)         150
Maximum beat frequency (MHz) 27.30
Sample rate (MHz)             150
```

With this information, set up the FMCW waveform used in the radar system.

```
waveform = phased.FMCWWaveform('SweepTime',tm,'SweepBandwidth',bw, ...
    'SampleRate',fs);
```

This is an up-sweep linear FMCW signal, often referred to as a sawtooth shape. Examine the time-frequency plot of the generated signal.

```
sig = waveform();
subplot(211); plot(0:1/fs:tm-1/fs,real(sig));
xlabel('Time (s)'); ylabel('Amplitude (v)');
title('FMCW signal'); axis tight;
subplot(212); spectrogram(sig,32,16,32,fs,'yaxis');
title('FMCW signal spectrogram');
```
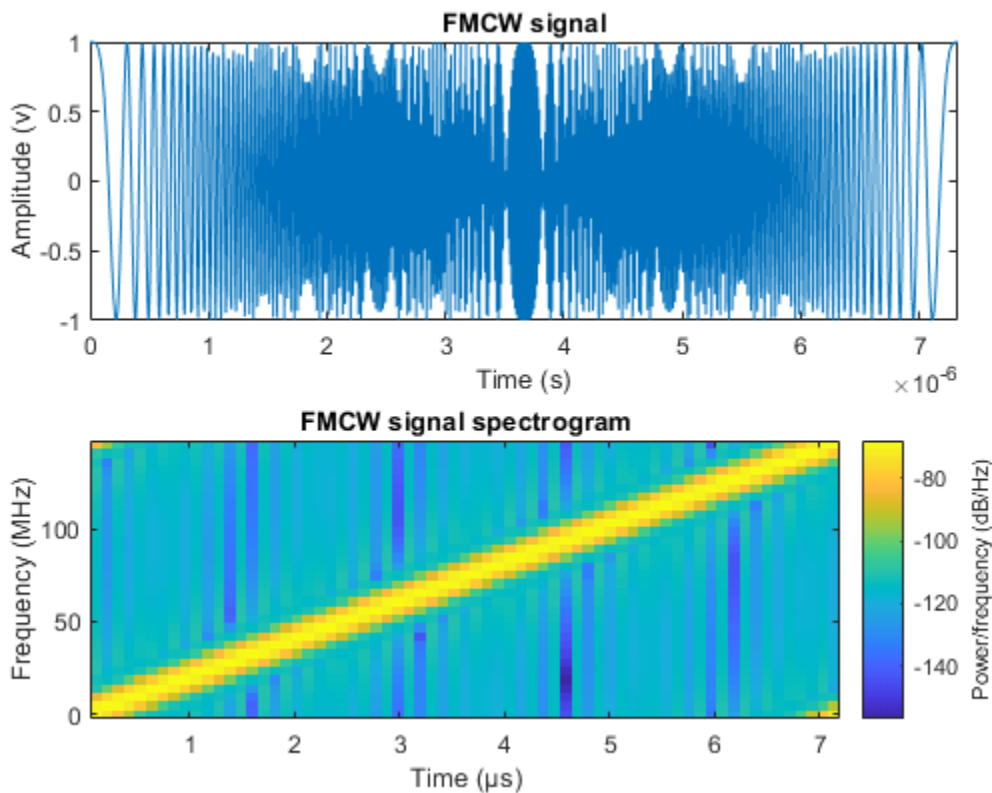

Target Model

The target of an ACC radar is usually a car in front of it. This example assumes the target car is moving 43 m ahead of the car with the radar, at a speed of 96 km/h along the x-axis.

Compute the radar cross section of a car, according to [1], based on the distance between the radar and the target car.

```
car_dist = 43;
car_speed = 96*1000/3600;
```

```
car_rcs = db2pow(min(10*log10(car_dist)+5,20));
cartarget = phased.RadarTarget('MeanRCS',car_rcs,'PropagationSpeed',c,...
    'OperatingFrequency',fc);
carmotion = phased.Platform('InitialPosition',[car_dist;0;0.5],...
    'Velocity',[car_speed;0;0]);
```

Assume the propagation model to be free space.

```
channel = phased.FreeSpace('PropagationSpeed',c,...
    'OperatingFrequency',fc,'SampleRate',fs,'TwoWayPropagation',true);
```

Radar System Setup

The rest of the radar system includes the transmitter, the receiver, and the antenna. This example uses the parameters presented in [1]. Note that this example models only main components and omits the effect from other components, such as coupler and mixer. In addition, for the sake of simplicity, the antenna is assumed to be isotropic and the gain of the antenna is included in the transmitter and the receiver.

```
ant_aperture = 6.06e-4;                          % in square meter
ant_gain = aperture2gain(ant_aperture,lambda);   % in dB

tx_ppower = db2pow(5)*1e-3;                       % in watts
tx_gain = 9+ant_gain;                            % in dB

rx_gain = 15+ant_gain;                           % in dB
rx_nf = 4.5;                                     % in dB

transmitter = phased.Transmitter('PeakPower',tx_ppower,'Gain',tx_gain);
receiver = phased.ReceiverPreamp('Gain',rx_gain,'NoiseFigure',rx_nf,...
    'SampleRate',fs);
```

Automotive radars are generally mounted on vehicles, so they are often in motion. This example assumes the radar is traveling at a speed of 100 km/h along x-axis. So the target car is approaching the radar at a relative speed of 4 km/h.

```
radar_speed = 100*1000/3600;
radarmotion = phased.Platform('InitialPosition',[0;0;0.5],...
    'Velocity',[radar_speed;0;0]);
```

Radar Signal Simulation

As briefly mentioned in earlier sections, an FMCW radar measures the range by examining the beat frequency in the dechirped signal. To extract this frequency, a dechirp operation is performed by mixing the received signal with the transmitted signal. After the mixing, the dechirped signal contains only individual frequency components that correspond to the target range.

In addition, even though it is possible to extract the Doppler information from a single sweep, the Doppler shift is often extracted among several sweeps because within one pulse, the Doppler frequency is indistinguishable from the beat frequency. To measure the range and Doppler, an FMCW radar typically performs the following operations:

1   The waveform generator generates the FMCW signal.

2   The transmitter and the antenna amplify the signal and radiate the signal into space.

3   The signal propagates to the target, gets reflected by the target, and travels back to the radar.

4   The receiving antenna collects the signal.

5   The received signal is dechirped and saved in a buffer.

6   Once a certain number of sweeps fill the buffer, the Fourier transform is performed in both range and Doppler to extract the beat frequency as well as the Doppler shift. can then estimate the range and speed of the target using these results. Range and Doppler can also be shown as an image and give an intuitive indication of where the target is in the range and speed domain.

The next section simulates the process outlined above. A total of 64 sweeps are simulated and a range Doppler response is generated at the end.

During the simulation, a spectrum analyzer is used to show the spectrum of each received sweep as well as its dechirped counterpart.

```matlab
specanalyzer = spectrumAnalyzer('SampleRate',fs, ...
    'Method','welch','AveragingMethod','running', ...
    'PlotAsTwoSidedSpectrum',true, 'FrequencyResolutionMethod','rbw', ...
    'Title','Spectrum for received and dechirped signal', ...
    'ShowLegend',true);
```

Next, run the simulation loop.

```matlab
rng(2012);
Nsweep = 64;
xr = complex(zeros(waveform.SampleRate*waveform.SweepTime,Nsweep));

for m = 1:Nsweep
    % Update radar and target positions
    [radar_pos,radar_vel] = radarmotion(waveform.SweepTime);
    [tgt_pos,tgt_vel] = carmotion(waveform.SweepTime);

    % Transmit FMCW waveform
    sig = waveform();
    txsig = transmitter(sig);

    % Propagate the signal and reflect off the target
    txsig = channel(txsig,radar_pos,tgt_pos,radar_vel,tgt_vel);
    txsig = cartarget(txsig);

    % Dechirp the received radar return
    txsig = receiver(txsig);
    dechirpsig = dechirp(txsig,sig);

    % Visualize the spectrum
    specanalyzer([txsig dechirpsig]);

    xr(:,m) = dechirpsig;
end
```
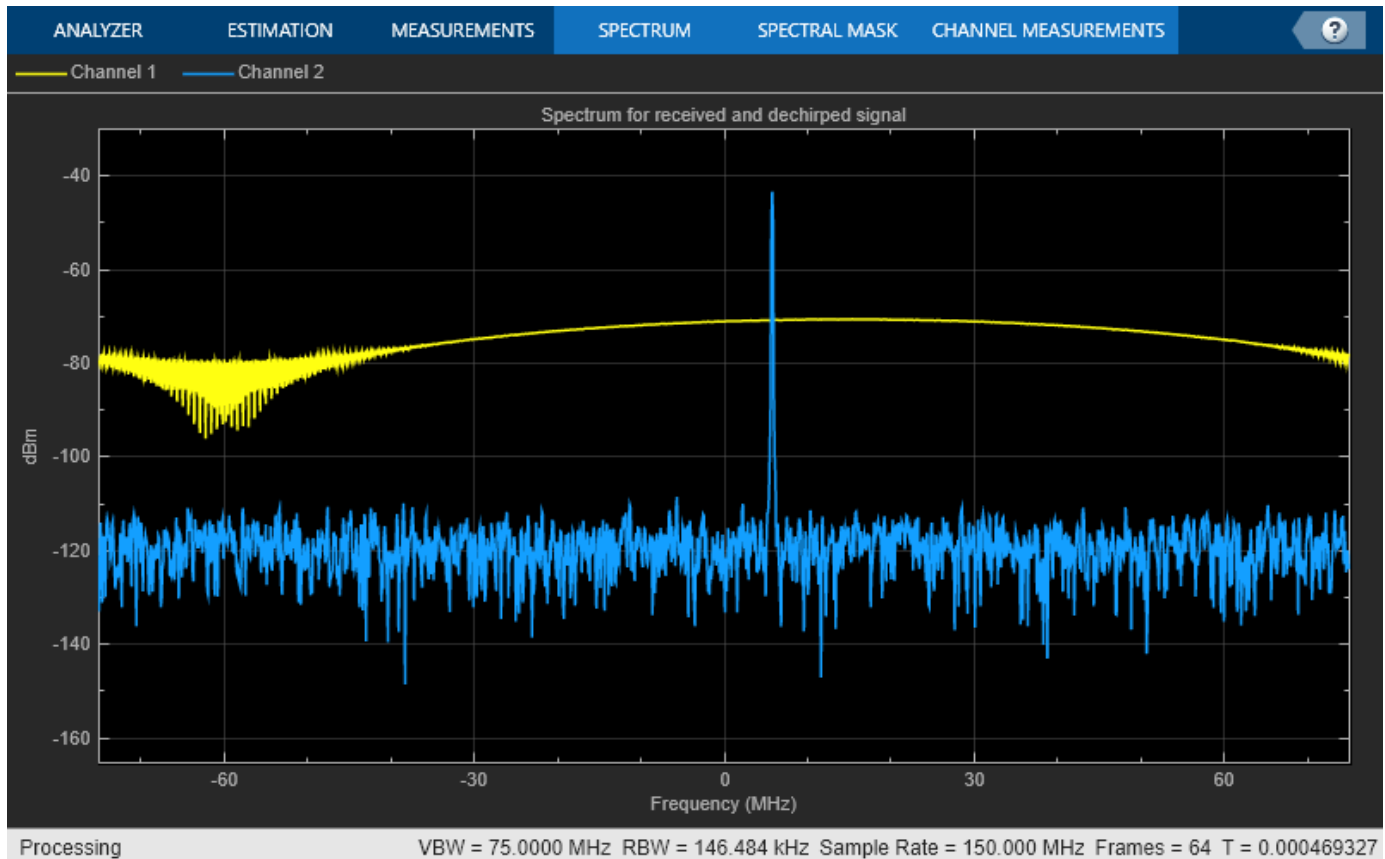
From the spectrum scope, you can see that although the received signal is wideband (channel 1), sweeping through the entire bandwidth, the dechirped signal becomes narrowband (channel 2).

Range and Doppler Estimation

Before estimating the value of the range and Doppler, take a look at the zoomed range Doppler response of all 64 sweeps.

```
rngdopresp = phased.RangeDopplerResponse('PropagationSpeed',c,...
    'DopplerOutput','Speed','OperatingFrequency',fc,'SampleRate',fs,...
    'RangeMethod','FFT','SweepSlope',sweep_slope,...
    'RangeFFTLengthSource','Property','RangeFFTLength',2048,...
    'DopplerFFTLengthSource','Property','DopplerFFTLength',256);

clf;
plotResponse(rngdopresp,xr);                    % Plot range Doppler map
axis([-v_max v_max 0 range_max])
clim = caxis;
```

Range-Speed Response Pattern
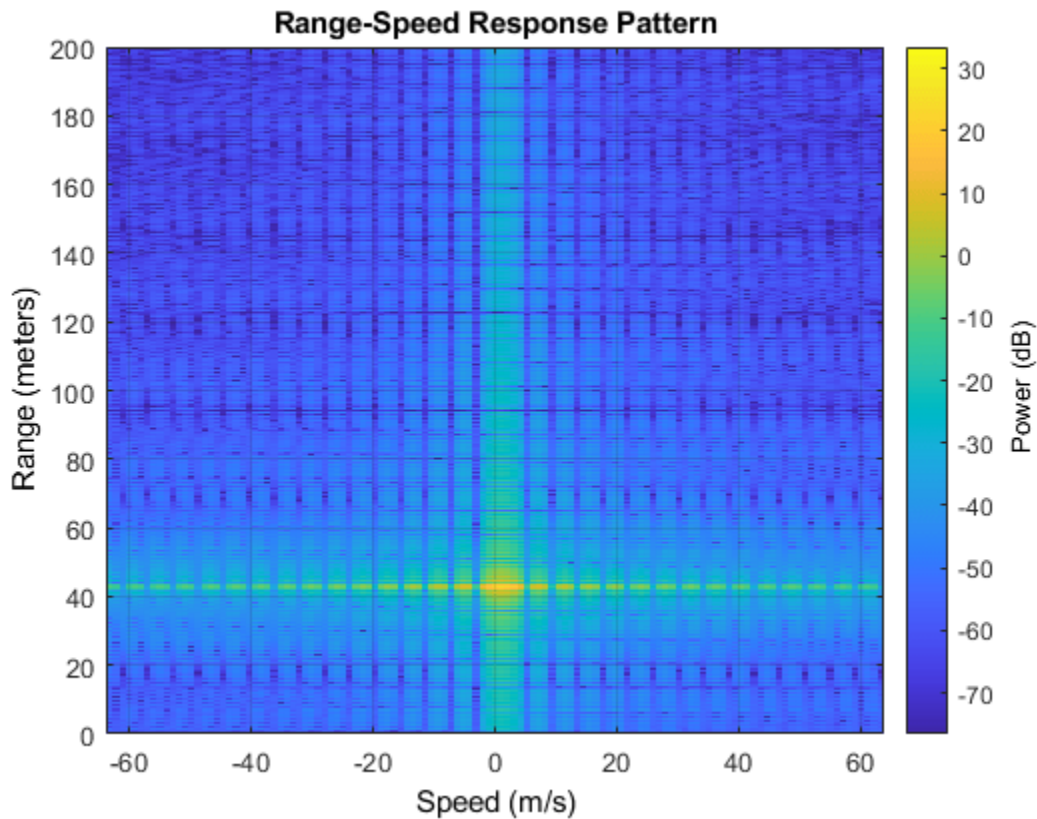
From the range Doppler response, you can see that the car in front is a bit more than 40 m away and appears almost static. This is expected because the radial speed of the car relative to the radar is only 4 km/h, which translates to a mere 1.11 m/s.

There are many ways to estimate the range and speed of the target car. For example, you can choose almost any spectral analysis method to extract both the beat frequency and the Doppler shift. This example uses the root MUSIC algorithm to extract both the beat frequency and the Doppler shift.

As a side note, although the received signal is sampled at 150 MHz so the system can achieve the required range resolution, after the dechirp, you need to sample it only at a rate that corresponds to the maximum beat frequency. Since the maximum beat frequency is in general less than the required sweeping bandwidth, the signal can be decimated to alleviate the hardware cost. The following code shows the decimation process.

```
Dn = fix(fs/(2*fb_max));
for m = size(xr,2):-1:1
    xr_d(:,m) = decimate(xr(:,m),Dn,'FIR');
end
fs_d = fs/Dn;
```

To estimate the range, firstly, estimate the beat frequency using the coherently integrated sweeps and then converted to the range.

```
fb_rng = rootmusic(pulsint(xr_d,'coherent'),1,fs_d);
rng_est = beat2range(fb_rng,sweep_slope,c)
```

```
rng_est =
```

```
    42.9976
```

Second, estimate the Doppler shift across the sweeps at the range where the target is present.

```
peak_loc = val2ind(rng_est,c/(fs_d*2));
fd = -rootmusic(xr_d(peak_loc,:),1,1/tm);
v_est = dop2speed(fd,lambda)/2
```

```
v_est =

    1.0830
```

Note that both range and Doppler estimation are quite accurate.

Range Doppler Coupling Effect

One issue associated with linear FM signals, such as an FMCW signal, is the range Doppler coupling effect. As discussed earlier, the target range corresponds to the beat frequency. Hence, an accurate range estimation depends on an accurate estimate of beat frequency. However, the presence of Doppler shift changes the beat frequency, resulting in a biased range estimation.

For the situation outlined in this example, the range error caused by the relative speed between the target and the radar is as follows

```
deltaR = rdcoupling(fd,sweep_slope,c)
```

```
deltaR =

   -0.0041
```

This error is so small that you can safely ignore it.

Even though the current design achieves the desired performance, one parameter warrants further attention. In the current configuration, the sweep time is about 7 microseconds. Therefore, the system needs to sweep a 150 MHz band within a very short period. Such an automotive radar might not be able to meet the cost requirement. Besides, given the velocity of a car, there is no need to make measurements every 7 microseconds. Hence, automotive radars often use a longer sweep time. For example, the waveform used in [2] has the same parameters as the waveform designed in this example except a sweep time of 2 ms.

A longer sweep time makes the range Doppler coupling more prominent. To see this effect, first reconfigure the waveform to use 2 ms as the sweep time.

```
waveform_tr = clone(waveform);
release(waveform_tr);
tm = 2e-3;
waveform_tr.SweepTime = tm;
sweep_slope = bw/tm;
```

Now calculate the range Doppler coupling.

```
deltaR = rdcoupling(fd,sweep_slope,c)
```

```
deltaR =

   -1.1118
```

A range error of 1.14 m can no longer be ignored and needs to be compensated. Naturally, you might consider doing so following the same procedure outlined in earlier sections, estimating both range and Doppler, figuring out the range Doppler coupling from the Doppler shift, and then remove the error from the estimate.

Unfortunately this process does not work very well with the long sweep time. The longer sweep time results in a lower sampling rate across the sweeps, thus reducing the capability of the radar for unambiguously detecting high speed vehicles. For instance, using a sweep time of 2 ms, the maximum unambiguous speed the radar system can detect using the traditional Doppler processing is as follows

```
v_unambiguous = dop2speed(1/(2*tm),lambda)/2
```

```
v_unambiguous =

   0.4870
```

The unambiguous speed is only 0.48 m/s, which means that the relative speed, 1.11 m/s, cannot be unambiguously detected. This means that not only the target car will appear slower in Doppler processing, but the range Doppler coupling also cannot be correctly compensated.

One way to resolve such ambiguity without Doppler processing is to adopt a triangle sweep pattern. Next section shows how the triangle sweep addresses the issue.

Triangular Sweep

In a triangular sweep, there is one up sweep and one down sweep to form one period, as shown in the following figure.
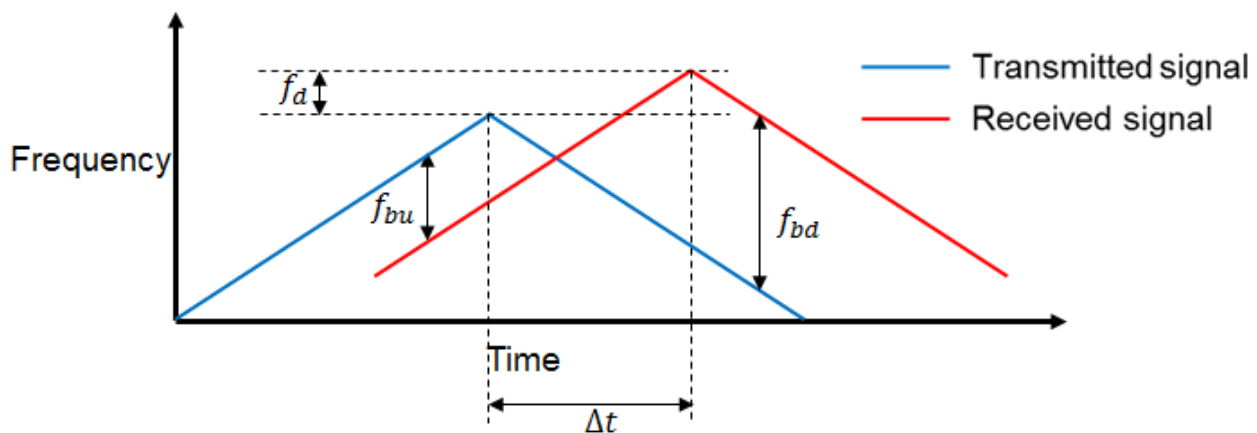
The two sweeps have the same slope except different signs. From the figure, you can see that the presence of Doppler frequency, f_d, affects the beat frequencies (f_{bu} and f_{bd}) differently in up and

down sweeps. Hence by combining the beat frequencies from both up and down sweep, the coupling effect from the Doppler can be averaged out and the range estimate can be obtained without ambiguity.

First, set the waveform to use triangular sweep.

```
waveform_tr.SweepDirection = 'Triangle';
```

Now simulate the signal return. Because of the longer sweep time, fewer sweeps (16 vs. 64) are collected before processing.

```
Nsweep = 16;
xr = helperFMCWSimulate(Nsweep,waveform_tr,radarmotion,carmotion,...
    transmitter,channel,cartarget,receiver);
```

The up sweep and down sweep are processed separately to obtain the beat frequencies corresponding to both up and down sweep.

```
fbu_rng = rootmusic(pulsint(xr(:,1:2:end),'coherent'),1,fs);
fbd_rng = rootmusic(pulsint(xr(:,2:2:end),'coherent'),1,fs);
```

Using both up sweep and down sweep beat frequencies simultaneously, the correct range estimate is obtained.

```
rng_est = beat2range([fbu_rng fbd_rng],sweep_slope,c)
```

```
rng_est =

   42.9658
```

Moreover, the Doppler shift and the velocity can also be recovered in a similar fashion.

```
fd = -(fbu_rng+fbd_rng)/2;
v_est = dop2speed(fd,lambda)/2
```

```
v_est =

   1.1114
```

The estimated range and velocity match the true values, 43 m and 1.11 m/s, very well.

Two-ray Propagation

To complete the discussion, in reality, the actual signal propagation between the radar and the target vehicle is more complicated than what is modeled so far. For example, the radio wave can also arrive at the target vehicle via reflections. A simple yet widely used model to describe such a multipath scenario is a two-ray model, where the signal propagates from the radar to the target vehicle via two paths, one direct path and one reflected path off the road, as shown in the following figure.
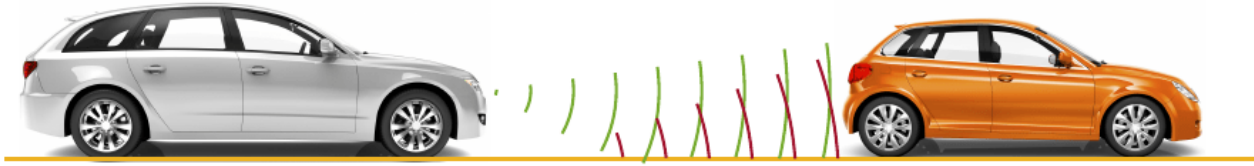
The reflection off the road affects the phase of the signal and the received signal at the target vehicle is a coherent combination of the signals via the two paths. The same thing happens on the return trip where the reflected signal from the target vehicle travels back to the radar. Hence depending on the distance between the vehicles, the signals from different paths may add constructively or destructively, making signal strength fluctuate over time. Such fluctuation can pose some challenge in the successive detection stage.

To showcase the multipath effect, next section uses the two-ray channel model to propagate the signal between the radar and the target vehicle.

```matlab
txchannel = twoRayChannel('PropagationSpeed',c,...
    'OperatingFrequency',fc,'SampleRate',fs);
rxchannel = twoRayChannel('PropagationSpeed',c,...
    'OperatingFrequency',fc,'SampleRate',fs);
Nsweep = 64;
xr = helperFMCWTwoRaySimulate(Nsweep,waveform,radarmotion,carmotion,...
    transmitter,txchannel,rxchannel,cartarget,receiver);
plotResponse(rngdopresp,xr);                     % Plot range Doppler map
axis([-v_max v_max 0 range_max]);
caxis(clim);
```

Range-Speed Response Pattern
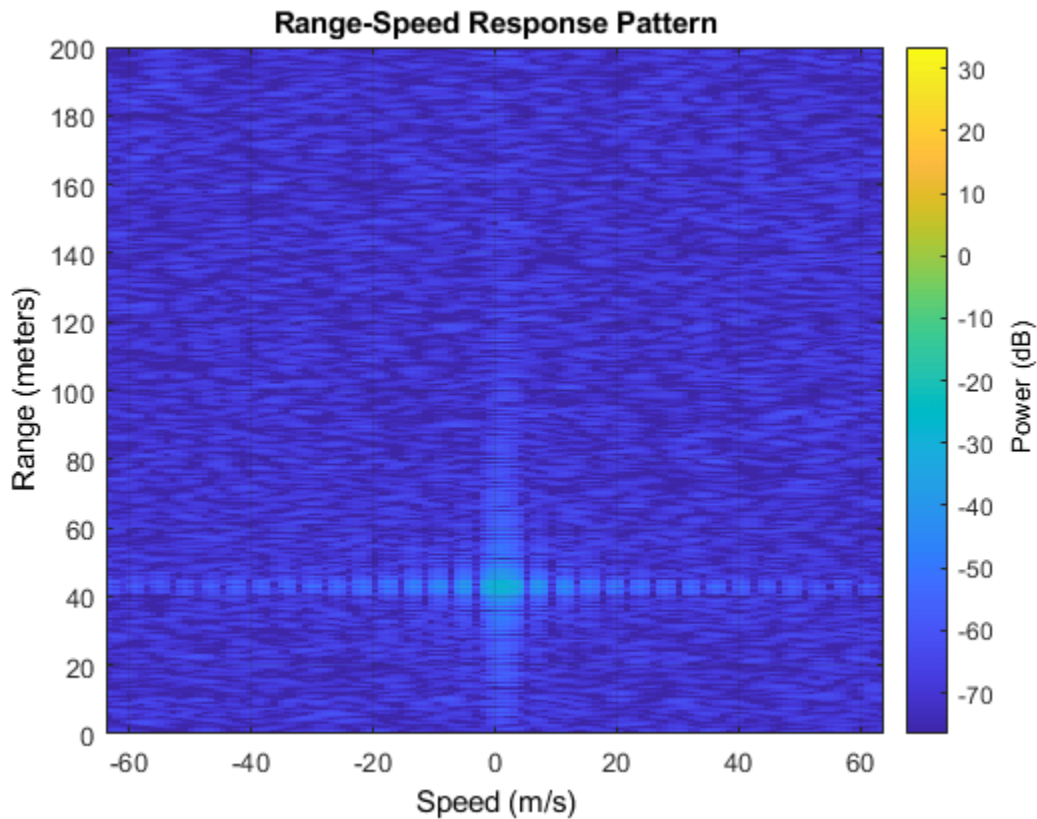
With all settings remaining same, the comparison of the resulting range-Doppler map with two-ray propagation and the range-Doppler map obtained before with a line of sight (LOS) propagation channel suggests that the signal strength dropped almost 40 dB, which is significant. Therefore, such effect must be considered during the design. One possible choice is to form a very sharp beam on the vertical direction to null out the reflections.

Summary

This example shows how to use FMCW signal to perform range and Doppler estimation in an automotive automatic cruise control application. The example also shows how to generate a range Doppler map from the received signal and how to use triangle sweep to compensate for the range Doppler coupling effect for the FMCW signal. Finally, the effect on the signal level due to the multipath propagation is discussed.

References

[1] Karnfelt, Camilla, et al. "77 GHz ACC Radar Simulation Platform." *2009 9th International Conference on Intelligent Transport Systems Telecommunications, (ITST), IEEE, 2009* , pp. 209–14. DOI.org (Crossref), https://doi.org/10.1109/ITST.2009.5399354.

[2] Rohling, H., and M. M. Meinecke. "Waveform Design Principles for Automotive Radar Systems." *2001 CIE International Conference on Radar Proceedings (Cat No.01TH8559)* , IEEE, 2001, pp. 1–4. DOI.org (Crossref), https://doi.org/10.1109/ICR.2001.984612.

Radar Signal Simulation and Processing for Automated Driving

This example shows how to model the hardware, signal processing, and propagation environment of an automotive radar. First you model a highway scenario using Automated Driving Toolbox™. Then, you develop a model of the radar transmit and receive hardware, signal processing and tracker using Radar Toolbox™. Finally, you simulate multipath propagation effects on the radar model.

Introduction

You can model vehicle motion by using the `drivingScenario` object from Automated Driving Toolbox. The vehicle ground truth can then be used as an input to the radar model to generate synthetic sensor detections. For an example of this workflow, see "Simulate Radar Ghosts Due to Multipath Return" on page 1-45. The automotive radar used in this example uses a statistical model that is parameterized according to high-level radar specifications. The generic radar architecture modeled in this example does not include specific antenna configurations, waveforms, or unique channel propagation characteristics. When designing an automotive radar, or when the specific architecture of a radar is known, use a radar model that includes this additional information.

Radar Toolbox enables you to evaluate different radar architectures. You can explore different transmit and receive array configurations, waveforms, and signal processing chains. You can also evaluate your designs against different channel models to assess their robustness to different environmental conditions. This modeling helps you to identify the specific design that best fits your application requirements.

In this example, you learn how to define a radar model from a set of system requirements for a long-range radar. You then simulate a driving scenario to generate detections from your radar model. A tracker is used to process these detections to generate precise estimates of the position and velocity of the vehicles detected by your automotive radar.

Calculate Radar Parameters from Long-Range Radar Requirements

The radar parameters are defined for the frequency-modulated continuous wave (FMCW) waveform, as described in the example "Automotive Adaptive Cruise Control Using FMCW Technology" on page 1-148. The radar operates at a center frequency of 77 GHz. This frequency is commonly used by automotive radars. For long-range operation, the radar must detect vehicles at a maximum range of 250-300 meters in front of the ego vehicle. The radar is required to resolve objects in range that are at least 1 meter apart. Because this is a forward-facing radar application, the radar also needs to handle targets with large closing speeds as high as 230 km/hr.

The radar is designed to use an FMCW waveform. These waveforms are common in automotive applications because they enable range and Doppler estimation through computationally efficient FFT operations. For illustration purpose, in this example, configure the radar to a maximum range of 100 meters.

```
% Set random number generator for repeatable results
rng(2017);

% Compute hardware parameters from specified long-range requirements
fc = 77e9;                              % Center frequency (Hz)
c = physconst('LightSpeed');            % Speed of light in air (m/s)
lambda = freq2wavelen(fc,c);                  % Wavelength (m)

% Set the chirp duration to be 5 times the max range requirement
```

```matlab
rangeMax = 100;                                % Maximum range (m)
tm = 5*range2time(rangeMax,c);                 % Chirp duration (s)

% Determine the waveform bandwidth from the required range resolution
rangeRes = 1;                                  % Desired range resolution (m)
bw = rangeres2bw(rangeRes,c);                  % Corresponding bandwidth (Hz)

% Set the sampling rate to satisfy both the range and velocity requirements
% for the radar
sweepSlope = bw/tm;                                % FMCW sweep slope (Hz/s)
fbeatMax = range2beat(rangeMax,sweepSlope,c); % Maximum beat frequency (Hz)

vMax = 230*1000/3600;                          % Maximum Velocity of cars (m/s)
fdopMax = speed2dop(2*vMax,lambda);        % Maximum Doppler shift (Hz)

fifMax = fbeatMax+fdopMax;    % Maximum received IF (Hz)
fs = max(2*fifMax,bw);        % Sampling rate (Hz)

% Configure the FMCW waveform using the waveform parameters derived from
% the long-range requirements
waveform = phased.FMCWWaveform('SweepTime',tm,'SweepBandwidth',bw,...
    'SampleRate',fs,'SweepDirection','Up');
if strcmp(waveform.SweepDirection,'Down')
    sweepSlope = -sweepSlope;
end
Nsweep = 192;
sig = waveform();
```

Model Automotive Radar Transceiver

The radar uses an isotropic element to transmit and a uniform linear array (ULA) to receive the radar waveforms. Using a linear array enables the radar to estimate the azimuthal direction of the reflected energy received from the target vehicles. The long-range radar needs to detect targets across a coverage area that spans 15 degrees in front of the ego vehicle. A six-element receive array satisfies this requirement by providing a 17-degree half-power beamwidth. On transmit, the radar uses only a single array element, enabling it to cover a larger area than on receive.

```matlab
% Model the antenna element
antElmnt = phased.IsotropicAntennaElement('BackBaffled',true);

% Construct the receive array
Ne = 6;
rxArray = phased.ULA('Element',antElmnt,'NumElements',Ne,...
    'ElementSpacing',lambda/2);

% Half-power beamwidth of the receive array
hpbw = beamwidth(rxArray,fc,'PropagationSpeed',c)
```

```
hpbw = 17.1800
```

Model the radar transmitter for a single transmit channel, and model a receiver preamplifier for each receive channel, using the parameters defined in the example "Automotive Adaptive Cruise Control Using FMCW Technology" on page 1-148.

```matlab
antAperture = 6.06e-4;                         % Antenna aperture (m^2)
antGain = aperture2gain(antAperture,lambda); % Antenna gain (dB)

txPkPower = db2pow(5)*1e-3;                     % Tx peak power (W)
```

```
txGain = antGain;                                    % Tx antenna gain (dB)

rxGain = antGain;                                    % Rx antenna gain (dB)
rxNF = 4.5;                                          % Receiver noise figure (dB)

% Waveform transmitter
transmitter = phased.Transmitter('PeakPower',txPkPower,'Gain',txGain);

% Radiator for single transmit element
radiator = phased.Radiator('Sensor',antElmnt,'OperatingFrequency',fc);

% Collector for receive array
collector = phased.Collector('Sensor',rxArray,'OperatingFrequency',fc);

% Receiver preamplifier
receiver = phased.ReceiverPreamp('Gain',rxGain,'NoiseFigure',rxNF,...
    'SampleRate',fs);

% Define radar
radar = radarTransceiver('Waveform',waveform,'Transmitter',transmitter,...
    'TransmitAntenna',radiator,'ReceiveAntenna',collector,'Receiver',receiver);
```

Define Radar Signal Processing Chain

The radar collects multiple sweeps of the waveform on each of the linear phased array antenna elements. These collected sweeps form a data cube, which is defined in "Radar Data Cube". These sweeps are coherently processed along the fast- and slow-time dimensions of the data cube to estimate the range and Doppler of the vehicles.

Estimate the direction-of-arrival of the received signals using a root MUSIC estimator. A beamscan is also used for illustrative purposes to help visualize the spatial distribution of the received signal energy.

```
% Direction-of-arrival estimator for linear phased array signals
doaest = phased.RootMUSICEstimator(...
    'SensorArray',rxArray,...
    'PropagationSpeed',c,'OperatingFrequency',fc,...
    'NumSignalsSource','Property','NumSignals',1);

% Scan beams in front of ego vehicle for range-angle image display
angscan = -80:80;
beamscan = phased.PhaseShiftBeamformer('Direction',[angscan;0*angscan],...
    'SensorArray',rxArray,'OperatingFrequency',fc);

% Form forward-facing beam to detect objects in front of the ego vehicle
beamformer = phased.PhaseShiftBeamformer('SensorArray',rxArray,...
    'PropagationSpeed',c,'OperatingFrequency',fc,'Direction',[0;0]);
```

Use the `phased.RangeDopplerResponse` object to perform the range and Doppler processing on the radar data cubes. Use a Hanning window to suppress the large sidelobes produced by the vehicles when they are close to the radar.

```
Nft = waveform.SweepTime*waveform.SampleRate; % Number of fast-time samples
Nst = Nsweep;                                 % Number of slow-time samples
Nr = 2^nextpow2(Nft);                         % Number of range samples
Nd = 2^nextpow2(Nst);                         % Number of Doppler samples
rngdopresp = phased.RangeDopplerResponse('RangeMethod','FFT',...
    'DopplerOutput','Speed','SweepSlope',sweepSlope,...
```

```
    'RangeFFTLengthSource','Property','RangeFFTLength',Nr,...
    'RangeWindow','Hann',...
    'DopplerFFTLengthSource','Property','DopplerFFTLength',Nd,...
    'DopplerWindow','Hann',...
    'PropagationSpeed',c,'OperatingFrequency',fc,'SampleRate',fs);
```

Identify detections in the processed range and Doppler data by using a constant false alarm rate (CFAR) detector. CFAR detectors estimate the background noise level of the received radar data. Detections are found at locations where the signal power exceeds the estimated noise floor by a certain threshold. Low threshold values result in a higher number of reported false detections due to environmental noise. Increasing the threshold produces fewer false detections, but also reduces the probability of detection of an actual target in the scenario. For more information on CFAR detection, see the example "Constant False Alarm Rate (CFAR) Detection".

```
% Guard cell and training regions for range dimension
nGuardRng = 4;
nTrainRng = 4;
nCUTRng = 1+nGuardRng+nTrainRng;

% Guard cell and training regions for Doppler dimension
dopOver = round(Nd/Nsweep);
nGuardDop = 4*dopOver;
nTrainDop = 4*dopOver;
nCUTDop = 1+nGuardDop+nTrainDop;

cfar = phased.CFARDetector2D('GuardBandSize',[nGuardRng nGuardDop],...
    'TrainingBandSize',[nTrainRng nTrainDop],...
    'ThresholdFactor','Custom','CustomThresholdFactor',db2pow(13),...
    'NoisePowerOutputPort',true,'OutputFormat','Detection index');

% Perform CFAR processing over all of the range and Doppler cells
freqs = ((0:Nr-1)'/Nr-0.5)*fs;
rnggrid = beat2range(freqs,sweepSlope);
iRngCUT = find(rnggrid>0);
iRngCUT = iRngCUT((iRngCUT>=nCUTRng)&(iRngCUT<=Nr-nCUTRng+1));
iDopCUT = nCUTDop:(Nd-nCUTDop+1);
[iRng,iDop] = meshgrid(iRngCUT,iDopCUT);
idxCFAR = [iRng(:) iDop(:)]';

% Perform clustering algorithm to group detections
clusterer = clusterDBSCAN('Epsilon',2);
```

The `phased.RangeEstimator` and `phased.DopplerEstimator` objects convert the locations of the detections found in the range-Doppler data into measurements and their corresponding measurement variances. These estimators fit quadratic curves to the range-Doppler data to estimate the peak location of each detection. The resulting measurement resolutions are a fraction of the range and Doppler sampling of the data.

The root-mean-square (RMS) range resolution of the transmitted waveform is needed to compute the variance of the range measurements. The Rayleigh range resolution for the long-range radar was defined previously as 1 meter. The Rayleigh resolution is the minimum separation at which two unique targets can be resolved. This value defines the distance between range resolution cells for the radar. However, the variance of the target within a resolution cell is determined by the RMS resolution of the waveform. For an LFM chirp waveform, the relationship between the Rayleigh resolution and the RMS resolution is given by [1].

$$\sigma_{RMS} = \sqrt{12}\Delta_{Rayleigh}$$

where σ_{RMS} is the RMS range resolution and $\Delta_{Rayleigh}$ is the Rayleigh range resolution.

The variance of the Doppler measurements depends on the number of sweeps processed.

Now, create the range and Doppler estimation objects using the parameters previously defined.

```
rmsRng = sqrt(12)*rangeRes;
rngestimator = phased.RangeEstimator('ClusterInputPort',true,...
    'VarianceOutputPort',true,'NoisePowerSource','Input port',...
    'RMSResolution',rmsRng);

dopestimator = phased.DopplerEstimator('ClusterInputPort',true,...
    'VarianceOutputPort',true,'NoisePowerSource','Input port',...
    'NumPulses',Nsweep);
```

To further improve the precision of the estimated vehicle locations, pass the radar detections to a tracker. Configure the tracker to use an extended Kalman filter (EKF), which converts the spherical radar measurements into the Cartesian coordinate frame of the ego vehicle. Also configure the tracker to use constant velocity dynamics for the detected vehicles. By comparing vehicle detections over multiple measurement time intervals, the tracker further improves the accuracy of the vehicle positions and provides vehicle velocity estimates.

```
tracker = radarTracker('FilterInitializationFcn',@initcvekf,...
    'AssignmentThreshold',50);
```

Model Free Space Propagation Channel

Use the free space channel to model the propagation of the transmitted and received radar signals.

In a free space model, the radar energy propagates along a direct line-of-sight between the radar and the target vehicles, as shown in the following illustration.

Simulate the Driving Scenario

Create a highway driving scenario with three vehicles traveling in the vicinity of the ego vehicle. The vehicles are modeled as cuboids and have different velocities and positions defined in the driving scenario. The ego vehicle is moving with a velocity of 80 km/hr and the other three cars are moving at 110 km/hr, 100 km/hr, and 130 km/hr, respectively. For details on modeling a driving scenario see the example "Create Actor and Vehicle Trajectories Programmatically" (Automated Driving Toolbox). The radar sensor is mounted on the front of the ego vehicle.

To create the driving scenario, use the `helperAutoDrivingRadarSigProc` function. To examine the contents of this function, use the `edit('helperAutoDrivingRadarSigProc')` command.

```
% Create driving scenario
[scenario,egoCar,radarParams] = ...
    helperAutoDrivingRadarSigProc('Setup Scenario',c,fc);
```

The following loop uses the `drivingScenario` object to advance the vehicles in the scenario. At every simulation time step, a radar data cube is assembled by collecting 192 sweeps of the radar waveform. The assembled data cube is then processed in range and Doppler. The range and Doppler processed data is then beamformed, and CFAR detection is performed on the beamformed data. Range, radial speed, and direction of arrival measurements are estimated for the CFAR detections. These detections are then assembled into `objectDetection` objects, which are then processed by the `radarTracker` object.

```
% Initialize display for driving scenario example
helperAutoDrivingRadarSigProc('Initialize Display',egoCar,radarParams,...
    rxArray,fc,vMax,rangeMax);

tgtProfiles = actorProfiles(scenario);
tgtProfiles = tgtProfiles(2:end);
tgtHeight = [tgtProfiles.Height];

% Run the simulation loop
sweepTime = waveform.SweepTime;
while advance(scenario)

    % Get the current scenario time
    time = scenario.SimulationTime;

    % Get current target poses in ego vehicle's reference frame
    tgtPoses = targetPoses(egoCar);
    tgtPos = reshape([tgtPoses.Position],3,[]);
    % Position point targets at half of each target's height
    tgtPos(3,:) = tgtPos(3,:)+0.5*tgtHeight;
    tgtVel = reshape([tgtPoses.Velocity],3,[]);

    % Assemble data cube at current scenario time
    Xcube = zeros(Nft,Ne,Nsweep);
    for m = 1:Nsweep

        ntgt = size(tgtPos,2);
        tgtStruct = struct('Position',mat2cell(tgtPos(:).',1,repmat(3,1,ntgt)),...
            'Velocity',mat2cell(tgtVel(:).',1,repmat(3,1,ntgt)),...
            'Signature',{rcsSignature,rcsSignature,rcsSignature});
        rxsig = radar(tgtStruct,time+(m-1)*sweepTime);
        % Dechirp the received signal
        rxsig = dechirp(rxsig,sig);

        % Save sweep to data cube
        Xcube(:,:,m) = rxsig;

        % Move targets forward in time for next sweep
        tgtPos = tgtPos+tgtVel*sweepTime;
    end

    % Calculate the range-Doppler response
    [Xrngdop,rnggrid,dopgrid] = rngdopresp(Xcube);

    % Beamform received data
```

```matlab
        Xbf = permute(Xrngdop,[1 3 2]);
        Xbf = reshape(Xbf,Nr*Nd,Ne);
        Xbf = beamformer(Xbf);
        Xbf = reshape(Xbf,Nr,Nd);

        % Detect targets
        Xpow = abs(Xbf).^2;
        [detidx,noisepwr] = cfar(Xpow,idxCFAR);

        % Cluster detections
        [~,clusterIDs] = clusterer(detidx.');

        % Estimate azimuth, range, and radial speed measurements
        [azest,azvar,snrdB] = ...
            helperAutoDrivingRadarSigProc('Estimate Angle',doaest,...
            conj(Xrngdop),Xbf,detidx,noisepwr,clusterIDs);
        azvar = azvar+radarParams.RMSBias(1)^2;

        [rngest,rngvar] = rngestimator(Xbf,rnggrid,detidx,noisepwr,clusterIDs);
        rngvar = rngvar+radarParams.RMSBias(2)^2;

        [rsest,rsvar] = dopestimator(Xbf,dopgrid,detidx,noisepwr,clusterIDs);

        % Convert radial speed to range rate for use by the tracker
        rrest = -rsest;
        rrvar = rsvar;
        rrvar = rrvar+radarParams.RMSBias(3)^2;

        % Assemble object detections for use by tracker
        numDets = numel(rngest);
        dets = cell(numDets,1);
        for iDet = 1:numDets
            dets{iDet} = objectDetection(time,...
                [azest(iDet) rngest(iDet) rrest(iDet)]',...
                'MeasurementNoise',diag([azvar(iDet) rngvar(iDet) rrvar(iDet)]),...
                'MeasurementParameters',{radarParams},...
                'ObjectAttributes',{struct('SNR',snrdB(iDet))});
        end

        % Track detections
        tracks = tracker(dets,time);

        % Update displays
        helperAutoDrivingRadarSigProc('Update Display',egoCar,dets,tracks,...
            dopgrid,rnggrid,Xbf,beamscan,Xrngdop);

        % Collect free space channel metrics
        metricsFS = helperAutoDrivingRadarSigProc('Collect Metrics',...
            radarParams,tgtPos,tgtVel,dets);
    end
```
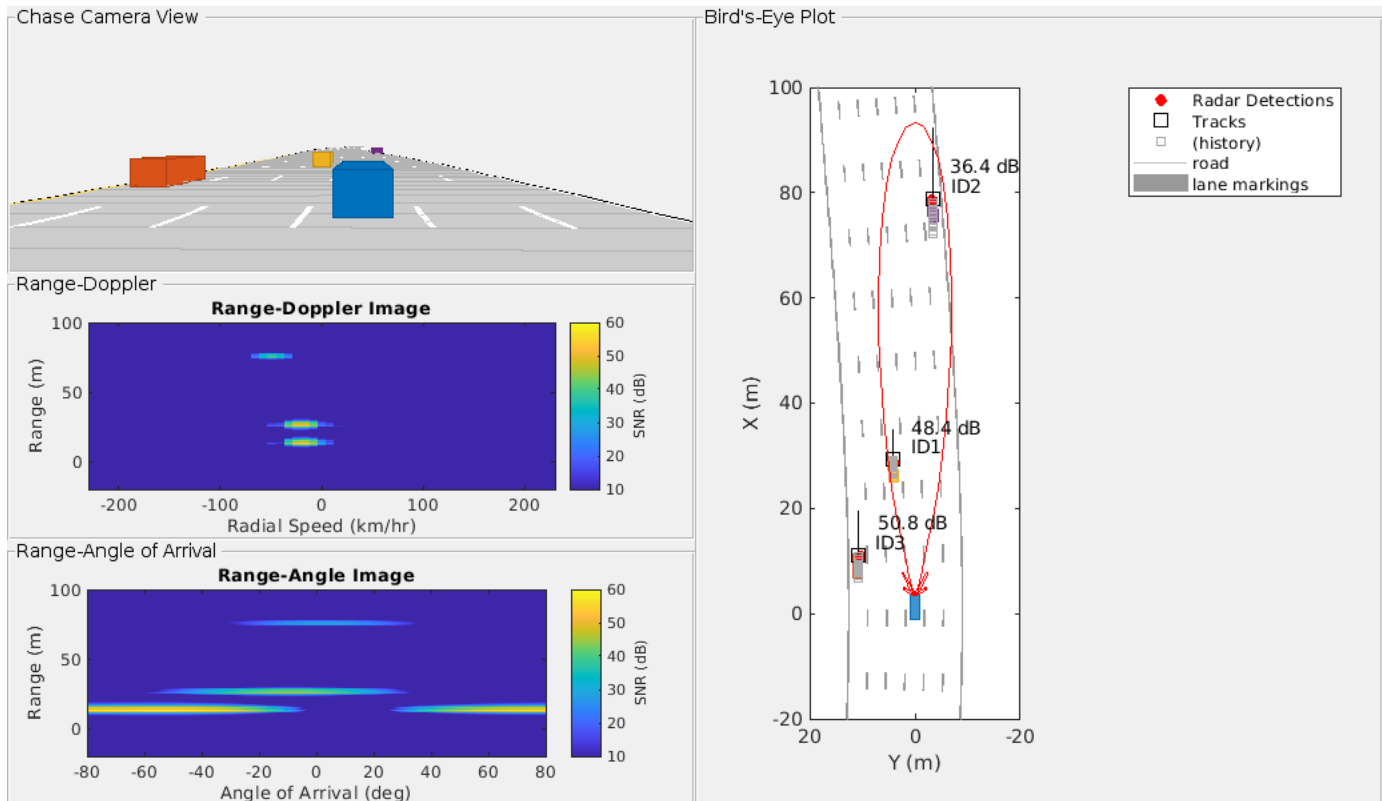
The previous figure shows the radar detections and tracks for the three target vehicles at 1.1 seconds of simulation time. The plot on the upper-left side shows the chase camera view of the driving scenario from the perspective of the ego vehicle (shown in blue). For reference, the ego vehicle is traveling at 80 km/hr and the other three cars are traveling at 110 km/hr (orange car), 100 km/hr (yellow car), and 130 km/hr (purple car).

The right side of the figure shows the bird's-eye plot, which presents a top down perspective of the scenario. All of the vehicles, detections, and tracks are shown in the coordinate reference frame of the ego vehicle. The estimated signal-to-noise ratio (SNR) for each radar measurement is printed next to each detection. The vehicle location estimated by the tracker is shown in the plot using black squares with text next to them indicating the ID of each track. The velocity for each vehicle estimated by the tracker is shown as a black line pointing in the direction of the velocity of the vehicle. The length of the line corresponds to the estimated speed, with longer lines denoting vehicles with higher speeds relative to the ego vehicle. The track of the purple car (ID2) has the longest line while the track of the yellow car (ID1) has the shortest line. The tracked speeds are consistent with the modeled vehicle speeds previously listed.

The two plots on the lower-left side show the radar images generated by the signal processing. The upper plot shows how the received radar echoes from the target vehicles are distributed in range and radial speed. Here, all three vehicles are observed. The measured radial speeds correspond to the velocities estimated by the tracker, as shown in the bird's-eye plot. The lower plot shows how the received target echoes are spatially distributed in range and angle. Again, all three targets are present, and their locations match what is shown in the bird's-eye plot.

Due to its close proximity to the radar, the orange car can still be detected despite the large beamforming losses due to its position well outside of the 3 dB beamwidth of the beam. These detections have generated a track (ID3) for the orange car.

Model a Multipath Channel

The previous driving scenario simulation used free space propagation. This is a simple model that models only direct line-of-sight propagation between the radar and each of the targets. In reality, the radar signal propagation is much more complex, involving reflections from multiple obstacles before reaching each target and returning back to the radar. This phenomenon is known as *multipath propagation*. The following illustration shows one such case of multipath propagation, where the signal impinging the target is coming from two directions: line-of-sight and a single bounce from the road surface.
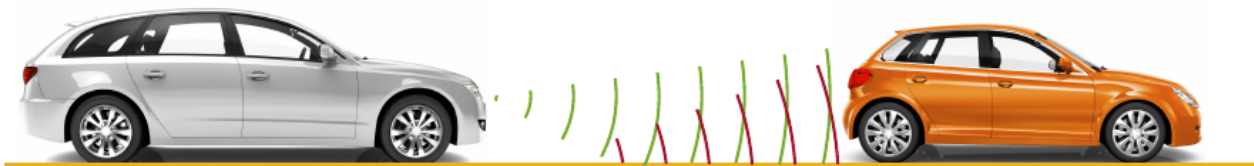
The overall effect of multipath propagation is that the received radar echoes can interfere constructively and destructively. This constructive and destructive interference results from path length differences between the various signal propagation paths. As the distance between the radar and the vehicles changes, these path length differences also change. When the differences between these paths result in echoes received by the radar that are almost 180 degrees out of phase, the echoes destructively combine, and the radar makes no detection for that range.

Replace the free space channel model with a two-ray channel model to demonstrate the propagation environment shown in the previous illustration. Reuse the remaining parameters in the driving scenario and radar model, and run the simulation again.

```
% Reset the driving scenario
[scenario,egoCar,radarParams,pointTgts] = ...
    helperAutoDrivingRadarSigProc('Setup Scenario',c,fc);

% Run the simulation again, now using the two-ray channel model
metrics2Ray = helperAutoDrivingRadarSigProc('Two Ray Simulation',...
    c,fc,rangeMax,vMax,Nsweep,...               % Waveform parameters
    rngdopresp,beamformer,cfar,idxCFAR,clusterer,...          % Signal processing
    rngestimator,dopestimator,doaest,beamscan,tracker,...   % Estimation
    radar,sig);                          % Hardware models
```
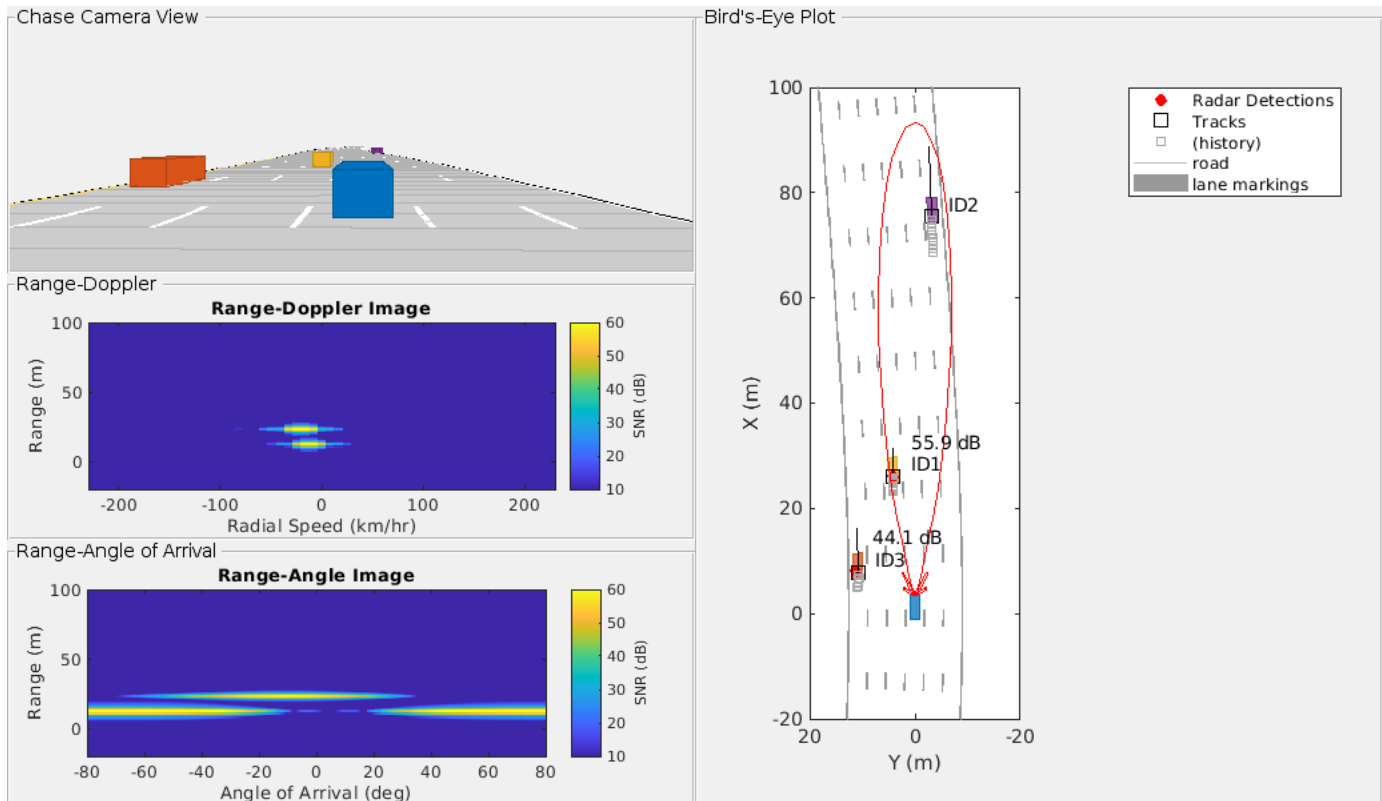
The previous figure shows the chase plot, bird's-eye plot, and radar images at 1.1 seconds of simulation time, just as was shown for the free space channel propagation scenario. Comparing these two figures, observe that for the two-ray channel, no detection is present for the purple car at this simulation time. This detection loss is because the path length differences for this car are destructively interfering at this range, resulting in a total loss of detection.

Plot the SNR estimates generated from the CFAR processing against the range estimates of the purple car from the free space and two-ray channel simulations.

```
helperAutoDrivingRadarSigProc('Plot Channels',metricsFS,metrics2Ray);
```

Propagation Channels
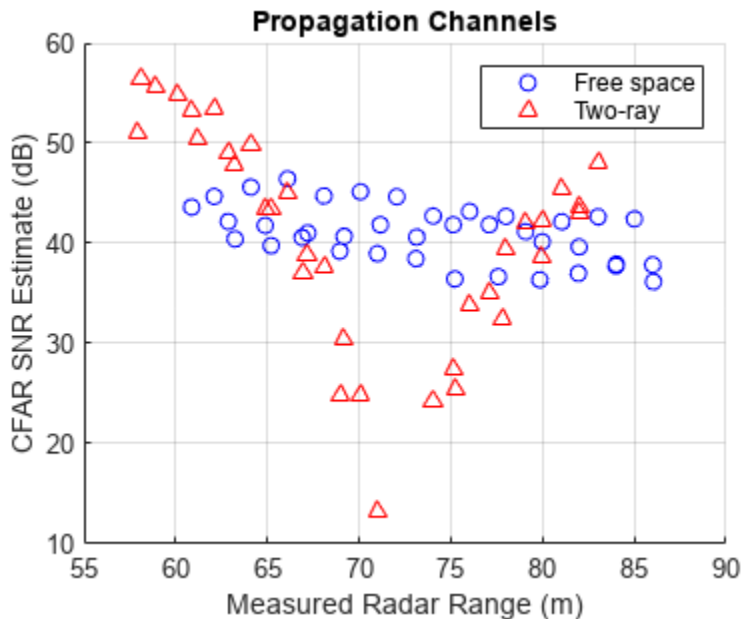
As the car approaches a range of 72 meters from the radar, a large loss in the estimated SNR from the two-ray channel is observed with respect to the free space channel. It is near this range that the multipath interference combines destructively, resulting in a loss in signal detections. However, observe that the tracker is able to coast the track during these times of signal loss and provide a predicted position and velocity for the purple car.

Summary

This example shows how to model the hardware and signal processing of an automotive radar using Radar Toolbox. You also learn how to integrate this radar model with the Automated Driving Toolbox driving scenario simulation. First you generate synthetic radar detections. Then you process these detections further by using a tracker to generate precise position and velocity estimates in the coordinate frame of the ego vehicle. Finally, you learn how to simulate multipath propagation effects.

The workflow presented in this example enables you to understand how your radar architecture design decisions impact higher-level system requirements. Using this workflow enables you select a radar design that satisfies your unique application requirements.

Reference

[1] Richards, Mark. *Fundamentals of Radar Signal Processing*. New York: McGraw Hill, 2005.

Adaptive Tracking of Maneuvering Targets with Managed Radar

This example shows how to use radar resource management to efficiently track multiple maneuvering targets. Tracking maneuvering targets requires the radar to revisit the targets more frequently than tracking non-maneuvering targets. An interacting multiple model (IMM) filter estimates when the target is maneuvering. This estimate helps to manage the radar revisit time and therefore enhances the tracking. This example uses the Radar Toolbox™ for the radar model and Sensor Fusion and Tracking Toolbox™ for the tracking.

Introduction

Multifunction radars can search for targets, confirm new tracks, and revisit tracks to update the state. To perform these functions, a multifunction radar is often managed by a resource manager that creates radar tasks for search, confirmation, and tracking. These tasks are scheduled according to priority and time so that, at each time step, the multifunction radar can point its beam in a desired direction. The "Search and Track Scheduling for Multifunction Phased Array Radar" on page 1-188 example shows a multifunction phased-array radar managed with a resource manager.

In this example, we extend the "Search and Track Scheduling for Multifunction Phased Array Radar" on page 1-188 example to the case of multiple maneuvering targets. There are two conflicting requirements for a radar used to track maneuvering targets:

1   The number of targets and their initial location are typically not known in advance. Therefore, the radar must continuously search the area of interest to find the targets. Also, the radar needs to detect and establish a track on each target as soon as it enters the radar coverage area.

2   The time period of target maneuvering is unknown in advance. If it is known that targets are not maneuvering, the radar can revisit the targets infrequently. However, since the maneuver start and end times are unknown, the radar must revisit each track frequently enough to be able to recognize when the maneuver starts and ends.

The radar must balance between providing enough beams centered on the tracked targets and leaving enough time to search for new targets. One approach is to simply define a revisit rate on each tracked target regardless of its maneuvering status and leave the remaining time for new target searching. This radar management scheme is sometimes referred to as *Active Tracking* [1]. As more targets become tracked, the radar can either perform fewer search tasks or it can track each target less frequently. Clearly, if the number of targets is large, the radar can be overwhelmed.

Active tracking treats all the tracks in the same way, which makes it a *mode*-based resource management algorithm. A more sophisticated way to manage the radar is based on the properties of each track. For example, use track properties such as the size of state uncertainty covariance, whether the track is maneuvering, and how fast it is moving towards an asset the radar site protects. When such properties are used, the radar resource management is referred to as *Adaptive Tracking* [1].

In this example, you compare the results of Active Tracking and Adaptive Tracking when the radar adapts based on estimated track maneuver.

Define Scenario and Radar Model

You define a scenario and a radar with an update rate of 20 Hz, which means that the radar has 20 beams per second allocated for either search, confirmation, or tracking. You load the benchmark trajectories used in the "Benchmark Trajectories for Multi-Object Tracking" (Sensor Fusion and

Tracking Toolbox) example. There are six benchmark trajectories and you define a trajectory for each one. The six platforms in the figure follow non-maneuvering legs interspersed with maneuvering legs. You can view the trajectories in the figure.

```matlab
% Create scenario
updateRate = 20;
scenario = trackingScenario('UpdateRate',updateRate);
% Add the benchmark trajectories
load('BenchmarkTrajectories.mat','-mat');
platform(scenario,'Trajectory',v1Trajectory);
platform(scenario,'Trajectory',v2Trajectory);
platform(scenario,'Trajectory',v3Trajectory);
platform(scenario,'Trajectory',v4Trajectory);
platform(scenario,'Trajectory',v5Trajectory);
platform(scenario,'Trajectory',v6Trajectory);

% Create visualization
f = figure;
mp = uipanel('Parent',f,'Title','Theater Plot','FontSize',12,...
    'BackgroundColor','white','Position',[.01 .25 .98 .73]);
tax = axes(mp,'ZDir','reverse');

% Visualize scenario
thp = theaterPlot('Parent',tax,'AxesUnits',["km","km","km"],'XLimits',[0 85000],'YLimits',[-45000
plp = platformPlotter(thp, 'DisplayName', 'Platforms');
pap = trajectoryPlotter(thp, 'DisplayName', 'Trajectories', 'LineWidth', 1);
dtp = detectionPlotter(thp, 'DisplayName', 'Detections');
cvp = coveragePlotter(thp, 'DisplayName', 'Radar Coverage');
trp = trackPlotter(thp, 'DisplayName', 'Tracks', 'ConnectHistory', 'on', 'ColorizeHistory', 'on'
numPlatforms = numel(scenario.Platforms);
trajectoryPositions = cell(1,numPlatforms);
for i = 1:numPlatforms
    trajectoryPositions{i} = lookupPose(scenario.Platforms{i}.Trajectory,(0:0.1:185));
end
plotTrajectory(pap, trajectoryPositions);
view(tax,3)
```
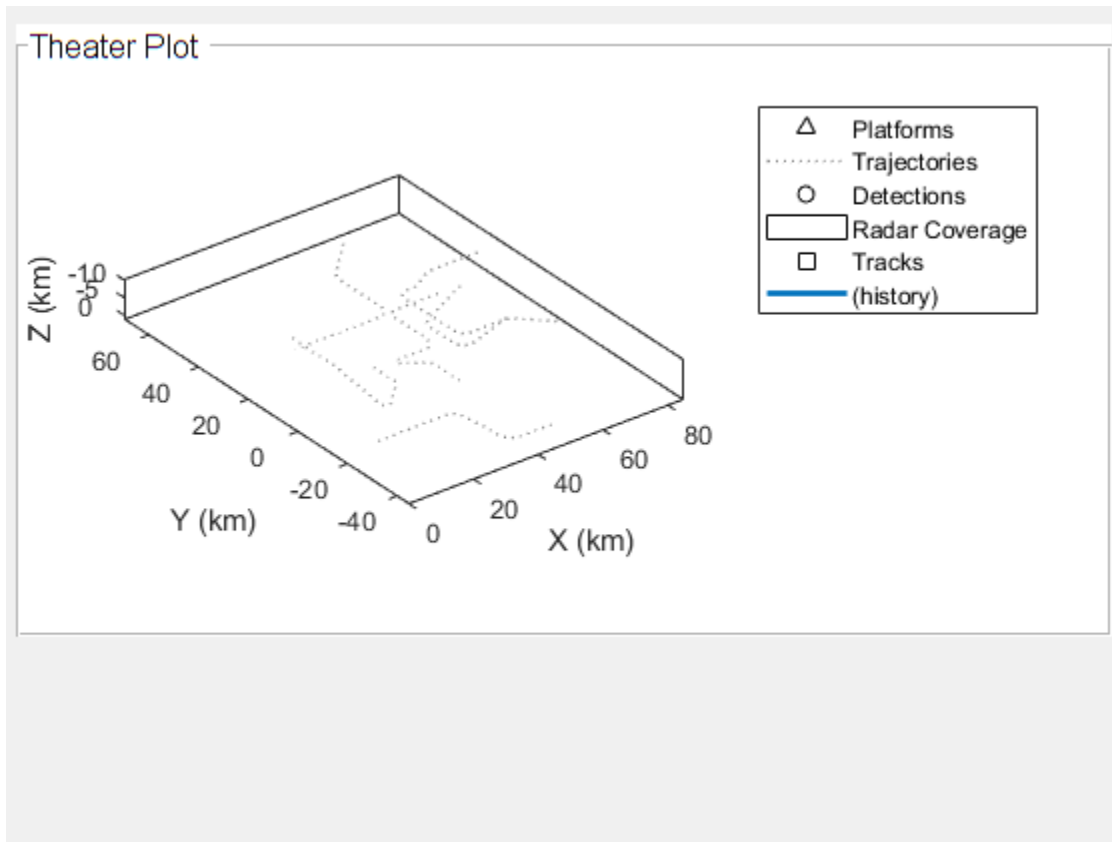
A probabilistic radar model is defined using the `radarDataGenerator` System object™. Setting the `'ScanMode'` property of this object to `'Custom'` allows the resource manager to control the radar look angle. This enables scheduling of the radar for searching, confirming, and tracking targets. The radar is mounted on a new platform in the scenario.

```
radar = radarDataGenerator(1, ...
    'ScanMode', 'Custom', ...
    'UpdateRate', updateRate, ...
    'MountingLocation', [0 0 -15], ...
    'AzimuthResolution', 1.5, ...
    'ElevationResolution', 10, ...
    'HasElevation', true, ...
    'DetectionCoordinates', 'Sensor spherical');
platform(scenario,'Position',[0 0 0],'Sensors',radar);
```

Define Tracker

After the radar detects objects, it feeds the detections to a tracker, which performs several operations. The tracker maintains a list of tracks that are estimates of target states in the area of interest. If a detection cannot be assigned to any track already maintained by the tracker, the tracker initiates a new track. In most cases, whether the new track represents a true target or false target is unclear. At first, a track is created with a tentative status. If enough detections are obtained, the track becomes confirmed. Similarly, if no detections are assigned to a track, the track is coasted (predicted without correction). If the track has a few missed updates, the tracker deletes the track.

In this example, you use a tracker that associates the detections to the tracks using a global nearest neighbor (GNN) algorithm. To track the maneuvering targets, you define a

FilterInitializationFcn function that initializes an IMM filter. The initMPARIMM function uses two motion models: a constant-velocity model and a constant-turn rate model. The trackingIMM (Sensor Fusion and Tracking Toolbox) filter is responsible for estimating the probability of each model, which you can access from its ModelProbabilities property. In this example, you classify a target as maneuvering when the probability of the constant-turn rate model is higher than 0.6.

```matlab
tracker = trackerGNN('FilterInitializationFcn',@initMPARIMM,...
    'ConfirmationThreshold',[2 3], 'DeletionThreshold',[5 5],...
    'HasDetectableTrackIDsInput',true,'AssignmentThreshold',150,...
    'MaxNumTracks',10,'MaxNumSensors',1);
posSelector = [1 0 0 0 0 0; 0 0 1 0 0 0; 0 0 0 0 1 0];
```

Radar Resource Management

This section only briefly outlines the radar resource management. For more details, see the "Adaptive Tracking of Maneuvering Targets with Managed Radar" on page 1-171 example.

Search Tasks

In this example, you assign search tasks deterministically. A raster scan is used to cover the desired airspace. The azimuth scanning limits are set to [-90 60] degrees and the elevation limits to [-9.9 0] degrees. If no other tasks exist, the radar scans the space one angular cell at a time. The size of an angular cell is determined by the radar's AzimuthResolution and ElevationResolution properties. Negative elevation angles mean that the radar points the beam from the horizon up.

```matlab
AzimuthLimits = [-90 60];
ElevationLimits = [-9.9 0];
azscanspan   = diff(AzimuthLimits);
numazscan    = floor(azscanspan/radar.AzimuthResolution)+1;
azscanangles = linspace(AzimuthLimits(1),AzimuthLimits(2),numazscan)+radar.MountingAngles(1);
elscanspan   = diff(ElevationLimits);
numelscan    = floor(elscanspan/radar.ElevationResolution)+1;
elscanangles = linspace(ElevationLimits(1),ElevationLimits(2),numelscan)+radar.MountingAngles(2)
[elscangrid,azscangrid] = meshgrid(elscanangles,azscanangles);
scanangles   = [azscangrid(:) elscangrid(:)].';
searchq = struct('JobType','Search','BeamDirection',num2cell(scanangles,1),...
    'Priority',1000,'WaveformIndex',1);
current_search_idx = 1;
```

Track Tasks

Unlike search tasks, track tasks cannot be planned in advance. Instead, the resource manager creates confirmation and tracking tasks based on the changing scenario. The main difference in this example from the "Adaptive Tracking of Maneuvering Targets with Managed Radar" on page 1-171 example is that the JobType for each track task can be either "TrackNonManeuvering" or "TrackManeuvering". The distinction between the two types of tracking tasks enables you to schedule tasks for each type of track at different revisit rates, making it an adaptive tracking algorithm. Similar to search tasks, tracking tasks are also managed in a job queue.

```matlab
trackq = repmat(struct('JobType',[],'BeamDirection',[],'Priority',3000,'WaveformIndex',[],...
    'Time',[],'Range',[],'TrackID',[]), 10, 1);
num_trackq_items = 0;
```

Group search and tracking queues together in a structure for easier reference in the simulation loop.

```matlab
jobq.SearchQueue  = searchq;
jobq.SearchIndex  = current_search_idx;
```

```
jobq.TrackQueue    = trackq;
jobq.NumTrackJobs = num_trackq_items;
jobq.PositionSelector = posSelector;
% Keep a reset state of jobq
resetJobQ = jobq;
```

Task Scheduling

In this example, for simplicity, the multifunction radar executes only one type of job within a small time period, often referred to as a dwell, but can switch tasks at the beginning of each dwell. For each dwell, the radar looks at all tasks that are due for execution and picks a confirmation or track task if its time to run has come. Otherwise, the radar picks a search task. To control the time to run tasks, you set the `managerPreferences` structure defined below. The highest revisit rate, which is equal to the radar update rate, is given to the `confirm` task to guarantee that a confirmation beam follows every new tentative track that exists. Similarly, you can control the revisit rate for non-maneuvering and maneuvering targets. In this case, you choose the values of 0.8 Hz and 4 Hz for non-maneuvering and maneuvering targets, respectively. Since there are 6 targets, the resource manager will use $6 \cdot 0.8 \approx 5$ updates per second on tracking targets if all of them are not maneuvering. Given the radar update rate is 20 Hz, the radar manager will perform approximately one target update in every four updates. Thus, the radar will be spending about 75% of the time in the search mode and 25% of the time in the tracking mode. When new tracks are initialized and when the tracker considers that the targets are maneuvering, the resource manager allocates more tracking beams at the expense of search beams.

The Analyze Results section shows the results of other options.

```
managerPreferences = struct(...
    'Type', {"Search","Confirm","TrackNonManeuvering","TrackManeuvering"}, ...
    'RevisitRate', {0, updateRate, 0.8, 4}, ...
    'Priority', {1000, 3000, 1500, 2500});
```

Run the Scenario

During the simulation, the radar beam is depicted by blue or purple colors representing search and track-related beams, respectively. You can also see the distribution of tasks between search and specific tracks for the last second of simulation using the Resource Allocation in the Last Second panel at the bottom of the figure. In the Theater Plot part of the figure, you can see the history of each track and compare it to the trajectory.

```
% Create a radar resource allocation display
rp = uipanel('Parent',f,'Title','Resource Allocation in the Last Second','FontSize',12,...
    'BackgroundColor','white','Position',[.01 0.01 0.98 0.23]);
rax = axes(rp);

% Run the scenario
allocationType = helperAdaptiveTrackingSim(scenario, thp, rax, tracker, resetJobQ, managerPrefere
```
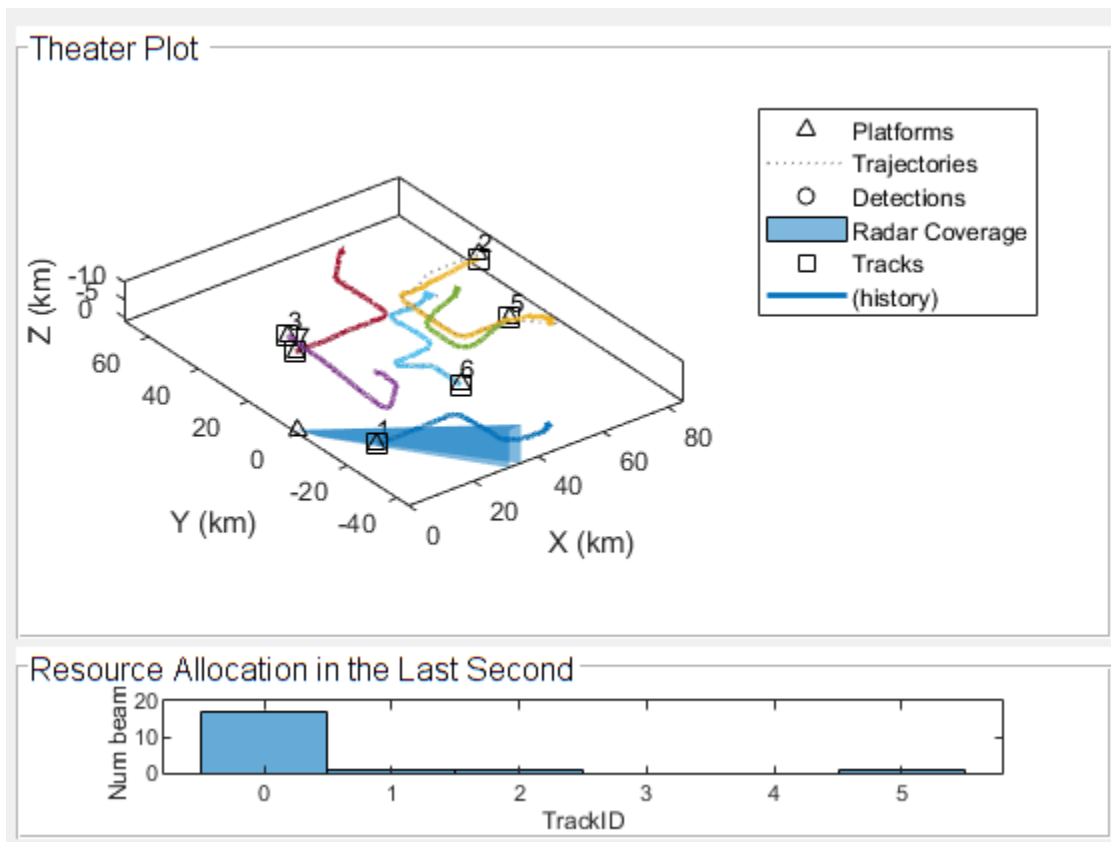
Analyze Results

Analyze the radar task load and its division between search, confirmation, and tracking jobs. The graph shows that most of the time the radar allocates about 75% search jobs and 25% tracking jobs, which is as expected when the targets are not maneuvering. When the targets are maneuvering, the resource manager adapts to allocate more tracking jobs. When more tracks are maneuvering at the same time, there are more tracking jobs, as seen near the 700th time step. The confirmation jobs occupy very little of the radar time because the tracker is configured to confirm tracks after two detection associations in three attempts. Therefore, the confirmation or rejection of tentative tracks is swift.

```
numSteps = numel(allocationType);
allocationSummary = zeros(3,numSteps);
for i = 1:numSteps
    for jobType = 1:3
        allocationSummary(jobType,i) = sum(allocationType(1,max(1,i-2*updateRate+1):i)==jobType),
    end
end
figure;
plot(1:numSteps,allocationSummary(:,1:numSteps))
title('Radar Allocation vs. Time Step');
xlabel('Time Step');
ylabel('Fraction of step in the last two seconds in each mode');
legend('Search','Confirmation','Tracking');
grid on
```
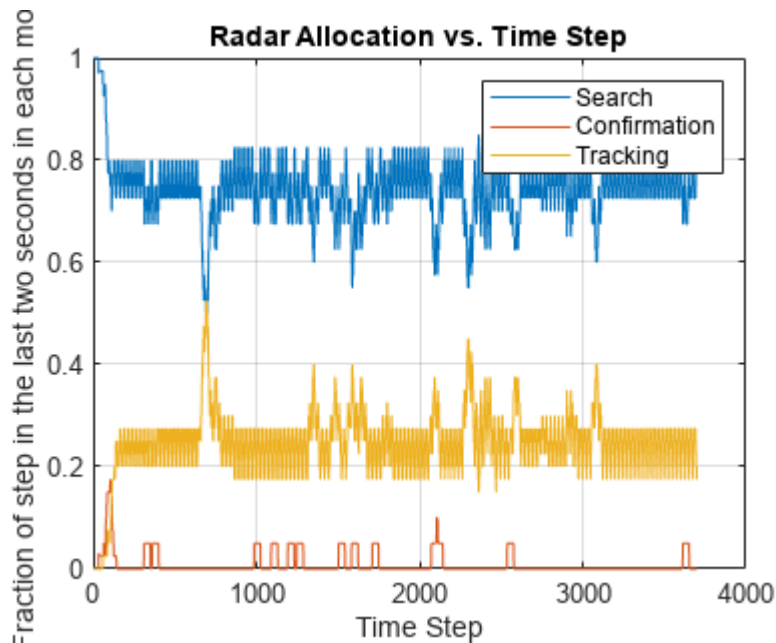
Compare this result with the result of Active Tracking at 0.8 Hz track revisit rate. The following figures show the tracking results and radar allocation graph for the Active Tracking case. The tracking results show that some tracks were lost and broken, but the radar resource allocation graph shows a similar 75% search and 25% tracking task division as in the case of the Adaptive Tracking. You can obtain these results by executing the code sample below.

```
clearData(plp);
clearData(dtp);
clearData(trp);
reset(tracker);
restart(scenario);

% Modify manager preferences to 0.8 Hz revisit rate for both non-maneuvering and maneuvering targ
managerPreferences = struct(...
    'Type', {"Search","Confirm","TrackNonManeuvering","TrackManeuvering"}, ...
    'RevisitRate', {0, updateRate, 0.8, 0.8}, ...
    'Priority', {1000, 3000, 1500, 2500});
% Run the scenario
allocationType = helperAdaptiveTrackingSim(scenario, thp, rax, tracker, resetJobQ, managerPrefere
```

Theater Plot

Resource Allocation in the Last Second

Radar Allocation vs. Time Step
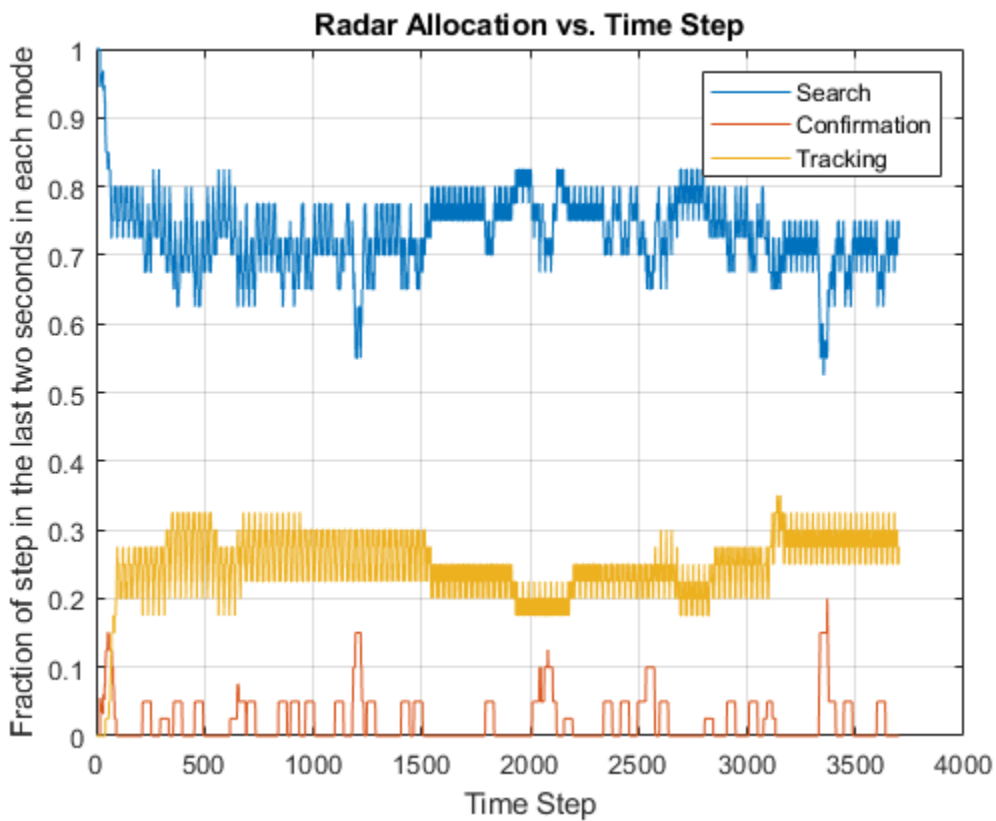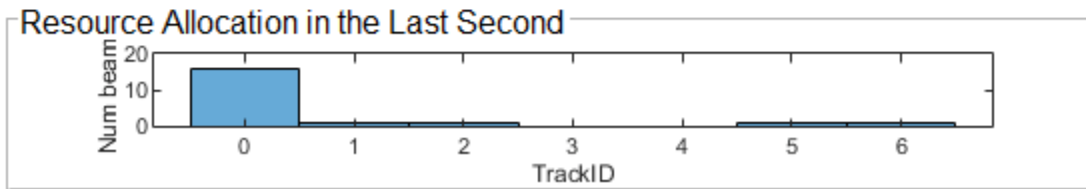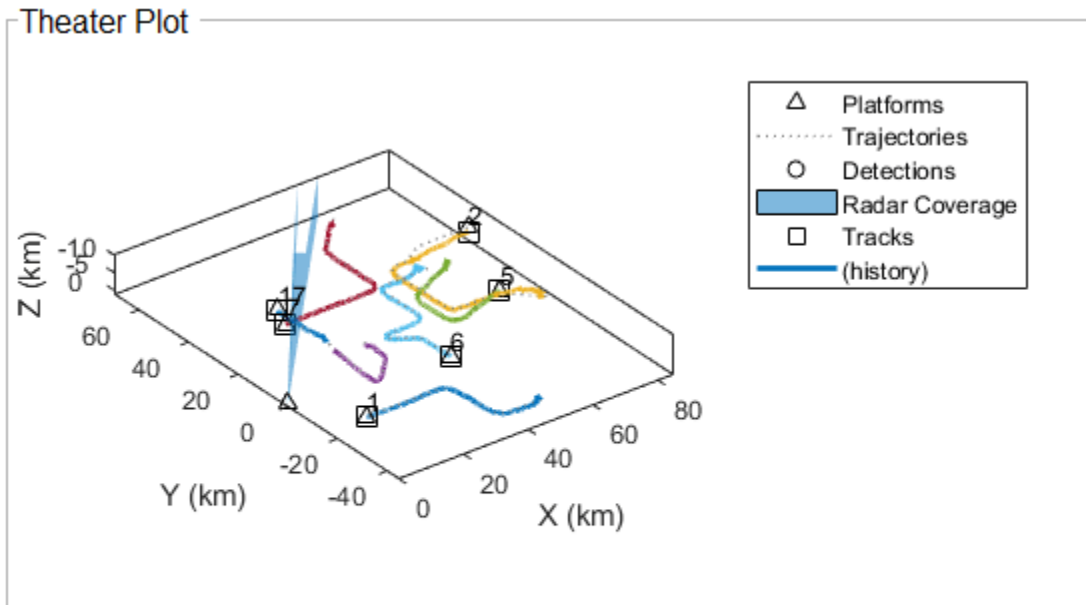
Clearly, a higher tracking revisit rate is needed for Active Tracking. The following two graphs show that increasing the track revisit rate to 2 Hz improves the tracking of maneuvering targets. However, the cost is that the radar dedicates more than 50% of its time to tracking tasks even when the tracks are not maneuvering. If the number of targets were greater, the radar would become overwhelmed.
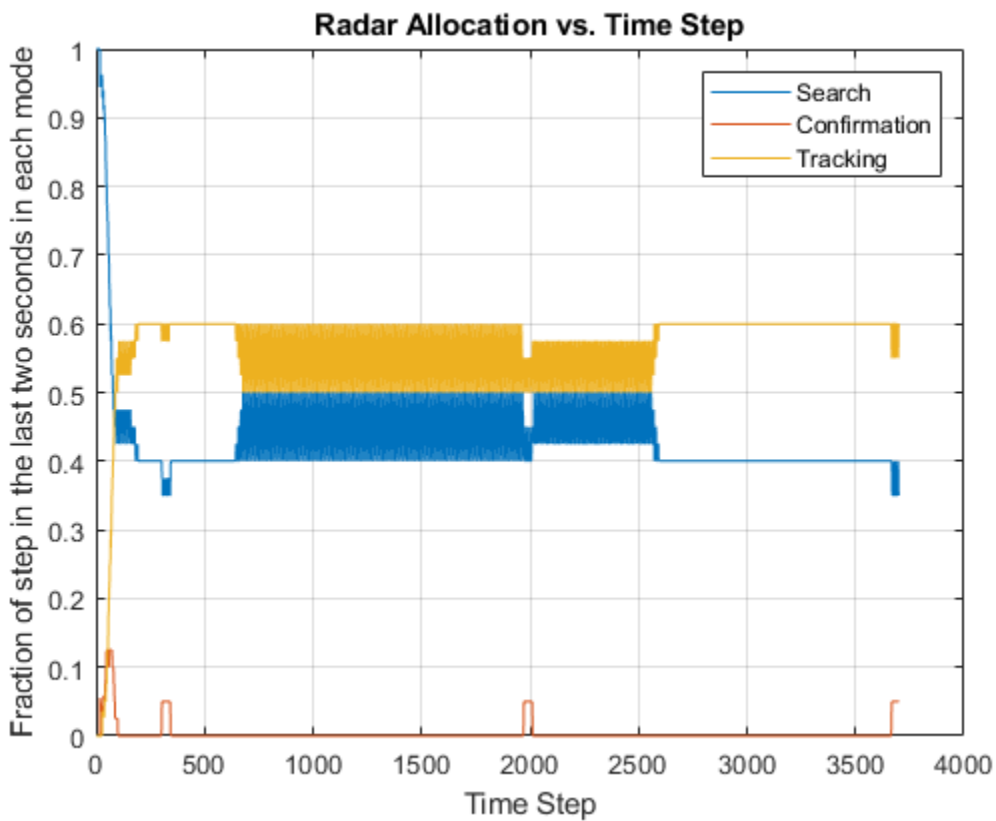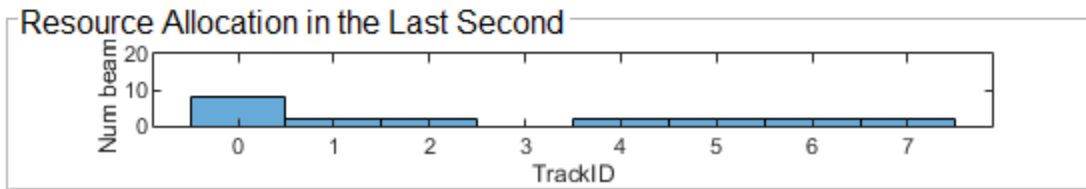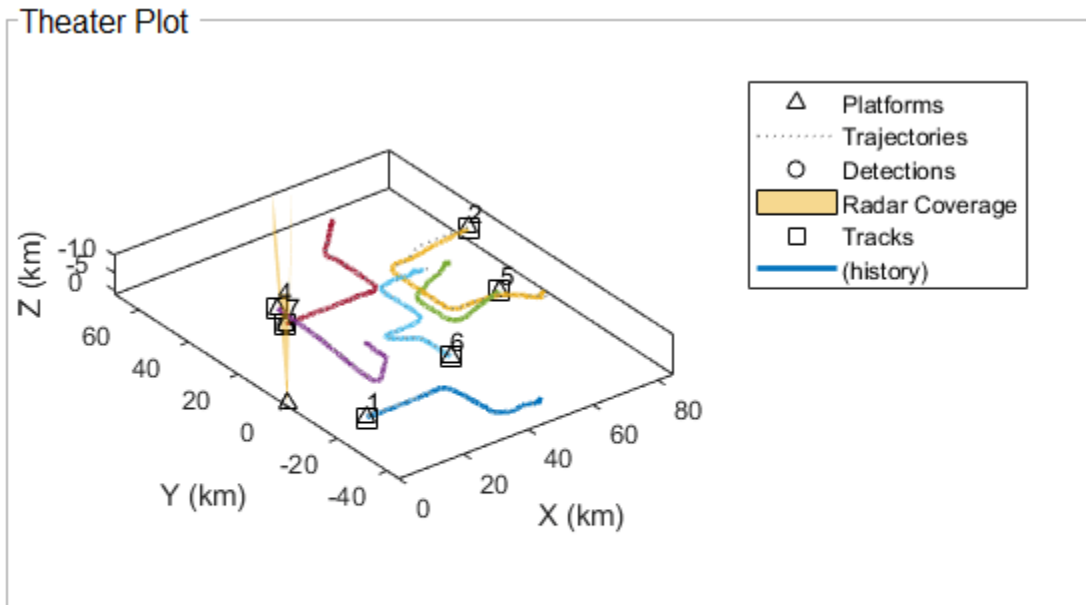
The previous results show that it is enough to revisit the maneuvering targets at a rate of 2 Hz. However, can Adaptive Tracking be used to lower the revisit rate of non-maneuvering targets beyond 0.8 Hz? The following graphs present results for 0.6 Hz and 4 Hz for non-maneuvering and maneuvering targets, respectively. With this setting, the radar resource allocation allows for 80%-85% of the time in search mode, leaving the radar with capacity to search and track even more targets.
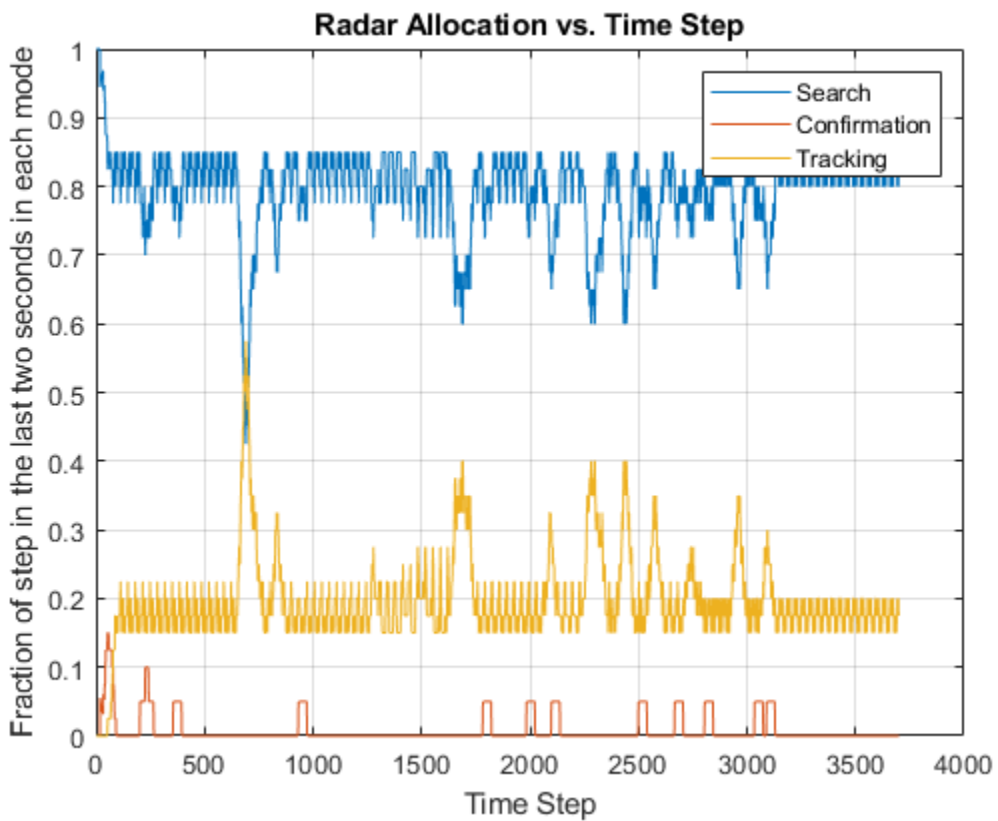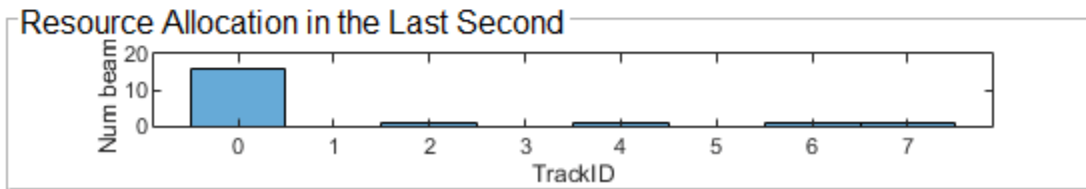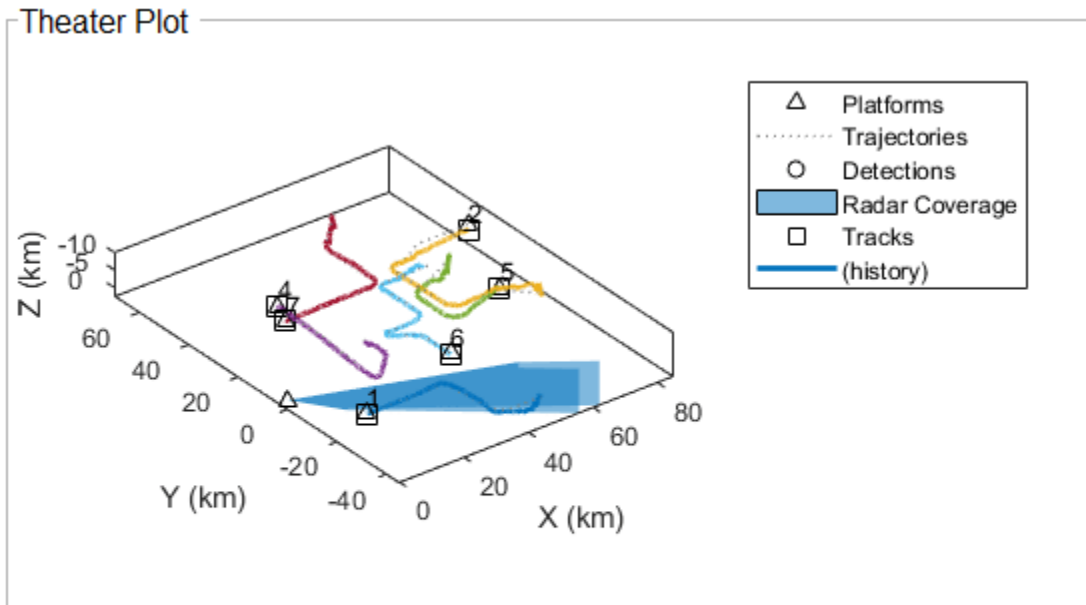
Summary

This example shows how to use the combination of tracking and radar resource management to adapt the revisit rate for maneuvering tracks. Adaptive tracking allows you to select revisit rate that is appropriate for each target type and maneuvering status. As a result, the radar becomes more efficient and can track a larger number of maneuvering targets.

References

[1] Charlish, Alexander, Folker Hoffmann, Christoph Degen, and Isabel Schlangen. "The Development From Adaptive to Cognitive Radar Resource Management." *IEEE Aerospace and Electronic Systems Magazine* 35, no. 6 (June 1, 2020): 8–19. https://doi.org/10.1109/MAES.2019.2957847.

Supporting Functions

getCurrentRadarTask

Returns the radar task that is used for pointing the radar beam.

```
type('getCurrentRadarTask.m')
```

```
function [currentjob,jobq] = getCurrentRadarTask(jobq,current_time)

searchq   = jobq.SearchQueue;
trackq    = jobq.TrackQueue;
searchidx = jobq.SearchIndex;
num_trackq_items = jobq.NumTrackJobs;

% Update search queue index
searchqidx = mod(searchidx-1,numel(searchq))+1;

% Find the track job that is due and has the highest priority
readyidx = find([trackq(1:num_trackq_items).Time]<=current_time);
[~,maxpidx] = max([trackq(readyidx).Priority]);
taskqidx = readyidx(maxpidx);

% If the track job found has a higher priority, use that as the current job
% and increase the next search job priority since it gets postponed.
% Otherwise, the next search job due is the current job.
if ~isempty(taskqidx) % && trackq(taskqidx).Priority >= searchq(searchqidx).Priority
    currentjob = trackq(taskqidx);
    for m = taskqidx+1:num_trackq_items
        trackq(m-1) = trackq(m);
    end
    num_trackq_items = num_trackq_items-1;
    searchq(searchqidx).Priority = searchq(searchqidx).Priority+100;
else
    currentjob = searchq(searchqidx);
    searchidx = searchqidx+1;

end

jobq.SearchQueue  = searchq;
jobq.SearchIndex  = searchidx;
jobq.TrackQueue   = trackq;
jobq.NumTrackJobs = num_trackq_items;
```

helperAdaptiveRadarSim

Runs the simulation

```
type('helperAdaptiveTrackingSim.m')

function allocationType = helperAdaptiveTrackingSim(scenario, thp, rax, tracker, resetJobQ, manag
% Initialize variables
radar = scenario.Platforms{end}.Sensors{1};
updateRate = radar.UpdateRate;
resourceAllocation = nan(1,updateRate);
h = histogram(rax,resourceAllocation,'BinMethod','integers');
xlabel(h.Parent,'TrackID')
ylabel(h.Parent,'Num beams');
numSteps = updateRate * 185;
allocationType = nan(1,numSteps);
currentStep = 1;
restart(scenario);
reset(tracker);
% Return to a reset state of jobq
jobq = resetJobQ;

% Plotters and axes
plp = thp.Plotters(1);
dtp = thp.Plotters(3);
cvp = thp.Plotters(4);
trp = thp.Plotters(5);

ax = gca;
colors = ax.ColorOrder;

% For repeatable results, set the random seed and revert it when done
s = rng(2020);
oc = onCleanup(@() rng(s));

% Main loop
tracks = {};


while advance(scenario)
    time = scenario.SimulationTime;

    % Update ground truth display
    poses = platformPoses(scenario);
    plotPlatform(plp, reshape([poses.Position],3,[])');

    % Point the radar based on the scheduler current job
    [currentJob,jobq] = getCurrentRadarTask(jobq,time);
    currentStep = currentStep + 1;
    if currentStep > updateRate
        resourceAllocation(1:end-1) = resourceAllocation(2:updateRate);
    end
    if strcmpi(currentJob.JobType,'Search')
        detectableTracks = zeros(0,1,'uint32');
        resourceAllocation(min([currentStep,updateRate])) = 0;
        allocationType(currentStep) = 1;
        cvp.Color = colors(1, :);
    else
        detectableTracks = currentJob.TrackID;
        resourceAllocation(min([currentStep,updateRate])) = currentJob.TrackID;
```

```
        if strcmpi(currentJob.JobType,'Confirm')
            allocationType(currentStep) = 2;
            cvp.Color = colors(2, :);
        else
            allocationType(currentStep) = 3;
            cvp.Color = colors(3, :);
        end
    end
    ra = resourceAllocation(~isnan(resourceAllocation));
    h.Data = ra;
    h.Parent.YLim = [0 updateRate];
    h.Parent.XTick = 0:max(ra);
    radar.LookAngle = currentJob.BeamDirection;
    plotCoverage(cvp, coverageConfig(scenario));

    % Collect detections and plot them
    detections = detect(scenario);
    if isempty(detections)
        meas = zeros(0,3);
    else
        dets = [detections{:}];
        meassph = reshape([dets.Measurement],3,[])';
        [x,y,z] = sph2cart(deg2rad(meassph(1)),deg2rad(meassph(2)),meassph(3));
        meas = (detections{1}.MeasurementParameters.Orientation*[x;y;z]+detections{1}.Measurement
    end
    plotDetection(dtp, meas);

    % Track and plot tracks
    if isLocked(tracker) || ~isempty(detections)
        [confirmedTracks,tentativeTracks,~,analysisInformation] = tracker(detections, time, detec
        pos = getTrackPositions(confirmedTracks,jobq.PositionSelector);
        plotTrack(trp,pos,string([confirmedTracks.TrackID]));

        tracks.confirmedTracks = confirmedTracks;
        tracks.tentativeTracks = tentativeTracks;
        tracks.analysisInformation = analysisInformation;
        tracks.PositionSelector = jobq.PositionSelector;
    end


    % Manage resources for next jobs
    jobq = manageResource(detections,jobq,tracker,tracks,currentJob,time,managerPreferences);
end
```

initMPARIMM

Initializes the IMM filter used by the tracker.

```
type('initMPARIMM.m')
```

```
function imm = initMPARIMM(detection)

cvekf = initcvekf(detection);
cvekf.StateCovariance(2,2) = 1e6;
cvekf.StateCovariance(4,4) = 1e6;
cvekf.StateCovariance(6,6) = 1e6;
```

```
ctekf = initctekf(detection);
ctekf.StateCovariance(2,2) = 1e6;
ctekf.StateCovariance(4,4) = 1e6;
ctekf.StateCovariance(7,7) = 1e6;
ctekf.ProcessNoise(3,3) = 1e6; % Large noise for unknown angular acceleration

imm = trackingIMM('TrackingFilters', {cvekf;ctekf}, 'TransitionProbabilities', [0.99, 0.99]);
```

manageResources

Manages the job queue and creates new tasks based on the tracking results.

```
type('manageResource.m')
```

```
function jobq = manageResource(detections,jobq,tracker,tracks,current_job,current_time,managerPre

trackq           = jobq.TrackQueue;
num_trackq_items = jobq.NumTrackJobs;
if ~isempty(detections)
    detection = detections{1};
else
    detection = [];
end

% Execute current job
switch current_job.JobType
    case 'Search'
        % For search job, if there is a detection, establish tentative
        % track and schedule a confirmation job
        if ~isempty(detection)
            % A search task can still find a track we already have. Define
            % a confirmation task only if it's a tentative track. There
            % could be more than one if there are false alarms. Create
            % confirm jobs for tentative tracks created at this update.

            numTentative = numel(tracks.tentativeTracks);
            for i = 1:numTentative
                if tracks.tentativeTracks(i).Age == 1 && tracks.tentativeTracks(i).IsCoasted ==
                    trackid = tracks.tentativeTracks(i).TrackID;
                    job = revisitTrackJob(tracker, trackid, current_time, managerPreferences, 'Co
                    num_trackq_items = num_trackq_items+1;
                    trackq(num_trackq_items) = job;
                end
            end
        end

    case 'Confirm'
        % For a confirm job, if the track ID is within the tentative
        % tracks, it means that we need to run another confirmation job
        % regardless of having a detection. If the track ID is within the
        % confirmed tracks, it means that we must have gotten a detection,
        % and the track passed the confirmation logic test. In this case we
        % need to schedule a track revisit job.
        trackid = current_job.TrackID;
        tentativeTrackIDs = [tracks.tentativeTracks.TrackID];
        confirmedTrackIDs = [tracks.confirmedTracks.TrackID];
        if any(trackid == tentativeTrackIDs)
            job = revisitTrackJob(tracker, trackid, current_time, managerPreferences, 'Confirm',
```

```
                num_trackq_items = num_trackq_items+1;
                trackq(num_trackq_items) = job;
            elseif any(trackid == confirmedTrackIDs)
                job = revisitTrackJob(tracker, trackid, current_time, managerPreferences, 'TrackNonMa
                num_trackq_items = num_trackq_items+1;
                trackq(num_trackq_items) = job;
            end

        otherwise % Covers both types of track jobs
            % For track job, if the track hasn't been dropped, update the track
            % and schedule a track job corresponding to the revisit time
            % regardless of having a detection. In the case when there is no
            % detection, we could also predict and schedule a track job sooner
            % so the target is not lost. This would require defining another
            % job type to control the revisit rate for this case.

            trackid = current_job.TrackID;
            confirmedTrackIDs = [tracks.confirmedTracks.TrackID];
            if any(trackid == confirmedTrackIDs)
                jobType = 'TrackNonManeuvering';
                mdlProbs = getTrackFilterProperties(tracker, trackid, 'ModelProbabilities');
                if mdlProbs{1}(2) > 0.6
                    jobType = 'TrackManeuvering';
                end

                job = revisitTrackJob(tracker, trackid, current_time, managerPreferences, jobType, tr
                num_trackq_items = num_trackq_items+1;
                trackq(num_trackq_items) = job;
            end
    end

    jobq.TrackQueue    = trackq;
    jobq.NumTrackJobs = num_trackq_items;
end

function job = revisitTrackJob(tracker, trackID, currentTime, managerPreferences, jobType, positi
    types = [managerPreferences.Type];
    inTypes = strcmpi(jobType,types);
    revisitTime = 1/managerPreferences(inTypes).RevisitRate + currentTime;
    predictedTrack = predictTracksToTime(tracker,trackID,revisitTime);

    xpred = getTrackPositions(predictedTrack,positionSelector);

    [phipred,thetapred,rpred] = cart2sph(xpred(1),xpred(2),xpred(3));
    job = struct('JobType',jobType,'Priority',managerPreferences(inTypes).Priority,...
        'BeamDirection',rad2deg([phipred thetapred]),'WaveformIndex',1,'Time',revisitTime,...
        'Range',rpred,'TrackID',trackID);

end
```

Search and Track Scheduling for Multifunction Phased Array Radar

This example shows how to simulate a multifunction phased array radar system. A multifunction radar can perform jobs that usually require multiple traditional radars. Examples of traditional radars are scanning radars, which are responsible for searching targets, and tracking radars, which are responsible for tracking targets. In this example, the multifunction phased array radar performs both scanning (searching) and tracking tasks. Based on the detections and tracks obtained from the current echo, the radar decides what to do next to ensure that targets of interest are tracked and the desired airspace is searched. The multifunction phased array radar works as a closed loop, including features such as task scheduling, waveform selection, detection generation, and target tracking.

Radar Configuration

Assume the multifunction radar operates at S band and must detect targets between 2 km and 100 km, with a minimum target radar cross section (RCS) of 1 square meters.

```
fc     = 2e9;                % Radar carrier frequency (Hz)
c      = 3e8;                % Propagation speed (m/s)
lambda = c/fc;               % Radar wavelength (m)

maxrng = 100e3;              % Maximum range (m)
minrng = 2e3;                % Minimum range (m)
```

Waveform

To satisfy the range requirement, define and use a linear FM waveform with a 1 MHz bandwidth.

```
bw     = 1e6;
fs     = 1.5*bw;
prf    = 1/range2time(maxrng,c);
dcycle = 0.1;

wav = phased.LinearFMWaveform('SampleRate', fs, ...
    'DurationSpecification', 'Duty cycle', 'DutyCycle', dcycle, ...
    'PRF', prf, 'SweepBandwidth', bw);
```

Calculate the range resolution achievable by the waveform.

```
rngres = bw2rangeres(bw,c)


rngres =

   150
```

Radar Antenna

The multifunction radar is equipped with a phased array that can electronically scan the radar beams in space. Use a 50-by-50 rectangular array with elements separated by half wavelength to achieve a half power beam width of approximately 2 degrees.

```
arraysz   = 50;
ant       = phased.URA('Size',arraysz,'ElementSpacing',lambda/2);
```

```matlab
ant.Element.BackBaffled = true;

arraystv  = phased.SteeringVector('SensorArray',ant,'PropagationSpeed',c);
radiator  = phased.Radiator('OperatingFrequency',fc, ...
    'PropagationSpeed', c, 'Sensor',ant, 'WeightsInputPort', true);
collector = phased.Collector('OperatingFrequency',fc, ...
    'PropagationSpeed', c, 'Sensor',ant);

beamw = rad2deg(lambda/(arraysz*lambda/2))
```

```
beamw =

    2.2918
```

Transmitter and Receiver

Use the detection requirements to derive the appropriate transmit power. Assume the noise figure on the receiving preamplifier is 7 dB.

```matlab
pd      = 0.9;                      % Probability of detection
pfa     = 1e-6;                     % Probability of false alarm
snr_min = albersheim(pd, pfa, 1);
ampgain = 20;
tgtrcs  = 1;
ant_snrgain = pow2db(arraysz^2);

ppower  = radareqpow(lambda,maxrng,snr_min,wav.PulseWidth,...
    'RCS',tgtrcs,'Gain',ampgain+ant_snrgain);

tx = phased.Transmitter('PeakPower',ppower,'Gain',ampgain,'InUseOutputPort',true);
rx = phased.ReceiverPreamp('Gain',ampgain,'NoiseFigure',7,'EnableInputPort',true);

rxpassthrough = phased.ReceiverPreamp('SampleRate',fs,'Gain',0,...
    'ReferenceTemperature',1);
radar = radarTransceiver('Waveform',wav,'Transmitter',tx,...
    'TransmitAntenna',radiator,'ReceiveAntenna',collector,...
    'Receiver',rxpassthrough,'ElectronicScanMode','Custom');
```

Signal Processing

The multifunction radar applies a sequence of operations, including matched filtering, time varying gain, monopulse, and detection, to the received signal to generate range and angle measurements of the detected targets.

```matlab
% matched filter
mfcoeff = getMatchedFilter(wav);
mf      = phased.MatchedFilter('Coefficients',mfcoeff,'GainOutputPort', true);

% time varying gain
tgrid   = unigrid(0,1/fs,1/prf,'[)');
rgates  = c*tgrid/2;
rngloss = 2*fspl(rgates,lambda);
refloss = 2*fspl(maxrng,lambda);
tvg     = phased.TimeVaryingGain('RangeLoss',rngloss,'ReferenceLoss',refloss);

% monopulse
```

```
monfeed = phased.MonopulseFeed('SensorArray',ant,'PropagationSpeed',c,...
    'OperatingFrequency',fc,'SquintAngle',1);
monest  = getMonopulseEstimator(monfeed);
```

Data Processing

The detections are fed into a tracker, which performs several operations. The tracker maintains a list of tracks, that is, estimates of target states in the area of interest. If a detection cannot be assigned to any track already maintained by the tracker, the tracker initiates a new track. In most cases, whether the new track represents a true target or false target is unclear. At first, a track is created with a tentative status. If enough detections are obtained, the track becomes confirmed. Similarly, if no detections are assigned to a track, the track is coasted (predicted without correction). If the track has a few missed updates, the tracker deletes the track.

The multifunction radar uses a tracker that associates the detections to the tracks using a global nearest neighbor (GNN) algorithm.

```
tracker = radarTracker('FilterInitializationFcn',@initSATGNN,...
    'ConfirmationThreshold',[2 3], 'DeletionThreshold',5,...
    'HasDetectableTrackIDsInput',true,'AssignmentThreshold',100,...
    'MaxNumTracks',2,'MaxNumSensors',1);
```

Group all radar components together in a structure for easier reference in the simulation loop.

```
mfradar.Tx      = tx;
mfradar.Rx      = rx;
mfradar.TxAnt   = radiator;
mfradar.RxAnt   = collector;
mfradar.Wav     = wav;
mfradar.Radar   = radar;
mfradar.RxFeed  = monfeed;
mfradar.MF      = mf;
mfradar.TVG     = tvg;
mfradar.DOA     = monest;
mfradar.STV     = arraystv;
mfradar.Tracker = tracker;
mfradar.IsTrackerInitialized = false;
```

Target and Scene Definition

This example assumes the radar is stationary at the origin with two targets in its field of view. One target departs from the radar and is at a distance of around 50 km. The other target approaches the radar and is 30 km away. Both targets have an RCS of 1 square meters.

```
% Define the targets.
tgtpos = [29875 49637; 0 4225; 0 0];
tgtvel = [-100 120; 0 100; 0 0];

ntgt = size(tgtpos,2);
tgtmotion = phased.Platform('InitialPosition',tgtpos,'Velocity',tgtvel);
target = phased.RadarTarget('MeanRCS',tgtrcs*ones(1,ntgt),'OperatingFrequency',fc);
```

Assume the propagation environment is free space.

```
channel = phased.FreeSpace('SampleRate',fs,'TwoWayPropagation',true,'OperatingFrequency',fc);
```

Group targets and propagation channels together in a structure for easier reference in the simulation loop.

```
env.Target           = target;
env.TargetMotion     = tgtmotion;
env.Channel          = channel;

scene = radarScenario('UpdateRate',prf);
radartraj = kinematicTrajectory('SampleRate',prf,'Position',[0 0 0],...
    'Velocity',[0 0 0],'AccelerationSource','Property',...
    'AngularVelocitySource','Property');
radarplat = platform(scene,'Position',[0 0 0],'Sensors',{radar});
tgtplat1 = platform(scene,'Trajectory',...
    kinematicTrajectory('SampleRate',prf,'Position',tgtpos(:,1).',...
    'Velocity',tgtvel(:,1).','AccelerationSource','Property',...
    'AngularVelocitySource','Property'));
tgtplat2 = platform(scene,'Trajectory',...
    kinematicTrajectory('SampleRate',prf,'Position',tgtpos(:,2).',...
    'Velocity',tgtvel(:,2).','AccelerationSource','Property',...
    'AngularVelocitySource','Property'));

env.Scene = scene;
env.RadarPlatform = radarplat;
```

Radar Resource Management

While using one multifunction radar to perform multiple tasks has its advantages, it also has a higher cost and more sophisticated logic. In general, a radar has finite resources to spend on its tasks. If resources are used for tracking tasks, then those resources are not available for searching tasks until the tracking tasks are finished. Because of this resource allocation, a critical component when using a multifunction radar is resource management.

Search Tasks

The search tasks can be considered as deterministic. In this example, a raster scan is used to cover the desired airspace. If no other tasks exist, the radar scans the space one angular cell at a time. The size of an angular cell is determined by the beam width of the antenna array.

Assume the radar scans a space from -30 to 30 degrees azimuth and 0 to 20 degrees elevation. Calculate the angular search grid using the beam width.

```
scanregion    = [-30, 30, 0, 20];
azscanspan    = diff(scanregion(1:2));
numazscan     = ceil(azscanspan/beamw);
azscanangles  = linspace(scanregion(1),scanregion(2),numazscan);
elscanspan    = diff(scanregion(3:4));
numelscan     = ceil(elscanspan/beamw);
elscanangles  = linspace(scanregion(3),scanregion(4),numelscan);
[elscangrid,azscangrid] = meshgrid(elscanangles,azscanangles);
scanangles    = [azscangrid(:) elscangrid(:)].';
```

The beam position grid and target scene are shown below.

```
sceneplot = helperSATTaskPlot('initialize',scanangles,azscanangles,maxrng,beamw,tgtpos);
```

Search Beam Grid

Radar Azimuth Coverage

Target 1

Target 2
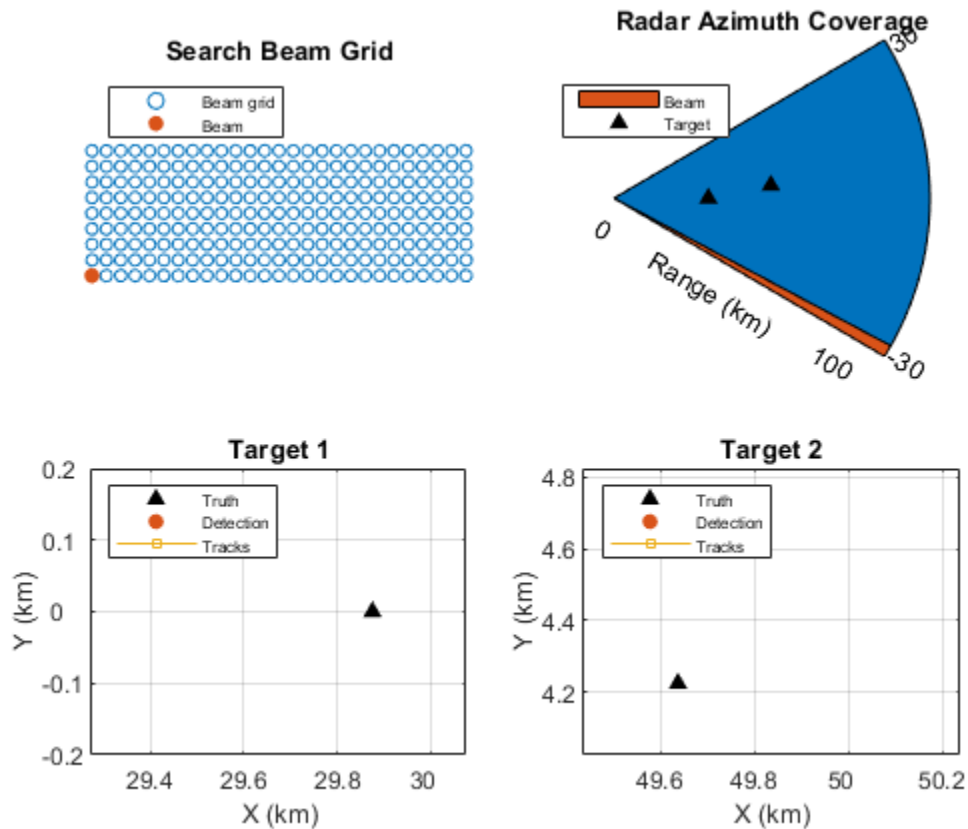
The search beams are transmitted one at a time sequentially until the entire search area is covered. Once the entire search area is covered, the radar repeats the search sequence. The searches are performed along the azimuthal direction, one elevation angle a time. The search tasks are often contained in a job queue.

```
searchq = struct('JobType','Search','BeamDirection',num2cell(scanangles,1),...
    'Priority',1000,'WaveformIndex',1);
current_search_idx = 1;
```

Each job in the queue specifies the job type as well as the pointing direction of the beam. It also contains a priority value for the job. This priority value is determined by the job type. This example uses a value of 1000 as the priority for search jobs.

```
disp(searchq(current_search_idx))
```

```
        JobType: 'Search'
  BeamDirection: [2×1 double]
       Priority: 1000
  WaveformIndex: 1
```

Track Tasks

Unlike search tasks, track tasks cannot be planned. Track tasks are created only when a target is detected by a search task or when the target has already been tracked. Track tasks are dynamic tasks that get created and executed based on the changing scenario. Similar to search tasks, track tasks are also managed in a job queue.

```matlab
trackq(10) = struct('JobType',[],'BeamDirection',[],'Priority',3000,'WaveformIndex',[],...
    'Time',[],'Range',[],'TrackID',[]);
num_trackq_items = 0;
disp(trackq(1))
         JobType: []
   BeamDirection: []
        Priority: []
   WaveformIndex: []
            Time: []
           Range: []
         TrackID: []
```

Group search and track queues together in a structure for easier reference in the simulation loop.

```matlab
jobq.SearchQueue  = searchq;
jobq.SearchIndex  = current_search_idx;
jobq.TrackQueue   = trackq;
jobq.NumTrackJobs = num_trackq_items;
```

Because a tracking job cannot be initialized before a target is detected, all tracking jobs start as empty jobs. Once a job is created, it contains the information such as its job type, the direction of the beam, and time to execute. The tracking task has a priority of 3000, which is higher than the priority of 1000 for a search job. This higher priority value means that when the time is in conflict, the system will execute the tracking job first.

The size limit for the queue in this example is set to 10.

Task Scheduling

In this example, for simplicity, the multifunction radar executes only one type of job within a small time period, often referred to as a dwell, but can switch tasks at the beginning of each dwell. For each dwell, the radar looks at all tasks that are due for execution and picks the one that has the highest priority. Consequently, jobs that get postponed will now have an increased priority and are more likely to be executed in the next dwell.

Simulation

This section of the example simulates a short run of the multifunction radar system. The entire structure of the multifunction radar simulation is represented by this diagram.
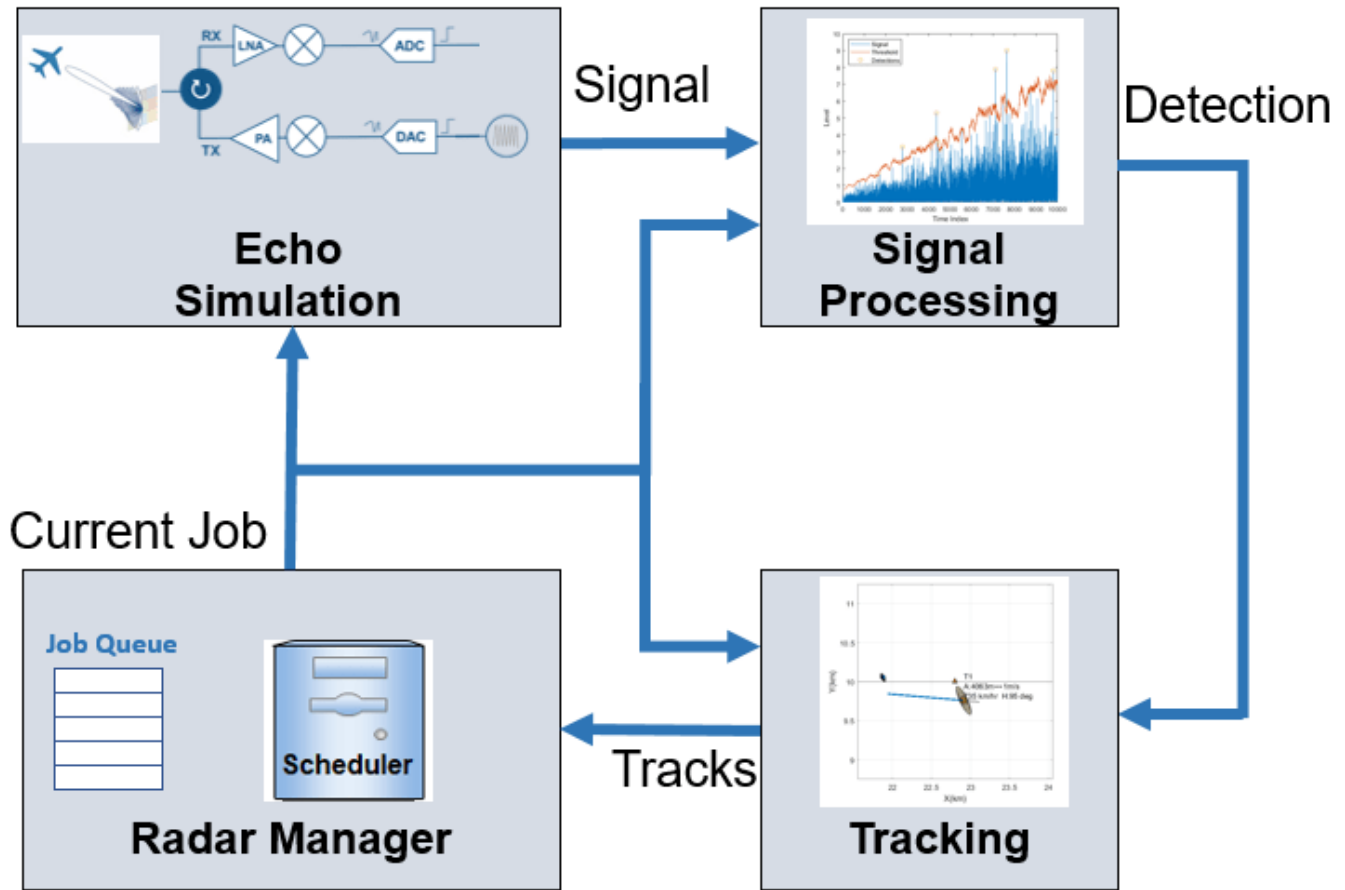
The simulation starts with the radar manager, which provides an initial job. Based on this job, the radar transmits the waveform, simulates the echo, and applies signal processing to generate the detection. The detection is processed by a tracker to create tracks for targets. The tracks then go back to the radar manager. Based on the tracks and the knowledge about the scene, the radar manager schedules new track jobs and picks the job for the next dwell.

The logic of the radar manager operation is shown in this flowchart and described in these steps.
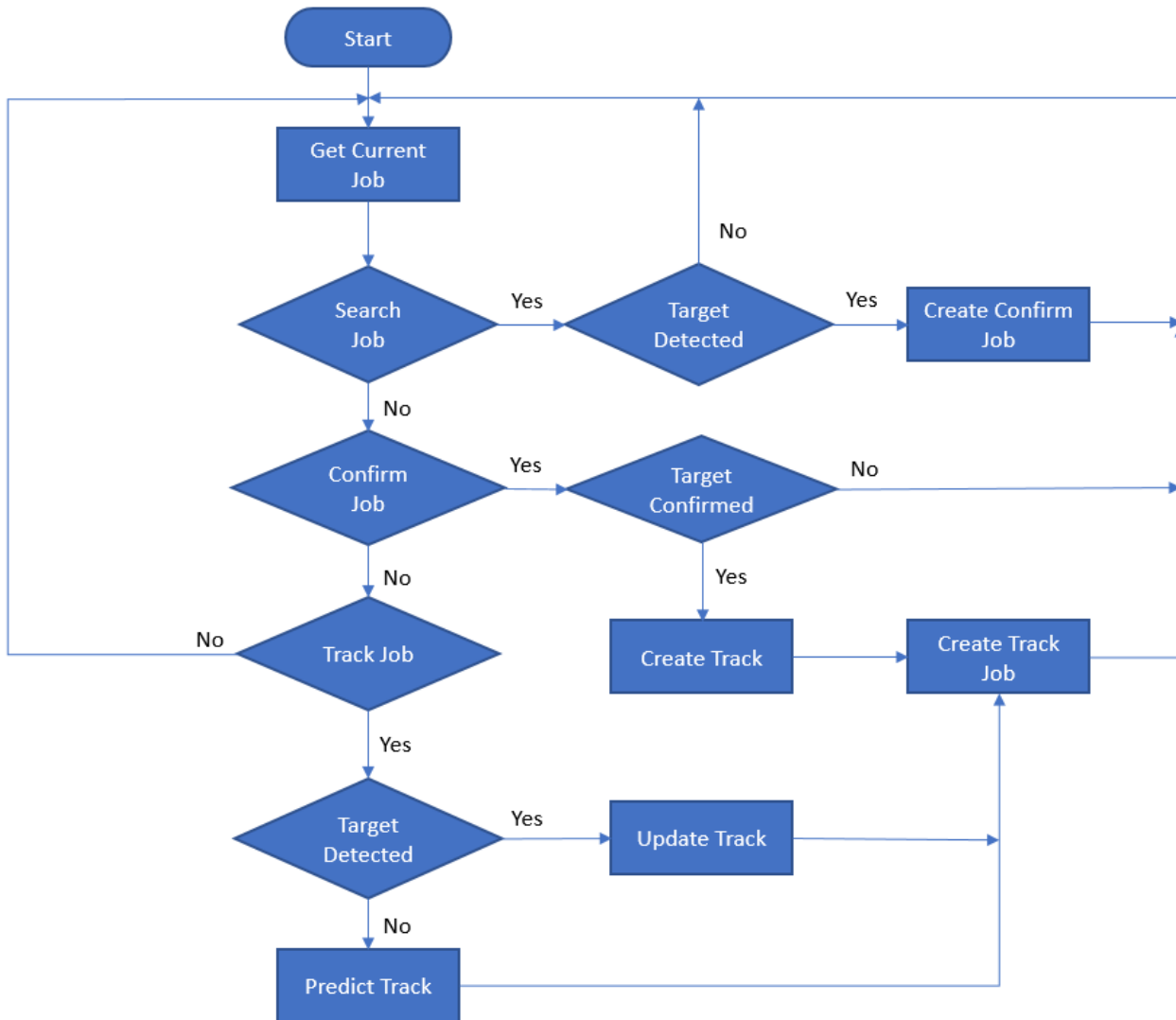
1   The radar starts with a search job.

2   If a target is present in the detection, the radar schedules a confirmation job in the same direction to ensure that the presence of this target is not a false alarm. The confirmation task has a higher priority than the search task but not as high as the track task. If the detection gets confirmed, a track is established, and a track job is created to be executed after a given revisit time. If the detection is not confirmed, then the original detection is considered as a false alarm, and no track is created.

3   If the current job is a track job, the radar performs the detection, updates the track, and creates a future track job.

4   Based on the priority and time for execution, the radar selects the next job.

Assume a dwell is 10 ms. At the beginning of the simulation, the radar is configured to search one beam at a time.

```
rng(2018);
current_time = 0;
```

```
Npulses     = 10;
numdwells   = 200;
dwelltime   = 0.01;

jobload.num_search_job = zeros(1,numdwells);
jobload.num_track_job  = zeros(1,numdwells);
```

You can run the example in its entirety to see the plots being dynamically updated during execution. In the top two plots, the color of the beams indicates the types of the current job: red for search, yellow for confirm, and purple for track. The two plots below show the true locations (triangle), detections (circle), and tracks (square) of the two targets, respectively. System log also displays in the command line to explain the system behavior at the current moment. Next, the example shows more details about several critical moments of the simulation.

Simulate the system behavior until it detects the first target. The simulation loop follows the previous system diagram.

```
for dwell_idx = 1:14
    % Scheduler to provide current job
    [current_job,jobq]       = currentRadarTask(jobq,current_time);

    % Simulate the received I/Q signal
    [xsum,xdaz,xdel,mfradar] = generateEchos(mfradar,env,current_job);

    % Signal processor to extract detection
    [detection,mfradar]      = generateDetections(xsum,xdaz,xdel,mfradar,current_job,current_time

    % Radar manager to perform data processing and update track queue
    [jobq,allTracks,mfradar] = updateTrackAndJobQueue(detection,jobq,mfradar,current_job,current_

    % Visualization
    helperSATTaskPlot('update',sceneplot,current_job,maxrng,beamw,tgtpos,allTracks,detection.Meas

    % Update time
    tgtpos = env.TargetMotion(dwelltime-Npulses/mfradar.Wav.PRF);
    current_time = current_time+dwelltime;

    % Record resource allocation
    if strcmp(current_job.JobType,'Search')
        jobload.num_search_job(dwell_idx) = 1;
    else
        jobload.num_track_job(dwell_idx)  = 1;
    end

end
```

```
0.000000 sec:    Search    [-30.000000 0.000000]
0.010000 sec:    Search    [-27.692308 0.000000]
0.020000 sec:    Search    [-25.384615 0.000000]
0.030000 sec:    Search    [-23.076923 0.000000]
0.040000 sec:    Search    [-20.769231 0.000000]
0.050000 sec:    Search    [-18.461538 0.000000]
0.060000 sec:    Search    [-16.153846 0.000000]
0.070000 sec:    Search    [-13.846154 0.000000]
0.080000 sec:    Search    [-11.538462 0.000000]
0.090000 sec:    Search    [-9.230769 0.000000]
0.100000 sec:    Search    [-6.923077 0.000000]
```

```
0.110000 sec:    Search    [-4.615385 0.000000]
0.120000 sec:    Search    [-2.307692 0.000000]
0.130000 sec:    Search    [0.000000 0.000000]    Target detected at 29900.000000 m
```

Search Beam Grid

Radar Azimuth Coverage
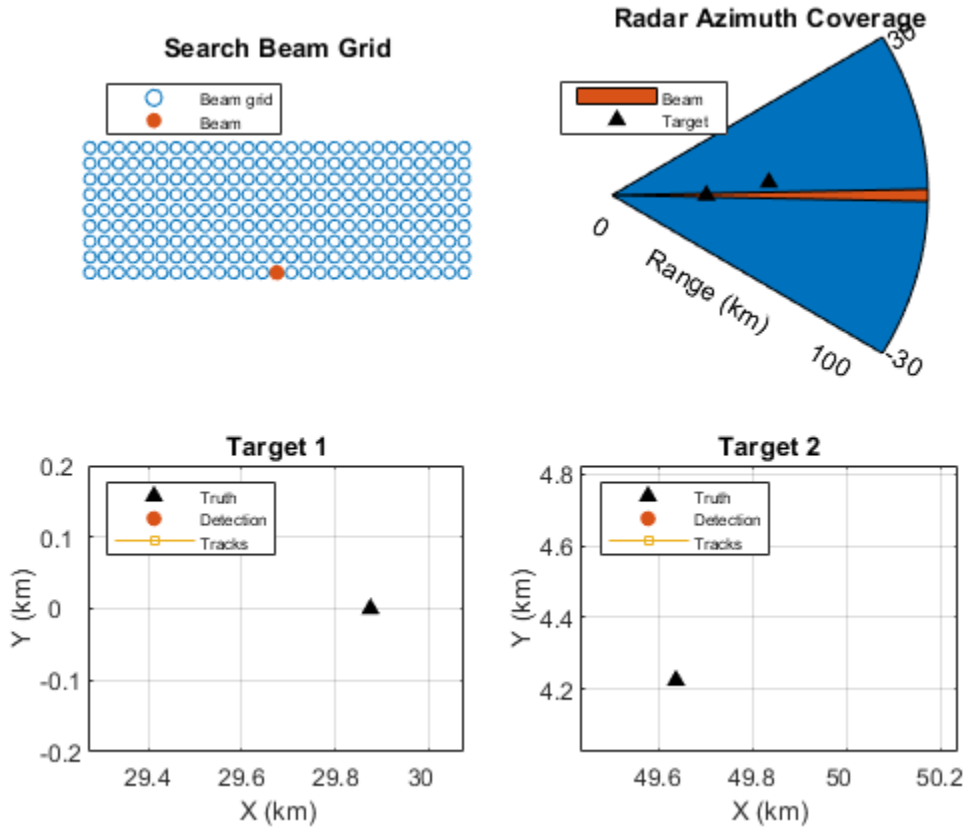
Target 1

Target 2

As expected, the radar gets a detection when the radar beam illuminates the target, as shown in the figure. When this happens, the radar sends a confirmation beam immediately to make sure it is not a false detection.

Next, show the results for the confirmation job. The rest of this example shows simplified code that combines the simulation loop into a system simulation function.

```
[mfradar,env,jobq,jobload,current_time,tgtpos] = SATSimRun(...
    mfradar,env,jobq,jobload,current_time,dwelltime,sceneplot,maxrng,beamw,tgtpos,15,15);
```

```
0.140000 sec:    Confirm    [0.001804 0.006157]    Created track 1 at 29900.000000 m
```

Search Beam Grid

Radar Azimuth Coverage
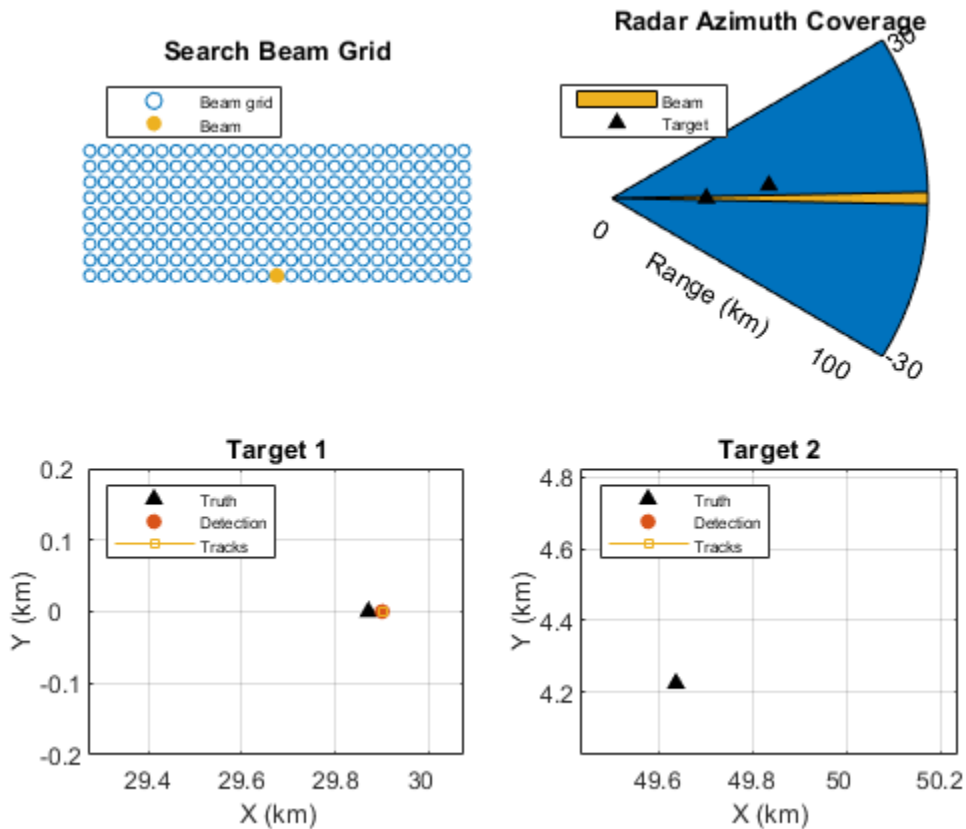
Target 1

Target 2

The figure now shows the confirmation beam. Once the detection is confirmed, a track is established for the target, and a track job is scheduled to execute after a short interval.

This process repeats for every detected target until the revisit time, at which point the multifunction radar stops the search sequence and performs the track task again.

```
[mfradar,env,jobq,jobload,current_time,tgtpos] = SATSimRun(...
    mfradar,env,jobq,jobload,current_time,dwelltime,sceneplot,maxrng,beamw,tgtpos,16,25);
```

```
0.150000 sec:    Search    [2.307692 0.000000]
0.160000 sec:    Search    [4.615385 0.000000]      Target detected at 49900.000000 m
0.170000 sec:    Confirm    [4.896994 0.014031]       Created track 2 at 49900.000000 m
0.180000 sec:    Search    [6.923077 0.000000]
0.190000 sec:    Search    [9.230769 0.000000]
0.200000 sec:    Search    [11.538462 0.000000]
0.210000 sec:    Search    [13.846154 0.000000]
0.220000 sec:    Search    [16.153846 0.000000]
0.230000 sec:    Search    [18.461538 0.000000]
0.240000 sec:    Track     [0.000833 -0.000127]     Track 1 at 29900.000000 m
```

Search Beam Grid

Radar Azimuth Coverage
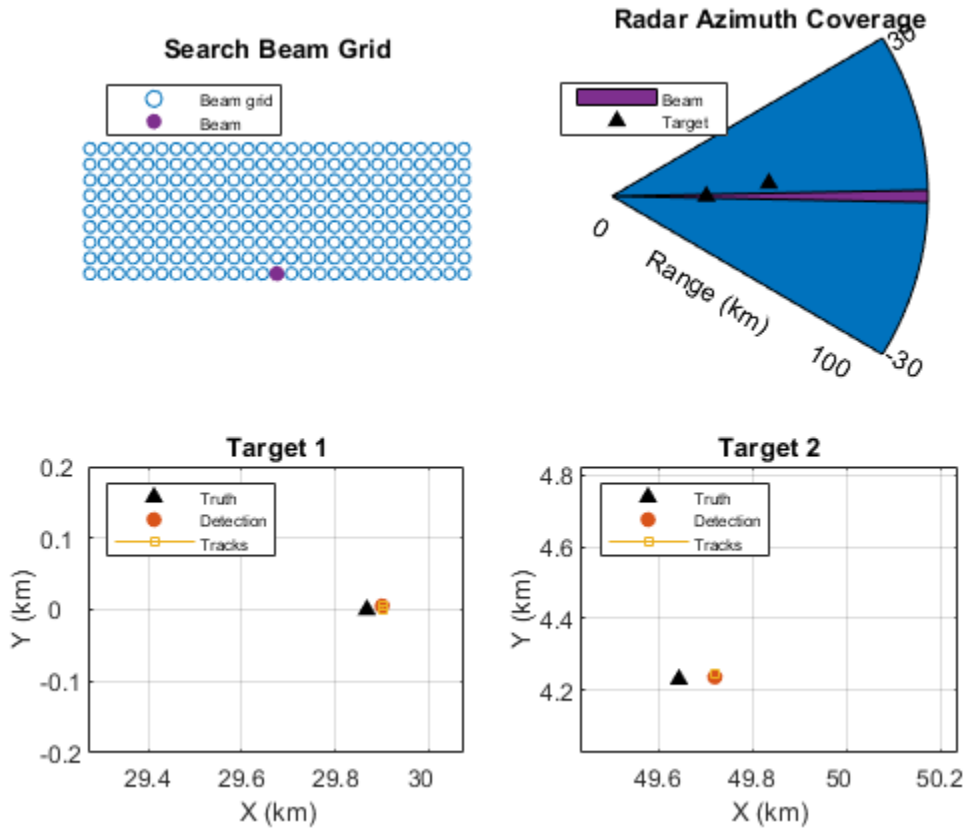
Target 1

Target 2

The results show that the simulation stops at a track beam. The zoomed-in figures around the two targets show how the tracks are updated based on the detection and measurements. A new track job for the next revisit is also added to the job queue during the execution of a track job.

This process repeats for each dwell. This simulation runs the radar system for a 2-second period. After a while, the second target is detected beyond 50 km. Based on this information, the radar manager reduces how often the system needs to track the second target. This reduction frees up resources for other, more urgent needs.

```
[mfradar,env,jobq,jobload,current_time,tgtpos] = SATSimRun(...
    mfradar,env,jobq,jobload,current_time,dwelltime,sceneplot,maxrng,beamw,tgtpos,26,numdwells);
```

```
0.250000 sec:    Search    [20.769231 0.000000]
0.260000 sec:    Search    [23.076923 0.000000]
0.270000 sec:    Track     [4.882021 0.002546]     Track 2 at 49900.000000 m
0.280000 sec:    Search    [25.384615 0.000000]
0.290000 sec:    Search    [27.692308 0.000000]
0.340000 sec:    Track     [0.014417 0.015384]     Track 1 at 29900.000000 m
0.370000 sec:    Track     [4.883901 0.012786]     Track 2 at 49900.000000 m
0.440000 sec:    Track     [0.002694 0.009423]     Track 1 at 29900.000000 m
0.470000 sec:    Track     [4.889519 -0.009622]     Track 2 at 49900.000000 m
0.540000 sec:    Track     [-0.003890 -0.006735]     Track 1 at 29900.000000 m
0.570000 sec:    Track     [4.905253 -0.023520]     Track 2 at 49900.000000 m
0.640000 sec:    Track     [-0.009124 0.012105]     Track 1 at 29900.000000 m
0.670000 sec:    Track     [4.908435 -0.012902]     Track 2 at 49900.000000 m
0.740000 sec:    Track     [-0.014700 -0.007442]     Track 1 at 29800.000000 m
```

```
0.770000 sec:     Track    [4.916246 -0.006127]    Track 2 at 49900.000000 m
0.840000 sec:     Track    [0.002371 -0.007959]    Track 1 at 29800.000000 m
0.870000 sec:     Track    [4.919202 0.000403]    Track 2 at 49900.000000 m
0.940000 sec:     Track    [0.003877 -0.018467]    Track 1 at 29800.000000 m
0.970000 sec:     Track    [4.931431 0.003165]    Track 2 at 49900.000000 m
1.040000 sec:     Track    [0.002358 -0.007887]    Track 1 at 29800.000000 m
1.070000 sec:     Track    [4.932988 -0.003006]    Track 2 at 49900.000000 m
1.140000 sec:     Track    [0.000982 -0.003077]    Track 1 at 29800.000000 m
1.170000 sec:     Track    [4.944549 -0.004130]    Track 2 at 50000.000000 m
1.240000 sec:     Track    [-0.000250 -0.003156]    Track 1 at 29800.000000 m
1.340000 sec:     Track    [0.000101 -0.005774]    Track 1 at 29800.000000 m
1.440000 sec:     Track    [0.000461 -0.003104]    Track 1 at 29800.000000 m
1.540000 sec:     Track    [-0.000965 -0.003565]    Track 1 at 29800.000000 m
1.640000 sec:     Track    [-0.001041 -0.003305]    Track 1 at 29800.000000 m
1.670000 sec:     Track    [4.979614 -0.000920]    Track 2 at 50000.000000 m
1.740000 sec:     Track    [-0.000550 -0.003633]    Track 1 at 29800.000000 m
1.840000 sec:     Track    [-0.002676 -0.003713]    Track 1 at 29800.000000 m
1.940000 sec:     Track    [-0.005176 -0.003055]    Track 1 at 29800.000000 m
```
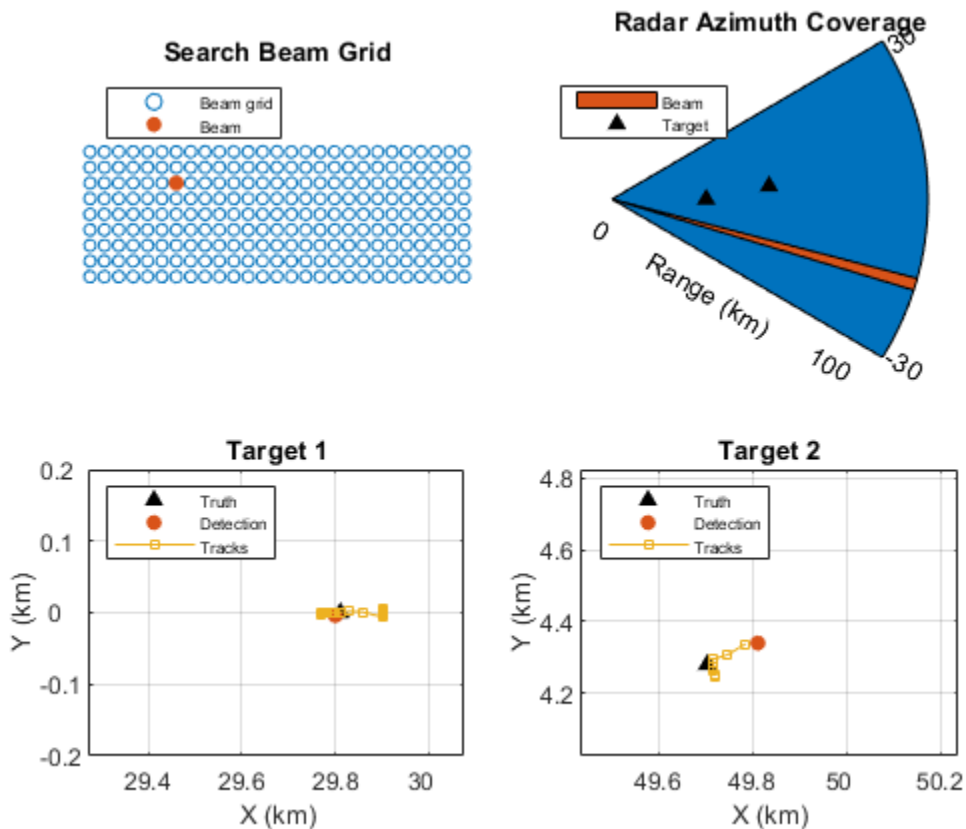

Resource Distribution Analysis

This section of the example shows how the radar resource is distributed among different tasks. This figure shows how the multifunction radar system in this example distributes its resources between search and track.

```
L = 10;
searchpercent = sum(buffer(jobload.num_search_job,L,L-1,'nodelay'))/L;
trackpercent  = sum(buffer(jobload.num_track_job,L,L-1,'nodelay'))/L;
```

```
figure;
plot((1:numel(searchpercent))*L*dwelltime,[searchpercent(:) trackpercent(:)]);
xlabel('Time (s)');
ylabel('Job Percentage');
title('Resource Distribution between Search and Track');
legend('Search','Track','Location','best');
grid on;
```
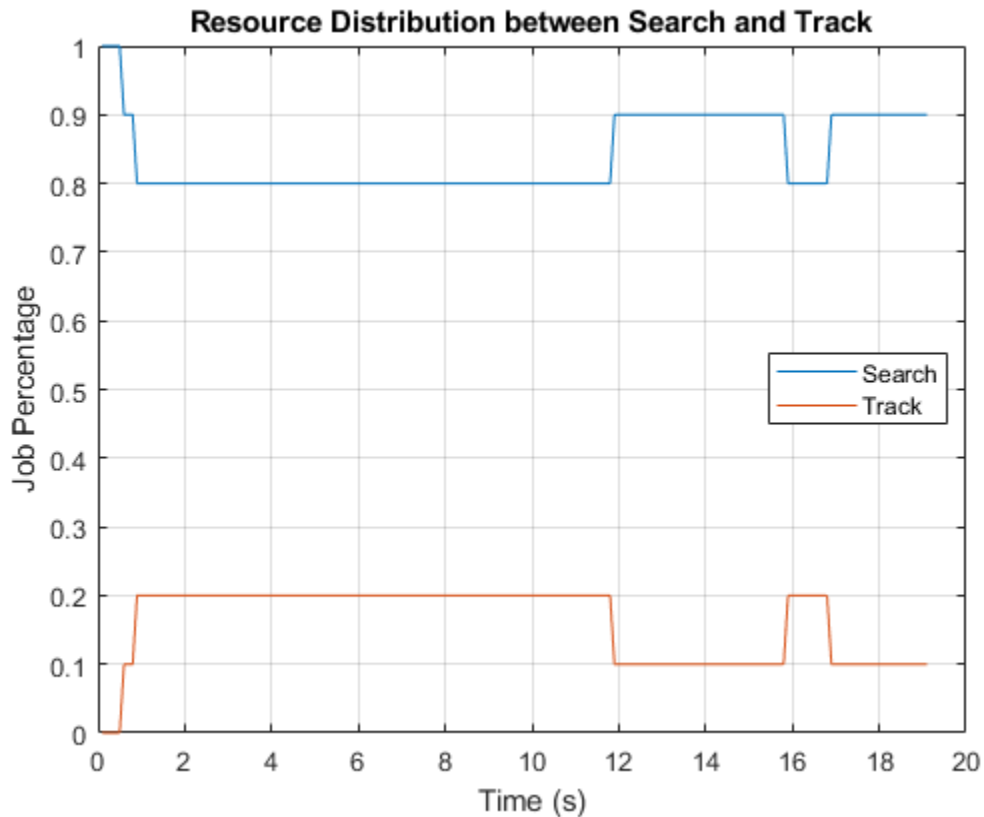

The figure suggests that, at the beginning of the simulation, all resources are spent on search. Once the targets are detected, the radar resources get split into 80% and 20% between search and track, respectively. However, once the second target gets farther away, more resources are freed up for search. The track load increases briefly when the time arrives to track the second target again.

Summary

This example introduces the concept of resource management and task scheduling in a multifunctional phased array radar system. It shows that, with the resource management component, the radar acts as a closed loop system. Although the multifunction radar in this example deals only with search and track tasks, the concept can be extended to more realistic situations where other functions, such as self-check and communication, are also involved.

References

[1] Walter Weinstock, "Computer Control of a Multifunction Radar", *Practical Phased Array Antenna Systems*, Lex Book, 1997

Appendices

These helper functions model the radar resource management workflow.

currentRadarTask

The function `currentRadarTask` compares the jobs in the search queue and the track queue and selects the job with the highest priority to execute.

```matlab
function [currentjob,jobq] = currentRadarTask(jobq,current_time)

searchq    = jobq.SearchQueue;
trackq     = jobq.TrackQueue;
searchidx  = jobq.SearchIndex;
num_trackq_items = jobq.NumTrackJobs;

% Update search queue index
searchqidx = mod(searchidx-1,numel(searchq))+1;

% Find the track job that is due and has the highest priority
readyidx = find([trackq(1:num_trackq_items).Time]<=current_time);
[~,maxpidx] = max([trackq(readyidx).Priority]);
taskqidx = readyidx(maxpidx);

% If the track job found has a higher priority, use that as the current job
% and increase the next search job priority since it gets postponed.
% Otherwise, the next search job due is the current job.
if ~isempty(taskqidx) && trackq(taskqidx).Priority >= searchq(searchqidx).Priority
    currentjob = trackq(taskqidx);
    for m = taskqidx+1:num_trackq_items
        trackq(m-1) = trackq(m);
    end
    num_trackq_items = num_trackq_items-1;
    searchq(searchqidx).Priority = searchq(searchqidx).Priority+100;
else
    currentjob = searchq(searchqidx);
    searchidx = searchqidx+1;

end

jobq.SearchQueue  = searchq;
jobq.SearchIndex  = searchidx;
jobq.TrackQueue   = trackq;
jobq.NumTrackJobs = num_trackq_items;
```

generateEchos

The function `generateEchos` simulates the complex (I/Q) baseband representation of the target echo received at the radar.

```matlab
function [xrsint,xrdazint,xrdelint,mfradar] = generateEchos(mfradar,env,current_job)

% Number of pulses and operating frequency
Npulses = 10;
fc = mfradar.TxAnt.OperatingFrequency;
```

```matlab
for m = 1:Npulses
    % Transmit weights
    w  = mfradar.STV(fc,current_job.BeamDirection);

    % Simulate echo
    advance(env.Scene);
    xr1 = mfradar.Radar(targetPoses(env.RadarPlatform),(m-1)/mfradar.Wav.PRF,conj(w));

    % Monopulse
    [xrs,xrdaz,xrdel] = mfradar.RxFeed(xr1,current_job.BeamDirection);

    % Pulse integration
    inuseflag = zeros(size(xr1,1),1);
    inuseflag(1:round(mfradar.Wav.DutyCycle/mfradar.Wav.PRF*mfradar.Wav.SampleRate))=1;
    if m == 1
        xrsint   = mfradar.Rx(xrs,~(inuseflag>0));
        xrdazint = mfradar.Rx(xrdaz,~(inuseflag>0));
        xrdelint = mfradar.Rx(xrdel,~(inuseflag>0));
    else
        xrsint   = xrsint+mfradar.Rx(xrs,~(inuseflag>0));
        xrdazint = xrdazint+mfradar.Rx(xrdaz,~(inuseflag>0));
        xrdelint = xrdelint+mfradar.Rx(xrdel,~(inuseflag>0));
    end
end
```

generateDetections

The function `generateDetections` applies signal processing techniques on the echo to generate target detection.

```matlab
function [detection,mfradar] = generateDetections(xrsint,xrdazint,xrdelint,mfradar,current_job,cu

% Compute detection threshold
nbw         = mfradar.Rx.SampleRate/(mfradar.Wav.SampleRate/mfradar.Wav.SweepBandwidth);
npower      = noisepow(nbw,mfradar.Rx.NoiseFigure,mfradar.Rx.ReferenceTemperature);
pfa         = 1e-6;
threshold   = npower * db2pow(npwgnthresh(pfa,1,'noncoherent'));
arraysz     = mfradar.TxAnt.Sensor.Size(1);
ant_snrgain = pow2db(arraysz^2);
mfcoeff     = getMatchedFilter(mfradar.Wav);
mfgain      = pow2db(norm(mfcoeff)^2);
threshold   = threshold * db2pow(mfgain+2*ant_snrgain);
threshold   = sqrt(threshold);

tgrid   = unigrid(0,1/mfradar.Wav.SampleRate,1/mfradar.Wav.PRF,'[)');
rgates  = mfradar.TxAnt.PropagationSpeed*tgrid/2;

% Matched filtering and time varying gain
xrsmf = mfradar.TVG(mfradar.MF(xrsint));

% Detection in range and angle estimation via monopulse
if any(abs(xrsmf)>threshold)
    [~,tgtidx] = findpeaks(abs(xrsmf),'MinPeakHeight',threshold,...
        'Sortstr','Descend','NPeaks',1);
    rng_est = rgates(tgtidx-(numel(mfcoeff)-1));
```

```
        ang_est = mfradar.DOA(xrsint(tgtidx-1),xrdazint(tgtidx-1),xrdelint(tgtidx-1),current_job.Bean
        % Form the detection object.
        measNoise = diag([0.1, 0.1, 150].^2);            % Measurement noise matrix
        detection = objectDetection(current_time,...
            [ang_est(1);ang_est(2);rng_est], 'MeasurementNoise', measNoise,...
            'MeasurementParameters',struct('Frame','spherical', 'HasVelocity', false));
    else
        detection = objectDetection.empty;
    end

    if current_time < 0.3 || strcmp(current_job.JobType,'Track')
        fprintf('\n%f sec:\t%s\t[%f %f]',current_time,current_job.JobType,current_job.BeamDirection(1
            current_job.BeamDirection(2));
    end
```

updateTrackAndJobQueue

The function `updateTrackAndJobQueue` tracks the detection and then passes tracks to the radar manager to update the track task queue.

```
function [jobq,allTracks,mfradar] = updateTrackAndJobQueue(detection,jobq,mfradar,current_job,cur

trackq          = jobq.TrackQueue;
num_trackq_items = jobq.NumTrackJobs;

% Execute current job
switch current_job.JobType
    case 'Search'
        % For search job, if there is a detection, establish tentative
        % track and schedule a confirmation job
        if ~isempty(detection)
            ang_est = detection.Measurement(1:2);
            rng_est = detection.Measurement(3);
            if ~mfradar.IsTrackerInitialized
                [~,~,allTracks] = mfradar.Tracker(detection,current_time,uint32([]));
                mfradar.IsTrackerInitialized = true;
            else
                [~,~,allTracks] = mfradar.Tracker(detection,current_time,uint32([]));
            end
            num_trackq_items = num_trackq_items+1;
            trackq(num_trackq_items) = struct('JobType','Confirm','Priority',2000,...
                'BeamDirection',ang_est,'WaveformIndex',1,'Time',current_time+dwellinterval,...
                'Range',rng_est,'TrackID',allTracks(~[allTracks.IsConfirmed]).TrackID);
            if current_time < 0.3 || strcmp(current_job.JobType,'Track')
                fprintf('\tTarget detected at %f m',rng_est);
            end
        else
            allTracks = [];
        end

    case 'Confirm'
        % For confirm job, if the detection is confirmed, establish a track
        % and create a track job corresponding to the revisit time
        if ~isempty(detection)
            trackid = current_job.TrackID;
```

```matlab
            [~,~,allTracks] = mfradar.Tracker(detection,current_time,trackid);

            rng_est = detection.Measurement(3);
            if rng_est >= 50e3
                updateinterval = 0.5;
            else
                updateinterval = 0.1;
            end
            revisit_time = current_time+updateinterval;
            predictedTrack = predictTracksToTime(mfradar.Tracker,trackid,revisit_time);
            xpred = predictedTrack.State([1 3 5]);
            [phipred,thetapred,rpred] = cart2sph(xpred(1),xpred(2),xpred(3));
            num_trackq_items = num_trackq_items+1;
            trackq(num_trackq_items) = struct('JobType','Track','Priority',3000,...
                'BeamDirection',rad2deg([phipred;thetapred]),'WaveformIndex',1,'Time',revisit_tim
                'Range',rpred,'TrackID',trackid);
            if current_time < 0.3 || strcmp(current_job.JobType,'Track')
                fprintf('\tCreated track %d at %f m',trackid,rng_est);
            end
        else
            allTracks = [];
        end

    case 'Track'
        % For track job, if there is a detection, update the track and
        % schedule a track job corresponding to the revisit time. If there
        % is no detection, predict and schedule a track job sooner so the
        % target is not lost.
        if ~isempty(detection)
            trackid = current_job.TrackID;
            [~,~,allTracks] = mfradar.Tracker(detection,current_time,trackid);

            rng_est = detection.Measurement(3);
            if rng_est >= 50e3
                updateinterval = 0.5;
            else
                updateinterval = 0.1;
            end

            revisit_time = current_time+updateinterval;
            predictedTrack = predictTracksToTime(mfradar.Tracker,trackid,revisit_time);
            xpred = predictedTrack.State([1 3 5]);
            [phipred,thetapred,rpred] = cart2sph(xpred(1),xpred(2),xpred(3));
            num_trackq_items = num_trackq_items+1;
            trackq(num_trackq_items) = struct('JobType','Track','Priority',3000,...
                'BeamDirection',rad2deg([phipred;thetapred]),'WaveformIndex',1,'Time',revisit_tim
                'Range',rpred,'TrackID',trackid);

            if current_time < 0.3 || strcmp(current_job.JobType,'Track')
                fprintf('\tTrack %d at %f m',trackid,rng_est);
            end
        else
            trackid = current_job.TrackID;
            [~,~,allTracks] = mfradar.Tracker(detection,current_time,trackid);

            updateinterval = 0.1;  % revisit sooner
            revisit_time = current_time+updateinterval;
            predictedTrack = predictTracksToTime(mfradar.Tracker,trackid,revisit_time);
```

```
            xpred = predictedTrack.State([1 3 5]);

            [phipred,thetapred,rpred] = cart2sph(xpred(1),xpred(2),xpred(3));
            num_trackq_items = num_trackq_items+1;
            trackq(num_trackq_items) = struct('JobType','Track','Priority',3000,...
                'BeamDirection',rad2deg([phipred;thetapred]),'WaveformIndex',1,'Time',revisit_tim
                'Range',rpred,'TrackID',trackid);

            if current_time < 0.3 || strcmp(current_job.JobType,'Track')
                fprintf('\tNo detection, track %d predicted',current_job.TrackID);
            end
        end

    end

    jobq.TrackQueue   = trackq;
    jobq.NumTrackJobs = num_trackq_items;
```

PRF Agility Based on Target Detection

In radar operation, it is often necessary to adjust the operation mode based on the target return. This example shows how to model a radar that changes its pulse repetition frequency (PRF) based on the radar detection.

Available Example Implementations

This example includes the Simulink® model: PRF Agility Based on Radar Detection: slexPRFSchedulingExample.slx

Dynamic PRF Selection Based on Radar Detection

This model simulates a monostatic radar that searches for targets with an unambiguous range of 5 km. If the radar detects a target within 2 km, then it will switch to a higher PRF to only look for targets with 2 km range and enhance its capability to detect high speed targets.

PRF Agility Based on Radar Detection

Copyright 2018 The MathWorks Inc.
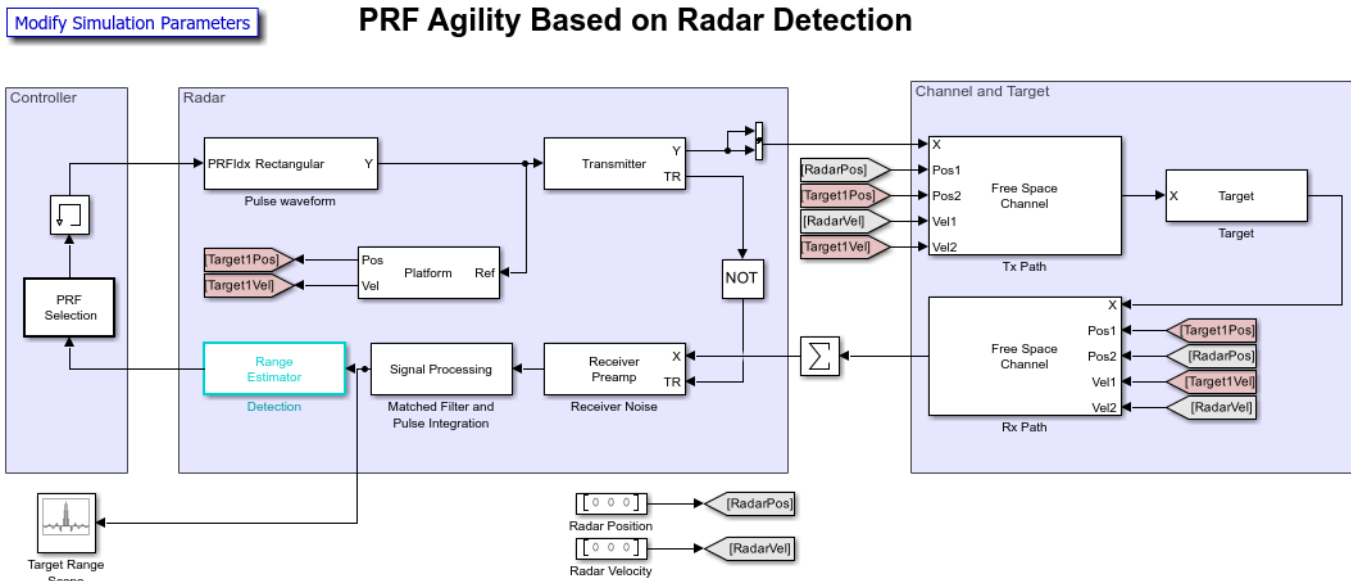
The system is very similar to what is used in the "Simulating Test Signals for a Radar Receiver in Simulink" example with the following differences:

1 The waveform block is no longer a source block. Instead, it takes an input, `idx`, to select which PRF to use. The available PRF values are specified in the PRF parameter of the waveform dialog box.

2 Each time a waveform is transmitted, its corresponding PRF also sets the time when the next pulse should be transmitted.

3 There is now a controller to determine which PRF to use for the next transmission. At the end of the signal processing chain, the target range is estimated. The controller will use this information to decide which PRF to choose for the next transmission.

4 Once the model is compiled, notice that the signal passing through the system can vary in length because of a possible change of the waveform PRF.

5 The model takes advantage of the new controllable sample time so the system runs at the appropriate time determined by the varying PRF values.

Exploring the Example

Several parameters of the model are calculated by the helper function helperslexPRFSchedulingSim. To open the function from the model, click the Modify Simulation Parameters block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs boxes. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

This figure shows the detected ranges of the targets. Target ranges are computed from the round-trip time delay of the reflected signals from the targets. When simulation starts, the radar detects two targets, one is slightly over 2 km away and the other one is at approximately 3.5 km away.
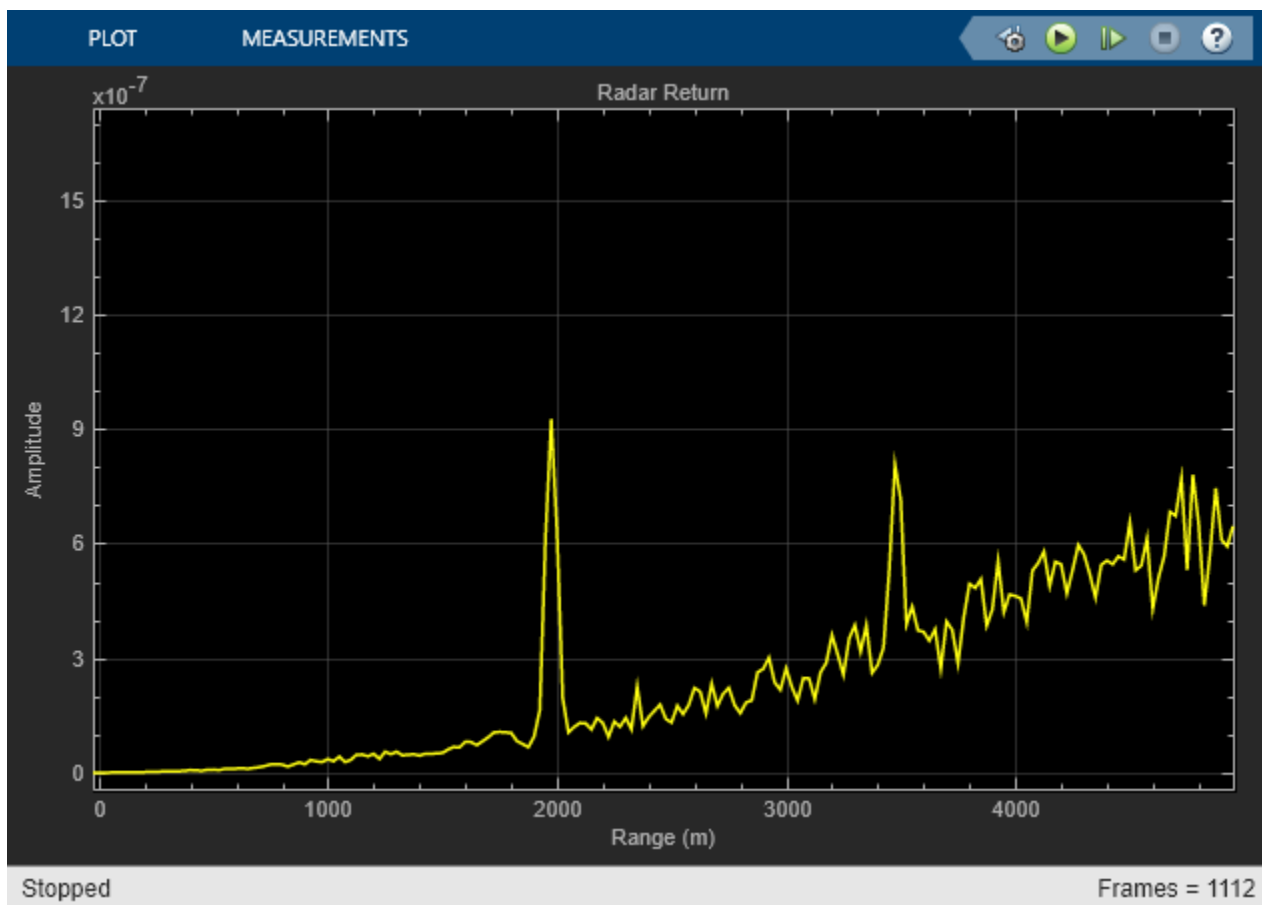
After some time, the first target moves into the 2 km zone and triggers a change of PRF. Then the received signal only covers the range up to 2 km. The display is zero padded to ensure that the plot limits do not change. Notice that the target at 3.5 km gets folded to the 1.5 km range due to range ambiguity.
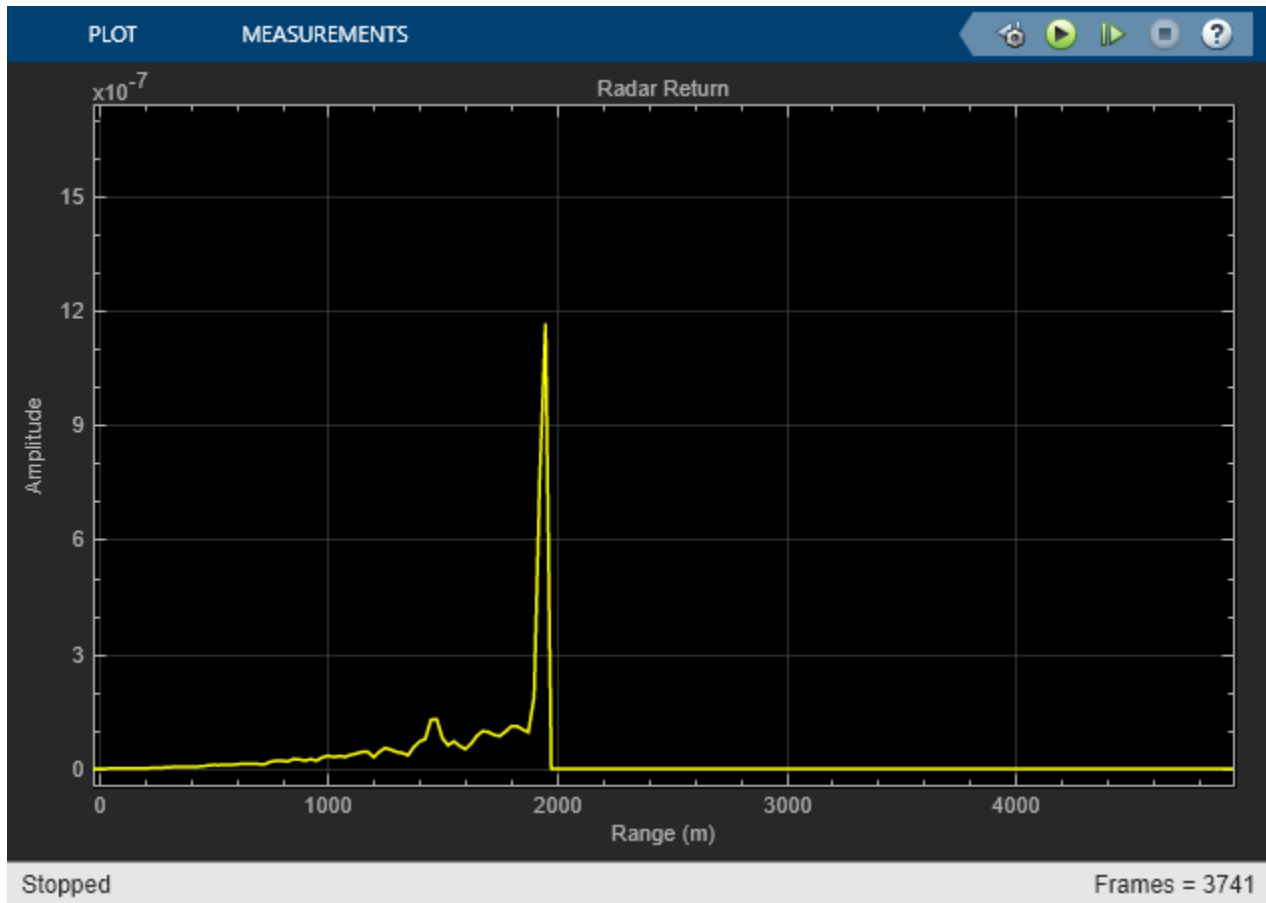
Summary

This example shows how to build a radar system in Simulink® that dynamically changes its PRF based on the target detection range. A staggered PRF system can be modeled similarly.

Interference Mitigation Using Frequency Agility Techniques

This example shows how to model frequency agility techniques to counter the effects of interference in radar, communications, and electronic warfare (EW) systems. This example uses Simulink® to create a scenario with a ground-based radar and an approaching aircraft that can also emit jamming signals. For a similar example using MATLAB®, see "Frequency Agility in Radar, Communications, and EW Systems" on page 1-217.

Introduction

In this model, a phased array radar is designed to detect an approaching aircraft. The aircraft is equipped with a jammer which can intercept the radar signal and transmit a spoofing signal back to confuse the radar. On the other hand, the radar system is capable of transmitting a waveform with different operating frequencies to mitigate the jamming effect. The model includes blocks for waveform generation, signal propagation, and radar signal processing. The example shows how the radar and jammer interact and gain advantage over each other.
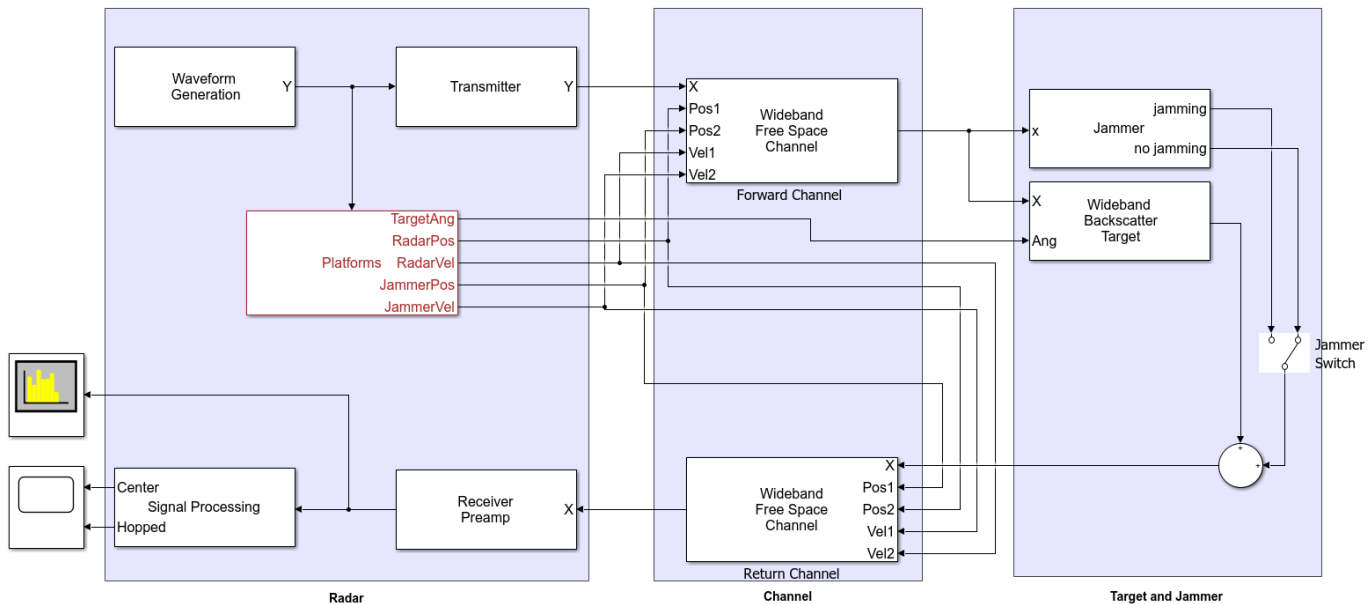
Copyright 2017-2023 The MathWorks Inc.

The radar is operating at 300 MHz with a sampling rate of 2 MHz. The radar is located at the origin and is assumed to be stationary. The target is located about 10 km away and approaching at approximately 100 meters per second.

Waveform Generation

The Waveform Generation subsystem includes a pulse waveform library containing a linear frequency modulated (LFM) waveform with different configurations. By varying the input index to the pulse waveform library, a frequency hopped waveform at a shifted center frequency is generated. Therefore, the radar system is capable of switching the transmit waveform either following a fixed schedule or when it detects jamming signals. This example assumes that the waveform can be

generated for two different frequencies, referred to as center band and hopped band. The center band is the subband around the carrier frequency, and the hopped band is the subband located quarter bandwidth above the carrier.
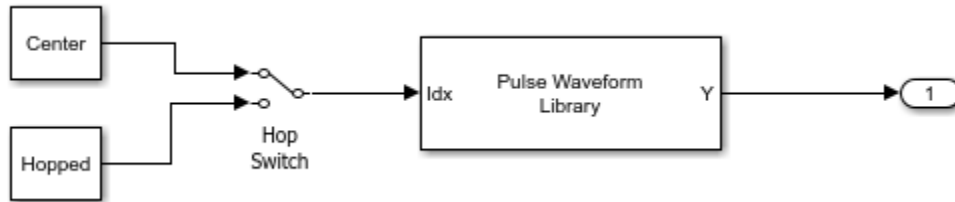
Propagation Channels

The signal propagation is modeled for both the forward channel and the return channel. Once the transmitted signal hits the target aircraft, the reflected signal travels back to the radar system via the return channel. In addition, the jammer analyzes the incoming signal and sends back a jamming signal to confuse the radar system. That jamming signal is also propagated through the return channel. Because different signals can occupy different frequency bands, wideband propagation channels are used.

Signal Processing

The radar receives both the target return and the jamming signal. Upon receiving the signal, a series filters are used to extract the signal from different bands. In this example, there are two series of filters to extract the signal from the center band and the hopped band. The signal in each band then passes through the corresponding matched filter to improve the signal-to-noise ratio (SNR) and be ready for detection.
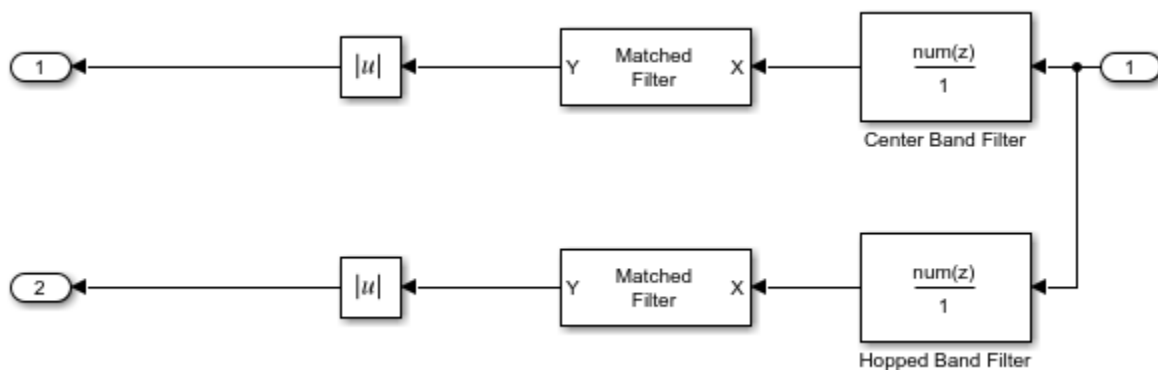
Exploring the Example

Several parameters of the model are calculated by the helper function `helperslexFrequencyAgilityParam`. To open the function from the model, click the Modify Simulation Parameters block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialog boxes. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

First run the model for the case when there is no jamming signal. The scope shows that there is one strong echo in the center band with a delay of approximately 67 microseconds, which corresponds to a target range of 10 km. Therefore, the target is correctly detected. Meanwhile, there is no return detected in the hopped band.
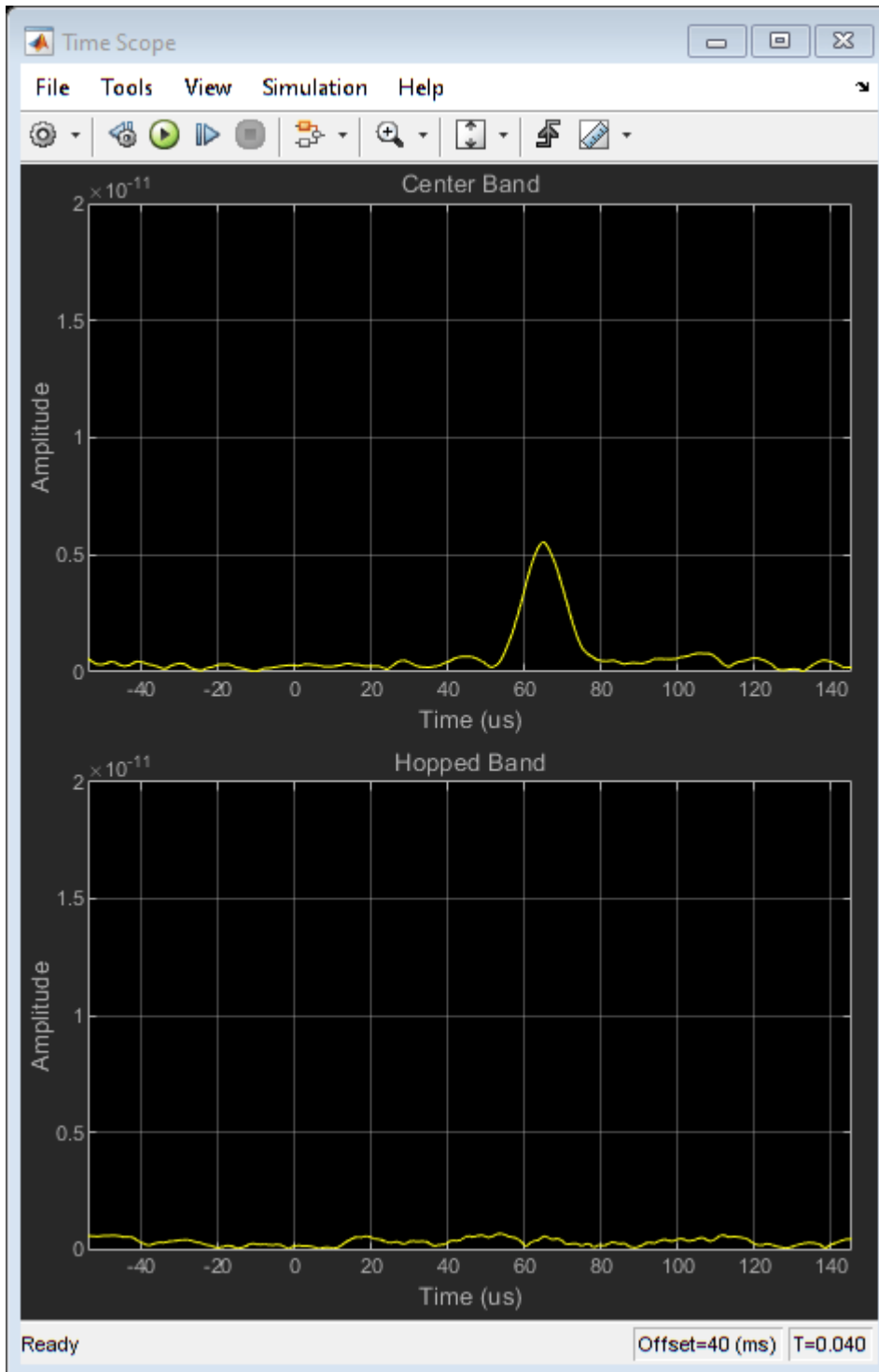
The spectrum analyzer shows that the received signal occupies the center band.

Now enable the jammer by clicking the Jammer Switch block. In this situation, the target intercepts the signal, amplifies it, and then sends it back with a delay corresponding to a different range. As a result, the scope now shows two returns in the center band. The true target return is still at the old

position, but the ghost return generated by the jammer appears stronger and closer to the radar, so the radar is likely to be confused and assign precious resource to track this fake target.
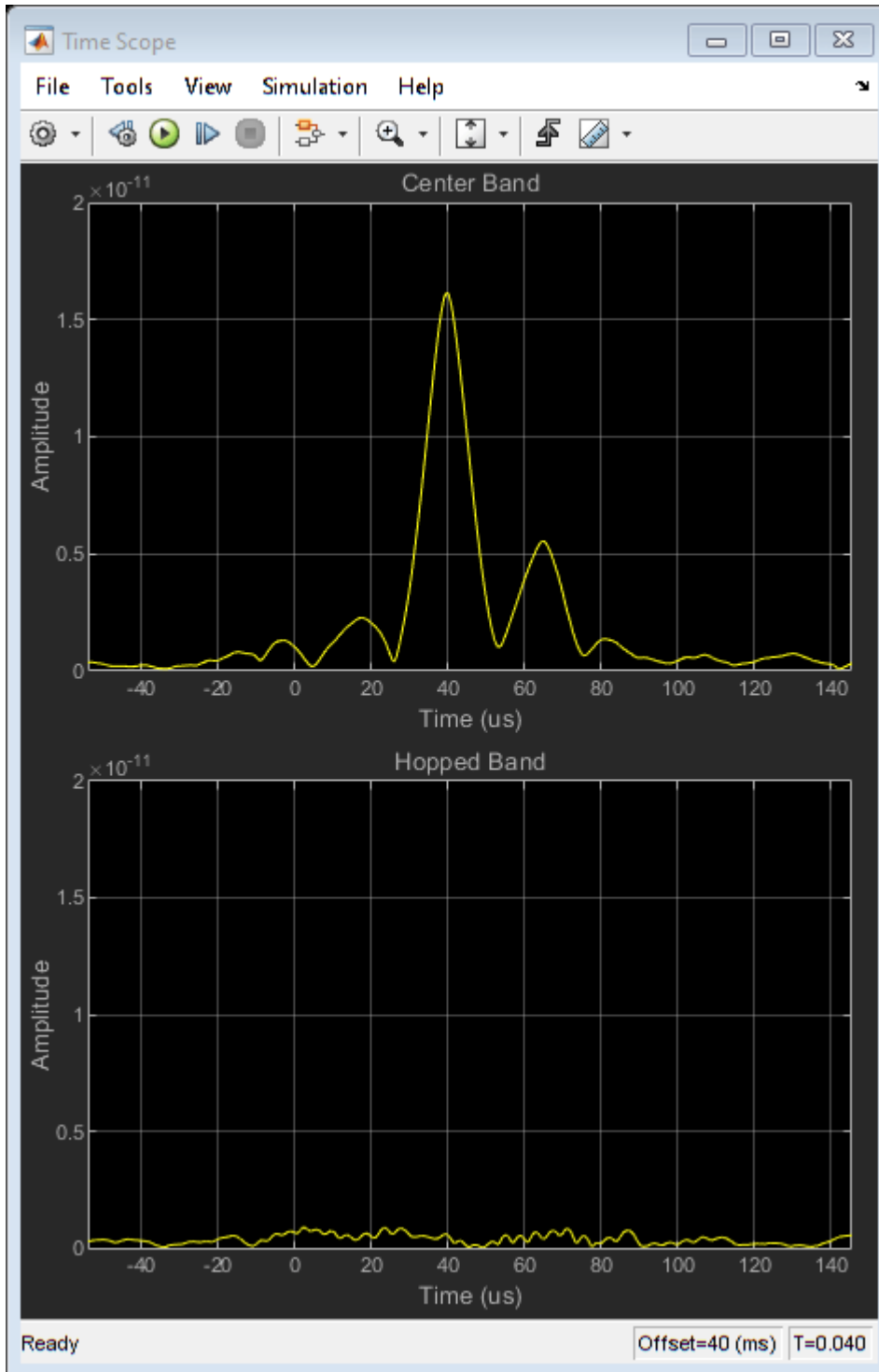
Note that both the jammer signal and the target return are in the center band, as shown in the spectrum analyzer.

If the radar has a preset frequency hopping schedule or is trained to know that it might have been confused by a jamming signal, it can switch to a different frequency band to operate. You can simulate such a by clicking the Hop Switch block so the radar signal is transmitted in the hopped band.
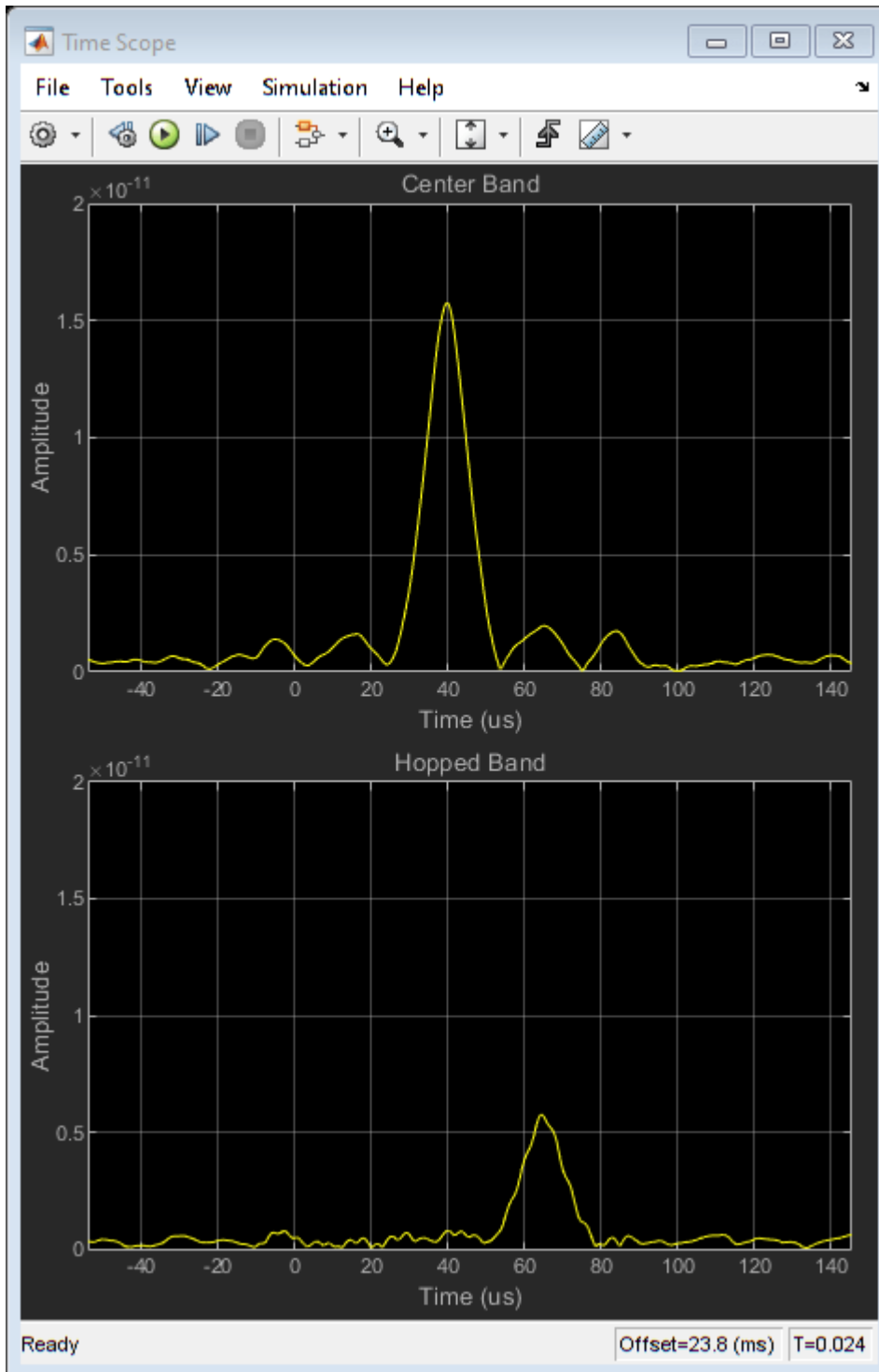
Because the radar now operates in the hopped band, the target echo is also in the hopped band. From the scope, the target echo is at the appropriate delay in the hopped band. Meanwhile, the jammer is yet to figure out the new operating band of the radar so the jamming signal still appears in the center band. Yet the jamming signal can no longer fool the radar.

The spectrum analyzer shows that the received signal now occupies two bands.

Summary

This example models a radar system detecting a target equipped with a jammer. It shows how frequency agility techniques can be used to mitigate the jamming effect.

Frequency Agility in Radar, Communications, and EW Systems

This example shows how to model frequency agility in radar, communications, and electronic warfare systems to counter the effects of interference.

Introduction

Active electronically-steered phased array systems can support multiple applications using the same array hardware. These applications may include radar, EW, and communications. However, the RF environments that these types of systems operate in are complex and sometimes hostile. For example, a repeater jammer can repeat the received radar signal and retransmit it to confuse the radar. In some literature, this is also called spoofing. Frequency agility can be an effective technique to counter the signals generated from interference sources and help support effective operations of these systems.

In this example, we first set up a scenario with a stationary monostatic radar and a moving aircraft target. The aircraft then generates a spoofing signal which confuses the radar. Once the radar detects the jamming source, frequency agility techniques can be employed which allows the radar to overcome the interference.

System Simulation In an Interference Free Environment

Assume an X-band monostatic radar at the origin.

```
fc = 10e9;        % Define frequency of operation for X-band Radar
fs = 2e6;         % Define sample rate for the systems
c = 3e8;
lambda = c/fc;

radar_pos = [0;0;0];
radar_vel = [0;0;0];
```

The radar receiver, which can also function as an EW receiver, is a 64-element (8-by-8) Uniform Rectangular Array (URA) with half-wavelength spacing.

```
% Array specification for the radar
taper = taylorwin(8);
taperURA = taper.*taper';
antenna = phased.URA('Element',phased.CosineAntennaElement, ...
    'Size',[8 8],'ElementSpacing',[lambda/2 lambda/2], ...
    'Taper',taperURA);
```

The beam pattern of the array is shown in the following figure.

```
pattern(antenna,fc,'Type','powerdb')
```
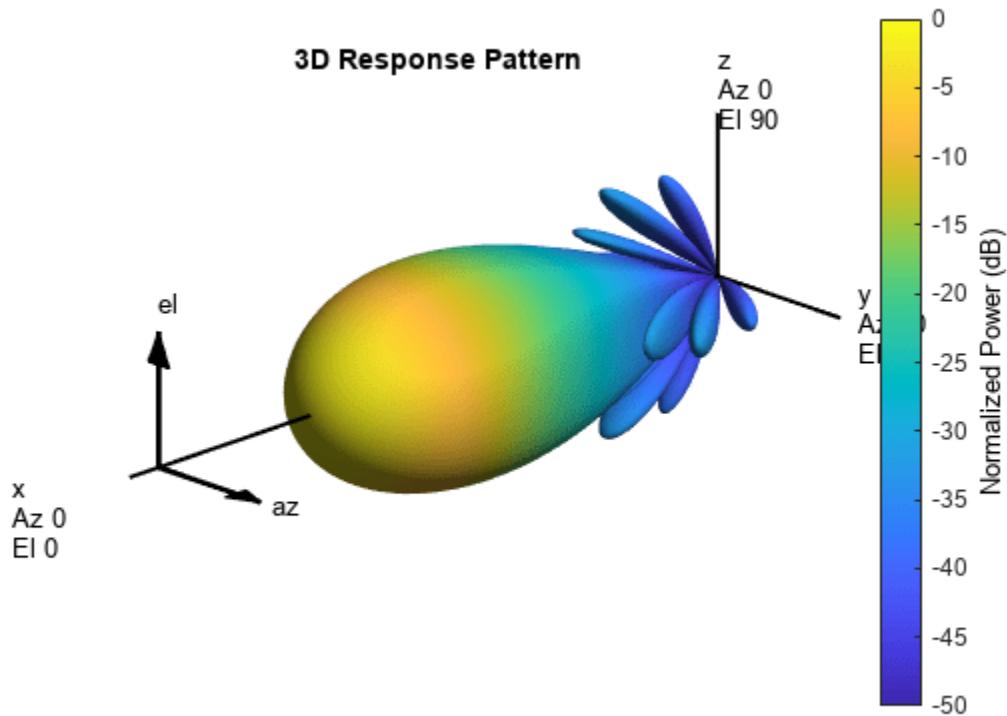
The radar transmits linear FM pulses. Set transmitter and receiver specifications.

```
wav = phased.LinearFMWaveform('SampleRate',fs, ...
    'PulseWidth', 10e-5, 'SweepBandwidth', 1e5,'PRF',4000, ...
    'FrequencyOffsetSource','Input port');
tx = phased.Transmitter('Gain',20,'PeakPower',500);
txArray = phased.WidebandRadiator('SampleRate',fs, ...
    'Sensor',antenna,'CarrierFrequency', fc);
rxArray = phased.WidebandCollector('SampleRate',fs, ...
    'Sensor',antenna,'CarrierFrequency',fc);
rxPreamp = phased.ReceiverPreamp('Gain',10,'NoiseFigure',5, ...
    'SampleRate',fs);
```

Specify the environment and target. This example uses wideband propagation channels to propagate waveforms with different carrier frequencies.

```
target = phased.RadarTarget('MeanRCS',100,'OperatingFrequency',fc);

target_pos = [8000;1000;1000];
target_vel = [100;0;0];

% Outgoing and incoming channel
envout = phased.WidebandFreeSpace('TwoWayPropagation',false, ...
    'SampleRate', fs,'OperatingFrequency',fc,'PropagationSpeed',c);
envin = phased.WidebandFreeSpace('TwoWayPropagation',false, ...
    'SampleRate', fs,'OperatingFrequency',fc,'PropagationSpeed',c);
```

This example uses two one-way propagation channels because the jamming signal only propagates through the return channel.

Simulate the received echo.

```
rng(2017);
[tgtRng, tgtAng] = rangeangle(target_pos, radar_pos);
x = wav(0);                                                 % waveform
xt = tx(x);                                                 % transmit
xtarray = txArray(xt, tgtAng);                             % radiate
yp = envout(xtarray,radar_pos,target_pos,radar_vel,target_vel); % propagate
yr = target(yp);                                            % reflect
ye = envin(yr,target_pos,radar_pos,target_vel,radar_vel);  % propagate
yt = rxArray(ye,tgtAng);                                    % collect
yt = rxPreamp(yt);                                          % receive
```

Estimate the direction of arrival using a 2-D beam scan and use the estimated azimuth and elevation angles to direct the beamformer.

```
estimator = phased.BeamscanEstimator2D('SensorArray',antenna,...
    'DOAOutputPort',true, ...
    'OperatingFrequency', fc, ...
    'NumSignals',1, ...
    'AzimuthScanAngles',-40:40, ...
    'ElevationScanAngles',-60:60);

[~,doa] = estimator(yt);

beamformer = phased.SubbandPhaseShiftBeamformer('SensorArray',antenna, ...
    'OperatingFrequency',fc,'DirectionSource','Input port', ...
    'SampleRate',fs,'WeightsOutputPort',true);

[ybf,~] = beamformer(yt,doa);
```

Pass the beamformed signal through a matched filter and detector.

```
mfcoeff1 = getMatchedFilter(wav);
mf1 = phased.MatchedFilter('Coefficients',mfcoeff1);
```

The following figure shows that the target produces a dominant peak in the received signal.

```
y1 = mf1(ybf);
nSamples = wav.SampleRate/wav.PRF;
t = ((0:nSamples-1)-(numel(mfcoeff1)-1))/fs;
r = t*c/2;
plot(r/1000,abs(y1),'-')
grid on
xlabel('Range (km)')
ylabel('Pulse Compressed Signal Magnitude')
```
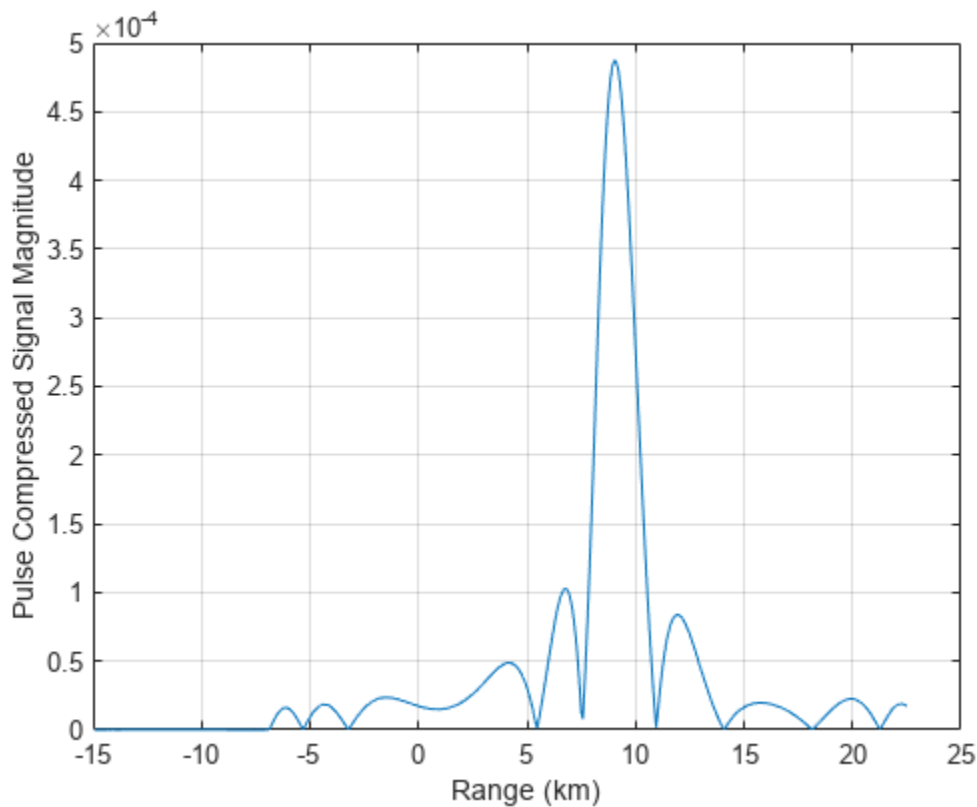
Signal Analysis of Target and Jammer

The radar works very well in the previous section. However, in a complex environment, interference can affect the radar performance. The interference may be from other systems, such as wireless communication signals, or jamming signals. Modern radar systems must be able to operate in such environments.

A phased array radar can filter out interference using spatial processing. If the target and the jamming source are well-separated in angular space, beamforming can be an effective way to suppress the jammer. More details can be found in the "Array Pattern Synthesis Part I: Nulling, Windowing, and Thinning" example.

This example focuses on the situation where the target and the interference are closely located so that spatial processing cannot be used to separate the two. Consider the case where the target aircraft can determine the characteristics of the signal transmitted from the radar and use that information to generate a pulse that will confuse the radar receiver. This is a common technique used in jamming or spoofing to draw the radar away from the true target.

Display the detected signal characteristics.

```
pw = (pulsewidth(abs(yp), fs));
prf = round(1/pulseperiod(abs([yp;yp]), fs));
bw = obw(yp,fs,[],95);
fprintf('Waveform Characteristics:\n');
```

```
Waveform Characteristics:
```

```matlab
fprintf('Pulse width:\t\t%f\n',pw);
```

Pulse width: 0.000100

```matlab
fprintf('PRF:\t\t\t%f\n',prf);
```

PRF: 4000.000000

```matlab
fprintf('Sweep bandwidth:\t%f\n',bw);
```

Sweep bandwidth: 112041.098255

The jammer needs time to do this analysis and prepare the jamming signal, so it is hard to create an effective spoofing signal right away. But generally within several pulse intervals the jamming signal is ready and the jammer can put it in an arbitrary position in a pulse to make the spoofing target look closer or farther compared to the true target. It is also worth noting that with the latest hardware, the time needed to estimate the characteristics of the signal decreases dramatically.

Assume the jammer wants to put the signal at about 5.5 km, the jammer could transmit the jamming signal at the right moment to introduce the corresponding delay. In addition, because this is a one way propagation from the jammer to the radar, the required power is much smaller. This is indeed what makes jamming very effective as it does not require much power to blind the radar.

```matlab
jwav = phased.LinearFMWaveform('SampleRate',fs, ...
    'PulseWidth',pw,'SweepBandwidth',bw,'PRF',prf);
xj = jwav();
Npad = ceil(5500/(c/fs));
xj = circshift(xj,Npad);        % pad zero to introduce corresponding delay
txjam = phased.Transmitter('Gain',0,'PeakPower',5);
xj = txjam(xj);
ye = envin(yr+xj,target_pos,radar_pos,target_vel,radar_vel);
yt = rxArray(ye,tgtAng);
yt = rxPreamp(yt);
ybfj = beamformer(yt,doa);
y1j = mf1(ybfj); % Jammer plus target return

plot(r/1000,abs(y1j))
grid on
xlabel('Range (km)')
ylabel('Magnitude')
title('Pulse Compressed Return From Target And Jammer')
```
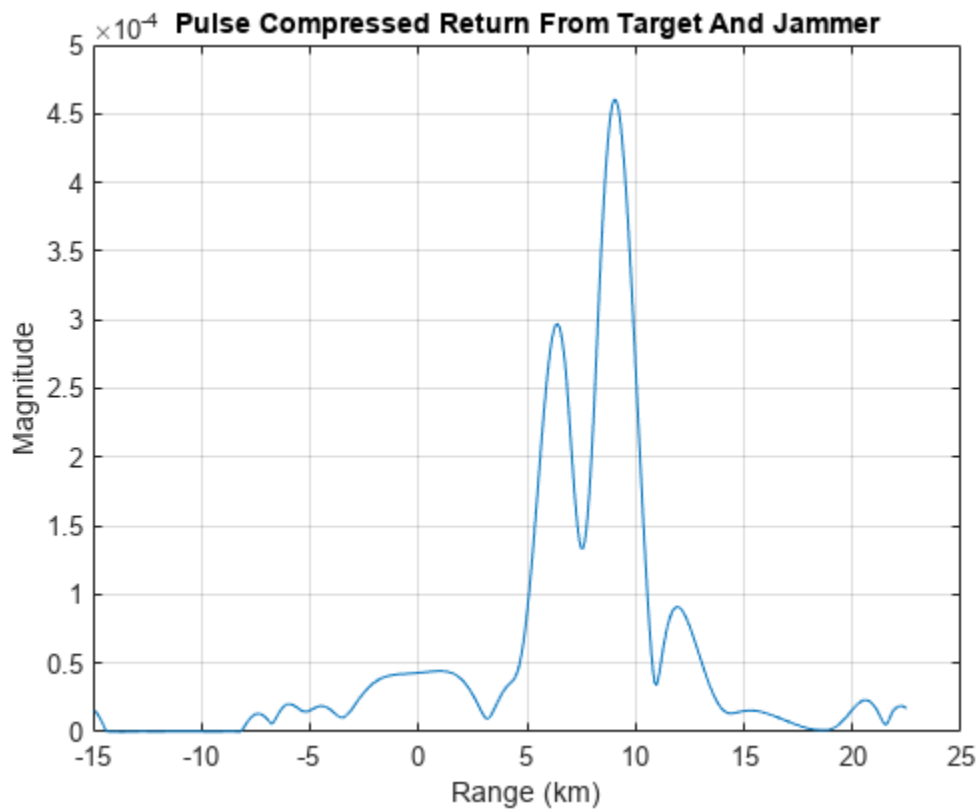
The received signal now contains both the desired target return and the jamming signal. In addition, the jamming signal appears to be closer. Therefore, the radar is more likely to lock in on the closest target thinking that it is the most prominent threat and spend less resources on the true target.

Frequency Agility to Counter Interference

One possible approach to mitigate the jamming effect at the radar receiver is to adopt a predefined frequency hopping schedule. In this case, the waveform transmitted from the radar can change carrier frequency from time to time. Since the hopping sequence is known only to the radar, the jammer would not be able to follow the change right away. Instead, it needs to take more time to acquire the correct carrier frequency before a new jamming signal can be generated. It also requires more advanced hardware on the jammer to be able to handle transmission of signals over a broader bandwidth. Thus, the frequency hop can create a time interval that the radar operates in without being affected by the spoofing signal. In addition, the radar can hop again before the jammer can effectively generate the spoofing signal.

In the following situation, assume that the transmitted signal hops 500 kHz from the original carrier frequency of 10 GHz. Therefore, the new waveform signal can be represented as follows.

```
deltaf = fs/4;
xh = wav(deltaf);   % hopped signal
```

This figure shows the spectrogram of both the original signal and the hopped signal. Note that the hopped signal is a shift in the frequency domain compared to the original signal.

```
pspectrum(x+xh,fs,'spectrogram')
```

Fres = 81.484 kHz, Tres = 31.5 μs
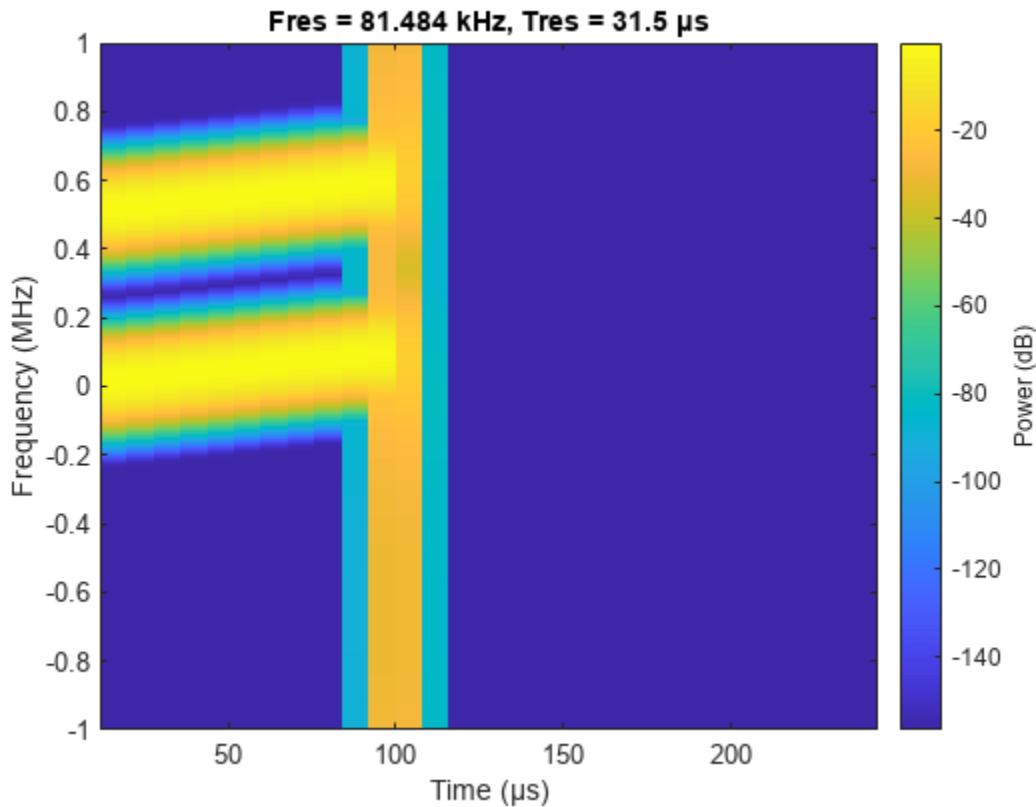
Using a similar approach outlined in the earlier sections, the radar echo can be simulated using the new waveform. Note that since jammer is not aware of this hop, the jamming signal is still the same.

```
xth = tx(xh);
xtharray = txArray(xth, tgtAng);
yph = envout(xtharray,radar_pos,target_pos,radar_vel,target_vel);
yrh = target(yph);

yeh = envin(yrh+xj,target_pos,radar_pos,target_vel,radar_vel);
yth = rxArray(yeh,tgtAng);
yth = rxPreamp(yth);

ybfh = beamformer(yth,doa);
```

Because the hopping schedule is known to the radar, the signal processing algorithm could use that information to extract only the frequency band that around the current carrier frequency. This helps reject the interference at other bands and also improves the SNR since the noise from other bands is suppressed. In addition, when the waveform hops, the matched filter needs to be updated accordingly.

Now apply the corresponding bandpass filters and matched filters to the received signal.

First, create a bandpass filter using the signal bandwidth of the signal.

```
buttercoef = butter(9,bw/fs);
```

Then modulate the resulting bandpass filter with a carrier to obtain the bandpass filter around that carrier frequency.

1-223

```
bf2 = buttercoef.*exp(1i*2*pi*deltaf*(0:numel(buttercoef)-1)/fs);
```

Similarly, the modulate the matched filter coefficient.

```
mfcoeff2 = getMatchedFilter(wav,'FrequencyOffset',deltaf);
mf2 = phased.MatchedFilter('Coefficients',mfcoeff2);

% extract bands and apply matched filters
yb2 = mf2(filter(bf2(:),1,ybfh));

% plot the matched filtered signal
plot(r/1000,abs(yb2))
grid on
xlabel('Range (km)')
ylabel('Magnitude')
title('Pulse Compressed Signal')
```
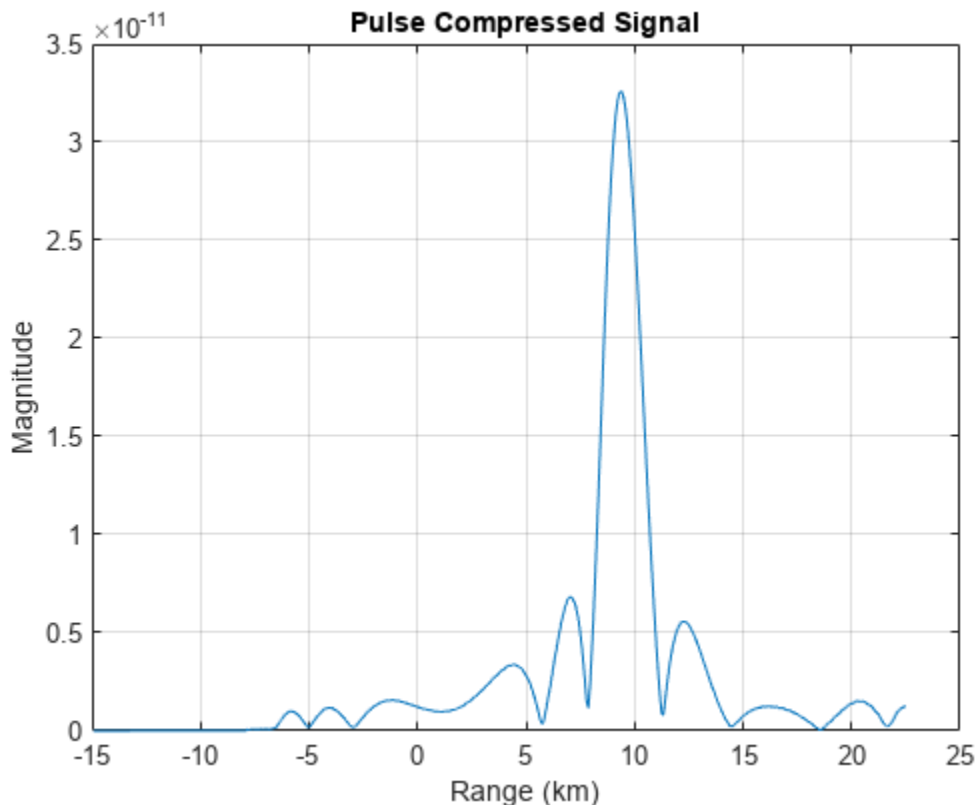

This figure shows that with the adoption of frequency hopping, the target echo and the jamming signal can be separated. Since the jammer is still in the original band, only the true target echo appears in the new frequency band that the waveform currently occupies, thus suppressing the impact of the jammer.

Summary

This example shows that adopting frequency agility can help counter the jamming effect in a complex RF environment. The example simulates a system with a frequency hopping waveform and verifies

that this technique helps the radar system to identify the true target echo without being confused by the jamming signal.

Waveform Scheduling Based on Target Detection

In radar operation, it is often necessary to adjust the operation mode based on the target return. This example shows how to model a radar that changes its pulse repetition frequency (PRF) based on the radar detection.

This example requires SimEvents®.

Available Example Implementations

This example includes one Simulink® model:

- Dynamic PRF Selection Based on Radar Detection: slexPRFSelectionSEExample.slx

Dynamic PRF Selection Based on Radar Detection

This model simulates a monostatic radar that searches for targets with an unambiguous range of 5 km. If the radar detects a target within 2 km, then it will switch to a higher PRF to only look for targets with 2 km range and enhance its capability to detect high speed targets.

```
helperslexPRFSelectionSim('openModel');
```
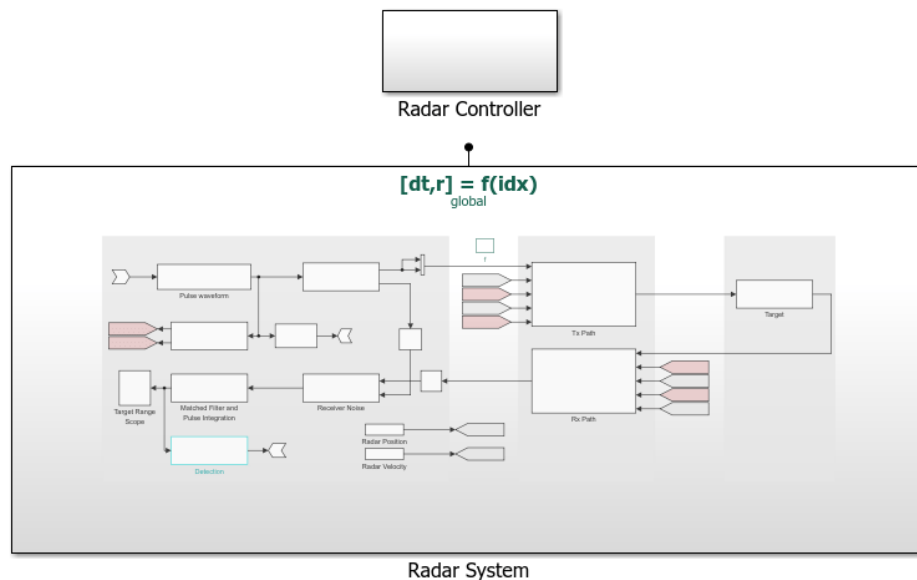

Copyright 2017 The MathWorks, Inc.

The model consists of two main subsystem, a radar system and its corresponding controller. From the top level, the radar system resides in a Simulink function block. Note that the underlying function is specified in the figure as `[dt,r] = f(idx)`. This means that the radar takes one input, `idx`, which specifies the index of selected PRF of the transmitted signal and returns two outputs: `dt`, the time the next pulse should be transmitted and `r`, the detected target range of the radar system. The radar controller, shown in the following figure, uses the detection and the time to schedule when and what to transmit next.

```
helperslexPRFSelectionSim('showController');
```
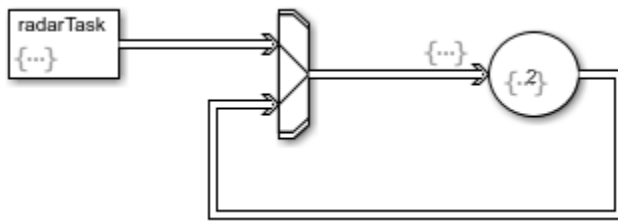

Radar System

The radar system resides in a Simulink function block and is shown in the following figure.

```
helperslexPRFSelectionSim('showRadar');
```
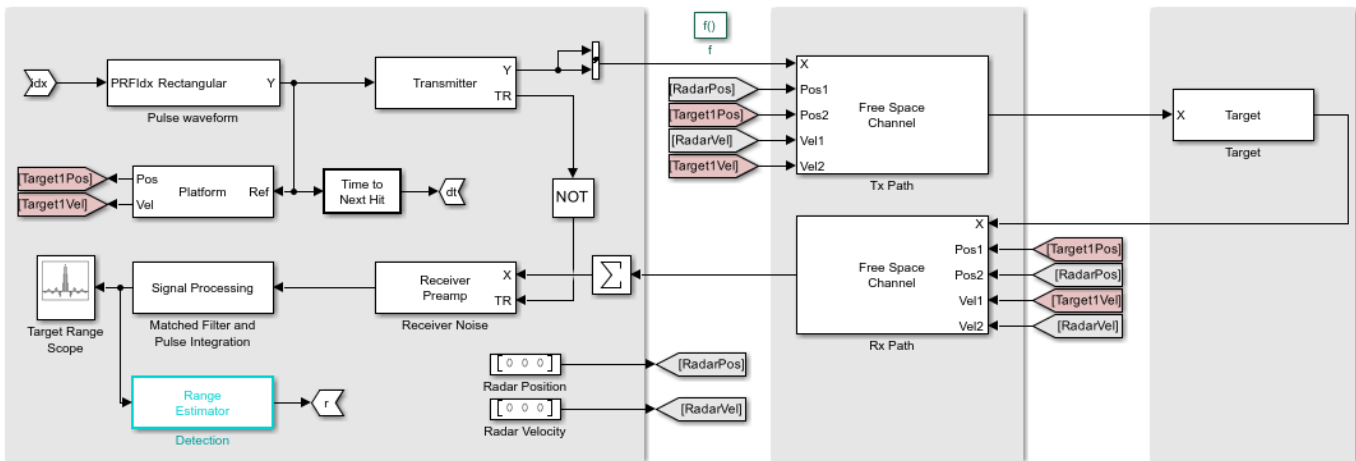

The system is very similar to what is used in the "Waveform Scheduling Based on Target Detection" example with the following notable difference:

1   The waveform block is no longer a source block. Instead, it takes an input, `idx`, to select which PRF to use. The available PRF values are specified in the PRF parameter of the waveform dialog.

2   The output of the waveform is also used to compute the time, `dt`, that the next pulse should be transmitted. Note that in this case, the time interval is proportional to the length of the transmitted signal.

3   At the end of the signal processing chain, the target range is estimated and returned in `r`. The controller will use this information to decide which PRF to choose for next transmission.

4   Once the model is compiled, notice that the signal passing through the system can vary in length because of a possible change of the waveform PRF. In addition, because the sample rate cannot be derived inside a Simulink function subsystem, the sample rate is specified in the block diagrams, such as the Tx and Rx paths, the receiver preamp, and other blocks.

```
helperslexPRFSelectionSim('updateModel');
```
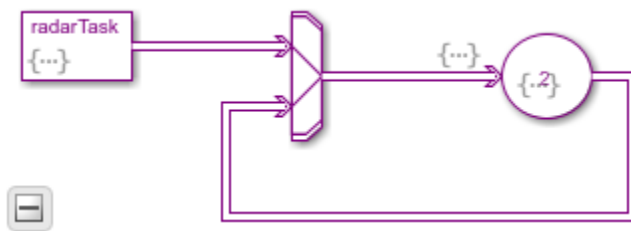
```
helperslexPRFSelectionSim('showRadar');
```
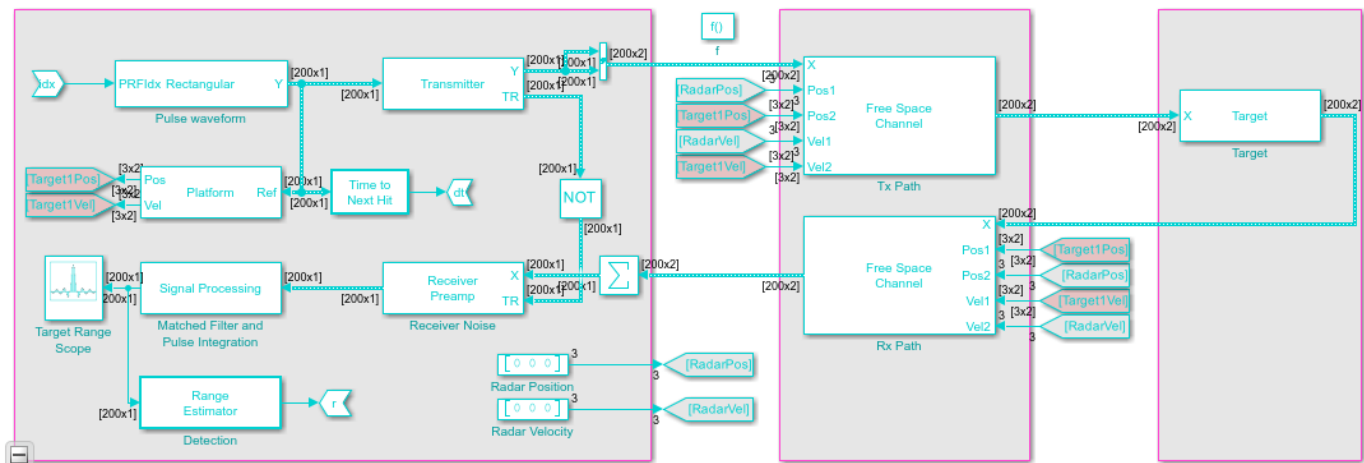

Exploring the Example

Several dialog parameters of the model are calculated by the helper function
helperslexPRFSelectionParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

The figure below shows the detected ranges of the targets. Target ranges are computed from the round-trip time delay of the reflected signals from the targets. At the simulation start, the radar detects two targets, one is slightly over 2 km away and the other one is at approximately 3.5 km away.

```
helperslexPRFSelectionSim('runModel',0.1);
helperslexPRFSelectionSim('showResult');
```
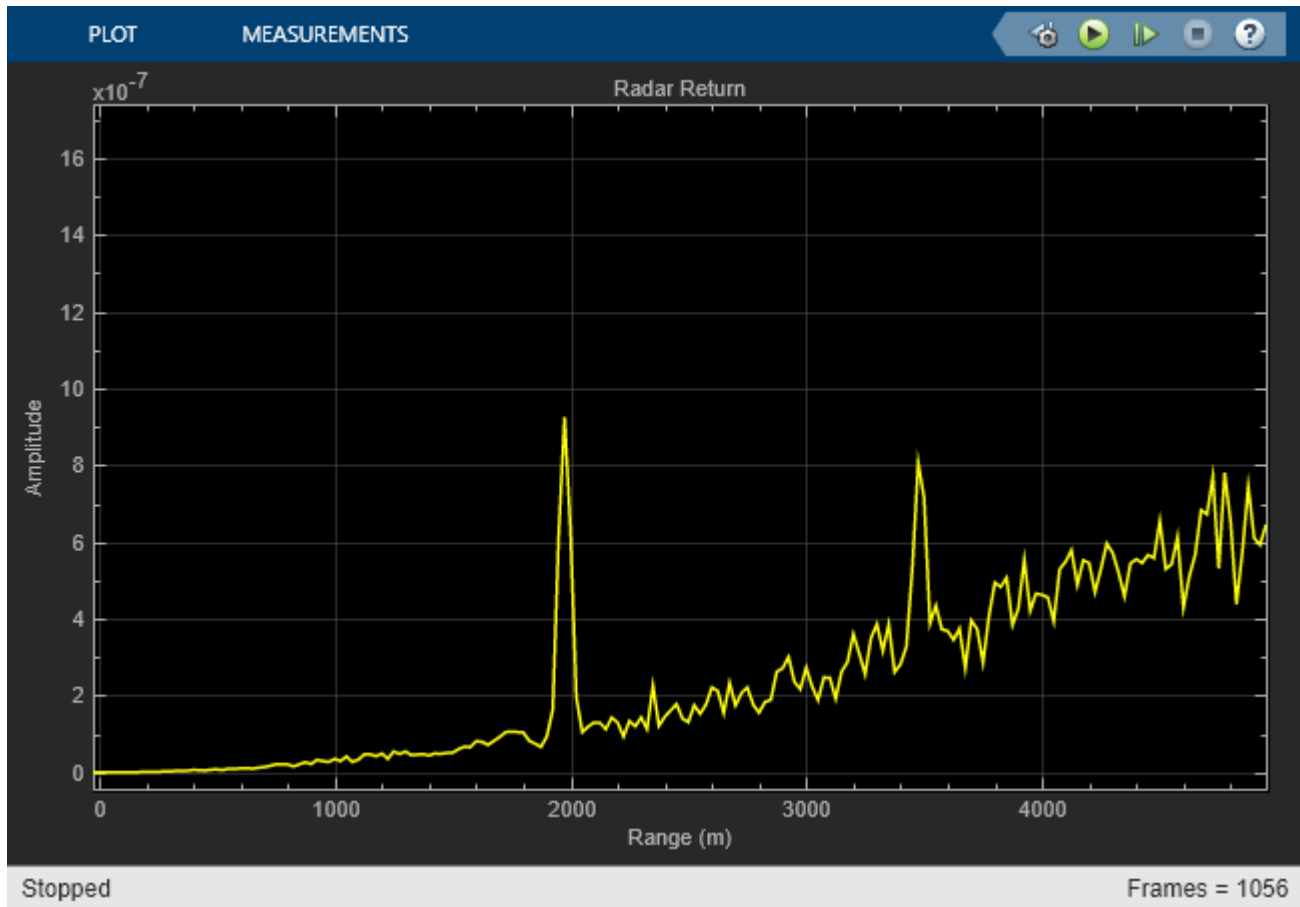
After some time, the first target moves into the 2 km zone and triggers a change of PRF. Then the received signal only covers the range up to 2 km. The display is zero padded to ensure that the plot limits do not change. Notice that the target at 3.5 km gets folded to the 1.5 km range due to range ambiguity.

```
helperslexPRFSelectionSim('runModel');
helperslexPRFSelectionSim('showResult');
```
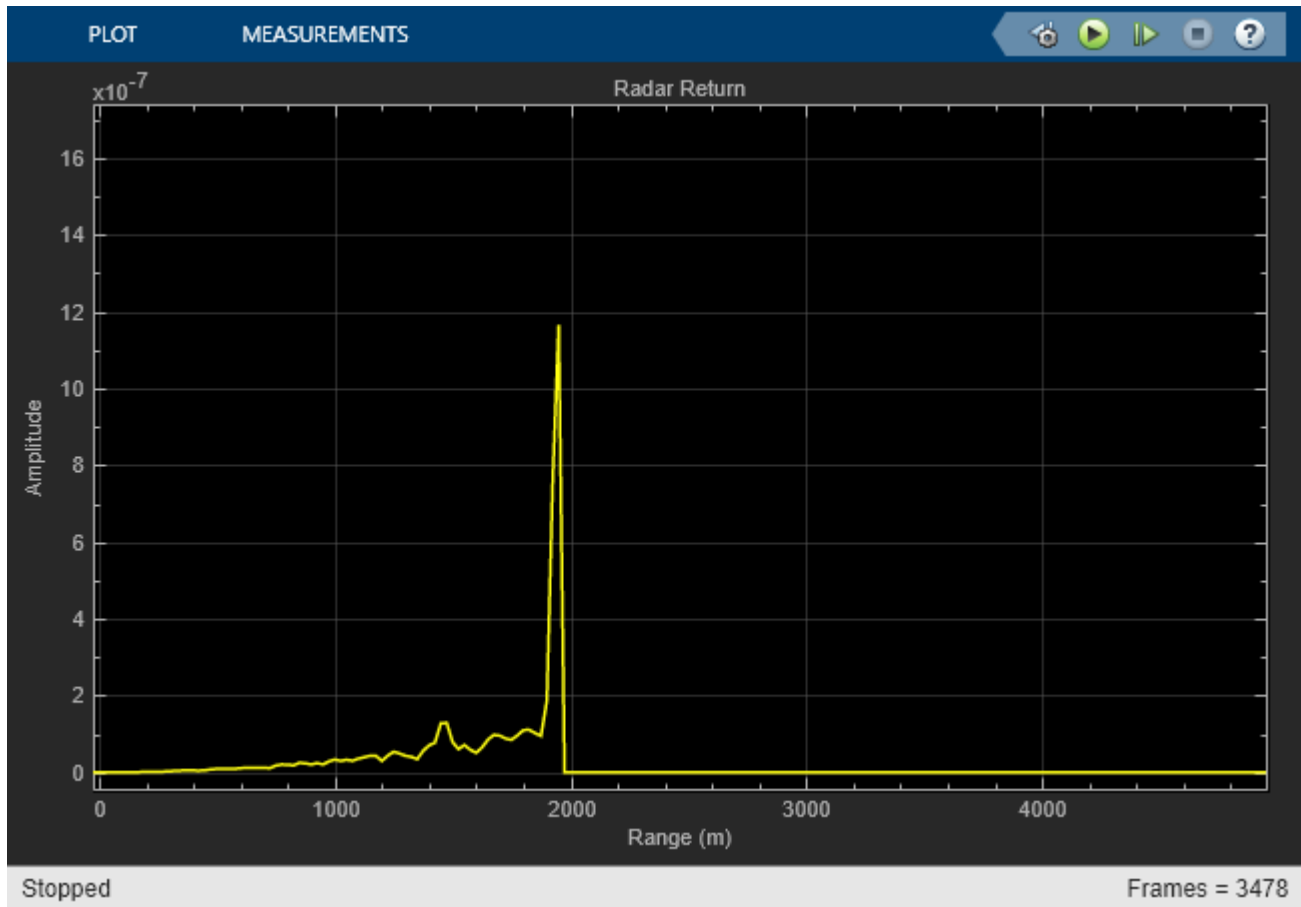
Summary

This example shows how to build a radar system in Simulink® that dynamically changes its PRF based on the target detection range. A staggered PRF system can be modeled similarly.

```
helperslexPRFSelectionSim('closeModel');
```

Label Radar Signals with Signal Labeler

This example shows how to label the main time and frequency features of pulse radar signals. This step facilitates the process of creating complete and accurate data sets to train artificial intelligence (AI) models. Signal Labeler eases this task in two ways. In manual mode, synchronized time and time-frequency views help you identify frequency features such as the waveform type, which can be `Rectangular`, `LinearFM`, or `SteppedFM`. In automated mode, you can register functions that measure the pulse repetition frequency (PRF), pulse width, duty cycle, and pulse bandwidth and only correct mistakes instead of labeling all signals from scratch. A built-in dashboard helps track the labeling progress and assess the quality of the labels.

Import Data into Signal Labeler

The radar signals in this example are stored in individual MAT files. Each file contains a data variable `x` and a corresponding sample rate variable `Fs`.

Open **Signal Labeler**. On the **Labeler** tab, click **Import** and select `From Folders` in the **Members** list. In the dialog box, select the directory with radar signal files. To specify the signal variables that you want to read, click **Specify** and enter `x`. Add time information: Choose the `Working in` option and select `Time` from the list. Select `Sample Rate Variable From File` in the list and enter `Fs`. Click **Import**. The imported files appear in the **Labeled Signal Set Browser**. Plot the first three signals by selecting the check boxes next to their names.
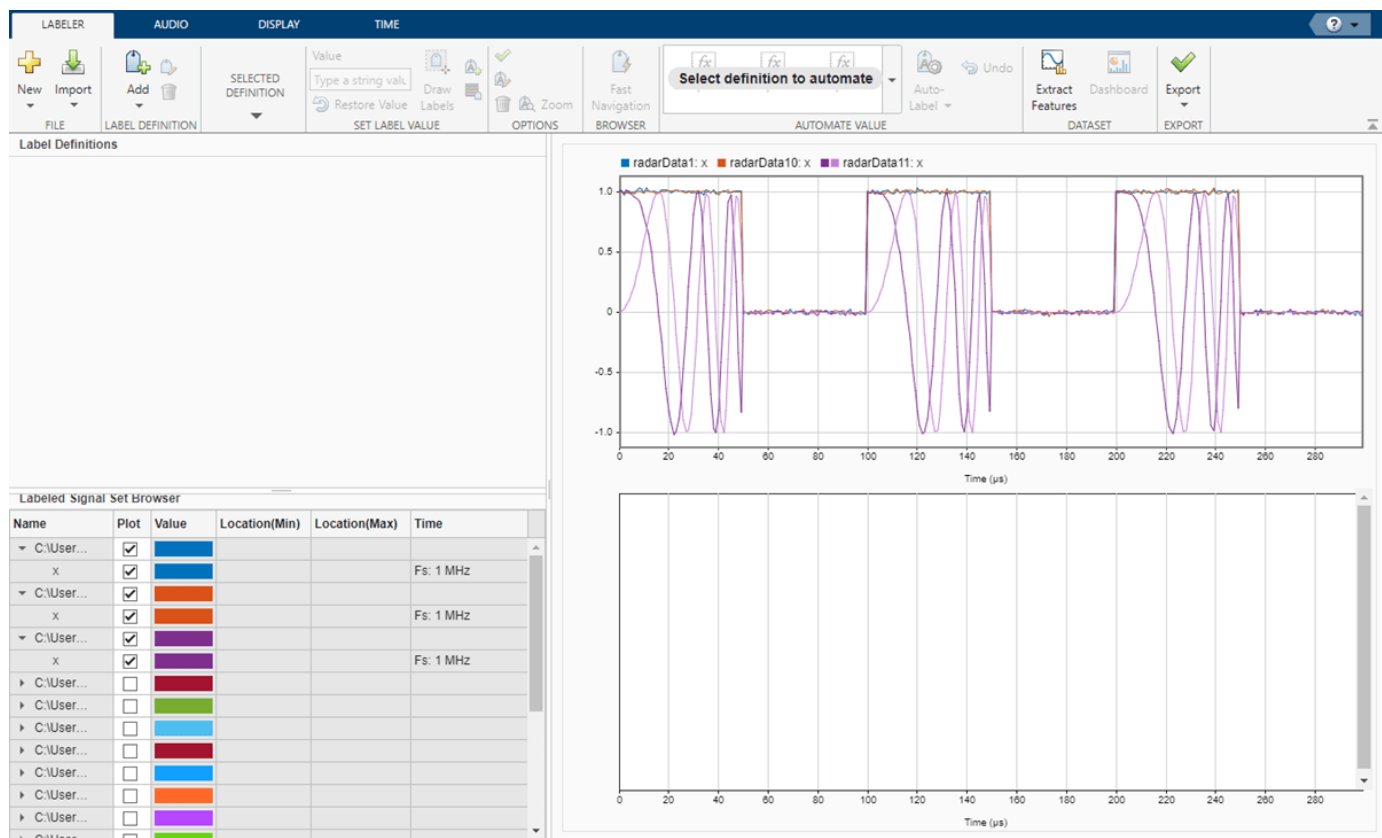
Define Labels

Create a label definition for the signal waveform type.

1 Click **Add** on the **Labeler** tab and select `Add Label Definition`.

2 In the dialog box, specify **Label Name** as `WaveformType`. Set **Label Type** to `Attribute` and **Data Type** to `string`.

3 Click **OK**.

Repeat these steps to create attribute label definitions for PRF, duty cycle, and bandwidth. Modify the label name for each and set the data type to `numeric`.

Create a region-of-interest (ROI) label for pulse width that corresponds to the regions showing initial and final crossings used to compute each pulse width. Specify **Label Name** as `PulseWidth`, **Label Type** as ROI, and **Data Type** as `numeric`. The label definitions appear in the **Label Definitions** browser.

Label Definitions
▦ WaveformType
▦ PRF
▦ DutyCycle
▦ Bandwidth
▨ PulseWidth
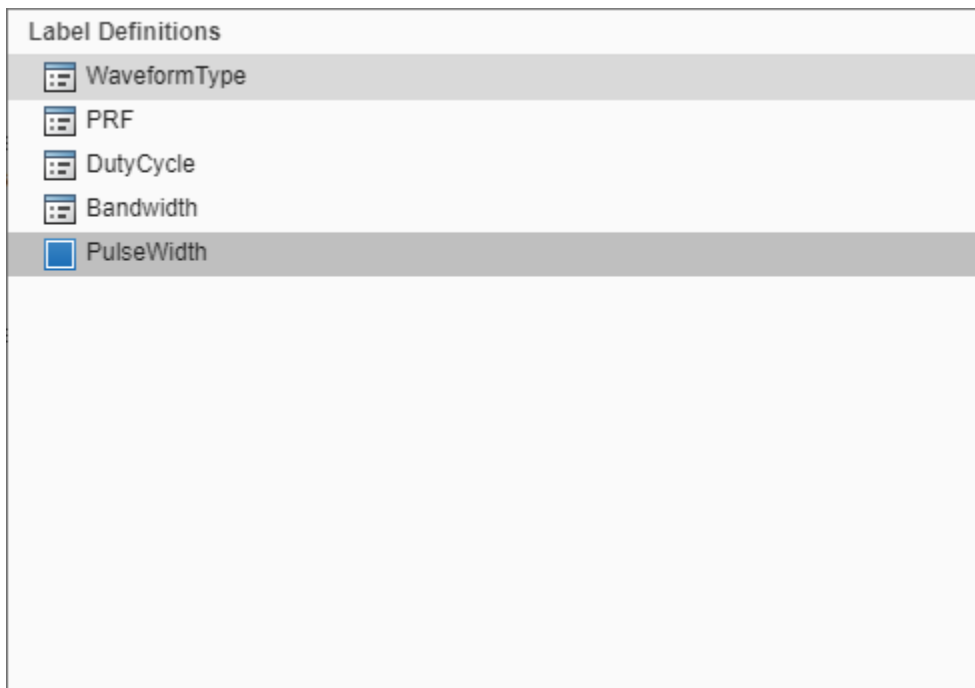
Create Custom Autolabeling Functions

Use four "Custom Labeling Functions" to label the PRF, bandwidth, duty cycle, and pulse width. Code for these functions is given in the Supporting Functions on page 1-239 section of the example. To create each function, on the **Labeler** tab, click **Automate Value** and select **Add Custom Function**. **Signal Labeler** shows a dialog box where you input the name, description, and label type of the function.

1 For the function that computes the PRF, enter `computePRF` on page 1-239 in the **Name** field and select `Attribute` as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.

2 For the function that computes the bandwidth, enter `computeBandWidth` on page 1-240 in the **Name** field and select `Attribute` as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.

3 For the function that computes the duty cycle, enter `computeDutyCycle` on page 1-240 in the **Name** field and select `Attribute` as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.

4 For the function that computes the pulse width, enter `computePulseWidth` on page 1-240 in the **Name** field and select `ROI` as the **Label Type**. You can leave the **Description** field empty or you can enter your own description.

If you already have written the functions, and the functions are in the current folder or in the MATLAB® path, **Signal Labeler** adds the functions to the gallery. If you have not written the functions, **Signal Labeler** opens blank templates in the Editor for you to type or paste the code. Save the files. Once you save the files, the functions appear in the gallery.
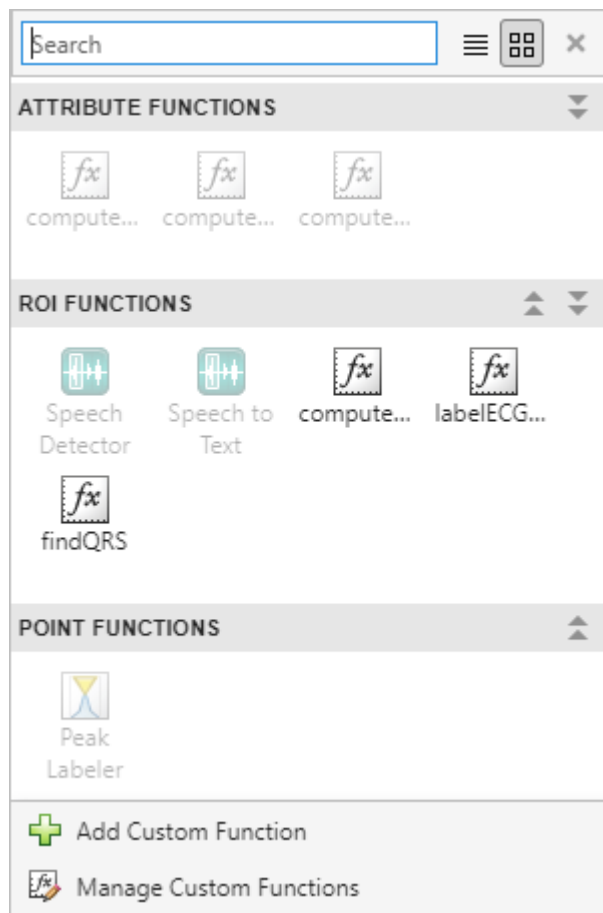
Label Waveform Type, PRF, Bandwidth, Duty Cycle, and Pulse Width

Set the waveform type of each signal:

1 In the **Labeled Signal Set Browser**, select the check box next to `radarData1`.

2 Click the **Display** tab and select **Spectrogram** in the **Views** section. The app displays a set of axes with the signal spectrogram and a **Spectrogram** tab with options to control the view.

3 Click the **Spectrogram** tab and set the overlap percentage to 99.

4 The spectrogram shows that the signal waveform is rectangular. In the label viewer attribute table, double-click the cell below **WaveFormType** and type `Rectangular`.

5 Repeat this manual labeling step for all the signals in the data set.
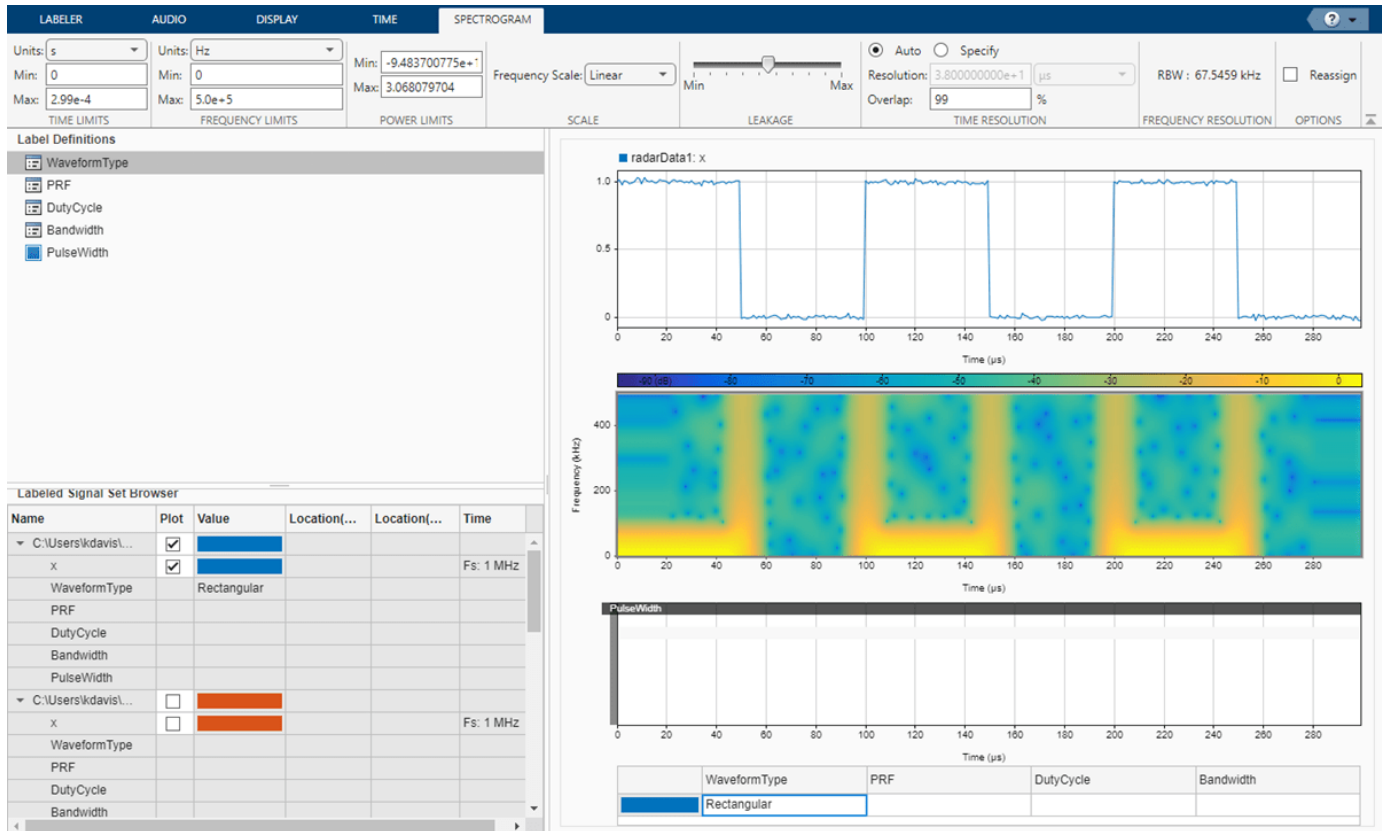
An example of a `LinearFM` waveform follows.
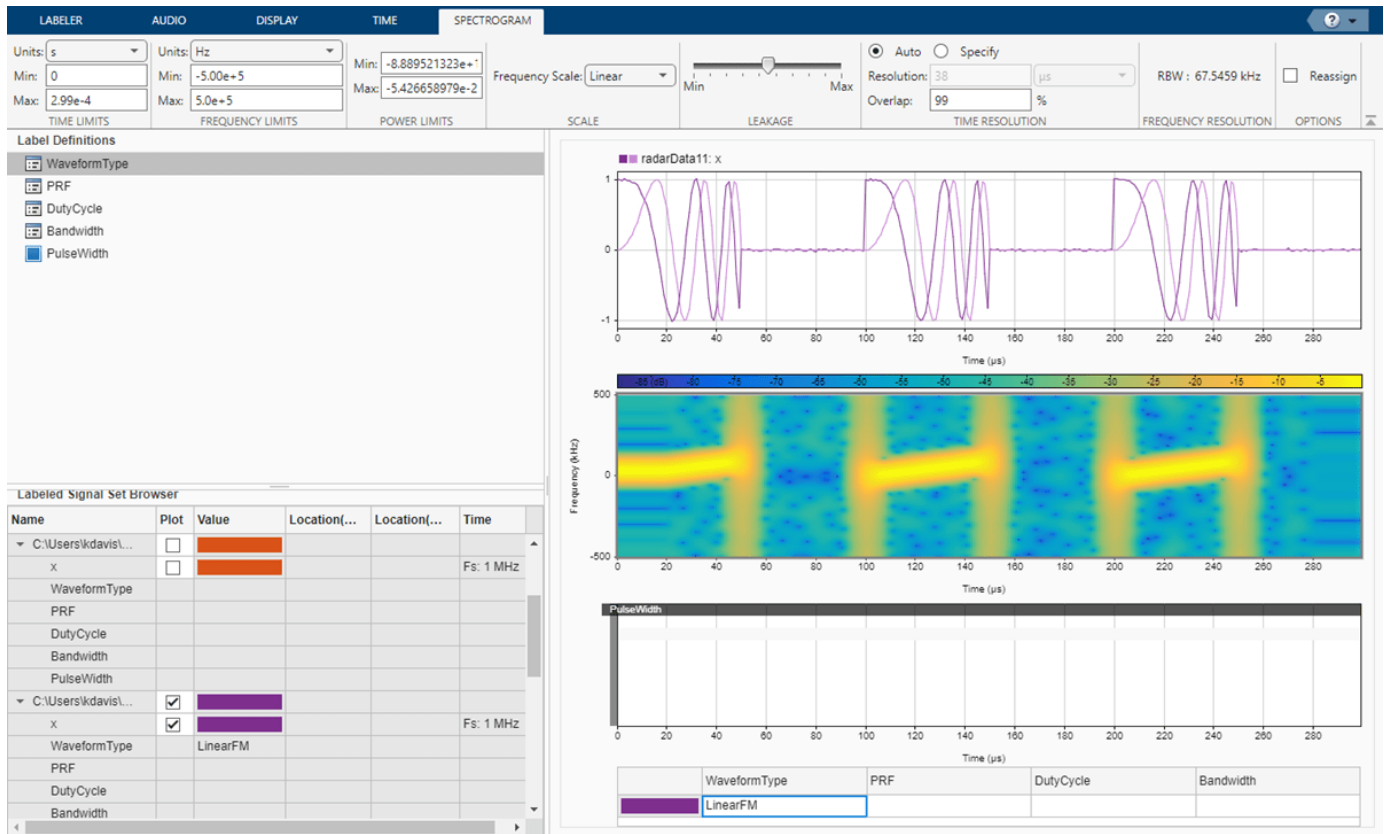
An example of a `SteppedFM` waveform follows.
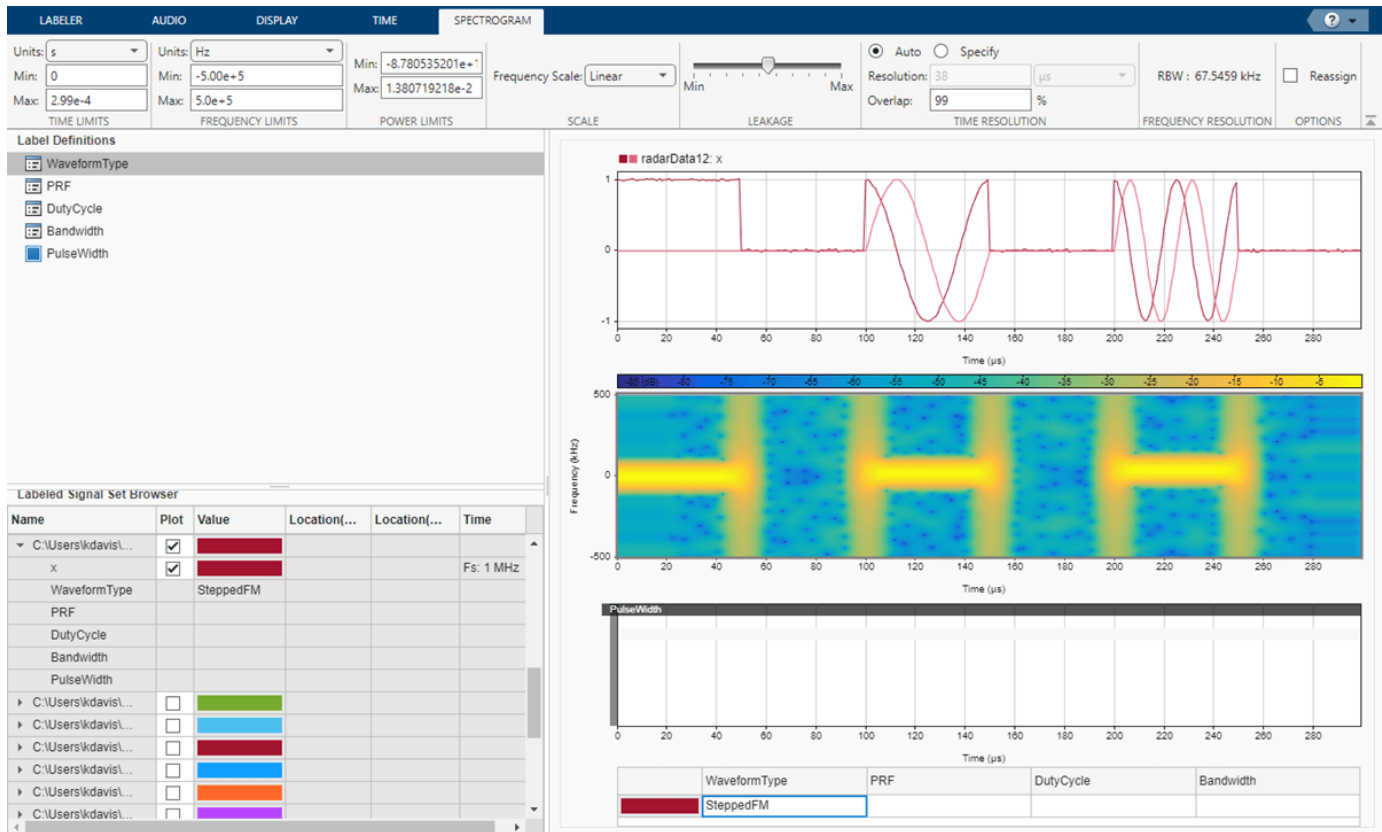
Compute and label the PRF of the input signals.

1 Select PRF in the **Label Definitions** browser.

2 In the **Automate Value** gallery, select `computePRF`.

3 Click **Auto-Label** and select `Auto-Label All Signals`. In the dialog box that appears, click **OK**.

Repeat the above steps for bandwidth, duty cycle, and pulse width by selecting the corresponding label definition and autolabeling function.

Signal Labeler computes and labels all signals, but displays labels only for the signals whose check boxes are selected. Plot the signals from the first five members.
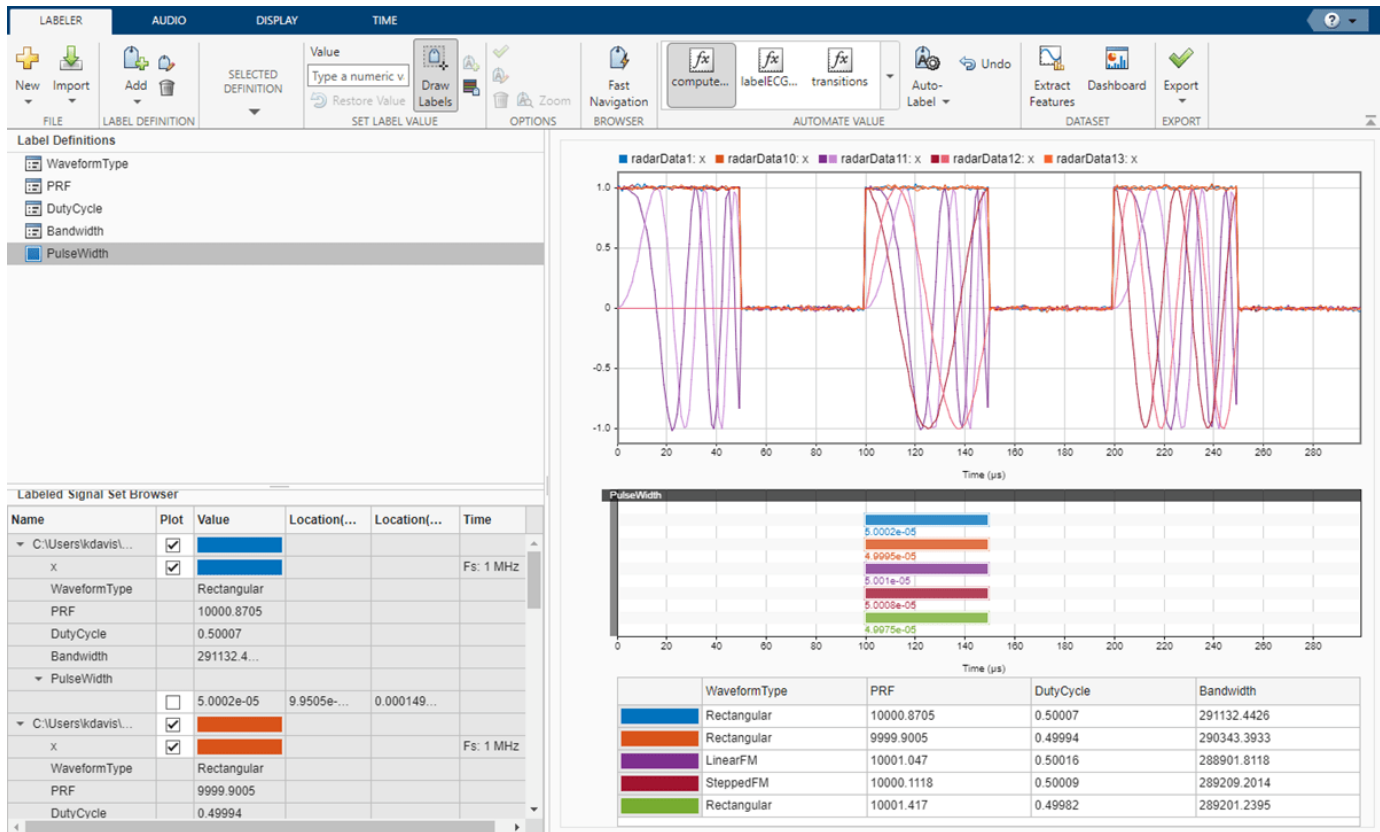
Validate Labeled Signals

View your labeling progress and verify that the computed label values are correct. Select **WaveFormType** in the **Label Definitions** browser and click **Dashboard** in the **Labeler** tab.

The plot on the left shows the labeling progress, which is 100% as all signals are labeled with the **WaveFormType** label. The plot on the right shows the number of signals with labels for each label value. You can use the **Label Distribution** pie chart to assess the accuracy of your labeling and confirm the results are as expected.
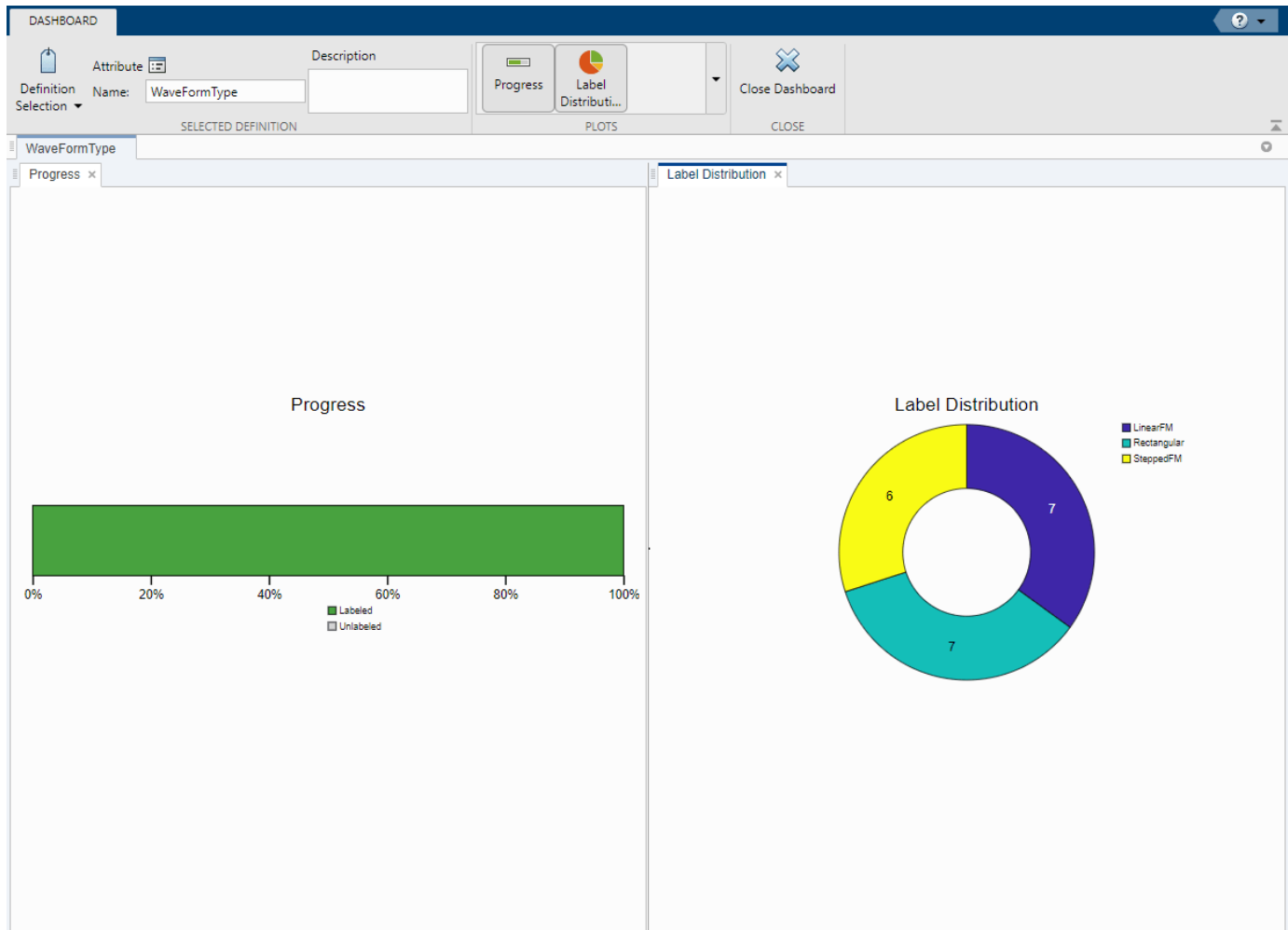
Next, validate that all pulse width label values are distributed around `5e-5`. To look at the time distribution of the pulse width label values, click **Definition Selection** on the **Dashboard** tab and select **PulseWidth**. Click on the time distribution plot and on the **Dashboard** tab, set **Bins** to 3, **X Min** to `4e-5` and **X Max** to `6e-5`. All signals have a pulse width distributed around `5e-5`.
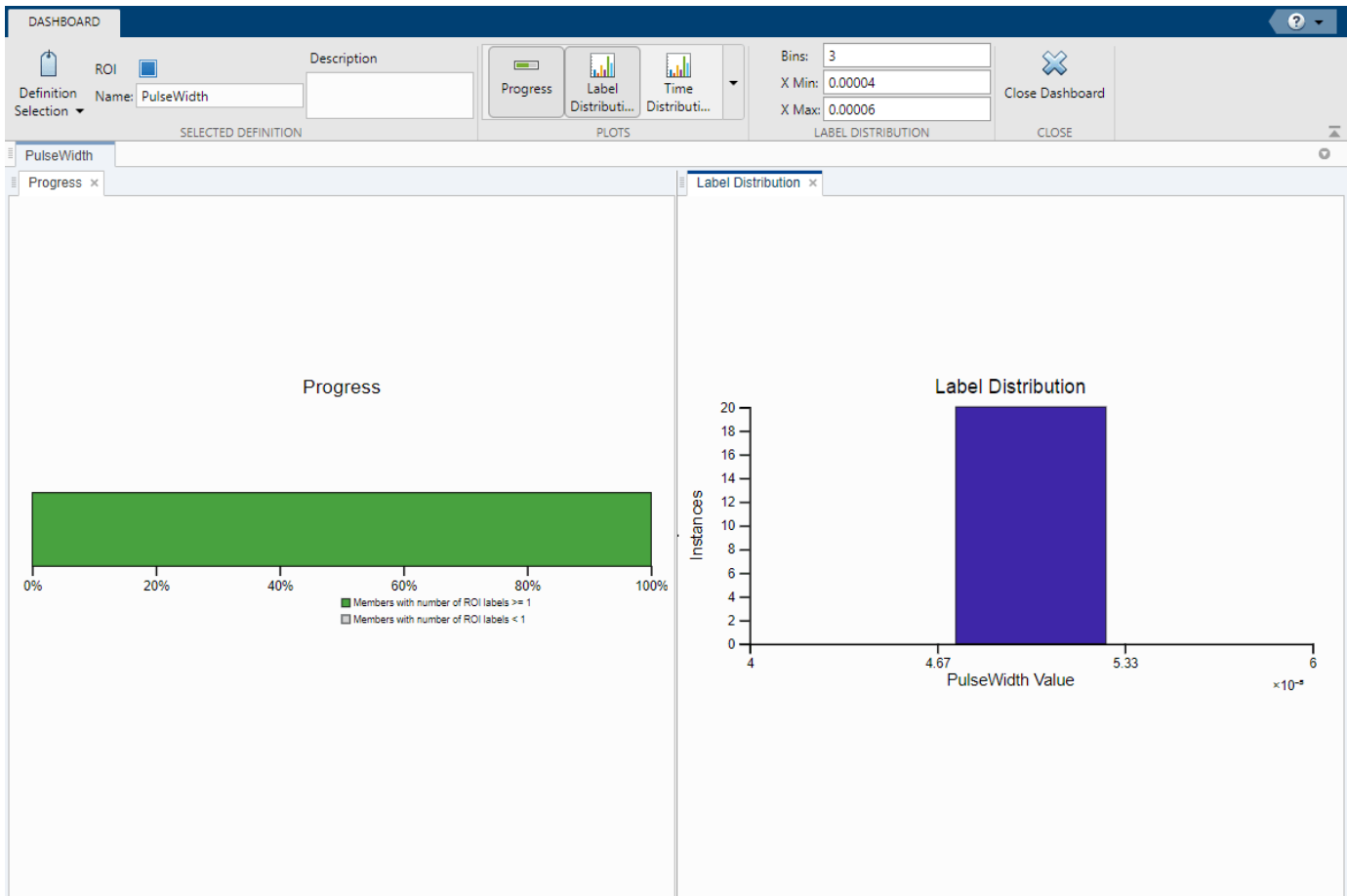
Close the dashboard.

Export Labeled Signals

Export the labeled signals to train AI models. On the **Labeler** tab, click **Export** and select `Labeled Signal Set To File`. In the dialog box that appears, give the name `radarDataLss.mat` to the labeled signal set and add an optional short description. Click **Export**.

Go back to the MATLAB® Command Window. Load the labeled signal set and create signal and label datastores from the labeled signal set. Create a combined datastore with the signal and label datastores. Use `read` or `readall` to get signal-label pairs that you can use to train AI models.

```
load radarDataLss.mat
[signalDS,labelDs] = ls.createDatastores("WaveFormType");
combineDs = combine(signalDS,labelDs);
```

Supporting Functions

computePRF Function: Calculate the pulse repetition frequency

The `computePRF` function computes and labels the PRF of the input signal. It uses the `pulseperiod` function.

```matlab
function [labelVal,labelLoc] = computePRF(x,t,parentLabelVal,parentLabelLoc,varargin)
% Function to calculate pulse repetition frequency of a radar pulse

if~isreal(x)
    x = abs(x);
end
pri = pulseperiod(x,t);
labelVal = 1/pri(1);
labelLoc = [];
end
```

computeBandWidth Function: Calculate the pulse bandwidth

The `computeBandWidth` function computes and labels the bandwidth of the input signal. It uses the `obw` function.

```matlab
function [labelVal,labelLoc] = computeBandWidth(x,t,parentLabelVal,~,varargin)
% Function to calculate occupied bandwidth of a radar pulse
if~isreal(x)
    x = abs(x);
end
fs = 1/mean(diff(t));
labelVal = obw(x,fs);
labelLoc = [];
end
```

computeDutyCycle Function: Calculate the pulse duty cycle

The `computeDutyCycle` function computes and labels the duty cycle of the input signal. It uses the `dutycycle` function.

```matlab
function [labelVal,labelLoc] = computeDutyCycle(x,t,parentLabelVal,parentLabelLoc,varargin)
% Function to calculate duty cycle of a radar pulse
if~isreal(x)
    x = abs(x);
end
labelVal = dutycycle(x,t);
labelLoc = [];
end
```

computePulseWidth Function: Calculate the pulse width

The `computePulseWidth` function computes and labels the pulse width of the input signal. It uses the `pulsewidth` function.

```matlab
function [labelVal,labelLoc] = computePulseWidth(x,t,parentLabelVal,parentLabelLoc,varargin)
% Function to calculate pulse width of a radar pulse
if~isreal(x)
    x = abs(x);
end
[pw,ic,fc] = pulsewidth(x,t);
labelVal = pw(1);
```

```
labelLoc = [ic(1) fc(1)];
end
```

See Also

Apps
Signal Labeler

Functions
labeledSignalSet

Related Examples

- "Label Signal Attributes, Regions of Interest, and Points"
- "Automate Signal Labeling with Custom Functions"

Pedestrian and Bicyclist Classification Using Deep Learning

This example shows how to classify pedestrians and bicyclists based on their micro-Doppler characteristics using a deep learning network and time-frequency analysis.

The movements of different parts of an object placed in front of a radar produce micro-Doppler signatures that can be used to identify the object. This example uses a convolutional neural network (CNN) to identify pedestrians and bicyclists based on their signatures.

This example trains the deep learning network using simulated data and then examines how the network performs at classifying two cases of overlapping signatures.

Synthetic Data Generation by Simulation

The data used to train the network is generated using `backscatterPedestrian` and `backscatterBicyclist` from Radar Toolbox™. These functions simulate the radar backscattering of signals reflected from pedestrians and bicyclists, respectively.

The helper function `helperBackScatterSignals` generates a specified number of pedestrian, bicyclist, and car radar returns. Because the purpose of the example is to classify pedestrians and bicyclists, this example considers car signatures as noise sources only. To get an idea of the classification problem to solve, examine one realization of a micro-Doppler signature from a pedestrian, a bicyclist, and a car. (For each realization, the return signals have dimensions N_{fast}-by-N_{slow}, where N_{fast} is the number of *fast-time* samples and N_{slow} is the number of *slow-time* samples. See "Radar Data Cube" for more information.)

```
numPed = 1; % Number of pedestrian realizations
numBic = 1; % Number of bicyclist realizations
numCar = 1; % Number of car realizations
[xPedRec,xBicRec,xCarRec,Tsamp] = helperBackScatterSignals(numPed,numBic,numCar);
```

The helper function `helperDopplerSignatures` computes the short-time Fourier transform (STFT) of a radar return to generate the micro-Doppler signature. To obtain the micro-Doppler signatures, use the helper functions to apply the STFT and a preprocessing method to each signal.

```
[SPed,T,F] = helperDopplerSignatures(xPedRec,Tsamp);
[SBic,~,~] = helperDopplerSignatures(xBicRec,Tsamp);
[SCar,~,~] = helperDopplerSignatures(xCarRec,Tsamp);
```

Plot the time-frequency maps for the pedestrian, bicyclist, and car realizations.

```
% Plot the first realization of objects
figure
subplot(1,3,1)
imagesc(T,F,SPed(:,:,1))
ylabel('Frequency (Hz)')
title('Pedestrian')
axis square xy

subplot(1,3,2)
imagesc(T,F,SBic(:,:,1))
xlabel('Time (s)')
title('Bicyclist')
axis square xy
```

```
subplot(1,3,3)
imagesc(T,F,SCar(:,:,1))
title('Car')
axis square xy
```
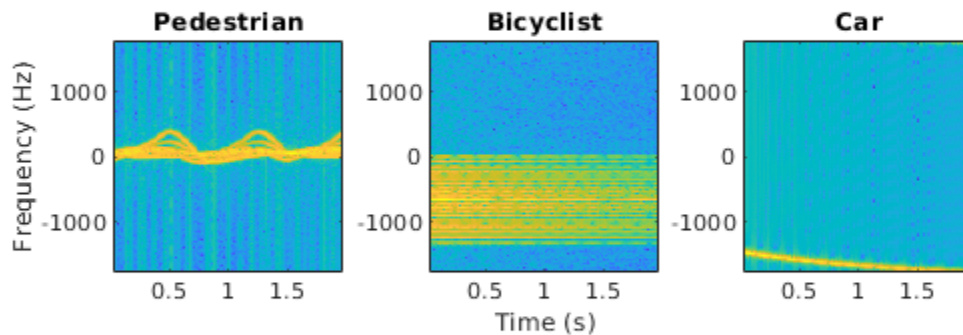

The normalized spectrograms (STFT absolute values) show that the three objects have quite distinct signatures. Specifically, the spectrograms of the pedestrian and the bicyclist have rich micro-Doppler signatures caused by the swing of arms and legs and the rotation of wheels, respectively. By contrast, in this example, the car is modeled as a point target with rigid body, so the spectrogram of the car shows that the short-term Doppler frequency shift varies little, indicating little micro-Doppler effect.

Combining Objects

Classifying a single realization as a pedestrian or bicyclist is relatively simple because the pedestrian and bicyclist micro-Doppler signatures are dissimilar. However, classifying multiple overlapping pedestrians or bicyclists, with the addition of Gaussian noise or car noise, is much more difficult.

If multiple objects exist in the detection region of the radar at the same time, the received radar signal is a summation of the detection signals from all the objects. As an example, generate the received radar signal for a pedestrian and bicyclist with Gaussian background noise.

```
% Configure Gaussian noise level at the receiver
rx = phased.ReceiverPreamp('Gain',25,'NoiseFigure',10);

xRadarRec = complex(zeros(size(xPedRec)));
for ii = 1:size(xPedRec,3)
```

```
    xRadarRec(:,:,ii) = rx(xPedRec(:,:,ii) + xBicRec(:,:,ii));
end
```

Then obtain micro-Doppler signatures of the received signal by using the STFT.

```
[S,~,~] = helperDopplerSignatures(xRadarRec,Tsamp);
```

```
figure
imagesc(T,F,S(:,:,1)) % Plot the first realization
axis xy
xlabel('Time (s)')
ylabel('Frequency (Hz)')
title('Spectrogram of a Pedestrian and a Bicyclist')
```
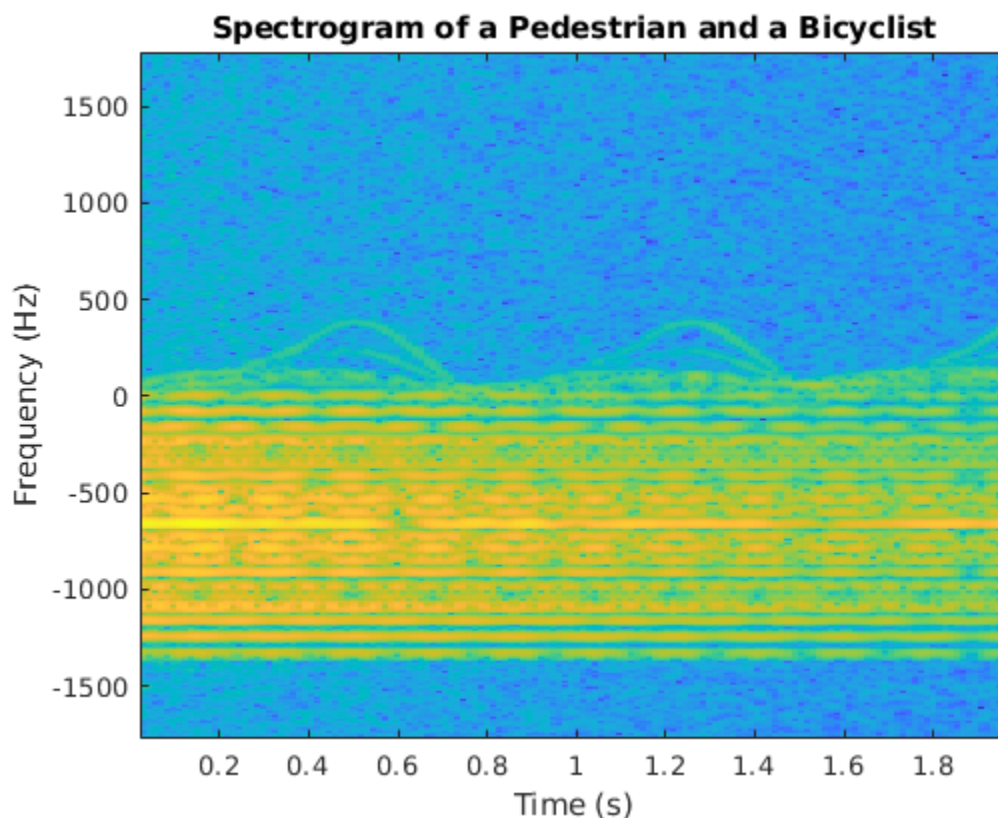

Because the pedestrian and bicyclist signatures overlap in time and frequency, differentiating between the two objects is difficult.

Generate Training Data

In this example, you train a CNN by using data consisting of simulated realizations of objects with varying properties—for example, bicyclists pedaling at different speeds and pedestrians with different heights walking at different speeds. Assuming the radar is fixed at the origin, in one realization, one object or multiple objects are uniformly distributed in a rectangular area of [5, 45] and [–10, 10] meters along the X and Y axes, respectively.
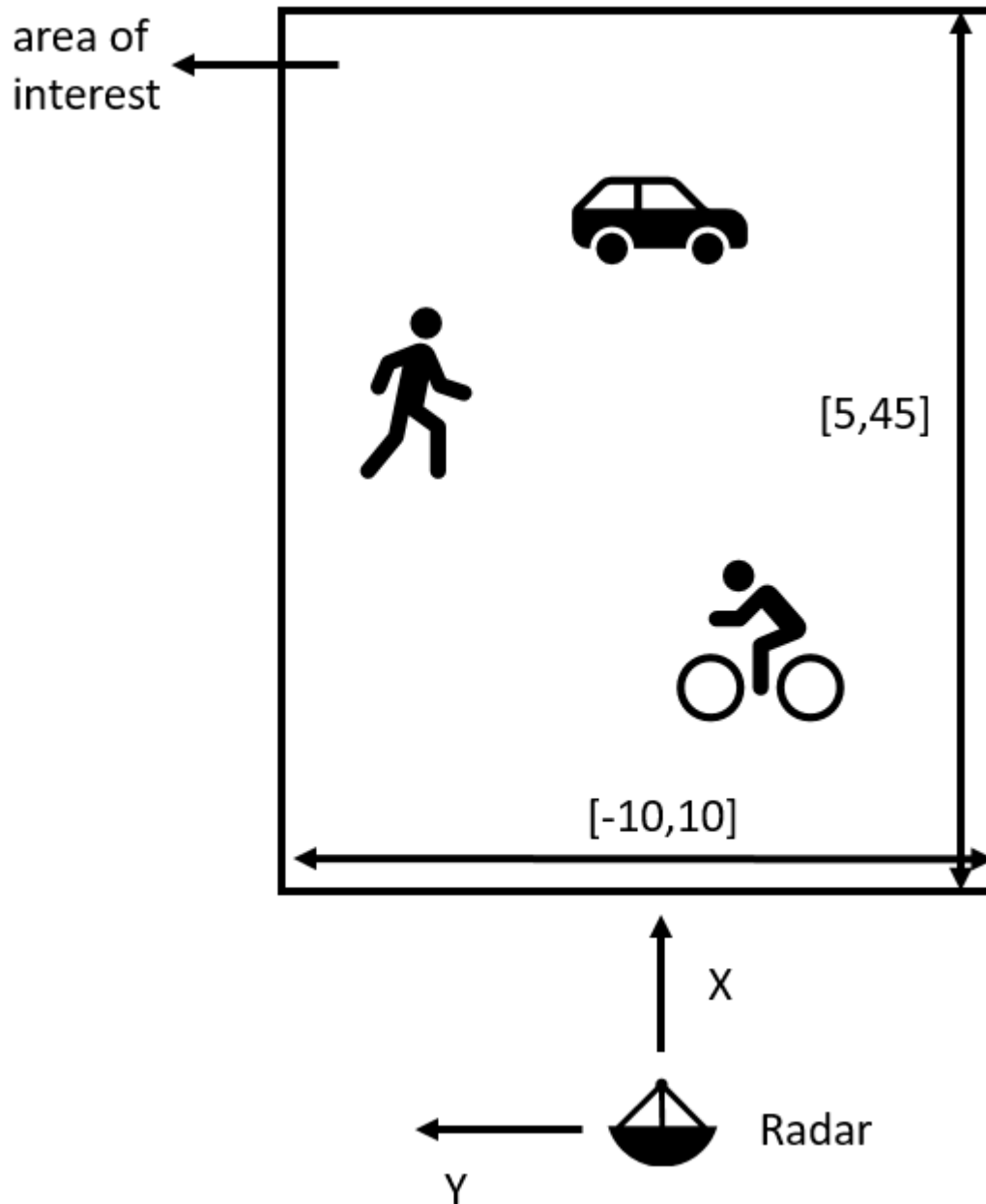
The other properties of the three objects that are randomly tuned are as follows:

1) Pedestrians

- Height — Uniformly distributed in the interval of [1.5, 2] meters
- Heading — Uniformly distributed in the interval of [–180, 180] degrees
- Speed — Uniformly distributed in the interval of [0, 1.4h] meters/second, where h is the height value

2) Bicyclists

- Heading — Uniformly distributed in the interval of [–180, 180] degrees
- Speed — Uniformly distributed in the interval of [1, 10] meters/second
- Gear transmission ratio — Uniformly distributed in the interval of [0.5, 6]
- Pedaling or coasting — 50% probability of pedaling (coasting means that the cyclist is moving without pedaling)

3) Cars

- Velocity — Uniformly distributed in the interval of [0, 10] meters/second along the X and Y directions

The input to the convolutional network is micro-Doppler signatures consisting of spectrograms expressed in decibels and normalized to [0, 1], as shown in this figure:
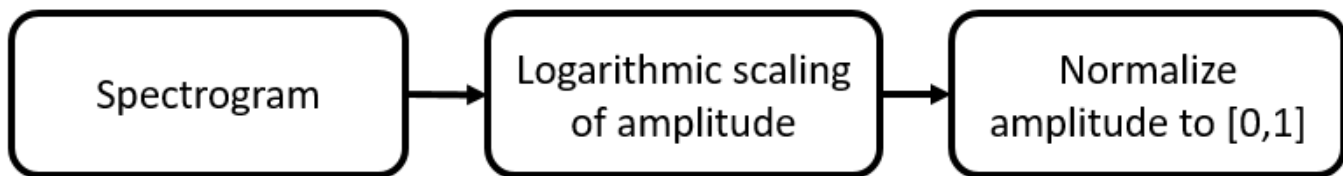
Radar returns originate from different objects and different parts of objects. Depending on the configuration, some returns are much stronger than others. Stronger returns tend to obscure weaker ones. Logarithmic scaling augments the features by making return strengths comparable. Amplitude normalization helps the CNN converge faster.

The data set contains realizations of the following scenes:

- One pedestrian present in the scene
- One bicyclist present in the scene
- One pedestrian and one bicyclist present in the scene
- Two pedestrians present in the scene
- Two bicyclists present in the scene

Download Data

The data for this example consists of 20,000 pedestrian, 20,000 bicyclist, and 12,500 car signals generated by using the helper functions `helperBackScatterSignals` and `helperDopplerSignatures`. The signals are divided into two data sets: one without car noise samples and one with car noise samples.

For the first data set (without car noise), the pedestrian and bicyclist signals were combined, Gaussian noise was added, and micro-Doppler signatures were computed to generate 5000 signatures for each of the five scenes to be classified.

In each category, 80% of the signatures (that is, 4000 signatures) are reserved for the training data set while 20% of the signatures (that is, 1000 signatures) are reserved for the test data set.

To generate the second data set (with car noise), the procedure for the first data set was followed, except that car noise was added to 50% of the signatures. The proportion of signatures with and without car noise is the same in the training and test data sets.

Download and unzip the data in your temporary directory, whose location is specified by MATLAB®'s `tempdir` command. The data has a size of 21 GB and the download process may take some time. If you have the data in a folder different from `tempdir`, change the directory name in the subsequent instructions.

```matlab
% Download the data
dataURL = 'https://ssd.mathworks.com/supportfiles/SPT/data/PedBicCarData.zip';
saveFolder = fullfile(tempdir,'PedBicCarData');
zipFile = fullfile(tempdir,'PedBicCarData.zip');
if ~exist(zipFile,'file')
    websave(zipFile,dataURL);
elseif ~exist(saveFolder,'dir')
    % Unzip the data
    unzip(zipFile,tempdir)
end
```

The data files are as follows:

- `trainDataNoCar.mat` contains the training data set `trainDataNoCar` and its label set `trainLabelNoCar`.
- `testDataNoCar.mat` contains the test data set `testDataNoCar` and its label set `testLabelNoCar`.
- `trainDataCarNoise.mat` contains the training data set `trainDataCarNoise` and its label set `trainLabelCarNoise`.
- `testDataCarNoise.mat` contains the test data set `testDataCarNoise` and its label set `testLabelCarNoise`.
- `TF.mat` contains the time and frequency information for the micro-Doppler signatures.

Network Architecture

Create a CNN with five convolution layers and one fully connected layer. The first four convolution layers are followed by a batch normalization layer, a rectified linear unit (ReLU) activation layer, and a max pooling layer. In the last convolution layer, the max pooling layer is replaced by an average pooling layer. The output layer is a classification layer after softmax activation. For network design guidance, see "Deep Learning Tips and Tricks" (Deep Learning Toolbox).

```matlab
layers = [
    imageInputLayer([size(S,1),size(S,2),1],'Normalization','none')

    convolution2dLayer(10,16,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(10,'Stride',2)

    convolution2dLayer(5,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(10,'Stride',2)

    convolution2dLayer(5,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(10,'Stride',2)

    convolution2dLayer(5,32,'Padding','same')
```

```
    batchNormalizationLayer
    reluLayer
    maxPooling2dLayer(5,'Stride',2)

    convolution2dLayer(5,32,'Padding','same')
    batchNormalizationLayer
    reluLayer
    averagePooling2dLayer(2,'Stride',2)

    fullyConnectedLayer(5)
    softmaxLayer

    classificationLayer]

layers =
  24x1 Layer array with layers:

     1   ''   Image Input            400x144x1 images
     2   ''   Convolution            16 10x10 convolutions with stride [1  1] and padding 'same
     3   ''   Batch Normalization    Batch normalization
     4   ''   ReLU                   ReLU
     5   ''   Max Pooling            10x10 max pooling with stride [2  2] and padding [0  0  0
     6   ''   Convolution            32 5x5 convolutions with stride [1  1] and padding 'same'
     7   ''   Batch Normalization    Batch normalization
     8   ''   ReLU                   ReLU
     9   ''   Max Pooling            10x10 max pooling with stride [2  2] and padding [0  0  0
    10   ''   Convolution            32 5x5 convolutions with stride [1  1] and padding 'same'
    11   ''   Batch Normalization    Batch normalization
    12   ''   ReLU                   ReLU
    13   ''   Max Pooling            10x10 max pooling with stride [2  2] and padding [0  0  0
    14   ''   Convolution            32 5x5 convolutions with stride [1  1] and padding 'same'
    15   ''   Batch Normalization    Batch normalization
    16   ''   ReLU                   ReLU
    17   ''   Max Pooling            5x5 max pooling with stride [2  2] and padding [0  0  0  0]
    18   ''   Convolution            32 5x5 convolutions with stride [1  1] and padding 'same'
    19   ''   Batch Normalization    Batch normalization
    20   ''   ReLU                   ReLU
    21   ''   Average Pooling        2x2 average pooling with stride [2  2] and padding [0  0  0
    22   ''   Fully Connected        5 fully connected layer
    23   ''   Softmax                softmax
    24   ''   Classification Output  crossentropyex
```

Specify the optimization solver and the hyperparameters to train the CNN using `trainingOptions`. This example uses the ADAM optimizer and a mini-batch size of 128. Train the network using either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox™. To see which GPUs are supported, see "GPU Computing Requirements" (Parallel Computing Toolbox). For information on other parameters, see `trainingOptions` (Deep Learning Toolbox). This example uses a GPU for training.

```
options = trainingOptions('adam', ...
    'ExecutionEnvironment','gpu',...
    'MiniBatchSize',128, ...
    'MaxEpochs',30, ...
    'InitialLearnRate',1e-2, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.1, ...
    'LearnRateDropPeriod',10, ...
    'Shuffle','every-epoch', ...
```

```
    'Verbose',false, ...
    'Plots','training-progress');
```

Classify Signatures Without Car Noise

Load the data set without car noise and use the helper function `helperPlotTrainData` to plot one example of each of the five categories in the training data set,

```
load(fullfile(tempdir,'PedBicCarData','trainDataNoCar.mat')) % load training data set
load(fullfile(tempdir,'PedBicCarData','testDataNoCar.mat')) % load test data set
load(fullfile(tempdir,'PedBicCarData','TF.mat')) % load time and frequency information
```

```
helperPlotTrainData(trainDataNoCar,trainLabelNoCar,T,F)
```
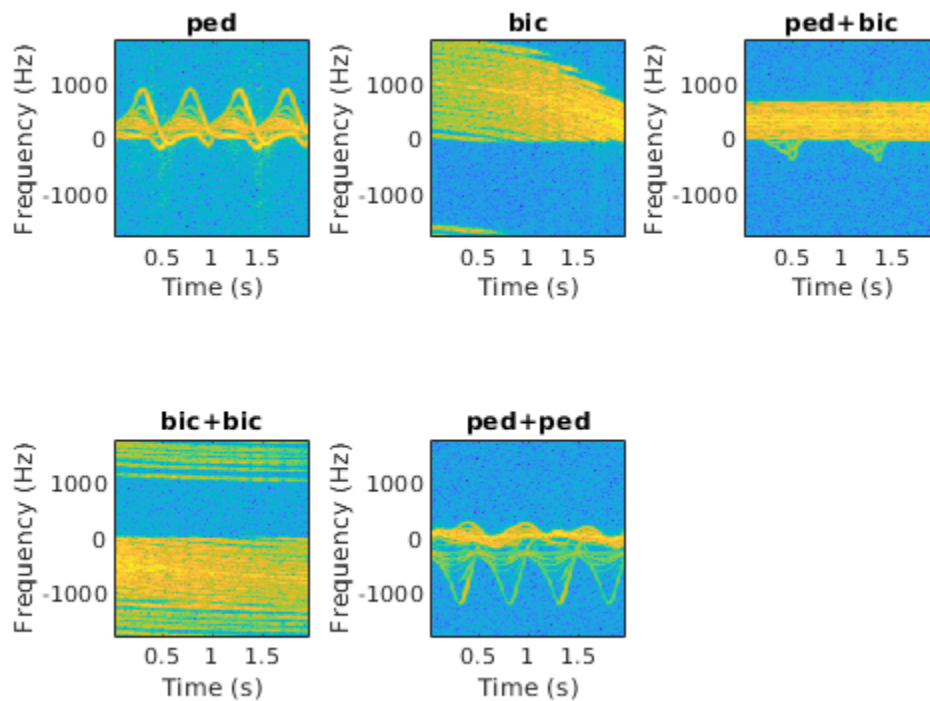

Train the CNN that you created. You can view the accuracy and loss during the training process. In 30 epochs, the training process achieves almost 95% accuracy.

```
trainedNetNoCar = trainNetwork(trainDataNoCar,trainLabelNoCar,layers,options);
```
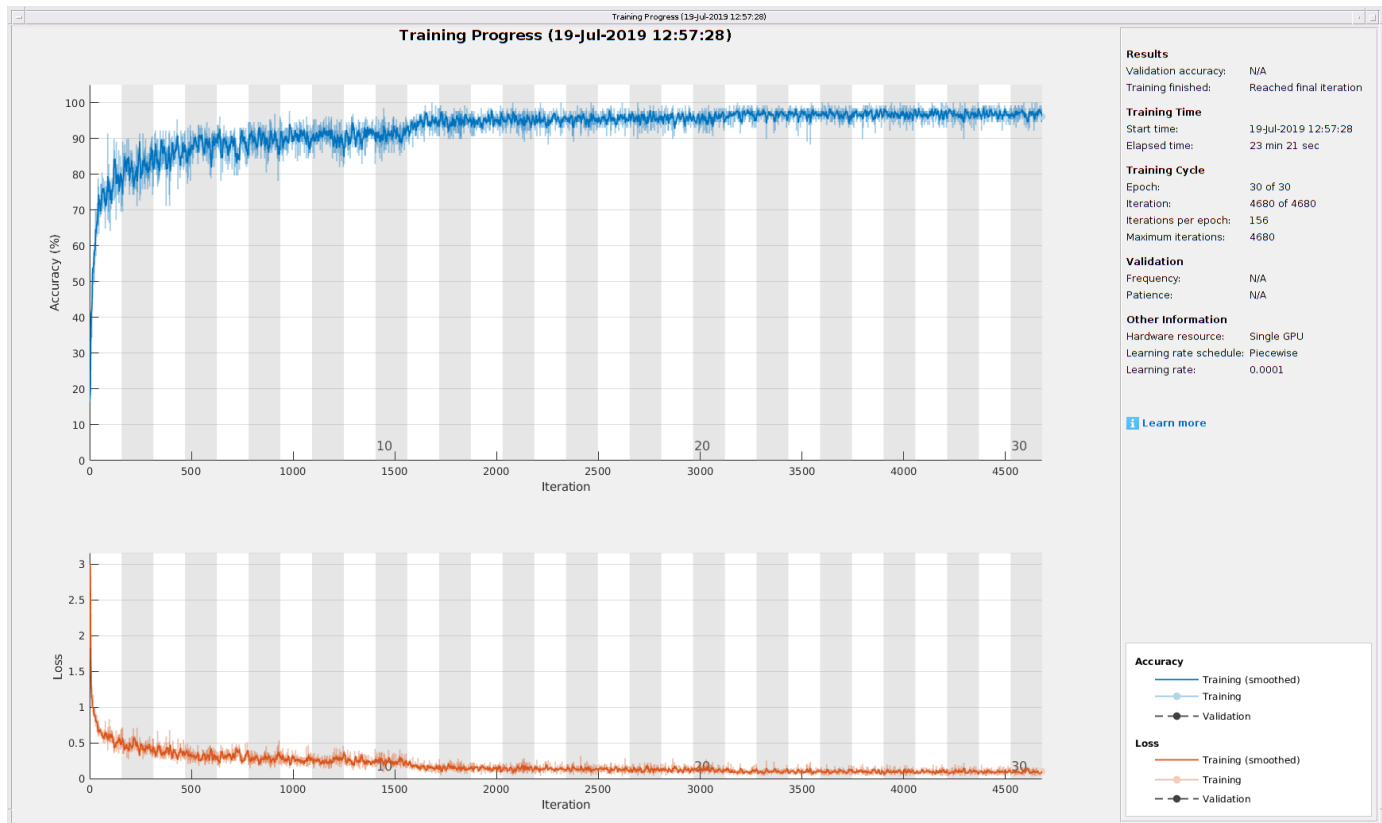
Use the trained network and the `classify` function to obtain the predicted labels for the test data set `testDataNoCar`. The variable `predTestLabel` contains the network predictions. The network achieves about 95% accuracy for the test data set without the car noise.

```
predTestLabel = classify(trainedNetNoCar,testDataNoCar);
testAccuracy = mean(predTestLabel == testLabelNoCar)
```

testAccuracy = 0.9530

Use a confusion matrix to view detailed information about prediction performance for each category. The confusion matrix for the trained network shows that, in each category, the network predicts the labels of the signals in the test data set with a high degree of accuracy.

```
figure
confusionchart(testLabelNoCar,predTestLabel);
```

Classify Signatures with Car Noise

To analyze the effects of car noise, classify data containing car noise with the `trainedNetNoCar` network, which was trained without car noise.

Load the car-noise-corrupted test data set `testDataCarNoise.mat`.

```
load(fullfile(tempdir,'PedBicCarData','testDataCarNoise.mat'))
```

Input the car-noise-corrupted test data set to the network. The prediction accuracy for the test data set with the car noise drops significantly, to around 70%, because the network never saw training samples containing car noise.

```
predTestLabel = classify(trainedNetNoCar,testDataCarNoise);
testAccuracy = mean(predTestLabel == testLabelCarNoise)
```

```
testAccuracy = 0.7176
```

The confusion matrix shows that most prediction errors occur when the network takes in scenes from the "pedestrian," "pedestrian+pedestrian," or "pedestrian+bicyclist" classes and classifies them as "bicyclist."

```
confusionchart(testLabelCarNoise,predTestLabel);
```

Car noise significantly impedes the performance of the classifier. To solve this problem, train the CNN using data that contains car noise.

Retrain CNN by Adding Car Noise to Training Data Set

Load the car-noise-corrupted training data set `trainDataCarNoise.mat`.

```
load(fullfile(tempdir,'PedBicCarData','trainDataCarNoise.mat'))
```

Retrain the network by using the car-noise-corrupted training data set. In 30 epochs, the training process achieves almost 90% accuracy.

```
trainedNetCarNoise = trainNetwork(trainDataCarNoise,trainLabelCarNoise,layers,options);
```
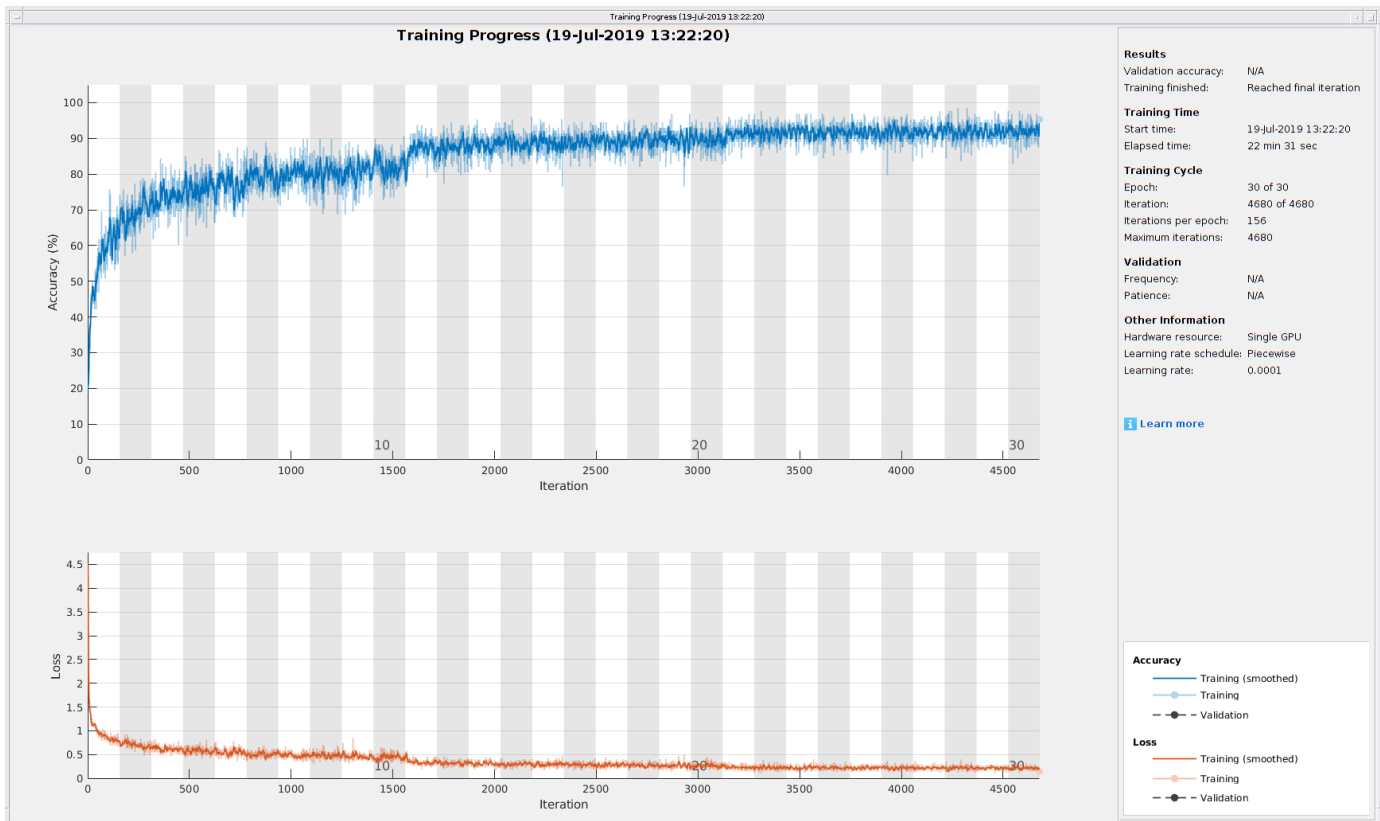
Input the car-noise-corrupted test data set to the network `trainedNetCarNoise`. The prediction accuracy is about 87%, which is approximately 15% higher than the performance of the network trained without car noise samples.

```
predTestLabel = classify(trainedNetCarNoise,testDataCarNoise);
testAccuracy = mean(predTestLabel == testLabelCarNoise)
```

```
testAccuracy = 0.8728
```

The confusion matrix shows that the network `trainedNetCarNoise` performs much better at predicting scenes with one pedestrian and scenes with two pedestrians.

```
confusionchart(testLabelCarNoise,predTestLabel);
```

1-253

	ped	bic	ped+bic	ped+ped	bic+bic
ped	908	1	1	90	
bic	15	911	28	3	43
ped+bic	12	78	835	39	36
ped+ped	165	1	4	830	
bic+bic	4	82	33	1	880

True Class / Predicted Class

Case Study

To better understand the performance of the network, examine its performance in classifying overlapping signatures. This section is just for illustration. Due to the non-deterministic behavior of GPU training, you may not get the same classification results in this section when you rerun this example.

For example, signature #4 of the car-noise-corrupted test data, which does not have car noise, has two bicyclists with overlapping micro-Doppler signatures. The network correctly predicts that the scene has two bicyclists.

```
k = 4;
imagesc(T,F,testDataCarNoise(:,:,:,k))
axis xy
xlabel('Time (s)')
ylabel('Frequency (Hz)')
title('Ground Truth: '+string(testLabelCarNoise(k))+', Prediction: '+string(predTestLabel(k)))
```

Ground Truth: bic+bic, Prediction: bic+bic
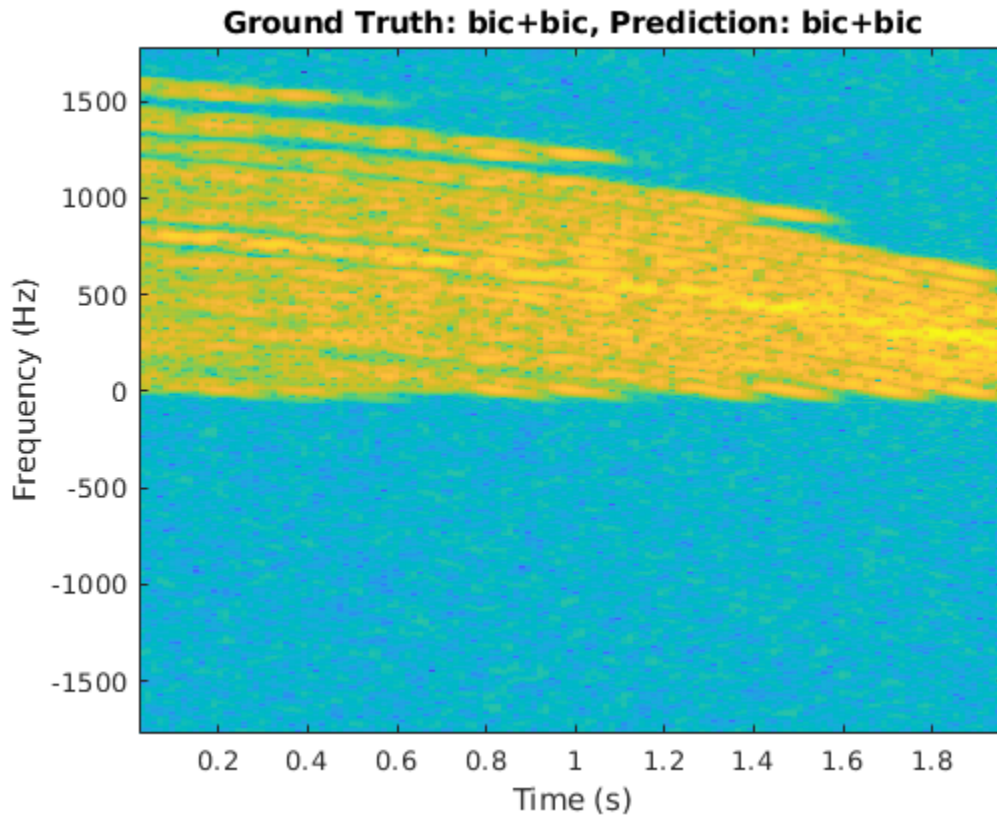
From the plot, the signature appears to be from only one bicyclist. Load the data `CaseStudyData.mat` of the two objects in the scene. The data contains return signals summed along the fast time. Apply the STFT to each signal.

```
load CaseStudyData.mat
M = 200; % FFT window length
beta = 6; % window parameter
w = kaiser(M,beta); % kaiser window
R = floor(1.7*(M-1)/(beta+1)); % ROUGH estimate
noverlap = M-R; % overlap length

[Sc,F,T] = stft(x,1/Tsamp,'Window',w,'FFTLength',M*2,'OverlapLength',noverlap);

for ii = 1:2
    subplot(1,2,ii)
    imagesc(T,F,10*log10(abs(Sc(:,:,ii))))
    xlabel('Time (s)')
    ylabel('Frequency (Hz)')
    title('Bicyclist')
    axis square xy
    title(['Bicyclist ' num2str(ii)])
    c = colorbar;
    c.Label.String = 'dB';
end
```
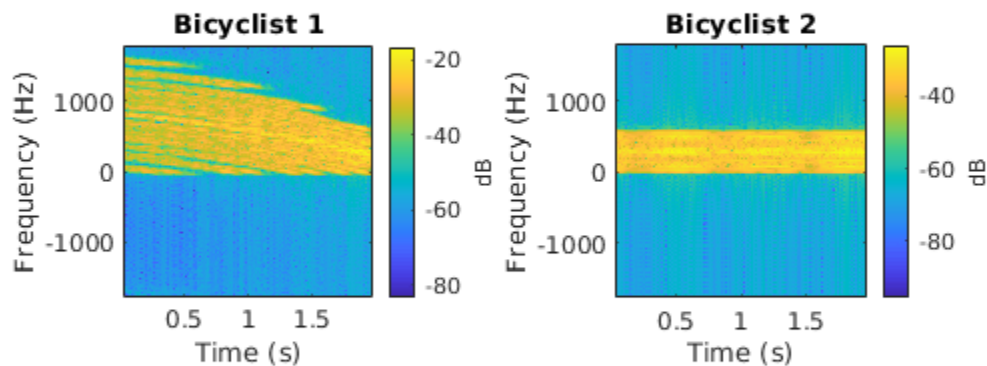
The amplitudes of the Bicyclist 2 signature are much weaker than those of Bicyclist 1, and the signatures of the two bicyclists overlap. When they overlap, the two signatures cannot be visually distinguished. However, the neural network classifies the scene correctly.

Another case of interest is when the network confuses car noise with a bicyclist, as in signature #267 of the car-noise-corrupted test data:

```
figure
k = 267;
imagesc(T,F,testDataCarNoise(:,:,:,k))
axis xy
xlabel('Time (s)')
ylabel('Frequency (Hz)')
title('Ground Truth: '+string(testLabelCarNoise(k))+', Prediction: '+string(predTestLabel(k)))
```

Ground Truth: bic, Prediction: ped+bic
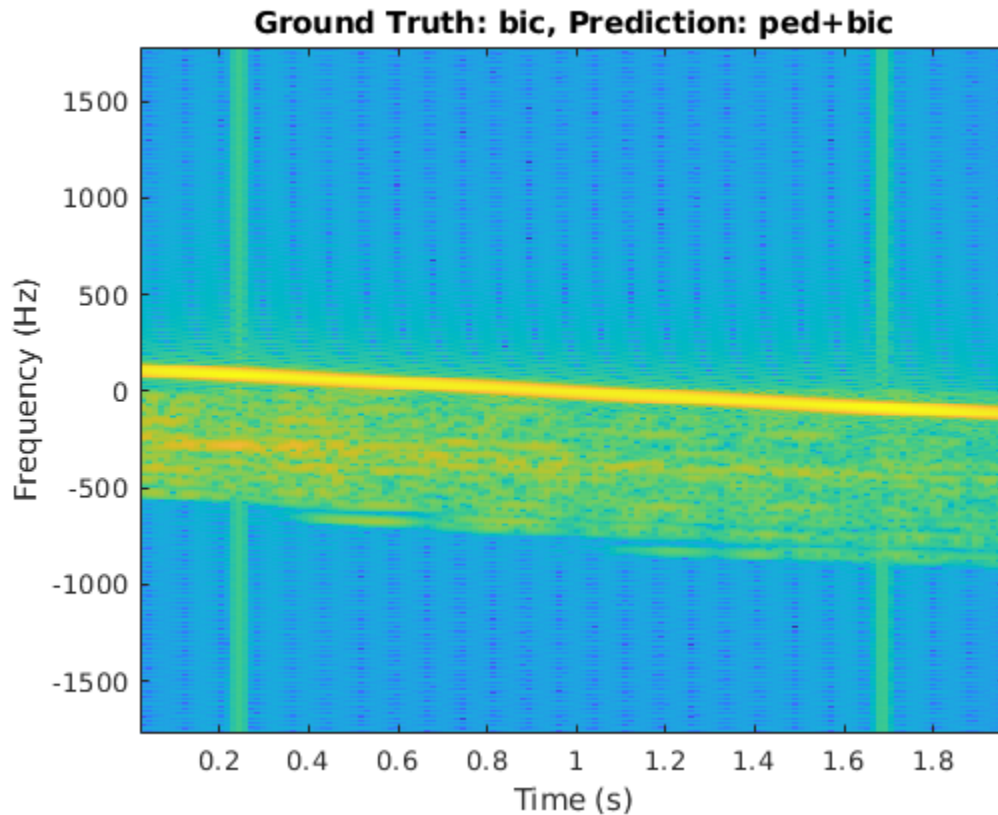
The signature of the bicyclist is weak compared to that of the car, and the signature has spikes from the car noise. Because the signature of the car closely resembles that of a bicyclist pedaling or a pedestrian walking at a low speed, and has little micro-Doppler effect, there is a high possibility that the network will classify the scene incorrectly.

References

[1] Chen, V. C. *The Micro-Doppler Effect in Radar*. London: Artech House, 2011.

[2] Gurbuz, S. Z., and Amin, M. G. "Radar-Based Human-Motion Recognition with Deep Learning: Promising Applications for Indoor Monitoring." *IEEE Signal Processing Magazine*. Vol. 36, Issue 4, 2019, pp. 16–28.

[3] Belgiovane, D., and C. C. Chen. "Micro-Doppler Characteristics of Pedestrians and Bicycles for Automotive Radar Sensors at 77 GHz." In *11th European Conference on Antennas and Propagation (EuCAP),* 2912–2916. Paris: European Association on Antennas and Propagation, 2017.

[4] Angelov, A., A. Robertson, R. Murray-Smith, and F. Fioranelli. "Practical Classification of Different Moving Targets Using Automotive Radar and Deep Neural Networks." *IET Radar, Sonar & Navigation*. Vol. 12, Number 10, 2017, pp. 1082–1089.

[5] Parashar, K. N., M. C. Oveneke, M. Rykunov, H. Sahli, and A. Bourdoux. "Micro-Doppler Feature Extraction Using Convolutional Auto-Encoders for Low Latency Target Classification." In *2017 IEEE Radar Conference (RadarConf)*, 1739–1744. Seattle: IEEE, 2017.

Radar Target Classification Using Machine Learning and Deep Learning

This example shows how to classify radar returns with both machine and deep learning approaches. The machine learning approach uses wavelet scattering feature extraction coupled with a support vector machine. Additionally, two deep learning approaches are illustrated: transfer learning using SqueezeNet and a Long Short-Term Memory (LSTM) recurrent neural network. Note that the data set used in this example does not require advanced techniques but the workflow is described because the techniques can be extended to more complex problems.

Introduction

Target classification is an important function in modern radar systems. This example uses machine and deep learning to classify radar echoes from a cylinder and a cone. Although this example uses the synthesized I/Q samples, the workflow is applicable to real radar returns.

RCS Synthesis

The next section shows how to create synthesized data to train the learning algorithms.

The following code simulates the RCS pattern of a cylinder with a radius of 1 meter and a height of 10 meters. The operating frequency of the radar is 850 MHz.

```
c = 3e8;
fc = 850e6;
[cylrcs,az,el] = rcscylinder(1,1,10,c,fc);
helperTargetRCSPatternPlot(az,el,cylrcs);
```
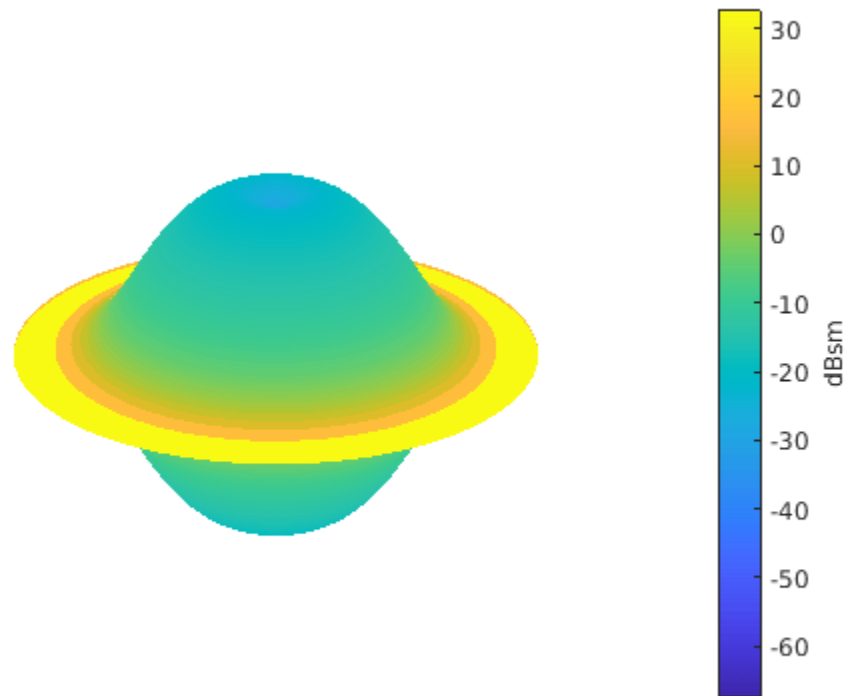
The pattern can then be applied to a backscatter radar target to simulate returns from different aspects angles.

```
cyltgt = phased.BackscatterRadarTarget('PropagationSpeed',c,...
    'OperatingFrequency',fc,'AzimuthAngles',az,'ElevationAngles',el,'RCSPattern',cylrcs);
```

The following plot shows how to simulate 100 returns of the cylinder over time. It is assumed that the cylinder under goes a motion that causes small vibrations around bore sight, as a result, the aspect angle changes from one sample to the next.

```
rng default;
N = 100;
az = 2*randn(1,N);
el = 2*randn(1,N);
cylrtn = cyltgt(ones(1,N),[az;el]);


plot(mag2db(abs(cylrtn)));
xlabel('Time Index')
ylabel('Target Return (dB)');
title('Target Return for Cylinder');
```

Target Return for Cylinder
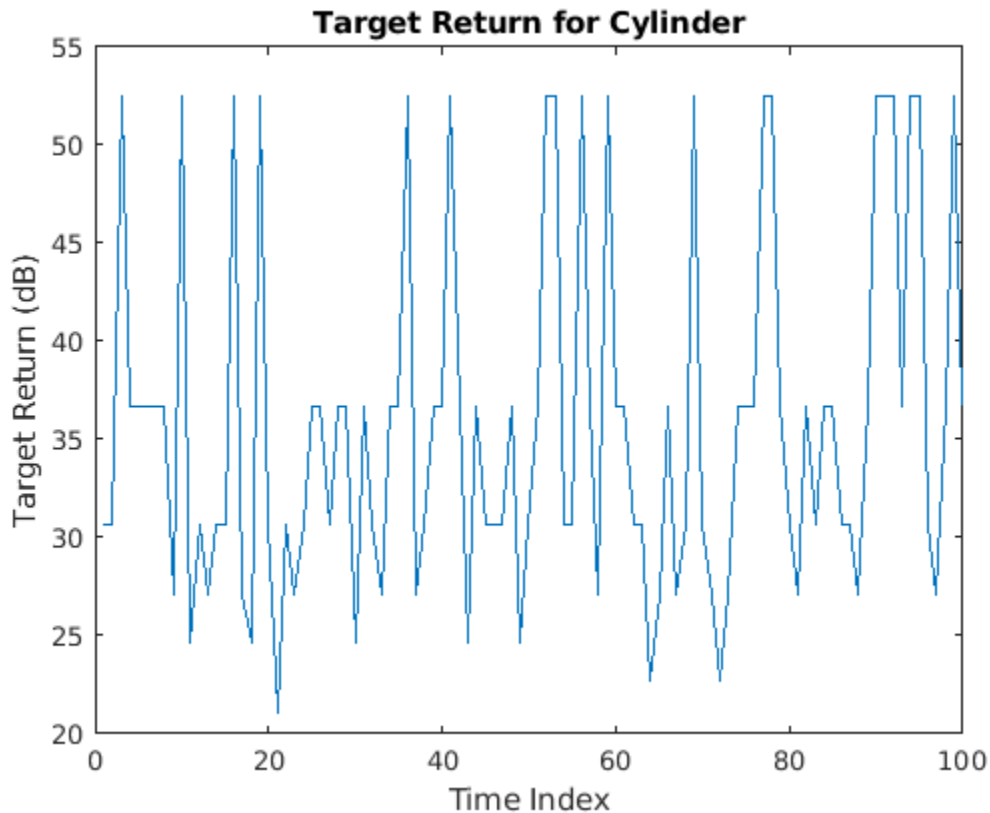
The return of the cone can be generated similarly. To create the training set, the above process is repeated for 5 arbitrarily selected cylinder radii. In addition, for each radius, 10 motion profiles are simulated by varying the incident angle following 10 randomly generated sinusoid curve around boresight. There are 701 samples in each motion profile, so there are 701-by-50 samples. The process is repeated for the cylinder target, which results in a 701-by-100 matrix of training data with 50 cylinder and 50 cone profiles. In the test set, we use 25 cylinder and 25 cone profiles to create a 701-by-50 training set. Because of the long computation time, the training data is precomputed and loaded below.

```
load('RCSClassificationReturnsTraining');
load('RCSClassificationReturnsTest');
```

As an example, the next plot shows the return for one of the motion profiles from each shape. The plots show how the values change over time for both the incident azimuth angles and the target returns.

```
subplot(2,2,1)
plot(cylinderAspectAngle(1,:))
ylim([-90 90])
grid on
title('Cylinder Aspect Angle vs. Time'); xlabel('Time Index'); ylabel('Aspect Angle (degrees)');
subplot(2,2,3)
plot(RCSReturns.Cylinder_1); ylim([-50 50]);
grid on
title('Cylinder Return'); xlabel('Time Index'); ylabel('Target Return (dB)');
subplot(2,2,2)
plot(coneAspectAngle(1,:)); ylim([-90 90]); grid on;
```

```
title('Cone Aspect Angle vs. Time'); xlabel('Time Index'); ylabel('Aspect Angle (degrees)');
subplot(2,2,4);
plot(RCSReturns.Cone_1); ylim([-50 50]); grid on;
title('Cone Return'); xlabel('Time Index'); ylabel('Target Return (dB)');
```
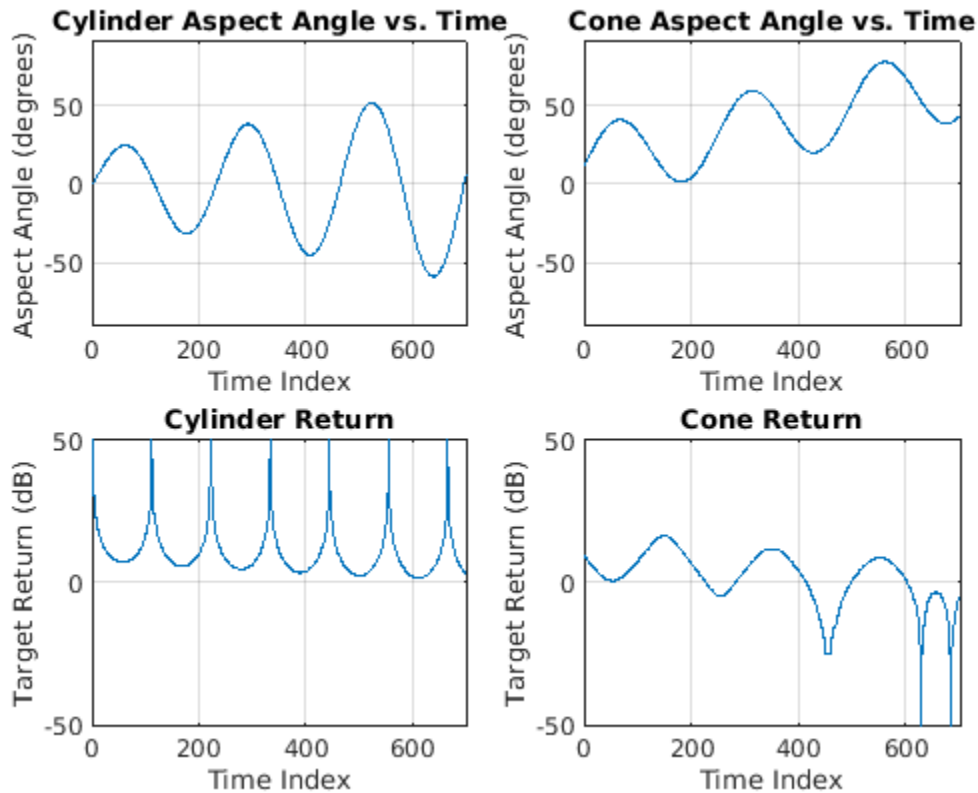

Wavelet Scattering

In the wavelet scattering feature extractor, data is propagated through a series of wavelet transforms, nonlinearities, and averaging to produce low-variance representations of time series. Wavelet time scattering yields signal representations insensitive to shifts in the input signal without sacrificing class discriminability.

The key parameters to specify in a wavelet time scattering network are the scale of the time invariant, the number of wavelet transforms, and the number of wavelets per octave in each of the wavelet filter banks. In many applications, the cascade of two filter banks is sufficient to achieve good performance. In this example, we construct a wavelet time scattering network with the two filter banks: 4 wavelets per octave in the first filter bank and 2 wavelets per octave in the second filter bank. The invariance scale is set to 701 samples, the length of the data.

```
sn = waveletScattering('SignalLength',701,'InvarianceScale',701,'QualityFactors',[4 2]);
```

Next, we obtain the scattering transforms of both the training and test sets.

```
sTrain = sn.featureMatrix(RCSReturns{:,:},'transform','log');
sTest = sn.featureMatrix(RCSReturnsTest{:,:},'transform','log');
```

For this example, use the mean of the scattering coefficients taken along each path.

```
TrainFeatures = squeeze(mean(sTrain,2))';
TestFeatures = squeeze(mean(sTest,2))';
```

Create the labels for training and learning

```
TrainLabels = repelem(categorical({'Cylinder','Cone'}),[50 50])';
TestLabels = repelem(categorical({'Cylinder','Cone'}),[25 25])';
```

Model Training

Fit a support vector machine model with a quadratic kernel to the scattering features and obtain the cross-validation accuracy.

```
template = templateSVM('KernelFunction', 'polynomial', ...
    'PolynomialOrder', 2, ...
    'KernelScale', 'auto', ...
    'BoxConstraint', 1, ...
    'Standardize', true);
classificationSVM = fitcecoc(...
    TrainFeatures, ...
    TrainLabels, ...
    'Learners', template, ...
    'Coding', 'onevsone', ...
    'ClassNames', categorical({'Cylinder','Cone'}));
partitionedModel = crossval(classificationSVM, 'KFold', 5);
[validationPredictions, validationScores] = kfoldPredict(partitionedModel);
validationAccuracy = (1 - kfoldLoss(partitionedModel, 'LossFun', 'ClassifError'))*100
```

```
validationAccuracy = 100
```

Target Classification

Using the trained SVM, classify the scattering features obtained from the test set.

```
predLabels = predict(classificationSVM,TestFeatures);
accuracy = sum(predLabels == TestLabels )/numel(TestLabels)*100
```

```
accuracy = 100
```

Plot the confusion matrix.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
ccDCNN = confusionchart(TestLabels,predLabels);
ccDCNN.Title = 'Confusion Chart';
ccDCNN.ColumnSummary = 'column-normalized';
ccDCNN.RowSummary = 'row-normalized';
```

Confusion Chart
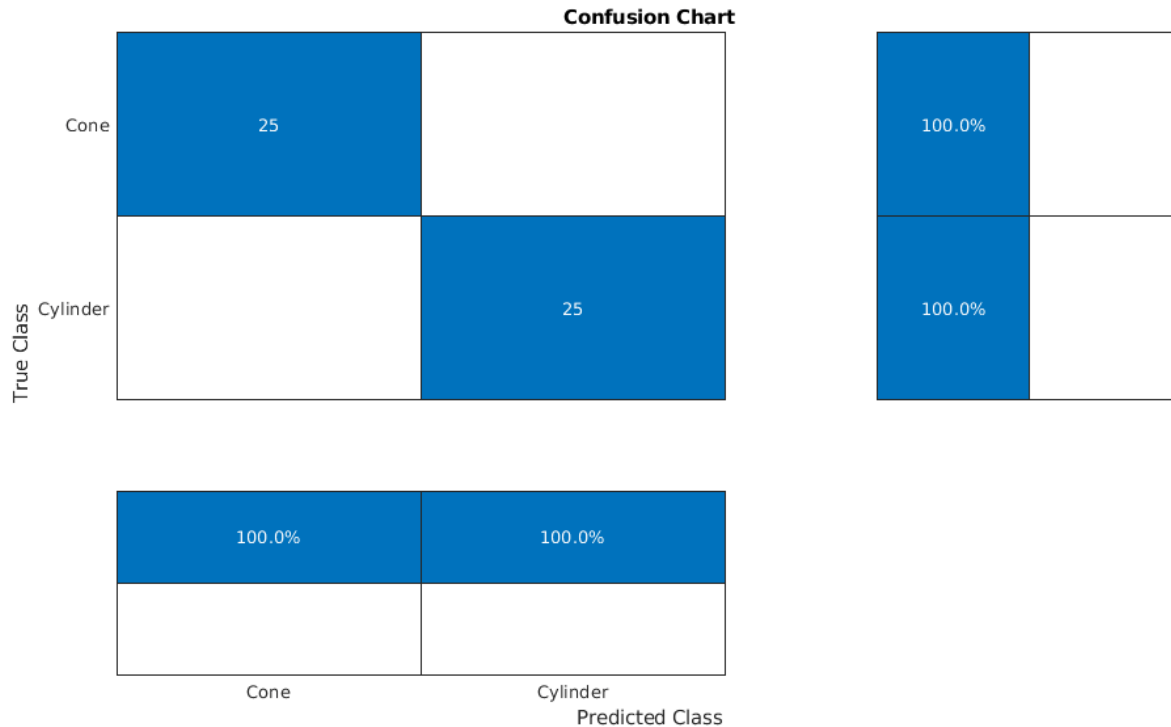
For more complex data sets, a deep learning workflow may improve performance.

Transfer Learning with a CNN

SqueezeNet is a deep convolutional neural network (CNN) trained for images in 1,000 classes as used in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC). In this example, we reuse the pre-trained SqueezeNet to classify radar returns belonging to one of two classes.

Load SqueezeNet.

```
snet = squeezenet;
```

SqueezeNet consists of 68 layers. Like all DCNNs, SqueezeNet cascades convolutional operators followed by nonlinearities and pooling, or averaging. SqueezeNet expects an image input of size 227-by-227-by-3, which you can see with the following code.

```
snet.Layers(1)

ans =
  ImageInputLayer with properties:

                    Name: 'data'
               InputSize: [227 227 3]

   Hyperparameters
         DataAugmentation: 'none'
            Normalization: 'zerocenter'
   NormalizationDimension: 'auto'
                     Mean: [1×1×3 single]
```

Additionally, SqueezeNet is configured to recognized 1,000 different classes, which you can see with the following code.

```
snet.Layers(68)

ans =
  ClassificationOutputLayer with properties:

           Name: 'ClassificationLayer_predictions'
        Classes: [1000×1 categorical]
   ClassWeights: 'none'
     OutputSize: 1000

  Hyperparameters
   LossFunction: 'crossentropyex'
```

In a subsequent section, we will modify select layers of SqueezeNet in order to apply it to our classification problem.

Continuous Wavelet Transform

SqueezeNet is designed to discriminate differences in images and classify the results. Therefore, in order to use SqueezeNet to classify radar returns, we must transform the 1-D radar return time series into an image. A common way to do this is to use a time-frequency representation (TFR). There are a number of choices for a time-frequency representation of a signal and which one is most appropriate depends on the signal characteristics. To determine which TFR may be appropriate for this problem, randomly choose and plot a few radar returns from each class.

```
rng default;
idxCylinder = randperm(50,2);
idxCone = randperm(50,2)+50;
```

It is evident that the radar returns previously shown are characterized by slowing varying changes punctuated by large transient decreases as described earlier. A wavelet transform is ideally suited to sparsely representing such signals. Wavelets shrink to localize transient phenomena with high temporal resolution and stretch to capture slowly varying signal structure. Obtain and plot the continuous wavelet transform of one of the cylinder returns.

```
cwt(RCSReturns{:,idxCylinder(1)},'VoicesPerOctave',8)
```
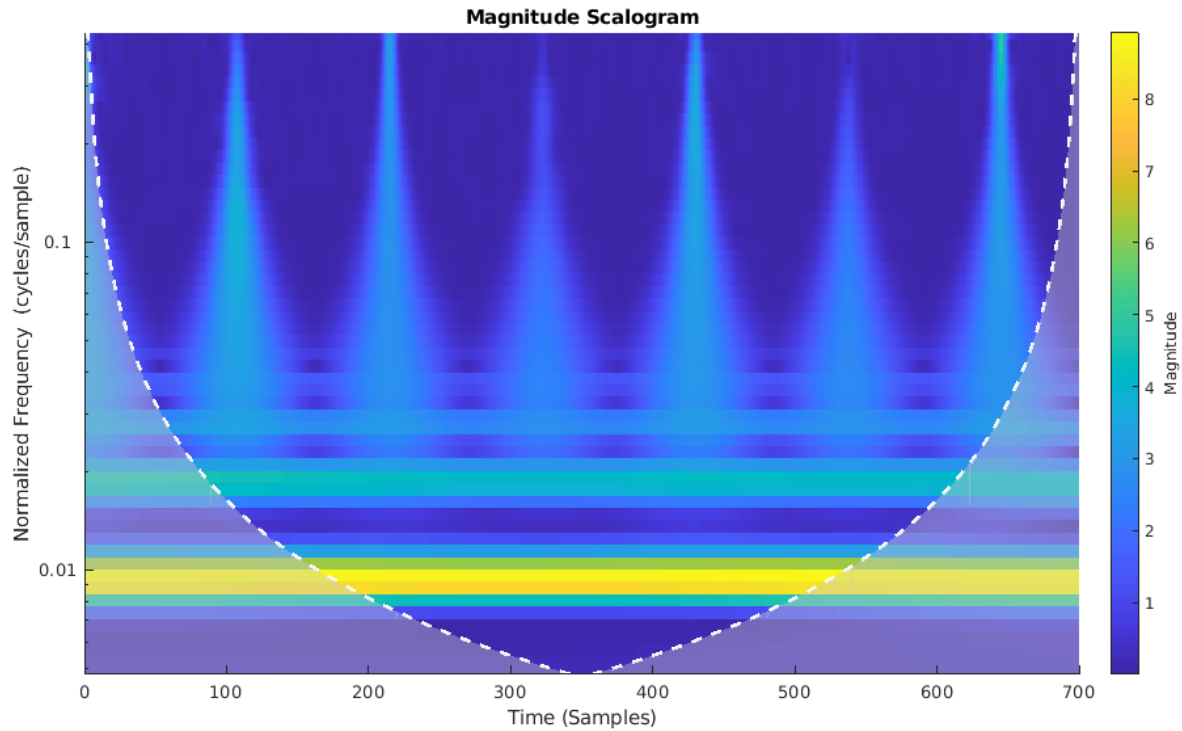
The CWT simultaneously captures both the slowly varying (low frequency) fluctuations and the transient phenomena. Contrast the CWT of the cylinder return with one from a cone target.

```
cwt(RCSReturns{:,idxCone(2)},'VoicesPerOctave',8);
```
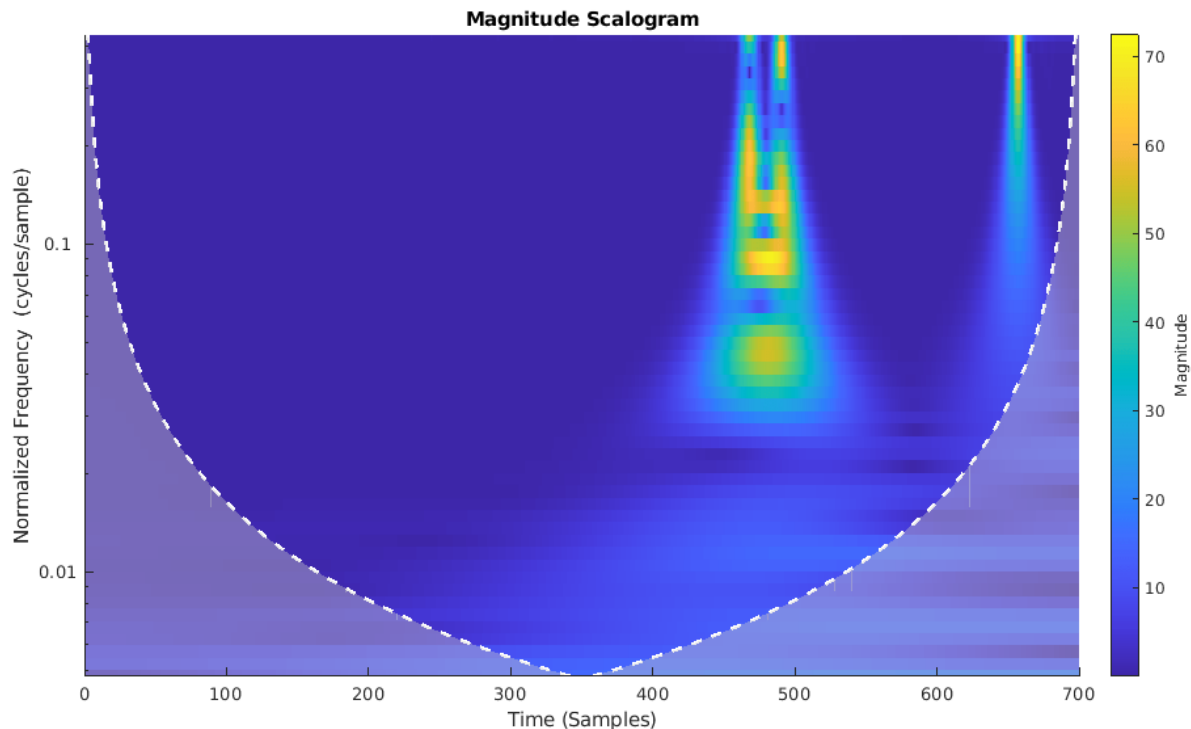
Because of the apparent importance of the transients in determining whether the target return originates from a cylinder or cone target, we select the CWT as the ideal TFR to use. After obtaining the CWT for each target return, we make images from the CWT of each radar return. These images are resized to be compatible with SqueezeNet's input layer and we leverage SqueezeNet to classify the resulting images.

Image Preparation

The helper function, `helpergenWaveletTFImg`, obtains the CWT for each radar return, reshapes the CWT to be compatible with SqueezeNet, and writes the CWT as a jpeg file. To run `helpergenWaveletTFImg`, choose a `parentDir` where you have write permission. This example uses `tempdir`, but you may use any folder on your machine where you have write permission. The helper function creates `Training` and `Test` set folders under `parentDir` as well as creating `Cylinder` and `Cone` subfolders under both `Training` and `Test`. These folders are populated with jpeg images to be used as inputs to SqueezeNet.

```
parentDir = tempdir;
helpergenWaveletTFImg(parentDir,RCSReturns,RCSReturnsTest)
```

```
Generating Time-Frequency Representations...Please Wait
   Creating Cylinder Time-Frequency Representations ... Done
   Creating Cone Time-Frequency Representations ... Done
   Creating Cylinder Time-Frequency Representations ... Done
   Creating Cone Time-Frequency Representations ... Done
```

Now use `imageDataStore` to manage file access from the folders in order to train SqueezeNet. Create datastores for both the training and test data.

```
trainingData= imageDatastore(fullfile(parentDir,'Training'), 'IncludeSubfolders', true,...
    'LabelSource', 'foldernames');
testData = imageDatastore(fullfile(parentDir,'Test'),'IncludeSubfolders',true,...
    'LabelSource','foldernames');
```

In order to use SqueezeNet with this binary classification problem, we need to modify a couple layers. First, we change the last learnable layer in SqueezeNet (layer 64) to have the same number of 1-by-1 convolutions as our new number of classes, 2.

```
lgraphSqueeze = layerGraph(snet);
convLayer = lgraphSqueeze.Layers(64);
numClasses = numel(categories(trainingData.Labels));
newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','binaryconv', ...
        'WeightLearnRateFactor',10, ...
        'BiasLearnRateFactor',10);
lgraphSqueeze = replaceLayer(lgraphSqueeze,convLayer.Name,newLearnableLayer);
classLayer = lgraphSqueeze.Layers(end);
newClassLayer = classificationLayer('Name','binary');
lgraphSqueeze = replaceLayer(lgraphSqueeze,classLayer.Name,newClassLayer);
```

Finally, set the options for re-training SqueezeNet. Set the initial learn rate to 1e-4, set the maximum number of epochs to 15, and the minibatch size to 10. Use stochastic gradient descent with momentum.

```
ilr = 1e-4;
mxEpochs = 15;
mbSize =10;
opts = trainingOptions('sgdm', 'InitialLearnRate', ilr, ...
    'MaxEpochs',mxEpochs , 'MiniBatchSize',mbSize, ...
    'Plots', 'training-progress','ExecutionEnvironment','cpu');
```

Train the network. If you have a compatible GPU, `trainNetwork` automatically utilizes the GPU and training should complete in less than one minute. If you do not have a compatible GPU, `trainNetwork` utilizes the CPU and training should take around five minutes. Training times do vary based on a number of factors. In this case, the training takes place on a cpu by setting the `ExecutionEnvironment` parameter to `cpu`.

```
CWTnet = trainNetwork(trainingData,lgraphSqueeze,opts);
```

Initializing input data normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Mini-batch Loss	Base Learning Rate
1	1	00:00:06	60.00%	2.6639	1.0000e-04
5	50	00:01:08	100.00%	0.0001	1.0000e-04
10	100	00:02:11	100.00%	0.0002	1.0000e-04
15	150	00:03:12	100.00%	2.2264e-05	1.0000e-04

Use the trained network to predict target returns in the held-out test set.

```
predictedLabels = classify(CWTnet,testData,'ExecutionEnvironment','cpu');
accuracy = sum(predictedLabels == testData.Labels)/50*100
```

```
accuracy = 100
```

Plot the confusion chart along with the precision and recall. In this case, 100% of the test samples are classified correctly.

```
figure('Units','normalized','Position',[0.2 0.2 0.5 0.5]);
ccDCNN = confusionchart(testData.Labels,predictedLabels);
ccDCNN.Title = 'Confusion Chart';
ccDCNN.ColumnSummary = 'column-normalized';
ccDCNN.RowSummary = 'row-normalized';
```
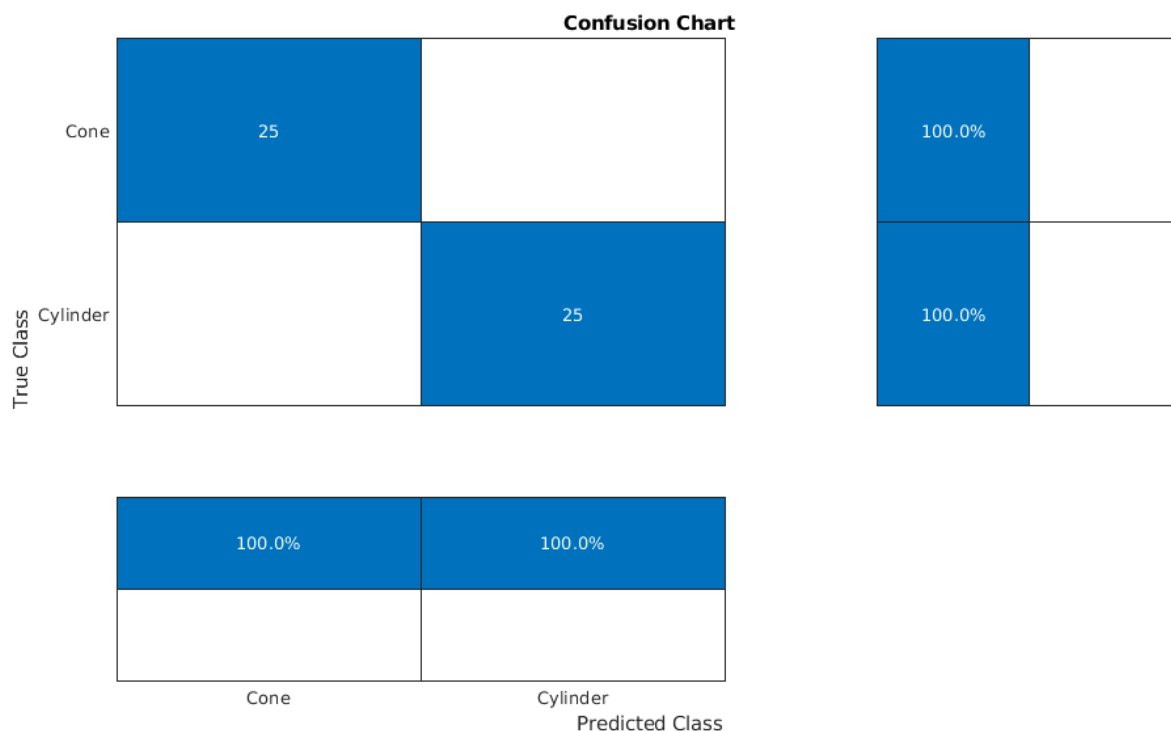

LSTM

In the final section of this example, an LSTM workflow is described. First the LSTM layers are defined:

```
LSTMlayers = [ ...
    sequenceInputLayer(1)
    bilstmLayer(100,'OutputMode','last')
    fullyConnectedLayer(2)
    softmaxLayer
    classificationLayer
    ];
options = trainingOptions('adam', ...
    'MaxEpochs',30, ...
    'MiniBatchSize', 150, ...
    'InitialLearnRate', 0.01, ...
```

```
    'GradientThreshold', 1, ...
    'plots','training-progress', ...
    'Verbose',false,'ExecutionEnvironment','cpu');
trainLabels = repelem(categorical({'cylinder','cone'}),[50 50]);
trainLabels = trainLabels(:);
trainData = num2cell(table2array(RCSReturns)',2);
testData = num2cell(table2array(RCSReturnsTest)',2);
testLabels = repelem(categorical({'cylinder','cone'}),[25 25]);
testLabels = testLabels(:);
RNNnet = trainNetwork(trainData,trainLabels,LSTMlayers,options);
```

The accuracy for this system is also plotted.

```
predictedLabels = classify(RNNnet,testData,'ExecutionEnvironment','cpu');
accuracy = sum(predictedLabels == testLabels)/50*100
```

```
accuracy = 100
```

Conclusion

This example presents a workflow for performing radar target classification using machine and deep learning techniques. Although this example used synthesized data to do training and testing, it can be easily extended to accommodate real radar returns. Because of the signal characteristics, wavelet techniques were used for both the machine learning and CNN approaches.

With this dataset we were also obtained to achieve similar accuracy by just feeding the raw data into a LSTM. In more complicated datasets, the raw data may be too inherently variable for the model to learn robust features from the raw data and you may have to resort to feature extraction prior to using a LSTM.

Radar and Communications Waveform Classification Using Deep Learning

This example shows how to classify radar and communications waveforms using the Wigner-Ville distribution (WVD) and a deep convolutional neural network (CNN).

Modulation classification is an important function for an intelligent receiver. Modulation classification has numerous applications, such as cognitive radar and software-defined radio. Typically, to identify these waveforms and classify them by modulation type, it is necessary to define meaningful features and input them into a classifier. While effective, this procedure can require extensive effort and domain knowledge to yield an accurate classification. This example explores a framework to automatically extract time-frequency features from signals and perform signal classification using a deep learning network.

The first part of this example simulates a radar classification system that synthesizes three pulsed radar waveforms and classifies them. The radar waveforms are:

- Rectangular
- Linear frequency modulation (LFM)
- Barker Code

A radar classification system does not exist in isolation. Rather, it resides in an increasingly occupied frequency spectrum, competing with other transmitted sources such as communications systems, radio, and navigation systems. The second part of this example extends the network to include additional communication modulation types. In addition to the first set of radar waveforms, the extended network synthesizes and identifies these communication waveforms:

- Gaussian frequency shift keying (GFSK)
- Continuous phase frequency shift keying (CPFSK)
- Broadcast frequency modulation (B-FM)
- Double sideband amplitude modulation (DSB-AM)
- Single sideband amplitude modulation (SSB-AM)

This example primarily focuses on radar waveforms, with the classification being extended to include a small set of amplitude and frequency modulation communications signals. See "Modulation Classification with Deep Learning" (Communications Toolbox) for a full workflow of modulation classification with a wide array of communication signals.

Generate Radar Waveforms

Generate 3000 signals with a sample rate of 100 MHz for each modulation type. Use `phased.RectangularWaveform` for rectangular pulses, `phased.LinearFMWaveform` for LFM, and `phased.PhaseCodedWaveform` for phase coded pulses with Barker code.

Each signal has unique parameters and is augmented with various impairments to make it more realistic. For each waveform, the pulse width and repetition frequency will be randomly generated. For LFM waveforms, the sweep bandwidth and direction are randomly generated. For Barker waveforms, the chip width and number are generated randomly. All signals are impaired with white Gaussian noise using the `awgn` function with a random signal-to-noise ratio in the range of [–6, 30] dB. A frequency offset with a random carrier frequency in the range of [`Fs/6`, `Fs/5`] is applied to

each signal using the `comm.PhaseFrequencyOffset` object. Lastly, each signal is passed through a multipath Rician fading channel, `comm.RicianChannel`.

The helper function `helperGenerateRadarWaveforms` creates and augments each modulation type.

```
rng default
[wav, modType] = helperGenerateRadarWaveforms();
```

Plot the Fourier transform for a few of the LFM waveforms to show the variances in the generated set.

```
idLFM = find(modType == "LFM",3);
nfft = 2^nextpow2(length(wav{1}));
f = (0:(nfft/2-1))/nfft*100e6;

figure
subplot(1,3,1)
Z = fft(wav{idLFM(1)},nfft);
plot(f/1e6,abs(Z(1:nfft/2)))
xlabel('Frequency (MHz)');ylabel('Amplitude');axis square
subplot(1,3,2)
Z = fft(wav{idLFM(2)},nfft);
plot(f/1e6,abs(Z(1:nfft/2)))
xlabel('Frequency (MHz)');ylabel('Amplitude');axis square
subplot(1,3,3)
Z = fft(wav{idLFM(3)},nfft);
plot(f/1e6,abs(Z(1:nfft/2)))
xlabel('Frequency (MHz)');ylabel('Amplitude');axis square
```
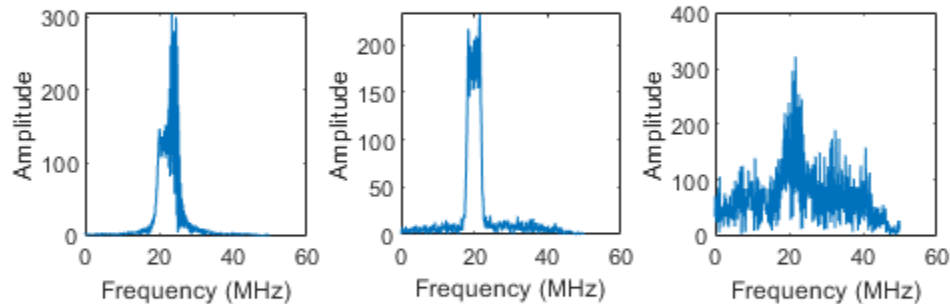
Feature Extraction Using Wigner-Ville Distribution

To improve the classification performance of machine learning algorithms, a common approach is to input extracted features in place of the original signal data. The features provide a representation of the input data that makes it easier for a classification algorithm to discriminate across the classes. The Wigner-Ville distribution represents a time-frequency view of the original data that is useful for time varying signals. The high resolution and locality in both time and frequency provide good features for the identification of similar modulation types. Use the wvd function to compute the smoothed pseudo WVD for each of the modulation types.

```
figure
subplot(1,3,1)
wvd(wav{find(modType == "Rect",1)},100e6,'smoothedPseudo')
axis square; colorbar off; title('Rect')
subplot(1,3,2)
wvd(wav{find(modType == "LFM",1)},100e6,'smoothedPseudo')
axis square; colorbar off; title('LFM')
subplot(1,3,3)
wvd(wav{find(modType == "Barker",1)},100e6,'smoothedPseudo')
axis square; colorbar off; title('Barker')
```
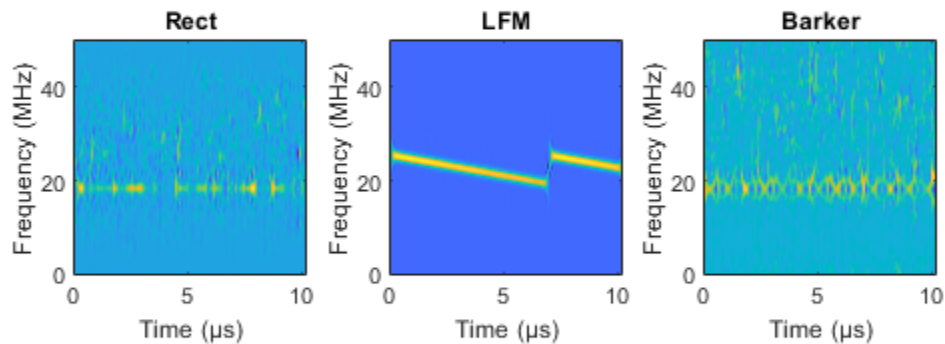
To store the smoothed-pseudo Wigner-Ville distribution of the signals, first create the directory `TFDDatabase` inside your temporary directory `tempdir`. Then create subdirectories in `TFDDatabase` for each modulation type. For each signal, compute the smoothed-pseudo Wigner-Ville distribution, and downsample the result to a 227-by-227 matrix. Save the matrix as a `.png` image file in the subdirectory corresponding to the modulation type of the signal. The helper function `helperGenerateTFDfiles` performs all these steps. This process will take several minutes due to the large database size and the complexity of the `wvd` algorithm. You can replace `tempdir` with another directory where you have write permission.

```
parentDir = tempdir;
dataDir = 'TFDDatabase';
helperGenerateTFDfiles(parentDir,dataDir,wav,modType,100e6)
```

Create an image datastore object for the created folder to manage the image files used for training the deep learning network. This step avoids having to load all images into memory. Specify the label source to be folder names. This assigns the modulation type of each signal according to the folder name.

```
folders = fullfile(parentDir,dataDir,{'Rect','LFM','Barker'});
imds = imageDatastore(folders,...
    'FileExtensions','.png','LabelSource','foldernames','ReadFcn',@readTFDForSqueezeNet);
```

The network is trained with 80% of the data and tested on with 10%. The remaining 10% is used for validation. Use the `splitEachLabel` function to divide the `imageDatastore` into training, validation, and testing sets.

```
[imdsTrain,imdsTest,imdsValidation] = splitEachLabel(imds,0.8,0.1);
```

1-273

Set Up Deep Learning Network

Before the deep learning network can be trained, define the network architecture. This example utilizes transfer learning SqueezeNet, a deep CNN created for image classification. Transfer learning is the process of retraining an existing neural network to classify new targets. This network accepts image input of size 227-by-227-by-3. Prior to input to the network, the custom read function `readTFDForSqueezeNet` transforms the two-dimensional time-frequency distribution to an RGB image of the correct size. SqueezeNet performs classification of 1000 categories in its default configuration.

Load SqueezeNet.

```
net = squeezenet;
```

Extract the layer graph from the network. Confirm that SqueezeNet is configured for images of size 227-by-227-by-3.

```
lgraphSqz = layerGraph(net);
lgraphSqz.Layers(1)
```

```
ans =
  ImageInputLayer with properties:

                    Name: 'data'
               InputSize: [227 227 3]

   Hyperparameters
         DataAugmentation: 'none'
            Normalization: 'zerocenter'
   NormalizationDimension: 'auto'
                     Mean: [1×1×3 single]
```

To tune SqueezeNet for this example, three of the last six layers need to be modified to classify the three radar modulation types of interest. Inspect the last six network layers.

```
lgraphSqz.Layers(end-5:end)
```

```
ans =
  6×1 Layer array with layers:

     1   'drop9'                            Dropout                      50% dropout
     2   'conv10'                           Convolution                  1000 1×1×512 convolutio
     3   'relu_conv10'                      ReLU                         ReLU
     4   'pool10'                           2-D Global Average Pooling   2-D global average pool
     5   'prob'                             Softmax                      softmax
     6   'ClassificationLayer_predictions'  Classification Output        crossentropyex with 'te
```

Replace the drop9 layer, the last dropout layer in the network, with a dropout layer of probability 0.6.

```
tmpLayer = lgraphSqz.Layers(end-5);
newDropoutLayer = dropoutLayer(0.6,'Name','new_dropout');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newDropoutLayer);
```

The last learnable layer in SqueezeNet is a 1-by-1 convolutional layer, conv10. Replace the layer with a new convolutional layer with the number of filters equal to the number of modulation types. Also increase the learning rate factors of the new layer.

```
numClasses = 3;
tmpLayer = lgraphSqz.Layers(end-4);
newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',20, ...
        'BiasLearnRateFactor',20);
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newLearnableLayer);
```

Replace the classification layer with a new one without class labels.

```
tmpLayer = lgraphSqz.Layers(end);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newClassLayer);
```

Inspect the last six layers of the network. Confirm the dropout, convolutional, and output layers have been changed.

```
lgraphSqz.Layers(end-5:end)
```

```
ans =
  6×1 Layer array with layers:

     1   'new_dropout'       Dropout                  60% dropout
     2   'new_conv'          Convolution              3 1×1 convolutions with stride [1  1] a
     3   'relu_conv10'       ReLU                     ReLU
     4   'pool10'            2-D Global Average Pooling   2-D global average pooling
     5   'prob'              Softmax                  softmax
     6   'new_classoutput'   Classification Output    crossentropyex
```

Choose options for the training process that ensures good network performance. Refer to the `trainingOptions` documentation for a description of each option.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',128, ...
    'MaxEpochs',5, ...
    'InitialLearnRate',1e-3, ...
    'Shuffle','every-epoch', ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'ValidationData',imdsValidation);
```

Train the Network

Use the `trainNetwork` command to train the created CNN. Because the data set is large in size, the process can take several minutes. If your machine has a GPU and Parallel Computing Toolbox™, then MATLAB® automatically uses the GPU for training. Otherwise, it uses the CPU. The training accuracy plots in the figure show the progress of the network's learning across all iterations. On the three radar modulation types, the network classifies almost 100% of the training signals correctly.

```
trainedNet = trainNetwork(imdsTrain,lgraphSqz,options);
```
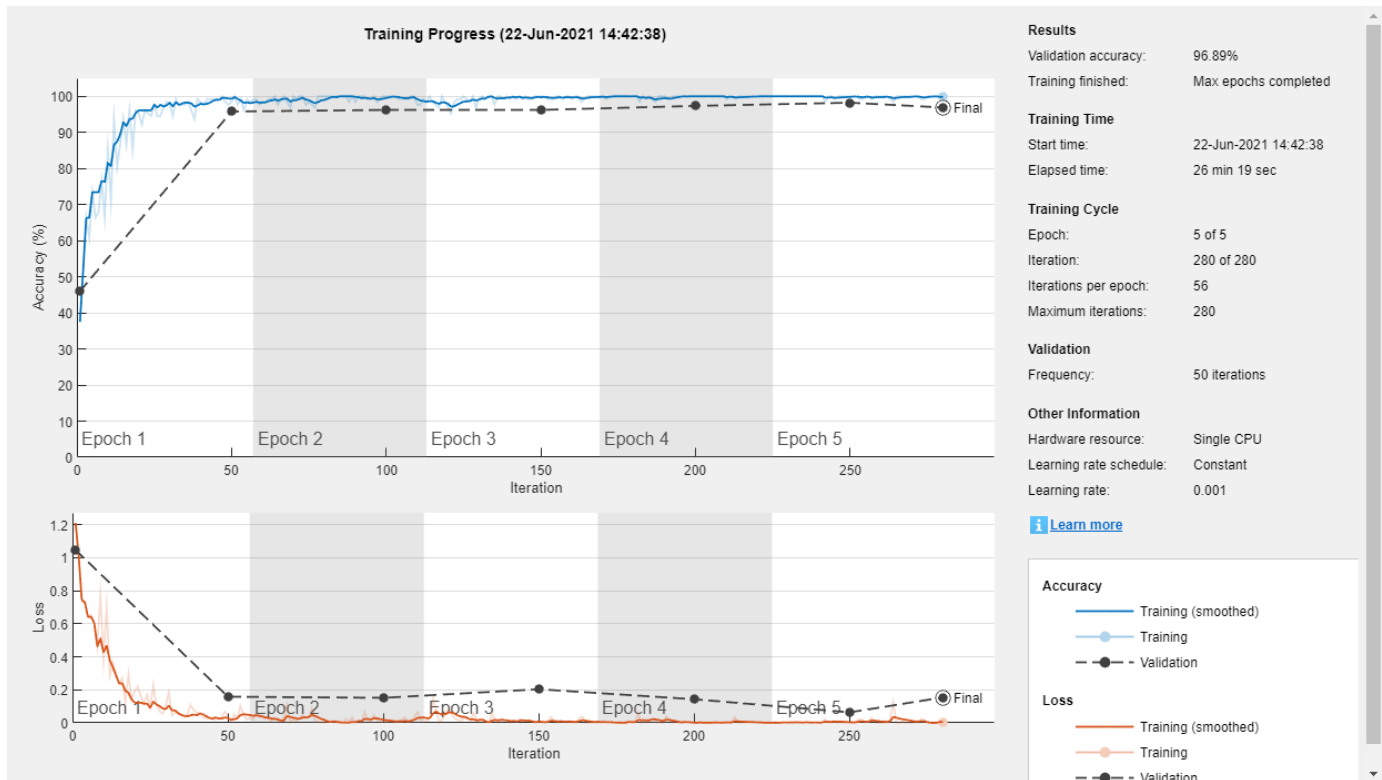
Evaluate Performance on Radar Waveforms

Use the trained network to classify the testing data using the `classify` command. A confusion matrix is one method to visualize classification performance. Use the `confusionchart` command to calculate and visualize the classification accuracy. For the three modulation types input to the network, almost all of the phase coded, LFM, and rectangular waveforms are correctly identified by the network.

```
predicted = classify(trainedNet,imdsTest);
figure
confusionchart(imdsTest.Labels,predicted,'Normalization','column-normalized')
```
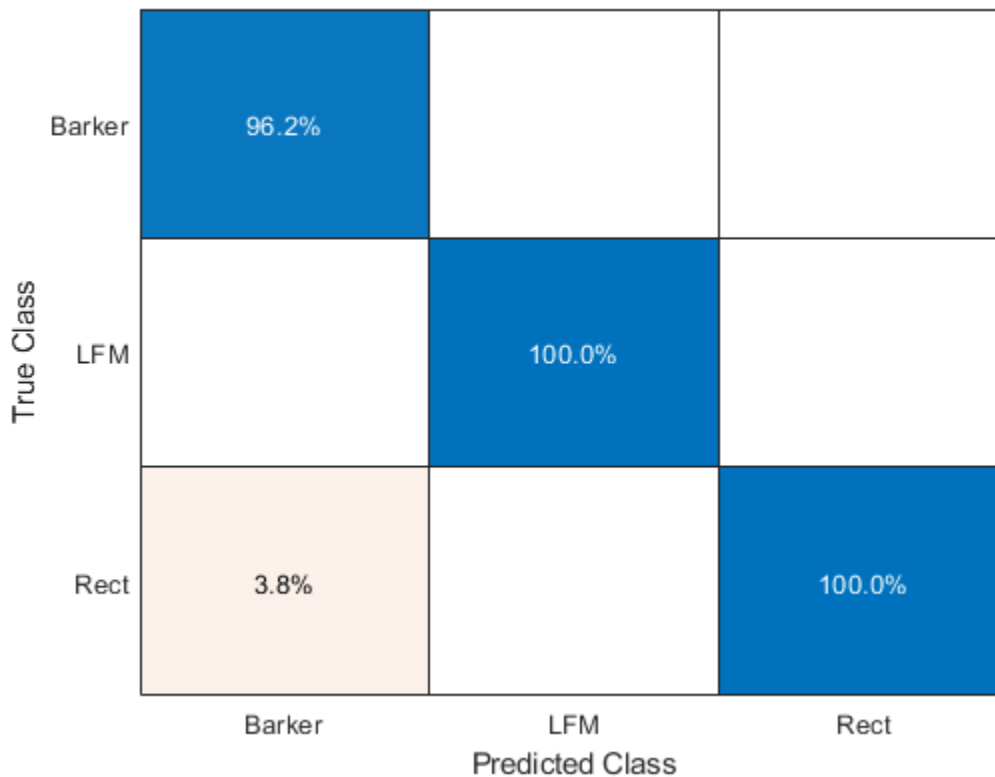
Generate Communications Waveforms and Extract Features

The frequency spectrum of a radar classification system must compete with other transmitted sources. See how the created network extends to incorporate other simulated modulation types. Another MathWorks® example, "Modulation Classification with Deep Learning" (Communications Toolbox), performs modulation classification of several different modulation types using Communications Toolbox™. The helper function `helperGenerateCommsWaveforms` generates and augments a subset of the modulation types used in that example. Since the WVD loses phase information, a subset of only the amplitude and frequency modulation types is used.

See the example link for an in-depth description of the workflow necessary for digital and analog modulation classification and the techniques used to create these waveforms. For each modulation type, use `wvd` to extract time-frequency features and visualize.

```
[wav, modType] = helperGenerateCommsWaveforms();

figure
subplot(2,3,1)
wvd(wav{find(modType == "GFSK",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('GFSK')
subplot(2,3,2)
wvd(wav{find(modType == "CPFSK",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('CPFSK')
subplot(2,3,3)
wvd(wav{find(modType == "B-FM",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('B-FM')
subplot(2,3,4)
```

```
wvd(wav{find(modType == "SSB-AM",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('SSB-AM')
subplot(2,3,5)
wvd(wav{find(modType == "DSB-AM",1)},200e3,'smoothedPseudo')
axis square; colorbar off; title('DSB-AM')
```
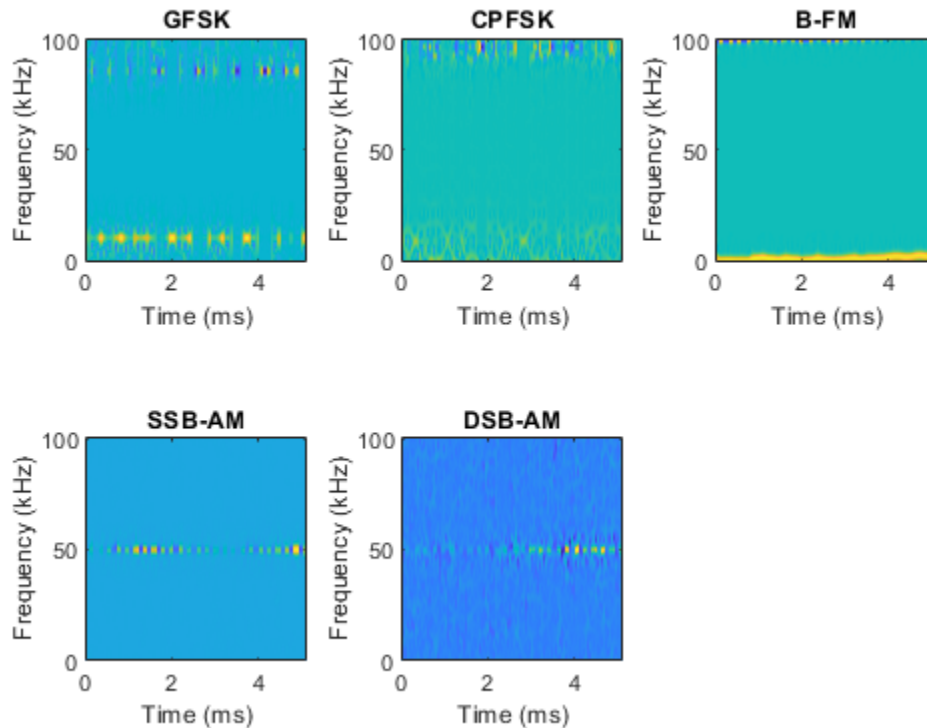

Use the helper function `helperGenerateTFDfiles` again to compute the smoothed pseudo WVD for each input signal. Create an image datastore object to manage the image files of all modulation types.

```
helperGenerateTFDfiles(parentDir,dataDir,wav,modType,200e3)
folders = fullfile(parentDir,dataDir,{'Rect','LFM','Barker','GFSK','CPFSK','B-FM','SSB-AM','DSB-/
imds = imageDatastore(folders,...
    'FileExtensions','.png','LabelSource','foldernames','ReadFcn',@readTFDForSqueezeNet);
```

Again, divide the data into a training set, a validation set, and a testing set using the `splitEachLabel` function.

```
rng default
[imdsTrain,imdsTest,imdsValidation] = splitEachLabel(imds,0.8,0.1);
```

Adjust Deep Learning Network Architecture

Previously, the network architecture was set up to classify three modulation types. This must be updated to allow classification of all eight modulation types of both radar and communication signals. This is a similar process as before, with the exception that the `fullyConnectedLayer` now requires an output size of eight.

```
numClasses = 8;
net = squeezenet;
lgraphSqz = layerGraph(net);

tmpLayer = lgraphSqz.Layers(end-5);
newDropoutLayer = dropoutLayer(0.6,'Name','new_dropout');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newDropoutLayer);

tmpLayer = lgraphSqz.Layers(end-4);
newLearnableLayer = convolution2dLayer(1,numClasses, ...
        'Name','new_conv', ...
        'WeightLearnRateFactor',20, ...
        'BiasLearnRateFactor',20);
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newLearnableLayer);

tmpLayer = lgraphSqz.Layers(end);
newClassLayer = classificationLayer('Name','new_classoutput');
lgraphSqz = replaceLayer(lgraphSqz,tmpLayer.Name,newClassLayer);
```

Create a new set of training options.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',150, ...
    'MaxEpochs',10, ...
    'InitialLearnRate',1e-4, ...
    'Shuffle','every-epoch', ...
    'Verbose',false, ...
    'Plots','training-progress',...
    'ValidationData',imdsValidation);
```

Use the `trainNetwork` command to train the created CNN. For all modulation types, the training converges with an accuracy of about 95% correct classification.

```
trainedNet = trainNetwork(imdsTrain,lgraphSqz,options);
```
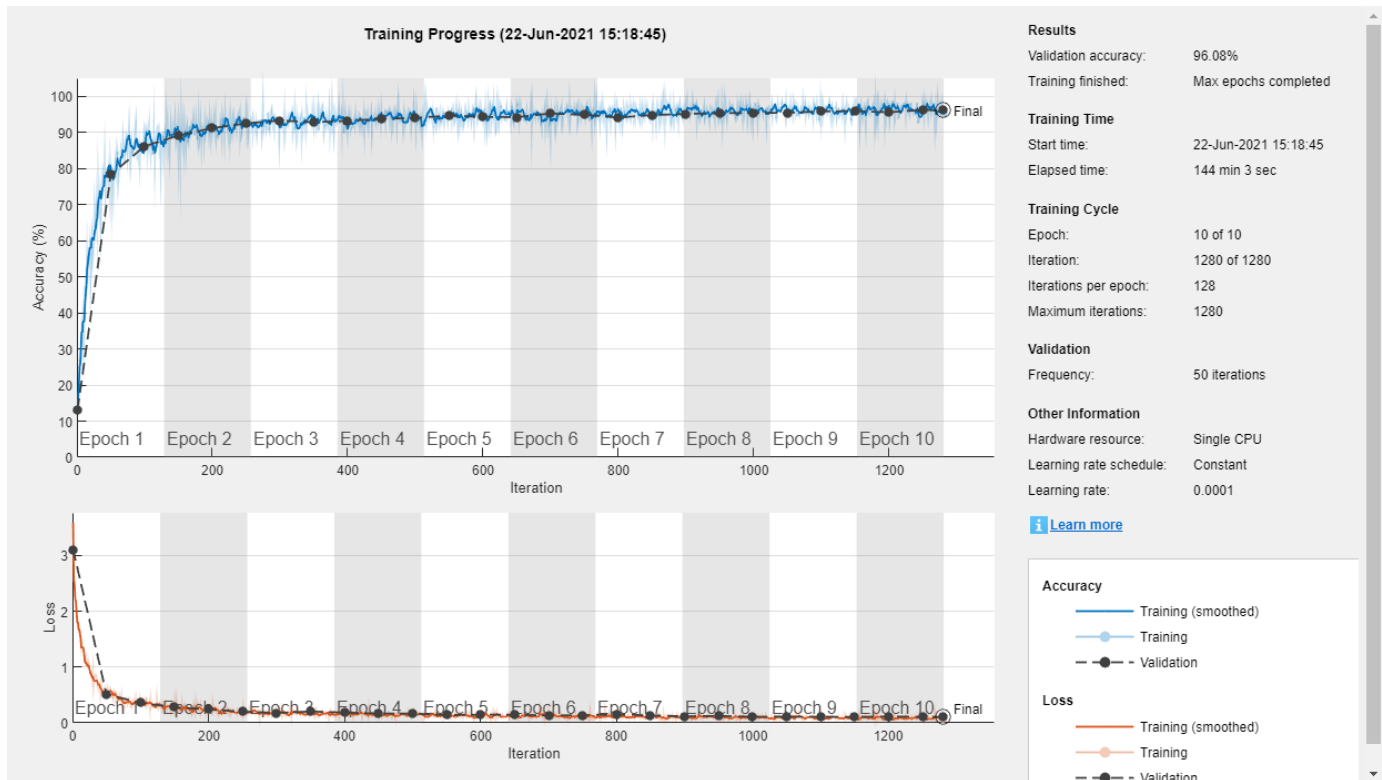
Evaluate Performance on All Signals

Use the `classify` command to classify the signals held aside for testing. Again, visualize the performance using `confusionchart`.

```
predicted = classify(trainedNet,imdsTest);
figure;
confusionchart(imdsTest.Labels,predicted,'Normalization','column-normalized')
```

For the eight modulation types input to the network, about 98% of B-FM, CPFSK, GFSK, Barker, and LFM modulation types were correctly classified. On average, about 85% of AM signals were correctly identified. From the confusion matrix, a high percentage of SSB-AM signals were misclassified as DSB-AM and DSB-AM signals as SSB-AM.

Investigate a few of these misclassifications to gain insight into the network's learning process. Use the `readimage` function on the image datastore to extract from the test dataset a single image from each class. The displayed WVD visually looks very similar. Since DSB-AM and SSB-AM signals have a very similar signature, this explains in part the network's difficulty in correctly classifying these two types. Further signal processing could make the differences between these two modulation types clearer to the network and result in improved classification.

```
DSB_DSB = readimage(imdsTest,find((imdsTest.Labels == 'DSB-AM') & (predicted == 'DSB-AM'),1));
DSB_SSB = readimage(imdsTest,find((imdsTest.Labels == 'DSB-AM') & (predicted == 'SSB-AM'),1));
SSB_DSB = readimage(imdsTest,find((imdsTest.Labels == 'SSB-AM') & (predicted == 'DSB-AM'),1));
SSB_SSB = readimage(imdsTest,find((imdsTest.Labels == 'SSB-AM') & (predicted == 'SSB-AM'),1));

figure
subplot(2,2,1)
imagesc(DSB_DSB(:,:,1))
axis square; title({'Actual Class: DSB-AM','Predicted Class: DSB-AM'})
subplot(2,2,2)
imagesc(DSB_SSB(:,:,1))
axis square; title({'Actual Class: DSB-AM','Predicted Class: SSB-AM'})
subplot(2,2,3)
imagesc(SSB_DSB(:,:,1))
axis square; title({'Actual Class: SSB-AM','Predicted Class: DSB-AM'})
```

```
subplot(2,2,4)
imagesc(SSB_SSB(:,:,1))
axis square; title({'Actual Class: SSB-AM','Predicted Class: SSB-AM'})
```
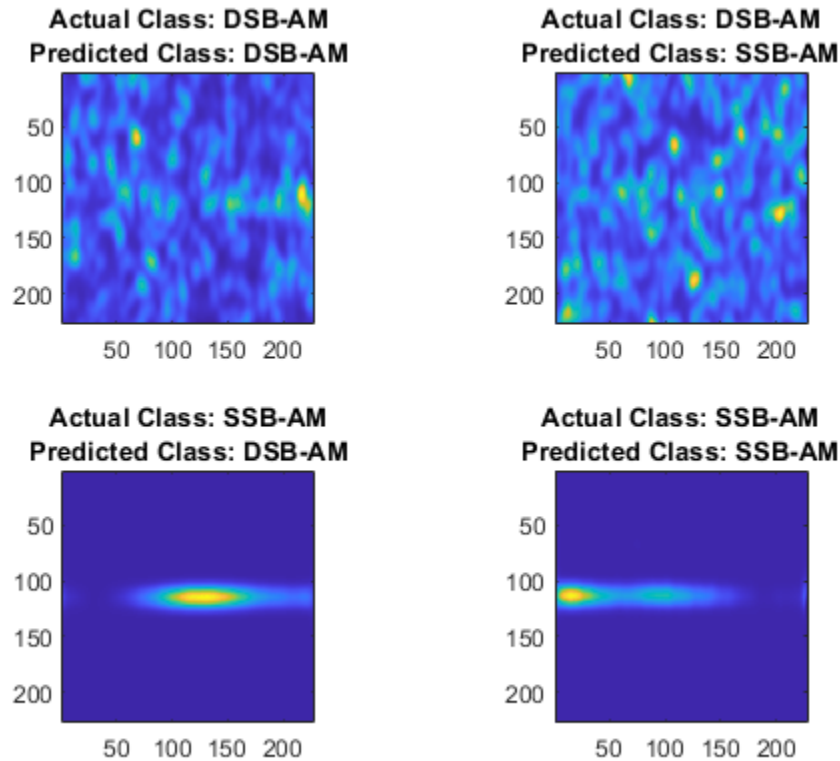

Summary

This example showed how radar and communications modulation types can be classified by using time-frequency techniques and a deep learning network. Further efforts for additional improvement could be investigated by utilizing time-frequency analysis available in Wavelet Toolbox™ and additional Fourier analysis available in Signal Processing Toolbox™.

References

[1] Brynolfsson, Johan, and Maria Sandsten. "Classification of one-dimensional non-stationary signals using the Wigner-Ville distribution in convolutional neural networks." *25th European Signal Processing Conference (EUSIPCO)*. IEEE, 2017.

[2] Liu, Xiaoyu, Diyu Yang, and Aly El Gamal. "Deep neural network architectures for modulation classification." *51st Asilomar Conference on Signals, Systems and Computers*. 2017.

[3] `Wang, Chao, Jian Wang, and Xudong Zhang. "Automatic radar waveform recognition based on time-frequency analysis and convolutional neural network." *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2017.

Spaceborne Synthetic Aperture Radar Performance Prediction

This example shows how to assess the performance of a spaceborne synthetic aperture radar (SAR) and compares theoretical limits with achievable requirements for a SAR system. SAR uses the motion of the radar antenna over a target region to provide finer azimuth resolution. Given the main parameters of both the radar (e.g. the operating frequency, antenna size, and bandwidth) and the platform it is mounted on (e.g. altitude, velocity, mounting position), determine performance parameters such as the footprint, the azimuth and range resolution, the signal to-noise ratio (SNR) of the SAR image, and the noise equivalent reflectivity (NER).

SAR System Parameters and Conventions

Consider a radar on a low Earth orbit (LEO) satellite operating in C-band at 5.5 GHz with a pulse bandwidth of 500 MHz. The satellite is at an altitude of 565 km and moves with a velocity of 7.0 km/s. The radar antenna dimensions are 5.2 m (along azimuth) by 1.1 m in height. Assume that returns are processed for 0.8 s and that the squint angle is 90 degrees.
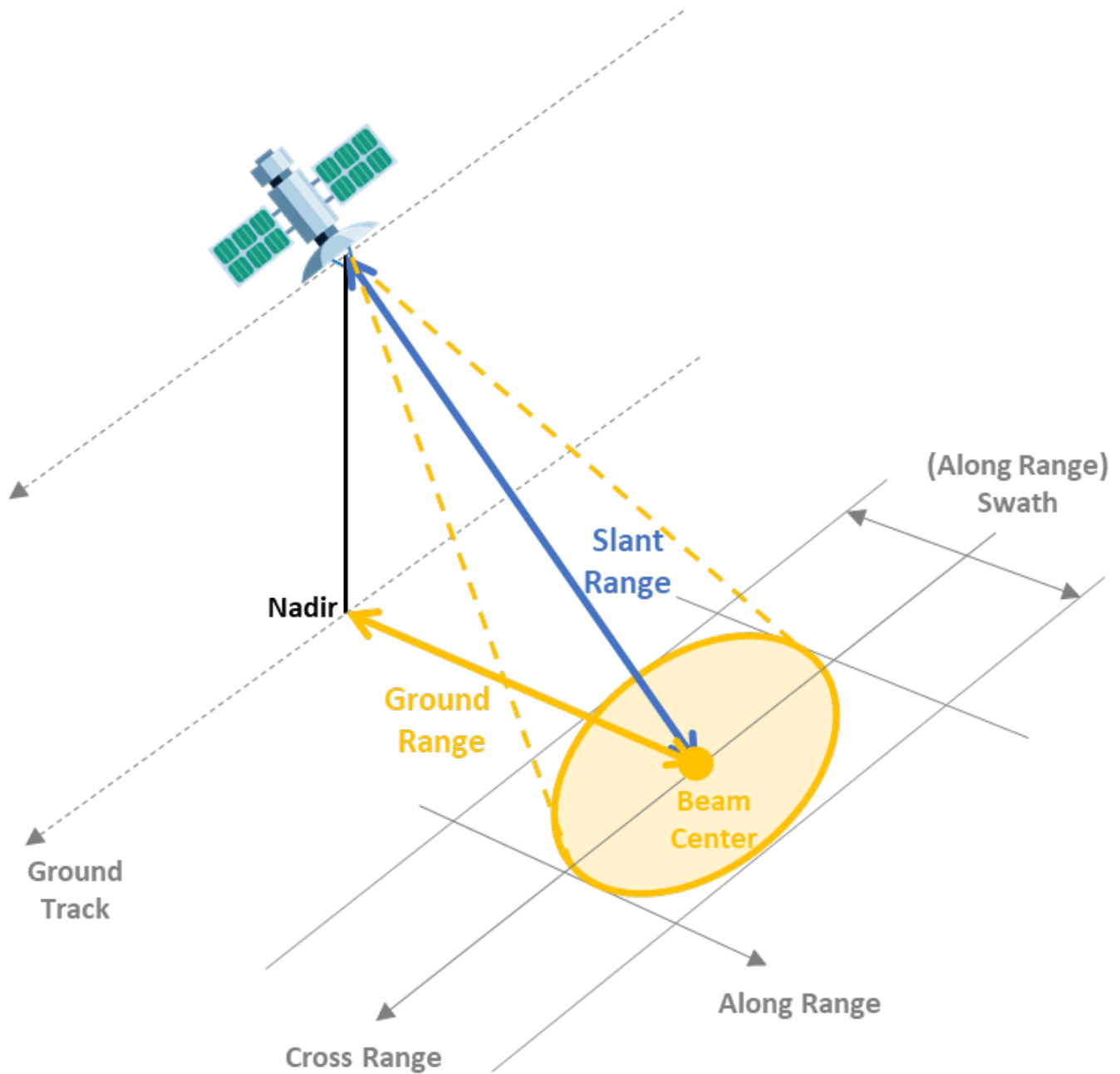
```
% Platform configuration
v = 7e3;                        % Velocity (m/s)
h = 565e3;                      % Altitude (m)

% Radar signal configuration
freq = 5.5e9;                   % Radar frequency (Hz)
lambda = freq2wavelen(freq);    % Wavelength (m)
bw = 500e6;                     % Bandwidth (Hz)
proctime = 0.8;                 % Processing time (s)

% Antenna dimensions
```

```
daz = 5.2;                  % Along azimuth (m)
del = 1.1;                  % Along height (m)
```

Real Antenna Beamwidth and Gain

Use the `ap2beamwidth` function to calculate the real antenna beamwidth.

```
realAntBeamwidth = ap2beamwidth([daz del],lambda) % [Az El] (deg)
```

```
realAntBeamwidth = 2×1

    0.6006
    2.8391
```

Use the `aperture2gain` function to calculate the antenna gain.

```
antGain = aperture2gain(daz*del, lambda) % dBi
```

```
antGain = 43.8369
```

Antenna Orientation

The depression angle is often used to define the antenna pointing direction.
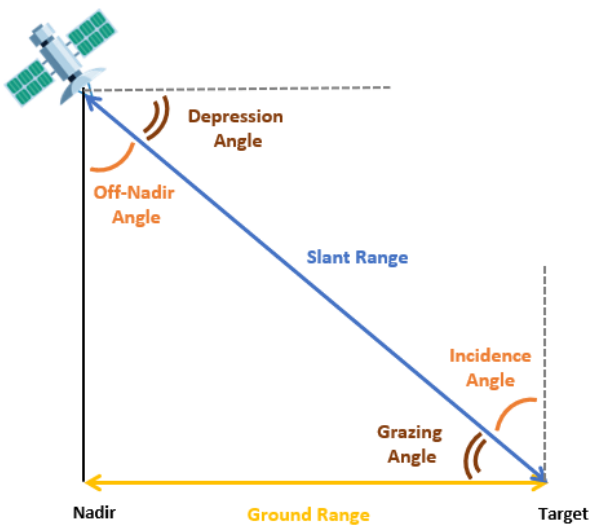
Flat Earth Geometry   **Spherical Earth Geometry**
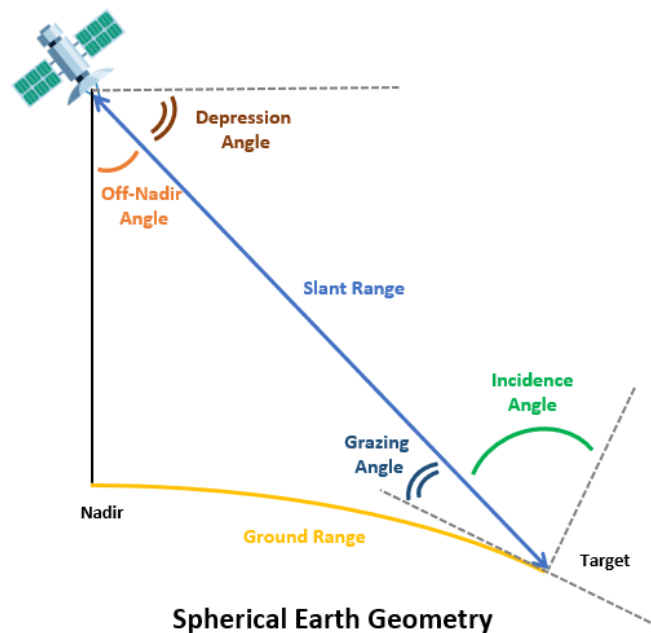
Earth Curvature Effects

A typical assumption for many radar systems is that the Earth is flat so that the depression angle is the same as the grazing angle as shown in the figure.

```
depang_flat = (45:85)';
grazang_flat = depang_flat;
losrng = h./sind(depang_flat);   % Line-of-sight range (m)
```

Use the `depressionang` and `grazingang` functions to calculate the depression and grazing angles respectively from the line-of-sight range. Using a spherical Earth model instead of the flat Earth model, observe that at ranges above 660 km the depression angle correction is greater than the half beamwidth, so it is critical to account for the Earth curvature in this scenario.

```
Rearth = physconst('earthradius');
depang_sph = depressionang(h,losrng,'Curved',Rearth);
grazang_sph = grazingang(h,losrng,'Curved',Rearth);

plot(losrng/1e3,[depang_sph-depang_flat grazang_flat-grazang_sph],'Linewidth',1.5)
grid on;
yline(realAntBeamwidth(2)/2,'--') % Half-beam width (Elevation)
xlabel('Line-of-sight Range (km)')
ylabel('Angle Correction (deg)')
legend('Depression angle','Grazing angle','Half Beamwidth (El)','Location','southeast')
title(['Correction due to Earth Curvature—Radar at ',num2str(h/1e3),' km Altitude'])
```
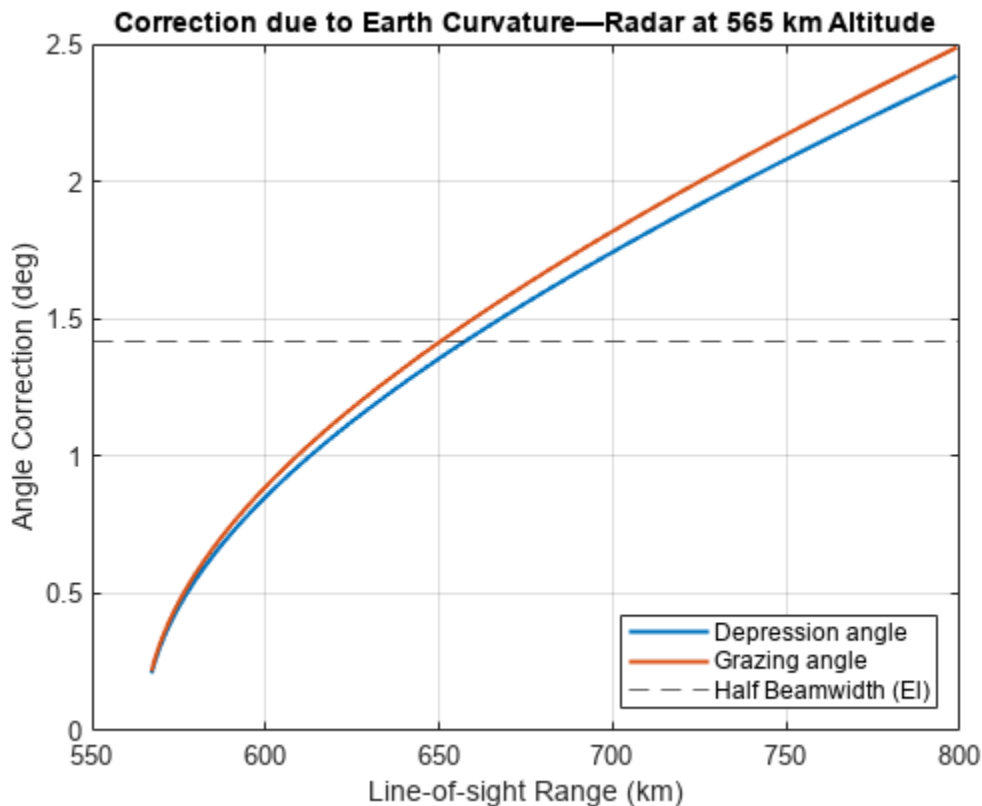

Non-line-of-sight propagation

Radar energy is refracted by the atmosphere and bends towards the Earth, allowing the radar to see beyond the horizon. Use the `effearthradius` function to model the effect of tropospheric refraction using the average radius of curvature method. Since the effective Earth radius is equal to the actual Earth radius, you can conclude that the tropospheric refraction is negligible in this scenario. Ionospheric refraction is ignored in this scenario.

```
tgtHeight = 0;                                  % Smooth Earth
NS = 313;                                       % Reference atmosphere N-Units
Re = effearthradius(min(losrng),h,tgtHeight,'SurfaceRefractivity',NS); % Effective Earth radius
Re/Rearth
```

```
ans = 1
```

For the rest of this example, select a depression angle of 68.96 degrees, which corresponds to a grazing angle of 67 degrees and a slant range of 609.4 km.

```
depang = depang_sph(24)
```

```
depang = 68.9629
```

```
grazang = grazang_sph(24)
```

```
grazang = 66.9953
```

```
slantrng = losrng(24)
```

```
slantrng = 6.0937e+05
```

Footprint and Resolution of Real and Synthetic Aperture Antennas

Radar Footprint

Next, calculate the antenna footprint using the `aperture2swath` function. The footprint is determined by the along range swath (or distance covered in the along range direction) and the cross-range swath (or distance covered in the cross-range direction).

```
[rangeswath, crngswath] = aperture2swath(slantrng,lambda,[del daz],grazang);
['Real antenna range footprint: ', num2str(round(engunits(rangeswath),1)), ' km']
```

```
ans =
'Real antenna range footprint: 32.8 km'
```

```
['Real antenna cross range footprint: ', num2str(round(engunits(crngswath),1)), ' km']
```

```
ans =
'Real antenna cross range footprint: 6.4 km'
```

Calculate the distance from the ground track (or nadir of the satellite) to the center of the radar beam on the ground. Notice that while the Earth curvature has a small effect on the footprint size, it shifts the beam center position by 9.5 km in the along range direction for a given slant range of 609.4 km.

```
distgrndtrack_flat = h/tand(grazang_flat(24)); % Flat Earth Model
['Distance from nadir to center of the footprint (Flat Earth Model): ', num2str(round(engunits(di
```

```
ans =
'Distance from nadir to center of the footprint (Flat Earth Model): 228.3 km'
```

1-287

```
distgrndtrack = Re*deg2rad(depang-grazang);     % Taking Earth curvature into account
['Distance from nadir to center of the footprint: ', num2str(round(engunits(distgrndtrack),1)),
```

```
ans =
'Distance from nadir to center of the footprint: 218.8 km'
```

Footprint Sensitivity to Frequency and Angle Variations

Analyze the antenna footprint sensitivity to frequency variations. Increasing the operating frequency of the radar reduces the antenna footprint.

```
f = [freq/2 3/4*freq freq 4/3*freq]'; % Operating frequencies (Hz) within C-band
lambdav = freq2wavelen(f);            % Wavelengths (m)
[rangeswathv,crngswathv] = aperture2swath(slantrng,lambdav,[del daz],grazang);
clf;
plotAntennaFootprintVsFreq(freq,f,rangeswathv,crngswathv);
```
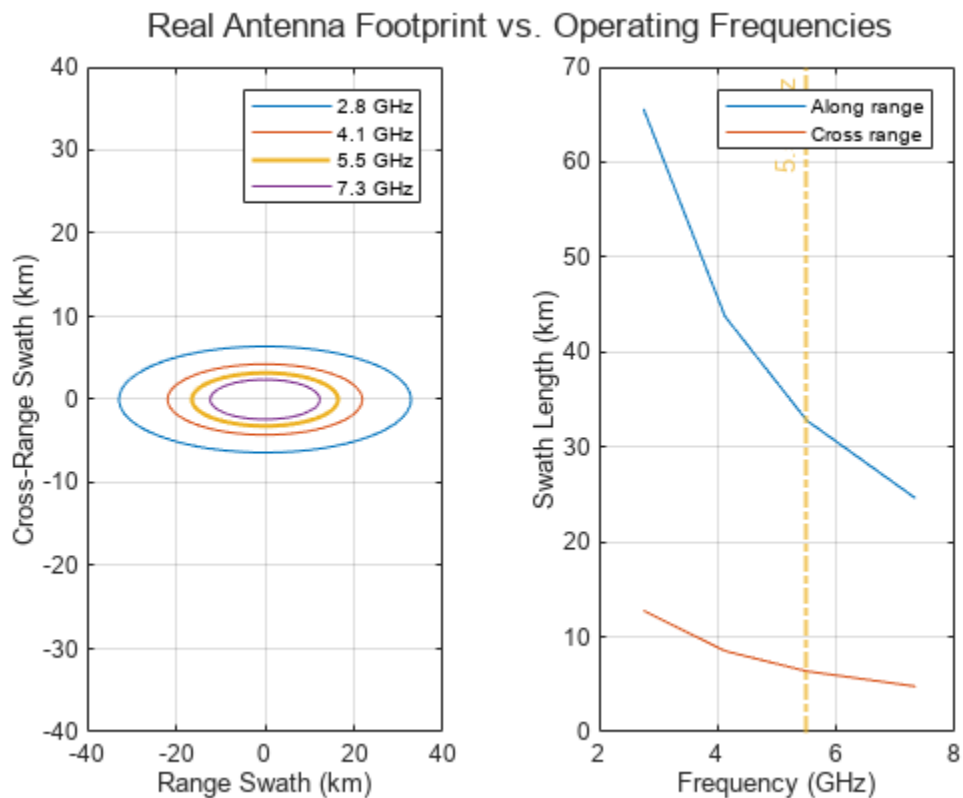

Next, fix the operating frequency back to 5.5 GHz and analyze the antenna footprint sensitivity to grazing angle variations. Plot the distance from the center of the radar footprint to the satellite ground track (the ground range) as a function of look angles. As expected, the beam footprint and the beam center distance to nadir decrease when the grazing angle increases.

```
grazv = grazang_sph(8:8:end);               % Grazing angles (degrees)
depangv = depang_sph(8:8:end);
slantrngv = losrng(8:8:end);                % Slant range to the ground
rangeswathv = zeros(1,numel(grazv));
```

```
crngswathv = zeros(1,numel(grazv));

distgrndtrackv = Re*deg2rad(depangv-grazv);
for i=1:numel(grazv)
    [rangeswathv(i),crngswathv(i)] = aperture2swath(slantrngv(i),lambda,[del daz],grazv(i));
end
clf;
plotAntennaFootprintVsGrazAng(grazv,distgrndtrackv,rangeswathv,crngswathv,grazang);
```
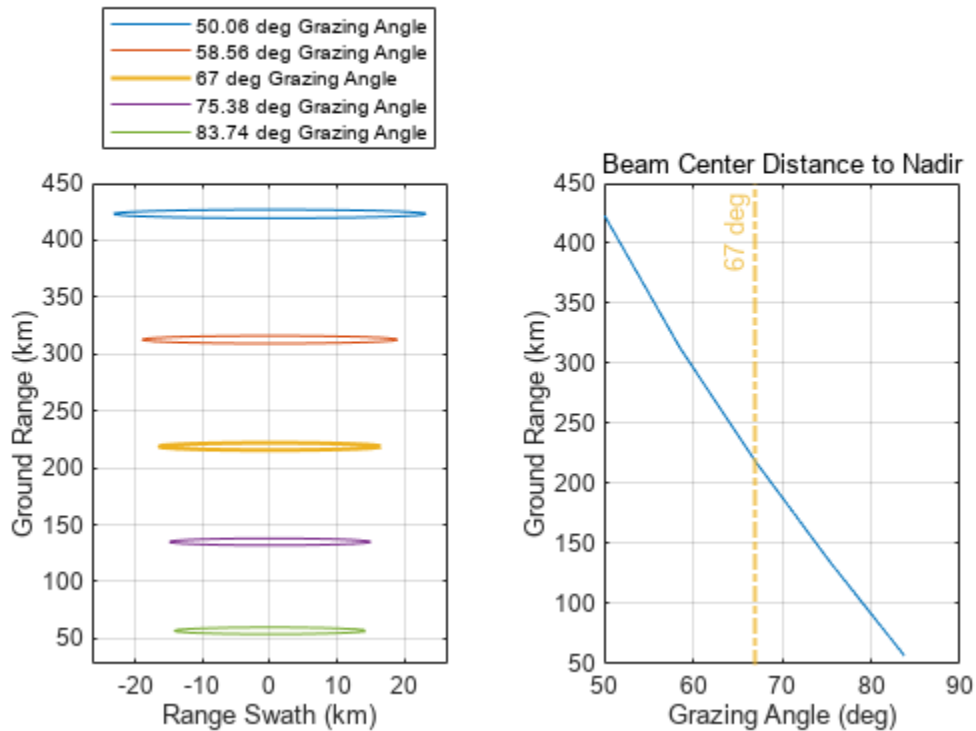

Real Antenna Resolution

The ground range resolution is the distance below which two point targets cannot be separated in the final image. Use the bw2rangeres function to calculate the slant range resolution of the real antenna, which is determined by the signal bandwidth.

```
slantrngres = bw2rangeres(bw)
```

slantrngres = 0.2998
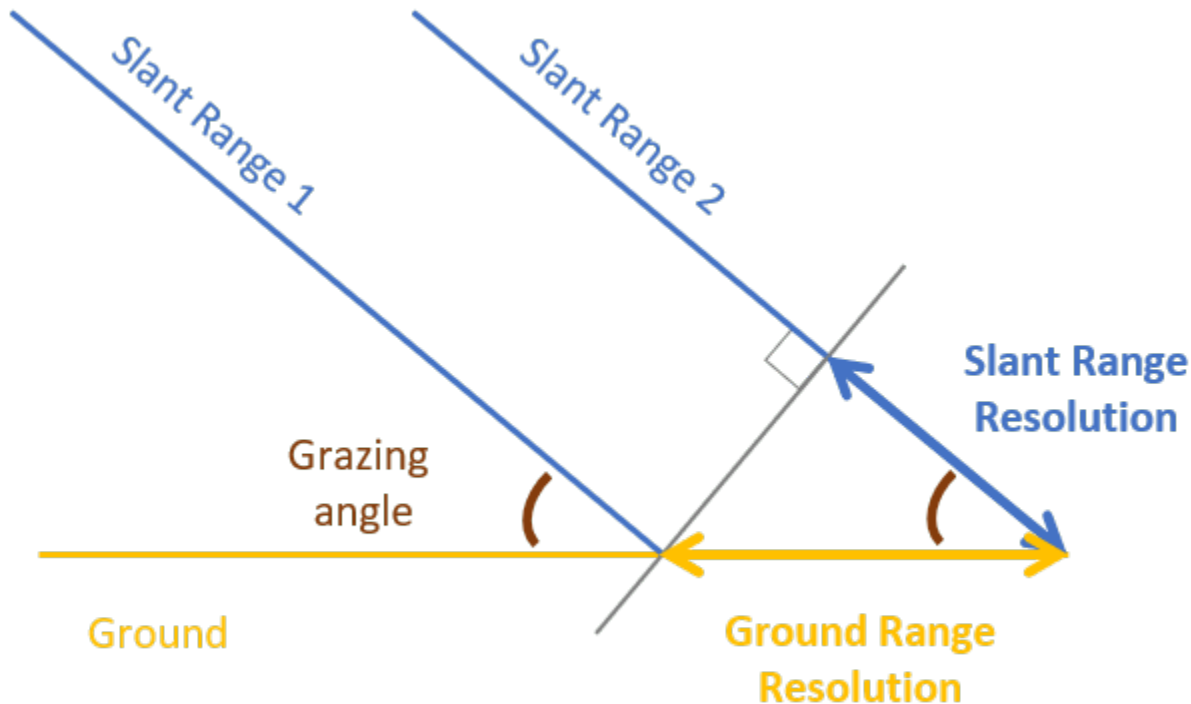
Project the slant range resolution to the ground plane for a given grazing angle. The ground range resolution is always worse than the slant range resolution. The difference between the two increases as the grazing angle increases. Notice that the cross-range resolution is the same as the cross-range footprint since no signal processing is performed to enhance the resolution. This cross-range resolution is clearly insufficient.

```matlab
rngres   = slant2grndrangeres(slantrngres,grazang);
disp(['Real antenna ground range resolution: ', num2str(round(rngres,2)), ' m'])
```

```
Real antenna ground range resolution: 0.77 m
```

```matlab
crngres = crngswath;
disp(['Real antenna cross-range resolution: ', num2str(round(engunits(crngres),1)), ' km'])
```

```
Real antenna cross-range resolution: 6.4 km
```

Next, analyze the ground range resolution sensitivity to variations of the grazing angle for a few signal bandwidths. The best range resolution is achieved with a high signal bandwidth and a low grazing angle. For a given bandwidth, the range resolution worsens as the grazing angle increases. At 500 MHz, the range resolution varies from 42 cm at a grazing angle of 45 degrees to 1.73 m at a grazing angle of 80 degrees.

```matlab
bwv = [100e6 200e6 500e6 1e9]; % Signal bandwidths
rngresv = zeros(numel(grazang_sph),numel(bwv));
for j=1:numel(grazang_sph)
    slantrngresv = bw2rangeres(bwv);
    rngresv(j,:) = slant2grndrangeres(slantrngresv,grazang_sph(j));
end
clf;
l = semilogy(grazang_sph,rngresv);
```

```
set(l(3),'LineWidth',2)
grid on
xline(grazang,'-.',{[num2str(round(grazang,2)),' deg']}); % Selected grazing angle
xlabel('Grazing Angle (deg)')
ylabel('Ground Range Resolution (m)')
title('Real Antenna Range Resolution vs. Grazing Angles')
legend('100 MHz BW', '200 MHz BW', '500 MHz BW', '1.0 GHz BW','Location','southeast')
```
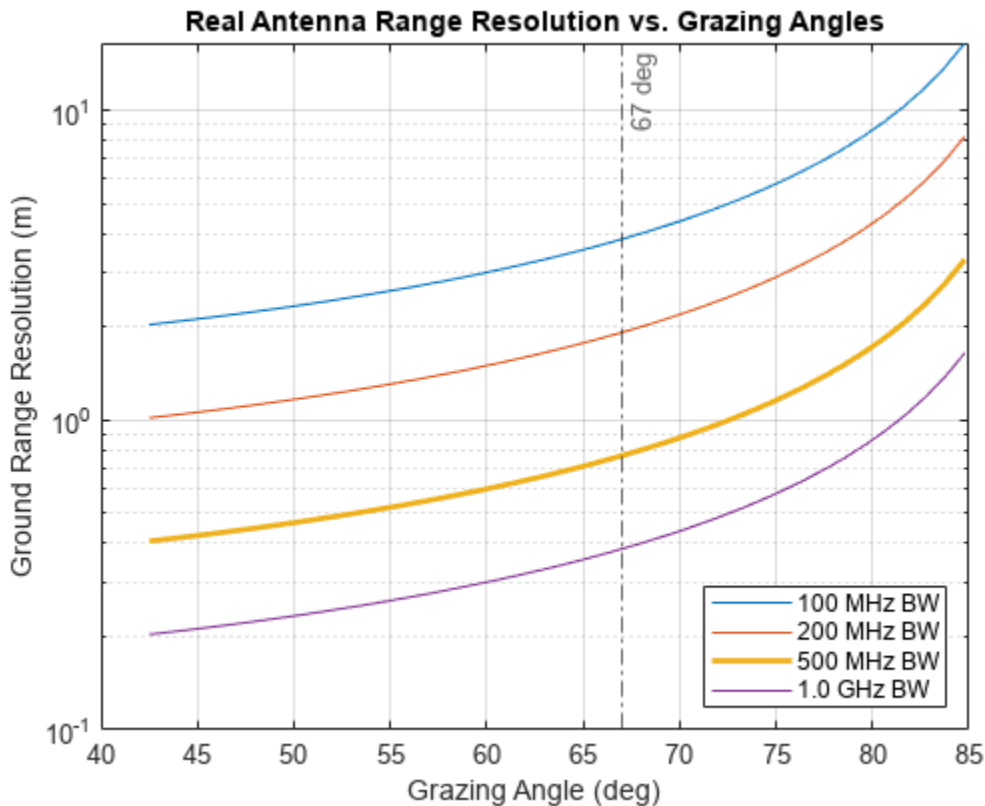

Ideal vs Effective Resolution of the Synthetic Antenna

The previous section covered the range and cross-range resolution of a real aperture radar, the next step is to use the `sarlen` and `sarazres` functions to calculate the synthetic aperture length and its cross-range (or azimuth) resolution. Notice that the range resolution, which is dependent on the transmit signal bandwidth and the grazing angle, remains the same for a real aperture radar and a SAR.

```
idealSyntheticApertureLength = sarlen(slantrng,lambda,daz);              % Synthetic aperture leng
azResolution = sarazres(lambda,slantrng,idealSyntheticApertureLength);  % Cross-range (azimuth)
integrationTime = sarinttime(idealSyntheticApertureLength,v);           % Ideal integration time

Parameters = ["Synthetic Aperture Length";"Range Resolution";"Cross Range Resolution";"Integratic
IdealSAR = [round(idealSyntheticApertureLength/1e3,1);rngres;round(azResolution,1);round(integra
Units = ["km";"m";"m";"ms"];
idealSAR = table(Parameters,IdealSAR,Units)
```

```
idealSAR=4×3 table
         Parameters               IdealSAR    Units

    _____       _____    _____

    "Synthetic Aperture Length"      6.4       "km"
    "Range Resolution"             0.76711     "m"
    "Cross Range Resolution"         2.6       "m"
    "Integration Time"             1095.9      "ms"
```

The best cross-range resolution to use SAR in this scenario is 2.6 m. That is a considerable improvement compared to the 6.4 km cross-range resolution of the real antenna. However, to achieve this performance, pulses need to be integrated for over 1.1 s. The system you analyze in this example specifies an integration time of only 800 ms. This affects the effective cross-range resolution of the radar.

```
effSyntheticApertureLength = sarlen(v,proctime);                         % Take processing time
effAzResolution = sarazres(lambda,slantrng,effSyntheticApertureLength);  % Effective azimuth res

RealAntenna = [NaN; rngres; round(crngres); NaN];
EffectiveSAR = [round(effSyntheticApertureLength/1e3,1);rngres;round(effAzResolution,1);round(pr

sar = table(Parameters,RealAntenna,IdealSAR,EffectiveSAR,Units)
```

```
sar=4×5 table
         Parameters               RealAntenna    IdealSAR    EffectiveSAR    Units

    _____       _____    _____    _____    _____

    "Synthetic Aperture Length"       NaN           6.4          5.6         "km"
    "Range Resolution"              0.76711       0.76711      0.76711       "m"
    "Cross Range Resolution"          6388          2.6           3          "m"
    "Integration Time"                NaN         1095.9         800         "ms"
```

By integrating pulses for only 800 ms, the synthetic aperture length is reduced by 800 m compared to the ideal case, and the cross-range resolution is reduced by 0.4 m to 3.0 m. Because data is processed in less than 1.11 s, the radar can keep the target within the beam footprint for the length of the computation.

Next, analyze the cross-range resolution sensitivity to frequency variations. The cross-range resolution improves at higher frequencies.

```
azresv = sarazres(lambdav,slantrng,idealSyntheticApertureLength);
effazresv = sarazres(lambdav,slantrng,effSyntheticApertureLength);
plot([f f]/1e9,[azresv(:) effazresv(:)])
grid on
legend('Ideal SAR','Effective SAR')
xlabel('Frequency (GHz)')
ylabel('Cross-range Resolution (m)')
title('Cross Range Resolution vs. Operating Frequency')
```

Cross Range Resolution vs. Operating Frequency
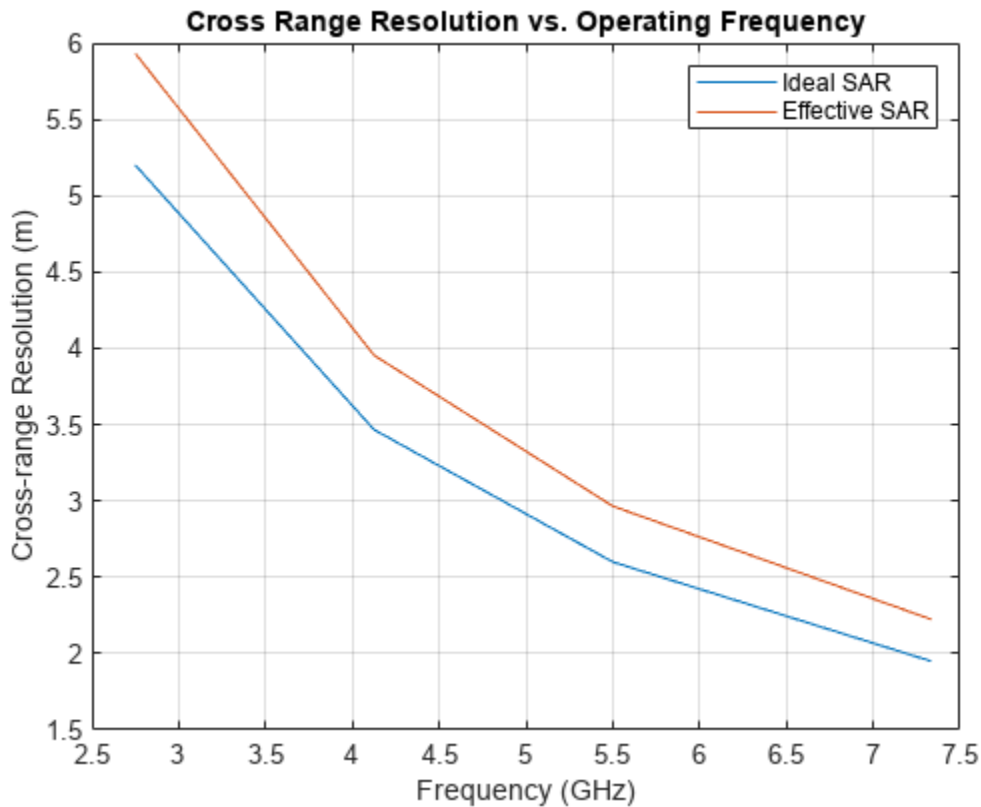
Range and Cross-Range Ambiguities in SAR Images

Coverage (swath length) and resolution cannot be chosen independently as they both impose constraints on the pulse repetition frequency (PRF). If the PRF is set too low, the radar suffers from grating lobes and Doppler (or cross-range) ambiguities. If the PRF is set too high, range measurements are ambiguous. The grazing angle also affects the PRF selection as it impacts the antenna footprint as seen in a previous section.

Upper and Lower PRF Bounds

Use the `sarprfbounds` function to determine the minimum and maximum PRF values for various range coverages (footprint) and cross-range resolutions given the satellite velocity and the grazing angle.

```
desiredCRngRes = (0.5:0.2:5);      % m
desiredRngCov = (16e3:5e3:100e3); % m
[prfminv, prfmaxv] = sarprfbounds(v,desiredCRngRes,desiredRngCov,grazang);
clf
plotPRFbounds(prfminv,desiredCRngRes,prfmaxv,desiredRngCov);
```
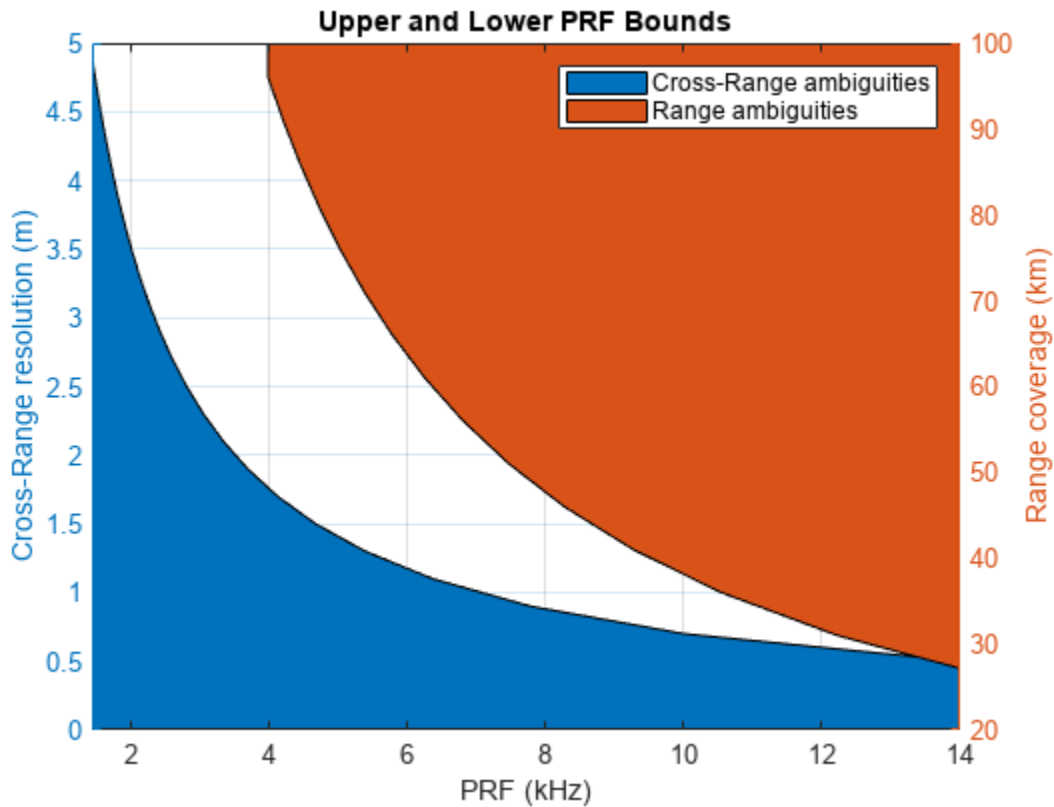
PRF Selection to Avoid Ghost Images

The PRF is typically programmable and can be optimized for each mission. Use the `sarprf` function to calculate the PRF of the radar based on the satellite velocity and the real antenna dimension along azimuth. Specify a constant roll-off factor as a safety margin to prevent mainlobe returns from aliasing in the PRF interval.

```
prf = sarprf(v,daz,'RollOff',1.5)
```

```
prf = 4.0385e+03
```

The selected PRF is within the PRF bounds. The movement of the SAR platform within pulse repetition interval (PRI) is 1.73 m.

```
t = 1/prf;
distpri = sarlen(v,t) % Distance moved by satellite in one PRI
```

```
distpri = 1.7333
```

Grating Lobes Analysis

Now model the synthetic array with the `phased.ula` System object using the effective aperture length and the element spacing.

```
sarArray = phased.ULA('NumElements',ceil(effSyntheticApertureLength/distpri)+1,'ElementSpacing',d
distpri/lambda
```

ans = 31.7998

Notice that the element spacing is over 31.8 times wavelength so the radar has grating lobes. Recall that the real antenna azimuth beamwidth is 0.6 degrees. Model the antenna response as a highly directional cosine pattern with the same beamwidth.

```
realAnt = phased.CosineAntennaElement('FrequencyRange',[freq-bw/2 freq+bw/2],'CosinePower',26e3)
realAntBeamwidth(1)
```

ans = 0.6006

```
b = beamwidth(realAnt,freq)
```

b = 0.6000

Plot the pattern response of both the synthetic array and the real antenna to verify that the first grating lobe of the array is located in the null of the real antenna pattern, so the radar does not suffer negative impact from the grating lobes.

```
clf
plotResponse(sarArray,freq,physconst('LightSpeed'),'RespCut','Az','AzimuthAngles',(-3:0.005:3));
hold on
plotResponse(realAnt,freq,'RespCut','Az','AzimuthAngles',(-3:0.005:3));
hold off
legend('Synthetic array response','Real antenna response','Location','northwest')
title('')
% Annotations
xl = xline(1.8,'-.',{'First grating lobe'});
xl.LabelVerticalAlignment = 'top';
xl.LabelHorizontalAlignment = 'left';
```
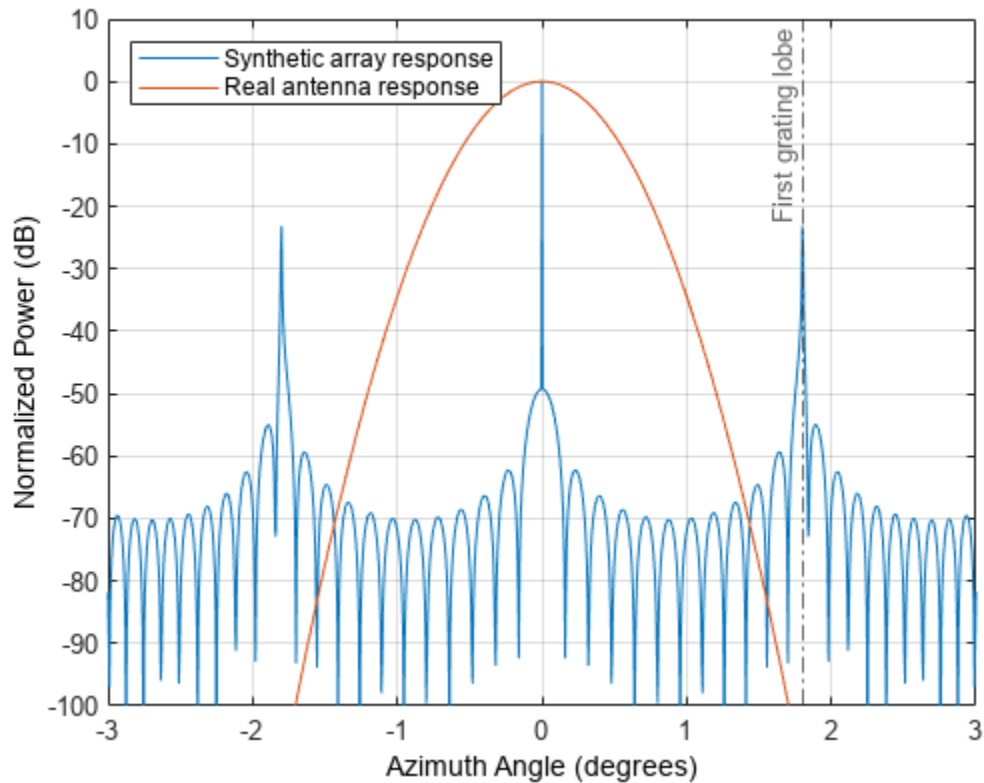
SAR Image SNR and Noise Equivalent Reflectivity

The next section examines different factors used in the SAR equation to calculate the image SNR. First, analyze the target (or surface) radar cross section (RCS) and the rain (or volume clutter) RCS.

Surface Roughness vs Frequency, Polarization and Grazing Angle

Use the `landreflectivity` function to calculate the reflectivity or the normalized radar cross section (NRCS) for a given grazing angle and operating frequency. The target RCS in the ground image plane is calculated using the `sarSurfaceRCS function` and taking the radar resolution into account. In general, the brightness of a SAR image area corresponds to the roughness of the surface so mountains appear brighter than flatland due to their higher RCS. Roughness is measured against wavelength so a surface appears rough at one frequency and smooth at another. Also, notice that the RCS increases as the grazing angle increases.

```
fsub = [2.5e9 5.5e9 7.5e9];    % Hz
grazv = 20:2:60;               % degrees
landtype = ["Mountains","Flatland"];
tgtrcsv = zeros(numel(grazv),numel(fsub),numel(landtype));
for k=1:numel(landtype)        % Land types with different roughness
    for j=1:numel(fsub)        % Frequencies
        for u =1:numel(grazv)  % Grazing angles
            nrcsv = landreflectivity(landtype(k),grazv(u),fsub(j));
            tgtrcsv(u,j,k) = sarSurfaceRCS(nrcsv,[slantrngres effAzResolution],grazv(u));
        end
    end
end
```

```
end
plotTargetRCSvsGrazAng(grazv,tgtrcsv,landtype,fsub)
```
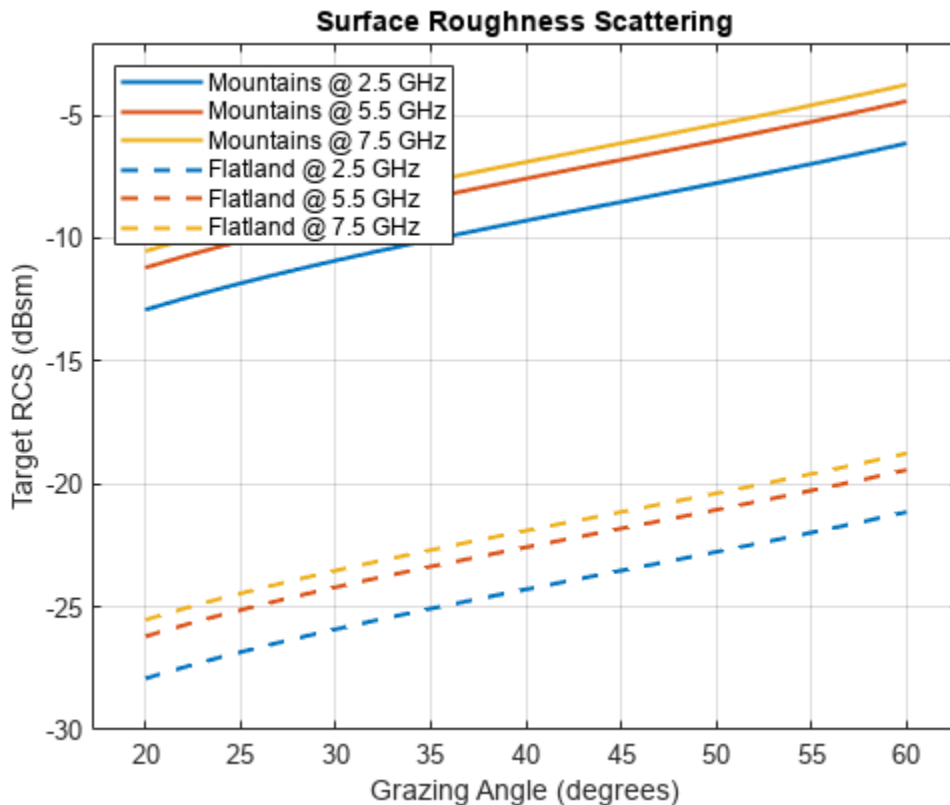

RCS also depends on the polarization of the radar. Use the `seareflectivity` function to analyze the polarization effect on the sea surface reflectivity for different sea surface roughness (that is sea states). Note that there is much more variation in the sea RCS at low grazing angles and that the RCS for horizontally polarized signals is lower than vertically polarized signals for the same sea state for grazing angles below 60 degrees. This figure also shows that the difference between polarizations decreases as the grazing angle increases from 10 to 60 degrees.

```
pol = ["H","V"];
seaStates = [1 3 5];
grazvpol = linspace(0.1,60,100); % Grazing angle (deg)
tgtrcsv = zeros(numel(grazvpol),numel(pol),numel(seaStates));
for n=1:numel(seaStates)          % Sea states
    for j=1:numel(pol)            % Polarizations
        for u =1:numel(grazvpol) % Grazing angles
            nrcsv = seareflectivity(seaStates(n),grazvpol(u),freq,'Polarization',pol(j)); % Calcu
            tgtrcsv(u,j,n) = sarSurfaceRCS(nrcsv,[slantrngres effAzResolution],grazvpol(u));
        end
    end
end
plotTargetRCSvsPol(grazvpol,tgtrcsv,seaStates,pol,freq)
```

Surface Roughness Scattering @ 5.5 GHz vs. Polarization
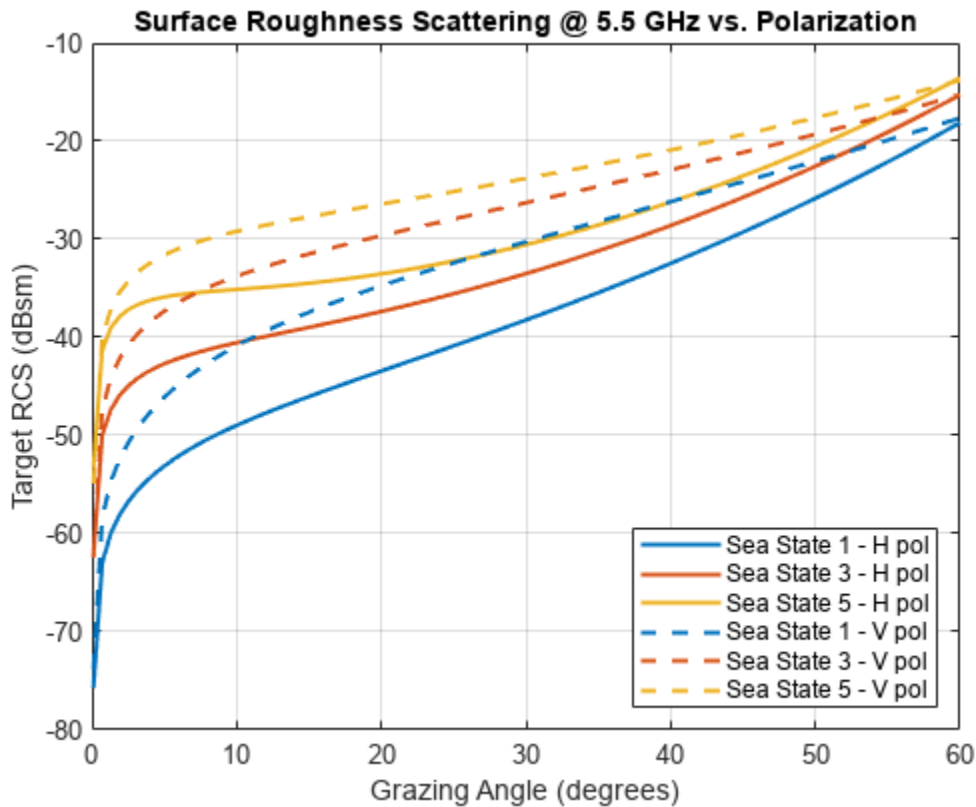
Rain Reflectivity and Signal-to-Clutter Ratio (SCR)

Competing echoes from undesired sources such as rain can cause degradation in SAR images. Rain falling in the vicinity of a target scene clutters the image of that scene. Use the `rainreflectivity` function to analyze rain reflectivity under varying rain rates and polarizations. Observe that the rain reflectivity is higher for linear polarizations than circular polarization and that it increases with the rain rate.

```
rainRate = [0.25 1 2 4 8 16];
pol = ["HH","HV","RCPRCP"];
rainrefl = zeros(numel(rainRate),numel(pol));
for i=1:numel(pol)
    rainrefl(:,i) = rainreflectivity(freq,rainRate,pol(i));  % 5.5 GHz operating frequency
end
plot(rainRate,pow2db(rainrefl),'LineWidth',1.5)
grid on
xlabel('Rain rate (mm/Hr)')
ylabel('Rain Reflectivity (dB/m)')
title('Polarization Effects on Rain Reflectivity')
legend('HH Polarization','HV Polarization','Circular Polarization','Location','southeast');
```

Polarization Effects on Rain Reflectivity
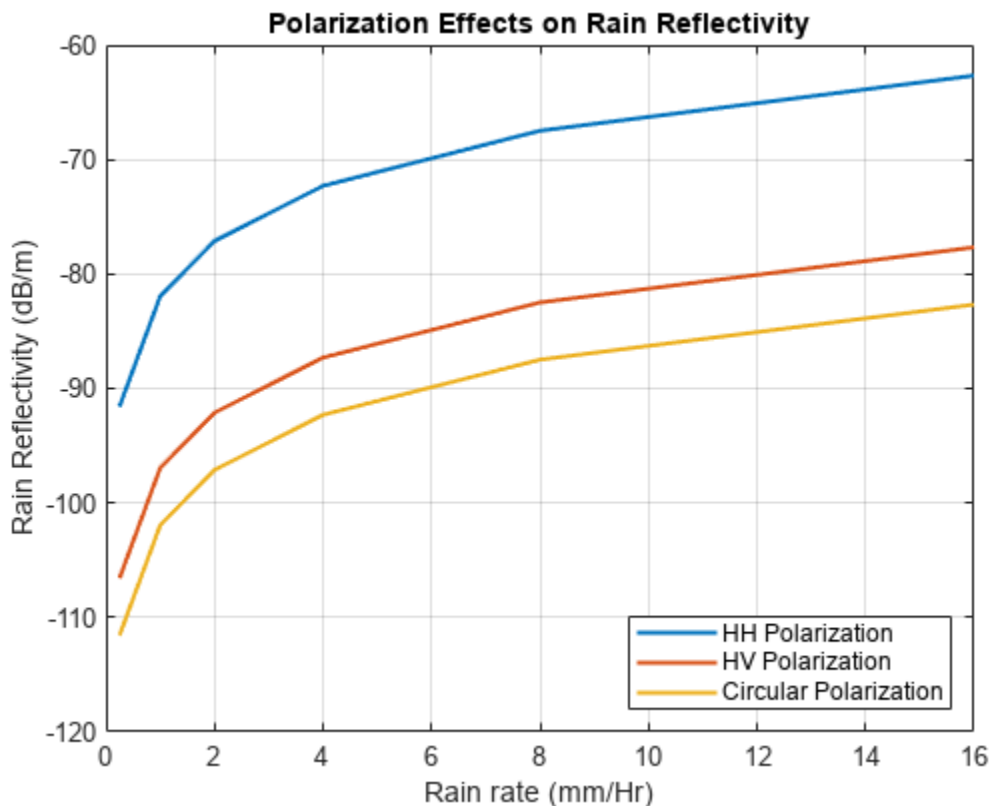
Next, use the `clutterVolumeRCS` function to calculate the rain RCS, which is defined by the radar 3-D resolution cell (slant range, azimuth, and elevation resolution). Use the `rainscr` function to calculate the ratio of the signal energy from a resolution cell of the target scene to the energy from the rain returns processed into the same resolution cell of the target scene (or rain SCR). Verify that the signal-to-clutter ratio remains above 20 dB for all rain rates and for all targets including the ones with a weak RCS such as smooth land. Therefore, this example does not expect rain to be a limiting factor for this radar in this configuration.

```
elbeamw = realAntBeamwidth(2);                            % Radar elevation beamwidth
maxRainHeight = 4000;                                     % m
rhoe = rainelres(slantrng,elbeamw,grazang,maxRainHeight); % Elevation resolution
res = [slantrngres effAzResolution rhoe];
rrcs = clutterVolumeRCS(rainrefl(:,1),res);               % Rain RCS for HH polarization

% The grazing angle of the geometry is slightly outside the default Barton
% land reflectivity model, so temporarily turn off the reflectivity
% warnings for the following calculation.
warning('off','radar:radar:outsideValidityRegion');

% Calculate SCR
landType = ["Wooded Hills","Farm","Smooth"];
scr = zeros(numel(rainRate),numel(landType));
tgtrcs = zeros(numel(landType),1);
for j=1:numel(landType)
    nrcsv(j) = landreflectivity(landType(j),grazang,freq); % Calculate normalized RCS
    tgtrcs(j) = sarSurfaceRCS(nrcsv(j),[slantrngres effAzResolution],grazang);
    scr(:,j) = rainscr(lambda,rrcs,tgtrcs(j),proctime)';   % Signal to rain clutter ratio
```

```
end
```

```
% Turn reflectivity model warnings back on
warning('on','radar:radar:outsideValidityRegion');
```

```
plotRainSCR(rainRate,rrcs,scr,landType)
```
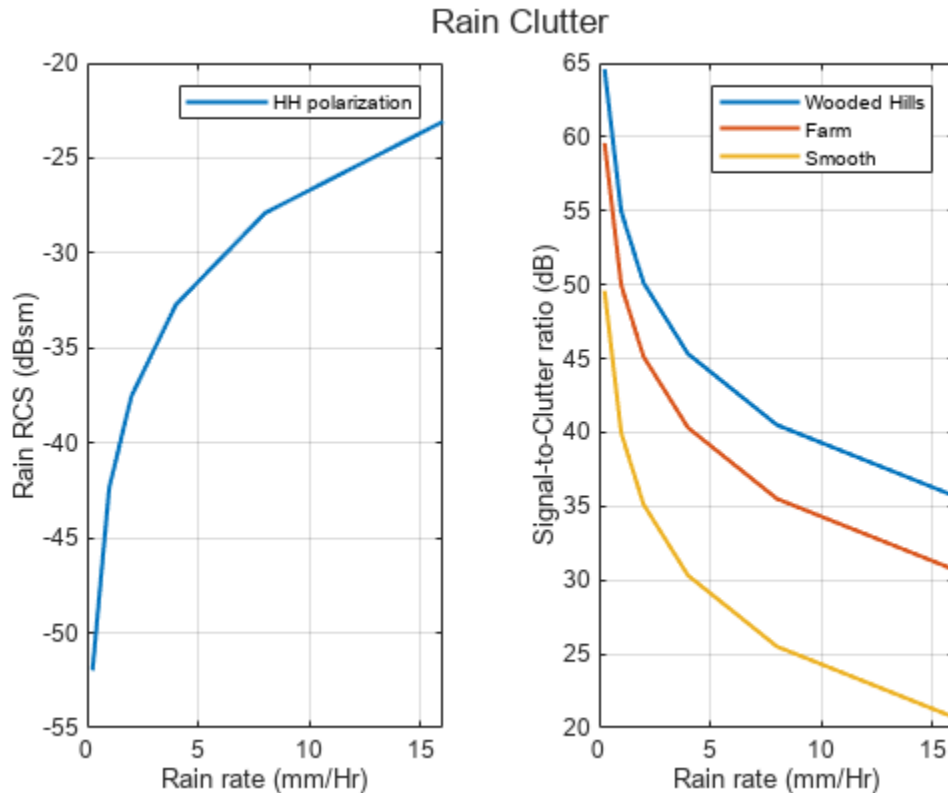

SAR Equation

Estimate the signal-to-noise ratio (SNR) available at the radar receiver using the SNR form of the radar equation. First, model and estimate the different sources of gain and loss for the radar system and its environment.

Processing Gains

Use the `matchinggain` function to calculate the range processing gain due to the noise bandwidth reduction after the matched filter.

```
d = 0.1;                    % 10 percent duty cycle
pw = (1/prf)*d;             % Effective pulse width (s)
rnggain = matchinggain(pw,bw)  % Range processing gain (dB)
```

```
rnggain = 40.9275
```

Use the `sarazgain` function to calculate the azimuth processing gain due to the coherent integration of pulses.

```
azgain = sarazgain(slantrng,lambda,v,effAzResolution,prf) % Azimuth processing gain (dB)
```

```
azgain = 35.0931
```

Losses and Noise Factor

Use the `noisefigure` function to estimate the noise figure of the cascaded receiver stages. Assume seven stages with the following values:

- Stage 1 LNA: Noise Figure = 1.0 dB, Gain = 15.0
- Stage 2 RF Filter: Noise Figure = 0.5 dB, Gain = -0.5
- Stage 3 Mixer: Noise Figure = 5.0 dB, Gain = -7.0
- Stage 4 IF Filter: Noise Figure = 1.0 dB, Gain = -1.0
- Stage 5 IF Preamplifier: Noise Figure = 0.6 dB, Gain = 15.0
- Stage 6 IF Stages: Noise Figure = 1.0 dB, Gain = 20.0
- Stage 7 Phase Detectors: Noise Figure = 6.0 dB, Gain = -5.0

```
nf  = [1.0, 0.5, 5.0, 1.0, 0.6, 1.0, 6.0];       % dB
g   = [15.0, -0.5, -7.0, -1.0, 15.0, 20.0, -5.0]; % dB
cnf = noisefigure(nf, g)
```

```
cnf = 1.5252
```

Use the `radarpropfactor` function to calculate the one-way radar propagation factor over smooth land.

```
[hgtsd, beta0, vegType] = landroughness('Smooth');
tgtheight = hgtsd;
Re = effearthradius(slantrng,h,tgtheight);
propf = radarpropfactor(slantrng,freq,h,tgtheight,'EffectiveEarthradius',Re,'TiltAngle',grazang,
    'SurfaceHeightStandardDeviation',hgtsd,'SurfaceSlope',beta0,'VegetationType',vegType)
```

```
propf = -5.3980e-05
```

Use the `tropopl` function to calculate losses due to atmospheric gaseous absorption.

```
atmoLoss = tropopl(slantrng,freq,tgtheight,grazang)
```

```
atmoLoss = 0.0437
```

Use the `rainpl` function to calculate losses due to rain according to the ITU model. Alternatively, you can use the `cranerainpl` function to make calculations according to the Crane model.

```
rainLoss = rainpl(slantrng,freq,rainRate(end),grazang)
```

```
rainLoss = 0.5389
```

Use the `radareqsarsnr` function to calculate the image SNR with the SAR radar equation. Assume a 5 kW peak power. You can also specify additional losses and factors including the azimuth beam shape loss, window loss, transmission loss, and receive line loss. Estimate the beam shape loss with the `beamloss` function and use 5 dB for all other fixed losses combined. For this analysis, specify `landType` as "Smooth" to use the weakest land target. A minimum image SNR of 10 dB is usually

regarded as sufficient to provide a good image quality, so at 35.9 dB for this target the system has ample margins.

```matlab
Lb = beamloss;
customLoss = 5;          % dB
Pt = 5e3;                % Peak power (W)
sntemp = systemp(cnf); % Noise Temperature
imgsnr = radareqsarsnr(slantrng,lambda,Pt,pw,rnggain,azgain,'Gain',antGain,'RCS',tgtrcs(3),...
    'AtmosphericLoss',atmoLoss,'Loss',cnf+rainLoss,'PropagationFactor',propf,...
    'Ts',sntemp,'CustomFactor',-Lb-customLoss)
```

```
imgsnr = 35.9558
```

Noise Equivalent Reflectivity (NER or NEZ0)

Finally, use the `sarnoiserefl` function to calculate the NER of the radar and analyze its sensitivity to frequency variations. The NER is the smallest distributed clutter that can be seen in the presence of receiver noise for a given surface reflectivity. It is a measure of the sensitivity of the radar to spatially distributed noise. For the smooth terrain in this calculation, the NER is –62.6 dB at 5.5.GHz, and it increases with frequency.

```matlab
neq = sarnoiserefl(f,freq,imgsnr,nrcsv(3));
clf;
plot(f/1e9,neq,'LineWidth',1.5)
grid on
xline(5.5,'--')
xlabel('Frequency (GHz)')
ylabel('Noise Equivalent Reflectivity (dB)')
title(['Smooth Terrain—', num2str(round(effAzResolution,1)), ' m Resolution'])
```
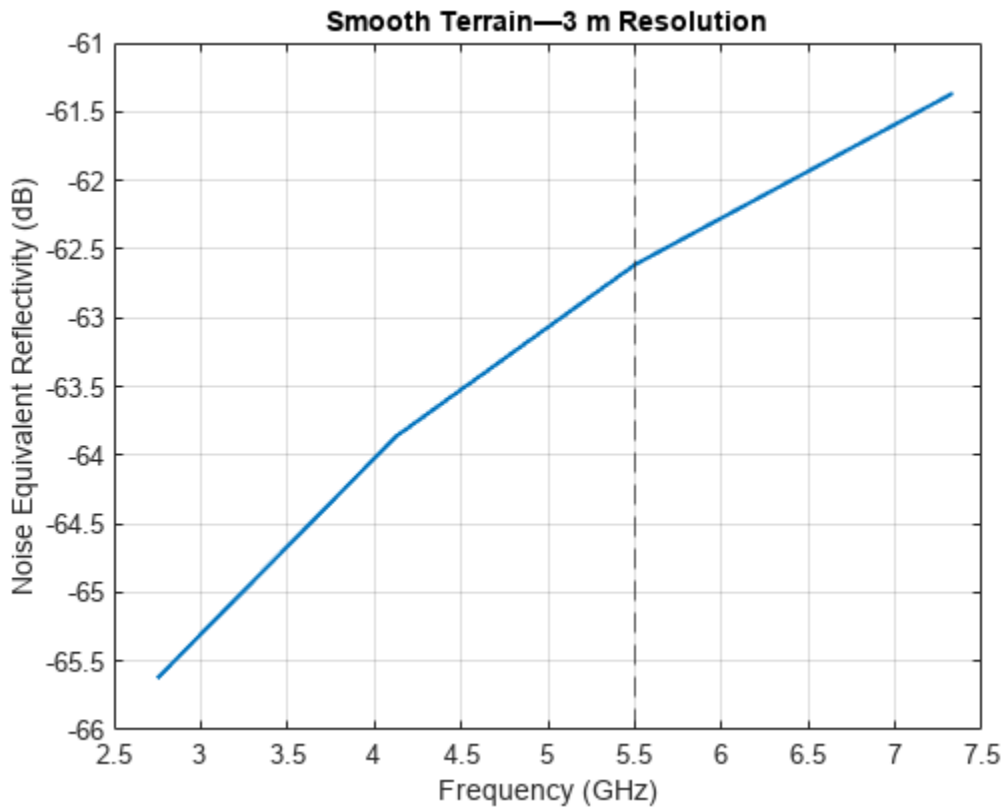
Summary

This example shows how to estimate performance parameters such as coverage, resolution and SNR for a spaceborne SAR system. First, you determine the upper and lower PRF bounds to avoid ghost images. Then you analyze target and rain clutter RCS for different polarizations and grazing angles. Then estimate processing gains and losses in the radar and its environment. Finally, you use the SAR equation to calculate the image SNR and the NER.

References

1 Doerry, Armin Walter. "Performance Limits for Synthetic Aperture Radar." Sandia National Laboratories, February 1, 2006.

2 O'Donnell, Robert. "Radar Systems Engineering ." IEEE Aerospace & Electronic Systems Society, and IEEE New Hampshire Section, 2013.

Supporting Functions

slant2grndrangeres

```
function grndres = slant2grndrangeres(slres, grazang)
% slant2grndrangeres Convert slant range resolution to ground range resolution
grndres = slres./cosd(grazang);
end
```

plotAntennaFootprintVsFreq

```matlab
function t = plotAntennaFootprintVsFreq(freq,f,rangeswathv,crngswathv)
t = tiledlayout(1,2);
nexttile
% Plot cross-range vs. range
theta_grid = linspace(0,2*pi)';
semix = 0.5*rangeswathv/1e3; % km
semiy = 0.5*crngswathv/1e3;  % km
l = plot(cos(theta_grid)*semix,sin(theta_grid)*semiy);
set(l(3),'LineWidth',1.5)
ylim(xlim)
grid on
xlabel('Range Swath (km)')
ylabel('Cross-Range Swath (km)')
legend([num2str(round(f(1)/1e9,1)) ' GHz'],[num2str(round(f(2)/1e9,1)) ' GHz'], ...
    [num2str(round(f(3)/1e9,1)) ' GHz'],[num2str(round(f(4)/1e9,1)) ' GHz'])

nexttile
% Plot Swath length vs. operating frequencies
plot([f f]/1e9,[rangeswathv(:) crngswathv(:)]/1e3)
xl = xline(freq/1e9,'-.',{[num2str(freq/1e9),' GHz']},'Color',l(3).Color); % Annotation
xl.LabelVerticalAlignment = 'top';
xl.LabelHorizontalAlignment = 'left';
xl.LineWidth = 1.5;
grid on
xlabel('Frequency (GHz)')
ylabel('Swath Length (km)')
legend('Along range','Cross range')

title(t,'Real Antenna Footprint vs. Operating Frequencies')
end
```

plotAntennaFootprintVsGrazAng

```matlab
function t = plotAntennaFootprintVsGrazAng(grazv,distgrndtrackv,rangeswathv,crngswathv,grazang)

t = tiledlayout(1,2);
nexttile
% Plot footprint and beam center distance to ground track
theta_grid = linspace(0,2*pi)';
semix = 0.5*rangeswathv/1e3; % km
semiy = 0.5*crngswathv/1e3;  % km
l = plot(cos(theta_grid)*semix,1e-3*distgrndtrackv(:)'+sin(theta_grid)*semiy);
set(l(3),'LineWidth',1.5)
grid on
xlabel('Range Swath (km)')
ylabel('Ground Range (km)')

legend([num2str(round(grazv(1),2)) ' deg Grazing Angle'], ...
    [num2str(round(grazv(2),2)) ' deg Grazing Angle'], ...
    [num2str(round(grazv(3),2)) ' deg Grazing Angle'], ...
    [num2str(round(grazv(4),2)) ' deg Grazing Angle'], ...
    [num2str(round(grazv(5),2)) ' deg Grazing Angle'], ...
    'Location', "northoutside")
axis padded

nexttile
```

```matlab
% Plot beam center distance to ground track vs. grazing angles
plot(grazv,distgrndtrackv/1e3)
xl = xline(grazang,'-.',{[num2str(round(grazang,2)),' deg']},'Color',l(3).Color); % Annotation
xl.LabelVerticalAlignment = 'top';
xl.LabelHorizontalAlignment = 'left';
xl.LineWidth = 1.5;
grid on
xlabel('Grazing Angle (deg)')
ylabel('Ground Range (km)')
subtitle('Beam Center Distance to Nadir')

title(t,'Real Antenna Footprint vs. Grazing Angles')
end
```

plotPRFbounds

```matlab
function plotPRFbounds(prfminv,desiredCRngRes,prfmaxv,desiredRngCov)
yyaxis left
fill([prfminv(1)/1e3;prfminv(:)/1e3;prfminv(end)/1e3],[0;desiredCRngRes(:);0],[0 0.4470 0.7410])
grid on
xlabel('PRF (kHz)')
ylabel('Cross-Range resolution (m)')
yyaxis right
fill([prfmaxv(1)/1e3;prfmaxv(:)/1e3;prfmaxv(end)/1e3],[100;desiredRngCov(:)/1e3;100],[0.8500 0.32
xlim([prfminv(end)/1e3 prfminv(1)/1e3])
ylabel('Range coverage (km)')
legend('Cross-Range ambiguities','Range ambiguities')
title('Upper and Lower PRF Bounds')
end
```

plotTargetRCSvsGrazAng

```matlab
function plotTargetRCSvsGrazAng(grazv,tgtrcsv,landtype,fsub)
plot(grazv,pow2db(tgtrcsv(:,:,1)),'LineWidth',1.5)
set(gca,'ColorOrderIndex',1)
hold on
plot(grazv,pow2db(tgtrcsv(:,:,2)),'--','LineWidth',1.5)
hold off
grid on
axis padded
xlabel('Grazing Angle (degrees)')
ylabel('Target RCS (dBsm)')
title('Surface Roughness Scattering')
legend([char(landtype(1)) ' @ ' num2str(round(fsub(1)/1e9,1)) ' GHz'],...
    [char(landtype(1)) ' @ ' num2str(round(fsub(2)/1e9,1)) ' GHz'],[char(landtype(1)) ' @ ' num2s
    [char(landtype(2)) ' @ ' num2str(round(fsub(1)/1e9,1)) ' GHz'],[char(landtype(2)) ' @ ' num2s
    [char(landtype(2)) ' @ ' num2str(round(fsub(3)/1e9,1)) ' GHz'],'Location','northwest')
end
```

plotTargetRCSvsPol

```matlab
function plotTargetRCSvsPol(grazvpol,tgtrcsv,seaStates,pol,freq)
plot(grazvpol,pow2db(squeeze(tgtrcsv(:,1,:))),'LineWidth',1.5)
set(gca,'ColorOrderIndex',1)
hold on
plot(grazvpol,pow2db(squeeze(tgtrcsv(:,2,:))),'--','LineWidth',1.5)
hold off
grid on
```

1-305

```
xlabel('Grazing Angle (degrees)')
ylabel('Target RCS (dBsm)')
title(['Surface Roughness Scattering @ ' num2str(round(freq/1e9,1)) ' GHz vs. Polarization'])
legend(['Sea State ' num2str(seaStates(1)) ' - ' char(pol(1)) ' pol'],...
    ['Sea State ' num2str(seaStates(2)) ' - ' char(pol(1)) ' pol'],...
    ['Sea State ' num2str(seaStates(3)) ' - ' char(pol(1)) ' pol'],...
    ['Sea State ' num2str(seaStates(1)) ' - ' char(pol(2)) ' pol'],...
    ['Sea State ' num2str(seaStates(2)) ' - ' char(pol(2)) ' pol'],...
    ['Sea State ' num2str(seaStates(3)) ' - ' char(pol(2)) ' pol'],'Location','southeast')
end
```

plotRainSCR

```
function plotRainSCR(rainRate,rrcs,scr,landType)
t = tiledlayout(1,2);
nexttile
% Plot rain RCS vs rain rate
plot(rainRate,pow2db(rrcs),'LineWidth',1.5)
grid on
xlabel('Rain rate (mm/Hr)')
ylabel('Rain RCS (dBsm)')
legend('HH polarization')

nexttile
% Plot signal-to-clutter ratio (SCR) vs. rain rate
plot(rainRate,scr,'LineWidth',1.5)
grid on
xlabel('Rain rate (mm/Hr)')
ylabel('Signal-to-Clutter ratio (dB)')
legend(landType(1),landType(2),landType(3))
title(t, 'Rain Clutter')
end
```

Airborne SAR System Design

This example shows how to design a synthetic aperture radar (SAR) sensor operating in the X-band and calculate the sensor parameters. SAR uses the motion of the radar antenna over a target region to provide an image of the target region. A synthetic aperture is created when the SAR platform travels over the target region while pulses are transmitted and received from the radar antenna.

This example focuses on designing an SAR sensor to meet a set of performance parameters. It outlines the steps to translate performance specifications, such as the azimuth resolution and the probability of detection, into SAR system parameters, such as the antenna dimension and the transmit power. It models design parameters for stripmap and spotlight modes of operation. Compared to the stripmap operation, spotlight mode can provide a better resolution, and a stronger signal from the scene at the cost of reduced scene size or area imaging rate. The example also models the parameters of the azimuth chirp signal.

The diagram below classifies the various system and performance parameters. This example covers the functions for selecting system parameters to meet performance parameters.
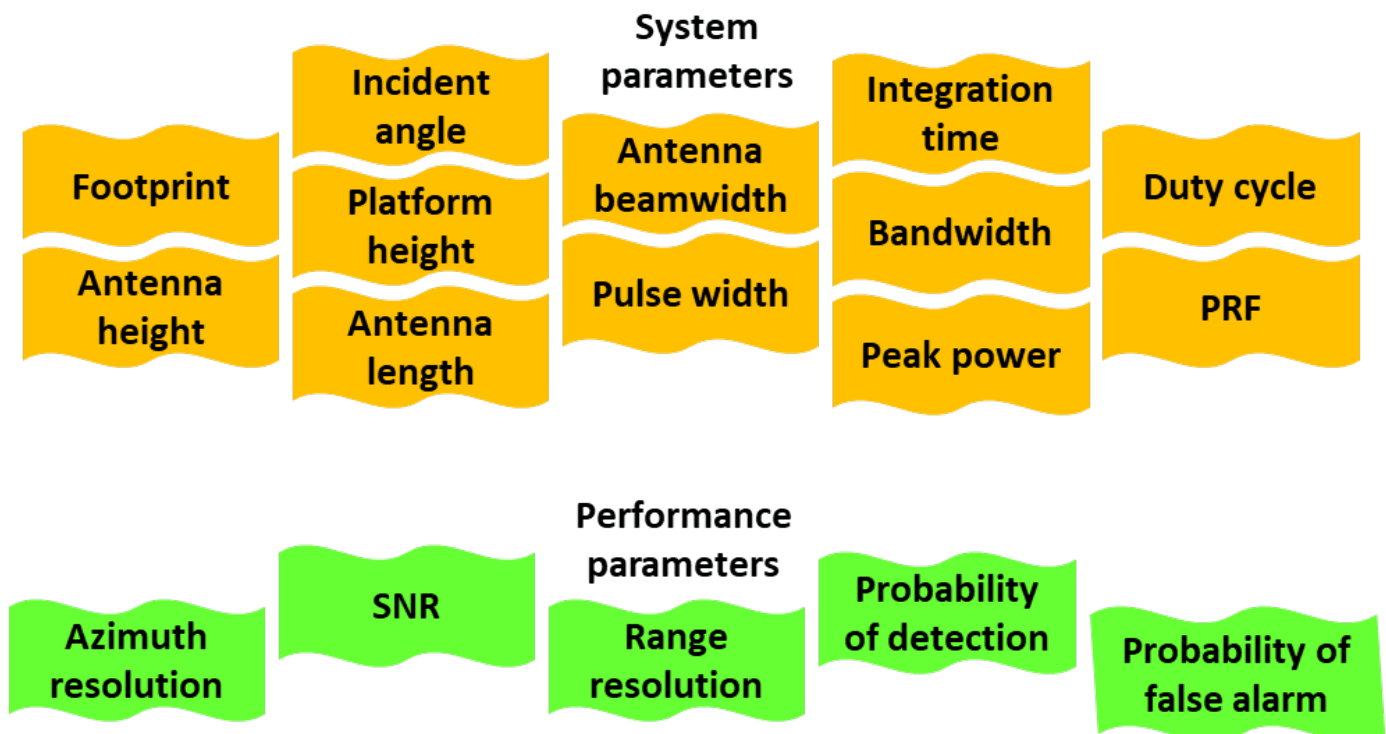
Design Specifications

The goal of this airborne SAR system is to provide an image of the target region at a distance up to 10 km from the airborne platform with a range and azimuth resolution of 1 m. The platform is operating at an altitude of 5 km and moving at a velocity of 100 m/s. The desired performance indices are the probability of detection (Pd) and probability of false alarm (Pfa). The Pd value must be 0.9 or greater. The Pfa value must be less than 1e-6.

```
slantrngres = 1;          % Required slant range resolution (m)
azres = 1;                % Required azimuth resolution (m)
maxrng = 10e3;            % Maximum unambiguous slant range (m)
pd = 0.9;                 % Probability of detection
pfa = 1e-6;               % Probability of false alarm
v = 100;                  % Velocity (m/s)
h = 5000;                 % Radar altitude (m)
```

Airborne SAR System Design

System parameters like length of synthetic aperture, integration time, coverage rate, beamwidth for stripmap as well as spotlight modes, and signal bandwidth are key parameters that define the operational capability of a SAR system. These parameters ensure that the SAR system covers the region of interest for the calculated integration time with a wide beam. The calculated signal bandwidth meets the desired range resolution.

Signal Configuration

To calculate the SAR system parameters, you must first know the wavelength of the propagating signal, which is inversely related is to the operating frequency of the system. For this example, set the operating frequency to 10 GHz which is typical airborne SAR systems.

Use the `freq2wavelen` function to calculate the wavelength of the propagating signal.

```
freq = 10e9;                      % Radar frequency within X-band (Hz)
lambda = freq2wavelen(freq)  % Wavelength (m)
```

```
lambda = 0.0300
```

The signal bandwidth maps to the slant range resolution of SAR and the slant range resolution is the factor needed to distinguish two targets separated by a distance. The slant range resolution gives you the minimum range difference needed to distinguish two targets. Use the `rangeres2bw` function to calculate the signal bandwidth, which is determined by the slant range resolution.

```
pulse_bw = rangeres2bw(slantrngres)     % Pulse bandwidth (Hz)
```

```
pulse_bw = 149896229
```

Stripmap SAR Mode

Stripmap SAR mode assumes a fixed pointing direction of the radar antenna relative to the direction of motion of platform. The antenna in this example points to the broadside direction.

Antenna Orientation

The depression angle is often used to define the antenna pointing direction in elevation. This example assumes that the earth is flat so that the depression angle is the same as the grazing angle.

Use the `grazingang` function to calculate the grazing angle from the line-of-sight range.

```
grazang = grazingang(h,maxrng,'Flat')  % Grazing angle (in degrees)
```

```
grazang = 30.0000
```

Antenna Azimuth Dimension

Next, use the `sarlen` and `sarazres` functions to analyze and calculate the synthetic aperture length and its azimuth resolution for selecting the antenna azimuth dimension. Plot the synthetic length as a function of cross-range resolution. Plot the antenna azimuth dimension as a function of synthetic length.

```
dazv = [1 1.5 2 2.5 3];     % Antenna azimuth dimensions (m)
striplenv = zeros(1,numel(dazv));
stripazresv = zeros(1,numel(dazv));
for i=1:numel(dazv)
    striplenv(i) = sarlen(maxrng,lambda,dazv(i));
    stripazresv(i) = sarazres(maxrng,lambda,striplenv(i));
end
```

```
helperPlotStripmapMode(stripazresv,striplenv,dazv,azres)
```
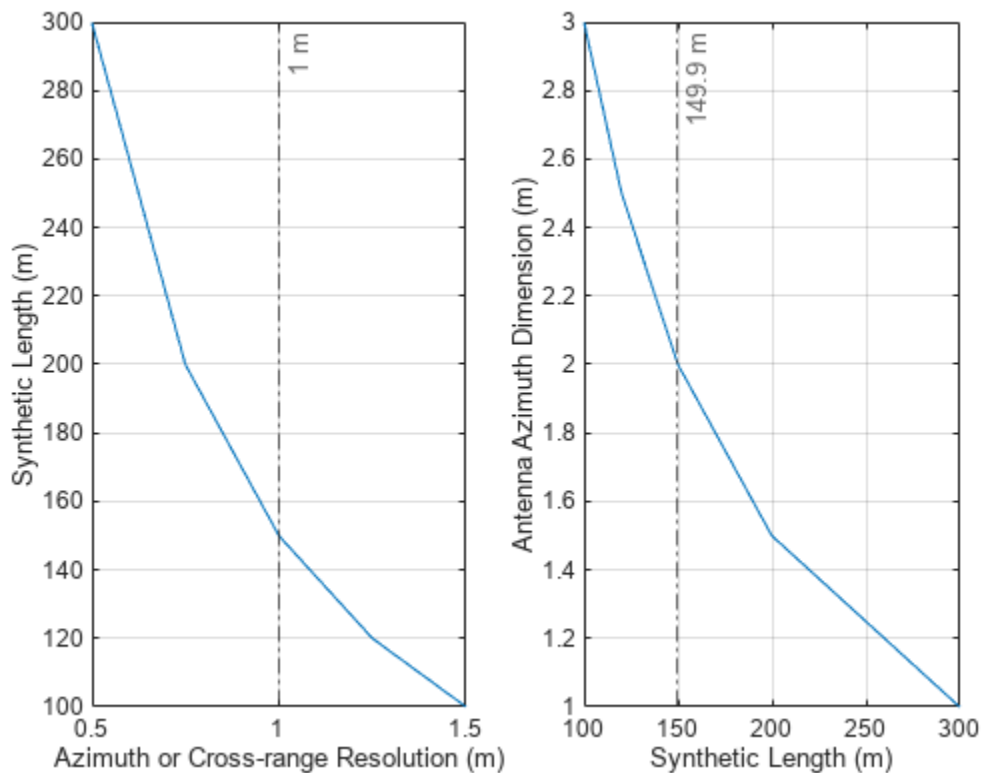

The figures show that a synthetic aperture length of 149.9 m for stripmap mode is a good value to meet a required azimuth resolution of 1 m. The smallest antenna azimuth dimension you can use for stripmap mode in this scenario is 2 m. Decrease the antenna azimuth dimension to obtain a better azimuth resolution than 1 m for stripmap mode.

Set the synthetic aperture length to 149.9 m for stripmap mode and the antenna azimuth dimension of 2 m.

```
daz = 2
```

```
daz = 2
```

```
striplen = 149.9
```

```
striplen = 149.9000
```

Antenna Elevation Dimension

Next, determine the antenna elevation dimension based on the required swath length. For this example, assume that the required swath length is 2.4 km.

Use the `aperture2swath` function to analyze the swath length for selecting an antenna elevation dimension.

```
rngswath = 2400;
delv = [0.15 0.2 0.25 0.3 0.35];     % Elevation dimensions (m)
rangeswathv = zeros(1,numel(delv));
for i=1:numel(delv)
    [rangeswathv(i),crngswath] = aperture2swath(maxrng,lambda,[delv(i) daz],grazang);
end
clf
plot(rangeswathv,delv)
grid on
xline(rngswath,'-.',{[num2str(round(rngswath,2)),' m']}); % Selected range swath
xlabel('Swath Length (m)')
ylabel('Antenna Elevation Dimension (m)')
```
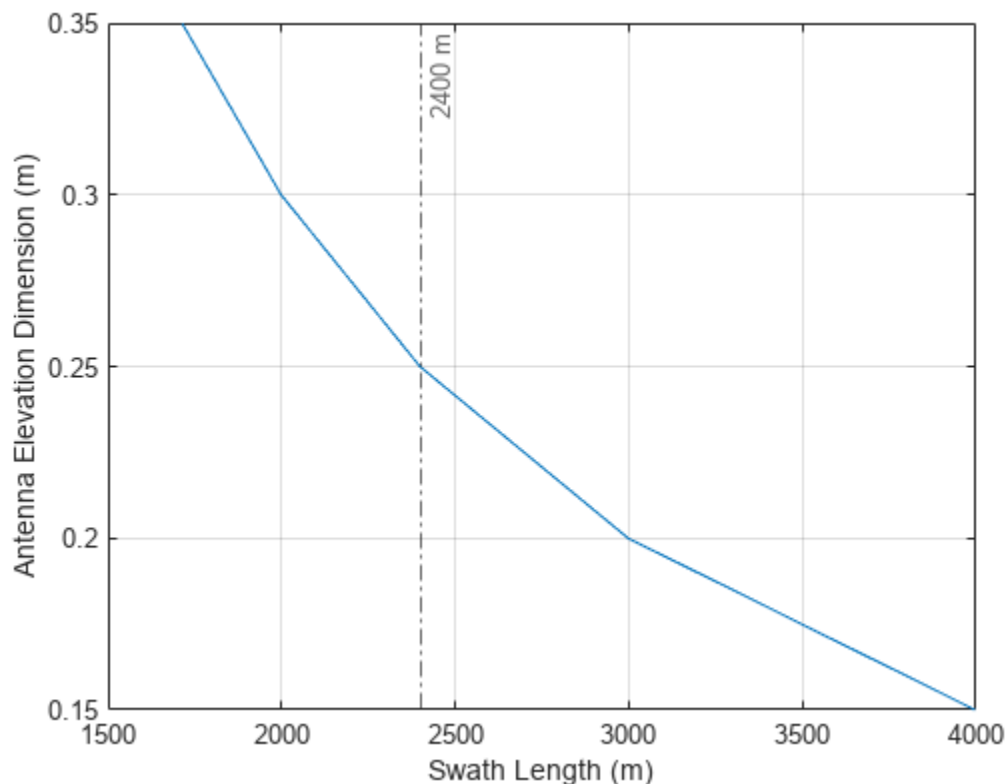

The figure indicates that an antenna elevation dimension of 0.25 m is appropriate given a swath length of 2400 m.

Set the antenna elevation dimension to 0.25 m.

```
del = 0.25
```

```
del = 0.2500
```

Real Antenna Beamwidth and Gain

Use the `ap2beamwidth` function to calculate the real antenna beamwidth.

```
realAntBeamwidth = ap2beamwidth([daz del],lambda) % [Az El] (deg)
```

```
realAntBeamwidth = 2×1

    0.8588
    6.8707
```

Use the `aperture2gain` function to calculate the antenna gain.

```
antGain = aperture2gain(daz*del, lambda) % dBi
```

```
antGain = 38.4454
```

Synthetic Beamwidth, Processing Time, and Constraints

Next, use the `sarbeamwidth`, `sarinttime`, `sarmaxcovrate`, and `sarmaxswath` functions to calculate the synthetic beamwidth, integration time, area coverage rate, and maximum swath length. Notice that the azimuth beamwidth for SAR system is much smaller than the azimuth beamwidth for a real aperture radar.

```
stripsynbw = sarbeamwidth(lambda,striplen); % Synthetic beamwidth (degrees)
stripinttime = sarinttime(v,striplen);      % Integration time (s)
stripcovrate = sarmaxcovrate(azres,grazang);      % Upper bound on coverage rate (m^2/s)
stripswlen = sarmaxswath(v,azres,grazang);        % Upper bound on swath length (m)

RealAntenna = [realAntBeamwidth(1); NaN; NaN; NaN];
Parameters = ["Synthetic Beamwidth";"Integration Time";"Upper Bound on Area Coverage Rate";...
    "Upper Bound on Swath Length"];
StripmapSAR = [stripsynbw;stripinttime;round(stripcovrate/1e6,1);round(stripswlen/1e3)];
Units = ["degrees";"s";"km^2/s";"km"];
sarparams = table(Parameters,RealAntenna,StripmapSAR,Units)
```

```
sarparams=4×4 table
              Parameters                    RealAntenna    StripmapSAR      Units

    _____    _____    _____    _____

    "Synthetic Beamwidth"                    0.85884       0.0057294     "degrees"
    "Integration Time"                          NaN           1.499      "s"
    "Upper Bound on Area Coverage Rate"         NaN           173.1      "km^2/s"
    "Upper Bound on Swath Length"               NaN           1731       "km"
```

The maximum possible azimuth resolution using SAR in this scenario is 1 m. However, to achieve this performance, pulses need to be integrated for over 1.5 s. The upper bound on the area coverage rate is 173 km^2/s. The upper bound on the maximum swath length is 1731 km.

Spotlight SAR Mode

Spotlight SAR is capable of extending the SAR imaging capability to high resolution imaging significantly. This is possible since the spotlight mode ensures that the radar antenna squints instantaneously around the region being imaged thereby illuminating the target region for longer duration as compared to stripmap mode.

Coherent Integration Angle

The azimuth resolution in stripmap mode is 1 m in this example. The resolution of spotlight mode is often expressed in terms of coherent integration angle of the radar boresight vector as the platform traverses the synthetic aperture length.

Use the `sarintang` and `sarlen` functions to calculate the coherent integration angle and synthetic aperture length.

```
ciang = sarintang(lambda,azres)     % (degrees)
```

```
ciang = 0.8589
```

```
len = sarlen(maxrng,'CoherentIntegrationAngle',ciang)    % (m)
```

```
len = 149.8976
```

The best possible azimuth resolution in stripmap mode is 1 m for an antenna azimuth dimension of 2 m. Use the same antenna azimuth dimension of 2 m to obtain a better azimuth resolution of 0.5 m in spotlight mode. In spotlight mode, steer the radar beam to keep the target within for a longer time and thus form a longer synthetic aperture.

Next, use the `sarlen` and `sarazres` functions to analyze the synthetic aperture length and its azimuth resolution over varying coherent integration angles.

```
spotazres = 0.5;        % Azimuth resolution in spotlight SAR (m)
intangv = 1:0.01:2.5;   % Coherent integration angles (degrees)
spotlenv = zeros(1,numel(intangv));
spotazresv = zeros(1,numel(intangv));
for i=1:numel(intangv)
    spotlenv(i) = sarlen(maxrng,'CoherentIntegrationAngle',intangv(i));
    spotazresv(i) = sarazres(maxrng,lambda,spotlenv(i));
end

helperPlotSpotlightMode(spotazresv,spotlenv,intangv,spotazres)
```
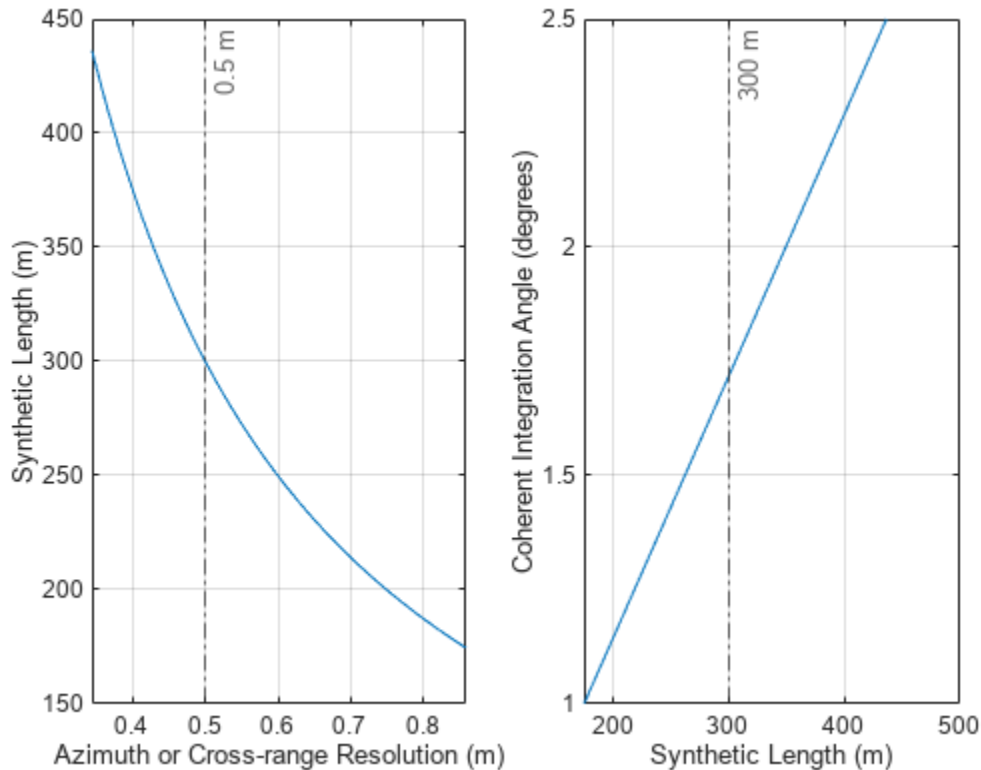
The figure indicates that in spotlight SAR mode, a synthetic aperture length of 300 m for the spotlight mode corresponding to an azimuth resolution of 0.5 m. For a coherent integration angle of 1.71 degrees, the azimuth resolution in spotlight mode is 0.5 m. It is important to note that decrease the antenna azimuth dimension to obtain a similar azimuth resolution in stripmap mode.

Set the synthetic aperture length to 300 m and the coherent integration angle to 1.71 degrees for spotlight mode.

```
spotlen = 300
```

spotlen = 300

```
intang = 1.71
```

intang = 1.7100

Synthetic Beamwidth, Processing Time, and Constraint

Compared to the stripmap mode, spotlight mode can provide a better resolution, and a stronger signal from the scene at the cost of reduced scene size or area imaging rate.

Use the `sarbeamwidth`, `sarinttime`, `sarmaxcovrate`, and `sarmaxswath` functions to calculate the synthetic beamwidth, integration time, area coverage rate, and maximum swath length. Notice that the area coverage rate and maximum swath length for spotlight SAR system are much smaller than for stripmap mode.

```
spotsynbw = sarbeamwidth(lambda,spotlen);          % Synthetic beamwidth (degrees)
spotinttime = sarinttime(v,spotlen);               % Integration time (s)
```

```
spotcovrate = sarmaxcovrate(spotazres,grazang);      % Upper bound on coverage rate (m^2/s)
spotswlen = sarmaxswath(v,spotazres,grazang);        % Upper bound on swath length (m)

SpotlightSAR = [spotsynbw;spotinttime;round(spotcovrate/1e6,1);round(spotswlen/1e3)];

sar = table(Parameters,StripmapSAR,SpotlightSAR,Units)
```

sar=4×4 table

Parameters	StripmapSAR	SpotlightSAR	Units
"Synthetic Beamwidth"	0.0057294	0.0028628	"degrees"
"Integration Time"	1.499	3	"s"
"Upper Bound on Area Coverage Rate"	173.1	86.5	"km^2/s"
"Upper Bound on Swath Length"	1731	865	"km"

Azimuth Chirp Signal Parameters

Determine the azimuth chirp signal parameters which are the azimuth chirp rate, Doppler bandwidth, beam compression ratio, and azimuth bandwidth after dechirp. You can derive the azimuth time-bandwidth product. These are important for designing an accurate synthetic aperture processing mechanism in azimuth.

Use the `sarchirprate` function to calculate the azimuth chirp rate, which is the rate at which the azimuth signal changes frequency as the sensor illuminates a scatterer.

```
azchirp = sarchirprate(maxrng,lambda,v); % (Hz/s)
```

Analyze the azimuth chirp rate sensitivity to range and Doppler cone angle variations. The plot shows that increasing the unambiguous range of the radar reduces the azimuth chirp rate.

```
dcang = 60:1:120;            % Doppler cone angles (in degrees)
rngv = 1e3:100:maxrng;
azchirpv = zeros(length(dcang),length(rngv));
for i = 1:length(dcang)
    azchirpv(i,:) = sarchirprate(rngv,lambda,v,dcang(i));
end
clf
mesh(rngv/1e3,dcang,azchirpv)
xlabel('Range (km)')
ylabel('Doppler Cone Angle (degrees)')
zlabel('Azimuth Chirp Rate (Hz/s)')
view([45 45]);
```
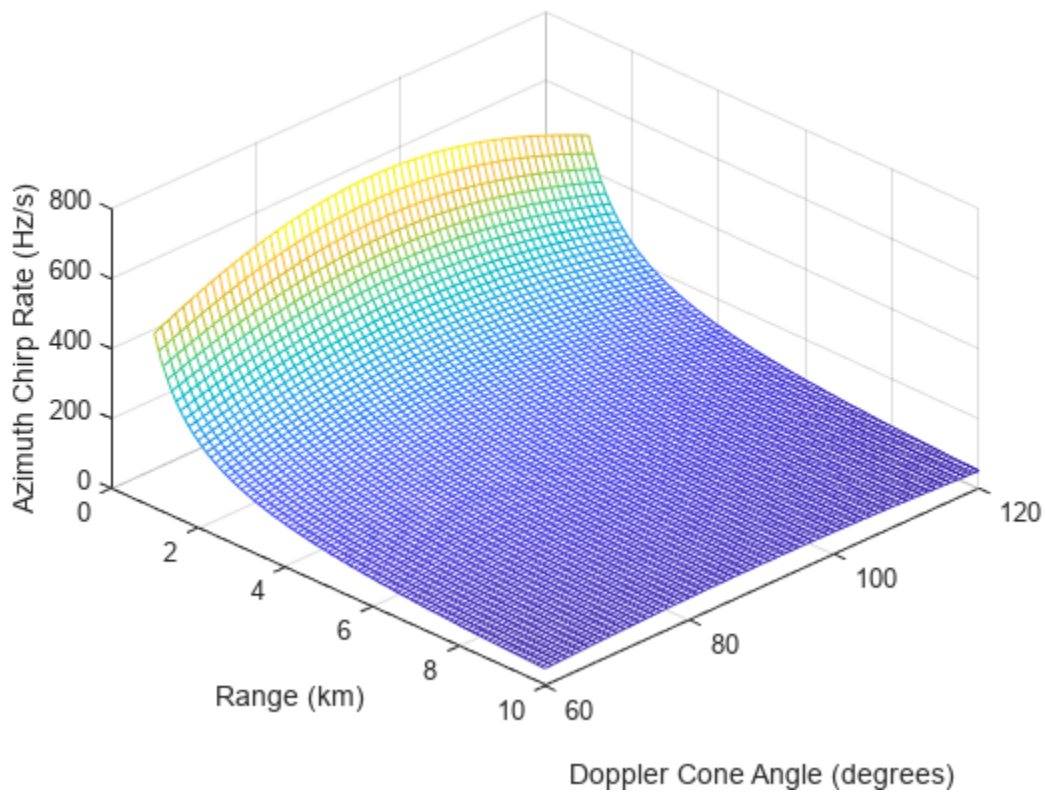
Use the `sarscenedopbw` function to calculate the scene bandwidth after azimuth dechirping. Assume a scene size of 916 m.

```
Wa = 916;
bwdechirp = sarscenedopbw(maxrng,lambda,v,Wa); % (Hz)
```

Analyze the scene bandwidth sensitivity to Doppler cone angle variations.

```
bwdechirpv = zeros(length(dcang),1);
for i = 1:length(dcang)
    bwdechirpv(i,:) = sarscenedopbw(maxrng,lambda,v,Wa,dcang(i));
end
clf
plot(dcang,bwdechirpv)
grid on
xlabel('Doppler Cone Angle (degrees)')
ylabel('Azimuth Bandwidth after Dechirp (Hz)')
```
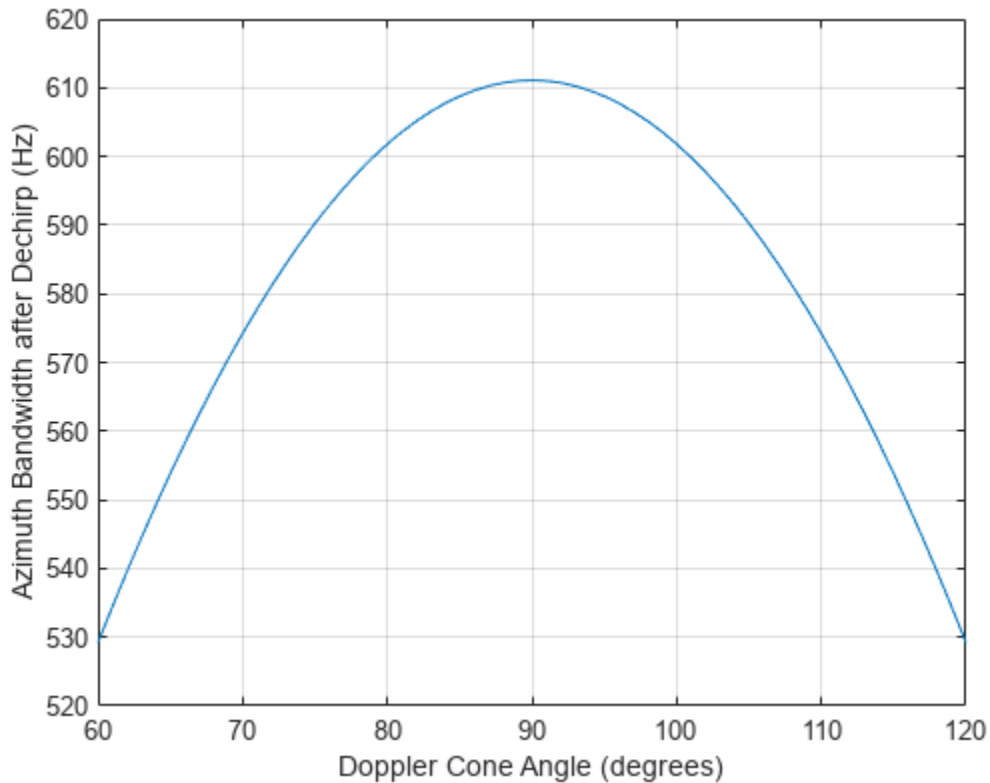
Next, use the `sarpointdopbw` and `sarbeamcompratio` functions to calculate the Doppler bandwidth of the received signal from a point scatterer and the beam compression ratio. Notice that the Doppler bandwidth, and beam compression ratio for a spotlight SAR mode are much greater than for stripmap SAR mode.

```
% Stripmap SAR Mode
stripbwchirp = sarpointdopbw(v,azres);   % (Hz)
striptbwaz = bwdechirp*stripinttime;     % Unitless
stripbcr = sarbeamcompratio(maxrng,lambda,striplen,Wa); % Unitless
% Spotlight SAR Mode
spotbwchirp = sarpointdopbw(v,spotazres); % (Hz)
spottbwaz = bwdechirp*spotinttime;        % Unitless
spotbcr = sarbeamcompratio(maxrng,lambda,spotlen,Wa); % Unitless

Parameters = ["Doppler Bandwidth from Point Scatterer";"Azimuth Time-Bandwidth Product";...
    "Beam Compression Ratio";"Azimuth Chirp Rate";"Azimuth Bandwidth after Dechirp"];
StripmapSAR = [stripbwchirp;striptbwaz;stripbcr;round(azchirp);bwdechirp];
SpotlightSAR = [spotbwchirp;round(spottbwaz);round(spotbcr);round(azchirp);bwdechirp];
Units = ["Hz";"unitless";"unitless";"Hz/s";"Hz"];
r = table(Parameters,StripmapSAR,SpotlightSAR,Units)
```

r=5×4 table

Parameters	StripmapSAR	SpotlightSAR	Units
"Doppler Bandwidth from Point Scatterer"	100	200	"Hz"
"Azimuth Time-Bandwidth Product"	916.02	1833	"unitless"

```
"Beam Compression Ratio"                        916.02          1833        "unitless"
"Azimuth Chirp Rate"                                67            67        "Hz/s"
"Azimuth Bandwidth after Dechirp"               611.09        611.09        "Hz"
```

SAR Power Calculation

Estimate the peak power that must be transmitted using the power form of the radar equation for stripmap SAR mode. The required peak power depends upon many factors, including the maximum unambiguous range, the required SNR at the receiver, and the pulse width of the waveform. Among these factors, the required SNR at the receiver is determined by the design goal for the Pd and Pfa. Model and estimate the target RCS, the PRF, and different sources of gain and loss for the radar system and its environment.

Receiver SNR

First, calculate the SNR required at the receiver. The relation between Pd, Pfa, and SNR can be best represented by a receiver operating characteristics (ROC) curve.

```
snr_db = [-inf, 0, 3, 10, 13];
rocsnr(snr_db);
```
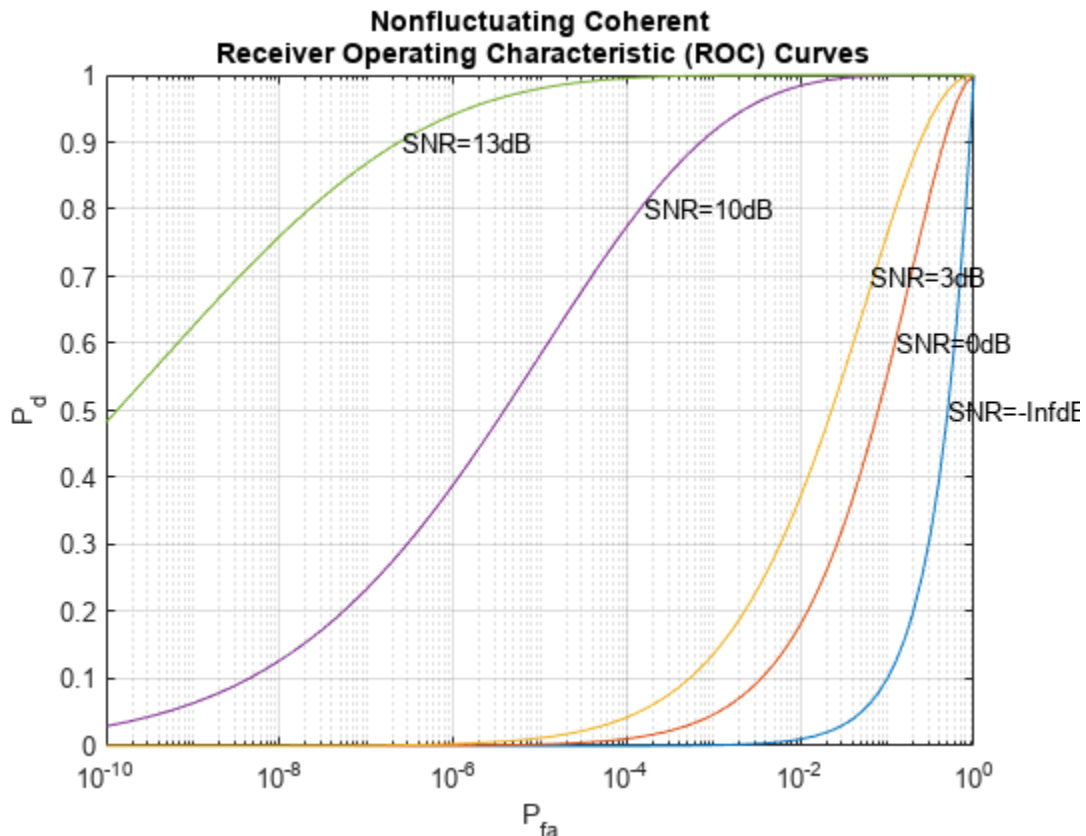

The ROC curves show that to satisfy the design goals of Pfa = 1e-6 and Pd = 0.9, the SNR of the received signal must exceed 13 dB. You can surmise the SNR value by looking at the plot, but calculating only the required value is more straightforward. Using the `albersheim` function, derive the required SNR.

```
snr_min = albersheim(pd, pfa)

snr_min = 13.1145
```

Target RCS

Use the `landreflectivity` function to calculate the reflectivity, which is the normalized radar cross-section (NRCS) for a given grazing angle and operating frequency. Then calculate the target RCS in the ground image plane using the `sarSurfaceRCS` function and taking the radar resolution into account.

```
landType = "Smooth";
nrcs = landreflectivity(landType,grazang,freq); % Calculate normalized RCS of smooth land with no
tgtrcs = sarSurfaceRCS(nrcs,[slantrngres azres],grazang);
```

Upper and Lower PRF Bounds

Use the `sarprfbounds` function to determine the minimum and maximum PRF values for a range swath and azimuth resolution given the radar velocity and the grazing angle.

```
[prfminv, prfmax] = sarprfbounds(v,azres,rngswath,grazang)

prfminv = 100

prfmax = 6.7268e+04
```

PRF Selection

The PRF is typically programmable and can be optimized for each application. Use the `sarprf` function to calculate the PRF of the radar based on the radar velocity and the real antenna dimension along the azimuth. Specify a constant roll-off factor as a safety margin to prevent mainlobe returns from aliasing in the PRF interval. If the PRF is set too low, the radar suffers from grating lobes and Doppler ambiguities. If the PRF is set too high, range measurements will be ambiguous.

```
prf = sarprf(v,daz,'RollOff',1.5)

prf = 150
```

The selected PRF is within the PRF bounds.

Processing Gains

Use the `matchinggain` function to calculate the range processing gain due to the noise bandwidth reduction after the matched filter.

```
d = 0.05;                              % 5 percent duty cycle
pw = (1/prf)*d;                        % Effective pulse width (s)
rnggain = matchinggain(pw,pulse_bw)    % Range processing gain (dB)

rnggain = 46.9867
```

Use the `sarazgain` function to calculate the azimuth processing gain due to the coherent integration of pulses.

```
azgain = sarazgain(maxrng,lambda,v,azres,prf); % Az processing gain (dB)
```

Losses and Noise Factor

Use the `noisefigure` function to estimate the noise figure of the cascaded receiver stages. Assume seven stages with the following values:

- Stage 1 LNA: Noise Figure = 1.0 dB, Gain = 15.0
- Stage 2 RF Filter: Noise Figure = 0.5 dB, Gain = -0.5
- Stage 3 Mixer: Noise Figure = 5.0 dB, Gain = -7.0
- Stage 4 IF Filter: Noise Figure = 1.0 dB, Gain = -1.0
- Stage 5 IF Preamplifier: Noise Figure = 0.6 dB, Gain = 15.0
- Stage 6 IF Stages: Noise Figure = 1.0 dB, Gain = 20.0
- Stage 7 Phase Detectors: Noise Figure = 6.0 dB, Gain = -5.0

```
nf  = [1.0, 0.5, 5.0, 1.0, 0.6, 1.0, 6.0];          % dB
g   = [15.0, -0.5, -7.0, -1.0, 15.0, 20.0, -5.0]; % dB
cnf = noisefigure(nf, g)
```

```
cnf = 1.5252
```

Use the `radarpropfactor` function to calculate the one-way radar propagation factor over smooth land.

```
[hgtsd, beta0, vegType] = landroughness('Smooth');
tgtheight = hgtsd;
Re = effearthradius(maxrng,h,tgtheight);
propf = radarpropfactor(maxrng,freq,h,tgtheight,'EffectiveEarthradius',Re,'TiltAngle',grazang,..
    'ElevationBeamwidth',realAntBeamwidth(2),'SurfaceHeightStandardDeviation',hgtsd,'SurfaceSlope
    'VegetationType',vegType)
```

```
propf = -0.0042
```

Use the `tropopl` function to calculate losses due to atmospheric gaseous absorption.

```
atmoLoss = tropopl(maxrng,freq,tgtheight,grazang)
```

```
atmoLoss = 0.0731
```

Transmit Power

Use the `radareqsarpow` function to calculate the peak power with the SAR radar equation. You can also specify additional losses and factors, including the azimuth beam shape loss, window loss, transmission loss, and receive line loss. Estimate the beam shape loss with the `beamloss` function, and use 5 dB for all other fixed losses combined. For this analysis, specify `landType` as "Smooth" to use the weakest land target. A finite data collection time limits the total energy collected, and signal processing in the radar increases the SNR in the SAR image by two major gain factors. The first is due to pulse compression, and the second is due to coherent integration of pulses.

```
imgsnr = snr_min + rnggain + azgain; % (dB)
Lb = beamloss;
customLoss = 5;          % dB
sntemp = systemp(cnf); % Noise Temperature
Pt = radareqsarpow(maxrng,lambda,imgsnr,pw,rnggain,azgain,'Gain',antGain,'RCS',tgtrcs,...
    'AtmosphericLoss',atmoLoss,'Loss',cnf,'PropagationFactor',propf,...
    'Ts',sntemp,'CustomFactor',-Lb-customLoss)
```

```
Pt = 535.0561
```

Summary

This example shows the aspects that must be calculated to design an X-band SAR system that can operate in stripmap and spotlight mode. The example shows that the same SAR system can operate in stripmap as well as spotlight modes and achieve varying levels of resolution depending upon the requirements at the cost of other parameters. First, you analyze and select the antenna dimensions to meet the required resolutions. Then you estimate the antenna gains, processing time, constraints, and the azimuth chirp signal parameters. Then estimate the required SNR, target RCS, PRF, processing gains and losses in the radar and its environment. Finally, you use the SAR equation to calculate the peak transmit power.

```
Parameters = ["Antenna Dimension in Azimuth";"Antenna Dimension in Elevation";"Synthetic Aperture
    "Azimuth Resolution";"Synthetic Beamwidth";"Integration Time";"Upper Bound on Area Coverage F
    "Upper Bound on Swath Length";"Coherent Integration Angle";"Doppler Bandwidth from Point Scat
    "Azimuth Time-Bandwidth Product";"Beam Compression Ratio";"Azimuth Chirp Rate";"Azimuth Bandw
Stripmap = [daz;del;striplen;azres;stripsynbw;stripinttime;round(stripcovrate/1e6,1);round(strips
    NaN;stripbwchirp;striptbwaz;stripbcr;round(azchirp);bwdechirp];
Spotlight = [daz;del;spotlen;spotazres;spotsynbw;spotinttime;round(spotcovrate/1e6,1);round(spots
    intang;spotbwchirp;round(spottbwaz);round(spotbcr);round(azchirp);bwdechirp];
Units = ["m";"m";"m";"m";"degrees";"s";"km^2/s";"km";"degrees";"Hz";"unitless";...
    "unitless";"Hz/s";"Hz"];
T = table(Parameters,Stripmap,Spotlight,Units)
```

T=*14×4 table*

Parameters	Stripmap	Spotlight	Units
"Antenna Dimension in Azimuth"	2	2	"m"
"Antenna Dimension in Elevation"	0.25	0.25	"m"
"Synthetic Aperture Length"	149.9	300	"m"
"Azimuth Resolution"	1	0.5	"m"
"Synthetic Beamwidth"	0.0057294	0.0028628	"degrees"
"Integration Time"	1.499	3	"s"
"Upper Bound on Area Coverage Rate"	173.1	86.5	"km^2/s"
"Upper Bound on Swath Length"	1731	865	"km"
"Coherent Integration Angle"	NaN	1.71	"degrees"
"Doppler Bandwidth from Point Scatterer"	100	200	"Hz"
"Azimuth Time-Bandwidth Product"	916.02	1833	"unitless"
"Beam Compression Ratio"	916.02	1833	"unitless"
"Azimuth Chirp Rate"	67	67	"Hz/s"
"Azimuth Bandwidth after Dechirp"	611.09	611.09	"Hz"

References

[1] Carrara, Walter G., Ronald M. Majewski, and Ron S. Goodman. Spotlight Synthetic Aperture Radar: Signal Processing Algorithms. Boston: Artech House, 1995.

Supporting Functions

helperPlotStripmapMode

```
function helperPlotStripmapMode(stripazresv,striplenv,dazv,azres)
% Plot azimuth resolution vs. synthetic aperture length
subplot(1,2,1)
plot(stripazresv,striplenv)
```

```
grid on
xline(azres,'-.',{[num2str(round(azres)),' m']}); % Selected azimuth resolution
xlabel('Azimuth or Cross-range Resolution (m)')
ylabel('Synthetic Length (m)')

stripidx = find(abs(striplenv-150)<1); % Index corresponding to required azimuth resolution

% Plot synthetic aperture length vs. antenna azimuth dimensions
subplot(1,2,2)
plot(striplenv,dazv)
grid on
xline(striplenv(stripidx),'-.',{[num2str(round(striplenv(stripidx),2)),' m']}); % Selected synthe
xlabel('Synthetic Length (m)')
ylabel('Antenna Azimuth Dimension (m)')
end
```

helperPlotSpotlightMode

```
function helperPlotSpotlightMode(spotazresv,spotlenv,intangv,spotazres)
% Plot azimuth resolution vs. synthetic aperture length
subplot(1,2,1)
plot(spotazresv,spotlenv)
grid on
xline(0.5,'-.',{[num2str(round(spotazres,2)),' m']}); % Selected azimuth resolution
xlabel('Azimuth or Cross-range Resolution (m)')
ylabel('Synthetic Length (m)')

spotidx = find(abs(spotlenv-300)<1); % Index corresponding to 0.5 m azimuth resolution

% Plot synthetic aperture length vs. coherent integration angles
subplot(1,2,2)
plot(spotlenv,intangv)
grid on
xline(spotlenv(spotidx),'-.',{[num2str(round(spotlenv(spotidx))),' m']}); % Selected synthetic le
xlabel('Synthetic Length (m)')
ylabel('Coherent Integration Angle (degrees)')
end
```

Stripmap Synthetic Aperture Radar (SAR) Image Formation

This example shows how to model a stripmap-based Synthetic Aperture Radar (SAR) system using a Linear FM (LFM) Waveform. SAR is a type of side-looking airborne radar where the achievable cross-range resolution is much higher as compared to a real aperture radar. The image generated using SAR has its own advantages primarily pertaining to the use of an active sensor (radar) as opposed to conventional imaging systems employing a passive sensor (camera) that rely on the ambient lighting to obtain the image. Since an active sensor is used, the system provides all-weather performance irrespective of snow, fog, or rain. Also, configuring the system to work at different frequencies such as L-, S-, or C-band can help analyze different layers on the ground based on varying depth of penetrations. Because the resolution of SAR depends upon the signal and antenna configuration, resolution can be much higher than for vision-based imaging systems. Using the stripmap mode, this example performs both a range migration algorithm [1] and an approximate form of a back-projection algorithm [2] to image stationary targets. The approximate form of the back-projection algorithm has been chosen for the reduced computational complexity as indicated in [3]. A Linear FM waveform offers the advantage of large time-bandwidth product at a considerably lower transmit power making it suitable for use in airborne systems.

Synthetic Aperture Radar Imaging

SAR generates a two-dimensional (2-D) image. The direction of flight is referred to as the cross-range or azimuth direction. The direction of the antenna boresight (broadside) is orthogonal to the flight path and is referred to as the cross-track or range direction. These two directions provide the basis for the dimensions required to generate an image obtained from the area within the antenna beamwidth through the duration of the data collection window. The cross-track direction is the direction in which pulses are transmitted. This direction provides the slant range to the targets along the flight path. The energy received after reflection off the targets for each pulse must then be processed (for range measurement and resolution). The cross-range or azimuth direction is the direction of the flight path and it is meaningful to process the ensemble of the pulses received over the entire flight path in this direction to achieve the required measurement and resolution. Correct focusing in both the directions implies a successful generation of image in the range and cross-range directions. It is a requirement for the antenna beamwidth to be wide enough so that the target is illuminated for a long duration by the beam as the platform moves along its trajectory. This helps provide more phase information. The key terms frequently encountered when working with SAR are:

1 Cross-range (azimuth): This parameter defines the range along the flight path of the radar platform.

2 Range: This parameter defines the range orthogonal to the flight path of the radar platform.

3 Fast-time: This parameter defines the time duration for the operation of each pulse.

4 Slow-time: This parameter defines the cross-range time information. The slow time typically defines the time instances at which the pulses are transmitted along the flight path.

Top View

Side View

Flight Path(Cross-Range)
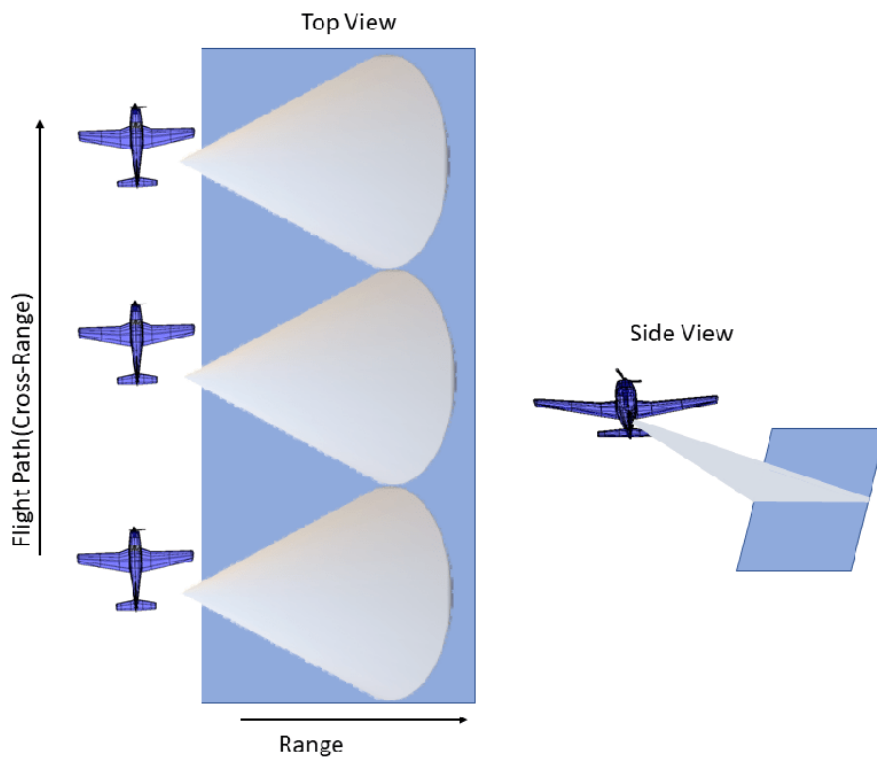
Range

Radar Configuration

Consider a SAR radar operating in C-band with a 4 GHz carrier frequency and a signal bandwidth of 50 MHz. This bandwidth yields a range resolution of 3 meters. The radar system collects data orthogonal to the direction of motion of the platform as shown in this figure. The received signal is a delayed replica of the transmitted signal. The delay corresponds in general to the slant range between the target and the platform. For a SAR system, the slant range varies over time as the platform traverses a path orthogonal to the direction of antenna beam. This section focuses on defining the parameters for the transmission waveform. The LFM sweep bandwidth can be decided based on the desired range resolution.

Set the physical constant for speed of light.

```
c = physconst('LightSpeed');
```

Set the SAR center frequency.

```
fc = 4e9;
```

Set the desired range and cross-range resolution to 3 meters.

```
rangeResolution = 3;
crossRangeResolution = 3;
```

The signal bandwidth is a parameter derived from the desired range resolution.

```
bw = c/(2*rangeResolution);
```

In a SAR system the PRF has dual implications. The PRF not only determines the maximum unambiguous range but also serves as the sampling frequency in the cross-range direction. If the PRF

is too low to achieve a higher unambiguous range, there is a longer duration pulses resulting in fewer pulses in a particular region. At the same time, if the PRF is too high, the cross-range sampling is achieved but at the cost of reduced range. Therefore, the PRF should be less than twice the Doppler frequency and should also satisfy the criteria for maximum unambiguous range

```
prf = 1000;
aperture = 4;
tpd = 3*10^-6;
fs = 120*10^6;
```

Configure the LFM signal of the radar.

```
waveform = phased.LinearFMWaveform('SampleRate',fs, 'PulseWidth', tpd, 'PRF', prf,...
    'SweepBandwidth', bw);
```

Assume the speed of aircraft is 100 m/s with a flight duration of 4 seconds.

```
speed = 100;
flightDuration = 4;
radarPlatform  = phased.Platform('InitialPosition', [0;-200;500], 'Velocity', [0; speed; 0]);
slowTime = 1/prf;
numpulses = flightDuration/slowTime +1;

maxRange = 2500;
truncrangesamples = ceil((2*maxRange/c)*fs);
fastTime = (0:1/fs:(truncrangesamples-1)/fs);
% Set the reference range for the cross-range processing.
Rc = 1000;
```

Configure the SAR transmitter and receiver. The antenna looks in the broadside direction orthogonal to the flight direction.

```
antenna = phased.CosineAntennaElement('FrequencyRange', [1e9 6e9]);
antennaGain = aperture2gain(aperture,c/fc);

transmitter = phased.Transmitter('PeakPower', 50e3, 'Gain', antennaGain);
radiator = phased.Radiator('Sensor', antenna,'OperatingFrequency', fc, 'PropagationSpeed', c);

collector = phased.Collector('Sensor', antenna, 'PropagationSpeed', c,'OperatingFrequency', fc);
receiver = phased.ReceiverPreamp('SampleRate', fs, 'NoiseFigure', 30);
```

Configure the propagation channel.

```
channel = phased.FreeSpace('PropagationSpeed', c, 'OperatingFrequency', fc,'SampleRate', fs,...
    'TwoWayPropagation', true);
```

Scene Configuration

In this example, three static point targets are configured at locations specified below. All targets have a mean RCS value of 1 meter-squared.

```
targetpos= [800,0,0;1000,0,0; 1300,0,0]';

targetvel = [0,0,0;0,0,0; 0,0,0]';

target = phased.RadarTarget('OperatingFrequency', fc, 'MeanRCS', [1,1,1]);
pointTargets = phased.Platform('InitialPosition', targetpos,'Velocity',targetvel);
% The figure below describes the ground truth based on the target
```

```
% locations.
figure(1);h = axes;plot(targetpos(2,1),targetpos(1,1),'*g');hold all;plot(targetpos(2,2),targetpo
set(h,'Ydir','reverse');xlim([-10 10]);ylim([700 1500]);
title('Ground Truth');ylabel('Range');xlabel('Cross-Range');
```
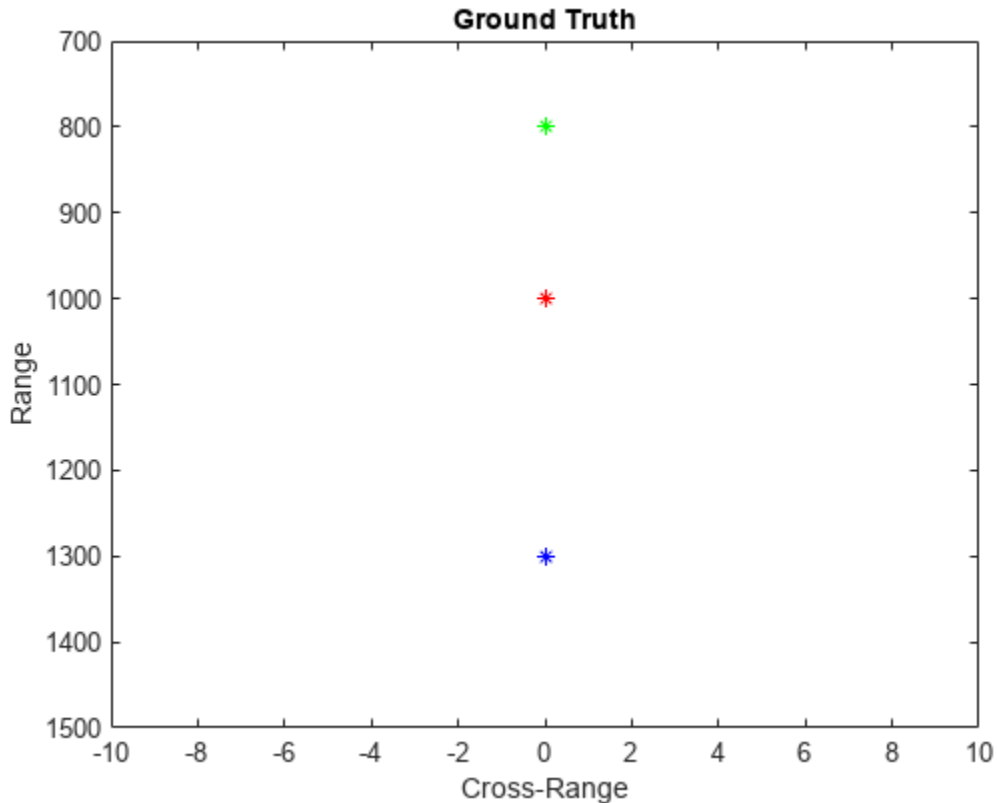

SAR Signal Simulation

The following section describes how the system operates based on the above configuration. Specifically, the section below shows how the data collection is performed for a SAR platform. As the platform moves in the cross-range direction, pulses are transmitted and received in directions orthogonal to the flight path. A collection of pulses gives the phase history of the targets lying in the illumination region as the platform moves. The longer the target lies in the illumination region, the better the cross-range resolution for the entire image because the process of range and cross-range focusing is generalized for the entire scene.

```
% Define the broadside angle
refangle = zeros(1,size(targetpos,2));
rxsig = zeros(truncrangesamples,numpulses);
for ii = 1:numpulses
    % Update radar platform and target position
    [radarpos, radarvel] = radarPlatform(slowTime);
    [targetpos,targetvel] = pointTargets(slowTime);

    % Get the range and angle to the point targets
    [targetRange, targetAngle] = rangeangle(targetpos, radarpos);

    % Generate the LFM pulse
```

```
    sig = waveform();
    % Use only the pulse length that will cover the targets.
    sig = sig(1:truncrangesamples);

    % Transmit the pulse
    sig = transmitter(sig);

    % Define no tilting of beam in azimuth direction
    targetAngle(1,:) = refangle;

    % Radiate the pulse towards the targets
    sig = radiator(sig, targetAngle);

    % Propagate the pulse to the point targets in free space
    sig = channel(sig, radarpos, targetpos, radarvel, targetvel);

    % Reflect the pulse off the targets
    sig = target(sig);

    % Collect the reflected pulses at the antenna
    sig = collector(sig, targetAngle);

    % Receive the signal
    rxsig(:,ii) = receiver(sig);

end
```

Visualize the received signal. The received signal can now be visualized as a collection of multiple pulses transmitted in the cross-range direction. The plots show the real part of the signal for the three targets. The range and cross-range chirps can be seen clearly. The target responses can be seen as overlapping as the pulse-width is kept longer to maintain average power.

```
imagesc(real(rxsig));title('SAR Raw Data')
xlabel('Cross-Range Samples')
ylabel('Range Samples')
```
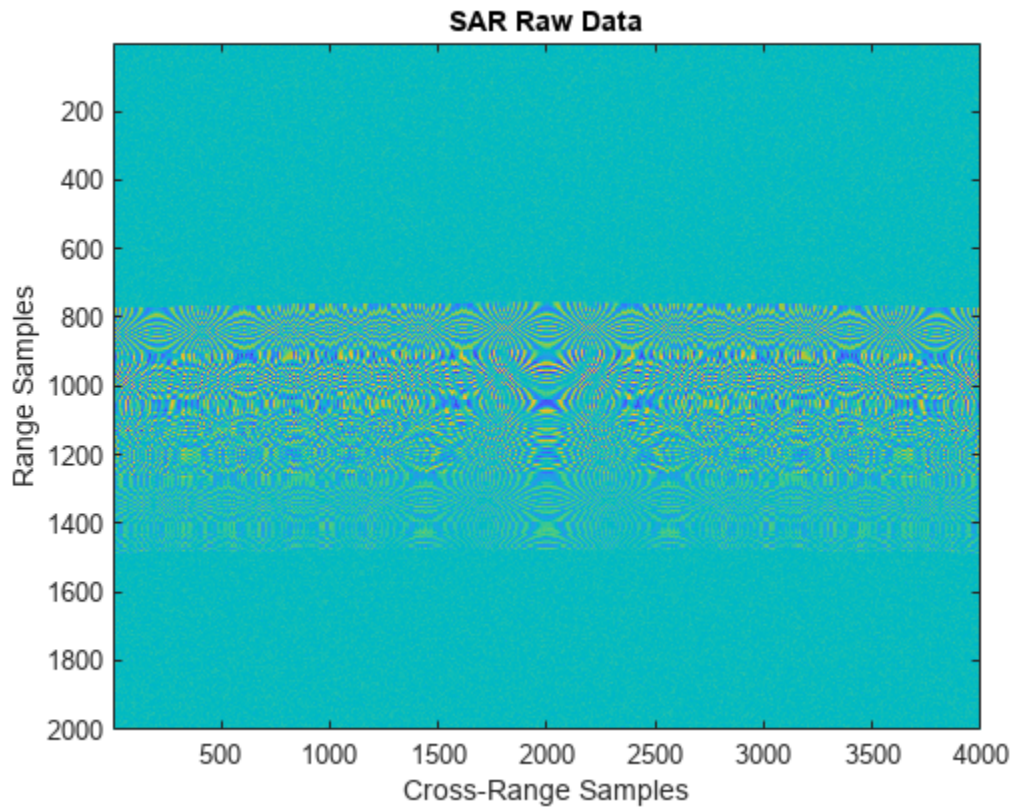
Perform range compression. Each row of the received signal, which contains all the information from each pulse, can be matched filtered to get the dechirped or range compressed signal.

```
pulseCompression = phased.RangeResponse('RangeMethod', 'Matched filter', 'PropagationSpeed', c,
matchingCoeff = getMatchedFilter(waveform);
[cdata, rnggrid] = pulseCompression(rxsig, matchingCoeff);
```

This figure shows the response after matched filtering has been performed on the received signal. The phase histories of the three targets are clearly visible along the cross-range direction and range focusing has been achieved.

```
imagesc(real(cdata));title('SAR Range Compressed Data')
xlabel('Cross-Range Samples')
ylabel('Range Samples')
```

SAR Range Compressed Data
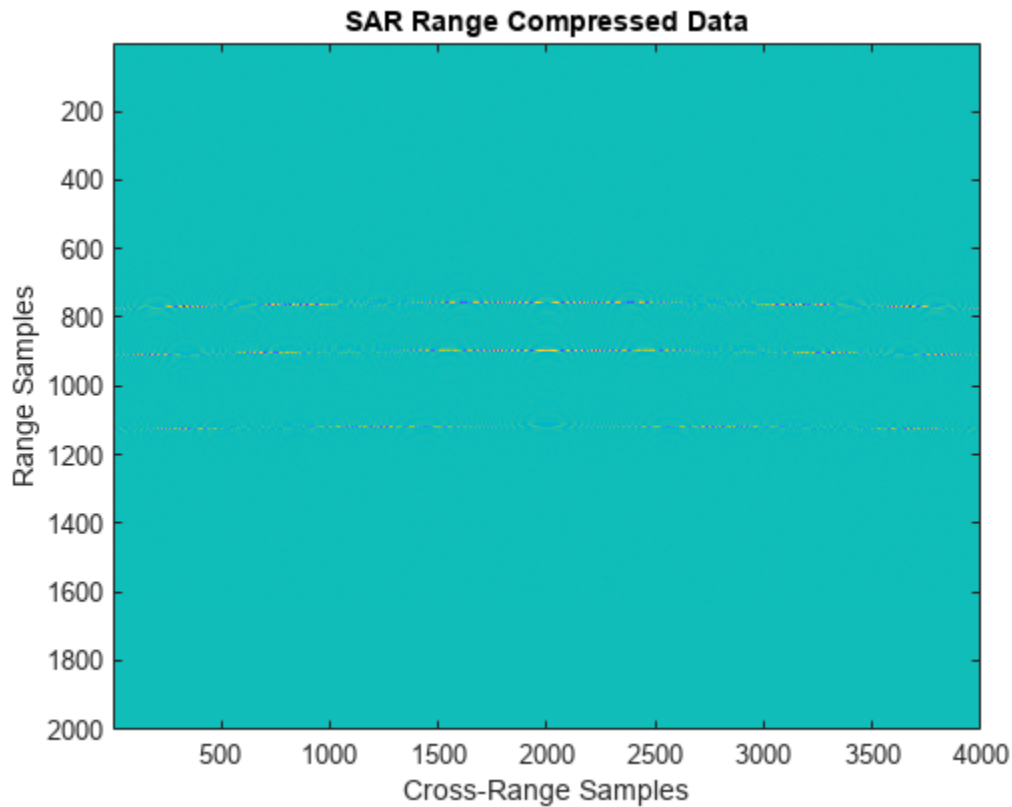
Perform azimuth Compression. There are multiple techniques to process the cross-range data and get the final image from the SAR raw data once range compression has been achieved. In essence, range compression helps achieve resolution in the fast-time or range direction and the resolution in the cross-range direction is achieved by azimuth or cross-range compression. Two such techniques are the range migration algorithm and the back-projection algorithm, which are demonstrated in this example.

```
rma_processed = helperRangeMigration(cdata,fastTime,fc,fs,prf,speed,numpulses,c,Rc);
bpa_processed = helperBackProjection(cdata,rnggrid,fastTime,fc,fs,prf,speed,crossRangeResolution
```

Visualize the final SAR image. Plot the focused SAR image using the range migration algorithm and the approximate back projection algorithm. Only a section of the image formed via the range migration algorithm is shown to accurately point the location of the targets.

The range migration and accurate form of the backprojection algorithm as shown by [2] and [3] provides theoretical resolution in the cross-track as well as along-track direction. Since the back-projection used here is of the approximate form, the spread in azimuth direction is evident in case of the back-projection, whereas the data processed via range migration algorithm shows that theoretical resolution is achieved.

```
figure(1);
imagesc((abs((rma_processed(1700:2300,600:1400).')))));
title('SAR Data focused using Range Migration algorithm ')
xlabel('Cross-Range Samples')
ylabel('Range Samples')
```

SAR Data focused using Range Migration algorithm
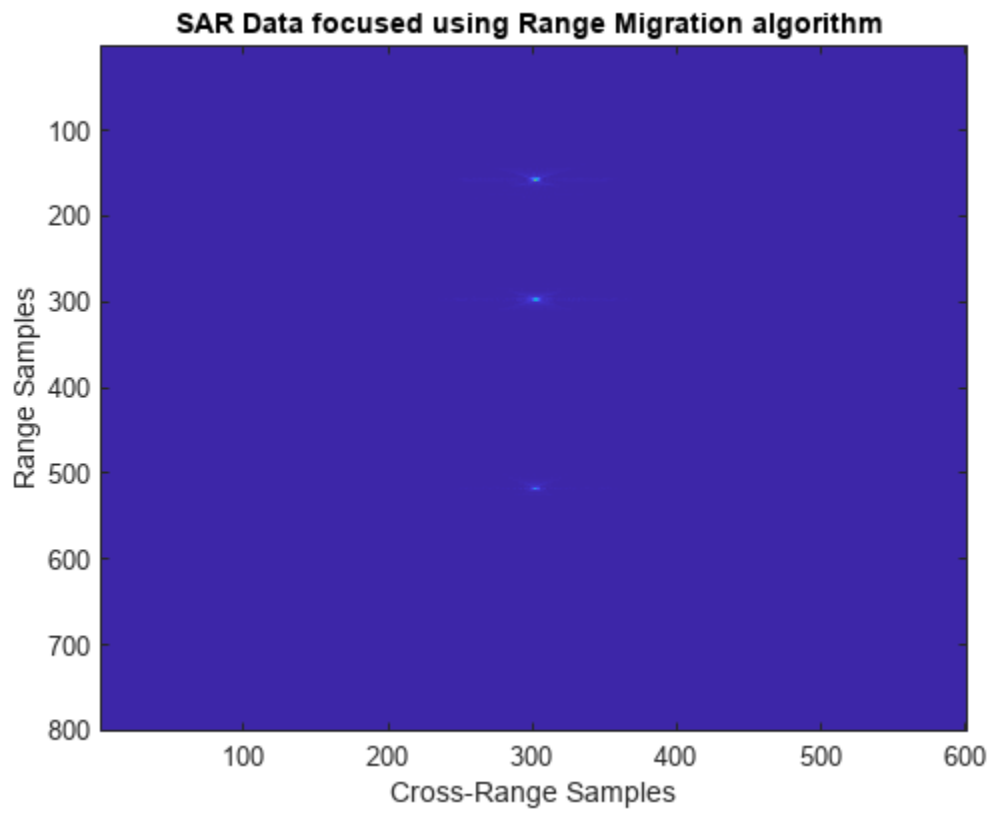

```
figure(2)
imagesc((abs(bpa_processed(600:1400,1700:2300))));
title('SAR Data focused using Back-Projection algorithm ')
xlabel('Cross-Range Samples')
ylabel('Range Samples')
```

SAR Data focused using Back-Projection algorithm
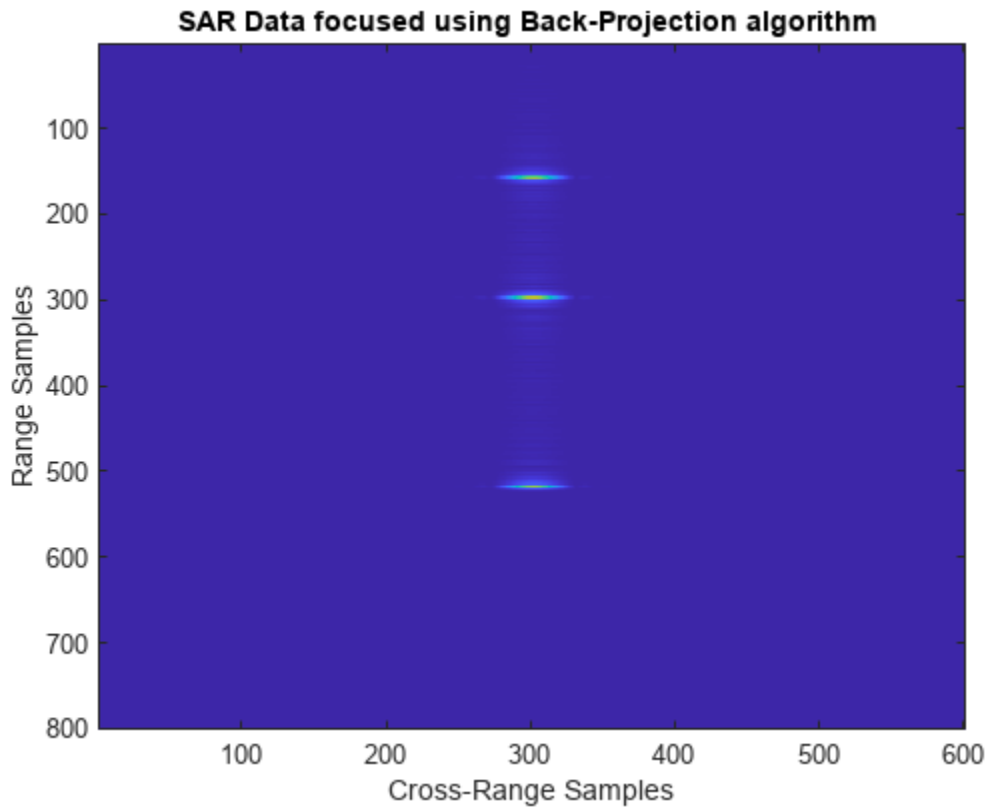
Summary

This example shows how to develop SAR processing leveraging a LFM signal in an airborne data collection scenario. The example also shows how to generate an image from the received signal via range migration and approximate form of the back-projection algorithms.

References

1   Cafforio, C., C. Prati, and F. Rocca. "SAR Data Focusing Using Seismic Migration Techniques." *IEEE Transactions on Aerospace and Electronic Systems* 27, no. 2 (March 1991): 194–207. https://doi.org/10.1109/7.78293.

2   Cumming, I., and J. Bennett. "Digital Processing of Seasat SAR Data." In *ICASSP '79. IEEE International Conference on Acoustics, Speech, and Signal Processing*, 4:710–18. Washington, DC, USA: Institute of Electrical and Electronics Engineers, 1979. https://doi.org/10.1109/ICASSP.1979.1170630.

3   Yibo Na, Yilong Lu, and Hongbo Sun. "A Comparison of Back-Projection and Range Migration Algorithms for Ultra-Wideband SAR Imaging." In *Fourth IEEE Workshop on Sensor Array and Multichannel Processing, 2006.*, 320–24. Waltham, Massachusetts, USA: IEEE, 2006. https://doi.org/10.1109/SAM.2006.1706146.

4   Yegulalp, A. F. "Fast Backprojection Algorithm for Synthetic Aperture Radar." In *Proceedings of the 1999 IEEE Radar Conference. Radar into the Next Millennium (Cat. No.99CH36249)*, 60–65. Waltham, MA, USA: IEEE, 1999. https://doi.org/10.1109/NRC.1999.767270.

Appendix

Range Migration Algorithm

```
function azcompresseddata = helperRangeMigration(sigData,fastTime,fc,fs,prf,speed,numPulses,c,Rc
```

This function demonstrates the range migration algorithm for imaging the side-looking synthetic aperture radar. The pulse compressed synthetic aperture data is considered in this algorithm.

Set the range frequency span.

```
frequencyRange = linspace(fc-fs/2,fc+fs/2,length(fastTime));
krange = 2*(2*pi*frequencyRange)/c;
```

Set the cross-range wavenumber.

```
kaz = 2*pi*linspace(-prf/2,prf/2,numPulses)./speed;
```

Generate a matrix of the cross-range wavenumbers to match the size of the received two-dimensional SAR signal.

```
kazimuth = kaz.';
kx = krange.^2-kazimuth.^2;
```

Set the final wavenumber to achieve azimuth focusing.

```
kx = sqrt(kx.*(kx > 0));
kFinal = exp(1i*kx.*Rc);
```

Perform a two-dimensional FFT on the range compressed signal.

```
sdata =fftshift(fft(fftshift(fft(sigData,[],1),1),[],2),2);
```

Perform bulk compression to get the azimuth compression at the reference range. Perform filtering of the 2-D FFT signal with the new cross-range wavenumber to achieve complete focusing at the reference range and as a by-product, partial focusing of targets not lying at the reference range.

```
fsmPol = (sdata.').*kFinal;
```

Perform Stolt interpolation to achieve focusing for targets that are not lying at the reference range.

```
stoltPol = fsmPol;
for i = 1:size((fsmPol),1)
    stoltPol(i,:) = interp1(kx(i,:),fsmPol(i,:),krange(1,:));
end
stoltPol(isnan(stoltPol)) = 1e-30;
stoltPol = stoltPol.*exp(-1i*krange.*Rc);
azcompresseddata = ifft2(stoltPol);
end
```

Back-Projection Algorithm

```
function data = helperBackProjection(sigdata,rnggrid,fastTime,fc,fs,prf,speed,crossRangeResolutio
```

This function demonstrates the time-domain back projection algorithm for imaging the side-looking synthetic aperture radar. The pulsed compressed synthetic aperture data is taken as input in this algorithm. Initialize the output matrix.

```
data = zeros(size(sigdata));
azimuthDist = -200:speed/prf:200;%azimuth distance
```

Limit the range and cross-range pixels being processed to reduce processing time.

```
rangelims = [700 1400];
crossrangelims = [-10 10];
```

Index the range grid in accordance with the range limits.

```
rangeIdx =  [find(rnggrid>rangelims(1), 1) find(rnggrid<rangelims(2),1,'last')];
```

Index the azimuth distance in accordance with the cross-range limits.

```
crossrangeIdxStart = find(azimuthDist>crossrangelims(1),1);
crossrangeIdxStop = find(azimuthDist<crossrangelims(2),1,'last');
for i= rangeIdx(1):rangeIdx(2)  % Iterate over the range indices

    % Using desired cross-range resolution, compute the synthetic aperture
    % length
    lsynth= (c/fc)* (c*fastTime(i)/2)/(2*crossRangeResolution);
    lsar = round(lsynth*length(azimuthDist)/azimuthDist(end)) ;
    % Ensure lsar is an odd number
    lsar = lsar + mod(lsar,2);

    % Construct hanning window for cross-range processing, to suppress the
    % azimuthal side lobes
    hn= hanning(lsar).';
    % Iterate over the cross-range indices
    for j= crossrangeIdxStart:crossrangeIdxStop
        % azimuth distance in x direction over cross-range indices
        posx= azimuthDist(j);
        % range in y-direction over range indices
        posy= c*fastTime(i)/2;
        % initializing count to zero
        count= 0;
        % Iterate over the synthetic aperture
        for k= j-lsar/2 +1:j+ lsar/2
            % Time delay for each of range and cross-range indices
            td= sqrt((azimuthDist(k)- posx)^2 + posy^2)*2/c;
            cell= round(td*fs) +1 ;
            signal = sigdata(cell,k);
            count= count + hn(k -(j-lsar/2))*signal *exp(1j*2*pi*fc*(td));
        end
        % Processed data at each of range and cross-range indices
        data(i,j)= count;
    end

end
end
```

# Squinted Spotlight Synthetic Aperture Radar (SAR) Image Formation

This example shows how to model a spotlight-based Synthetic Aperture Radar (SAR) system using a Linear FM (LFM) Waveform. In case of squint mode, the SAR platform is squinted either to look forward or backward by certain angle from broadside depending upon the need. The squint mode helps in imaging regions lying ahead of the current radar platform location or to image locations lying behind the platform location for interferometric applications. The challenge in squint mode is higher due to the range azimuth coupling. Because the resolution of SAR depends upon the signal and antenna configuration, resolution can be much higher than for vision-based imaging systems. Using spotlight mode, this example performs range migration algorithm [1,3] to image stationary targets lying ahead of the SAR platform location. A Linear FM waveform offers the advantage of large time-bandwidth product at a considerably lower transmit power making it suitable for use in airborne systems. For more details on the terminology used in this example, see "Stripmap Synthetic Aperture Radar (SAR) Image Formation" on page 1-322.
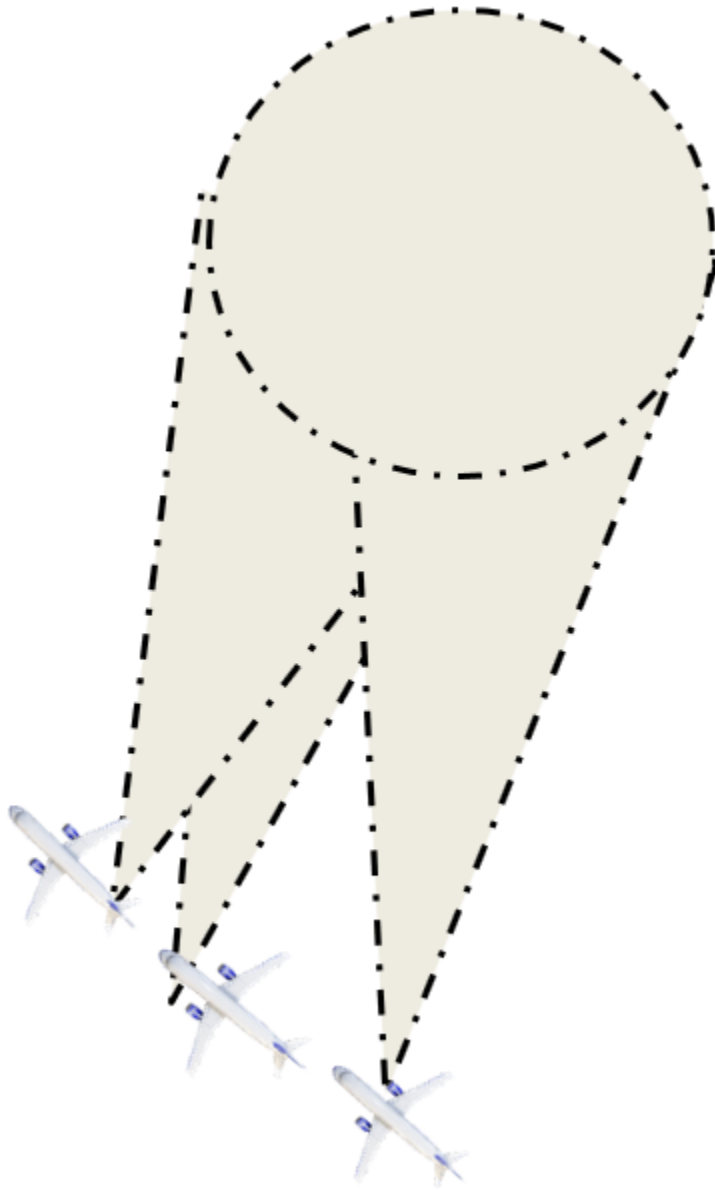
Radar Configuration

Consider an airborne SAR operating in C-band with a 4 GHz carrier frequency and a signal bandwidth of 50 MHz. This bandwidth yields a range resolution of 3 meters. The radar system collects data at a squint angle of 33 degrees from the broadside as shown in the figure above. The delay corresponds in general to the slant range between the target and the platform. For a SAR system, the slant range will vary over time as the platform traverses a path orthogonal to the direction of antenna beam. This section below focuses on defining the parameters for the transmission waveform. The LFM sweep bandwidth can be decided based on the desired range resolution.

```
c = physconst('LightSpeed');
```

Set the SAR center frequency.

```
fc = 4e9;% Hz
```

Set the desired range and cross-range resolution to 3 meters.

```
rangeResolution = 3;% meters
crossRangeResolution = 3;% meters
```

The signal bandwidth is a parameter derived from the desired range resolution.

```
bw = c/(2*rangeResolution);

prf = 1000;% Hz
aperture = 4;% sq. meters
tpd = 3*10^-6; % sec
fs = 120*10^6; % Hz
```

Configure the LFM signal of the radar.

```
waveform = phased.LinearFMWaveform('SampleRate',fs, 'PulseWidth', tpd, 'PRF', prf,...
    'SweepBandwidth', bw);
```

Assume the speed of aircraft is 100 m/s with a flight duration of 4 seconds.

```
speed = 100;% m/s
flightDuration = 4;% sec
radarPlatform  = phased.Platform('InitialPosition', [0;-600;500], 'Velocity', [0; speed; 0]);
slowTime = 1/prf;
numpulses = flightDuration/slowTime +1;
eta1 = linspace(0,flightDuration ,numpulses)';

maxRange = 2500;
truncrangesamples = ceil((2*maxRange/c)*fs);
fastTime = (0:1/fs:(truncrangesamples-1)/fs);
% Set the reference range for the cross-range processing.
Rc = 1e3;% meters
```

Configure the SAR transmitter and receiver. The antenna looks in the broadside direction orthogonal to the flight direction.

```
antenna = phased.CosineAntennaElement('FrequencyRange', [1e9 6e9]);
antennaGain = aperture2gain(aperture,c/fc);

transmitter = phased.Transmitter('PeakPower', 1e3, 'Gain', antennaGain);
radiator = phased.Radiator('Sensor', antenna,'OperatingFrequency', fc, 'PropagationSpeed', c);

collector = phased.Collector('Sensor', antenna, 'PropagationSpeed', c,'OperatingFrequency', fc);
receiver = phased.ReceiverPreamp('SampleRate', fs, 'NoiseFigure', 30);
```

Configure the propagation channel.

```
channel = phased.FreeSpace('PropagationSpeed', c, 'OperatingFrequency', fc,'SampleRate', fs,...
    'TwoWayPropagation', true);
```

Scene Configuration

In this example, two static point targets are configured at locations specified below. The entire scene as shown further in the simulation lies ahead of the platform. The data collection ends before the airborne platform is abreast of the target location. All targets have a mean RCS value of 1 meter squared.

```
targetpos= [900,0,0;1000,-30,0]';

targetvel = [0,0,0;0,0,0]';
```

The squint angle calculation depends on the flight path and center of the target scene which is located at nearly 950 meters in this case.

```
squintangle = atand(600/950);
target = phased.RadarTarget('OperatingFrequency', fc, 'MeanRCS', [1,1]);
pointTargets = phased.Platform('InitialPosition', targetpos,'Velocity',targetvel);
% The figure below describes the ground truth based on the target
% locations.
figure(1);
h = axes;plot(targetpos(2,1),targetpos(1,1),'*b');hold all;plot(targetpos(2,2),targetpos(1,2),'*
set(h,'Ydir','reverse');xlim([-50 10]);ylim([800 1200]);
title('Ground Truth');ylabel('Range');xlabel('Cross-Range');
```
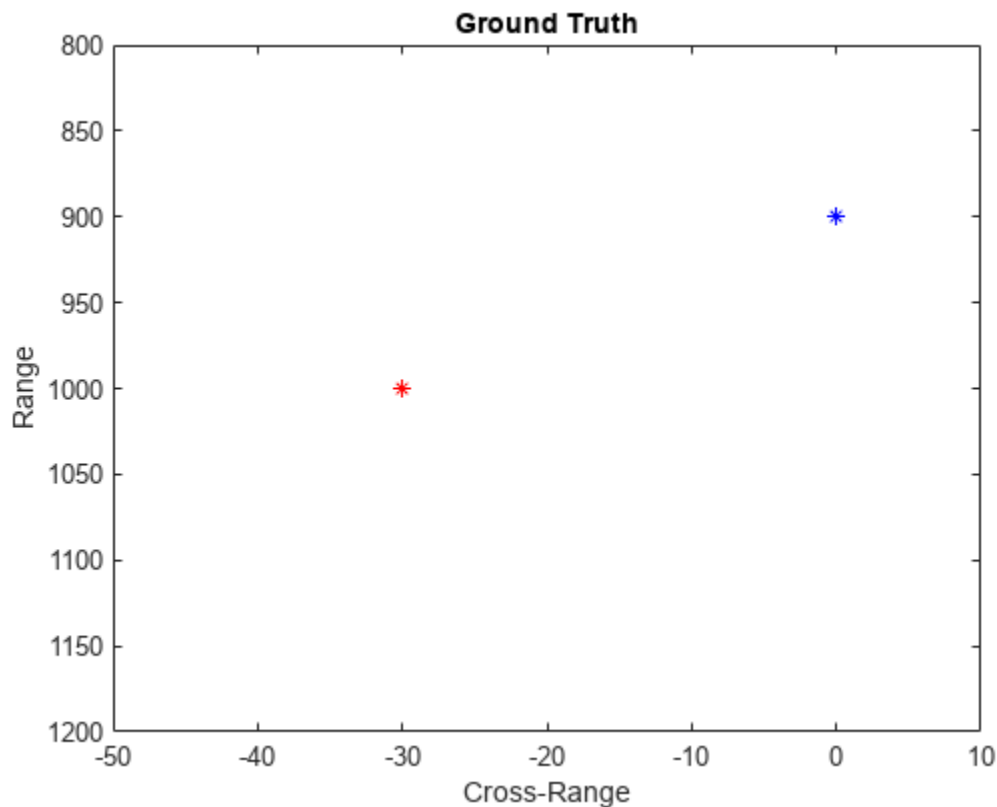

SAR Signal Simulation

The following section describes how the system operates based on the above configuration. Specifically, the section below shows how the data collection is performed for a SAR platform. As the platform moves in the cross-range direction, pulses are transmitted and received in directions defined by the squint angle with respect to the flight path. A collection of pulses gives the phase history of the targets lying in the illumination region as the platform moves. The longer the target lies in the illumination region, the better the cross-range resolution for the entire image because the process of range and cross-range focusing is generalized for the entire scene.

```
rxsig = zeros(truncrangesamples,numpulses);
for ii = 1:numpulses
    % Update radar platform and target position
    [radarpos, radarvel] = radarPlatform(slowTime);
    [targetpos,targetvel] = pointTargets(slowTime);

    % Get the range and angle to the point targets
    [targetRange, targetAngle] = rangeangle(targetpos, radarpos);

    % Generate the LFM pulse
    sig = waveform();
    % Use only the pulse length that will cover the targets.
    sig = sig(1:truncrangesamples);

    % Transmit the pulse
    sig = transmitter(sig);

    % Radiate the pulse towards the targets
    sig = radiator(sig, targetAngle);

    % Propagate the pulse to the point targets in free space
    sig = channel(sig, radarpos, targetpos, radarvel, targetvel);

    % Reflect the pulse off the targets
    sig = target(sig);

    % Collect the reflected pulses at the antenna
    sig = collector(sig, targetAngle);

    % Receive the signal
    rxsig(:,ii) = receiver(sig);

end

kc = (2*pi*fc)/c;
% Compensate for the doppler due to the squint angle
rxsig=rxsig.*exp(-1i.*2*(kc)*sin(deg2rad(squintangle))*repmat(speed*eta1,1,truncrangesamples)).'
```

Visualize the received signal.

The received signal can now be visualized as a collection of multiple pulses transmitted in the cross-range direction. The plots show the real part of the signal for the two targets. The chirps appear tilted due to the squint angle of the antenna.

```
imagesc(real(rxsig));title('SAR Raw Data')
xlabel('Cross-Range Samples')
ylabel('Range Samples')
```
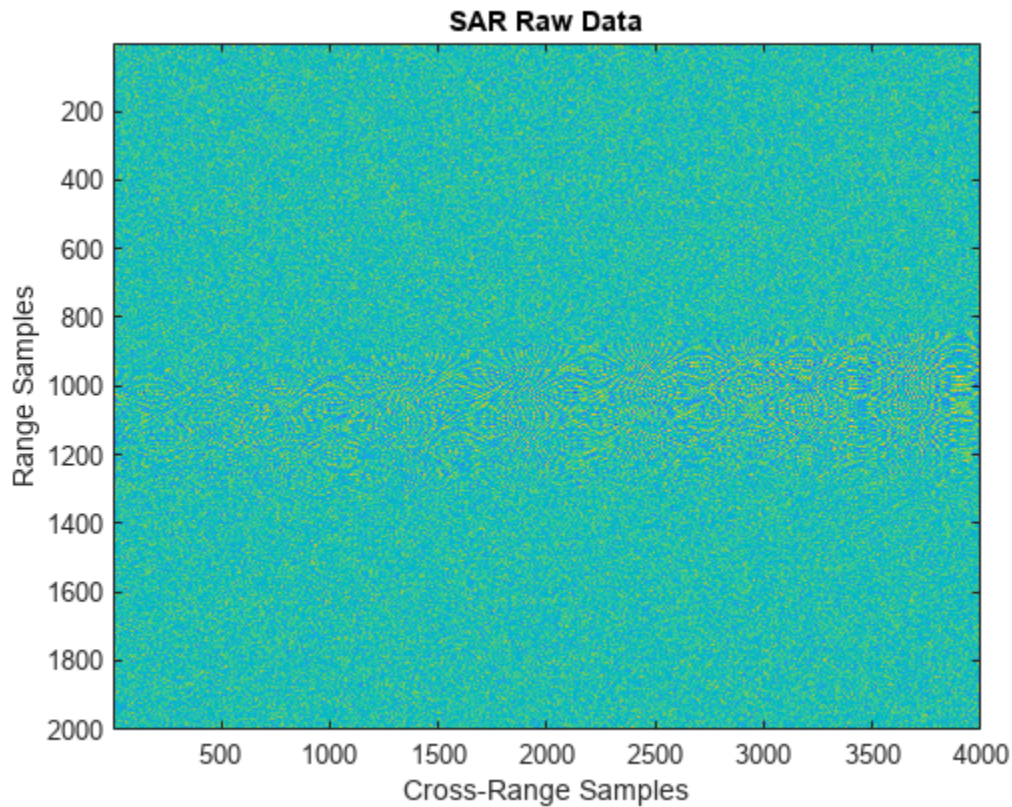
Perform range compression.

The range compression will help achieve the desired range resolution for a bandwidth of 50 MHz.

```
pulseCompression = phased.RangeResponse('RangeMethod', 'Matched filter', 'PropagationSpeed', c,
matchingCoeff = getMatchedFilter(waveform);
[cdata, rnggrid] = pulseCompression(rxsig, matchingCoeff);
```

The figure below shows the response after range compression has been achieved on the received signal. The phase histories of the two targets are clearly visible along the cross-range direction and range focusing has been achieved.

```
imagesc(real(cdata(800:1100,:)));title('SAR Range Compressed Data')
xlabel('Cross-Range Samples')
ylabel('Range Samples')
```
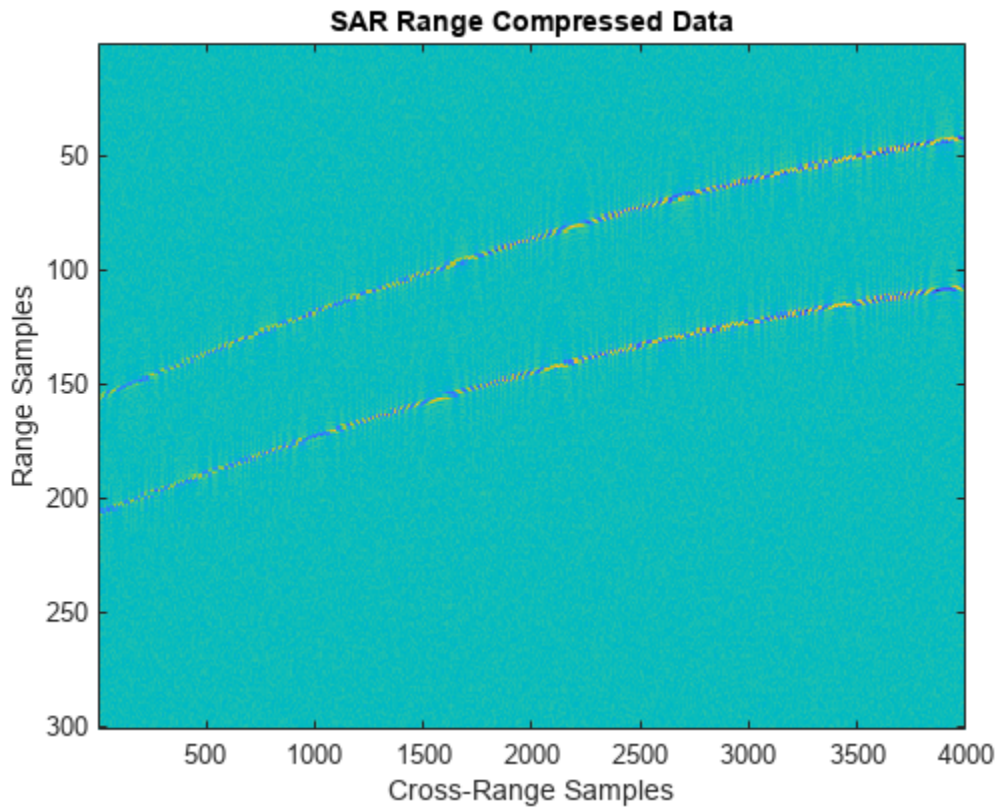
Azimuth Compression

There are multiple techniques to process the cross-range data and get the final image from the SAR raw data once range compression has been achieved. In essence, the range compression helps achieve resolution in the fast-time or range direction and the resolution in the cross-range direction is achieved by azimuth or cross-range compression. Range Migration algorithm for squint case has been demonstrated in this example. The azimuth focusing needs to account for the squint induced due to antenna tilt.

```
rma_processed = helperSquintRangeMigration(cdata,fastTime,fc,fs,prf,speed,numpulses,c,Rc,squintan
```

Visualize the final SAR image.

Plot the focused SAR image using the range migration algorithm. Only a section of the image formed via the range migration algorithm is shown to accurately point the location of the targets. The range migration as shown by [1], [2] and [3] provides theoretical resolution in the cross-track as well as along-track direction.

```
figure(2);
imagesc(abs(rma_processed(2300:3600,1100:1400).'));
title('SAR Data focused using Range Migration algorithm ')
xlabel('Cross-Range Samples')
ylabel('Range Samples')
```

SAR Data focused using Range Migration algorithm
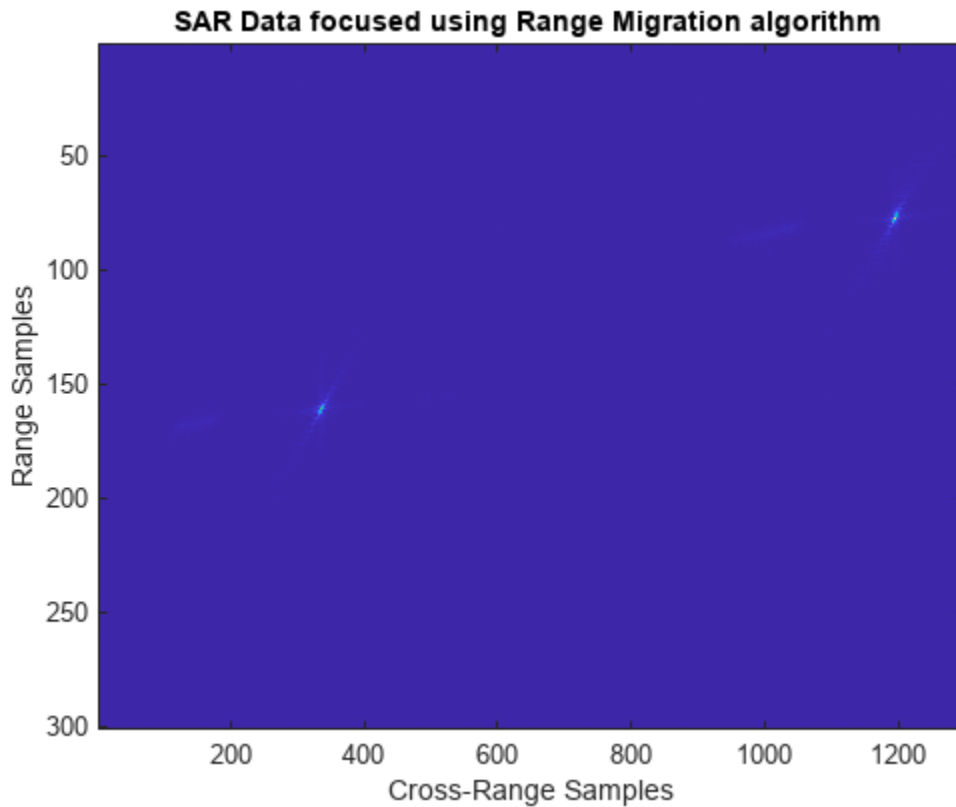
Summary

This example shows how to simulate and develop Squint mode Spotlight SAR processing leveraging a LFM signal in an airborne data collection scenario. The example also demonstrates image generation from the received signal via modified range migration algorithm to handle the effect due to squint.

References

1   Cafforio, C., C. Prati, and F. Rocca. "SAR data focusing using seismic migration techniques." *IEEE Transactions on Aerospace and Electronic Systems*, 27(2), 1991, pp.194–207. https://doi.org/10.1109/7.78293.

2   Soumekh, M. *Synthetic Aperture Radar Signal Processing with MATLAB Algorithms*. John Wiley & Sons, 1999.

3   Stolt, R. H. "Migration by Fourier Transform Techniques." *Geophysics*, 1978, 43, pp. 23–48.

Appendix

Range Migration Algorithm

```
function azcompresseddata = helperSquintRangeMigration(sigData,fastTime,fc,fs,prf,speed,numPulses
```

This function demonstrates the range migration algorithm for imaging the side looking synthetic aperture radar. The pulsed compressed synthetic aperture data is considered in this algorithm.

Set the range frequency span.

```
frequencyRange = linspace(fc-fs/2,fc+fs/2,length(fastTime));
krange = 2*(2*pi*frequencyRange)/c;
```

Set the cross-range wavenumber.

```
kaz = 2*pi*linspace(-prf/2,prf/2,numPulses)./speed;
```

Generate a matrix of the cross-range wavenumbers to match the size of the received two-dimensional SAR signal

```
kc = 2*pi*fc/3e8;
kazimuth = kaz.';
kus=2*(kc)*sin(deg2rad(squintangle));
kx = krange.^2-(kazimuth+kus).^2;
```

The wavenumber has been modified to accommodate shift due to squint and achieve azimuth focusing.

```
thetaRc = deg2rad(squintangle);
kx = sqrt(kx.*(kx > 0));

kFinal = exp(1i*(kx.*cos(thetaRc)+(kazimuth).*sin(thetaRc)).*Rc);
kfin = kx.*cos(thetaRc)+(kazimuth+kus).*sin(thetaRc);
```

Perform a two-dimensional FFT on the range compressed signal.

```
sdata =fftshift(fft(fftshift(fft(sigData,[],1),1),[],2),2);
```

Perform bulk compression to get the azimuth compression at the reference range. Perform filtering of the 2-D FFT signal with the new cross-range wavenumber to achieve complete focusing at the reference range and as a by-product, partial focusing of targets not lying at the reference range.

```
fsmPol = (sdata.').*kFinal;
```

Perform Stolt Interpolation to achieve focusing for targets that are not lying at the reference range

```
stoltPol = fsmPol;
for i = 1:size((fsmPol),1)
    stoltPol(i,:) = interp1(kfin(i,:),fsmPol(i,:),krange(1,:));
end
stoltPol(isnan(stoltPol)) = 1e-30;

azcompresseddata = ifftshift(ifft2(stoltPol),2);

end
```

Synthetic Aperture Radar System Simulation and Image Formation

This example demonstrates how to model a Synthetic Aperture RADAR (SAR) system using stepped frequency modulated (SFM) waveform and generate a SAR image in Simulink®. In this example, a SAR platform is defined along with the waveform it transmits. The receiver is modeled to handle the matched filtering and azimuth processing of different subpulses of the SFM waveform. For a similar example in MATLAB® using linear frequency modulated (LFM) waveform, see "Stripmap Synthetic Aperture Radar (SAR) Image Formation" on page 1-322.

Introduction

The model shows a SAR system setup to simulate IQ returns and perform image formation from the IQ data. The airborne platform in this example exploits the SFM waveform and its corresponding processing. The SFM waveform is an alternative technique to obtain a larger bandwidth, and it has some advantages over the conventional LFM waveform.

As the SFM constructs a wideband chirp from a burst of narrowband chirps, it provides improved noise figure and receiver sensitivity because of the narrow instantaneous bandwidth.

```
helperslexSARSystemSim('openModel');
```


Synthetic Aperture Radar (SAR) System Simulation and Image Formation

Copyright 2020-2022 The Mathworks Inc.
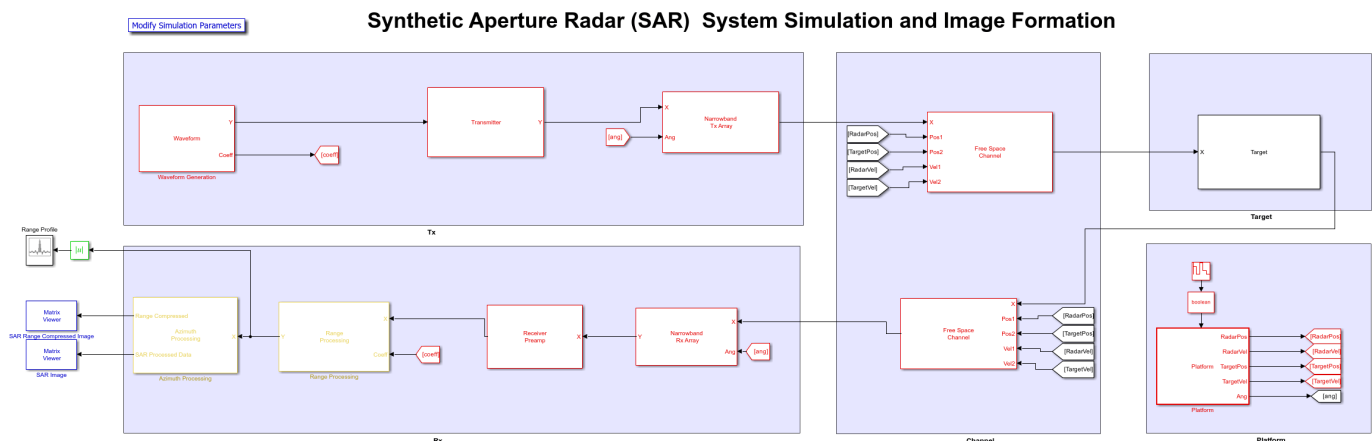
The model has various sections: Tx, Channel, Target, Platform, and Rx. The Tx section simulates the generation and transmission of the SFM waveform. The Channel section models the two-way propagation of the signal to and from the target. The Target section models three point targets located at 800 m, 1000 m, and 1300 m, each with a mean RCS of 1 m^2/m^2. The Platform subsystem models the flight path. The Rx section simulates the reception, range, and azimuth processing of the SAR raw data.

Waveform Generation

The Waveform Generation subsystem includes an SFM waveform generating module. The transmitted SFM burst consists of four subpulses with the PRF of 1000 Hz, such that it satisfies the criteria for maximum unambiguous range and maximum Doppler. The coefficients for matched filtering the IQ data at the receiver are also generated in this module.

```
helperslexSARSystemSim('showWaveform');
```
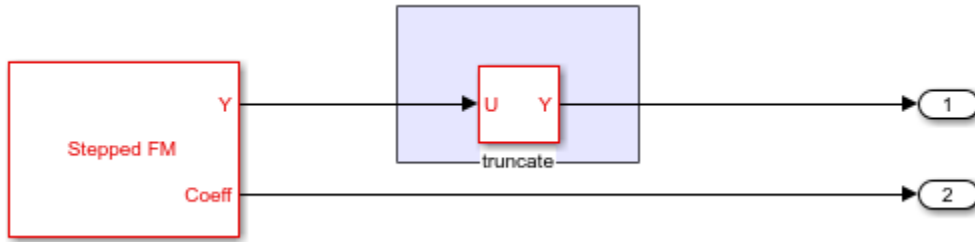
Platform

The Platform subsystem simulates the motion of aircraft on which the synthetic aperture radar is mounted. It is a triggered subsystem that mimics the stop-and-go assumption as the platform moves from position A to position B, transmitting and receiving the burst of pulses at each position. Hence, the platform remains at a position until all the subpulse in a burst are transmitted and received. The subpulses here implies the four steps of the SFM waveform transmitted and received at each position along the flight path. The platform is at a height of 500 m moving in the cross-range direction at a velocity of 100 m/s.

```
helperslexSARSystemSim('showPlatform');
```
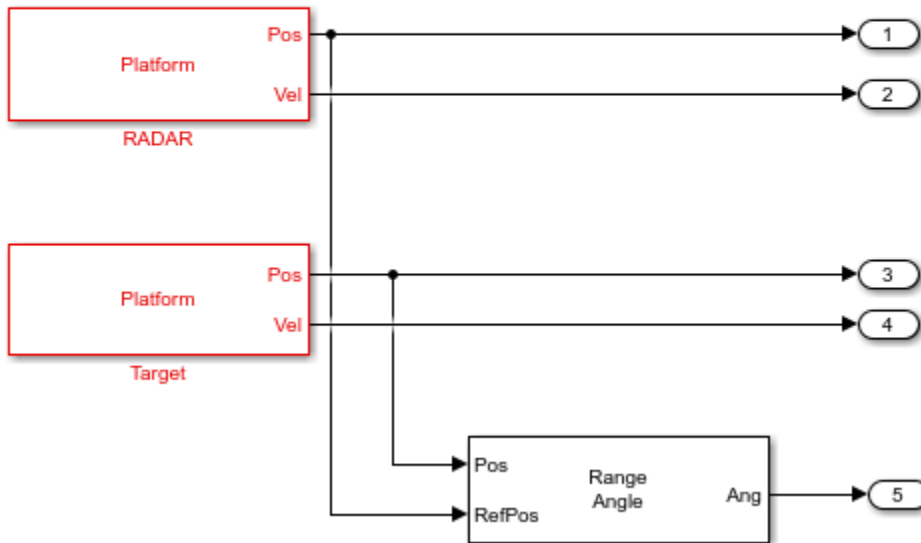
Range Processing

This is the first step in obtaining the SAR image. The Range Processing subsystem depicts the matched filtering for the waveform burst. The most convenient method for matched filtering the SFM waveform burst is the pulse-by-pulse processing. In pulse-by-pulse processing, first, each individual

subpulse in the burst is match filtered using the matched coefficients generated from the waveform generation subsystem. Secondly, the subpulses are integrated to complete the matched filtering of the entire burst. The pulse compressed subpulses are coherently integrated to obtain the range compressed data.

helperslexSARSystemSim('showRangeProcessing');
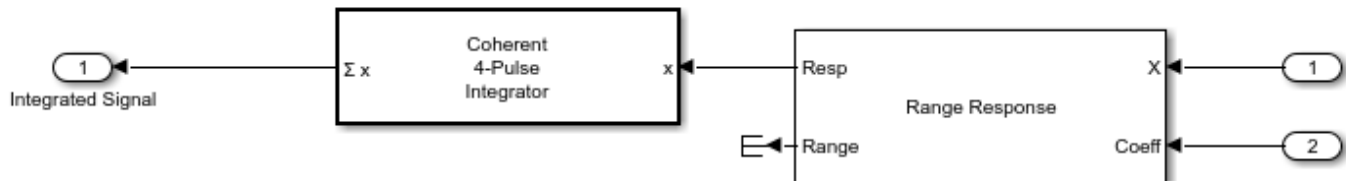
Azimuth Processing

The Azimuth Processing subsystem models the final step in obtaining the SAR image. This subsystem implements range migration algorithm (RMA) to focus the SAR image. The range compressed data is buffered to obtain data from every range bin before performing azimuth processing, which focuses the image in the cross-range or azimuth direction.

helperslexSARSystemSim('showAzimuthProcessing');
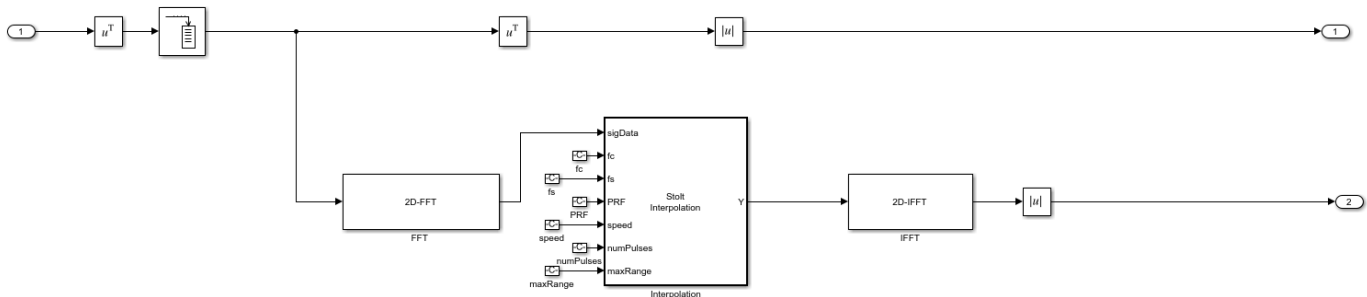
Exploring the Example

Several dialog parameters of the model are calculated by the helper function helperslexSARSystemParam. To open the function from the model, click the `Modify Simulation Parameters` block. This function is executed once the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialog boxes. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Display

The results show the processed data at different stages of the simulation.

helperslexSARSystemSim('runModel');

This image shows the data after range processing of the burst waveform. Run the helper function.

helperslexSARSystemSim('showRangeCompressed');
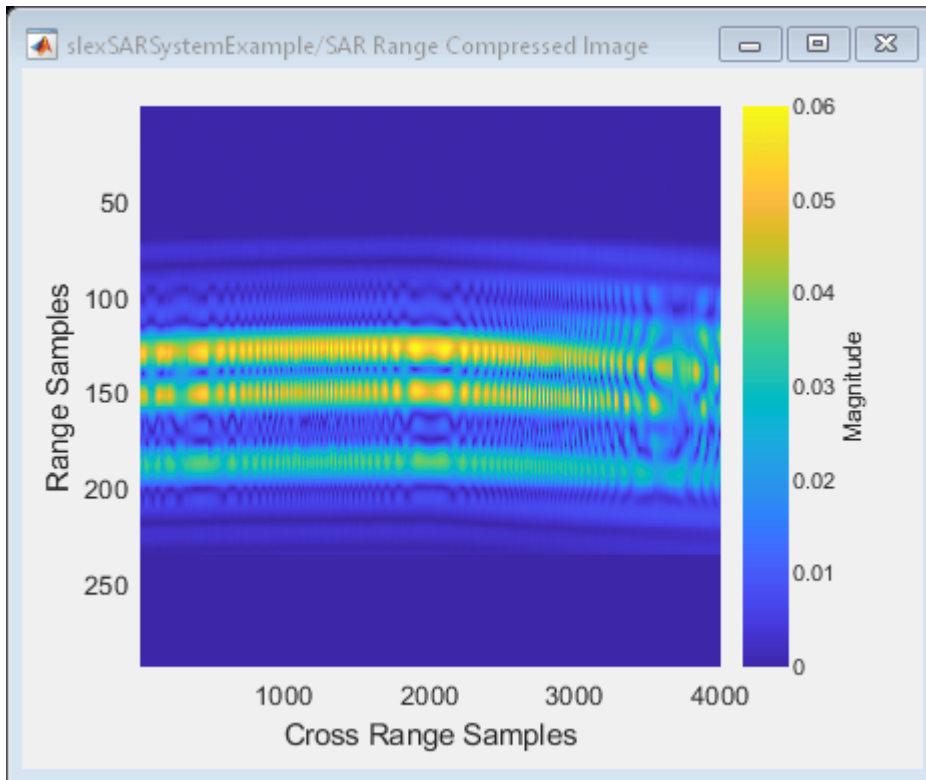
This image shows the SAR image after range and azimuth focusing.

```
helperslexSARSystemSim('showSARImage');
```
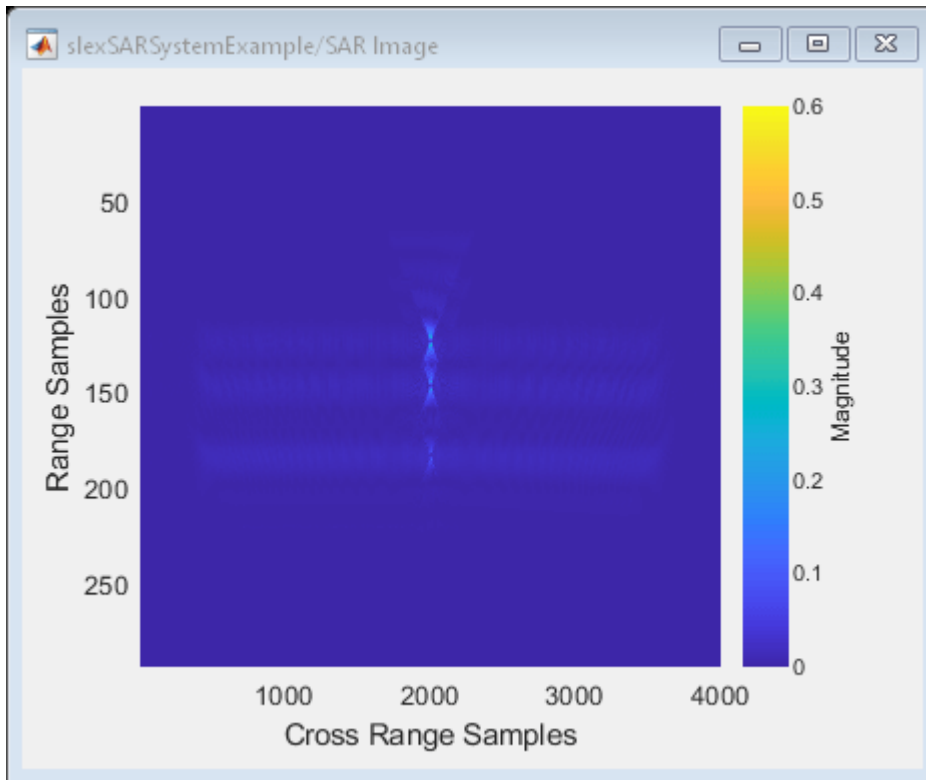
Summary

This example shows how to use an SFM waveform for a SAR imaging system mounted on an aircraft to image targets on the ground. The example models the entire system in Simulink, and shows how to use the SFM waveform and its corresponding processing.

```
helperslexSARSystemSim('closeModel');
```

Processing Radar Reflections Acquired with the Demorad Radar Sensor Platform

Introduction

This example demonstrates how to process and visualize frequency-modulated continuous-wave (FMCW) echoes acquired via the Demorad Radar Sensor Platform with the Phased Array System Toolbox™. By default, I/Q samples and operating parameters are read from a binary file that is provided with this example. Optionally, the same procedure can be used to transmit, receive, and process FMCW reflections from live I/Q samples with your own Demorad by following the instructions later in the example. Acquiring and processing results in one environment decreases development time and facilitates the rapid prototyping of radar signal processing systems.

Required Hardware and Software

This example requires Phased Array System Toolbox™.

For live processing:

- Optionally, Analog Devices® Demorad Radar Sensor Platform (and drivers)
- Optionally, Phased Array System Toolbox™ Add-On for Demorad

The Analog Devices® Demorad Radar Sensor Platform has an operating frequency of 24 GHz and a maximum bandwidth of 250 MHz. The array on the platform includes 2 transmit elements and 4 receive elements. The receive elements are spaced every half-wavelength of the operating frequency, arranged as a linear array. The transmit elements are also arranged as a linear array and are spaced three half-wavelengths apart. The transmit and receive elements can be seen in the image below.
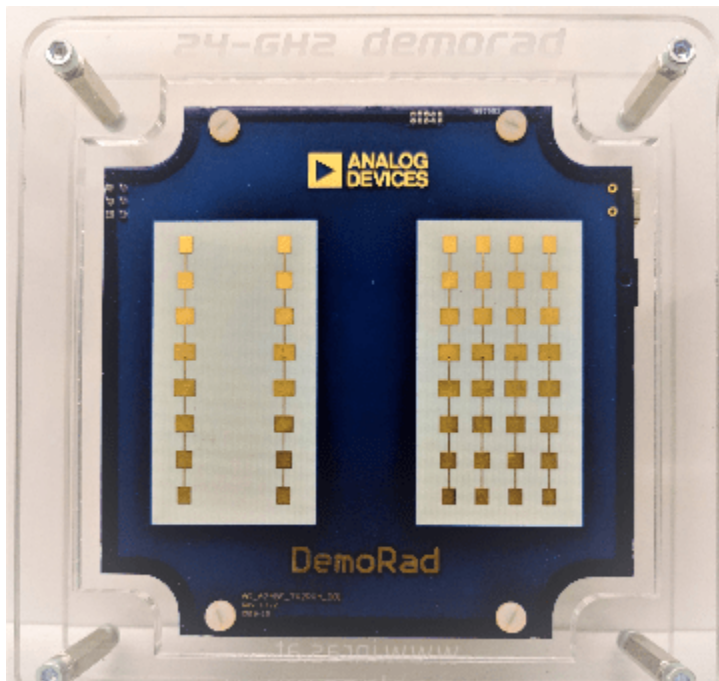
Installing the Drivers and Add-on

1. Download the Demorad drivers and MATLAB files from the USB-drive provided by Analog Devices® with the Demorad
2. Save these to a permanent location on your PC
3. Add the folder containing the MATLAB files to the MATLAB path permanently
4. Power up the Demorad and connect to the PC via the Mini-USB port
5. Navigate to the **Device Manager** and look for the "BF707 Bulk Device"
6. Right-click on "BF707 Bulk Device" and select "Update driver"
7. Select the option "Browse my computer for driver software"
8. Browse to and select the "drivers" folder from the USB-drive
9. Install the Phased Array System Toolbox™ Add-On for Demorad from the MATLAB Add-On Manager

Radar Setup and Connection

In this section, we setup the source of the I/Q samples as the default option of the binary file reader. Also contained in this file are the parameters that define the transmitted FMCW chirp, which were written at the time the file was created. If you would like to run this example using live echoes from the Demorad, follow the steps in **Installing the Drivers and Add-On** on page 1-347 and set the "usingHW" flag below to "true". The example will then communicate with the Demorad to transmit an FMCW waveform with the radar operating parameters defined below and send the reflections to MATLAB. The "setup" method below is defined by the object representing the Demorad. "setup" serves both to power on and send parameters to the Demorad.

```matlab
usingHW = false;  % Set to "true" to use Demorad Radar Sensor Platform

if ~usingHW
  % Read I/Q samples from a recorded binary file
  radarSource = RadarBasebandFileReader('./DemoradExampleIQData.bb',256);
else
  % Instantiate the Demorad Platform interface
  radarSource = DemoradBoard;

  radarSource.SamplesPerFrame = 256;   % Number of samples per frame
  radarSource.AcquisitionTime = 30;    % Length of time to acquire samples (s)

  % Define operating parameters
  radarSource.RampTime = 280e-6;       % Pulse ramp time (s)
  radarSource.PRI = 300e-6;            % Pulse Repetition Interval (s)
  radarSource.StartFrequency = 24e9;   % Pulse start frequency (Hz)
  radarSource.StopFrequency = 24.25e9; % Pulse stop frequency (Hz)
  radarSource.NumChirps = 1;           % Number of pulses to process

  % Establish the connection with the Demorad
  setup(radarSource);
end
```

Radar Capabilities

Based on the operating parameters defined above, the characteristics of the radar system can be defined for processing and visualizing the reflections. The equations used for calculating the capabilities of the radar system with these operating parameters can be seen below:

Range Resolution

The range resolution (in meters) for a radar with a chirp waveform is defined by the equation

$$\Delta R = \frac{c_0}{2B},$$

where B is the bandwidth of the transmitted pulse in Hz:

```
c0 = physconst('LightSpeed');
wfMetadata = radarSource.Metadata;              % Struct of waveform metadata
bandwidth = wfMetadata.StopFrequency ...
  - wfMetadata.StartFrequency;                  % Chirp bandwidth (Hz)
rangeRes = c0/(2*bandwidth)                      % Range resolution (m)
```

```
rangeRes = 0.5996
```

Maximum Range

The Demorad platform transmits an FMCW pulse as a chirp, or sawtooth waveform. As such, the theoretical maximum range of the radar (in meters) can be calculated using

$$R_{max} = \frac{F_s c_0}{2k_f},$$

where k_f is the chirp rate. The effective range in practice may vary due to environmental factors such as signal-to-noise ratio (SNR), interference, or size of the testing facility.

```
kf = bandwidth/wfMetadata.RampTime;       % Chirp rate
maxRange = radarSource.SampleRate*c0/(2*kf)  % Maximum range (m)
```

```
maxRange = 158.2904
```

Beamwidth

The effective beamwidth of the radar board can be approximated by the equation

$$\Theta \approx \frac{\lambda}{N_{elements}\Delta x_{elements}},$$

where λ is the wavelength of the pulse and $\delta x_{elements}$ is the element spacing.

```
lambda = c0/radarSource.CenterFrequency;                % Signal wavelength (m)
rxElementSpacing = lambda/2;
beamwidth = rad2deg(lambda/ ...
  (radarSource.NumChannels*rxElementSpacing))           % Effective beamwidth (deg)
```

```
beamwidth = 28.6479
```

With a transmit bandwidth of 250 MHz and a 4-element receive array, the range and angular resolution are sufficient to resolve multiple closely spaced objects. The I/Q samples recorded in the binary file are returned from the Demorad platform without any additional digital processing. FMCW reflections received by the Demorad are down-converted to baseband in hardware, decimated, and transferred to MATLAB.

Signal Processing Components

The algorithms used in the signal processing loop are initialized in this section. After receiving the I/Q samples, a 3-pulse canceller removes detections from stationary objects. The output of the 3-pulse canceller is then beamformed and used to calculate the range response. A CFAR detector is used following the range response algorithm to detect any moving targets.

3-Pulse Canceller

The 3-pulse canceller used following the acquisition of the I/Q samples removes any stationary clutter in the environment. The impulse response of a 3-pulse canceller is given as

$$h(t) = -\delta(t) + 2\delta(t - T_s) - \delta(t - 2T_s)$$

This equation is implemented in the pulse canceller algorithm defined below.

```
threePulseCanceller = PulseCanceller('NumPulses',3);
```

Range Response

The algorithms for calculating the range response are initialized below. For beamforming, the sensor array is modeled using the number of antenna elements and the spacing of the receive elements. The sensor array model and the operating frequency of the Demorad are required for the beamforming algorithm. Because the Demorad transmits an FMCW waveform, the range response is calculated using an FFT.

```
antennaArray = phased.ULA('NumElements',radarSource.NumChannels, ...
  'ElementSpacing',rxElementSpacing);

beamFormer = phased.PhaseShiftBeamformer('SensorArray',antennaArray, ...
'Direction',[0;0],'OperatingFrequency',radarSource.CenterFrequency);

% Setup the algorithm for processing
NFFT = 4096;
rangeResp = phased.RangeResponse( ...
  'DechirpInput', false, ...
  'RangeMethod','FFT', ...
  'ReferenceRangeCentered', false, ...
  'PropagationSpeed',c0, ...
  'SampleRate',radarSource.SampleRate, ...
  'SweepSlope',kf*2, ...
  'RangeFFTLengthSource','Property', ...
  'RangeFFTLength',NFFT, ...
  'RangeWindow', 'Hann');
```

CFAR Detector

A constant false alarm rate (CFAR) detector is then used to detect any moving targets.

```
cfar = phased.CFARDetector('NumGuardCells',6,'NumTrainingCells',10);
```

Scopes

Setup the scopes to view the processed FMCW reflections. We set the viewing window of the range-time intensity scope to 15 seconds.

```
timespan = 15;
```

```
% The Demorad returns data every 128 pulse repetition intervals
rangeScope = phased.RTIScope( ...
  'RangeResolution',maxRange/NFFT,...
  'TimeResolution',wfMetadata.PRI*128, ...
  'TimeSpan', timespan, ...
  'IntensityUnits','power');
```

Simulation and Visualization

Next, the samples are received from the binary file reader, processed, and shown in the scopes. This loop will continue until all samples are read from the binary file. If using the Demorad, the loop will continue for 30 seconds, defined by the "AcquisitionTime" property of the object that represents the board. Only ranges from 0 to 15 meters are shown, since we have a priori knowledge the target recorded in the binary file is within this range.

```
while ~isDone(radarSource)
  % Retrieve samples from the I/Q sample source
  x = radarSource();

  % Cancel out any pulses from non-moving objects and beamform
  y = threePulseCanceller(x);
  y = beamFormer(y);

  % Calculate the range response and convert to power
  resp = rangeResp(y);
  rangepow = abs(resp).^2;

  % Use the CFAR detector to detect any moving targets from 0 - 15 meters
  maxViewRange = 15;
  rng_grid = linspace(0,maxRange,NFFT).';
  [~,maxViewIdx] = min(abs(rng_grid - maxViewRange));
  detIdx = false(NFFT,1);
  detIdx(1:maxViewIdx) = cfar(rangepow,1:maxViewIdx);

  % Remove non-detections and set a noise floor at 1/10 of the peak value
  rangepow = rangepow./max(rangepow(:)); % Normalize detections
  noiseFloor = 1e-1;
  rangepow(~detIdx & (rangepow < noiseFloor)) = noiseFloor;

  % Display ranges from 0 - 15 meters in the range-time intensity scope
    rangeScope(rangepow(1:maxViewIdx));
end
```
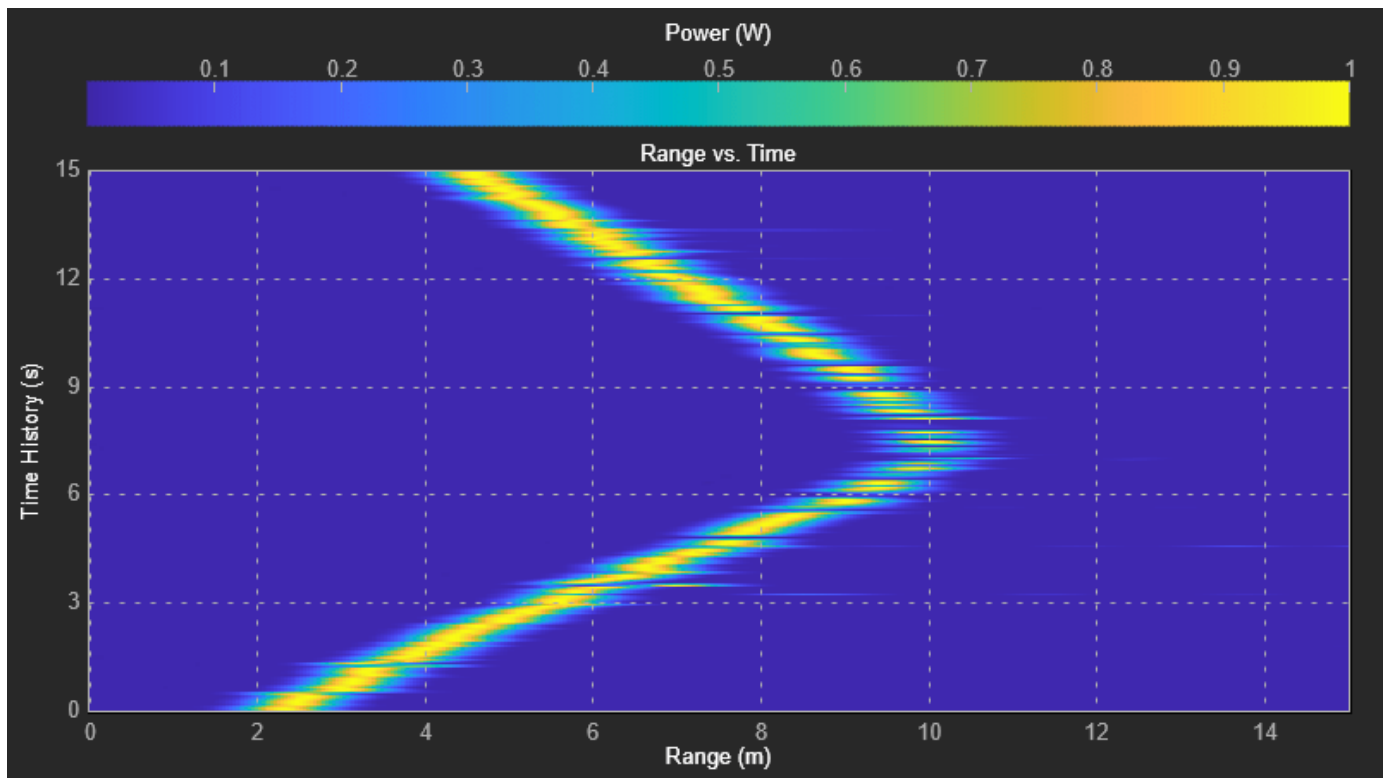
The scope shows a single target moving away from the Demorad Radar Sensor Platform until about ~10 meters away, then changing direction again to move back towards the platform. The range-time intensity scope shows the detection ranges.

Summary

This example demonstrates how to interface with the Analog Devices® Demorad Radar Sensor Platform to acquire, process, and visualize radar reflections from live data. This capability enables the rapid prototyping and testing of radar signal processing systems in a single environment, drastically decreasing development time.

Detector Performance Analysis Using ROC Curves

This example shows how you can assess the performance of both coherent and noncoherent systems using receiver operating characteristic (ROC) curves. The example assumes that the detector operates in an additive complex white Gaussian noise environment.

ROC curves are often used to assess the performance of a radar or sonar detector. ROC curves are plots of the probability of detection (Pd) versus the probability of false alarm (Pfa) for a given signal-to-noise ratio (SNR).

Introduction

Pd is the probability of saying that 1 is true given that event 1 occurred. Pfa is the probability of saying that 1 is true given that the 0 event occurred. In applications such as sonar and radar, the 1 event indicates that a target is present, and the 0 event indicates that a target is not present.

The performance of a detector is measured by its ability to achieve a certain Pd and Pfa for a given SNR. Examining the ROC curves provides insight into its performance. You can use the `rocsnr` function to calculate and plot the ROC curves.

Single Pulse Detection

Given an SNR value, you can calculate the Pd and Pfa values that a linear or square-law detector can achieve using a single pulse. Set the Pd value assuming that you have an SNR value of 8 dB and the requirements dictate a Pfa value of at most 1%. You can use the `rocsnr` function to calculate the Pd and Pfa values and then determine what value of Pd corresponds to Pfa = 0.01. Note that by default the `rocsnr` function assumes coherent detection.

```
[Pd,Pfa] = rocsnr(8);
idx = find(Pfa==0.01); % find index for Pfa=0.01
```

Using the index determined above you can find the Pd value that corresponds to Pfa = 0.01.

```
Pd(idx)
```

```
ans = 0.8899
```

One feature of the `rocsnr` function is that you can specify a vector of SNR values and the function calculates the ROC curve for each of these SNR values. Instead of individually calculating Pd and Pfa values for a given SNR, your can view the results in a plot of ROC curves. The `rocsnr` function plots the ROC curves by default if no output arguments are specified. Calling the `rocsnr` function with an input vector of four SNR values and no output arguments produces a plot of the ROC curves.

```
SNRvals = [2 4 8 9.4];
rocsnr(SNRvals);
```
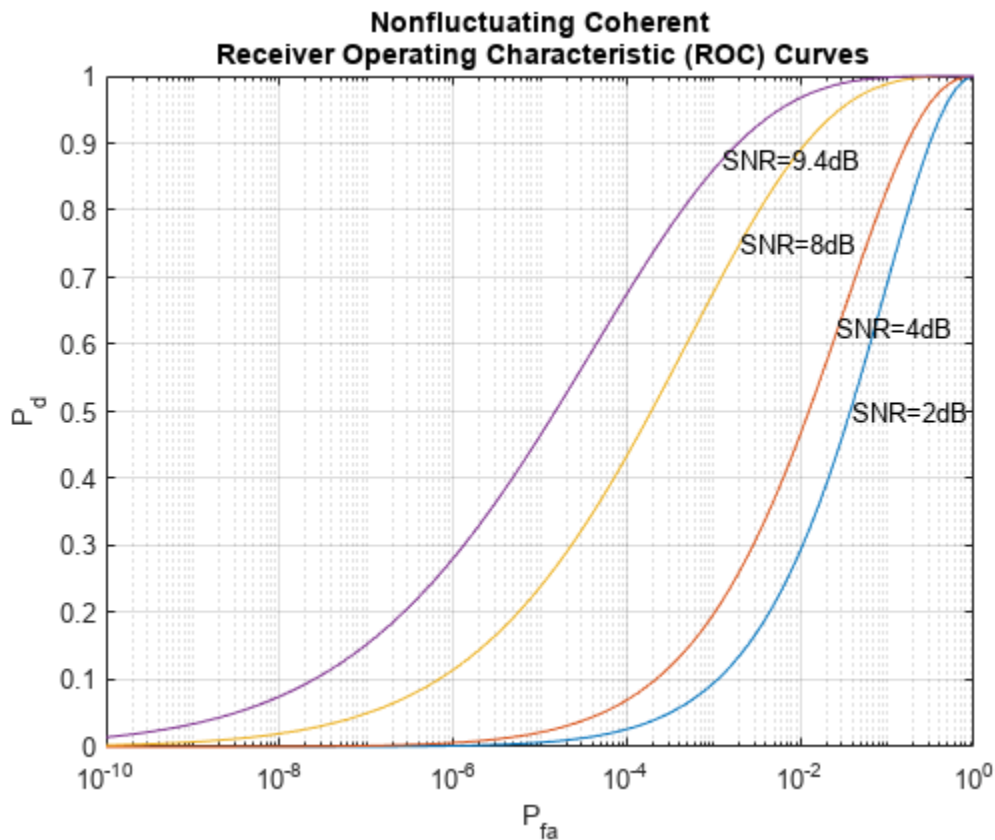
In the plot, select the data cursor button in the toolbar (or in the Tools menu) and then select the SNR = 8 dB curve at the point where Pd = 0.9 to verify that Pfa is approximately 0.01.

Multiple Pulse Detection

One way to improve detector performance is to average over several pulses. This is particularly useful in cases where the signal of interest is known and occurs in additive complex white noise. Although this still applies to both linear and square-law detectors, the result for square-law detectors could be off by about 0.2 dB. Analyze the performance by assuming an SNR of 8 dB and averaging over two pulses.

```
rocsnr(8,'NumPulses',2);
```
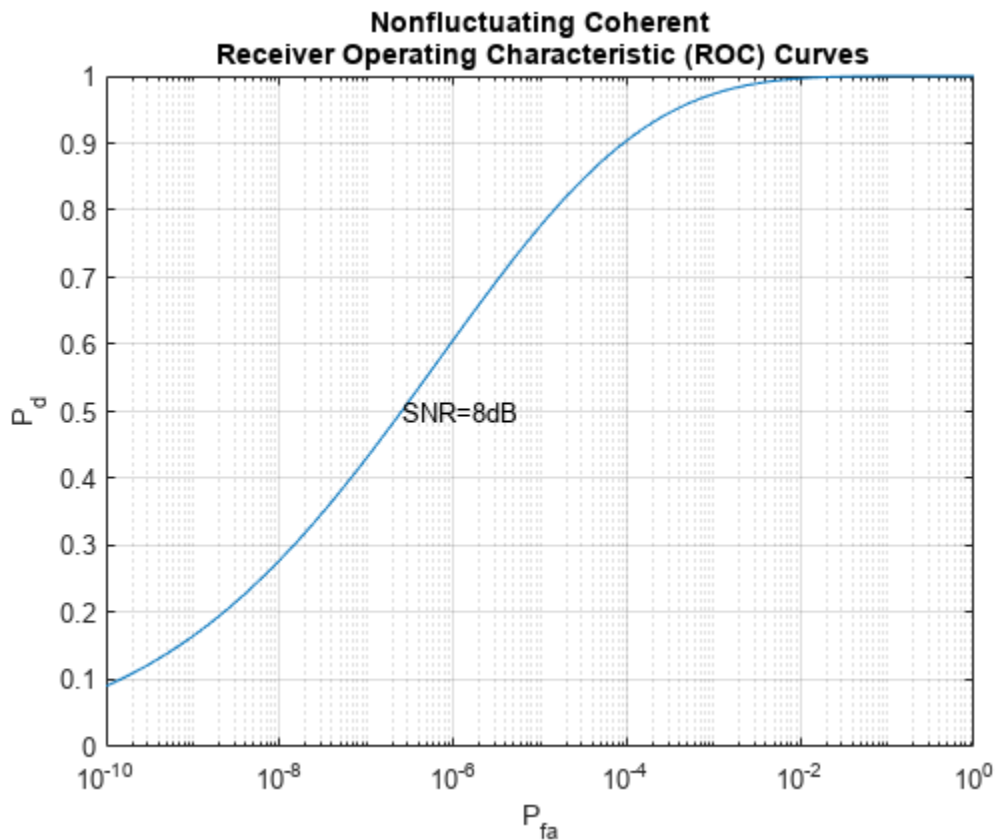
By inspecting the plot you can see that averaging over two pulses resulted in a higher probability of detection for a given false alarm rate. With an SNR of 8 dB and averaging over two pulses, you can constrain the probability of false alarm to be at most 0.0001 and achieve a probability of detection of 0.9. Recall that for a single pulse, you had to allow the probability of false alarm to be as much as 1% to achieve the same probability of detection.

Noncoherent Detector

To this point, the example deals with a known signal in complex white Gaussian noise. The `rocsnr` function by default assumes a coherent detector. To analyze the performance of a detector for the case where the signal is known except for the phase, you can specify a noncoherent detector. Using the same SNR values as before, analyze the performance of a noncoherent detector.

```
rocsnr(SNRvals,'SignalType','NonfluctuatingNoncoherent');
```
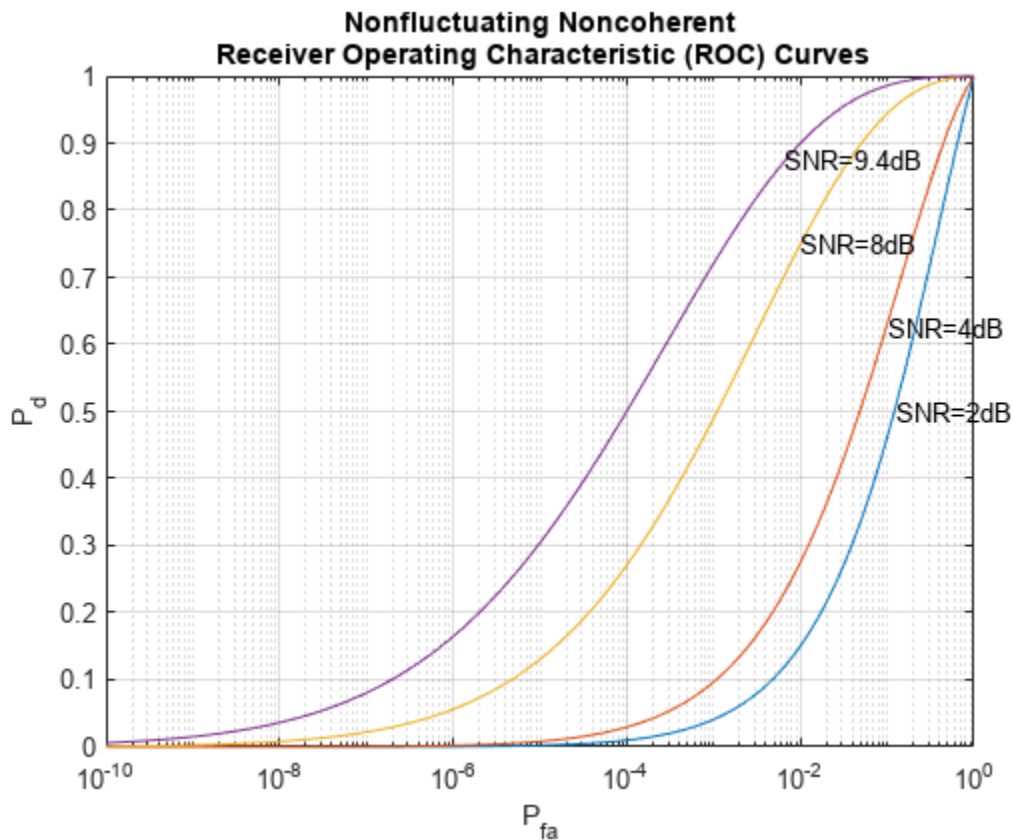
Focus on the ROC curve corresponding to an SNR of 8 dB. By inspecting the graph with the data cursor, you can see that to achieve a probability of detection of 0.9, you must tolerate a false-alarm probability of up to 0.05. Without using phase information, you will need a higher SNR to achieve the same Pd for a given Pfa. For noncoherent linear detectors, use Albersheim's equation to determine what value of SNR will achieve the desired Pd and Pfa.

```
SNR_valdB = albersheim(0.9,.01)   % Pd=0.9 and Pfa=0.01
```

SNR_valdB = 9.5027

Plotting the ROC curve for the SNR value approximated by Albersheim's equation, you can see that the detector will achieve Pd = 0.9 and Pfa = 0.01. Note that the Albersheim's technique applies only to noncoherent detectors.

```
rocsnr(SNR_valdB,'SignalType','NonfluctuatingNoncoherent');
```
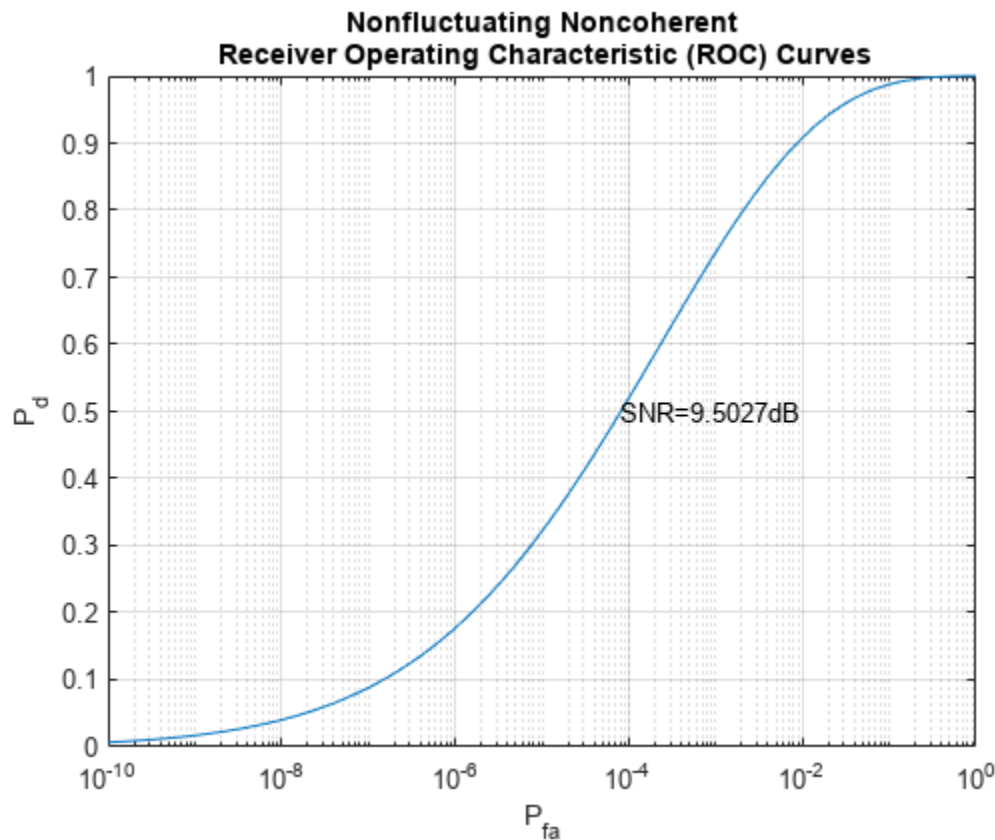
Detection of Fluctuating Targets

All the discussions above assume that the target is nonfluctuating, which means that the statistical characteristics of the target do not change over time. However, in real scenarios, targets can accelerate and decelerate as well as roll and pitch. These factors cause the radar cross section (RCS) of the target to vary over time. A set of statistical models called Swerling models are often used to describe the random variation in target RCS.

There are four Swerling models, namely Swerling 1 -- 4. The nonfluctuating target is often termed either Swerling 0 or Swerling 5. Each Swerling model describes how the RCS of a target varies over time and the probability distribution of the variation.

Because the target RCS is varying, the ROC curves for fluctuating targets are not the same as the nonfluctuating ones. In addition, because Swerling targets add random phase into the received signal, it is harder to use a coherent detector for a Swerling target. Therefore, noncoherent detection techniques are often used for Swerling targets.

Now compare the ROC curves for a nonfluctuating target and a Swerling 1 target. In particular, you want to explore what the SNR requirements are for both situations if you want to achieve the same Pd and Pfa. For such a comparison, it is often easy to plot the ROC curve as Pd against SNR with varying Pfa. We can use the `rocpfa` function to plot the ROC curve in this form.

Assume noncoherent detection with 10 integrated pulses, with the desired Pfa being at most 1e-8. First, plot the ROC curve for a nonfluctuating target.

```
rocpfa(1e-8,'NumPulses',10,'SignalType','NonfluctuatingNoncoherent')
```
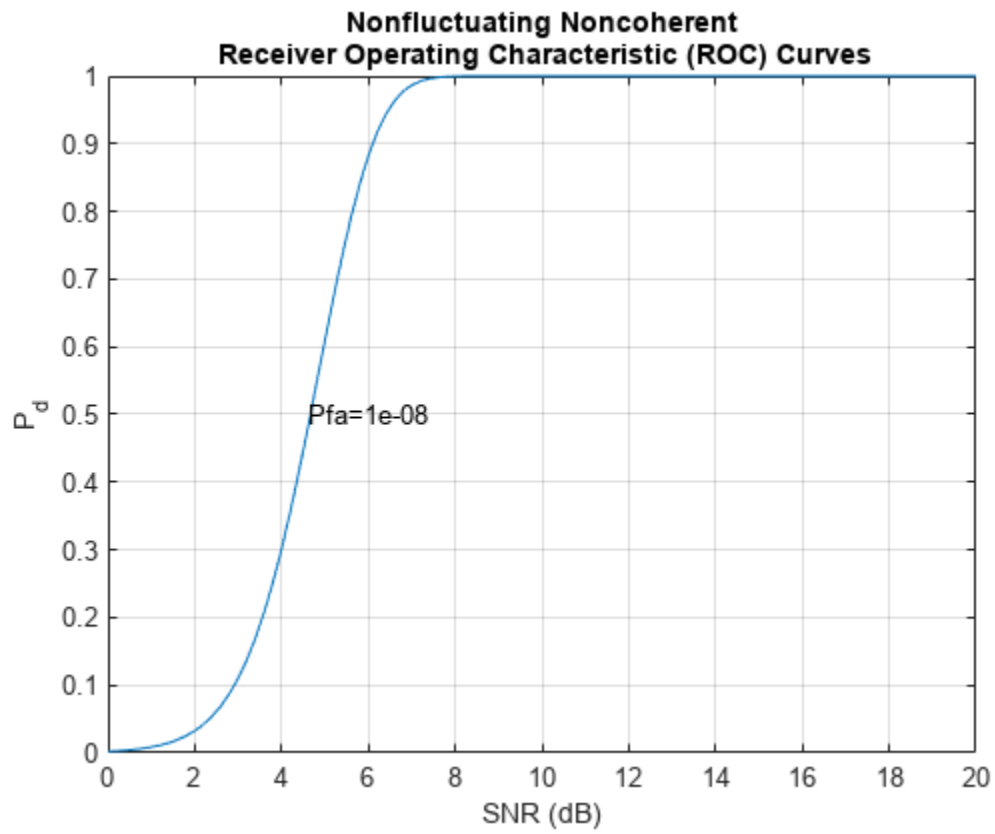
Then plot the ROC curve for a Swerling 1 target for comparison.

```
rocpfa(1e-8,'NumPulses',10,'SignalType','Swerling1')
```
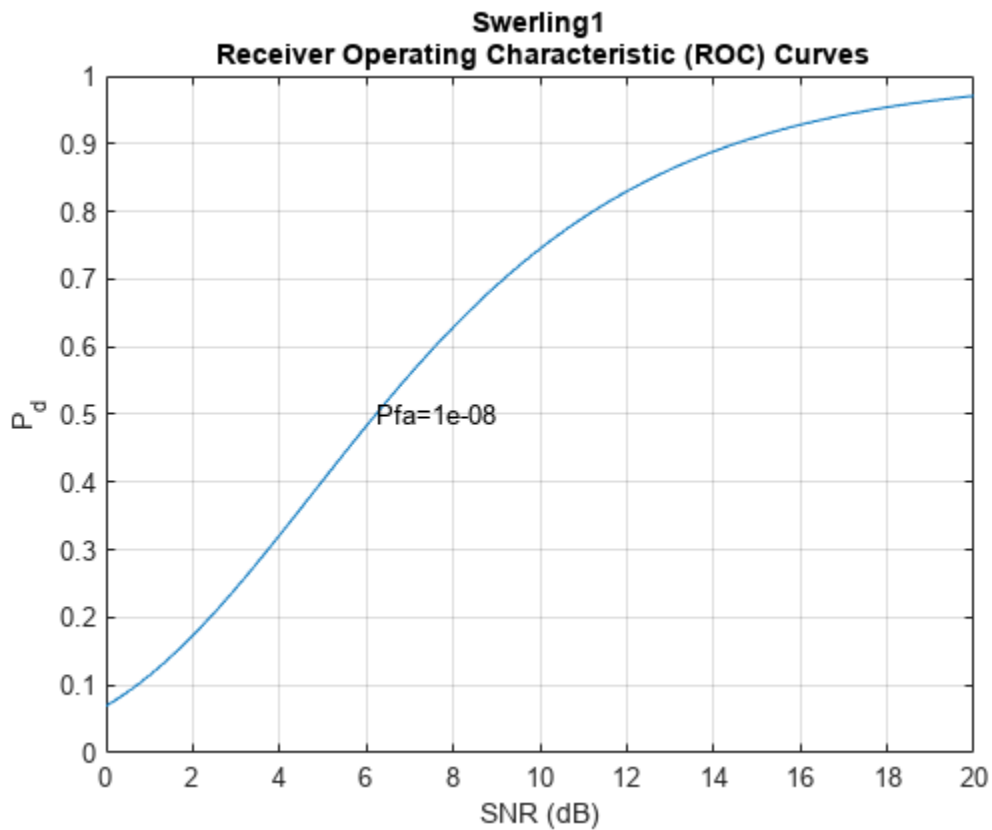
You can see from the figures, we can see that for a Pd of 0.9, you need an SNR of about 6 dB if the target is nonfluctuating. However, if the target is a Swerling case 1 model, the required SNR jumps to more than 14 dB, an 8 dB difference. This will greatly affect the design of the system.

As in the case of nonfluctuating targets, you have approximation equations to help determine the required SNR without having to plot all the curves. The equation used for fluctuating targets is Shnidman's equation. For the scenario used to plot the ROC curves, the SNR requirements can be derived using the `shnidman` function.

```
snr_sw1_db = shnidman(0.9,1e-8,10,1)   % Pd=0.9, Pfa=1e-8, 10 pulses,

snr_sw1_db = 14.7131

                                       % Swerling case 1
```

The calculated SNR requirement matches the value derived from the curve.

Summary

ROC curves are useful for analyzing detector performance, both for coherent and noncoherent systems. This example used the `rocsnr` function to analyze the effectiveness of a linear detector for various SNR values. It also reviewed the improvement in detector performance achieved by averaging multiple samples. Lastly, the example showed how you can use the `rocsnr` and `rocpfa` functions to analyze detector performance when using a noncoherent detector for both nonfluctuating and fluctuating targets.

Monte Carlo ROC Simulation

This example shows how to generate a receiver operating characteristic (ROC) curve of a radar system using a Monte Carlo simulation. The receiver operating characteristic determines how well the system can detect targets while rejecting large spurious signal values when a target is absent (false alarms). A detection system will declare presence or absence of a target by comparing the received signal value to a preset threshold. The probability of detection (*Pd*) of a target is the probability that the instantaneous signal value is larger than the threshold whenever a target is actually present. The probability of false alarm (*Pfa*) is the probability that the signal value is larger than the threshold when a target is absent. In this case, the signal is due to noise and its properties depend on the noise statistics. The Monte Carlo simulation generates a very large number of radar returns with and without a target present. The simulation computes *Pd* and *Pfa* are by counting the proportion of signal values in each case that exceed the threshold.

A ROC curve plots *Pd* as a function of *Pfa*. The shape of a ROC curve depends on the received SNR of the signal. If the arriving signal SNR is known, then the ROC curve shows how well the system performs in terms of *Pd* and *Pfa*. If you specify *Pd* and *Pfa*, then you can determine how much power is needed to achieve this requirement.

You can use the function `rocsnr` to compute theoretical ROC curves. This example shows a ROC curve generated by a Monte Carlo simulation of a single-antenna radar system and compares that curve with a theoretical curve.

Specify Radar Requirements

Set the desired probability of detection to be 0.9 and the probability of false alarm to be 10^{-6}. Set the maximum range of the radar to 4000 meters and the range resolution to 50 meters. Set the actual target range to 3000 meters. Set the target radar cross-section to 1.5 square meters and set the operating frequency to 10 GHz. All computations are performed in baseband.

```
c = physconst('LightSpeed');
pd = 0.9;
pfa = 1e-6;
max_range = 4000;
target_range = 3000.0;
range_res = 50;
tgt_rcs = 1.5;
fc = 10e9;
lambda = c/fc;
```

Any simulation that computes *Pfa* and *pd* requires processing of many signals. To keep memory requirements low, process the signals in chunks of pulses. Set the number of pulses to process to 45000 and set the size of each chunk to 10000.

```
Npulse = 45000;
Npulsebuffsize = 10000;
```

Select Waveform and Signal Parameters

Calculate the waveform pulse bandwidth using the pulse range resolution. Calculate the pulse repetition frequency from the maximum range. Because the signal is baseband, set the sampling frequency to twice the bandwidth. Calculate the pulse duration from the pulse bandwidth.

```
pulse_bw = c/(2*range_res);
prf = c/(2*max_range);
```

```
fs = 2*pulse_bw;
pulse_duration = 10/pulse_bw;
waveform = phased.LinearFMWaveform('PulseWidth',pulse_duration,...
    'SampleRate',fs,'SweepBandwidth',...
    pulse_bw,'PRF',prf);
```

Achieving a particular *Pd* and *Pfa* requires that sufficient signal power arrive at the receiver after the target reflects the signal. Compute the minimum SNR needed to achieve the specified probability of false alarm and probability of detection by using the Albersheim equation.

```
snr_min = albersheim(pd,pfa);
```

To achieve this SNR, sufficient power must be transmitted to the target. Use the radar equation to estimate the peak transmit power, `peak_power`, required to achieve the specified SNR in dB for the target at a range of 3000 meters. The received signal also depends on the target radar cross-section (RCS). which is assumed to follow a nonfluctuating model (Swerling 0). Set the radar to have identical transmit and receive gains of 20 dB. The radar equation is given

```
txrx_gain = 20;
peak_power = ((4*pi)^3*noisepow(1/pulse_duration)*target_range^4*...
    db2pow(snr_min))/(db2pow(2*txrx_gain)*tgt_rcs*lambda^2)
```

peak_power = 293.1827

Set Up the Transmitter System Objects

Create System objects that make up the transmission part of the simulation: radar platform, antenna, transmitter, and radiator.

```
antennaplatform = phased.Platform(...
    'InitialPosition',[0; 0; 0],...
    'Velocity',[0; 0; 0]);
antenna = phased.IsotropicAntennaElement(...
    'FrequencyRange',[5e9 15e9]);
transmitter = phased.Transmitter(...
    'Gain',txrx_gain,...
    'PeakPower',peak_power,...
    'InUseOutputPort',true);
radiator = phased.Radiator(...
    'Sensor',antenna,...
    'OperatingFrequency',fc);
```

Set Up the Target System Object

Create a target System object™ corresponding to an actual reflecting target with a non-zero target cross-section. Reflections from this target will simulate actual radar returns. In order to compute false alarms, create a second target System object with zero radar cross section. Reflections from this target are zero except for noise.

```
target{1} = phased.RadarTarget(...
    'MeanRCS',tgt_rcs,...
    'OperatingFrequency',fc);
targetplatform{1} = phased.Platform(...
    'InitialPosition',[target_range; 0; 0]);
target{2} = phased.RadarTarget(...
    'MeanRCS',0,...
    'OperatingFrequency',fc);
```

```
targetplatform{2} = phased.Platform(...
    'InitialPosition',[target_range; 0; 0]);
```

Set Up Free-Space Propagation System Objects

Model the propagation environment from the radar to the targets and back.

```
channel{1} = phased.FreeSpace(...
    'SampleRate',fs,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
channel{2} = phased.FreeSpace(...
    'SampleRate',fs,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
```

Set Up Receiver System Objects

Specify the noise by setting the `NoiseMethod` property to `'Noise temperature'` and the `ReferenceTemperature` property to 290 K.

```
collector = phased.Collector(...
    'Sensor',antenna,...
    'OperatingFrequency',fc);
receiver = phased.ReceiverPreamp(...
    'Gain',txrx_gain,...
    'NoiseMethod','Noise temperature',...
    'ReferenceTemperature',290.0,...
    'NoiseFigure',0,...
    'SampleRate',fs,...
    'EnableInputPort',true);
receiver.SeedSource = 'Property';
receiver.Seed = 2010;
```

Specify Fast-Time Grid

The fast-time grid is the set of time samples within one pulse repetition time interval. Each sample corresponds to a range bin.

```
fast_time_grid = unigrid(0,1/fs,1/prf,'[)');
rangebins = c*fast_time_grid/2;
```

Create Transmitted Pulse from Waveform

Create the waveform you want to transmit.

```
wavfrm = waveform();
```

Create the transmitted signal that includes transmitted antenna gains.

```
[sigtrans,tx_status] = transmitter(wavfrm);
```

Create matched filter coefficients from the waveform System object. Then create the matched filter System object.

```
MFCoeff = getMatchedFilter(waveform);
matchingdelay = size(MFCoeff,1) - 1;
filter = phased.MatchedFilter(...
```

```
        'Coefficients',MFCoeff,...
        'GainOutputPort',false);
```

Compute Target Range Bin

Compute the target range, and then compute the index into the range bin array. Because the target and radar are stationary, use the same values of position and velocity throughout the simulation loop. You can assume that the range bin index is constant for the entire simulation.

```
ant_pos = antennaplatform.InitialPosition;
ant_vel = antennaplatform.Velocity;
tgt_pos = targetplatform{1}.InitialPosition;
tgt_vel = targetplatform{1}.Velocity;
[tgt_rng,tgt_ang] = rangeangle(tgt_pos,ant_pos);
rangeidx = val2ind(tgt_rng,rangebins(2)-rangebins(1),rangebins(1));
```

Loop Over Pulses

Create a signal processing loop. Each step is accomplished by executing the System objects. The loop processes the pulses twice, once for the target-present condition and once for target-absent condition.

1   Radiate the signal into space using `phased.Radiator`.

2   Propagate the signal to the target and back to the antenna using `phased.FreeSpace`.

3   Reflect the signal from the target using `phased.Target`.

4   Receive the reflected signals at the antenna using `phased.Collector`.

5   Pass the received signal though the receive amplifier using `phased.ReceiverPreamp`. This step also adds the random noise to the signal.

6   Match filter the amplified signal using `phased.MatchedFilter`.

7   Store the matched filter output at the target range bin index for further analysis.

```
rcv_pulses = zeros(length(sigtrans),Npulsebuffsize);
h1 = zeros(Npulse,1);
h0 = zeros(Npulse,1);
Nbuff = floor(Npulse/Npulsebuffsize);
Nrem = Npulse - Nbuff*Npulsebuffsize;
for n = 1:2 % H1 and H0 Hypothesis
    trsig = radiator(sigtrans,tgt_ang);
    trsig = channel{n}(trsig,...
        ant_pos,tgt_pos,...
        ant_vel,tgt_vel);
    rcvsig = target{n}(trsig);
    rcvsig = collector(rcvsig,tgt_ang);

    for k = 1:Nbuff
        for m = 1:Npulsebuffsize
            rcv_pulses(:,m) = receiver(rcvsig,~(tx_status>0));
        end
        rcv_pulses = filter(rcv_pulses);
        rcv_pulses = buffer(rcv_pulses(matchingdelay+1:end),size(rcv_pulses,1));
        if n == 1
            h1((1:Npulsebuffsize) + (k-1)*Npulsebuffsize) = rcv_pulses(rangeidx,:).';
        else
            h0((1:Npulsebuffsize) + (k-1)*Npulsebuffsize) = rcv_pulses(rangeidx,:).';
        end
```

```
            end
        if (Nrem > 0)
            for m = 1:Nrem
                rcv_pulses(:,m) = receiver(rcvsig,~(tx_status>0));
            end
            rcv_pulses = filter(rcv_pulses);
            rcv_pulses = buffer(rcv_pulses(matchingdelay+1:end),size(rcv_pulses,1));
             if n == 1
                h1((1:Nrem) + Nbuff*Npulsebuffsize) = rcv_pulses(rangeidx,1:Nrem).';
            else
                h0((1:Nrem) + Nbuff*Npulsebuffsize) = rcv_pulses(rangeidx,1:Nrem).';
             end
        end
end
```

Create Histogram of Matched Filter Outputs

Compute histograms of the target-present and target-absent returns. Use 100 bins to give a rough estimate of the spread of signal values. Set the range of histogram values from the smallest signal to the largest signal.

```
h1a = abs(h1);
h0a = abs(h0);
thresh_low = min([h1a;h0a]);
thresh_hi  = max([h1a;h0a]);
nbins = 100;
binedges = linspace(thresh_low,thresh_hi,nbins);
figure
histogram(h0a,binedges)
hold on
histogram(h1a,binedges)
hold off
title('Target-Absent Vs Target-Present Histograms')
legend('Target Absent','Target Present')
```
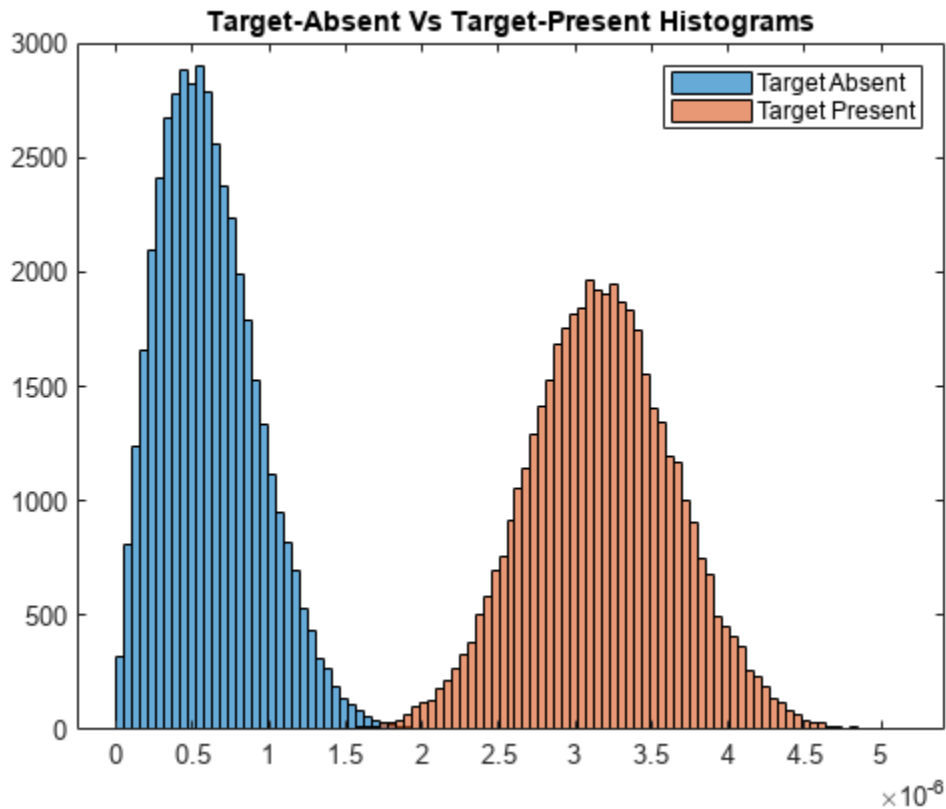
Compare Simulated and Theoretical *Pd* and *Pfa*

To compute *Pd* and *Pfa*, calculate the number of instances that a target-absent return and a target-present return exceed a given threshold. This set of thresholds has a finer granularity than the bins used to create the histogram in the previous simulation. Then, normalize these counts by the number of pulses to get an estimate of the probabilities. The vector `sim_pfa` is the simulated probability of false alarm as a function of the threshold, `thresh`. The vector `sim_pd` is the simulated probability of detection, also a function of the threshold. The receiver sets the threshold so that it can determine whether a target is present or absent. The histogram above suggests that the best threshold is around 1.8.

```
nbins = 1000;
thresh_steps = linspace(thresh_low,thresh_hi,nbins);
sim_pd = zeros(1,nbins);
sim_pfa = zeros(1,nbins);
for k = 1:nbins
    thresh = thresh_steps(k);
    sim_pd(k) = sum(h1a >= thresh);
    sim_pfa(k) = sum(h0a >= thresh);
end
sim_pd = sim_pd/Npulse;
sim_pfa = sim_pfa/Npulse;
```

To plot the experimental ROC curve, you must invert the Pfa curve so that you can plot *Pd* against *Pfa*. You can invert the *Pfa* curve only when you can express *Pfa* as a strictly monotonic decreasing function of `thresh`. To express *Pfa* this way, find all array indices where the *Pfa* is the constant over neighboring indices. Then, remove these values from the *Pd* and *Pfa* arrays.

```
pfa_diff = diff(sim_pfa);
idx = (pfa_diff == 0);
sim_pfa(idx) = [];
sim_pd(idx) = [];
```

Limit the smallest Pfa to 10^{-6}.

```
minpfa = 1e-6;
N = sum(sim_pfa >= minpfa);
sim_pfa = fliplr(sim_pfa(1:N)).';
sim_pd = fliplr(sim_pd(1:N)).';
```

Compute the theoretical *Pfa* and *Pd* values from the smallest *Pfa* to 1. Then plot the theoretical *Pfa* curve.

```
[theor_pd,theor_pfa] = rocsnr(snr_min,'SignalType',...
    'NonfluctuatingNoncoherent',...
    'MinPfa',minpfa,'NumPoints',N,'NumPulses',1);
semilogx(theor_pfa,theor_pd)
hold on
semilogx(sim_pfa,sim_pd,'r.')
title('Simulated and Theoretical ROC Curves')
xlabel('Pfa')
ylabel('Pd')
grid on
legend('Theoretical','Simulated','Location','SE')
```
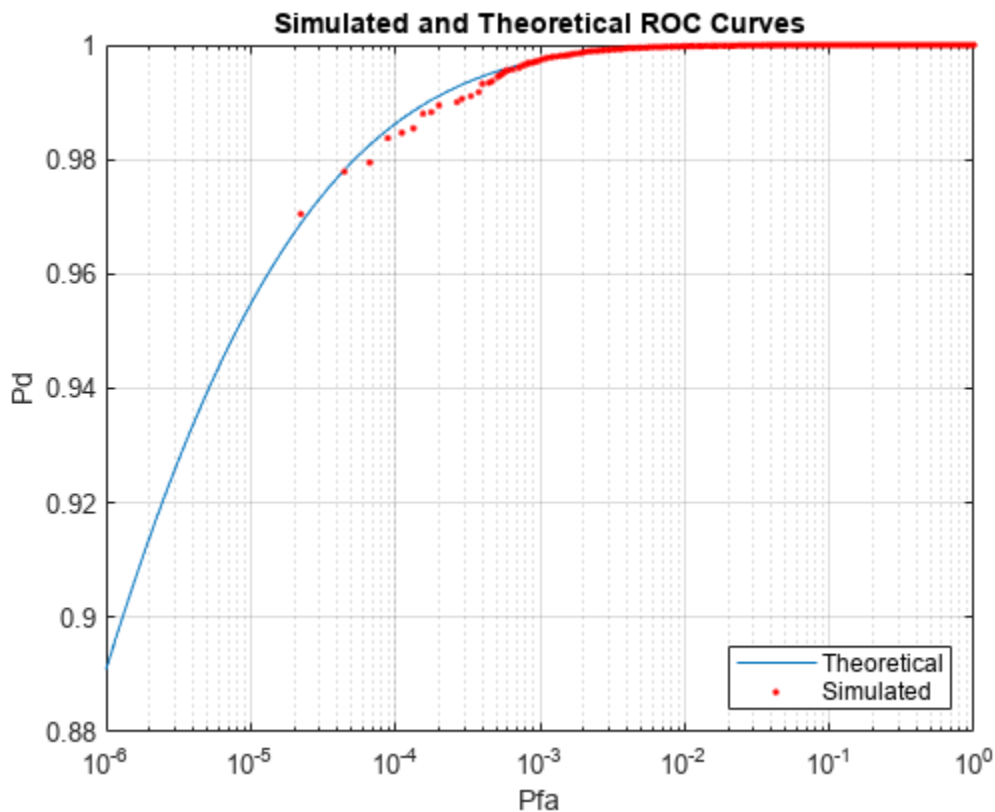
Improve Simulation Using One Million Pulses

In the preceding simulation, *Pd* values at low *Pfa* do not fall along a smooth curve and do not even extend down to the specified operating regime. The reason for this is that at very low *Pfa* levels, very few, if any, samples exceed the threshold. To generate curves at low *Pfa*, you must use a number of samples on the order of the inverse of *Pfa*. This type of simulation takes a long time. The following curve uses one million pulses instead of 45,000. To run this simulation, repeat the example, but set `Npulse` to 1000000.
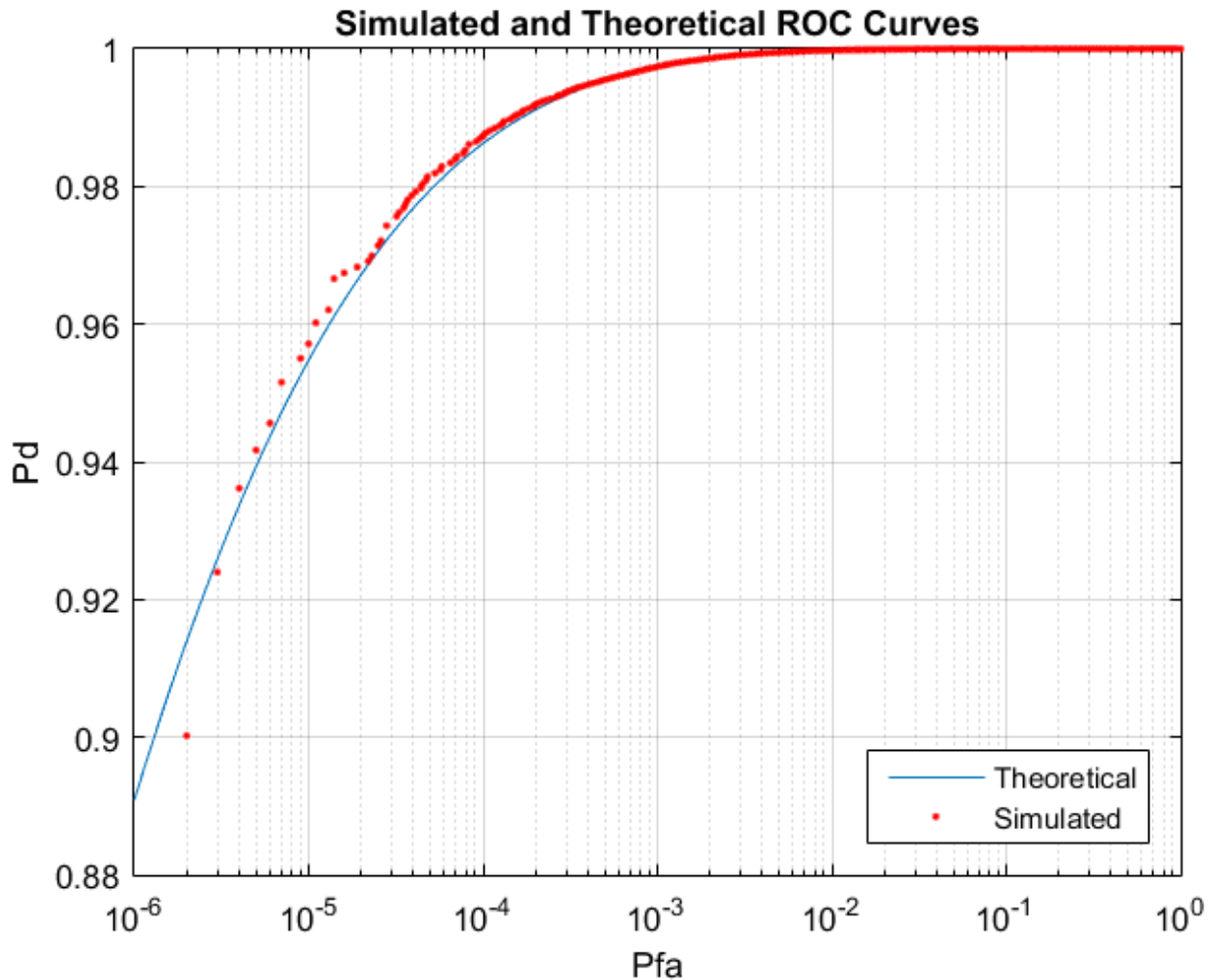
Assessing Performance with the Tracker Operating Characteristic

This example shows how to assess the:

- Probability of target track: The target track detection probability for a single target with and without the presence of false alarms
- Probability of false track: False track probability due to false alarms in the neighborhood of a single target

This example discusses different methods to perform these calculations with varying levels of fidelity and computation time.

In the assessment of tracker performance, four types of probabilities are often calculated:

1. Probability of a single target track in the absence of false alarms (probability of false alarm = $P_{fa} = 0$)
2. Probability of a single false track in the absence of targets (probability of detection = $P_d = 0$)
3. Probability of a single target track in the presence of false alarms
4. Probability of a single false track in the presence of targets

This example first calculates the probability of single target track in the absence of false alarms using the Bernoulli sum. Then, it discusses the common gate history (CGH) algorithm that can be used to calculate all 4 types of probabilities and introduces the concept of the tracker operating characteristic (TOC), which is similar in form to the receiver operating characteristic (ROC). The CGH algorithm provides an estimate of system capacity and offers a means to assess end-to-end system performance. Lastly, the example presents the CGH algorithm as applied to an automotive radar design scenario and assists users in the selection of:

- Required target signal-to-noise ratio (SNR)
- Number of false tracks
- Tracker confirmation threshold

Calculate Single Target Track Probability in the Absence of False Alarms

Bernoulli Sum

The Bernoulli sum allows for quick and easy performance analysis in the case of a single target in the absence of false alarms. The track detection probability P_{dt} can be defined in terms of the receiver detection probability P_d for a window period defined as

$T_w = NT,$

where T is the basic sampling period and N represents the number of opportunities for a detection.

For a confirmation threshold logic of M-of-N, the target track probability P_{dt} is defined as

$P_{dt} = \sum_{i=M}^{N} C(N, i) P_d^i (1 - P_d)^{(N-i)},$

where

$$C(N, i) = \frac{N!}{(N-i)! i!}.$$

The confirmation threshold logic denoted as *M*-of-*N* or *M*/*N* is a one-stage logic where a track must associate to a detection, also known as a hit, at least *M* times out of *N* consecutive looks. For example, consider a 2-of-3 logic. In the following figure, the solid yellow represents a hit, which can be from either a target or a false alarm. The patterned blue blocks represent misses. The cases represented below are not intended to be exhaustive, but the figure indicates cases 2 and 3 satisfy the threshold, but case 1 does not.

Case 1
M = 1, N = 3
Confirmation Threshold Not Satisfied

Case 2
M = 2, N = 3
Confirmation Threshold Satisfied

Case 3
M = 2, N = 3
Confirmation Threshold Satisfied

Investigate the probability of target track versus probability of detection for a confirmation threshold of 2/3 using the Bernoulli sum method. Perform the Bernoulli sum calculation, assuming that one hit is required to initialize a track.

```
% Define probabilities for analysis
Pd = linspace(0,1,100).';

% Define confirmation threshold M/N
M = 2; % Number of hits
N = 3; % Number of observations or opportunities

% Calculate Bernoulli sum, assuming 1 hit is required to initialize a track
```

```
tic
PdtBernoulli = helperBernoulliSum(Pd,M,N);
elapsedTime = toc;
helperUpdate('Bernoulli',elapsedTime);
```

Bernoulli calculation completed. Total computation time is 0.0198 seconds.

```
% Plot the probability of detection versus the probability of target track
hAxes = helperPlot(Pd,PdtBernoulli,'M/N = 2/3','P_D','P_{DT}', ...
    sprintf('Bernoulli Sum\nProbability of Target Track in the Absence of False Alarms'));

% Set desired probability of target track
yline(hAxes,0.9,'--','LineWidth',2,'DisplayName','Desired P_{DT}');
```
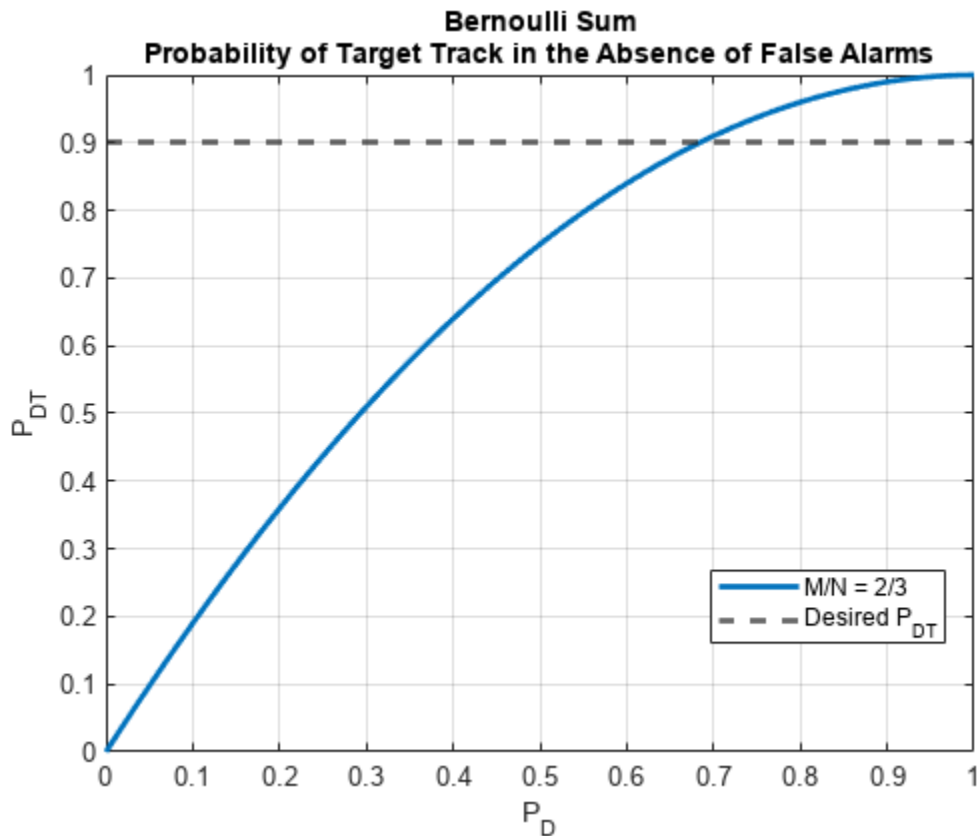

Assuming a required probability of target track of 0.9, the plot above indicates that a detection probability of about 0.7 is necessary.

Calculate Probabilities for Targets in Clutter

The values obtained from a Bernoulli sum calculation are useful in quick analyses, but are not generally representative of real tracking environments, where target tracks are affected by the presence of false alarms. Consider the scenario where a target is operating in the presence of clutter.

Assuming that false alarms occur on a per-look per-cell basis, the probability of false alarms in a tracking gate depend upon the number of cells in the gate. Assume three types of events:

- Miss: No detection

- Hit: False alarm
- Hit: Target detection

The number of cells in a gate depends upon the history of events and the order in which events occur. These factors dictate the gate growth sequence of the tracker.

The Bernoulli sum method assumes that there are no false alarms and that the order of detections does not matter. Thus, when you use the Bernoulli sum in the target in clutter scenario, it produces overly optimistic results.

One approach for analyzing such scenarios is to evaluate every possible track sequence and determine which sequences satisfy the confirmation threshold logic. This brute-force approach of building the Markov chain is generally too computationally intensive.

Another approach is to utilize a Monte Carlo-type analysis. Rather than generating the full Markov chain, a Monte Carlo simulation manually generates random sequences of N events. The confirmation threshold is applied to each sequence, and statistics are aggregated. The Monte Carlo method is based upon the law of large numbers, so performance improves as the number of iterations increases. Monte Carlo analysis lends itself well to parallelization, but in the case of small probabilities of false alarm, the number of iterations can become untenable. Thus, alternative methods to quickly calculate track probability measures are needed.

The Common Gate History Algorithm

The common gate history (CGH) algorithm greatly reduces computation times and memory requirements. The algorithm avoids the need for manual generation of sequences, as in the case of Monte Carlo analysis, which can be costly for low-probability events.

The algorithm begins by making the assumption that there are three types of tracks, which can contain:

1 Detections from targets
2 Detections from targets and false alarms
3 Detections from false alarms only

A target track is defined as any track that contains at least one target detection and satisfies the M/N confirmation threshold. Thus, track types 1 and 2 are considered to be target tracks, whereas 3 is considered to be a false track.

Given the previously defined track types and with a confirmation threshold logic M/N, a unique track state is defined as

$$\omega = [\omega_l, \omega_{lt}, \lambda],$$

where ω_l is the number of time steps since the last detection (target or false alarm), ω_{lt} is the number of time steps since the last target detection, and λ is the total count of detections (targets or false alarms). As the algorithm proceeds, the track state vector evolves according to a Markov chain.

The algorithm assumes that a track can be started given only two types of events:

- Target detection
- False alarm

Once a track is initiated, the following four types of events continue a track:

- No detection
- Target detection
- False alarm
- Target detection and false alarm

The track probability at look m is multiplied by the probability of the event that continues the track at look $m + 1$. The tracks are then lumped by adding the track probabilities of the track files with a common gate history vector. This lumping keeps the number of track states in the Markov chain within reasonable limits.

The assumptions of the CGH algorithm are as follows:

- The probability of more than one false alarm in a gate is low, which is true when the probability of false alarm is low (10^{-3} or less)
- The location of a target in a gate has a uniform spatial distribution
- A track splitting algorithm is used

The CGH algorithm can be used to calculate all four probability types:

- Probability of a single target track in the absence of false alarms ($P_{fa} = 0$)
- Probability of a single false track in the absence of targets ($P_d = 0$)
- Probability of a single target track in the presence of false alarms
- Probability of a single false track in the presence of targets

The probability of target track and the probability of false track form the basis of the tracker operating characteristic (TOC). The TOC compliments the receiver operating characteristic (ROC), which is commonly used in the analysis and performance prediction of receivers. Combining the ROC and the TOC provides an end-to-end system analysis tool.

Calculate and plot the TOC using ROC curves from `rocsnr` as inputs. Assume a signal-to-noise ratio (SNR) of 8 dB. Continue to use the 2/3 confirmation threshold logic as in the Bernoulli sum example. Use the `toccgh` built-in tracker.

```
% Receiver operating characteristic (ROC)
snrdB = 8; % SNR (dB)
[Pd,Pfa] = rocsnr(snrdB,'MaxPfa',1e-3,'MinPfa',1e-12,'NumPoints',20);

% Plot ROC
helperPlotLog(Pfa,Pd,snrdB, ...
    'Probability of False Alarm (P_{FA})', ...
    'Probability of Detection (P_D)', ...
    'Receiver Operating Characteristic');
```
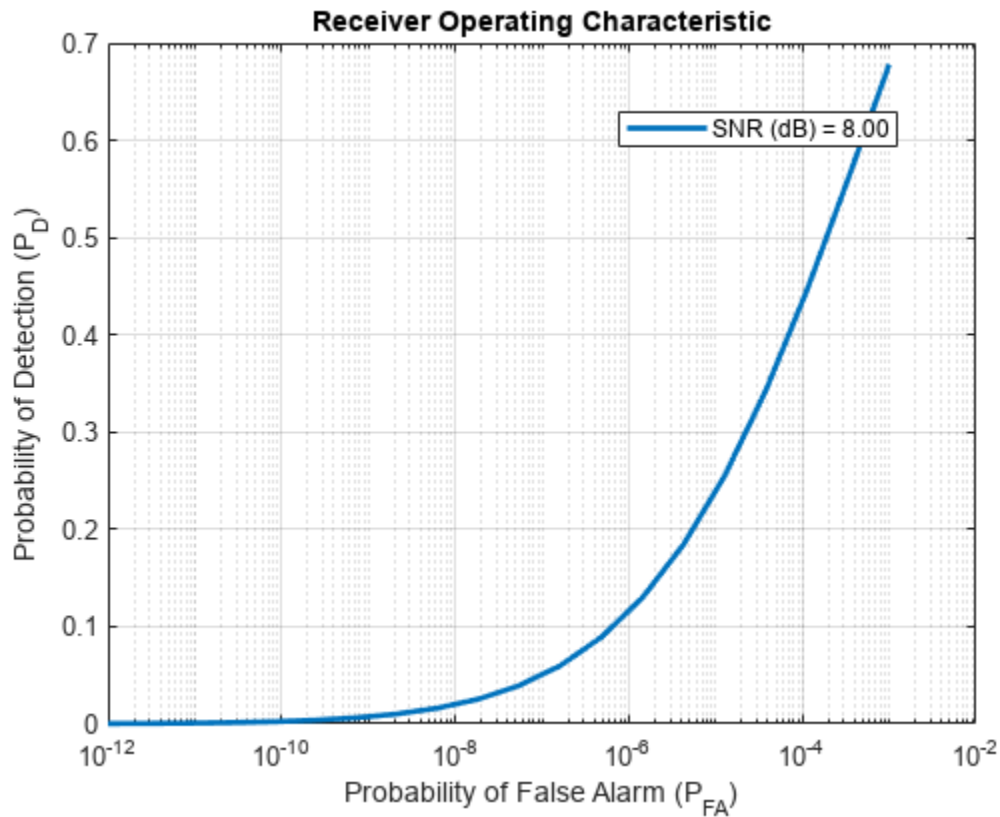
```
% CGH algorithm
tic
[PdtCGH,PftCGH] = toccgh(Pd,Pfa,'ConfirmationThreshold',[M N]);
elapsedTime = toc;
helperUpdate('Common Gate History',elapsedTime);
```

Common Gate History calculation completed. Total computation time is 1.1001 seconds.

```
% Plot CGH results
hAxes = helperPlotLog(PftCGH,PdtCGH,'CGH','P_{FT}','P_{DT}', ...
    'Tracker Operating Characteristic (TOC) Curve');
```
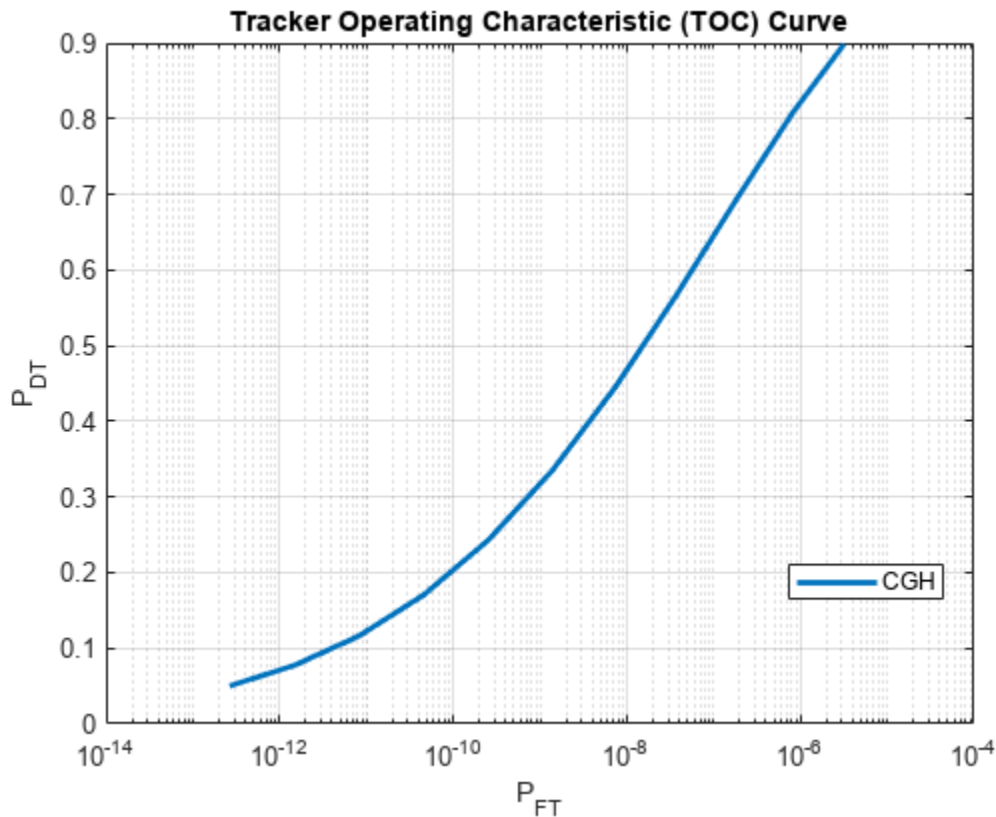
The CGH algorithm permits the assessment of tracker performance similar to a Monte Carlo analysis but with acceptable computation times despite low-probability events. The CGH algorithm thus permits high-level investigation and selection of options prior to more intensive, detailed simulations.

Using CGH with Custom Trackers

Consider a tracker for an automotive application. Define a custom, one-dimensional, nearly constant velocity (NCV) tracker using `trackingKF`. Assume that the update rate Δt is 1 second. Assume that the state transition matrix is of the form

$$A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

and the process noise is of the form

$$Q = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 \end{bmatrix} q,$$

where q is a tuning factor defined as

$$q = a_{\max}^2 \Delta t.$$

The input $a_{\max}^2$ is the maximum target acceleration expected. Assume that a maximum acceleration of 4 m/s2 is expected for vehicles.

```matlab
% Define the state transition matrix
dt = 1; % Update rate (sec)
A = [1 dt; 0 1];

% Define the process noise
Q = [dt^4/4 dt^3/2; dt^3/2 dt^2];

% Tune the process noise
amax = 4; % Maximum target acceleration (m/s^2)
q = amax^2*dt;

% Update the process noise
Q = Q.*q;

% Initialize the Kalman filter
trkfilt = trackingKF('MotionModel','Custom', ...
    'StateCovariance', [0 0; 0 0], ...
    'StateTransitionModel',A, ...
    'ProcessNoise',Q, ...
    'MeasurementNoise',0, ...
    'MeasurementModel',[1 0]);
```

An error ellipse is used to model track uncertainty. From this uncertainty ellipse, the gate growth sequence can be calculated.

1-σ Error Ellipse
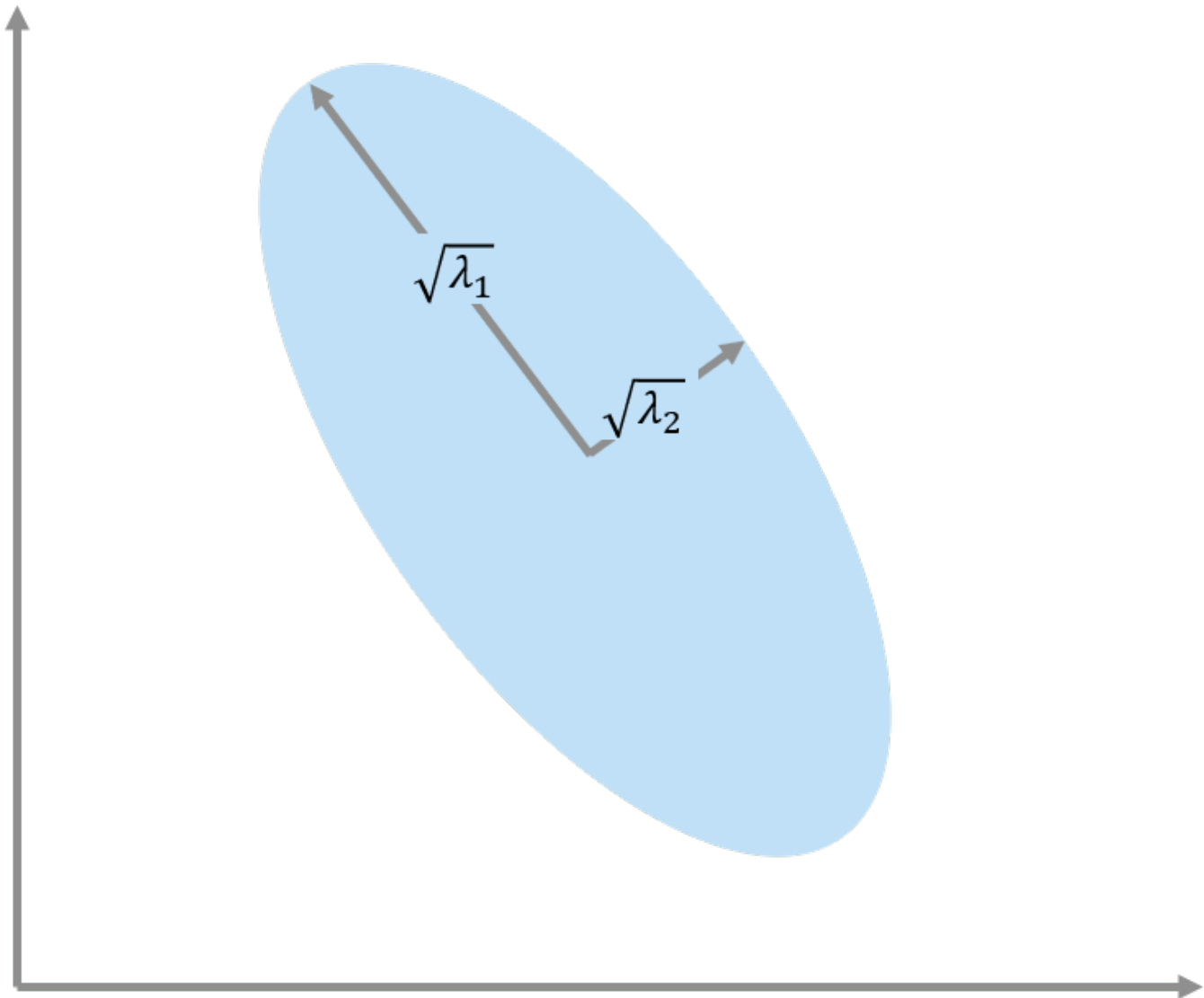
The 1-σ values of the error ellipse are calculated as the square root of the eigenvalues λ of the predicted state covariance $P_{k+1|k}$:

$$[\sqrt{\lambda_1}, \sqrt{\lambda_2}] = \sqrt{\mathrm{eig}(P_{k+1|k})}.$$

The area of the error ellipse is then calculated as

Error Ellipse Area $= \pi \sqrt{\lambda_1} \sqrt{\lambda_2}.$

The area of the bins is calculated as

Bin Area $= (\Delta\text{Range})(\Delta\text{Range Rate}).$

Finally, the gate size in bins is

Gate Size $= \dfrac{\text{Error Ellipse Area}}{\text{Bin Area}}$.

The gate size is thus dependent upon the tracker, the event sequence, and the resolution of the bins. Calculate the gate growth sequence, assuming a confirmation threshold N equal to 3. Assume the range and range rate resolutions for the automotive radar are equal to 1 meter and 1 m/s, respectively.

```
% Calculate gate growth sequence
res = [1, 1]; % Bin resolutions [range (m), range-rate (m/s)]
gs = helperCalculateGateSize(N,trkfilt,res)
```

```
gs = 1×3

    1    51    124
```

```
% CGH algorithm
tic
[PdtCGHcustom,PftCGHcustom] = toccgh(Pd,Pfa,'ConfirmationThreshold',[M N],'GateGrowthSequence',gs
elapsedTime = toc;
helperUpdate('Common Gate History',elapsedTime);
```

```
Common Gate History calculation completed. Total computation time is 0.0880 seconds.
```

```
% Add plot to previous plot
helperAddPlotLog(hAxes,PftCGHcustom,PdtCGHcustom,'CGH with Custom Gate Growth Sequence');
```
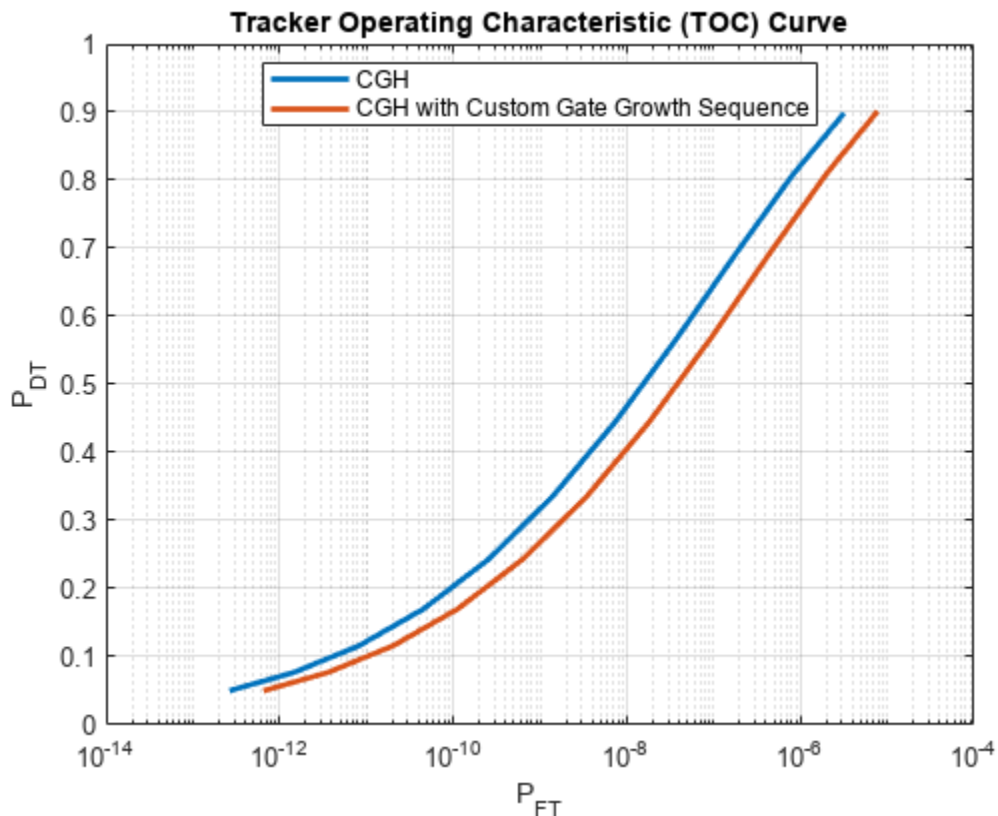
Tracker Performance Assessment for an Automotive Radar System

Probability of False Alarm and Probability of Target Track Requirements

By using the ROC and TOC in conjunction, a system analyst can select a detector's operation point that satisfies the overall system requirements. Consider an automotive radar case. Due to the nature of the application, it is desired that false alarms remain very low probability events. Additionally, the probability of target track should be high for safety purposes. Consider the following two requirements.

- Requirement 1 − Probability of false alarm must be less than 10^{-6}
- Requirement 2 − Probability of target track must be equal to 0.9 or above

Calculate the ROC curves for SNRs equal to 6, 8, 10, and 12 dB.

```matlab
% Calculate ROC curves
snrdB = [6 8 10 12]; % SNR (dB)
[Pd,Pfa] = rocsnr(snrdB,'MaxPfa',1e-3,'MinPfa',1e-10,'NumPoints',10);
```

Use the calculated ROC curves as inputs to the `toccgh` function to generate the associated TOC curves. Use the same confirmation threshold and gate growth sequence previously generated.

```matlab
% Generate TOC curves
tic
toccgh(Pd,Pfa,'ConfirmationThreshold',[M N],'GateGrowthSequence',gs);
elapsedTime = toc;
helperUpdate('Common Gate History',elapsedTime);
```

Common Gate History calculation completed. Total computation time is 1.0398 seconds.

```matlab
% Requirement 1: Probability of false alarms must be less than 1e-6
hAxesROC = subplot(2,1,1);
xlim([1e-10 1e-2])
ylim([0 1])
reqPfa = 1e-6;
helperColorZonesReqPfa(hAxesROC,reqPfa)
legend(hAxesROC,'Location','eastoutside')

% Requirement 2: Probability of target track must be equal to 0.9 or above
hAxesTOC = subplot(2,1,2);
xlim([1e-14 1e-4])
ylim([0 1])
reqPdt = 0.9;
helperColorZonesReqPdt(hAxesTOC,reqPdt)
```

Receiver Operating Characteristic (ROC) Curve

Tracker Operating Characteristic (TOC) Curve
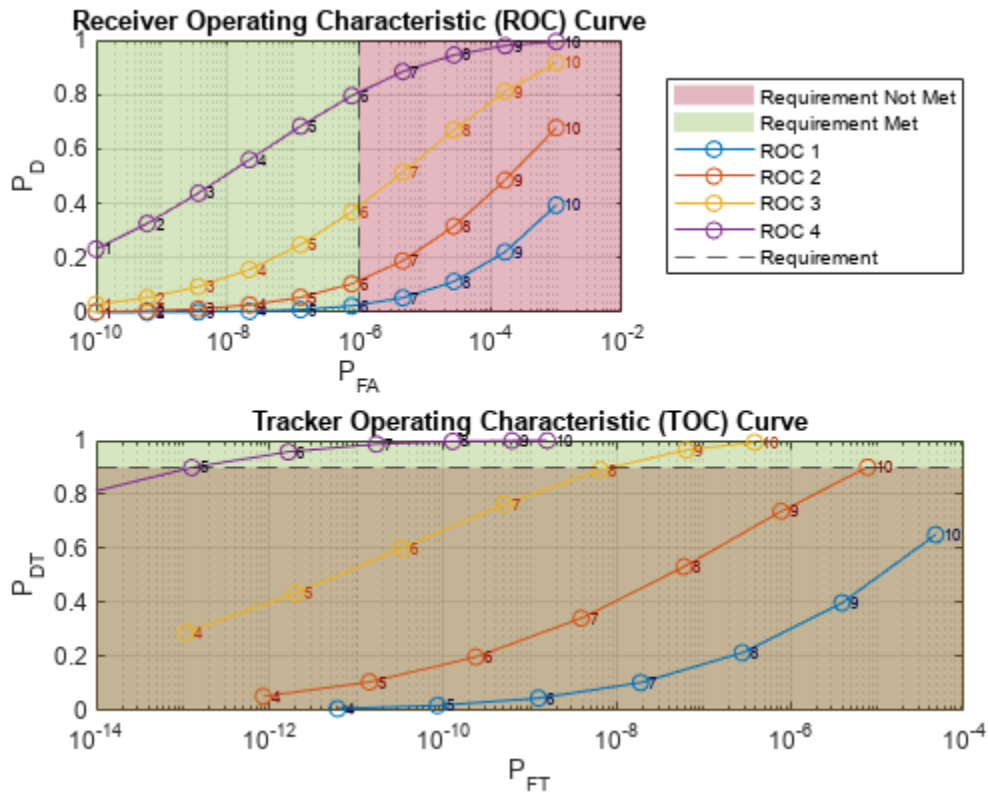
Requirement 1 indicates that only points 1 through 6 on the ROC curves can be included for further analysis since they satisfy a probability lower than 10^{-6}. Points 7 through 10 do not meet this requirement.

Regarding requirement 2, the ROC curve corresponding to 6 dB SNR does not meet the second requirement at any point. The only curves to continue considering are the 8, 10, and 12 dB curves. Requirement 2 is met only for point 10 on the 8 dB curve, points 9 and 10 on the 10 dB curve, and points 5 through 10 on the 12 dB curve.

Combining requirements 1 and 2, only two analysis points satisfy both requirements — points 5 and 6 on the 12 dB curve. Point 5 corresponds to a probability of target track of 0.90 and a probability of false track of 1.27×10^{-13}, which corresponds to a probability of detection of 0.68 and a probability of false alarm of 1.29×10^{-7}. Similarly, point 6 corresponds to a probability of target track of 0.96, probability of false track of 1.64×10^{-12}, probability of detection of 0.80, and a probability of false alarm of 7.74×10^{-7}. Select point 6, which represents a tradeoff of improved probability of target track at the expense of a slightly higher but reasonable probability of false track.

The CGH algorithm permits an estimation of the expected number of false tracks based on the number of targets anticipated in an environment and the number of cells in the radar data. The expected number of false tracks E_{ft} is calculated as

$$E_{ft} = P_{ft, nt}N_c + P_{ft}N_t,$$

where $P_{\text{ft, nt}}$ is the probability of false track in the absence of targets, N_c is the number of cells, P_{ft} is the probability of false track in the presence of targets, and N_t is the number of targets.

Consider an environment where the number of targets is expected to be equal to 10 and the number of cells is equal to

Number of Cells = (Number of Range Cells) × (Number of Range Rate Cells) = $1000 \times 100 = 10^5$.

```matlab
% Calculate expected number of false tracks using toccgh
numCells = 1e5;          % Number of cells in radar data
numTargets = 10;         % Number of targets in scenario
selectedPd = Pd(6,4) ;   % Selected probability of detection
selectedPfa = Pfa(6);    % Selected probability of false alarm
[Pdt,Pft,Eft] = toccgh(selectedPd,selectedPfa, ...
    'ConfirmationThreshold',[M N],'GateGrowthSequence',gs, ...
    'NumCells',numCells,'NumTargets',10);

% Output results
helperPrintTrackProbabilities(Pdt,Pft,Eft);
```

```
Probability of Target Track in Presence of False Alarms = 0.9581
Probability of False Track in the Presence of Targets = 1.6410e-12
Expected Number of False Tracks = 5
```

Thus, based on the system parameters, you can expect about five false tracks.

Analysis of Confirmation Thresholds

Consider the same automotive radar design case but investigate the effect of confirmation thresholds 2/4, 3/4, and 4/4. Assume the following:

- Requirement 1 – Probability of false alarm must be less than 10^{-6}
- Requirement 2 – Probability of target track must be equal to 0.9 or above

First, calculate the ROC curve using the `rocpfa` function.

```matlab
% Calculate ROC curve assuming a probability of false alarm of 1e-6
Pfa = 1e-6;
numPoints = 20;
[Pd,snrdB] = rocpfa(Pfa,'NumPoints',numPoints,'MaxSNR',15);
```

Update the gate growth sequence due to the greater number of observations.

```matlab
% Update the gate growth sequence
N = 4;
trkfilt = trackingKF('MotionModel','Custom', ...
    'StateCovariance', [0 0; 0 0], ...
    'StateTransitionModel',A, ...
    'ProcessNoise',Q, ...
    'MeasurementNoise',0, ...
    'MeasurementModel',[1 0]);
gs = helperCalculateGateSize(N,trkfilt,res)
```

```
gs = 1×4

     1    51    124    225
```

Calculate the TOC given the ROC curves as inputs and confirmation thresholds equal to 2/4, 3/4, and 4/4.

```
% Calculate TOC
cp = [2 4; 3 4; 4 4];
numCp = size(cp,1);
PdtMat = zeros(numPoints,numCp);
PftMat = zeros(numPoints,numCp);
EftMat = zeros(numPoints,numCp);
for ii = 1:numCp
    [PdtMat(:,ii),PftMat(:,ii),EftMat(:,ii)] = toccgh(Pd.',Pfa, ...
        'ConfirmationThreshold',cp(ii,:),'GateGrowthSequence',gs, ...
        'NumCells',numCells,'NumTargets',10);
end
```

```
% Plot ROC and TOC
reqPdt = 0.9;
helperPlotROCTOC(reqPdt,Pfa,Pd,snrdB,PdtMat,cp);
```
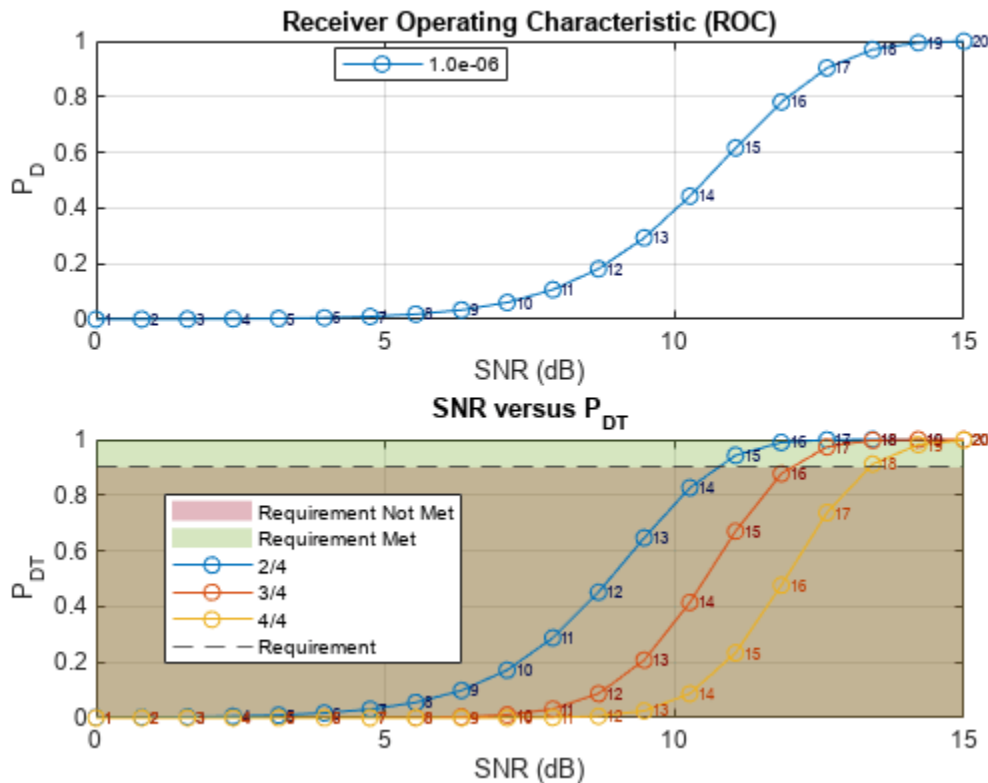


```
helperPrintReqValues(reqPdt,Pd,snrdB,PdtMat,EftMat,cp);
```

```
Confirmation Threshold          = 2/4
Required Probability of Detection = 0.55
Required SNR (dB)               = 10.76
Expected Number of False Tracks = 18

Confirmation Threshold          = 3/4
Required Probability of Detection = 0.81
```

```
Required SNR (dB)                = 12.03
Expected Number of False Tracks  = 1

Confirmation Threshold           = 4/4
Required Probability of Detection = 0.97
Required SNR (dB)                = 13.37
Expected Number of False Tracks  = 1
```

Reviewing the results, you can see that the more stringent the confirmation threshold, the higher the required SNR. However, the more stringent confirmation thresholds result in improved numbers of false tracks.

Summary

In the assessment of tracker performance, four types of probabilities are often calculated:

1   Probability of a single target track in the absence of false alarms ($P_{fa} = 0$)

2   Probability of a single false track in the absence of targets ($P_d = 0$)

3   Probability of a single target track in the presence of false alarms

4   Probability of a single false track in the presence of targets

For the calculation of 1, Bernoulli sums can be used. However, to obtain the other probabilities, a different method must be employed. Monte Carlo analysis can be used for the computation of the latter three types of probabilities, but the computational resources and time required can become untenable, which is particularly true for events with low probability. The common gate history (CGH) algorithm can be used to calculate all four quantities and greatly reduces computational resources needed.

The CGH algorithm can be used to generate the tracker operating characteristic (TOC). The TOC compliments the receiver operating characteristic (ROC) and provides a means to assess overall system performance. The TOC and ROC curves can be used in a multitude of ways such as determining:

•   Required target signal-to-noise ratio (SNR)

•   Tracker confirmation threshold

Finally, the CGH algorithm permits the calculation of an expected number of false tracks, which offers insights into system capacity. The expected number of false tracks can be used to ascertain computational load and to assist with decisions related to hardware and processing.

References

1   Bar-Shalom, Y., et al. "From Receiver Operating Characteristic to System Operating Characteristic: Evaluation of a Track Formation System." *IEEE Transactions on Automatic Control*, vol. 35, no. 2, Feb. 1990, pp. 172-79. DOI.org (Crossref), doi:10.1109/9.45173.

2   Bar-Shalom, Yaakov, et al. T*racking and Data Fusion: A Handbook of Algorithms*. YBS Publishing, 2011.

Helper Functions

```
function Pcnf = helperBernoulliSum(Pd,Mc,Nc)
% Calculate simple Bernoulli sum. Use the start TOC logic, which assumes
% that there is already one hit that initializes the track.
```

```matlab
% Update M and N for probability of deletion
Nd = Nc - 1;      % Need one hit to start counting. Assume first hit initializes track.
Md = Nc - Mc + 1; % Need this many misses to delete

% Bernoulli sum. Probability of deletion calculation.
ii = Md:Nd;
C = arrayfun(@(k) nchoosek(Nd,k),ii);
P = (1 - Pd);
Pdel = sum(C.*P(:).^ii.*(1 - P(:)).^(Nd - ii),2);

% Probability of confirmation
Pcnf = 1 - Pdel;
end

function helperUpdate(calculationType,elapsedTime)
% Output elapsed time
fprintf('%s calculation completed. Total computation time is %.4f seconds.\n', ...
        calculationType,elapsedTime);
end

function varargout = helperPlot(x,y,displayName,xAxisName,yAxisName,titleName,varargin)
% Create a plot with logarithmic scaling on the x-axis

% Create a figure
figure('Name',titleName)
hAxes = gca;

% Plot data
plot(hAxes,x,y,'LineWidth',2,'DisplayName',displayName,varargin{:})
hold(hAxes,'on')
grid(hAxes,'on')

% Update axes
hAxes.Title.String = titleName;
hAxes.XLabel.String = xAxisName;
hAxes.YLabel.String = yAxisName;

% Make sure legend is on and in best location
legend(hAxes,'Location','Best')

% Set axes as optional output
if nargout == 1
    varargout{1} = hAxes;
end
end

function varargout = helperPlotLog(x,y,displayName,xAxisName,yAxisName,titleName,varargin)
% Create a plot with logarithmic scaling on the x-axis

% Create a figure
figure('Name',titleName)
hAxes = gca;

% Plot data
numCol = size(y,2);
for ii = 1:numCol
    idxX = min(ii,size(x,2));
    hLine = semilogx(hAxes,x(:,idxX),y(:,ii),'LineWidth',2,varargin{:});
```

```matlab
        if ischar(displayName)
            hLine.DisplayName = displayName;
        else
            hLine.DisplayName = sprintf('SNR (dB) = %.2f',displayName(ii));
        end
        hold on
    end
    grid on

    % Update axes
    hAxes.Title.String = titleName;
    hAxes.XLabel.String = xAxisName;
    hAxes.YLabel.String = yAxisName;

    % Make sure legend is on and in best location
    legend(hAxes,'Location','Best')

    % Set axes as optional output
    if nargout == 1
        varargout{1} = hAxes;
    end
end

function helperAddPlotLog(hAxes,x,y,displayName,varargin)
% Add an additional plot to the axes hAxes with logarithmic scaling on the
% x-axis

% Plot data
hold on
hLine = semilogx(hAxes,x,y,'LineWidth',2,varargin{:});
hLine.DisplayName = displayName;
end

function gs = helperCalculateGateSize(N,trkfilt,res)
% Calculate a gate growth sequence in bins

% Initialize tracker gate growth sequence
gs = zeros(1,N); % Gate growth sequence

% Calculate gate growth sequence by projecting state uncertainty using
% linear approximations.
areaBin = prod(res(:),1);
for n = 1:N
    [~,Ppred] = predict(trkfilt); % Predict

    % Calculate the products of the 1-sigma values
    E = eig(Ppred);
    E(E<0) = 0; % Remove negative values
    sigma1Prod = sqrt(prod(E(:),1));

    % Calculate error ellipse area
    areaErrorEllipse = pi*sigma1Prod; % Area of ellipse = pi*a*b

    % Translate to bins
    gs(n) = max(ceil(areaErrorEllipse/areaBin),1);
end
end
```

```matlab
function helperColorZonesReqPfa(hAxes,req)
% Plot color zones for requirement type 1

% Vertical requirement line
xline(req,'--','DisplayName','Requirement',...
    'HitTest','off');

% Get axes limits
xlims = get(hAxes,'XLim');
ylims = get(hAxes,'YLim');

% Green box
pos = [xlims(1) ylims(1) req ylims(2)];
x = [pos(1) pos(1) pos(3) pos(3) pos(1)];
y = [pos(1) pos(4) pos(4) pos(1) pos(1)];
hP = patch(hAxes,x,y,[0.4660 0.6740 0.1880], ...
    'FaceAlpha',0.3,'EdgeColor','none','DisplayName','Requirement Met');
uistack(hP,'bottom');

% Red box
pos = [req ylims(1) xlims(2) ylims(2)];
x = [pos(1) pos(1) pos(3) pos(3) pos(1)];
y = [pos(1) pos(4) pos(4) pos(1) pos(1)];
hP = patch(hAxes,x,y,[0.6350 0.0780 0.1840], ...
    'FaceAlpha',0.3,'EdgeColor','none','DisplayName','Requirement Not Met',...
    'HitTest','off');
uistack(hP,'bottom');
end

function helperColorZonesReqPdt(hAxes,req)
% Plot color zones for requirement type 2

% Horizontal requirement line
yline(req,'--','DisplayName','Requirement',...
    'HitTest','off')

% Get axes limits
xlims = get(hAxes,'XLim');
ylims = get(hAxes,'YLim');

% Green box
pos = [xlims(1) req xlims(2) ylims(2)];
x = [pos(1) pos(1) pos(3) pos(3) pos(1)];
y = [pos(1) pos(4) pos(4) pos(1) pos(1)];
hP = patch(hAxes,x,y,[0.4660 0.6740 0.1880], ...
    'FaceAlpha',0.3,'EdgeColor','none','DisplayName','Requirement Met',...
    'HitTest','off');
uistack(hP,'bottom');

% Red box
pos = [xlims(1) req xlims(2) req];
x = [pos(1) pos(1) pos(3) pos(3) pos(1)];
y = [pos(1) pos(4) pos(4) pos(1) pos(1)];
hP = patch(hAxes,x,y,[0.6350 0.0780 0.1840], ...
    'FaceAlpha',0.3,'EdgeColor','none','DisplayName','Requirement Not Met',...
    'HitTest','off');
uistack(hP,'bottom');
end
```

```matlab
function helperPrintTrackProbabilities(Pdt,Pft,Eft)
% Print out results

fprintf('Probability of Target Track in Presence of False Alarms = %.4f\n',Pdt)
fprintf('Probability of False Track in the Presence of Targets = %.4e\n',Pft)
fprintf('Expected Number of False Tracks = %d\n',Eft)
end

function helperPlotROCTOC(reqPdt,Pfa,Pd,snrdB,PdtMat,cp)
% Plot ROC/TOC

% Plot ROC curves
figure
hAxesROC = subplot(2,1,1);
plot(hAxesROC,snrdB,Pd,'-o')
title(hAxesROC,'Receiver Operating Characteristic (ROC)')
xlabel(hAxesROC,'SNR (dB)')
ylabel(hAxesROC,'P_D')
grid(hAxesROC,'on')
legend(hAxesROC,sprintf('%.1e',Pfa),'Location','Best')

% Plot SNR versus probability of target track
hAxesTOC = subplot(2,1,2);
numCp = size(cp,1);
for ii = 1:numCp
    plot(hAxesTOC,snrdB,PdtMat(:,ii),'-o', ...
        'DisplayName',sprintf('%d/%d',cp(ii,1),cp(ii,2)))
    hold(hAxesTOC,'on')
end
title(hAxesTOC,'SNR versus P_{DT}')
xlabel(hAxesTOC,'SNR (dB)')
ylabel(hAxesTOC,'P_{DT}')
grid(hAxesTOC,'on')
legend(hAxesTOC,'Location','Best')

% Label points
colorVec = get(hAxesROC,'ColorOrder');
numSnr = numel(snrdB);
textArray = arrayfun(@(x) sprintf('  %d',x),1:numSnr,'UniformOutput',false).';
xPosROC = snrdB;
colorFont = brighten(colorVec,-0.75);
numColors = size(colorVec,1);
idxC = mod(1:numCp,numColors); % Use only available default colors
idxC(idxC == 0) = numColors; % Do not let color index equal 0

% Label points ROC
yPosROC = Pd;
text(hAxesROC,xPosROC,yPosROC,textArray,'FontSize',6,'Color',colorFont(1,:),'Clipping','on')

% Label points TOC
xPosTOC = snrdB;
for ii = 1:numCp
    yPosTOC = PdtMat(:,ii);
    text(hAxesTOC,xPosTOC,yPosTOC,textArray,'FontSize',6,'Color',colorFont(idxC(ii),:),'Clipping
end

% Add requirement zone color blocks
```

```matlab
    helperColorZonesReqPdt(hAxesTOC,reqPdt);
end

function helperPrintReqValues(reqPdt,Pd,snrdB,PdtMat,EftMat,cp)
% Output information about required values given a required probability of
% target track requirement

% Get values
numCp = size(PdtMat,2);
reqPd = zeros(1,numCp);
reqSNRdB = zeros(1,numCp);
expEft = zeros(1,numCp);
for ii = 1:numCp
    reqPd(ii) = interp1(PdtMat(:,ii),Pd,reqPdt);
    reqSNRdB(ii) = interp1(PdtMat(:,ii),snrdB,reqPdt);
    expEft(ii) = interp1(PdtMat(:,ii),EftMat(:,ii),reqPdt);
end

 % Display required probability of detection, SNR, and expected false
 % tracks
 for ii = 1:numCp
    fprintf('Confirmation Threshold           = %d/%d\n',cp(ii,1),cp(ii,2));
    fprintf('Required Probability of Detection = %.2f\n',reqPd(ii));
    fprintf('Required SNR (dB)                = %.2f\n',reqSNRdB(ii));
    fprintf('Expected Number of False Tracks  = %d\n\n',expEft(ii));
 end
end
```

Modeling Radar Detectability Factors

This example shows how to model antenna, transmitter, and receiver gains and losses for a detailed radar range equation analysis. We start by computing the available signal-to-noise ratio (SNR) at the radar receiver using the SNR form of the radar equation. Next, we define the detectability factor as a threshold SNR required to make a detection with specified probabilities of detection, P_d, and false alarm, P_{fa}. We then estimate the maximum range of the system as the range at which the available SNR is equal to the detectability factor, that is the maximum range at which the detection with the specified P_d and P_{fa} is still possible. The example further explores the impact of losses introduced by different components of the radar system on the estimated maximum range. We first consider the effects of the sensitivity time control (STC) and eclipsing on the available SNR. Scanning and signal processing losses that require an increase in the radar detectability factor are considered next. The example concludes by computing the resulting P_d at the detector output to show the impact of the losses on the detection performance of the radar system.

Available SNR

The radar equation combines the main parameters of a radar system and allows a radar engineer to compute a maximum detection range, a required peak transmit power, or a maximum available SNR for a radar system. The radar equation is typically a family of relatively simple formulas each corresponding to one of these three key performance characteristics. A common form of the radar equation for computing the maximum available SNR at a range R is:

$$SNR = \frac{P_t \tau G_t G_r \lambda^2 \sigma}{(4\pi)^3 k T_s R^4 L}$$

where

P_t is the peak transmit power

τ is the transmitted pulse width

G_t is the transmit antenna gain

G_r is the receive antenna gain

λ is the radar wavelength

σ is the radar target cross section (RCS)

k is Boltzmann's constant

T_s is the system noise temperature

L is the general loss factor that combines losses along the transmitter-target-receiver path that reduce the received signal energy.

On the right-hand side all parameters except the target range and RCS are under control of the radar designer. This equation states that for a target of a given size located at a range R, the SNR available at the receiver can be increased by transmitting more power, increasing the antenna size, using lower frequency, or having a more sensitive receiver.

Consider an S-band airport surveillance radar operating at the frequency of 3 GHz. The peak transmit power is 0.2 MW, the transmit and receive antenna gain is 34 dB, the pulse duration is 11 μs, and the noise figure is 4.1 dB. Assume the radar is required to detect a target with 1 m2 RCS at the maximum range R_m of 100 km.

```matlab
lambda = freq2wavelen(3e9);          % Wavelength (m)
Pt = 0.2e6;                          % Peak power (W)
tau = 1.1e-5;                        % Pulse width (s)
G = 34;                              % Transmit and receive antenna gain (dB)
Ts = systemp(4.1);                   % System temperature (K)
rcs = 1;                             % Target radar cross section (m^2)
Rm = 100e3;                          % Required maximum range (m)
```

To start, assume no losses, that is $L = 0$ dB. We use the radar equation to compute the available SNR at the receiver as a function of target range.

```matlab
L = 0;                               % Combined transmission line and propagation losses (dB)
R = (1:40:130e3).';                  % Range samples (m)
SNR = radareqsnr(lambda,R,Pt,tau,'Gain',G,'Ts',Ts,'RCS',rcs,'Loss',L);
```

Calculate the available SNR at the required maximum range of 100 km.

```matlab
SNRatRm = SNR(find(R>=Rm,1))
```

```
SNRatRm = 18.3169
```

Plot the maximum range requirement together with the computed available SNR.

```matlab
radarmetricplot(R*1e-3,SNR,'MetricName','Available SNR','MaxRangeRequirement',Rm*1e-3,...
    'RangeUnit','km','MetricUnit','dB','RadarName',{'Surveillance Radar'});
legend('Location','best');
```

Available SNR vs Range
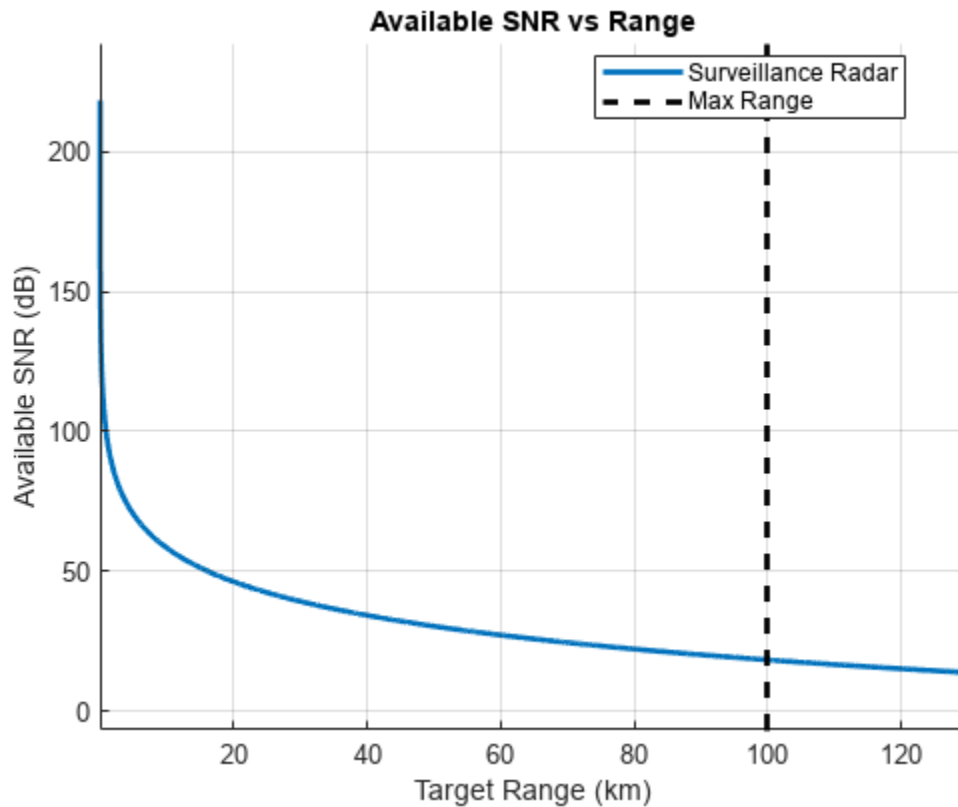
Required SNR

Is the computed available SNR high enough to make a detection? Since the signal processed by a radar receiver is a combination of the transmitted waveform and the random noise, the answer to this question depends on the desired probability of detection P_d and the maximum acceptable probability of false alarm P_{fa}. These probabilities define the required SNR, also known as the detectability factor (or detectability). The detectability factor is the minimum SNR required to declare a detection with the specified probabilities of detection and false alarm. It also depends on the RCS fluctuation and the type of the detector. Compute the detectability factor for a single pulse received from a steady (Swerling 0) target by a square-law detector assuming P_d = 0.9 and P_{fa} = 1e-6.

```
Pd = 0.9;
Pfa = 1e-6;
D0 = detectability(Pd,Pfa,1,'Swerling0')
```

```
D0 = 13.1217
```

Calculate the detectability factor for a Swerling 1 fluctuating target, which is a more accurate model for real-world targets. For the Swerling 1 target the single-pulse detectability factor is significantly higher.

```
D1 = detectability(Pd,Pfa,1,'Swerling1')
```

```
D1 = 21.1436
```

The resultant required SNR is higher than the available SNR, which means that a Swerling 1 target will not be detected with a single pulse. A common way to lower the detectability factor is to perform pulse integration. Calculate the detectability factor for $N = 10$ noncoherently integrated pulses.

```
N = 10;
DN = detectability(Pd,Pfa,N,'Swerling1')
```

```
DN = 13.5033
```

This is lower than the available SNR. Thus, after a non-coherent integration of 10 pulses the radar system will be able to detect a 1 m2 target at the required maximum range of 100 km with the probability of detection 0.9 and a false alarm of 1e-6.

The detectability factor computed for a Swerling 1 target and N pulses combines the effects of the integration gain and the fluctuation loss. The integration gain is a difference between the SNR required to detect a steady target using a single pulse and the SNR required to detect a steady target using N pulses.

```
Gi = detectability(Pd,Pfa,1,'Swerling0') - detectability(Pd,Pfa,N,'Swerling0')
```

```
Gi = 7.7881
```

The fluctuation loss is a difference between the SNR required to detect a fluctuating target and the SNR required to detect a steady target.

```
Lf = detectability(Pd,Pfa,N,'Swerling1') - detectability(Pd,Pfa,N,'Swerling0')
```

```
Lf = 8.1696
```

Use the `radarbudgetplot` function to illustrate the components of the detectability factor.

```
radarbudgetplot([D0 -Gi Lf], {'Single-pulse steady target','Pulse integration gain','Fluctuation
title('Detectability Factor')
```
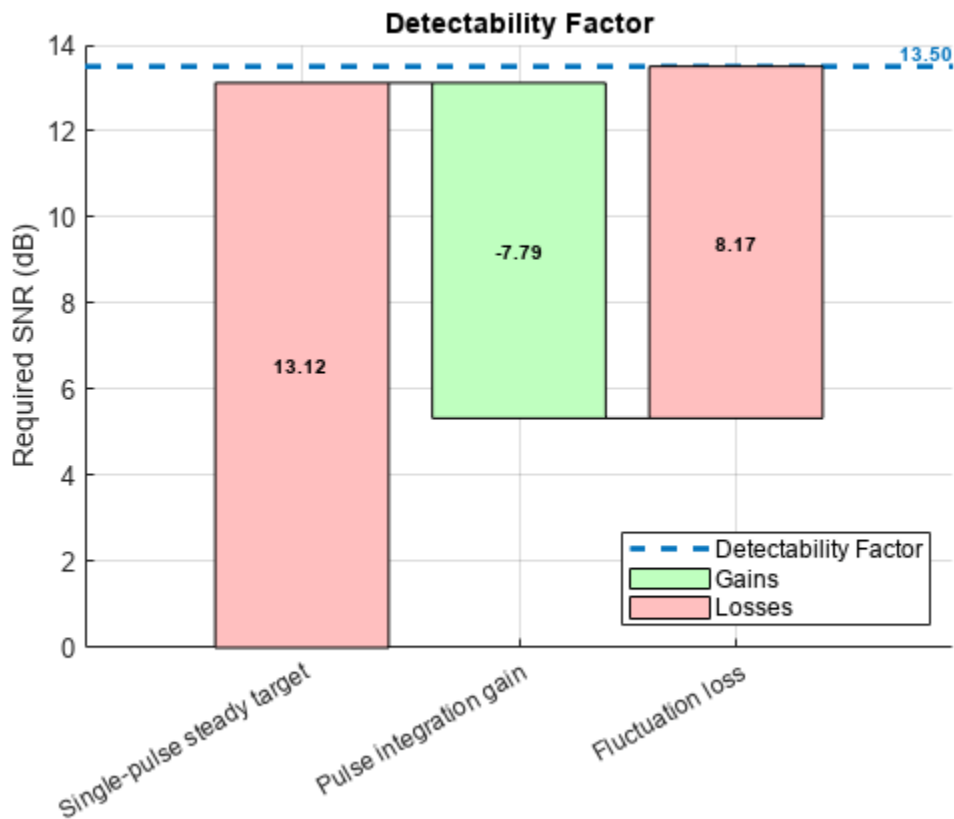
Substitute the detectability factor into the range form of the radar equation as the minimum required SNR to evaluate the actual maximum range of the system.

```
radareqrng(lambda,DN,Pt,tau,'Gain',G,'Ts',Ts,'RCS',rcs,'Loss',L,'unitstr','km')
```

```
ans = 131.9308
```

To clearly indicate at which ranges the detection with the desired P_d and the maximum acceptable P_{fa} is possible, we add the computed detectability factor as a horizontal line to the SNR vs Range plot. We also use a stoplight chart to color code ranges and SNR levels according to the computed detectability. At the ranges where the available SNR curve passes through the green zone the radar meets the detection requirement, while at the range where it is in the red zone the detection with the specified P_d and P_{fa} is not possible.

```
radarmetricplot(R*1e-3,SNR,DN, ...
    'MetricName','Available SNR', ...
    'RequirementName','Detectability', ...
    'MaxRangeRequirement',Rm*1e-3, ...
    'RangeUnit','km','MetricUnit','dB', ...
    'ShowStoplight',true, ...
    'RadarName',{'Surveillance Radar'});
title([{'Available SNR vs Range'}, {'(No Losses)'}]);
legend('Location','best');
```

Available SNR vs Range (No Losses)
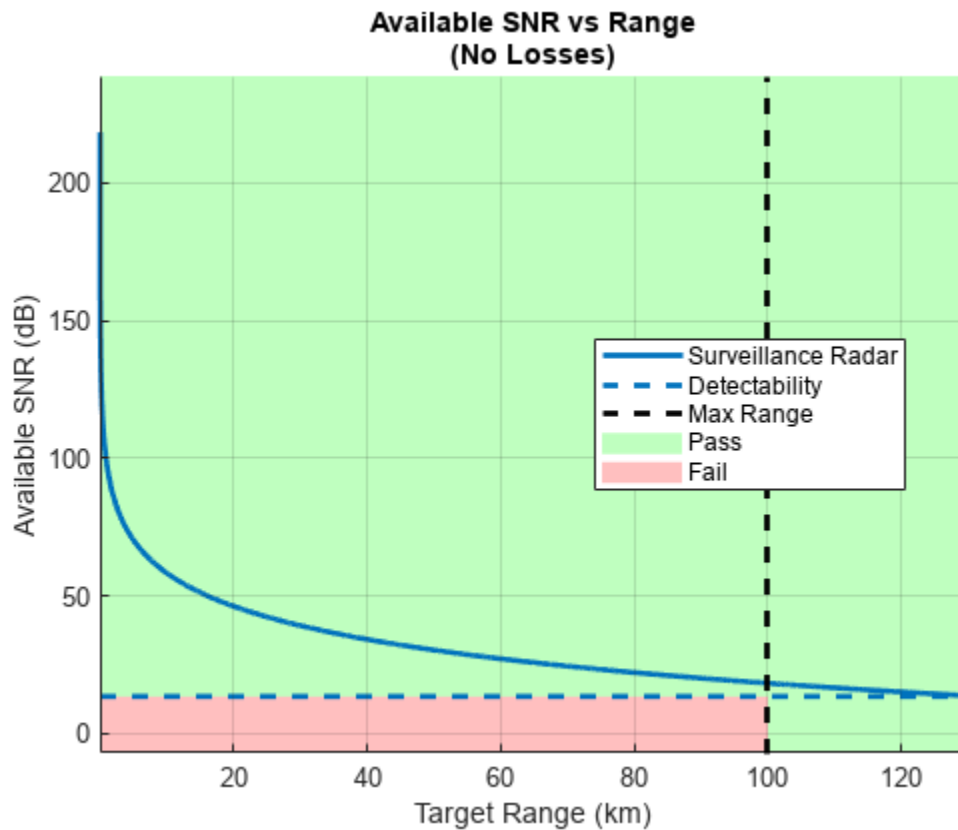
All ranges beyond the required maximum range are colored green and marked as Pass.

This analysis assumes zero losses and therefore cannot adequately predict the range of the actual radar system. A real radar system with the specified parameters will have a shorter maximum range due to:

- Propagation effects caused by the earth's surface and atmosphere. These effects reduce the amount of available signal energy at the receiver.
- Various losses experienced throughout the radar system. Some losses in this category reduce the available SNR, while other losses result in increased detectability factor.

The following sections consider in more detail the impact of the losses belonging to the second category on the range performance of the radar system.

Range-dependent Factors

When designing a surveillance radar system, several factors must be included in the radar equation to account for the decrease in the available signal energy at the receiver.

Eclipsing

Pulse radar systems turn off their receivers during the pulse transmission. Thus, the target echoes arriving from the ranges within one pulse length from the radar or within one pulse length around the unambiguous range will be eclipsed by the transmitted pulse resulting in only a fraction of the pulse being received and processed. The radar system considered in this example has a pulse width of $11\mu s$. The closest range from which a full pulse can be received is the minimum range R_{min}.

```
Rmin = time2range(tau)
```

```
Rmin = 1.6489e+03
```

The echoes from the targets that are closer than 1649 m will arrive before the pulse transmission has been completed. Similar effect can be observed for the targets located at or near the multiples of the unambiguous range. Assuming the pulse repetition frequency is 1350 Hz (pulse repetition interval $T \approx 0.75$ ms), calculate the unambiguous range of the system.

```
prf = 1350;                          % Pulse repetition frequency
Rua = time2range(1/prf)
```

```
Rua = 1.1103e+05
```

The echoes arriving from the ranges $R_{ua} \pm R_{min}$ will be eclipsed by the next transmitted pulse. This diagram illustrates pulse eclipsing. The arrowheads indicate the front of the pulse.
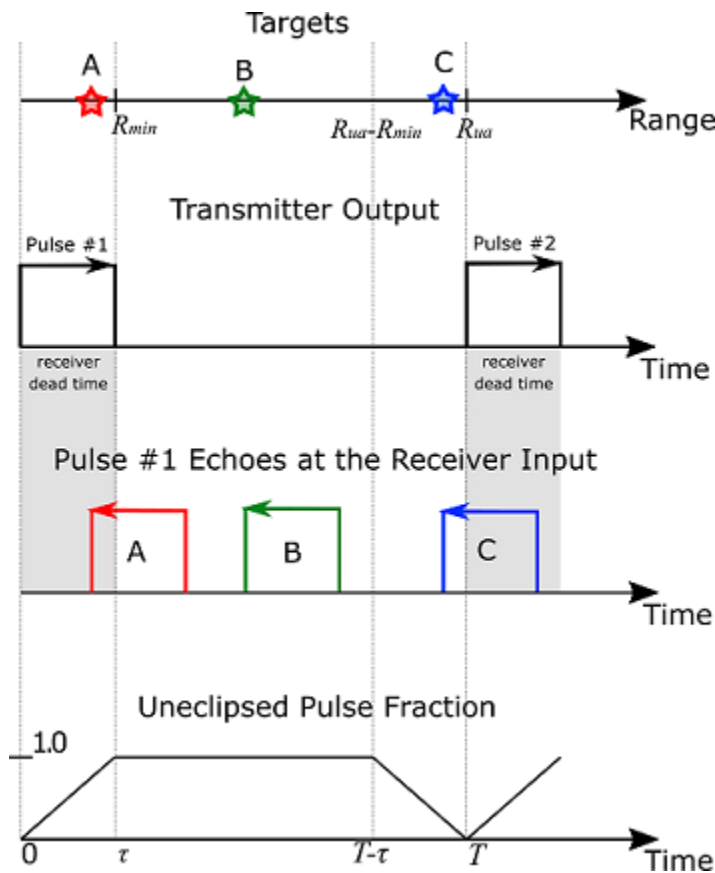
Due to eclipsing, the available SNR will have deep notches at 0 range and the ranges equal to the multiples of R_{ua}. Add the eclipsing factor to the radar equation in order to account for the loss in the available SNR due to pulse eclipsing.

```
Du = tau*prf;                        % Duty cycle
Fecl = eclipsingfactor(R,Du,prf);    % Eclipsing factor
```

```
SNR = radareqsnr(lambda,R,Pt,tau,'Gain',G,'Ts',Ts,'RCS',rcs,'CustomFactor',Fecl,'Loss', L);
```

```
radarmetricplot(R*1e-3,SNR,DN, ...
```

```
    'MetricName','Available SNR', ...
    'RequirementName','Detectability', ...
    'MaxRangeRequirement',Rm*1e-3, ...
    'RangeUnit','km','MetricUnit','dB', ...
    'ShowStoplight',true, ...
    'RadarName',{'Surveillance Radar'});
title([{'Available SNR vs Range'}, {'(With Eclipsing)'}]);
legend('Location','best');
```
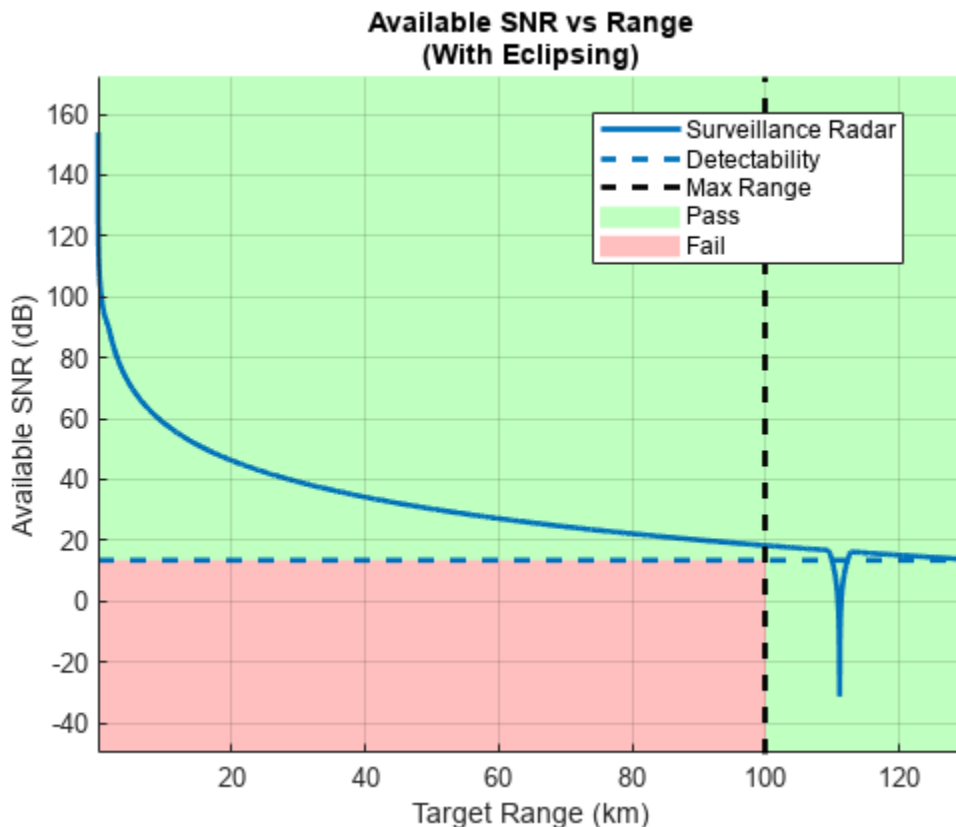

The real-world radar systems utilize PRF diversity in order to prevent the eclipsing loss and to extend the unambiguous range of the system.

Sensitivity Time Control (STC)

A typical surveillance radar system must transmit a considerable amount of power to detect targets at long ranges. Although the available energy rapidly decays with range, at very close ranges even small targets can have very strong returns due to high peak transmit power. Such strong returns from small nuisance targets (birds, insects) might result in undesirable detections, while the regular size targets or nearby clutter can saturate the receiver. It is highly desirable for the surveillance radar system to avoid these kind of nuisance detections. To solve this problem the radar systems use STC. It scales the receiver gain up to a cutoff range R_{stc} to maintain a constant signal strength as a target approaches the radar.

```
Rstc = 60e3;                    % STC cutoff range (m)
Xstc = 4;                       % STC exponent selected to maintain target detectability at
Fstc = stcfactor(R,Rstc,Xstc);  % STC factor
```

```
SNR = radareqsnr(lambda,R,Pt,tau,'Gain',G,'Ts',Ts,'RCS',rcs,'CustomFactor',Fecl+Fstc,'Loss',L);

radarmetricplot(R*1e-3,SNR,DN, ...
    'MetricName','Available SNR', ...
    'RequirementName','Detectability', ...
    'MaxRangeRequirement',Rm*1e-3, ...
    'RangeUnit','km','MetricUnit','dB', ...
    'ShowStoplight',true, ...
    'RadarName',{'Surveillance Radar'});
title([{'Available SNR vs Range'}, {'(With STC and Eclipsing for 1 m^2 Target)'}]);
legend('Location','best');
ylim([-30 30])
```
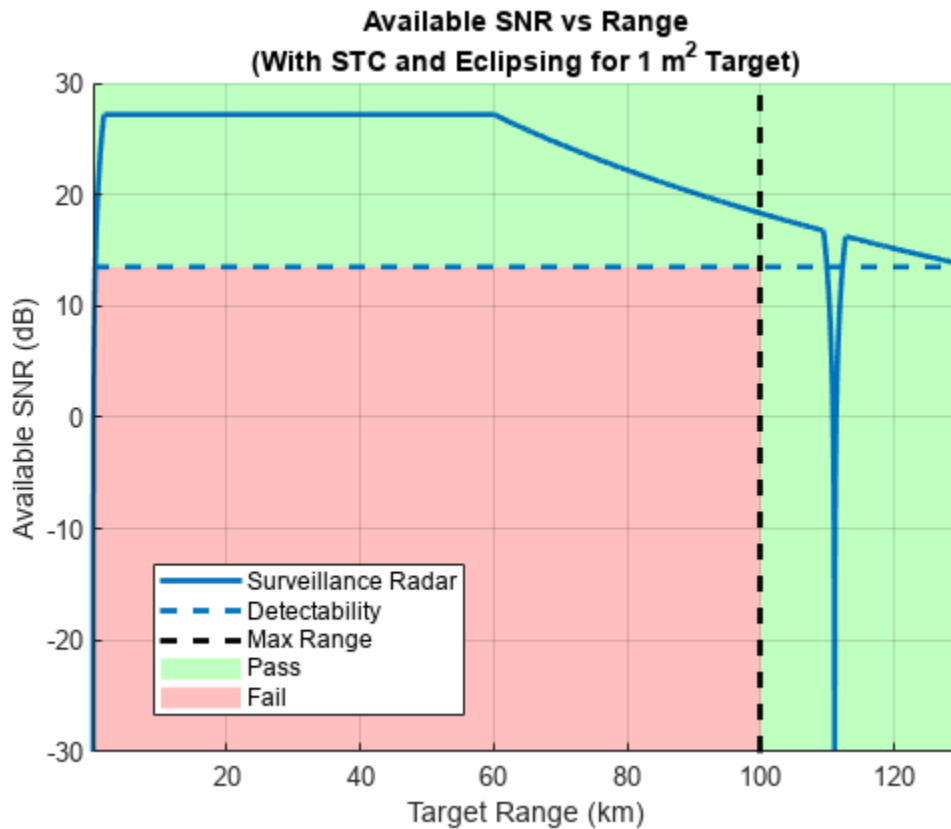

After adding the STC factor, the plot shows that the 1 m2 RCS target is still detected everywhere up to the maximum range R_m, while a small target with RCS of 0.03 m2 will not be able to reach the required P_d of 0.9 at any range and thus will be rejected.

```
SNRsmallRCS = radareqsnr(lambda,R,Pt,tau,'Gain',G,'Ts',Ts,'RCS',0.03,'CustomFactor',Fecl+Fstc,'Lo

radarmetricplot(R*1e-3,SNRsmallRCS,DN, ...
    'MetricName','Available SNR', ...
    'RequirementName','Detectability', ...
    'MaxRangeRequirement',Rm*1e-3, ...
    'RangeUnit','km','MetricUnit','dB', ...
    'ShowStoplight',true, ...
    'RadarName',{'Surveillance Radar'});
title([{'Available SNR vs Range'}, {'(With STC and Eclipsing for 0.03 m^2 Target)'}]);
```

```
legend('Location','best');
ylim([-30 20])
```

Available SNR vs Range
(With STC and Eclipsing for 0.03 m^2 Target)
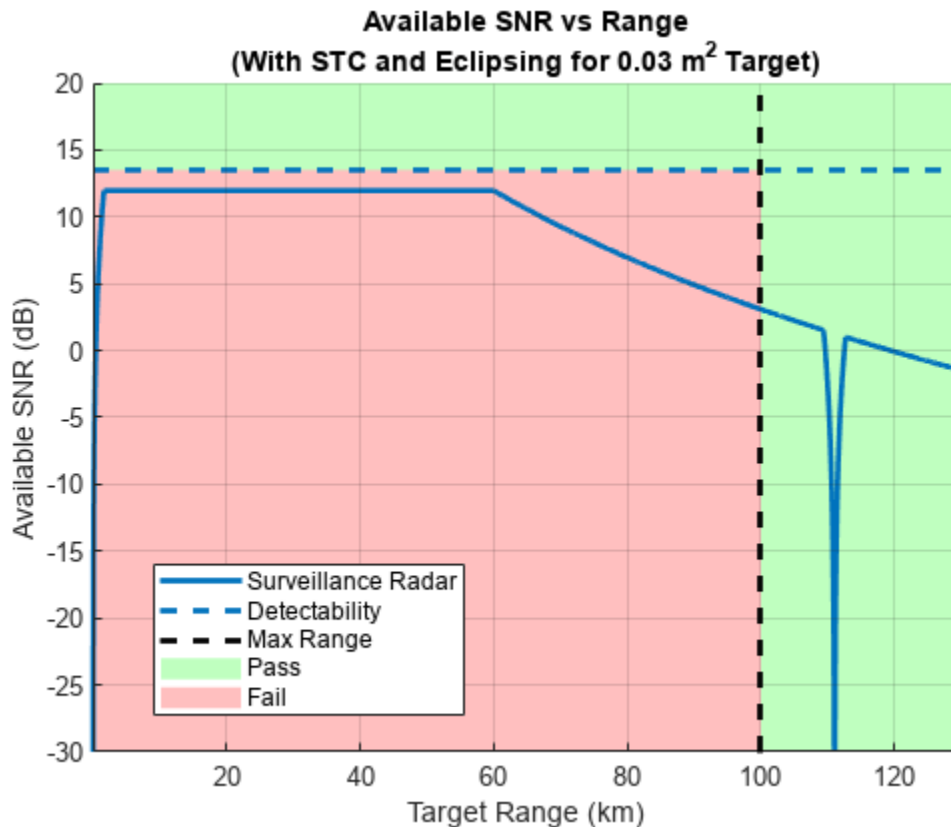
It is clear from these plots that the STC only scales the available SNR up to the specified cutoff range and does not affect the available SNR at the maximum range of interest.

Scanning

A radar system can scan a search volume either by mechanically rotating the antenna or by using the phased array antenna and performing electronic scanning. An imperfect shape of the antenna beam and the process of sweeping the beam across a search volume introduces additional losses to the system.

Beam Shape Loss

The radar equation uses a peak value of the antenna gain assuming each received pulse has the maximum amplitude. In reality, as the beam passes the target, the received pulses are modulated by the two-way pattern of the scanning antenna resulting in beam shape loss. Computing the exact value of this loss would require knowing the exact antenna pattern. This information might not be available in the early stages of the radar system design when this type of analysis is usually performed. Instead, the shape of the main lobe of a typical practical antenna can be well approximated by a Gaussian shape. Assuming that the radar system performs dense sampling in the spatial domain (the beam moves by less than 0.71 of the half power beamwidth), calculate the beam shape loss for one dimensional scanning.

```
Lb = beamloss
```

```
Lb = 1.2338
```

The beam shape loss is doubled if the radar system scans in both azimuth and elevation.

```
beamloss(true)
```

```
ans = 2.4677
```

Scan Sector Loss

In this example we assume the radar system employs an electronically steered phased array to perform scanning. Using the phased array antenna will cause an increase in the required SNR due to two effects: 1) beam broadening due to the reduced projected array area in the beam direction, and 2) reduction of the effective aperture area of the individual array elements at off-broadside angles. To account for these effects, add the scan sector loss to the detectability factor. Assume that the system in the example scans only in the azimuth dimension and the scan sector spans from –60 to 60 degrees. Compute the resultant loss.

```
theta = [-60 60];
Larray = arrayscanloss(Pd,Pfa,N,theta,'Swerling1')
```

```
Larray = 2.7745
```

Signal Processing

Prior to detection the received radar echoes must pass through the radar signal processing chain. The purpose of different components in the signal processing chain is to guarantee the required probabilities of detection and false alarm, reject unwanted echoes from clutter, and account for variable or non-Gaussian noise. We further consider several components of the signal processing loss that must be accounted for in a surveillance radar system.

MTI

Moving target indicator (MTI) is a process of rejecting fixed or slowly moving clutter while passing echoes from targets moving with significant velocities. Typical MTI uses 2, 3, or 4-pulse canceller that implements a high-pass filter to reject echoes with low Doppler shifts. Passing the received signal through the MTI pulse canceller introduces correlation between the noise samples. This in turn reduces the total number of independent noise samples available for integration resulting in MTI noise correlation loss. Additionally, the MTI canceller significantly suppresses targets with velocities close to the nulls of its frequency response causing an additional MTI velocity response loss. Assuming a 2-pulse canceller is used, calculate these two components of the MTI loss.

```
m = 2;
[Lmti_a, Lmti_b] = mtiloss(Pd,Pfa,N,m,'Swerling1')
```

```
Lmti_a = 1.4468
```

```
Lmti_b = 8.1562
```

In a system that uses a single PRF, the MTI velocity response loss can be very high for the high required probability of detection. To eliminate this loss, PRF diversity is almost always used in real radar systems.

Binary Integration

Binary integration is a suboptimal noncoherent integration technique also known as the M-of-N integration. If M out of N received pulses exceed a predetermined threshold, a target is declared to

be present. The binary integrator is a relatively simple automatic detector and is less sensitive to the effects of a single large interference pulse that might exist along with the target echoes. Therefore, the binary integrator is more robust when the background noise or clutter is non-Gaussian. Since the binary integration is a suboptimal technique, it results in a binary integration loss compared to optimal noncoherent integration. The optimal value of M is not a sensitive selection and it can be quite different from the optimum without significant penalty resulting in the binary integration loss being lower than 1.4 dB. Calculate the binary integration loss when N is 10 and M is set to 6.

```
M = 6;
Lbint = binaryintloss(Pd,Pfa,N,M)
```

Lbint = 1.0549

The `binaryintloss` function computes the loss assuming a steady (Swerling 0) target. Since the fluctuation loss is included in the detectability factor, the same binary integration loss computation can be used in the case of a fluctuating target.

CFAR

Constant false alarm rate (CFAR) detector is used to maintain an approximately constant rate of false target detections when the noise or the interference levels vary. Since CFAR averages a finite number of reference cells to estimate the noise level, the estimates are subject to an error which leads to a CFAR loss. CFAR loss is an increase in the SNR required to achieve a desired detection performance using CFAR when the noise levels are unknown compared to a fixed threshold with a known noise level. Calculate the CFAR loss assuming that total 120 cells are used for cell-averaging CFAR.

```
Nrc = 120;
Lcfar = cfarloss(Pfa,Nrc)
```

Lcfar = 0.2500

Effective Detectability Factor

The scanning and the signal processing losses increase the detectability factor, which means that more energy is required to make a detection. The resulting detectability factor that includes effects of all these losses is called the effective detectability factor. Use the `radarbudgetplot` function to create a waterfall chart showing the combined effect of the computed scanning and signal processing losses on the detectability factor.

```
D = [D0 -Gi Lf Lmti_a+Lmti_b Lbint Lcfar Larray Lb];
radarbudgetplot(D, {'Single-pulse steady target','Pulse integration gain','Fluctuation loss'...
    'MTI loss', 'Binary integration loss', 'CFAR loss', 'Scan sector loss', 'Beam shape loss'});
title('Detectability Factor')
```
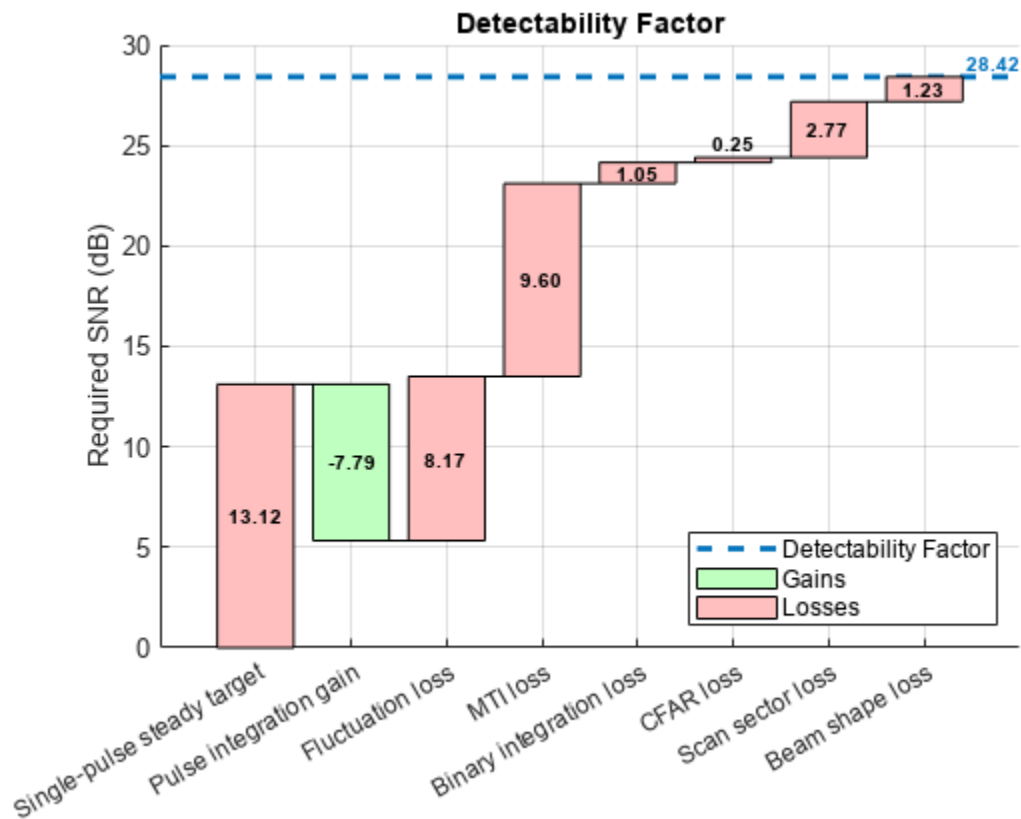
The resulting effective detectability factor equals 28.42 dB. By taking the scanning and the signal processing losses into account, the required SNR increases by almost 15 dB. The analysis shows that the system cannot actually meet the stated requirement of detecting a 1 m2 RCS target at 100 km with P_d = 0.9 and P_{fa} = 1e-6.

```
radarmetricplot(R*1e-3,SNR,sum(D), ...
    'MetricName','Available SNR', ...
    'RequirementName','Detectability', ...
    'MaxRangeRequirement',Rm*1e-3, ...
    'RangeUnit','km','MetricUnit','dB', ...
    'ShowStoplight',true, ...
    'RadarName',{'Surveillance Radar'});
title([{'Available SNR vs Range'}, {'(With STC, Eclipsing, Scanning and Signal Processing Losses)
legend('Location','best')
ylim([-10 30]);
```

Available SNR vs Range
(With STC, Eclipsing, Scanning and Signal Processing Losses)
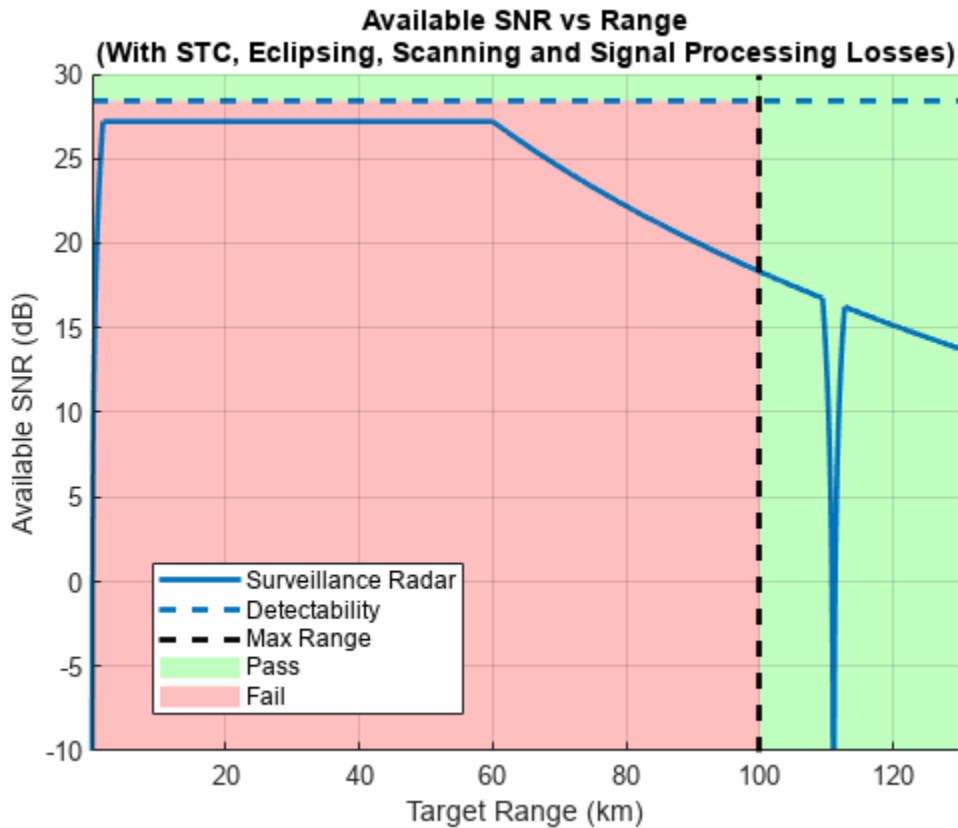
This problem can be addressed either by increasing the available SNR or decreasing the required SNR. Transmitting more power or increasing the antenna gains brings up the available SNR, while increasing the integration time lowers the required SNR. However, in some applications, a subset of the system parameters can be constrained by other requirements and thus cannot be changed. For example, if the analysis is performed for an existing system, increasing the available SNR might not be an option. In that case, making adjustments to the signal processing chain to lower the detectability factor could be an acceptable solution. To lower the required SNR, in the following sections we assume that the number of pulses N is increased from 10 to 40.

In addition, we can change the requirements on the maximum range and the detection probabilities. Instead of a single number specifying the desired probability of detection or the maximum range, a pair of `Objective` and `Threshold` values can be defined. The `Objective` requirement describes the desired performance level of the system that would be needed to fully satisfy the mission needs. The `Threshold` requirement describes a minimum acceptable performance level of the system. Using a pair of values to define a requirement instead of a single value provides more flexibility to the design and creates a trade-space for selecting the system parameters. In this example, we assume that the `Objective` requirement for P_d is 0.9 and set the `Threshold` value to 0.8. Similarly, the `Objective` maximum range requirement remains 100 km, while the `Threshold` value is set to 90 km. The detectability factor is now computed both for the `Objective` and the `Threshold` P_d.

```
N = 40;
M = 18;
Pd = [0.9 0.8];

[Lmti_a, Lmti_b] = mtiloss(Pd,Pfa,N,m,'Swerling1');
```

```
Dx = detectability(Pd,Pfa,N,'Swerling1') + cfarloss(Pfa,Nrc) + beamloss ...
    + Lmti_a + Lmti_b + binaryintloss(Pd,Pfa,N,M) + arrayscanloss(Pd,Pfa,N,theta,'Swerling1')
```

```
Dx = 2×1

   24.2522
   18.0494
```

```
Rm = [100e3 90e3];
radarmetricplot(R*1e-3,SNR,Dx(1),Dx(2), ...
    'MetricName','Available SNR', ...
    'RequirementName','Detectability', ...
    'MaxRangeRequirement',Rm*1e-3, ...
    'RangeUnit','km','MetricUnit','dB', ...
    'ShowStoplight',true, ...
    'RadarName',{'Surveillance Radar'});
title([{'Available SNR vs Range'}, {'(N=40)'}])
legend('Location','best')
ylim([-10 30]);
```
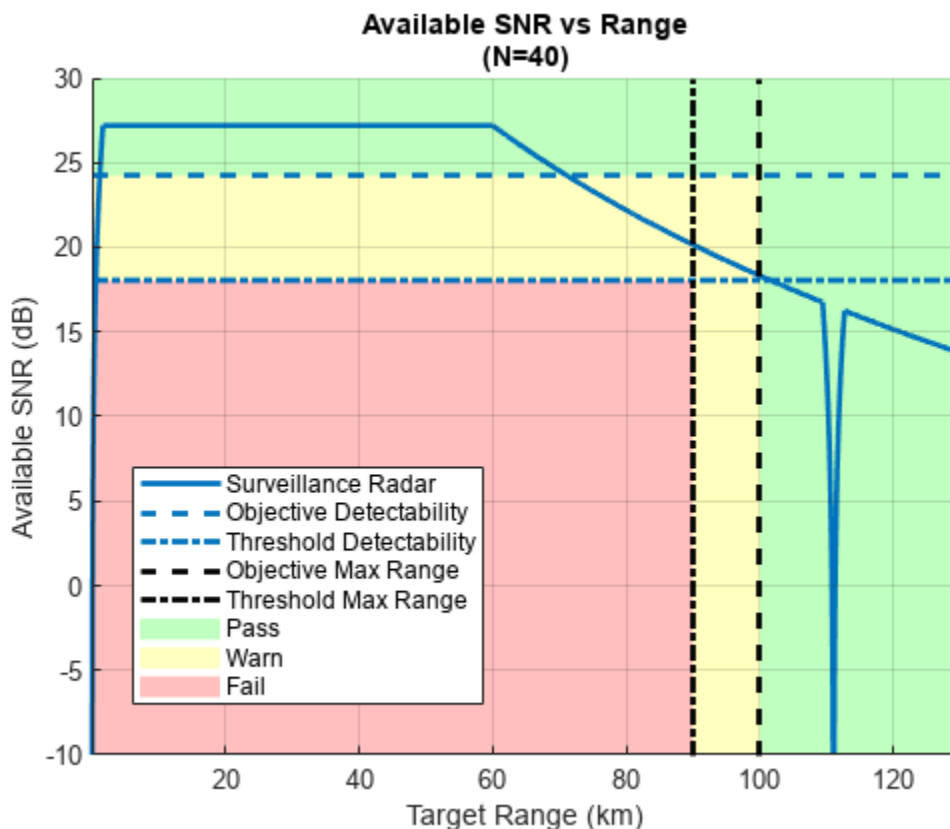

The SNR vs Range plot now has a yellow Warn zone indicating the SNR values and target ranges where the performance of the system is between the `Objective` and the `Threshold` requirements. We can see that up to approximately 70 km the system meets the `Objective` requirement for P_d. From 70 km to 100 km the `Objective` requirement for P_d is violated while the `Threshold` requirement is still satisfied.

Effective Probability of Detection

The SNR vs Range plot above shows that the detection performance of the radar system varies with range. A 1 m2 target at ranges below 70 km will be detected with the probability of detection greater or equal to 0.9, while between 70 km and 100 km it will be detected with P_d of at least 0.8. Since some of the considered losses depend on the probability of detection, the actual P_d at the detector output varies with range. We can use the ROC curve to compute P_d as a function of range.

```
% Generate a vector of probability values at which to compute the ROC curve
p = probgrid(0.1,0.9999,100);

% Compute the required SNR at these probabilities
[lmti_a, lmti_b] = mtiloss(p,Pfa,N,m,'Swerling1');
dx = detectability(p,Pfa,N,'Swerling1') + cfarloss(Pfa,Nrc) + beamloss ...
    + lmti_a + lmti_b + binaryintloss(p,Pfa,N,M) + arrayscanloss(p,Pfa,N,theta,'Swerling1');

% Plot the ROC curve
helperRadarPdVsSNRPlot(dx,p,[0.1 0.9999]);
```
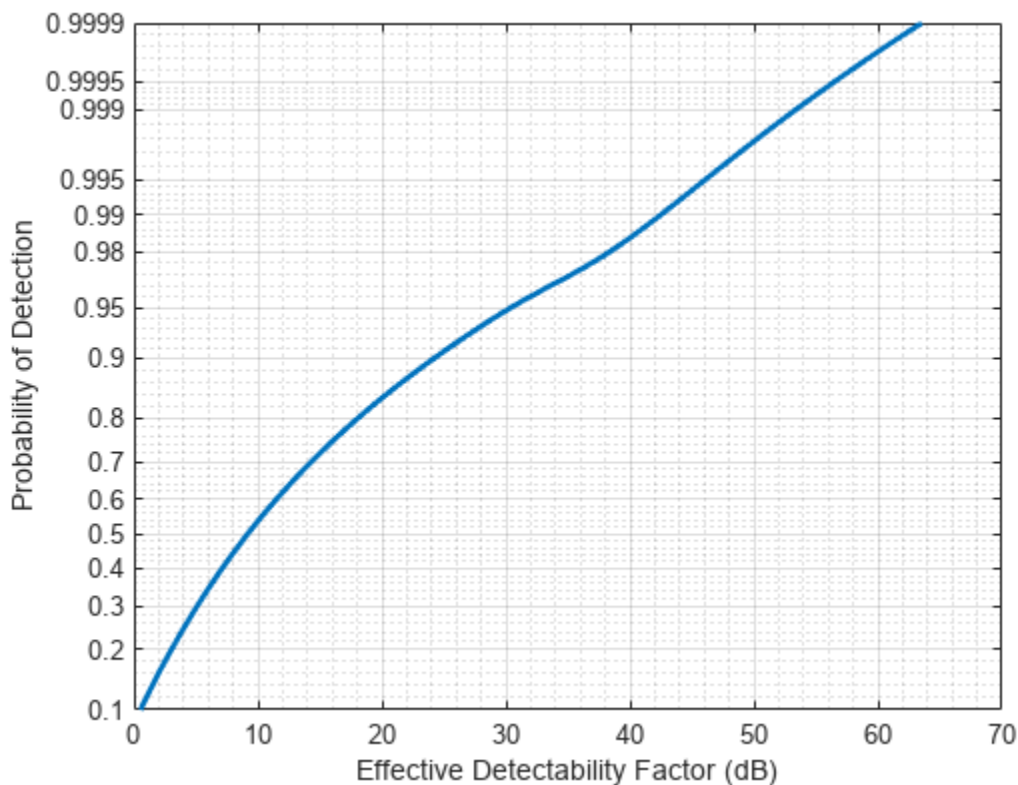

The effective probability of detection at the detector output can now be computed by interpolating this ROC curve at the available SNR values.

```
% Interpolate the ROC curve at the available SNR
Pdeff = rocinterp(dx,p,SNR,'snr-pd');

% Plot the effective Pd as a function of range
radarmetricplot(R*1e-3,Pdeff,Pd(1),Pd(2), ...
```

```
        'MetricName','Effective P_d', ...
        'RequirementName','P_d', ...
        'MaxRangeRequirement',Rm*1e-3, ...
        'RangeUnit','km', ...
        'ShowStoplight',true, ...
        'RadarName',{'Surveillance Radar'});
legend('Location','best')
ylim([0.5 1.0])
```
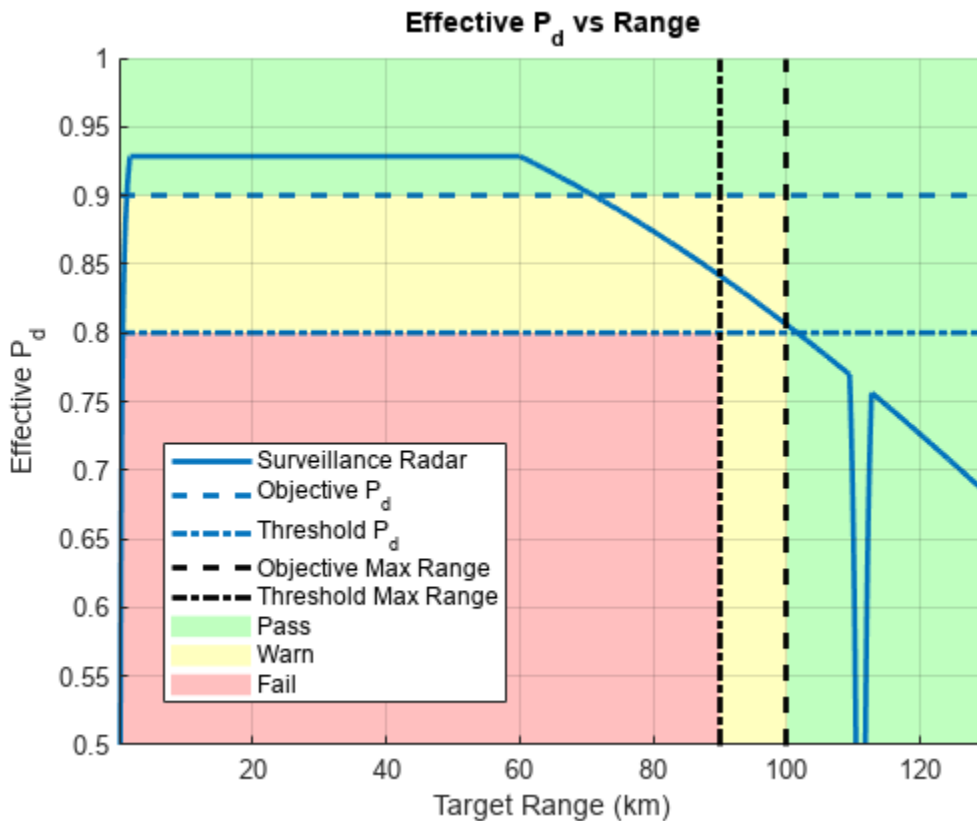

This result shows that due to application of the STC the probability of detection is almost constant for ranges from 2 km to 60 km. For a target with 1 m2 RCS it is higher than 0.92. In the range between 70 km and 87 km the effective P_d is above 0.85. At the Threshold value of the maximum range requirement the probability of detection is approximately 0.84, and at the Objective range of 100 km it is slightly above 0.8.

Summary

This example shows how various losses impact the detection performance of a radar system. It starts with a radar equation and introduces the concepts of the available SNR and the detectability factor. Considering an example surveillance radar system, it shows how the available SNR is reduced by STC and eclipsing, while the detectability factor is increased by scanning and signal processing losses. Finally, the example demonstrates how to compute the effective probability of detection at the receiver output for different target ranges.

References

1 Barton, D. K. *Radar equations for modern radar.* Artech House, 2013.

2 Richards, M. A., Scheer, J. A. & Holm, W. A. *Principle of Modern Radar: Basic Principles.* SciTech Publishing, 2010.

MTI Improvement Factor for Land-Based Radar

This example discusses the moving target indication (MTI) improvement factor and investigates the effects of the following on MTI performance:

- Frequency
- Pulse repetition frequency (PRF)
- Number of pulses
- Coherent versus noncoherent processing

This example also introduces sources of error that limit MTI cancellation. Lastly, the example shows the improvement in clutter-to-noise ratio (CNR) for a land-based, MTI radar system.

MTI Improvement Factor

At a high-level, there are two types of MTI processing, coherent and noncoherent. Coherent MTI refers to the case in which the transmitter is coherent over the number of pulses used in the MTI canceler or when the coherent oscillator of the system receiver is locked to the transmitter pulse, which is also known as a coherent-on-receive system. A noncoherent MTI system uses samples of the clutter to establish a reference phase against which the target and clutter are detected.

The MTI improvement factor I_m is defined as

$$I_m = \frac{C_i}{C_o},$$

where C_i is the clutter power into the receiver and C_o is the clutter power after MTI processing.

Effect of Frequency

Investigate the effect of frequency on MTI performance using the `mtifactor` function. Use a pulse repetition frequency (PRF) of 500 Hz and analyze for 1- through 3-delay MTI cancelers for both the coherent and noncoherent cases.

```
% Setup parameters
m    = 2:4;                     % Number of pulses in (m-1) delay canceler
freq = linspace(1e9,10e9,1000); % Frequency (Hz)
prf  = 500;                     % Pulse repetition frequency (Hz)

% Initialize outputs
numM          = numel(m);
numFreq       = numel(freq);
ImCoherent    = zeros(numFreq,numM);
ImNoncoherent = zeros(numFreq,numM);

% Calculate MTI improvement factor versus frequency
for im = 1:numM
    % Coherent MTI
    ImCoherent(:,im) = mtifactor(m(im),freq,prf,'IsCoherent',true);

    % Noncoherent MTI
    ImNoncoherent(:,im) = mtifactor(m(im),freq,prf,'IsCoherent',false);
end
```

```
% Plot results
helperPlotLogMTI(m,freq,ImCoherent,ImNoncoherent);
```


Frequency versus MTI Improvement
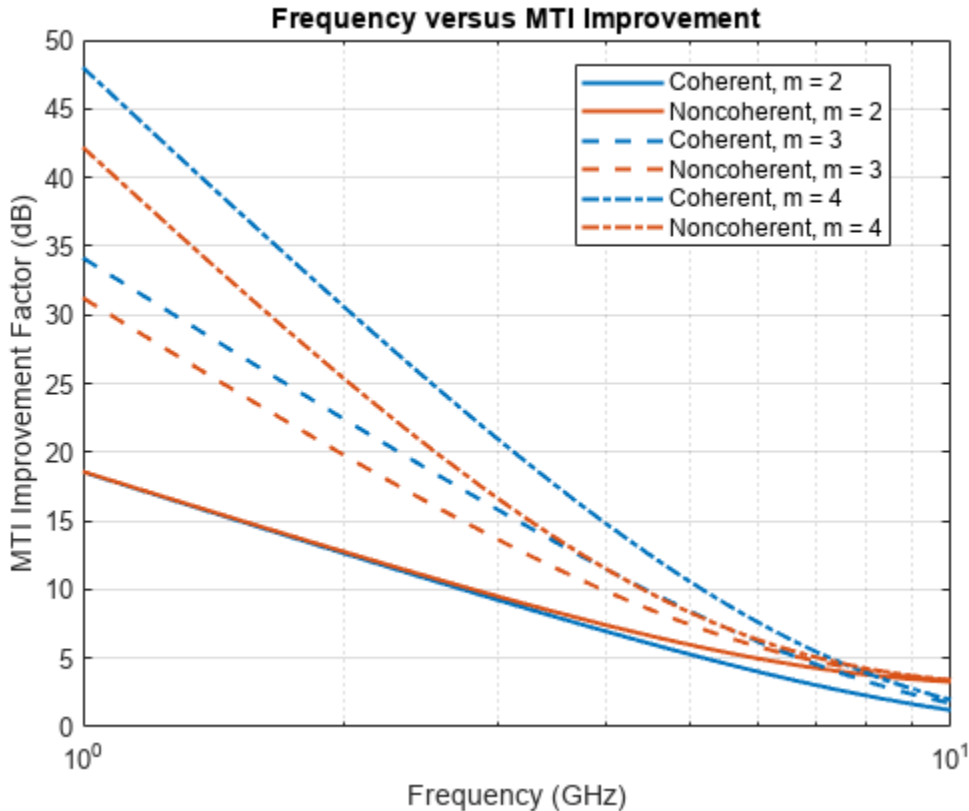
There are several takeaways from the results. First, the difference between the coherent and noncoherent results decreases with increasing frequency for the same m for the m = 3 and 4 cases. The results for the m = 2 case show that the improvement factor is very similar at lower frequencies but performance diverges at higher frequencies. Second, increasing m improves the clutter cancellation for both coherent and noncoherent MTI. Third, when the PRF is held constant, the MTI improvement factor decreases with increasing frequency. Lastly, for m = 3 and 4, the coherent performance is better than the noncoherent performance.

Effect of PRF

Next, consider the effect of PRF on the performance of the MTI filter. Calculate results for an L-band frequency at 1.5 GHz.

```
% Set parameters
m    = 2:4;                     % Number of pulses in (m-1) delay canceler
freq = 1.5e9;                   % Frequency (Hz)
prf  = linspace(100,1000,1000); % Pulse repetition frequency (Hz)

% Calculate MTI improvement factor versus PRF
for im = 1:numM
    ImCoherent(:,im) = mtifactor(m(im),freq,prf,'IsCoherent',true);
    ImNoncoherent(:,im) = mtifactor(m(im),freq,prf,'IsCoherent',false);
end
```

```
% Plot results
helperPlotMTI(m,prf,ImCoherent,ImNoncoherent,'Pulse Repetition Frequency (Hz)','PRF versus MTI Im
```
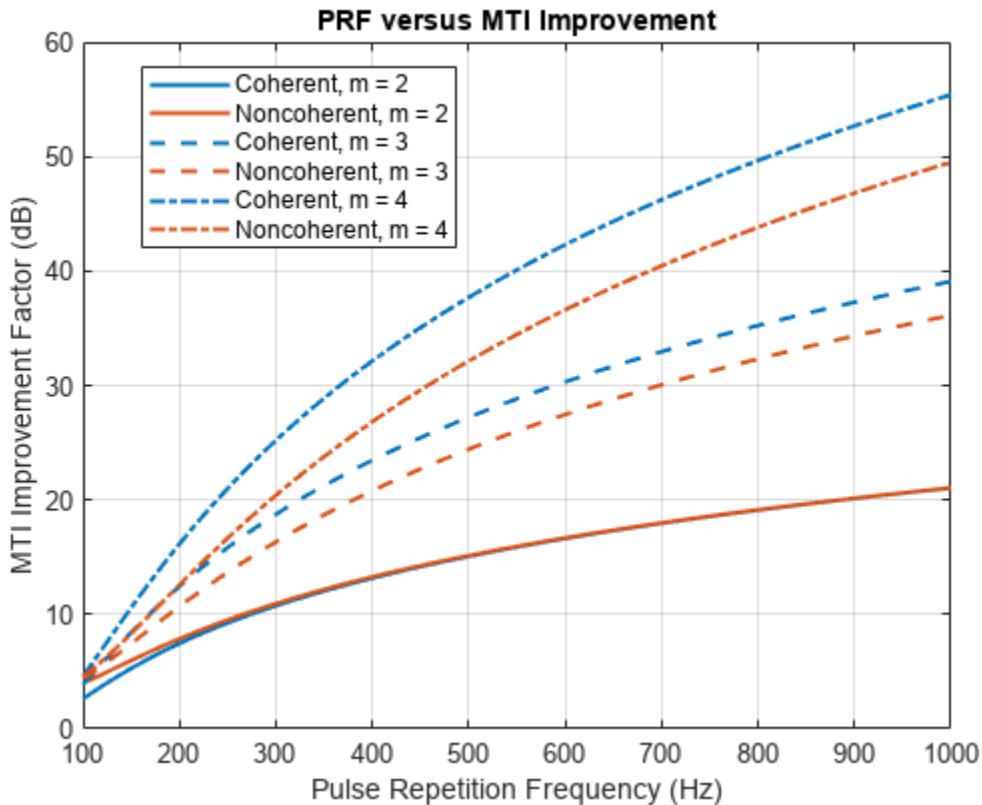

When frequency is held constant, there are several results to note. First, the difference between the coherent and noncoherent improvement factors increase for increasing frequency for the same m for the m = 3 and 4 cases. The results for the m = 2 case show that the improvement factor is very similar over the majority of PRFs investigated. Second, the MTI performance improves with increasing PRF. Lastly, for m = 3 and 4, the coherent performance is better than the noncoherent performance.

Combined Effects of Frequency and PRF

Next, consider the combined effects of frequency and PRF on the MTI improvement factor. This will allow a system analyst to get a better sense of the entire analysis space. Perform the calculation for a coherent MTI system using a 3-delay canceler.

```
% Set parameters
m    = 4;                          % Number of pulses in (m-1) delay canceler
freq = linspace(1e9,10e9,100); % Frequency (Hz)
prf  = linspace(100,1000,100); % Pulse repetition frequency (Hz)

% Initialize
numFreq           = numel(freq);
numPRF            = numel(prf);
ImCoherentMatrix = zeros(numPRF,numFreq);
```

```
% Calculate coherent MTI improvement factor over a range of PRFs and
% frequencies
for ip = 1:numPRF
    ImCoherentMatrix(ip,:) = mtifactor(m,freq,prf(ip),'IsCoherent',true);
end

% Plot results
helpPlotMTImatrix(m,freq,prf,ImCoherentMatrix);
```

Coherent MTI Improvement, m = 4
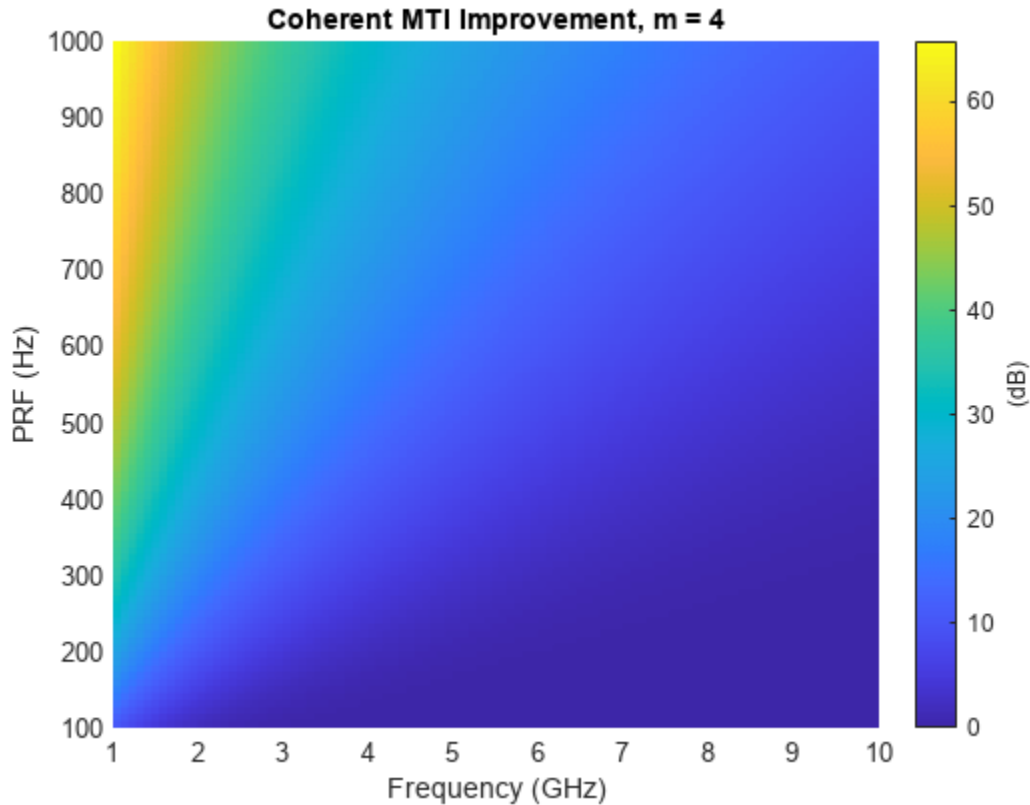
Note that the same behaviors as previously noted are shown here:

- MTI performance improves with increasing PRF
- MTI performance decreases with increasing frequency

MTI Performance Limitations

MTI processing is based on a requirement of target and clutter stationarity within a receive window. When successive returns are received and subtracted from one another, clutter is canceled. Any effect, whether internal or external to the radar, that inhibits stationarity within the receive window will result in imperfect cancellation.

A wide variety of effects can decrease the performance of the MTI cancellation. Examples include but are not limited to:

- Transmitter frequency instabilities
- Pulse repetition interval (PRI) jitter

- Pulse width jitter
- Quantization noise
- Uncompensated motion either in the radar platform or the clutter

The next two sections will discuss the effects of null velocity errors and clutter spectrum spread.

Null Velocity Errors

MTI performance declines when the clutter velocity is not centered on the null velocity. The effect of these null velocity errors results in a decreased MTI improvement factor since more clutter energy exists outside of the MTI filter null.

Consider the case of a radar operating in an environment with rain. Rain clutter has a non-zero average Doppler as the clutter approaches or recedes from the radar system. Unless the motion of the rain clutter is detected and compensated, the cancellation of the MTI filtering will be worse.

In this example, assume a null velocity centered at 0 Doppler. Investigate the effects on the improvement factor given clutter velocities over the range of -20 to 20 m/s for the coherent MTI processing case.

```matlab
% Setup parameters
m          = 2:4;                      % Number of pulses in (m-1) delay canceler
clutterVels = linspace(-20,20,100);    % MTI null velocity (m/s)
nullVel    = 0;                        % True clutter velocity (m/s)
freq       = 1.5e9;                    % Frequency (Hz)
prf        = 500;                      % Pulse repetition frequency (Hz)

% Initialize
numM          = numel(m);
numVels       = numel(clutterVels);
ImCoherent    = zeros(numVels,numM);
ImNoncoherent = nan(numVels,numM);

% Compute MTI improvement factor
for im = 1:numM
    for iv = 1:numVels
        ImCoherent(iv,im) = mtifactor(m(im),freq,prf,'IsCoherent',true,'ClutterVelocity',clutterV
    end
end

% Plot results
nullError = (clutterVels - nullVel).';
helperPlotMTI(m,nullError,ImCoherent,ImNoncoherent,'Null Error (m/s)','MTI Improvement with Null
```

MTI Improvement with Null Velocity Errors
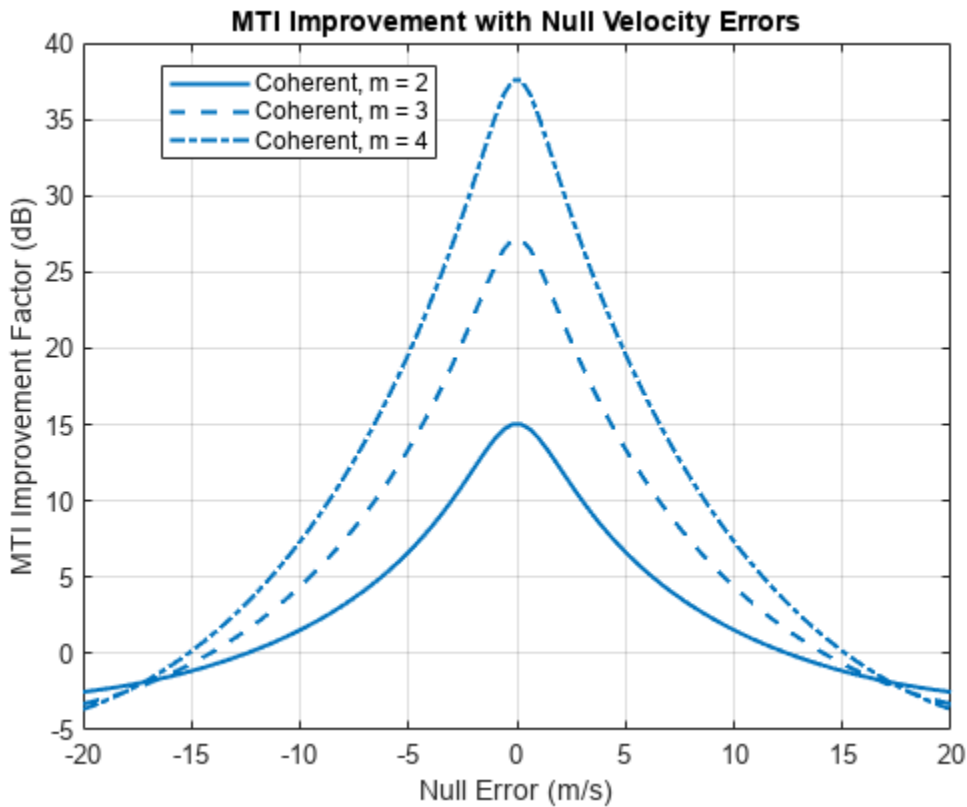
The coherent MTI experiences a rapid decrease in the improvement as the null error increases. The rate at which the improvement suffers increases with increasing number of pulses in the (m-1) delay canceler. In the case where m = 4, only a slight offset of 1.1 m/s results in a loss in the improvement factor of 3 dB.

Clutter Spread

A wider clutter spread results in more clutter energy outside of an MTI filter null and thus results in less clutter cancellation. While clutter spread is due in part to the inherent motion of the clutter scatterers, other sources of clutter spread can be due to:

- Phase jitter due to sampling
- Phase drift, which can be due to instability in coherent local oscillators
- Uncompensated radar platform motion

Consider the effect of the clutter spread on the MTI improvement factor.

```
% Setup parameters
m       = 2:4;                        % Number of pulses in (m-1) delay canceler
sigmav  = linspace(0.1,10,100);       % Standard deviation of clutter spread (m/s)
freq    = 1.5e9;                      % Frequency (Hz)
prf     = 500;                        % Pulse repetition frequency (Hz)

% Calculate MTI improvement
numSigma = numel(sigmav);
for im = 1:numM
```

```
        for is = 1:numSigma
            ImCoherent(is,im) = mtifactor(m(im),freq,prf,'IsCoherent',true,'ClutterStandardDeviation
            ImNoncoherent(is,im) = mtifactor(m(im),freq,prf,'IsCoherent',false,'ClutterStandardDeviat
        end
    end

    % Plot results
    helperPlotMTI(m,sigmav,ImCoherent,ImNoncoherent,'Standard Deviation of Clutter Spread (m/s)','MTI
```
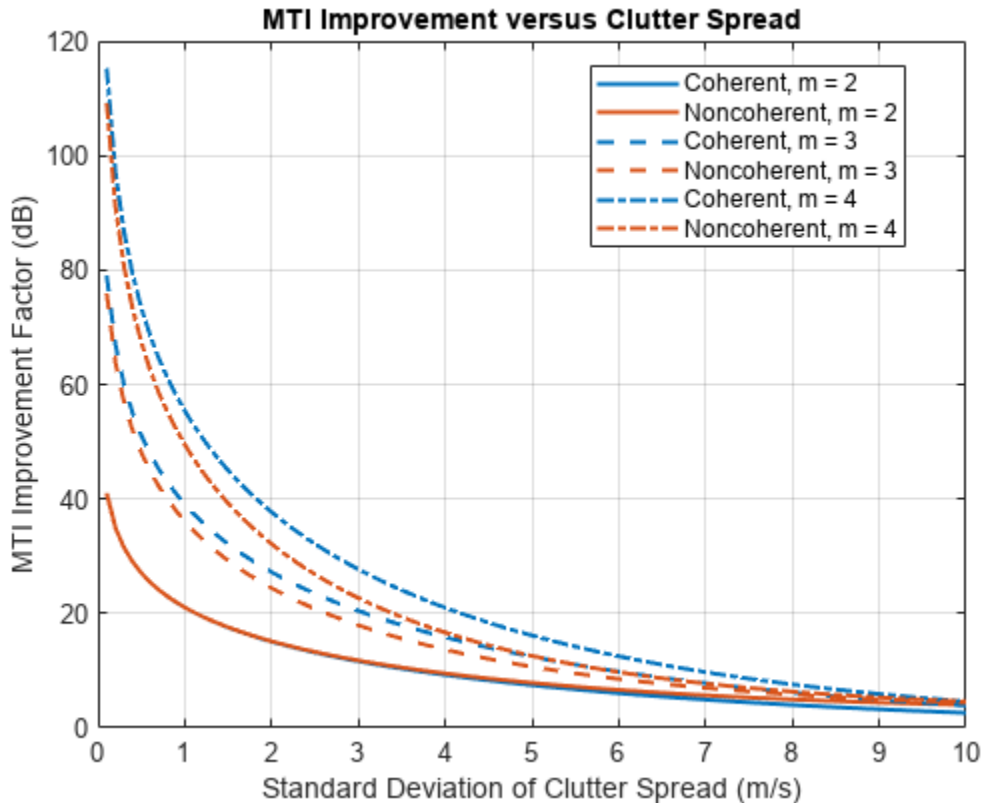

As can be seen from the figure, the standard deviation of the clutter spread is a great limiting factor of the MTI improvement regardless of whether the MTI is coherent or noncoherent. As the standard deviation of the clutter spread increases, the MTI improvement factor decreases significantly until the improvement falls below 5 dB for all values of m in both coherent and non-coherent cases.

Clutter Analysis for a Land-Based, MTI Radar

Consider a land-based MTI radar system. Calculate the clutter-to-noise ratio with and without MTI processing.

First, setup the radar and MTI processing parameters.

```
% Radar properties
freq    = 1.5e9;    % L-band frequency (Hz)
anht    = 15;       % Height (m)
ppow    = 100e3;    % Peak power (W)
tau     = 1.5e-6;   % Pulse width (sec)
prf     = 500;      % PRF (Hz)
```

```
Gtxrx   = 45;        % Tx/Rx gain (dB)
Ts      = 450;       % Noise figure (dB)
Ntx     = 20;        % Number of transmitted pulses

% MTI settings
nullVel = 0;         % Null velocity (m/s)
m       = 4;         % Number of pulses in the (m-1) delay canceler
N       = Ntx - m;   % Number of coherently integrated pulses
```

Consider a low-relief, wooded operating environment with a clutter spread of 1 m/s and a mean clutter velocity of 0 m/s. Calculate the land surface reflectivity for the grazing angles of the defined geometry. Use the `landroughness` and `landreflectivity` functions for the physical surface properties and reflectivity calculation, respectively. For this example, use the Nathanson land reflectivity model.

```
% Clutter properties
sigmav     = 1;      % Standard deviation of clutter spread (m/s)
clutterVel = 0;      % Clutter velocity (m/s)
landType   = 'Woods';

% Get the surface standard deviation of height (m), surface slope (deg),
% and vegetation type
[surfht,beta0,vegType] = landroughness(landType);

% Calculate maximum range for simulation
Rua = time2range(1/prf); % Maximum unambiguous range (m)
Rhoriz = horizonrange(anht,'SurfaceHeight',surfht); % Horizon range (m)
RhorizKm = Rhoriz.*1e-3; % Horizon range (km)
Rmax = min(Rua,Rhoriz); % Maximum range (m)

% Generate vector of ranges for simulation
Rm  = linspace(100,Rmax,1000); % Range (m)
Rkm = Rm*1e-3;                  % Range (km)

% Calculate land clutter reflectivity using Nathanson's land reflectivity
% model
grazAng = grazingang(anht,Rm,'TargetHeight',surfht);
nrcs = landreflectivity(landType,grazAng,freq,'Model','Nathanson');
nrcsdB = pow2db(nrcs);
```

Next, calculate and plot the radar cross section (RCS) of the clutter using the `clutterSurfaceRCS` function. The vertical line denotes the horizon.

```
% Calculate azimuth and elevation beamwidth
azbw = sqrt(32400/db2pow(Gtxrx));
elbw = azbw;

% Calculate clutter RCS
rcs = clutterSurfaceRCS(nrcs,Rm,azbw,elbw,grazAng(:),tau);

% Plot clutter RCS including horizon line
rcsdB = pow2db(rcs); % Convert to decibels for plotting
hAxes = helperPlot(Rkm,rcsdB,'RCS','Range (km)','Clutter RCS (dBsm)','Clutter Radar Cross Section
helperAddHorizLine(hAxes,RhorizKm);
```
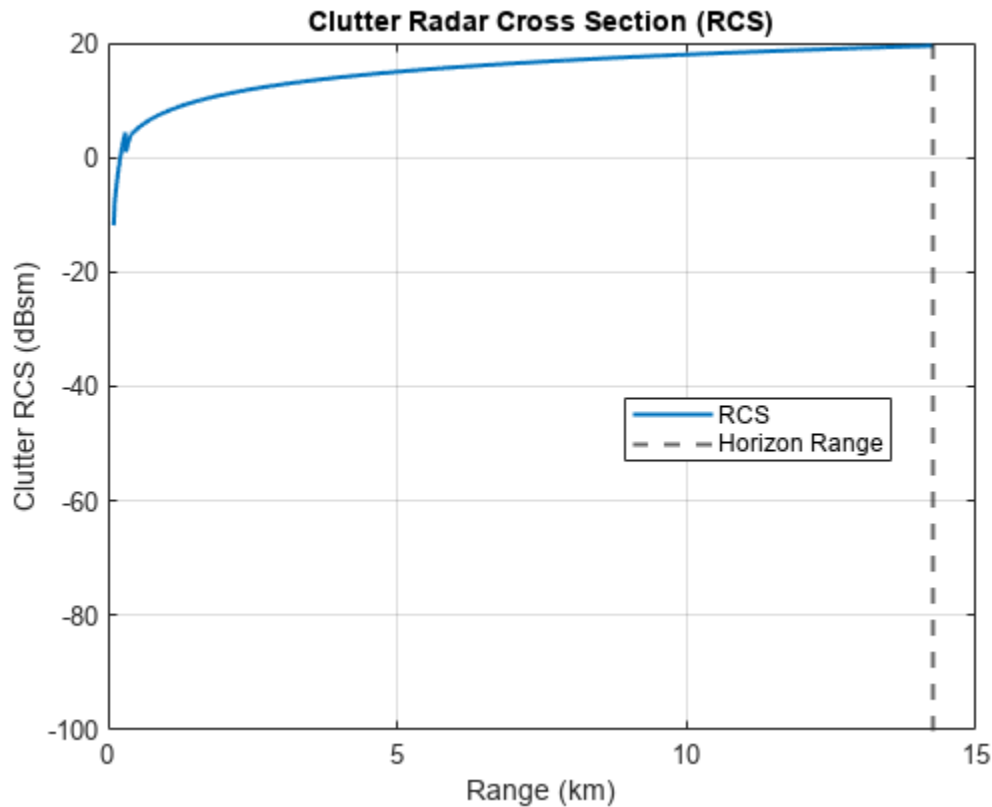
Since the propagation path deviates from free space, include the clutter propagation factor and atmospheric losses in the calculation.

The default permittivity calculation underlying the `radarpropfactor` function is a sea water model. In order to more accurately simulate the propagation path over land, calculate the permittivity for the vegetation using the `earthSurfacePermittivity` function.

```
% Calculate land surface permittivity for vegetation
temp = 20; % Ambient temperature (C)
wc = 0.5;  % Gravimetric water contnt
epsc = earthSurfacePermittivity('vegetation',freq,temp,wc);
```

Calculate the clutter propagation factor using the `radarpropfactor` function. Include the vegetation type in the calculation. At higher frequencies, the presence of vegetation can cause additional losses.

```
% Calculate clutter propagation factor
Fc = radarpropfactor(Rm,freq,anht,surfht, ...
    'SurfaceHeightStandardDeviation',surfht, ...
    'SurfaceSlope',beta0, ...
    'VegetationType',vegType, ...
    'SurfaceRelativePermittivity',epsc, ...
    'ElevationBeamwidth',elbw);
helperPlot(Rkm,Fc,'Clutter Propagation Factor','Range (km)', ...
    'Propagation Factor (dB)', ...
    'One-Way Clutter Propagation Factor F_C');
```

One-Way Clutter Propagation Factor F_C
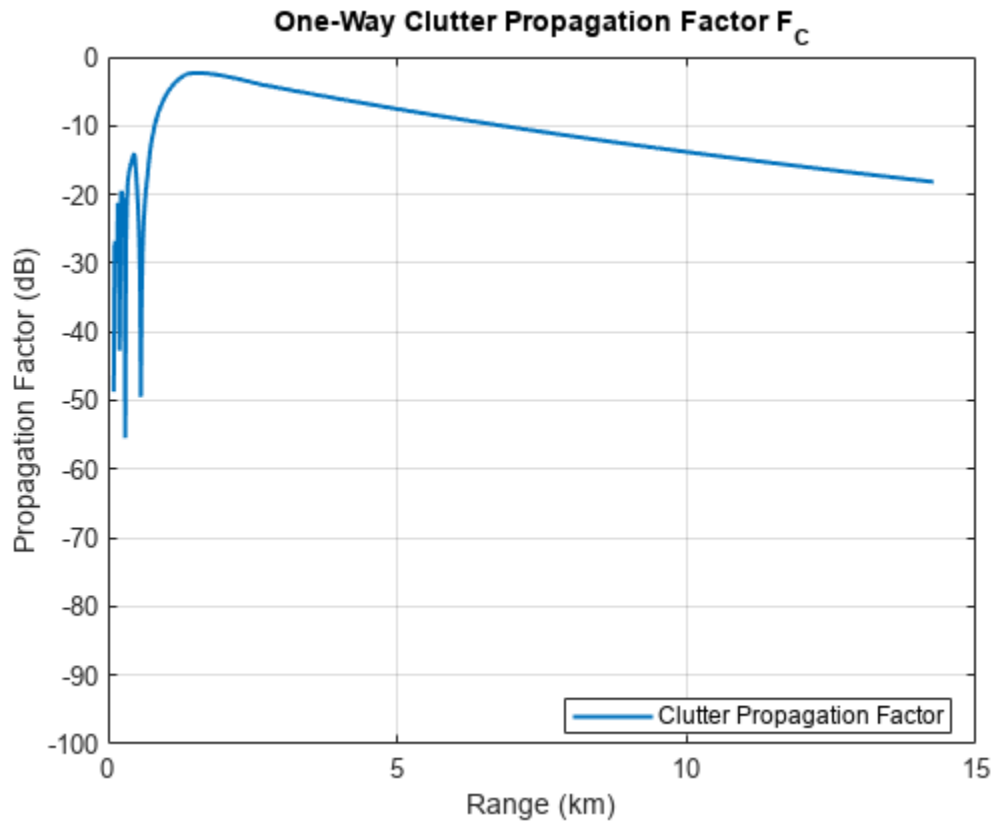
Next, calculate the atmospheric losses in this simulation. Assume the default standard atmosphere. Perform the calculation using the `tropopl` function.

```
% Calculate the atmospheric loss due to water and oxygen attenuation
elAng = height2el(surfht,anht,Rm); % Elevation angle (deg)
numEl = numel(elAng);
Latmos = zeros(numEl,1);
Llens = zeros(numEl,1);
for ie = 1:numEl
    Latmos(ie,:) = tropopl(Rm(ie),freq,anht,elAng(ie));
end
hAxes = helperPlot(Rkm,Latmos,'Atmospheric Loss','Range (km)','Loss (dB)','One-Way Atmospheric L
ylim(hAxes,[0 0.1]);
```
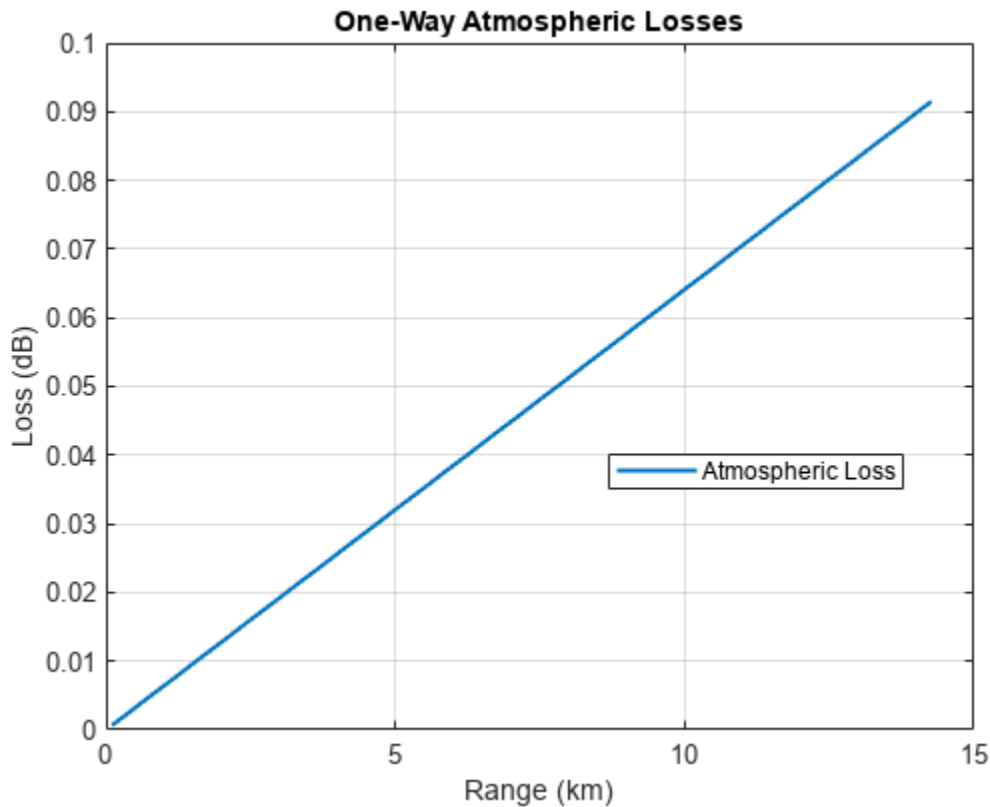
Calculate the CNR using the `radareqsnr` function and plot results with and without MTI. Again, note the drop in CNR as the simulation range approaches the radar horizon.

```matlab
% Calculate CNR
lambda = freq2wavelen(freq);
cnr = radareqsnr(lambda,Rm(:),ppow,tau, ...
        'gain',Gtxrx,'rcs',rcs,'Ts',Ts, ...
        'PropagationFactor',Fc, ...
        'AtmosphericLoss',Latmos);
coherentGain = pow2db(N);
cnr = cnr + coherentGain;
hAxes = helperPlot(Rkm,cnr,'CNR','Range (km)','CNR (dB)','CNR Clutter-to-Noise Ratio');
helperAddHorizLine(hAxes,RhorizKm);

% Calculate CNR with MTI
Im = mtifactor(m,freq,prf,'IsCoherent',true,...
    'ClutterVelocity',clutterVel, ...
    'ClutterStandardDeviation',sigmav, ...
    'NullVelocity',nullVel)
```

```matlab
Im = 55.3986
```

```matlab
cnrMTI = cnr - Im;
helperAddPlot(Rkm,cnrMTI,'CNR + MTI',hAxes);
```

Lastly, calculate the MTI improvement factor assuming a null error exists due to the true clutter velocity being 3 m/s while the null velocity remains centered at 0 m/s.

```
% Calculate CNR with null velocity error
trueClutterVel = 3;                    % Clutter velocity (m/s)
nullError = trueClutterVel - nullVel; % Null error (m/s)
ImNullError = mtifactor(m,freq,prf,'IsCoherent',true,...
    'ClutterVelocity',trueClutterVel, ...
    'ClutterStandardDeviation',sigmav, ...
    'NullVelocity',nullVel)
```

```
ImNullError = 33.6499
```

```
cnrMTINullError = cnr - ImNullError;
helperAddPlot(Rkm,cnrMTINullError, ...
    sprintf('CNR + MTI with %.1f (m/s) Null Error',nullError), ...
    hAxes);
```
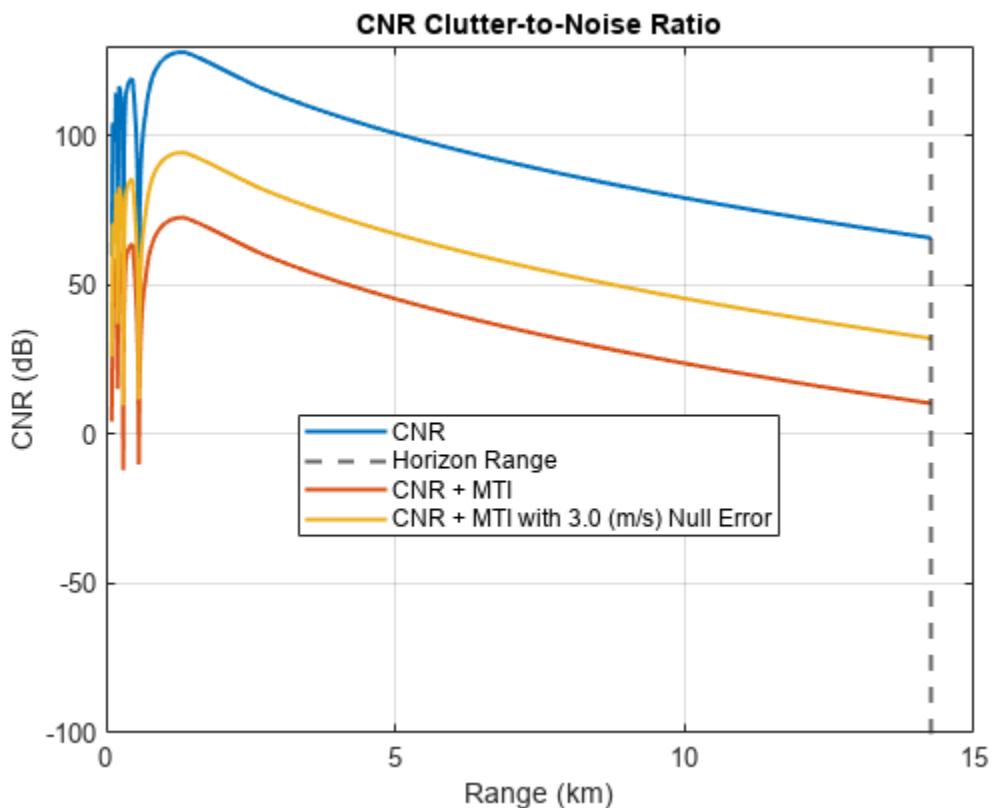


```
ImLoss = Im - ImNullError
```

```
ImLoss = 21.7488
```

Note the dramatic decrease in the CNR due to the MTI processing. When the null velocity is set to the clutter velocity, the improvement is 55 dB. When there is an uncompensated motion, the cancellation decreases to 34 dB. This is a loss of about 22 dB of cancellation. This shows the need to properly compensate for motion or to steer the null to the appropriate velocity.

Summary

This example discusses the MTI improvement factor and investigates a multitude of effects on MTI performance. Using the `mtifactor` function, we see that MTI performance:

- Improves with increasing PRF
- Decreases with increasing frequency
- Improves with increasing the number of pulses in the (m - 1)-delay canceler

Additionally, we see that the performance of coherent MTI is generally better than noncoherent MTI.

Lastly, we investigate limitations of MTI performance within the context of a land-based MTI radar system, showing the need to properly compensate for unexpected clutter velocities.

References

1   Barton, David Knox. *Radar Equations for Modern Radar*. Artech House Radar Series. Boston, Mass: Artech House, 2013.

2   Richards, M. A., Jim Scheer, William A. Holm, and William L. Melvin, eds. *Principles of Modern Radar*. Raleigh, NC: SciTech Pub, 2010.

```matlab
function helperPlotLogMTI(m,freq,ImCoherent,ImNonCoherent)
% Used for the MTI plots that have a logarithmic x axis

hFig = figure;
hAxes = axes(hFig);
lineStyles = {'-','--','-.'};
numM = numel(m);
for im = 1:numM
    semilogx(hAxes,freq.*1e-9,ImCoherent(:,im),'LineWidth',1.5, ...
        'LineStyle',lineStyles{im}, 'Color',[0 0.4470 0.7410], ...
        'DisplayName',sprintf('Coherent, m = %d',m(im)))
    hold(hAxes,'on')
    semilogx(hAxes,freq.*1e-9,ImNonCoherent(:,im),'LineWidth',1.5, ...
        'LineStyle',lineStyles{im}, 'Color',[0.8500 0.3250 0.0980], ...
        'DisplayName',sprintf('Noncoherent, m = %d',m(im)))
end
grid(hAxes,'on');
xlabel(hAxes,'Frequency (GHz)')
ylabel(hAxes,'MTI Improvement Factor (dB)')
title('Frequency versus MTI Improvement')
legend(hAxes,'Location','Best')
end

function helperPlotMTI(m,x,ImCoherent,ImNonCoherent,xLabelStr,titleName)
% Used for the MTI plots that have an x-axis in linear units

hFig = figure;
hAxes = axes(hFig);
lineStyles = {'-','--','-.'};
numM = numel(m);
for im = 1:numM
    plot(hAxes,x,ImCoherent(:,im),'LineWidth',1.5, ...
        'LineStyle',lineStyles{im}, 'Color',[0 0.4470 0.7410], ...
        'DisplayName',sprintf('Coherent, m = %d',m(im)))
    hold(hAxes,'on')
    if any(~isnan(ImNonCoherent)) % Don't plot if NaN
        plot(hAxes,x,ImNonCoherent(:,im),'LineWidth',1.5, ...
            'LineStyle',lineStyles{im}, 'Color',[0.8500 0.3250 0.0980], ...
            'DisplayName',sprintf('Noncoherent, m = %d',m(im)))
    end
```

```
end
grid(hAxes,'on');
xlabel(hAxes,xLabelStr)
ylabel(hAxes,'MTI Improvement Factor (dB)')
title(titleName)
legend(hAxes,'Location','Best')
end

function helpPlotMTImatrix(m,freq,prf,ImMat)
% Creates image of MTI improvement factor with Frequency on the x-axis and
% PRF on the y-axis

hFig = figure;
hAxes = axes(hFig);
hP = pcolor(hAxes,freq.*1e-9,prf,ImMat);
hP.EdgeColor = 'none';
xlabel(hAxes,'Frequency (GHz)')
ylabel(hAxes,'PRF (Hz)')
title(sprintf('Coherent MTI Improvement, m = %d',m))
hC = colorbar;
hC.Label.String = '(dB)';
end

function varargout = helperPlot(x,y,displayName,xlabelStr,ylabelStr,titleName)
% Used for CNR analysis

% Create new figure
hFig = figure;
hAxes = axes(hFig);

% Plot
plot(hAxes,x,y,'LineWidth',1.5,'DisplayName',displayName);
grid(hAxes,'on');
hold(hAxes,'on');
xlabel(hAxes,xlabelStr)
ylabel(hAxes,ylabelStr);
title(hAxes,titleName);
ylims = get(hAxes,'Ylim');
set(hAxes,'Ylim',[-100 ylims(2)]);

% Add legend
legend(hAxes,'Location','Best')

% Output axes
if nargout ~= 0
    varargout{1} = hAxes;
end
end

function helperAddPlot(x,y,displayName,hAxes)
% Add additional CNR plots

plot(hAxes,x,y,'LineWidth',1.5,'DisplayName',displayName);
end

function helperAddHorizLine(hAxes,val)
% Add vertical line indicating horizon range
```

```
xline(hAxes,val,'--','DisplayName','Horizon Range','LineWidth',1.5);
end
```

Radar Link Budget Analysis

This example shows how to use the **Radar Designer** app to perform radar link budget analysis and design a radar system based on a set of performance requirements. **Radar Designer** allows a user to design a new radar system starting from one of five preset radar types, set the performance requirements, compute radar metrics, configure the environment, and compare several alternative designs. You can also export the design as a MATLAB® script for further analysis.

Introduction

The radar range equation is a powerful tool that ties together the main parameters of a radar system. It can give a radar engineer a good idea about performance of the system without resorting to complex analysis and simulations. The radar equation is especially useful in early stages of the design when specific information about various components (for example, transmitted waveform, size or shape of the antenna array, signal processing algorithms, and so on) might not yet be available. Although the radar equation provides only approximate results, the fidelity of the analysis can be significantly improved by considering losses introduced by the components of the radar system and the signal propagation medium. The **Radar Designer** app is a tool for performing the radar equation analysis also known as the radar link budget analysis. It provides a user with many tunable parameters for the radar system, the target, and the environment, and offers a set of visualizations to aid with a choice of these parameters. The **Radar Designer** app also allows to design a radar system based on a set of performance requirements.

This example shows how to use the **Radar Designer** app to design an X-band surveillance radar for detecting small targets. The design is based on the following requirements specification:

*   The peak transmit power should not exceed 2000 W
*   The radar should provide 360-degree coverage in azimuth and 60 degrees coverage in elevation
*   The radar should detect small manned aircraft with the radar cross section of 1 m^2 at ranges from 300 m to 18 km
*   The radar should detect small unmanned aircraft (UAS) with the radar cross section of 0.03 m^2 at ranges from 300 m to 8 km
*   The probability of detection and false alarm should be 0.9 and 1e-6, respectively
*   The radar should resolve two targets with the same azimuth and elevation separated in range by 30 m
*   The radar should have range, azimuth, and elevation accuracies of 2 m, 0.2 deg, and 0.5 deg, respectively
*   The radar should detect targets with velocities up to 180 km/h
*   The radar should maintain its accuracy and detection performance in heavy rain (16 mm/hr) conditions

New Session

The **Radar Designer** app can be launched by using the command:

```
radarDesigner
```

By default, the application allows a user to either start a new or open an existing session using the corresponding buttons in the toolstrip. A new session offers a choice of one of the five predefined

radar types: airborne, airport, automotive, tracking, and weather. In this example we use the default airport radar as a starting point.
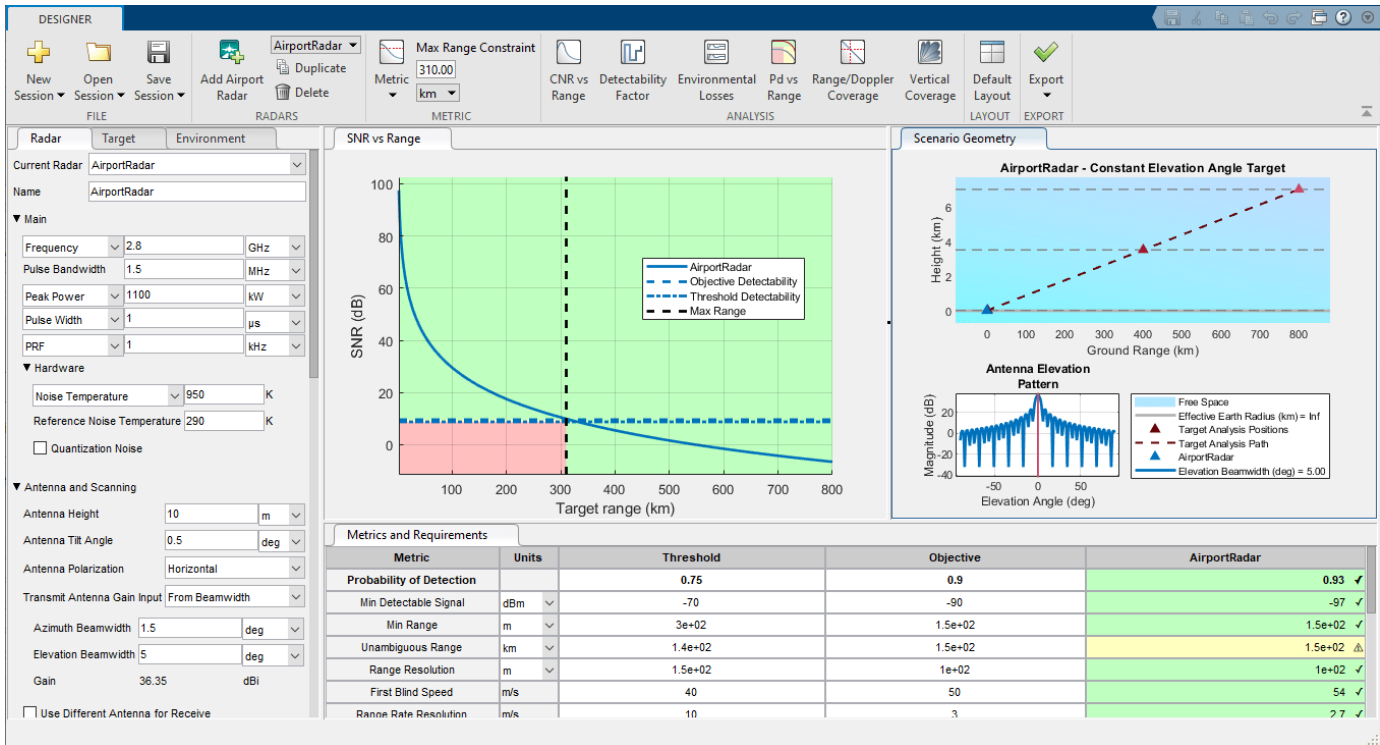
Once a new session is loaded, the **Radar Designer** app presents a user the following document groups:

- Radar, Target, and Environment panels on the left
- SNR vs Range and Scenario Geometry plots in the center and right
- Metrics and Requirements table at the bottom

As a first step after opening a new session, we change the name of the current design at the top of the **Radar** panel to `NewDesign`.

Metrics and Requirements

In this example we have a specification detailing the performance of the final system. However, it does not specify all the design parameters needed to achieve the required levels of performance. Most of the design parameters must be derived from the requirements given in the specification. The **Radar Designer** app provides a way to derive the radar design parameters from the performance requirements.

The performance metrics and the corresponding requirements are housed in the `Metrics and Requirements` table. For each performance metric the app has two requirement values:

- `Threshold` — Describes the minimum performance level for the metric
- `Objective` — Defines the value of the metric that will enable the new system to fully satisfy mission needs

The values between `Threshold` and `Objective` constitute the trade-space that can be used by a radar engineer to balance multiple, sometimes conflicting, performance requirements.

The computed metrics, which are shown in the **Metrics and Requirements** table, can be constrained by either the maximum range or the probability of detection. A choice of which variable to use as a constraint is made through the **Metric** button in the **Metric** section of the toolstrip. Selecting maximum range as a constraint means that the performance metrics displayed in the **Metrics and Requirements** table are computed at the specified maximum range. Selecting the probability of detection as a constraint means that the displayed metrics are computed assuming the specified value of the probability of detection. In this example we are interested in two ranges: 1) 18 km for manned aircraft, and 2) 8 km for UAS. We start with the manned aircraft and set the `Metric` in the toolstrip to the maximum range constraint of 18 km.
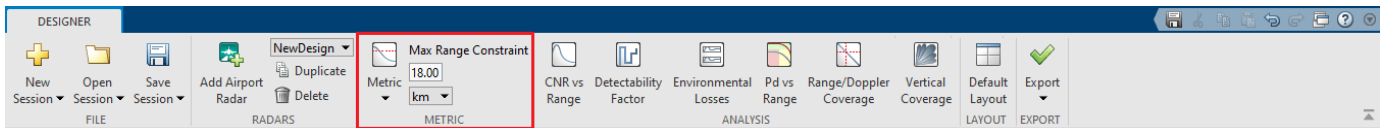
As the next step, we populate the `Threshold` and `Objective` values of the **Metric and Requirements** table with the numbers given in the specification. In this example the specification provides only a single value for each performance metric. We use this value to set the `Objective` requirement. We then set the corresponding `Threshold` to a reasonable value close to the `Objective`. Although the desired performance of the system is defined by the `Objective` requirement, the system is considered to have an acceptable performance if the `Threshold` requirement is met. This flexibility is needed to create the trade-space for selecting the design parameters, which otherwise could be difficult or impossible to choose. Since the specification does not provide requirements for all the metrics shown in the table, we leave the requirements for these metrics set to the default values.

Metrics and Requirements

Metric	Units	Threshold	Objective
Probability of Detection		0.75	0.9
Min Detectable Signal	dBm	-70	-90
Min Range	m	5e+02	3e+02
Unambiguous Range	km	8	18
Range Resolution	m	50	30
First Blind Speed	m/s	80	1e+02
Range Rate Resolution	m/s	10	3
Range Accuracy	m	5	2
Azimuth Accuracy	deg	0.4	0.2
Elevation Accuracy	deg	1	0.5
Range Rate Accuracy	m/s	3	1
Probability of True Track		0.95	0.99
Probability of False Track		1e-08	1e-12
Effective Isotropic Radiated Power	MW	1e+03	2.5e+03
Power-Aperture Product	W·m²	1e+02	2e+02

Target Parameters

The target parameters are set in the **Target** panel. Since we are considering the small manned aircraft first, we set the radar cross section of the target to 1 m². The Swerling Model is changed to `Swerling 1` to model more realistic fluctuating targets.

Radar Parameters

After the requirements and the target parameters have been set, we can start adjusting the radar design parameters such that the computed metrics meet the stated requirements. The Radar Designer app provides a convenient way to monitor the status of the computed metrics while changing the value of the design parameters. The entries of the **Metrics and Requirements** table are color coded to indicate the status of the computed metrics. Metrics that meet the corresponding `Objective` requirement are colored in green, metrics with the values between the `Threshold` and `Objective` are colored in yellow, and the metrics that do not meet the `Threshold` requirement are colored in red. The same colors are also used in the SNR vs Range and Pd vs Range plot to show the ranges at which the detection requirements are satisfied.

To guarantee that the 1 m² RCS target is detected at the desired range of 18 km, we adjust the radar design parameters to make sure that the SNR curve in the SNR vs Range plot is above the `Objective Detectability` line at `Max Range`.

The radar design parameters are divided into four sections. Each section is adjusted as follows:

- **Main**. The radar operating frequency and the peak power are set to the values given in the specification. The pulse bandwidth is adjusted to meet the range resolution requirements and the pulse width is set to achieve high enough available SNR at the maximum range. The 7 kHz PRF value is selected to find a trade-off between the 21.4 km unambiguous range and the maximum unambiguous velocity of 198 km/h (the first blind speed of 396 km/h). The Range/Doppler Coverage plot, accessed through the **Range/Doppler Coverage** button in the **Analysis** section of the toolstrip, is used to visualize the trade-space between the range and velocity of the target.
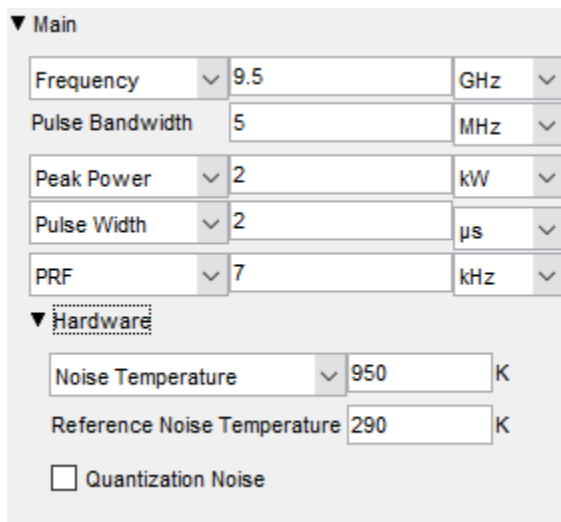
- Antenna and Scanning. Antenna height, tilt, and polarization are left unchanged. The azimuth and elevation antenna beamwidth are set to 2 and 6 degrees, respectively, to meet the azimuth and elevation accuracy requirements. The **Scan Mode** is set to Mechanical to facilitate 360-degree coverage in azimuth, and the scan sector elevation dimension is set to 60 degrees as given in this specification. The Antenna and Scanning section also shows that the size of the search volume is 5.441 steradians and the time it takes to scan this volume is 6.38 seconds. Including scanning in the analysis adds the beam shape loss and the beam-dwell factor to the link budget.

▼ Antenna and Scanning

Antenna Height	10	m
Antenna Tilt Angle	0.5	deg
Antenna Polarization	Horizontal	
Transmit Antenna Gain Input	From Beamwidth	
Azimuth Beamwidth	2	deg
Elevation Beamwidth	6	deg
Gain	34.31	dBi

☐ Use Different Antenna for Receive

Scan Mode	Mechanical	
Azimuth Scan Sector Size	360	deg
Elevation Scan Limits	0 60	deg
Max Scan Rate	466.7	deg/s
Search Volume Size	5.441	sr
Search Time	6.38	s

- Detection and Tracking. The probability of false alarm is set to the required value of 1e-6. The number of coherently integrated pulses is selected such that the Objective Detectability value that determines the SNR required to detect a Swerling 1 case target with the desired probabilities of detection and false alarm of 0.9 and 1e-6 respectively, is below the available SNR at the specified maximum range constraint of 18 km.

- **Loss Factors**. To account for losses due to pulse eclipsing we add the statistical eclipsing loss to the link budget analysis.

After these adjustments, the **Metric and Requirements** table shows that this design satisfies the specification for small manned aircraft with RCS of 1 m² or larger. From the SNR vs Range plot we can see that the detectability factor required to achieve the `Objective` detection probability of 0.9 is approximately 10 dB, while the detectability factor for the `Threshold` requirement of 0.75 is close to 5 dB. Since the available SNR curve is above the `Objective Detectability` line at 18 km, the resulting probability of detection is higher than the required `Objective` value and equals to 0.92. The **Metrics and Requirements** table also displays that the minimum detectable signal required to detect a 1 m² target with this probability of detection is –92 dBm.

Metric	Units		Threshold	Objective	NewDesign
Probability of Detection			0.75	0.9	0.92 ✓
Min Detectable Signal	dBm	∨	-70	-90	-92 ✓
Min Range	m	∨	5e+02	3e+02	3e+02 ✓
Unambiguous Range	km	∨	8	18	21 ✓
Range Resolution	m	∨	50	30	30 ✓
First Blind Speed	m/s		80	1e+02	1.1e+02 ✓
Range Rate Resolution	m/s		10	3	3.7 ⚠
Range Accuracy	m	∨	5	2	1.6 ✓
Azimuth Accuracy	deg	∨	0.4	0.2	0.11 ✓
Elevation Accuracy	deg	∨	1	0.5	0.32 ✓
Range Rate Accuracy	m/s		3	1	0.19 ✓
Probability of True Track			0.95	0.99	0.99 ✓
Probability of False Track			1e-08	1e-12	3e-14 ✓
Effective Isotropic Radiated Power	MW	∨	1e+03	2.5e+03	5.4 ✗
Power-Aperture Product	W·m²	∨	1e+02	2e+02	4.3e+02 ✓

Environment Parameters

Until now this example assumed free space propagation without any atmospheric attenuation. To make the analysis more accurate, losses due to propagation and atmospheric attenuation can be included and parametrized through the `Environment` panel.

The specification states that the radar under design must maintain the required detection performance and the measurement accuracy in heavy rain (16 mm/hr). To include the path loss due to precipitation in the analysis, we set `Precipitation Type` to `Rain` in the `Precipitation` section of the `Environment` panel. We then choose ITU model and set the precipitation ranges such that 16 mm/hr rain is present in all ranges of interest. Now the `Metrics and Requirements` table and the SNR vs Range plot show that the probability of detection at the maximum range is much lower than the required 0.9.
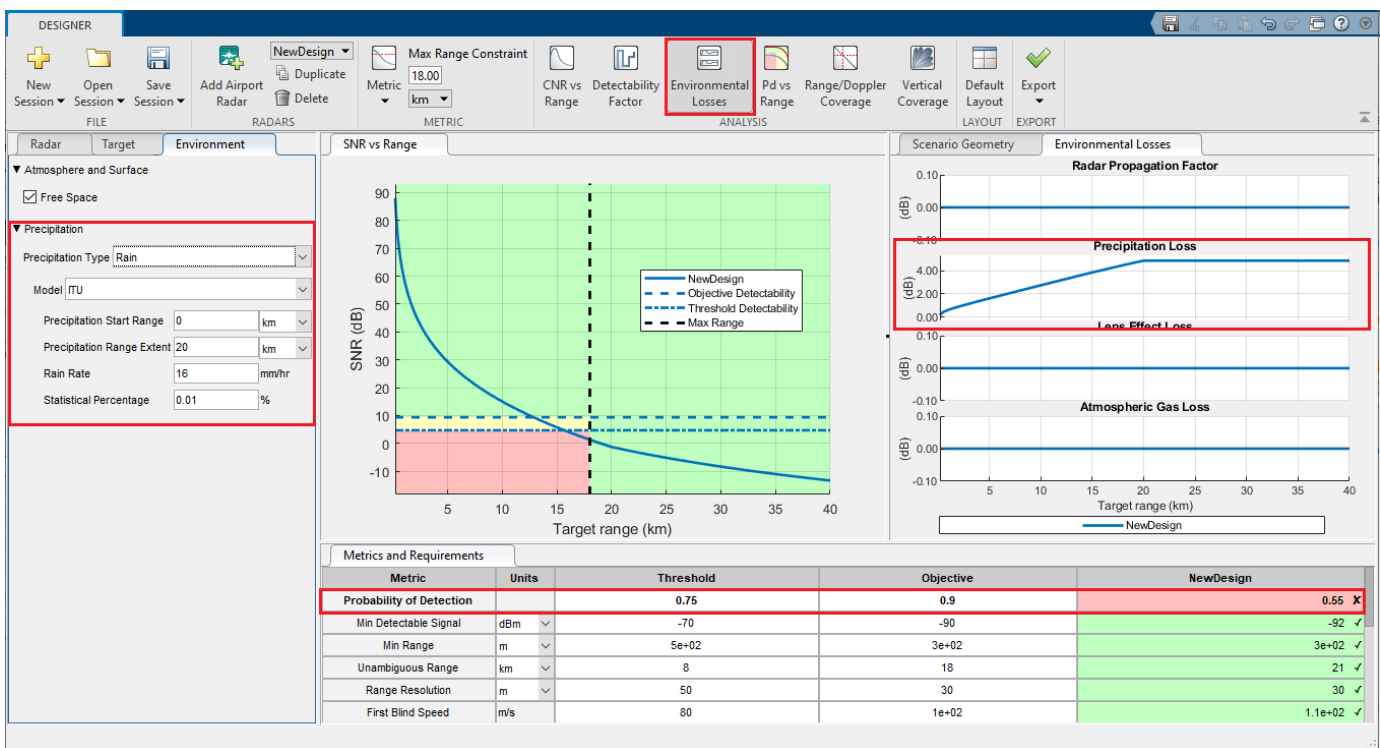
The `Environmental Losses` plot gives a better idea about the contribution of the precipitation loss to the overall loss budget. It is accessed through the **Environmental Losses** button in the Analysis section. This plot visualizes four types of range-dependent losses due to propagation and atmospheric attenuation. The `Precipitation Loss` subplot shows that 16 mm/hr rain creates an additional loss of 4.8 dB at 18 km. This results in the probability of detection dropping from 0.92 to 0.55, which is below the `Threshold` requirement. Thus, the performance of the system becomes unacceptable in heavy rain conditions and does not meet the specification.

M-of-N CPI Integration

The probability of detection can be improved by either increasing the SNR available at the receiver or by decreasing the SNR required to make a detection (the detectability factor). The latter approach might be more attractive in practice since decreasing the detectability factor can be accomplished through application of signal processing techniques that do not require making changes in the hardware. The detectability factor can be decreased by integrating more pulses. However, target RCS

fluctuation usually imposes a limit on the amount of pulses that can be coherently integrated. A possible solution to integrating more pulses while addressing the problem of the RCS fluctuation is the M-of-N integration over multiple coherent processing intervals (CPIs). Within each CPI the pulses are integrated coherently, and then the M-of-N integration is applied across CPIs. Navigating to the `Radar` panel and setting the number of CPIs in the `Detection and Tracking` section to 3 and the number of CPIs with a detection to 2 increases the resulting probability of detection from 0.55 to 0.81.
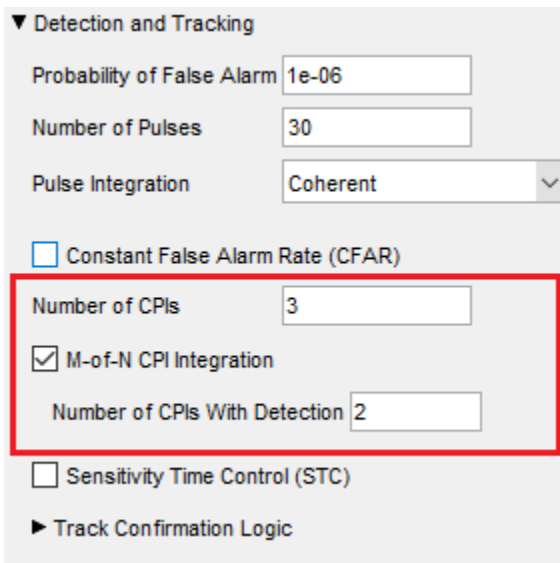
Although the probability of detection is still below the specified `Objective` value, it meets the `Threshold` requirement. This means that the system has an acceptable detection performance in heavy rain. Similarly, the range, azimuth, and elevation accuracies clear the `Threshold` requirement but are below their respective `Objective` values.

Metric	Units		Threshold	Objective	NewDesign
Probability of Detection			0.75	0.9	0.81 ⚠
Min Detectable Signal	dBm	⌄	-70	-90	-99 ✓
Min Range	m	⌄	5e+02	3e+02	3e+02 ✓
Unambiguous Range	km	⌄	8	18	21 ✓
Range Resolution	m	⌄	50	30	30 ✓
First Blind Speed	m/s	⌄	80	1e+02	1.1e+02 ✓
Range Rate Resolution	m/s	⌄	10	3	3.7 ⚠
Range Accuracy	m	⌄	5	2	3.7 ⚠
Azimuth Accuracy	deg	⌄	0.4	0.2	0.3 ⚠
Elevation Accuracy	deg	⌄	1	0.5	0.91 ⚠
Range Rate Accuracy	m/s		3	1	0.45 ✓
Probability of True Track			0.95	0.99	0.96 ⚠
Probability of False Track			1e-08	1e-12	1.8e-13 ✓
Effective Isotropic Radiated Power	MW	⌄	1e+03	2.5e+03	5.4 ✗
Power-Aperture Product	W·m²	⌄	1e+02	2e+02	4.3e+02 ✓

Open this design in **Radar Designer**.

```
radarDesigner('SurveillanceRadarSmallTargets.mat')
```

Small UAS

To verify whether this radar design will have a satisfactory performance when the target is a small UAS, we change the target RCS to 0.03 m² and set the metric constraint to maximum range of 8 km. The SNR vs Range plot shows that the available SNR for this design is above the `Objective Detectability` line at 8 km, and the resulting probability of detection at that range is 0.94, which is well above the required value. The system is able to satisfy the Objective requirement for the probability of detection at 8 km because the impact of the atmospheric attenuation is smaller at closer ranges.
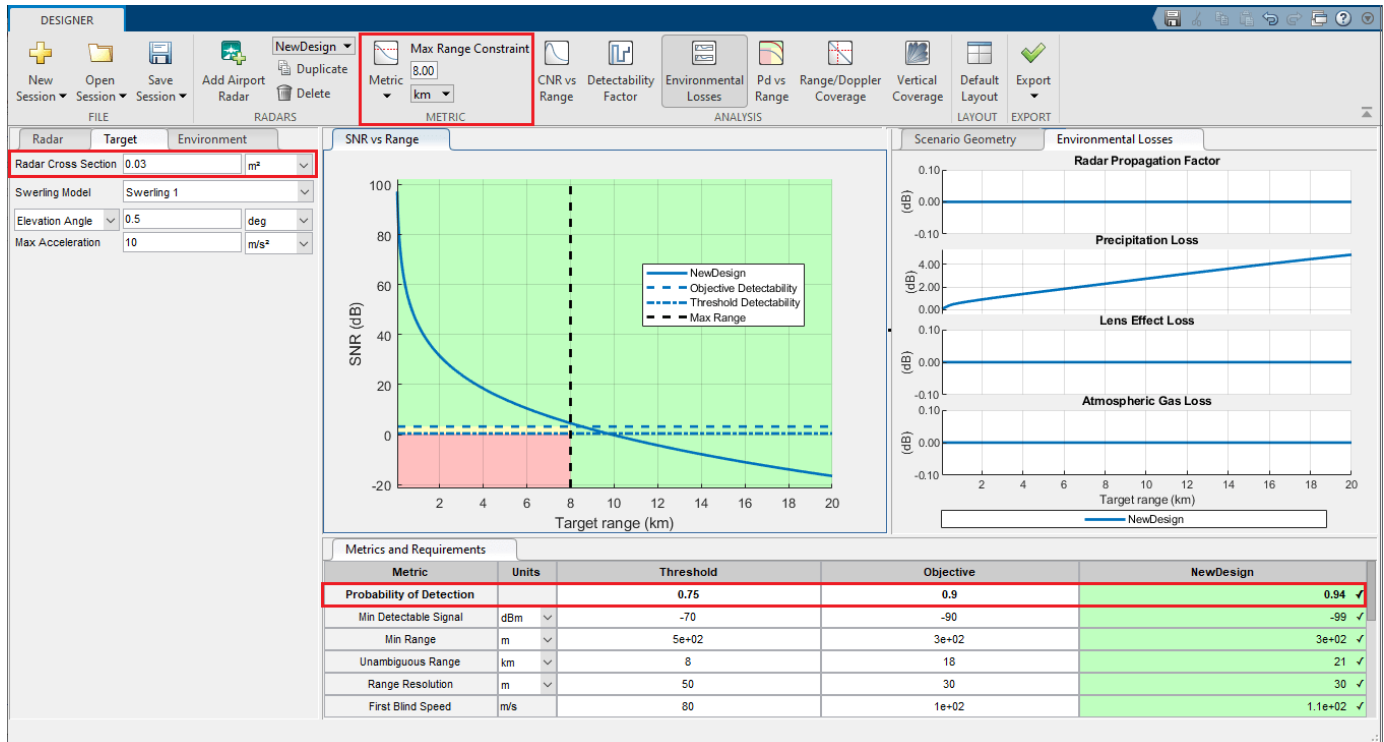
The resulting range and the elevation accuracies, however, are still below the `Objective` and above the `Threshold` requirements.

.

Export

The **Radar Designer** allows exporting the created design as a MATLAB script by clicking the **Export** button in the **Export** section of the toolstrip and selecting **Export SNR vs Range MATLAB Script**. The exported script contains the selected radar, target, and environment parameters, and reproduces the SNR vs Range plot. It can be used to further experiment, enhance, and modify the design. In addition, the results shown in the **Metrics and Requirements** table can also be exported as a separate MATLAB script by clicking **Export** and then choosing **Generate Metrics Report.** When executed, this script outputs a formatted report for the computed metrics.
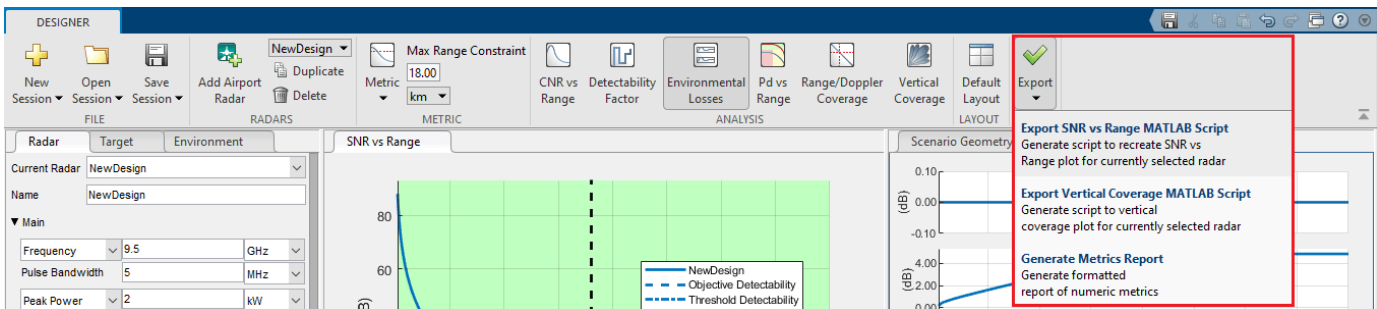
Summary

This example shows how to use the **Radar Designer** app to perform link budget analysis of a surveillance radar system for detecting small targets. The example starts with a specification and a set of performance requirements. It shows how to set the `Objective` and `Threshold` requirements based on the values provided in the specification. Then it shows how to adjust the radar design parameters with the help of the SNR vs Range plot and the stoplight color coding such that the design meets the stated requirements. The example also shows how to change the target parameters to model manned and unmanned aircraft, and how to configure the environment settings to include atmospheric loss due to precipitation into the analysis.

Planning Radar Network Coverage over Terrain

This example shows how to plan a radar network using propagation modelling over terrain. DTED level-1 terrain data is imported for a region that contains five candidate monostatic radar sites. The radar equation is used to determine whether target locations can be detected, where additional path loss is calculated using either the Longley-Rice propagation model or Terrain Integrated Rough Earth Model™ (TIREM™). The best three sites are selected for detection of a target that is flying at 500 meters above ground level. The scenario is updated to model a target that is flying at 250 meters above ground level. Radar coverage maps are shown for both scenarios.

Import Terrain Data

Import DTED-format terrain data for a region around Boulder, Colorado, US. The terrain file was downloaded from the SRTM Void Filled data set available from the United States Geological Survey (USGS). The file is DTED level-1 format and has a sampling resolution of about 90 meters. A single DTED file defines a region that spans 1 degree in both latitude and longitude.

```
dtedfile = "n39_w106_3arc_v2.dt1";
attribution = "SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey.";
addCustomTerrain("southboulder",dtedfile,"Attribution",attribution)
```

Open Site Viewer using the imported terrain. Visualization with high-resolution satellite map imagery requires an Internet connection.

```
viewer = siteviewer("Terrain","southboulder");
```

Show Candidate Radar Sites

The region contains mountains to the west and flatter areas to the east. Radars will be placed in the flat area to detect targets over the mountainous region. Define five candidate locations for placing the radars and show them on the map. The candidate locations are chosen to correspond to local high points on the map outside of residential areas.

Create collocated transmitter and receiver sites at each location to model monostatic radars, where the radar antennas are assumed to be 10 meters above ground level.

```
names = "Radar site" + (1:5);
rdrlats = [39.6055 39.6481 39.7015 39.7469 39.8856];
rdrlons = [-105.1602 -105.1378 -105.1772 -105.2000 -105.2181];

% Create transmitter sites associated with radars
rdrtxs = txsite("Name",names, ...
    "AntennaHeight",10, ...
    "Latitude",rdrlats, ...
    "Longitude",rdrlons);

% Create receiver sites associated with radars
rdrrxs = rxsite("Name",names, ...
    "AntennaHeight",10, ...
    "Latitude",rdrlats, ...
    "Longitude",rdrlons);

% Show just the radar transmitter sites
show(rdrtxs);
```
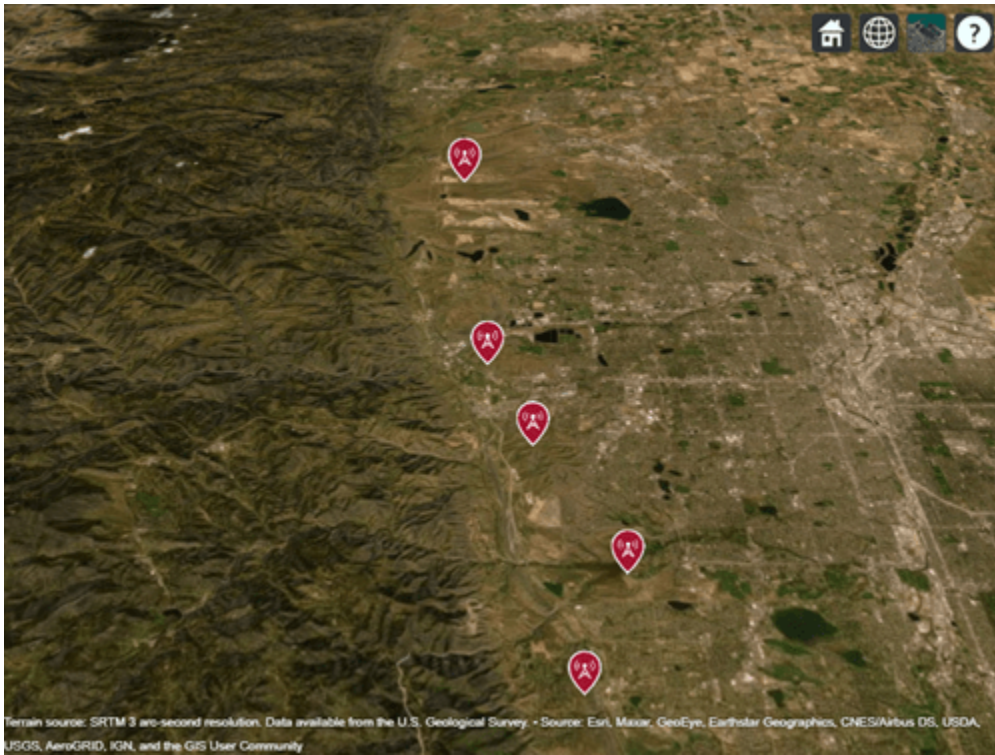
Zoom and rotate the map to view the 3-D terrain around the candidate radar sites. Select a site to view the location, antenna height, and ground elevation.

Design Monostatic Radar System

Design a basic monostatic pulse radar system to detect nonfluctuating targets with 0.1 square meter radar cross section (RCS) at a distance up to 35,000 meters from the radar with a range resolution of 5 meters. The desired performance index is a probability of detection (Pd) of 0.9 and probability of false alarm (Pfa) below 1e-6. The radars are assumed to be rotatable and support the same antenna gain in all directions, where the antenna gain corresponds to a highly directional antenna array.

```
pd = 0.9;           % Probability of detection
pfa = 1e-6;         % Probability of false alarm
maxrange = 35000;   % Maximum unambiguous range (m)
rangeres = 5;       % Required range resolution (m)
tgtrcs = .1;        % Required target radar cross section (m^2)
```

Use pulse integration to reduce the required SNR at the radar receiver. Use 10 pulses and compute the SNR required to detect a target.

```
numpulses = 10;
snrthreshold = albersheim(pd, pfa, numpulses); % Unit: dB
disp(snrthreshold);
```

```
    4.9904
```

Define radar center frequency and antenna gain assuming a highly directional antenna array.

```
fc = 10e9;      % Transmitter frequency: 10 GHz
antgain = 38; % Antenna gain: 38 dB
c = physconst('LightSpeed');
lambda = c/fc;
```

Calculate the required peak pulse power (Watts) of the radar transmitter using the radar equation.

```
pulsebw = c/(2*rangeres);
pulsewidth = 1/pulsebw;
Ptx = radareqpow(lambda,maxrange,snrthreshold,pulsewidth,...
    'RCS',tgtrcs,'Gain',antgain);
disp(Ptx)
```

```
    3.1521e+05
```

Define Target Positions

Define a grid containing 2500 locations to represent the geographic range of positions for a moving target in the region of interest. The region of interest spans 0.5 degrees in both latitude and longitude and includes mountains to the west as well as some of the area around the radar sites. The goal is to detect targets that are in the mountainous region to the west.

```
% Define region of interest
latlims = [39.5 40];
lonlims = [-105.6 -105.1];

% Define grid of target locations in region of interest
tgtlatv = linspace(latlims(1),latlims(2),50);
tgtlonv = linspace(lonlims(1),lonlims(2),50);
[tgtlons,tgtlats] = meshgrid(tgtlonv,tgtlatv);
tgtlons = tgtlons(:);
tgtlats = tgtlats(:);
```

Compute the minimum, maximum, and mean ground elevation for the target locations.

```
% Create temporary array of sites corresponding to target locations and query terrain
Z = elevation(txsite("Latitude",tgtlats,"Longitude",tgtlons));
[Zmin, Zmax] = bounds(Z);
Zmean = mean(Z);

disp("Ground elevation (meters): Min    Max    Mean" + newline + ...
     "                           " + round(Zmin) + "   " + round(Zmax) + "    " + round(Zmean))

Ground elevation (meters): Min    Max    Mean
                           1257   3953   2373
```

Target altitude can be defined with reference to mean sea level or to ground level. Use ground level as the reference and define a target altitude of 500 meters.

```
% Target altitude above ground level (m)
tgtalt = 500;
```

Show the region of interest as a solid green area on the map.

```
viewer.Name = "Radar Coverage Region of Interest";
regionData = propagationData(tgtlats,tgtlons,'Area',ones(size(tgtlats)));
contour(regionData,'ShowLegend',false,'Colors','green','Levels',0)
```
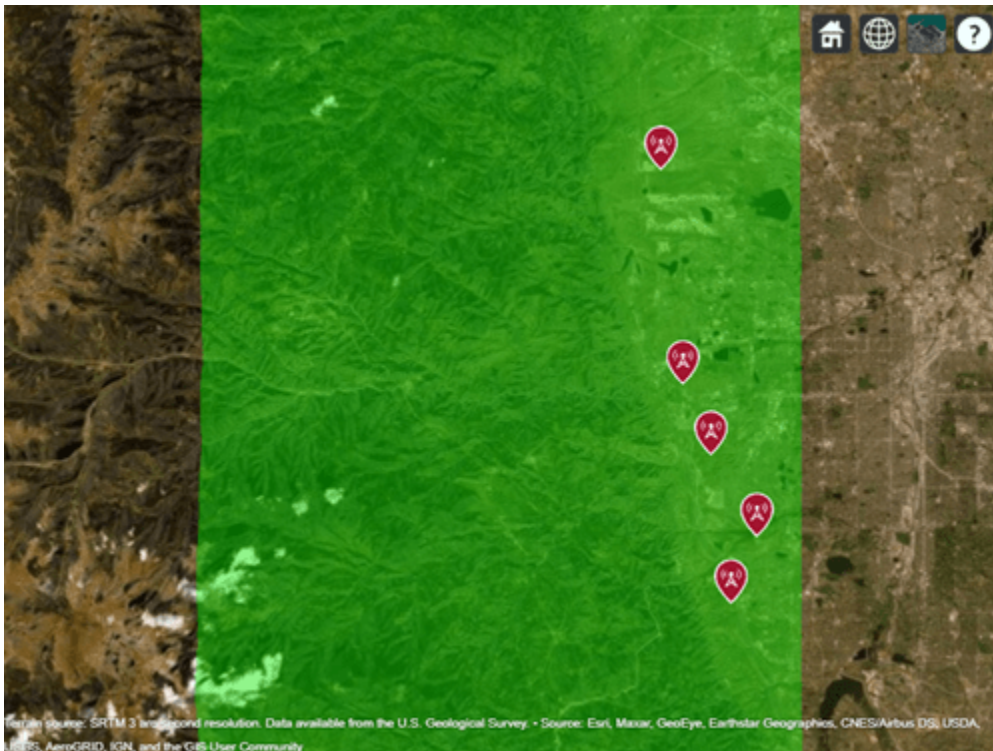

Calculate SNR for Target Positions with Terrain

The radar equation includes free space path loss and has a parameter for additional losses. Use a terrain propagation model to predict the additional path loss over terrain. Use TIREM™ from Huntington Ingalls Industries if it is available, or else use the Longley-Rice (ITM) model. TIREM™ supports frequencies up to 1000 GHz, whereas Longley-Rice is valid up to 20 GHz. Compute total additional loss including propagation from the radar to the target and then back from the target to the receiver.

```matlab
% Create a terrain propagation model, using TIREM or Longley-Rice
tiremloc = tiremSetup;
if ~isempty(tiremloc)
    pm = propagationModel('tirem');
else
    pm = propagationModel('longley-rice');
end

% Compute additional path loss due to terrain and return distances between radars and targets
[L, ds] = helperPathlossOverTerrain(pm, rdrtxs, rdrrxs, tgtlats, tgtlons, tgtalt);
```

Use the radar equation to compute the SNR at each radar receiver for the signal reflected from each target.

```matlab
% Compute SNR for all radars and targets
numtgts = numel(tgtlats);
numrdrs = numel(rdrtxs);
rawsnr = zeros(numtgts,numrdrs);
for tgtind = 1:numtgts
    for rdrind = 1:numrdrs
        rawsnr(tgtind,rdrind) = radareqsnr(lambda,ds(tgtind,rdrind),Ptx,pulsewidth, ...
            'Gain',antgain,'RCS',tgtrcs,'Loss',L(tgtind,rdrind));
    end
end
```

Optimize Radar Coverage

A target is detected if the radar receiver SNR exceeds the SNR threshold computed above. Consider all combinations of radar sites and select the three sites that produce the highest number of detections. Compute the SNR data as the best SNR available at the receiver of any of the selected radar sites.

```matlab
bestsitenums = helperOptimizeRadarSites(rawsnr, snrthreshold);
snr = max(rawsnr(:,bestsitenums),[],2);
```

Display radar coverage showing the area where the SNR meets the required threshold to detect a target. The three radar sites selected for best coverage are shown using red markers.

The coverage map shows straight edges on the north, east, and south sides corresponding to the limits of the region of interest. The coverage map assumes that the radars can rotate and produce the same antenna gain in all directions and that the radars can transmit and receive simultaneously so that there is no minimum coverage range.

The coverage map has jagged portions on the western edge where the coverage areas are limited by terrain effects. A smooth portion of the western edge appears where the coverage is limited by the design range of the radar system, which is 35000 meters.

```matlab
% Show selected radar sites using red markers
viewer.Name = "Radar Coverage";
clearMap(viewer)
show(rdrtxs(bestsitenums))

% Plot radar coverage
rdrData = propagationData(tgtlats,tgtlons,"SNR",snr);
legendTitle = "SNR" + newline + "(dB)";
contour(rdrData, ...
    "Levels",snrthreshold, ...
```

```
    "Colors","green", ...
    "LegendTitle",legendTitle)
```


Vary the Number of Pulses to Integrate

The previous analysis optimized radar transmitter power and site locations based on a system that integrates 10 pulses. Now investigate the impact on radar coverage for different modes of operation of the system, where the number of pulses to integrate is varied. Compute the SNR thresholds required to detect a target for varying number of pulses.

```
% Calculate SNR thresholds corresponding to different number of pulses
numpulses = 1:10;
snrthresholds = zeros(1,numel(numpulses));
for k = 1:numel(numpulses)
    snrthresholds(k) = albersheim(pd, pfa, numpulses(k));
end

% Plot SNR thresholds vs number of pulses to integrate
plot(numpulses,snrthresholds,'-*')
title("SNR at Radar Receiver Required for Detection")
xlabel("Number of pulses to integrate")
ylabel("SNR (dB)")
grid on;
```

SNR at Radar Receiver Required for Detection
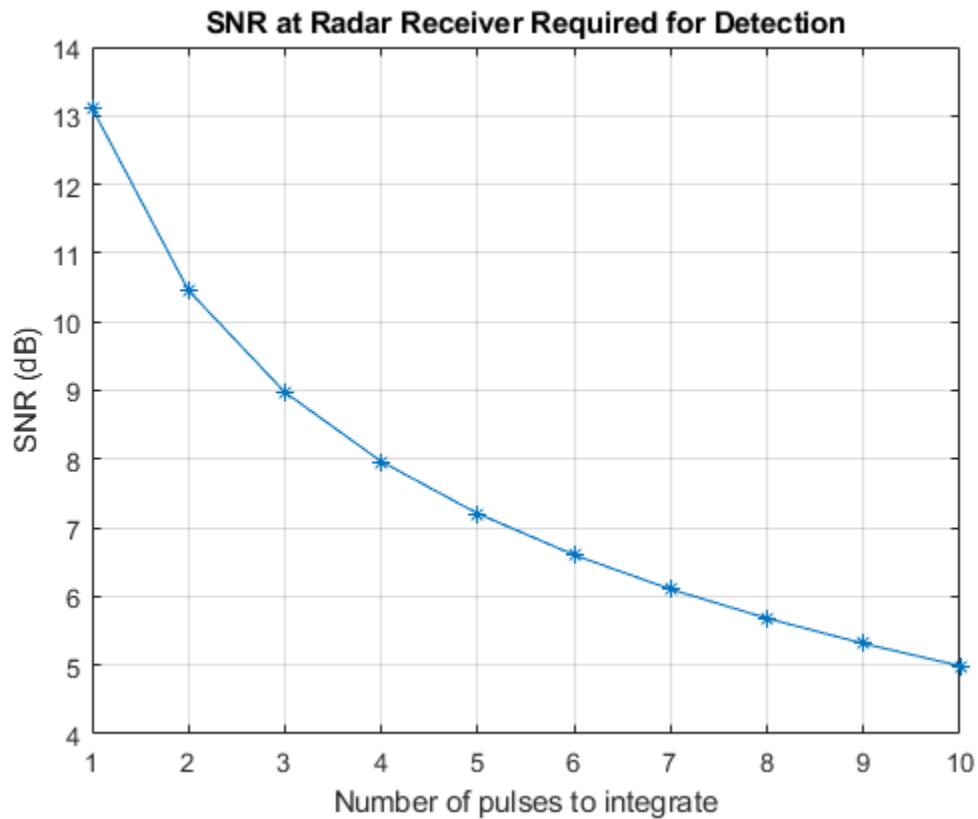
Show the radar coverage map for SNR thresholds corresponding to a few different numbers of pulses to integrate. Increasing the number of pulses to integrate decreases the required SNR and therefore produces a larger coverage region.

```matlab
% Show best sites
viewer.Name = "Radar Coverage for Multiple SNR Thresholds";
show(rdrtxs(bestsitenums))

colors = jet(4);
colors(4,:) = [0 1 0];
contour(rdrData, ...
    "Levels",snrthresholds([1 2 5 10]), ...
    "Colors",colors, ...
    "LegendTitle",legendTitle)
```
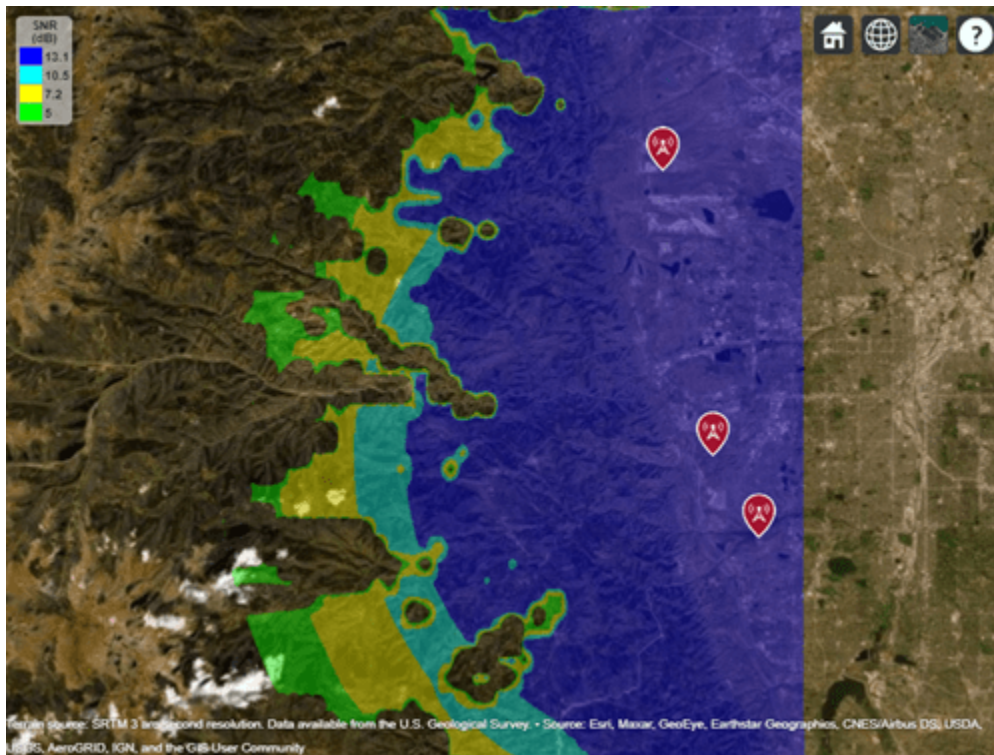
Update Target Altitude

Update the scenario so that target positions are 250 meters above ground level instead of 500 meters above ground level. Rerun the same analysis as above to select the three best radar sites and visualize coverage. The new coverage map shows that reducing the visibility of the targets also decreases the coverage area.

```
% Target altitude above ground (m)
tgtalt = 250;
[L, ds] = helperPathlossOverTerrain(pm, rdrtxs, rdrrxs, tgtlats, tgtlons, tgtalt);

% Compute SNR for all radars and targets
numrdrs = numel(rdrtxs);
rawsnr = zeros(numtgts,numrdrs);
for tgtind = 1:numtgts
    for rdrind = 1:numrdrs
        rawsnr(tgtind,rdrind) = radareqsnr(lambda,ds(tgtind,rdrind),Ptx,pulsewidth, ...
            'Gain',antgain,'RCS',tgtrcs,'Loss',L(tgtind,rdrind));
    end
end

% Select best combination of 3 radar sites
bestsitenums = helperOptimizeRadarSites(rawsnr, snrthreshold);
snr = max(rawsnr(:,bestsitenums),[],2);

% Show best sites
viewer.Name = "Radar Coverage";
clearMap(viewer);
show(rdrtxs(bestsitenums))
```

```
% Plot radar coverage
rdrData = propagationData(tgtlats,tgtlons,"SNR",snr);
contour(rdrData, ...
    "Levels",snrthreshold, ...
    "Colors","green", ...
    "LegendTitle",legendTitle)
```
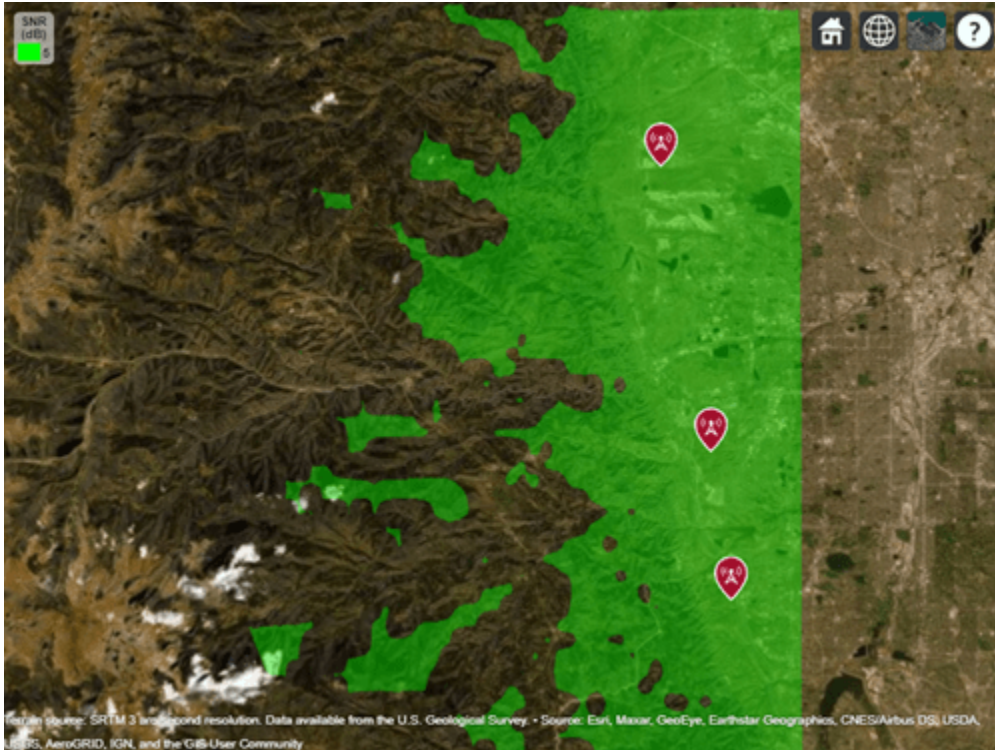

Show the radar coverage map for multiple SNR thresholds.

```
% Show best sites
viewer.Name = "Radar Coverage for Multiple SNR Thresholds";
show(rdrtxs(bestsitenums))

contour(rdrData, ...
    "Levels",snrthresholds([1 2 5 10]), ...
    "Colors",colors, ...
    "LegendTitle",legendTitle)
```
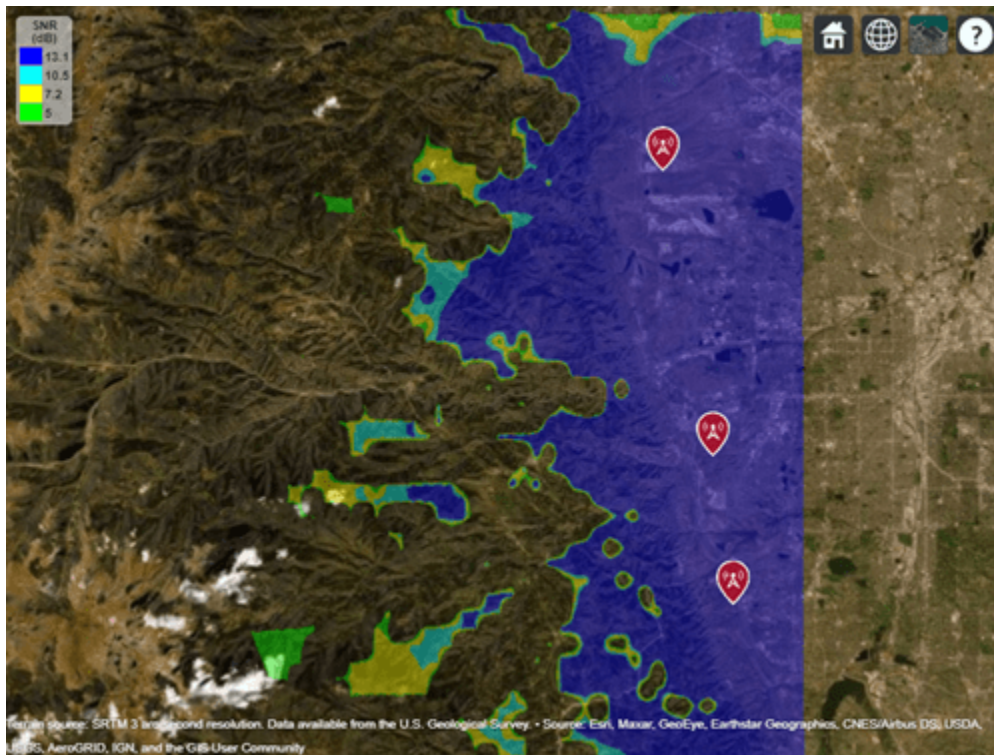
Conclusion

A monostatic radar system was designed to detect non-fluctuating targets with 0.1 square meter radar cross section (RCS) at a distance up to 35000 meters. Radar sites were selected among five candidate sites to optimize number of detections over a region of interest. Two target altitudes were considered: 500 meters above ground level and 250 meters above ground level. The coverage maps suggest the importance of line-of-sight visibility between the radar and target in order to achieve detection. The second scenario results in targets that are closer to ground level and therefore more likely to be blocked from line-of-sight visibility with a radar. This can be seen by rotating the map to view terrain, where non-coverage areas are typically located in shadow regions of the mountains.
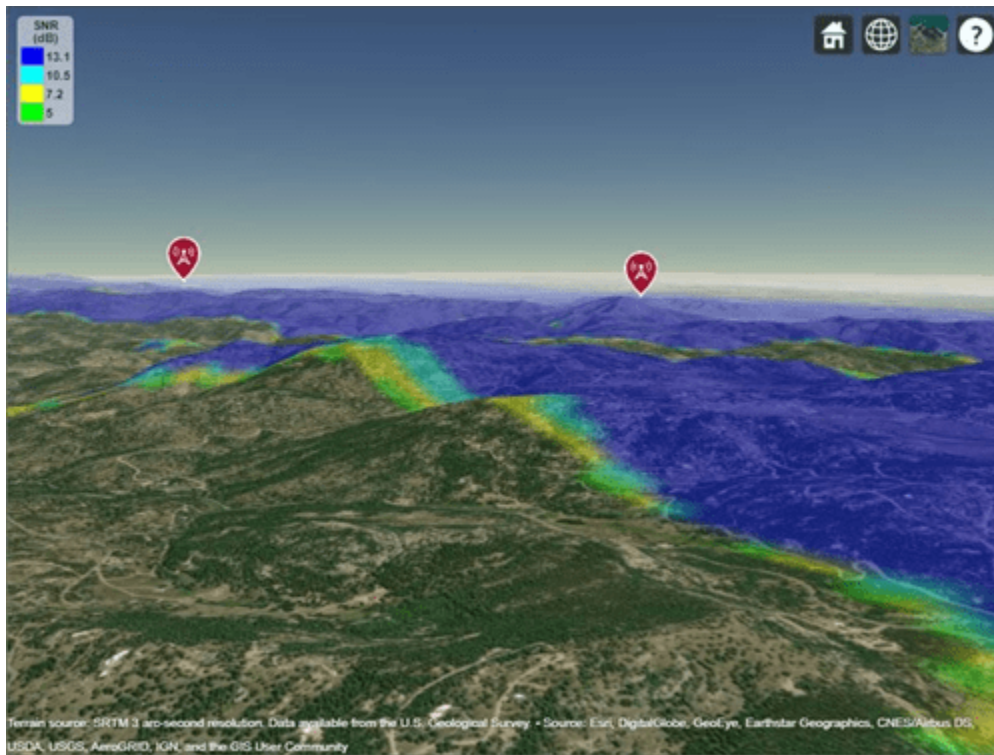
Clean up by closing Site Viewer and removing the imported terrain data.

```
close(viewer)
removeCustomTerrain("southboulder")
```

Maritime Radar Sea Clutter Modeling

This example will introduce a sea clutter simulation for a maritime surveillance radar system. This example first discusses the physical properties associated with sea states. Next, it discusses the reflectivity of sea surfaces, investigating the effect of sea state, frequency, polarization, and grazing angle. Lastly, the example calculates the clutter-to-noise ratio (CNR) for a maritime surveillance radar system, considering the propagation path and weather effects.

Overview of Sea States

In describing sea clutter, it is important first to establish the physical properties of the sea surface. In modeling sea clutter for radar, there are three important parameters:

1 σ_h is the standard deviation of the wave height. The wave height is defined as the vertical distance between the wave crest and the adjacent wave trough.

2 β_0 is the slope of the wave.

3 v_w is the wind speed.

Due to the irregularity of waves, the physical properties of the sea are often described in terms of sea states. The Douglas sea state number is a widely used scale that represents a wide range of physical sea properties such as wave heights and associated wind velocities. At the lower end of the scale, a sea state of 0 represents a calm, glassy sea state. The scale then proceeds from a slightly rippled sea state at 1 to rough seas with high wave heights at sea state 5. Wave heights at a sea state of 8 can be greater than 9 meters or more.

Using the `searoughness` function, plot the sea properties for sea states 1 through 5. Note the slow increase in the wave slope β_0 with sea state. This is a result of the wavelength and wave height increasing with wind speed, albeit with different factors.

```
% Analyze for sea states 1 through 5
ss = 1:5; % Sea states

% Initialize outputs
numSeaStates = numel(ss);
hgtsd = zeros(1,numSeaStates);
beta0 = zeros(1,numSeaStates);
vw= zeros(1,numSeaStates);

% Obtain sea state properties
for is = 1:numSeaStates
    [hgtsd(is),beta0(is),vw(is)] = searoughness(ss(is));
end

% Plot results
helperPlotSeaRoughness(ss,hgtsd,beta0,vw);
```
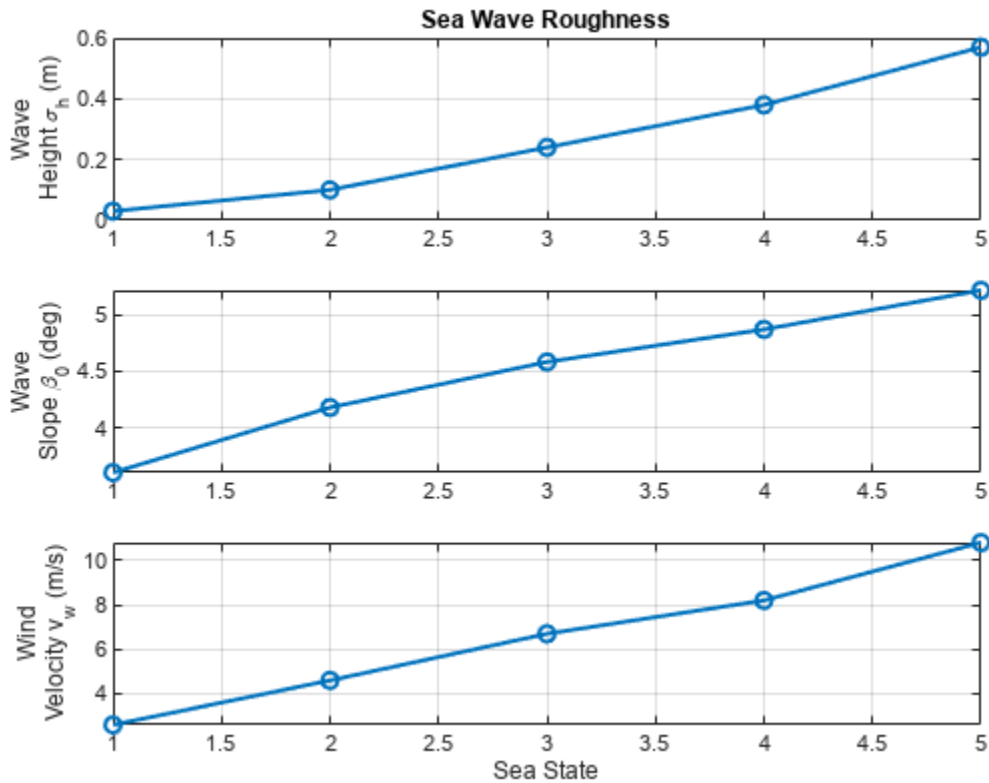
The physical properties you introduce is an important part in developing the geometry and environment of the maritime scenario. Furthermore, as you will see, radar returns from a sea surface exhibit strong dependence on sea state.

Reflectivity

The sea surface is composed of water with an average salinity of about 35 parts per thousand. The reflection coefficient of sea water is close to -1 for microwave frequencies and at low grazing angles.

For smooth seas, the wave height is small, and the sea appears as an infinite, flat conductive plate with little-to-no backscatter. As the sea state number increases and the wave height increases, the surface roughness increases. This results in increased scattering that is directionally dependent. Additionally, the reflectivity exhibits a strong proportional dependence on wave height and a dependence that increases with increasing frequency on wind speed.

Investigate sea surface reflectivity versus frequency for various sea states using the `seareflectivity` function. Set the grazing angle equal to 0.5 degrees and consider frequencies over the range of 500 MHz to 35 GHz.

```
grazAng = 0.5; % Grazing angle (deg)
freq = linspace(0.5e9,35e9,100); % Frequency (Hz)
pol = 'H'; % Horizontal polarization

% Initialize reflectivity output
numFreq = numel(freq);
nrcsH = zeros(numFreq,numSeaStates);
```

```matlab
% Calculate reflectivity
for is = 1:numSeaStates
    nrcsH(:,is) = seareflectivity(ss(is),grazAng,freq,'Polarization',pol);
end

% Plot reflectivity
helperPlotSeaReflectivity(ss,grazAng,freq,nrcsH,pol);
```
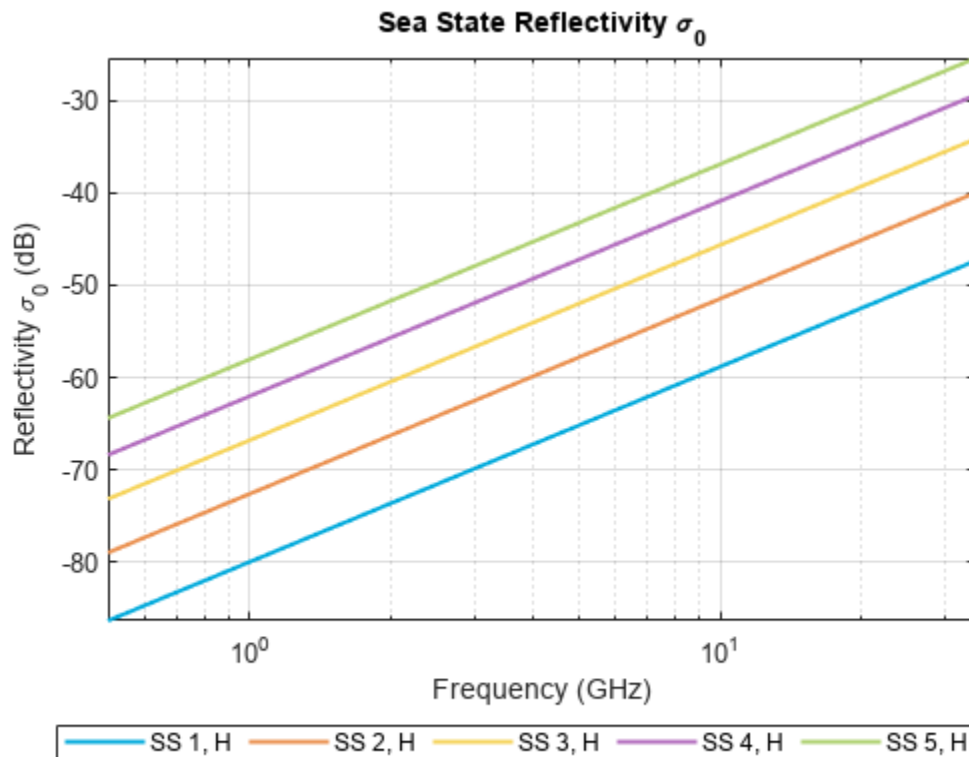

The figure shows that the sea surface reflectivity is proportional to frequency. Additionally, as the sea state number increases, which corresponds to increasing roughness, the reflectivity also increases.

Polarization Effects

Next, consider polarization effects on the sea surface reflectivity. Maintain the same grazing angle and frequency span from the previous section.

```matlab
pol = 'V'; % Vertical polarization

% Initialize reflectivity output
numFreq = numel(freq);
nrcsV = zeros(numFreq,numSeaStates);

% Calculate reflectivity
for is = 1:numSeaStates
    nrcsV(:,is) = seareflectivity(ss(is),grazAng,freq,'Polarization',pol);
end

% Plot reflectivity
```

```
hAxes = helperPlotSeaReflectivity(ss,grazAng,freq,nrcsH,'H');
helperPlotSeaReflectivity(ss,grazAng,freq,nrcsV,'V',hAxes);
```
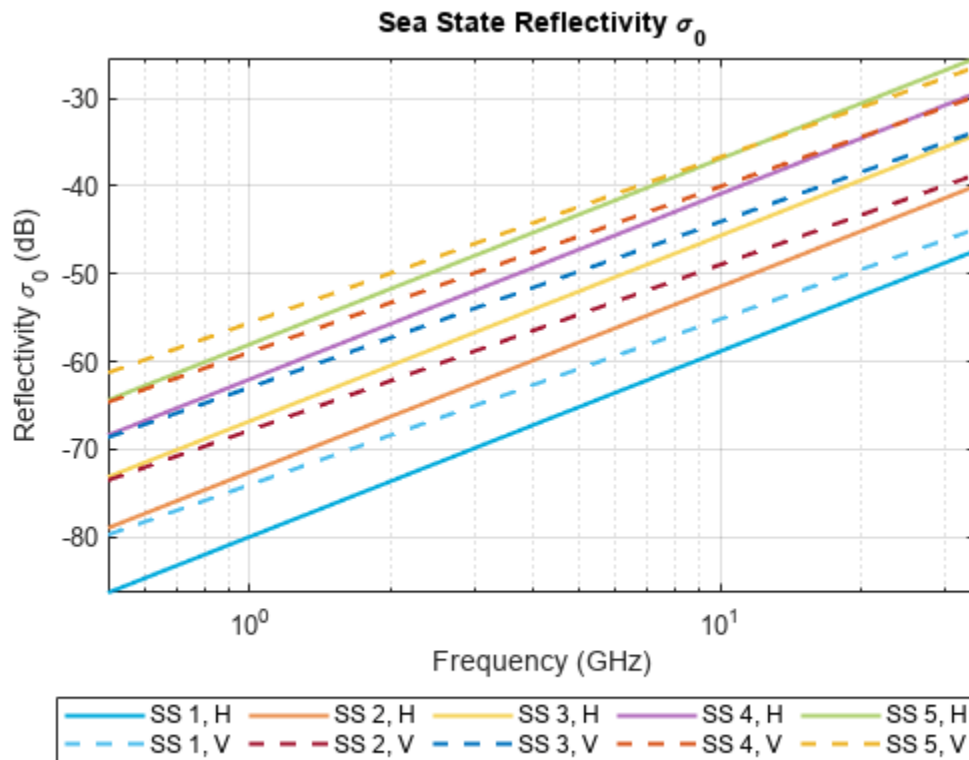

The figure shows that there is a noticeable effect on the reflectivity based on polarization. Notice that the difference between horizontal and vertical polarizations is greater at lower frequencies than at higher frequencies. As the sea state number increases, the difference between horizontal and vertical polarizations decreases. Thus, there is a decreasing dependence on polarization with increasing frequency.

Grazing Angle Effects

Consider the effect of grazing angle. Compute the sea reflectivity over the range of 0.1 to 60 degrees at an L-band frequency of 1.5 GHz.

```
grazAng = linspace(0.1,60,100); % Grazing angle (deg)
freq = 1.5e9; % L-band frequency (Hz)

% Initialize reflectivity output
numGrazAng = numel(grazAng);
nrcsH = zeros(numGrazAng,numSeaStates);
nrcsV = zeros(numGrazAng,numSeaStates);

% Calculate reflectivity
for is = 1:numSeaStates
    nrcsH(:,is) = seareflectivity(ss(is),grazAng,freq,'Polarization','H');
    nrcsV(:,is) = seareflectivity(ss(is),grazAng,freq,'Polarization','V');
end
```

```
% Plot reflectivity
hAxes = helperPlotSeaReflectivity(ss,grazAng,freq,nrcsH,'H');
helperPlotSeaReflectivity(ss,grazAng,freq,nrcsV,'V',hAxes);
ylim(hAxes,[-60 -10]);
```


Sea State Reflectivity σ_0
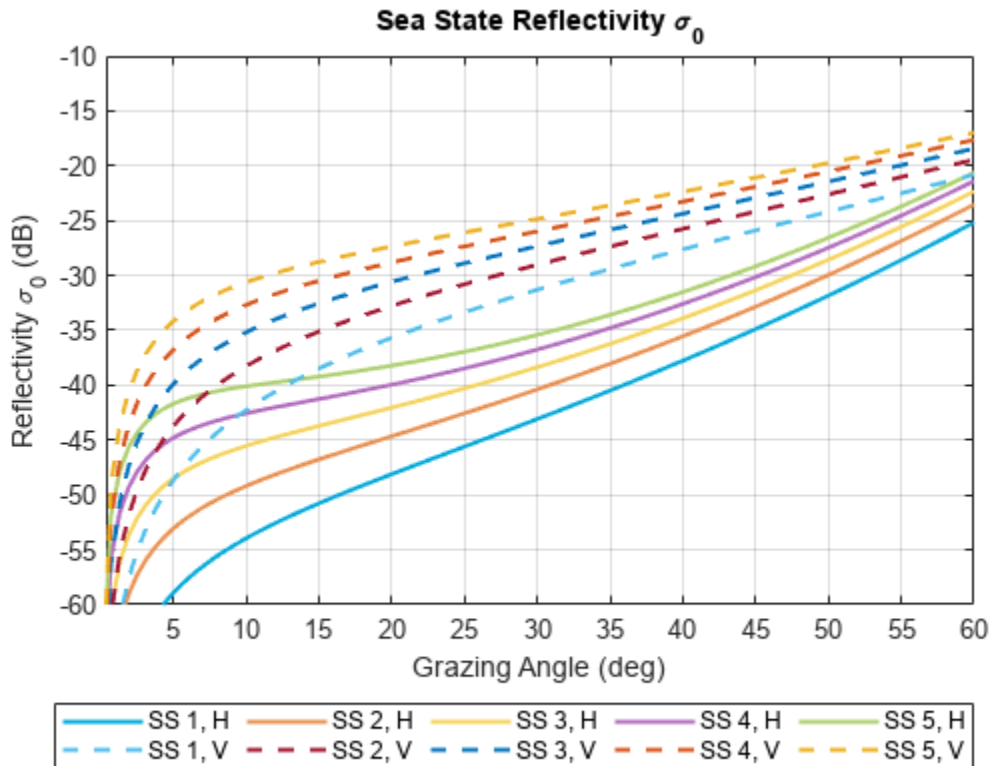
From the figure, note that there is much more variation in the sea reflectivity at lower grazing angles, and differences exist between vertical and horizontal polarization. The figure shows that the dependence on grazing angle decreases as the grazing angle increases. Furthermore, the reflectivity for horizontally polarized signals is less than vertically polarized signals for the same sea state over the range of grazing angles considered.

Maritime Surveillance Radar Example

Calculating Clutter-to-Noise Ratio

Consider a horizontally polarized maritime surveillance radar system operating at 6 GHz (C-band). Define the radar system.

```
% Radar parameters
freq = 6e9;           % C-band frequency (Hz)
anht = 20;            % Height (m)
ppow = 200e3;         % Peak power (W)
tau  = 200e-6;        % Pulse width (sec)
prf  = 300;           % PRF (Hz)
azbw = 10;            % Half-power azimuth beamwidth (deg)
elbw = 30;            % Half-power elevation beamwidth (deg)
```

```
Gt  = 22;          % Transmit gain (dB)
Gr  = 10;          % Receive gain (dB)
nf  = 3;           % Noise figure (dB)
Ts  = systemp(nf); % System temperature (K)
```

Next, simulate an operational environment where the sea state is 2. Calculate and plot the sea surface reflectivity for the grazing angles of the defined geometry.

```
% Sea parameters
ss = 2;            % Sea state

% Calculate surface state
[hgtsd,beta0] = searoughness(ss);

% Setup geometry
anht = anht + 2*hgtsd;        % Average height above clutter (m)
surfht = 3*hgtsd;            % Surface height (m)

% Calculate maximum range for simulation
Rua = time2range(1/prf); % Maximum unambiguous range (m)
Rhoriz = horizonrange(anht,'SurfaceHeight',surfht); % Horizon range (m)
Rmax = min(Rua,Rhoriz); % Maximum simulation range (m)

% Generate vector of ranges for simulation
Rm = linspace(100,Rmax,1000); % Range (m)
Rkm = Rm*1e-3;              % Range (km)

% Calculate sea clutter reflectivity. Temporarily permit values outside of
% the NRL sea reflectivity model grazing angle bounds of 0.1 - 60 degrees.
% Specifically, this is to permit analysis of grazing angles less than 0.1
% degrees that are close to the horizon.
grazAng = grazingang(anht,Rm,'TargetHeight',surfht);
warning('off','radar:radar:outsideValidityRegion'); % Permit values outside model
nrcs = seareflectivity(ss,grazAng,freq);
warning('on','radar:radar:outsideValidityRegion'); % Turn warnings back on
helperPlotSeaReflectivity(ss,grazAng,freq,nrcs,'H');
```
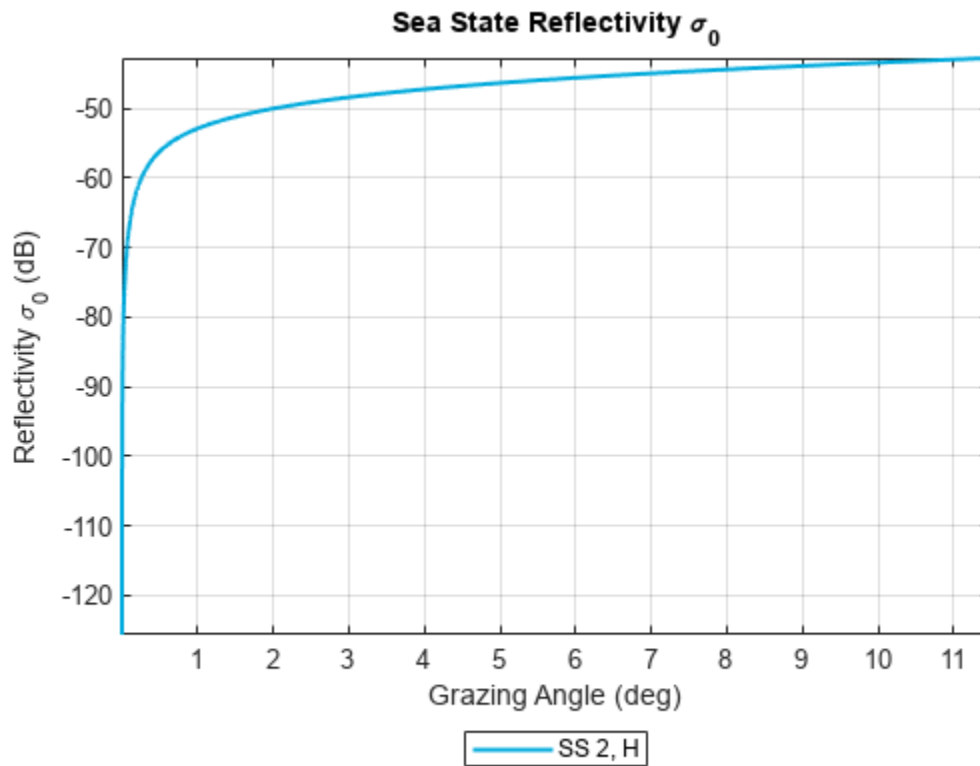
Next, calculate the radar cross section (RCS) of the clutter using the `clutterSurfaceRCS` function. Note the drop in the clutter RCS as the radar horizon range is reached.

```
% Calculate clutter RCS
rcs = clutterSurfaceRCS(nrcs,Rm,azbw,elbw,grazAng(:),tau);
rcsdB = pow2db(rcs); % Convert to decibels for plotting
hAxes = helperPlot(Rkm,rcsdB,'RCS','Clutter RCS (dBsm)','Clutter Radar Cross Section (RCS)');
helperAddHorizLine(hAxes,Rhoriz);
```

Clutter Radar Cross Section (RCS)
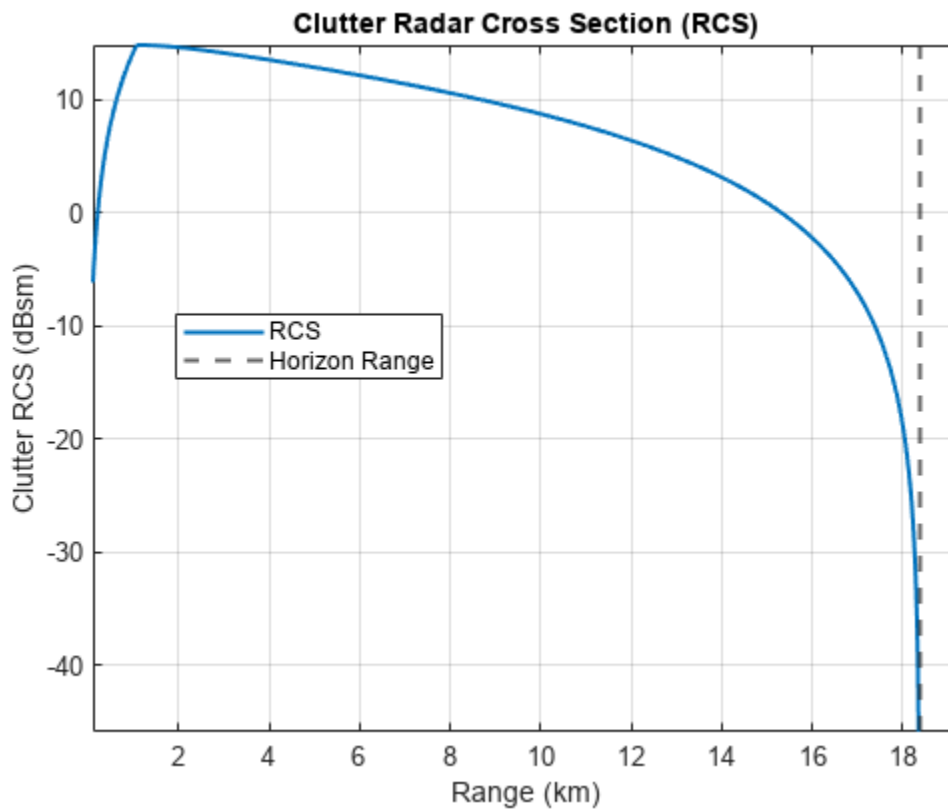
Calculate the clutter-to-noise ratio (CNR) using the `radareqsnr` function. Again, note the drop in CNR as the simulation range approaches the radar horizon. Calculate the range at which the clutter falls below the noise.

```
% Convert frequency to wavelength
lambda = freq2wavelen(freq);

% Calculate and plot the clutter-to-noise ratio
cnr = radareqsnr(lambda,Rm(:),ppow,tau,...
        'gain',[Gt Gr],'rcs',rcs,'Ts',Ts); % dB
hAxes = helperPlot(Rkm,cnr,'CNR','CNR (dB)','Clutter-to-Noise Ratio (CNR)');
ylim(hAxes,[-80 100]);
helperAddHorizLine(hAxes,Rhoriz);
helperAddBelowClutterPatch(hAxes);
```
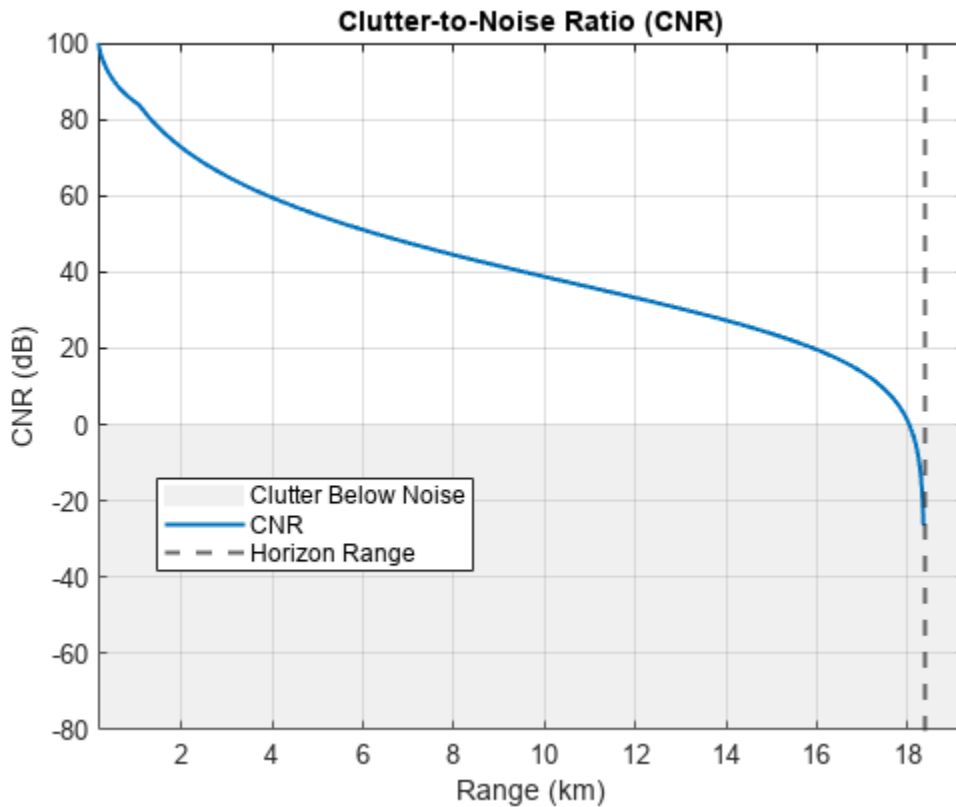
Clutter-to-Noise Ratio (CNR)

```
% Range when clutter falls below noise
helperFindClutterBelowNoise(Rkm,cnr);
```

```
Range at which clutter falls below noise (km) = 18.04
```

Considering the Propagation Path

When the path between the radar and clutter deviates from free space conditions, include the clutter propagation factor and the atmospheric losses on the path. You can calculate the clutter propagation factor using the `radarpropfactor` function.

```
% Calculate radar propagation factor for clutter
Fc = radarpropfactor(Rm,freq,anht,surfht, ...
    'SurfaceHeightStandardDeviation',hgtsd,...
    'SurfaceSlope',beta0,...
    'ElevationBeamwidth',elbw);
helperPlot(Rkm,Fc,'Propagation Factor', ...
    'Propagation Factor (dB)', ...
    'One-Way Clutter Propagation Factor F_C');
```

One-Way Clutter Propagation Factor F$_C$
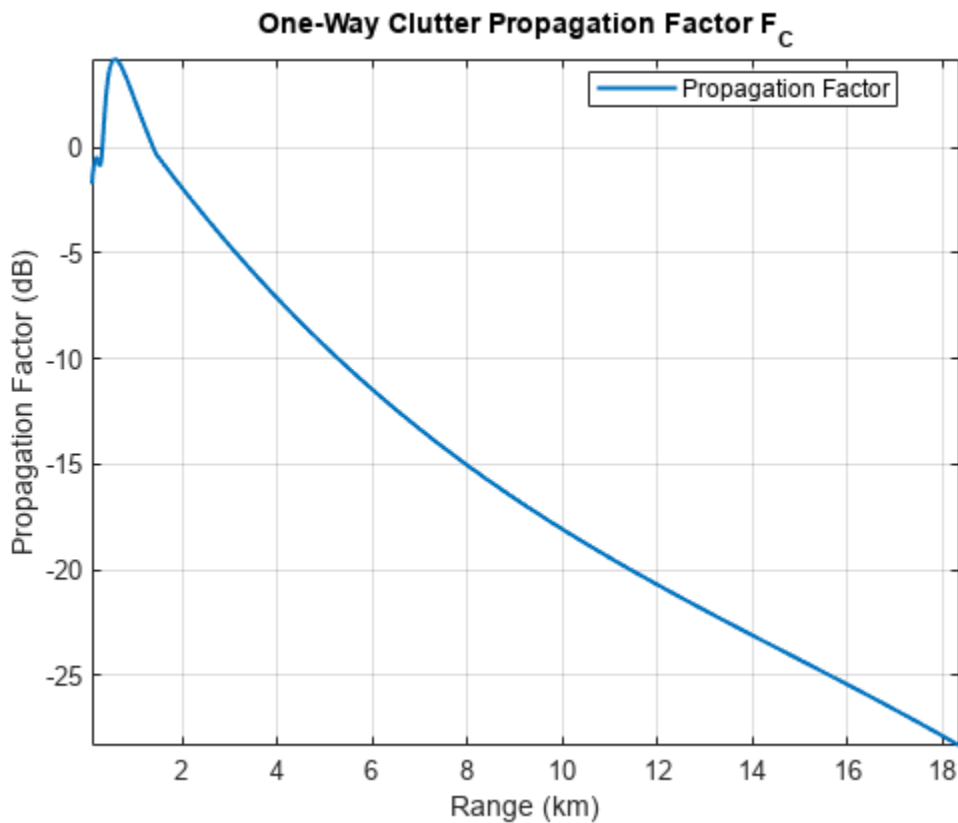
Within the above plot, two propagation regions are visible:

1   Interference region: This is the region where reflections interfere with the direct ray. This is exhibited over the ranges where there is lobing.

2   Intermediate region: This is the region between the interference and diffraction region, where the diffraction region is defined as a shadow region beyond the horizon. The intermediate region, which in this example occurs at the kink in the curve at about 1.5 km, is generally estimated by an interpolation between the interference and diffraction regions.

Typically, the clutter propagation factor and the sea reflectivity are combined as the product $\sigma_C F_C^4$, because measurements of surface reflectivity are generally measurements of the product rather than just the reflectivity σ_C. Calculate this product and plot the results.

```
% Combine clutter reflectivity and clutter propagation factor
FcLinear = db2mag(Fc); % Convert to linear units
combinedFactor = nrcs.*FcLinear.^2;
combinedFactordB = pow2db(combinedFactor);
helperPlot(Rkm,combinedFactordB,'\sigma_CF_C', ...
    '\sigma_CF_C (dB)', ...
    'One-Way Sea Clutter Propagation Factor and Reflectivity');
```
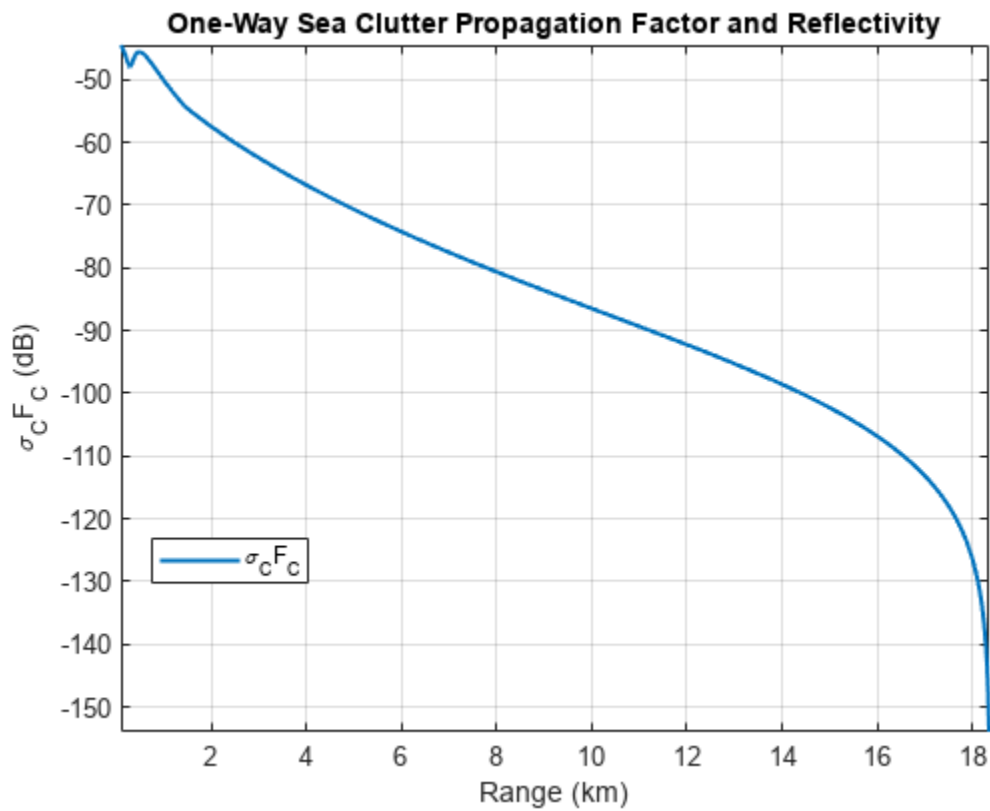
Next, calculate the atmospheric loss on the path using the slant-path `tropopl` function. Use the default standard atmospheric model for the calculation.

```
% Calculate one-way loss associated with atmosphere
elAng = height2el(surfht,anht,Rm); % Elevation angle (deg)
numEl = numel(elAng);
Latmos = zeros(numEl,1);
for ie = 1:numEl
    Latmos(ie,:) = tropopl(Rm(ie),freq,anht,elAng(ie));
end
helperPlot(Rkm,Latmos,'Atmospheric Loss','Loss (dB)','One-Way Atmospheric Loss');
```

One-Way Atmospheric Loss
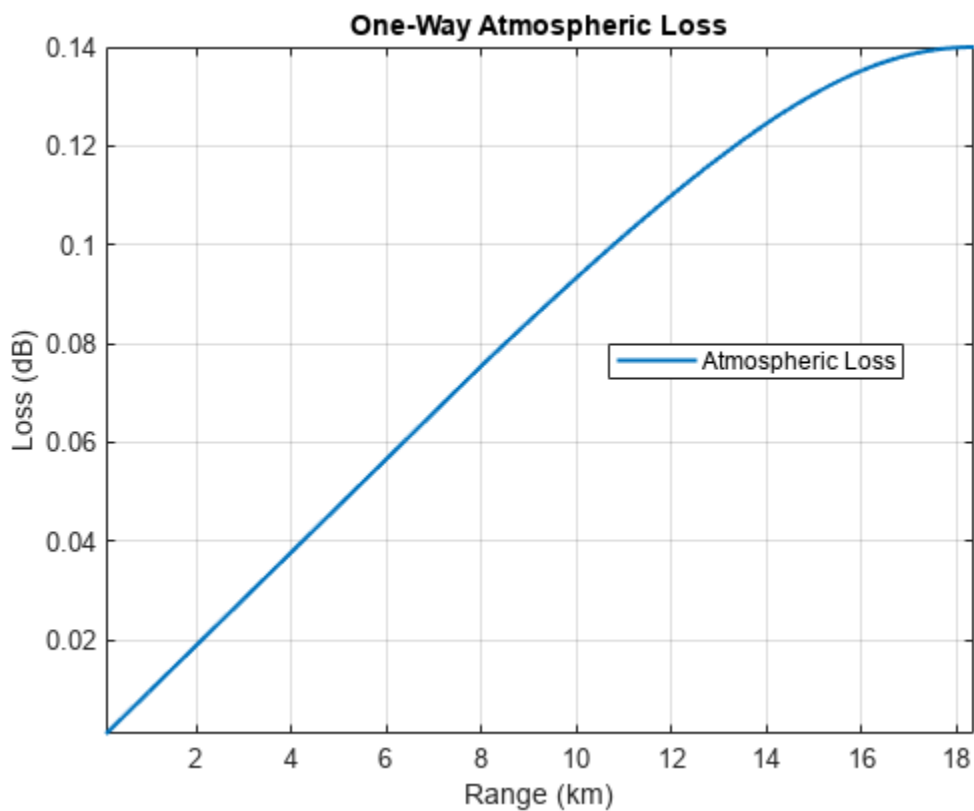
Recalculate the CNR. Include the propagation factor and atmospheric loss in the calculation. Note the change in the shape of the CNR curve. The point at which the clutter falls below the noise is much closer in range when you include these factors.

```
% Re-calculate CNR including radar propagation factor and atmospheric loss
cnr = radareqsnr(lambda,Rm(:),ppow,tau,...
        'gain',[Gt Gr],'rcs',rcs,'Ts',Ts, ...
        'PropagationFactor',Fc,...
        'AtmosphericLoss',Latmos); % dB
helperAddPlot(Rkm,cnr,'CNR + Propagation Factor + Atmospheric Loss',hAxes);
```
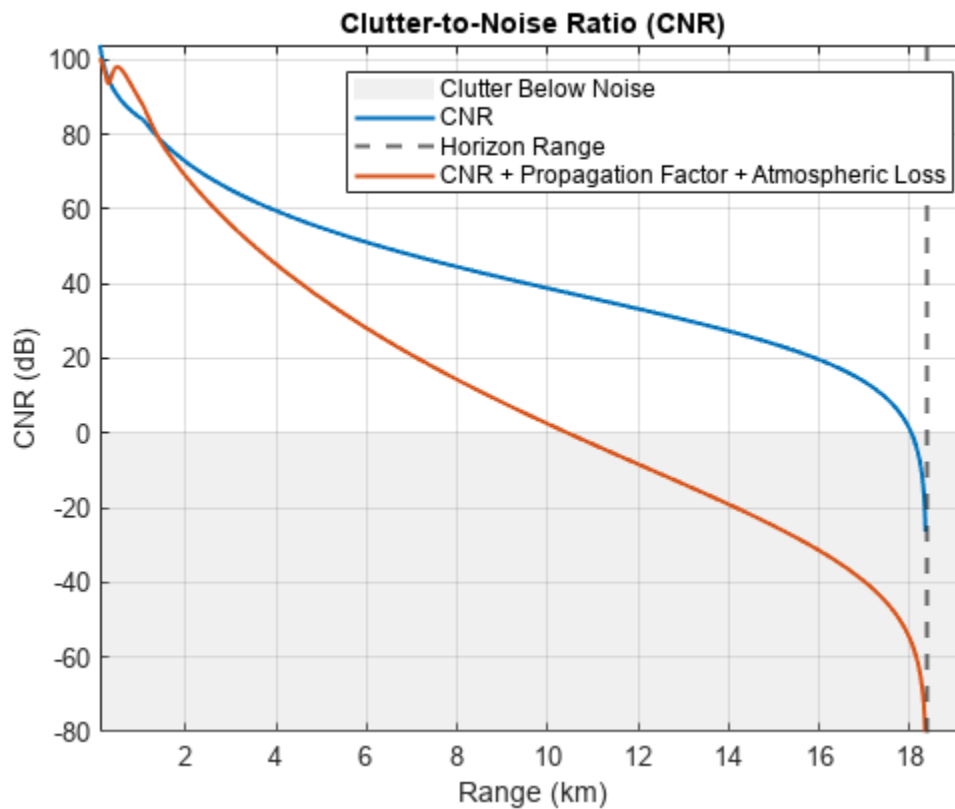
```
% Range when clutter falls below noise
helperFindClutterBelowNoise(Rkm,cnr);
```

Range at which clutter falls below noise (km) = 10.44

Understanding Weather Effects

Just as the atmosphere affects the detection of a target, weather also affects the detection of clutter. Consider the effect of rain over the simulated ranges. First calculate the rain attenuation.

```
% Calculate one-way loss associated with rain
rr = 50;                              % Rain rate (mm/h)
polAng = 0;                           % Polarization tilt angle (0 degrees for horizontal)
elAng = height2el(surfht,anht,Rm);    % Elevation angle (deg)
numEl = numel(elAng);
Lrain = zeros(numEl,1);
for ie = 1:numEl
    Lrain(ie,:) = cranerainpl(Rm(ie),freq,rr,elAng(ie),polAng);
end
helperPlot(Rkm,Lrain,'Rain Loss','Loss (dB)','One-Way Rain Loss');
```
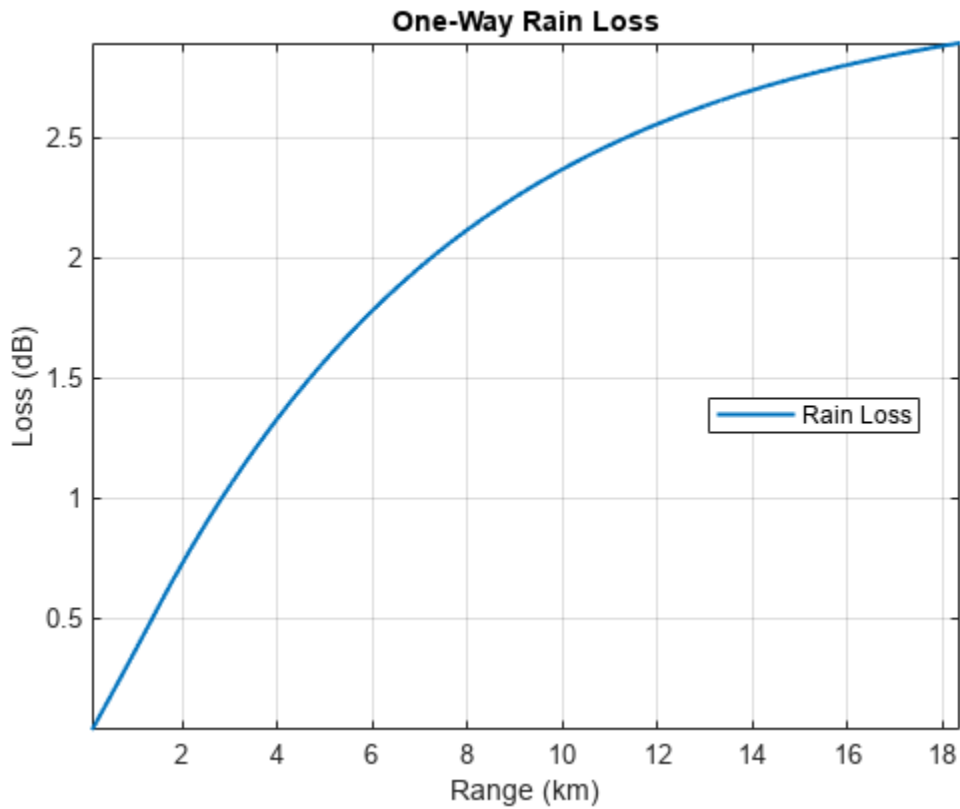
Recalculate the CNR. Include the propagation path and the rain loss. Note that there is only a slight decrease in the CNR due to the presence of the rain.

```
% Re-calculate CNR including radar propagation factor, atmospheric loss,
% and rain loss
cnr = radareqsnr(lambda,Rm(:),ppow,tau,...
        'gain',[Gt Gr],'rcs',rcs,'Ts',Ts, ...
        'PropagationFactor',Fc, ...
        'AtmosphericLoss',Latmos + Lrain); % dB
helperAddPlot(Rkm,cnr,'CNR + Propagation Factor + Atmospheric Loss + Rain',hAxes);
```
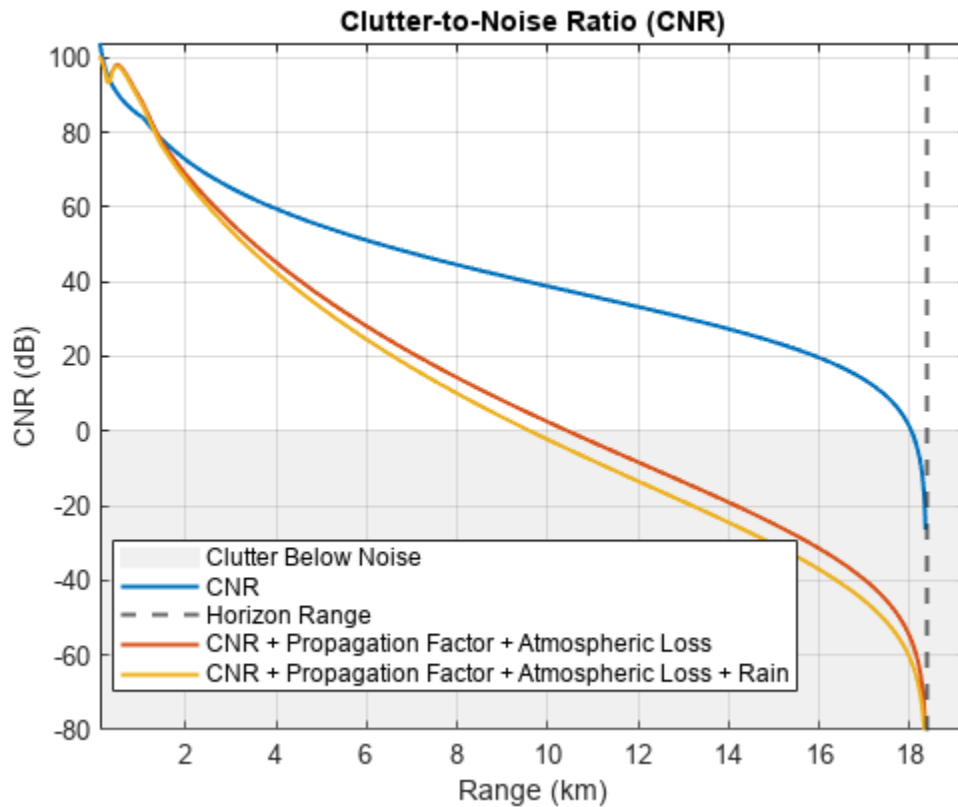
```
% Range when clutter falls below noise
helperFindClutterBelowNoise(Rkm,cnr);
```

```
Range at which clutter falls below noise (km) = 9.61
```

Summary

This example introduces concepts regarding the simulation of sea surfaces. The sea reflectivity exhibits the following properties:

- A strong dependence on sea state
- A proportional dependence on frequency
- A dependence on polarization that decreases with increasing frequency
- A strong dependence on grazing angle at low grazing angles

This example also discusses how to use the sea state physical properties and reflectivity for the calculation of the clutter-to-noise ratio for a maritime surveillance radar system. Additionally, the example explains ways to improve simulation of the propagation path.

References

1 Barton, David Knox. *Radar Equations for Modern Radar*. Artech House Radar Series. Boston, Mass: Artech House, 2013.

2 Blake, L. V. *Machine Plotting of Radar Vertical-Plane Coverage Diagrams*. NRL Report, 7098, Naval Research Laboratory, 1970.

3 Gregers-Hansen, V., and R. Mittal. *An Improved Empirical Model for Radar Sea Clutter Reflectivity*. NRL/MR, 5310-12-9346, Naval Research Laboratory, 27 Apr. 2012.

4 Richards, M. A., Jim Scheer, William A. Holm, and William L. Melvin, eds. *Principles of Modern Radar*. Raleigh, NC: SciTech Pub, 2010.

```matlab
function helperPlotSeaRoughness(ss,hgtsd,beta0,vw)
% Creates 3x1 plot of sea roughness outputs

% Create figure
figure

% Plot standard deviation of sea wave height
subplot(3,1,1)
plot(ss,hgtsd,'-o','LineWidth',1.5)
ylabel([sprintf('Wave\nHeight ') '\sigma_h (m)'])
title('Sea Wave Roughness')
grid on;

% Plot sea wave slope
subplot(3,1,2)
plot(ss,beta0,'-o','LineWidth',1.5)
ylabel([sprintf('Wave\nSlope ') '\beta_0 (deg)'])
grid on;

% Plot wind velocity
subplot(3,1,3)
plot(ss,vw,'-o','LineWidth',1.5)
xlabel('Sea State')
ylabel([sprintf('Wind\nVelocity ')  'v_w (m/s)'])
grid on;
end

function hAxes = helperPlotSeaReflectivity(ss,grazAng,freq,nrcs,pol,hAxes)
% Plot sea reflectivities

% Create figure and new axes if axes are not passed in
newFigure = false;
if nargin < 6
    figure();
    hAxes = gca;
    newFigure = true;
end

% Get polarization string
switch lower(pol)
    case 'h'
        lineStyle = '-';
    otherwise
        lineStyle = '--';
end

% Plot
if numel(grazAng) == 1
    hLine = semilogx(hAxes,freq(:).*1e-9,pow2db(nrcs),lineStyle,'LineWidth',1.5);
    xlabel('Frequency (GHz)')
else
    hLine = plot(hAxes,grazAng(:),pow2db(nrcs),lineStyle,'LineWidth',1.5);
```

```matlab
    xlabel('Grazing Angle (deg)')
end

% Set display names
numLines = size(nrcs,2);
for ii = 1:numLines
    hLine(ii).DisplayName = sprintf('SS %d, %s',ss(ii),pol);
    if newFigure
        hLine(ii).Color = brighten(hLine(ii).Color,0.5);
    end
end

% Update labels and axes
ylabel('Reflectivity \sigma_0 (dB)')
title('Sea State Reflectivity \sigma_0')
grid on
axis tight
hold on;

% Add legend
legend('Location','southoutside','NumColumns',5,'Orientation','Horizontal');
end

function varargout = helperPlot(Rkm,y,displayName,ylabelStr,titleName)
% Used in CNR analysis

% Create figure
hFig = figure;
hAxes = axes(hFig);

% Plot
plot(hAxes,Rkm,y,'LineWidth',1.5,'DisplayName',displayName);
grid(hAxes,'on');
hold(hAxes,'on');
xlabel(hAxes,'Range (km)')
ylabel(hAxes,ylabelStr);
title(hAxes,titleName);
axis(hAxes,'tight');

% Add legend
legend(hAxes,'Location','Best')

% Output axes
if nargout ~= 0
    varargout{1} = hAxes;
end
end

function helperAddPlot(Rkm,y,displayName,hAxes)
% Used in CNR analysis

% Plot
ylimsIn = get(hAxes,'Ylim');
plot(hAxes,Rkm,y,'LineWidth',1.5,'DisplayName',displayName);
axis(hAxes,'tight');
ylimsNew = get(hAxes,'Ylim');
set(hAxes,'Ylim',[ylimsIn(1) ylimsNew(2)]);
end
```

```matlab
function helperAddHorizLine(hAxes,Rhoriz)
% Add vertical line indicating horizon range

xline(Rhoriz.*1e-3,'--','DisplayName','Horizon Range','LineWidth',1.5);
xlims = get(hAxes,'XLim');
xlim([xlims(1) Rhoriz.*1e-3*(1.05)]);
end

function helperAddBelowClutterPatch(hAxes)
% Add patch indicating when clutter falls below the noise

xlims = get(hAxes,'Xlim');
ylims = get(hAxes,'Ylim');
x = [xlims(1) xlims(1) xlims(2) xlims(2) xlims(1)];
y = [ylims(1) 0 0 ylims(1) ylims(1)];
hP = patch(hAxes,x,y,[0.8 0.8 0.8], ...
    'FaceAlpha',0.3,'EdgeColor','none','DisplayName','Clutter Below Noise');
uistack(hP,'bottom');
end

function helperFindClutterBelowNoise(Rkm,cnr)
% Find the point at which the clutter falls below the noise
idxNotNegInf = ~isinf(cnr);
Rclutterbelow = interp1(cnr(idxNotNegInf),Rkm(idxNotNegInf),0);
fprintf('Range at which clutter falls below noise (km) = %.2f\n',Rclutterbelow)
end
```

Radar Scenario Tutorial

This example shows how to construct and visualize a simple radar scenario programmatically using the `radarScenario`, `theaterPlot`, and `radarDataGenerator` objects.

Scenario Setup

To begin, create an empty radar scenario. All scenario properties have default values. The scenario does not contain any platform by default.

```
scenario = radarScenario
```

```
scenario =

  radarScenario with properties:

       IsEarthCentered: 0
            UpdateRate: 10
        SimulationTime: 0
              StopTime: Inf
      SimulationStatus: NotStarted
             Platforms: {}
         SurfaceManager: [1x1 radar.scenario.SurfaceManager]
      AtmosphereManager: [1x1 radar.scenario.AtmosphereManager]
```

Adding Platforms

A scenario is comprised of objects, called platforms, upon which you may mount sensors and emitters. To add a platform, use the `platform` object function. Here you create a simple tower.

```
tower = platform(scenario)
```

```
tower =

  Platform with properties:

         PlatformID: 1
            ClassID: 0
           Position: [0 0 0]
        Orientation: [0 0 0]
         Dimensions: [1x1 struct]
         Trajectory: [1x1 kinematicTrajectory]
       PoseEstimator: [1x1 insSensor]
           Emitters: {}
            Sensors: {}
          Signatures: {[1x1 rcsSignature]}
```

Platform Identification

When you first construct a platform, it has a `PlatformID`, which is a unique identifier you can use to identify the platform. The scenario assigns platform identifiers in the order that the platforms are created. You can specify a `ClassID` to denote the platform's classification. For example, here you use a 3 to denote a tower.

```
tower.ClassID = 3;
```

Platform Signatures

Sensors can detect platforms. For radar sensors, a relevant signature is the radar cross-section (RCS). By default a uniform RCS signature is used:

```
tower.Signatures{1}
```

```
ans =

  rcsSignature with properties:

    EnablePolarization: 0
                Pattern: [2x2 double]
                Azimuth: [-180 180]
              Elevation: [2x1 double]
              Frequency: [0 1.0000e+20]
        FluctuationModel: Swerling0
```

Load predefined cylinder RCS data and use the data to define the RCS of the tower platform.

```
load('RCSCylinderExampleData.mat','cylinder');
```

```
tower.Signatures{1} = rcsSignature('Pattern', cylinder.RCSdBsm, ...
        'Azimuth', cylinder.Azimuth, 'Elevation', cylinder.Elevation, ...
        'Frequency', cylinder.Frequency);
```

Platform Dimensions

By default, platforms have no dimensions and are modeled as point targets. You may optionally specify the length, width, and height to denote the extent of the object, along with an offset of the body frame origin from its from its geometric center. You can specify platform dimensions using the `Dimensions` property.

You specify the dimensions of the tower like this:

```
tower.Dimensions = struct( ...
    'Length', 10, ...
    'Width', 10, ...
    'Height', 60, ...
    'OriginOffset', [0 0 30]);
```

The tower has a length, width, and height of 10, 10, and 60 meters. The origin offset of [0 0 30] indicates that its body frame origin (rotational center) is 30 meters in the positive z-direction of the platform's local body axis.

Platform Trajectories

You can obtain a platform's current position and orientation through its `Position` and `Orientation` properties. You can obtain more information about platforms using the scenario's platformPoses method:

```
scenario.platformPoses
```

```
ans =
```

```
struct with fields:

        PlatformID: 1
          ClassID: 3
         Position: [0 0 0]
         Velocity: [0 0 0]
     Acceleration: [0 0 0]
      Orientation: [1x1 quaternion]
  AngularVelocity: [0 0 0]
```

You can specify a platform's position and orientation over time using its `Trajectory` property. You can specify the trajectory of a platform using a `kinematicTrajectory`, `waypoinTrajectory`, or `geoTrajectory` object.

By default, a platform consists of a static `kineticTrajectory` that whose body axis is perfectly centered and aligned with the scenario axes:

`tower.Trajectory`

```
ans =

  kinematicTrajectory with properties:

             SampleRate: 100
               Position: [0 0 0]
            Orientation: [1x1 quaternion]
               Velocity: [0 0 0]
     AccelerationSource: 'Input'
  AngularVelocitySource: 'Input'
```

To obtain a pitch angle of 4 degrees, you use the `Orientation` property of the trajectory object. Specify the orientation using a quaternion obtained from Euler angles.

```
tYaw = 0;
tPitch = 4;
tRoll = 0;
tower.Trajectory.Orientation = quaternion([tYaw tPitch tRoll],'eulerd','zyx','frame');
```

Axes Conventions

Most examples in Radar Toolbox, use a "North-East-Down" convention. This means that the x-axis points towards north, the y-axis points toward east, and the z-axis points downwards. Also, the x-, y-, and z- directions of the local body frame are the forward, right, and downward directions, respectively.

Visualizing a Scenario

The `theaterPlot` object provides an interface to plot objects dynamically in a 3-D scene. You may use standard MATLAB axes plotting methods to add or remove annotations to the theater plot's axes, which you can obtain via its `Parent` property.

Use a `platformPlotter` to plot platforms.

As expected, the tower is centered at the origin in NED coordinates with a pitch of 4 degrees.

```
tPlot = theaterPlot('XLim',[-50 50],'YLim',[-50 50],'ZLim',[-100 0]);
pPlotter = platformPlotter(tPlot,'DisplayName','tower');

pose = platformPoses(scenario);
towerPose = pose(1);
towerDims = tower.Dimensions;

plotPlatform(pPlotter, towerPose.Position, towerDims, towerPose.Orientation);
set(tPlot.Parent,'YDir','reverse', 'ZDir','reverse');
view(tPlot.Parent, -37.5, 30)
```
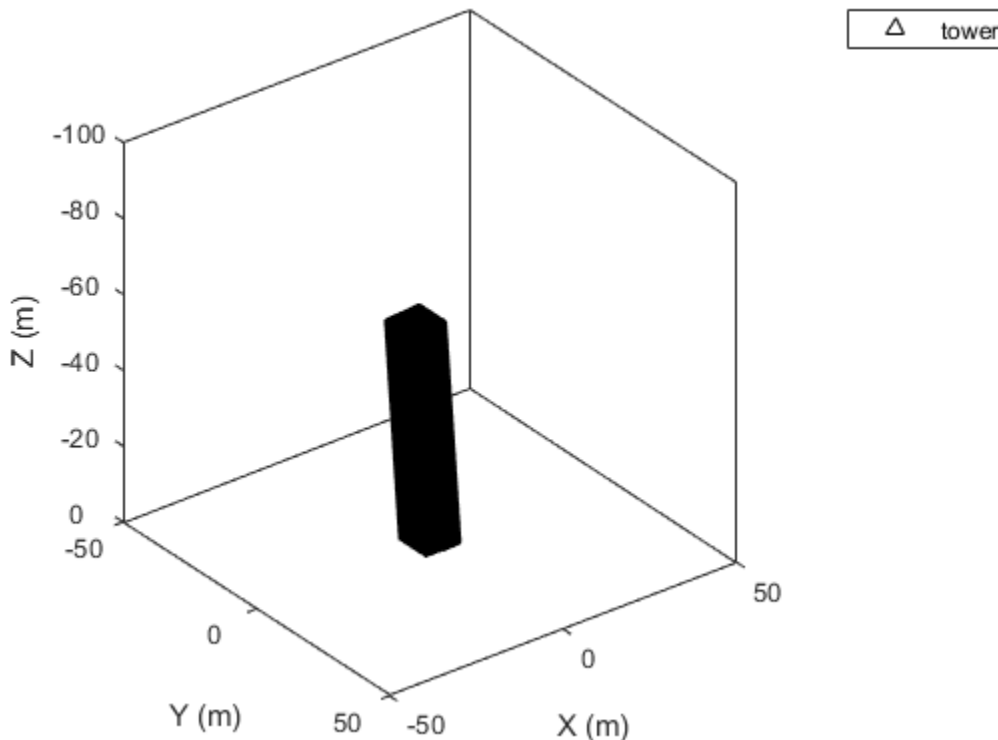

Adding Sensors to Platforms

To add a radar sensor to the platform, you can add a `radarDataGenerator` object on the platform by using its `Sensors` property.

Keep in mind that in a NED coordinate system, the "Z" direction points down. Therefore, if you want to mount a radar at the top of the tower, you should set its "z" position to -60 meters.

The `radarDataGenerator` has the option to report detections in scenario coordinates. Reporting detections in scenario coordinates makes it easier to compare the generated detections with the positions of the objects in the scenario.

```
towerRadar = radarDataGenerator('SensorIndex', 1, ...
    'UpdateRate', 10, ...
    'MountingLocation', [0 0 -60], ...
    'ScanMode', 'No scanning', ...
```

```
        'HasINS', true, ...
        'TargetReportFormat', 'Detections', ...
        'DetectionCoordinates', 'Scenario');

tower.Sensors = towerRadar;
```

Visualizing Coverage Areas

To see sensor coverages in a scenario, you use a coverage plotter and plot the coverage configuration of the scenario. You can widen the theater plot's range by adjusting the limits of its internal axes:

```
tPlot.XLimits = [-5000 5000];
tPlot.YLimits = [-5000 5000];
tPlot.ZLimits = [-1000 0];

covPlotter = coveragePlotter(tPlot,'DisplayName','Sensor Coverage');
plotCoverage(covPlotter, coverageConfig(scenario));
```
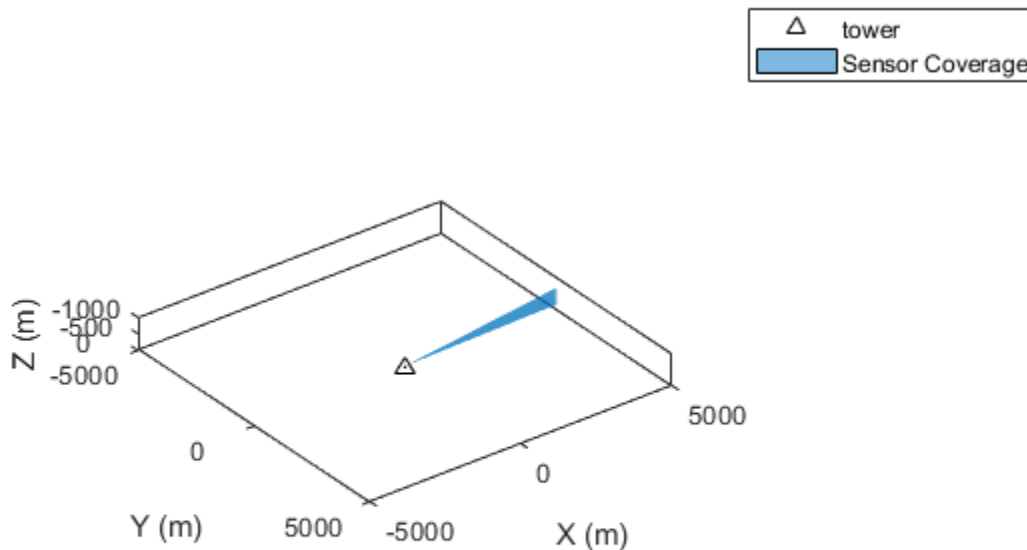

Platform Signatures

You can add other platforms in the scenario and adjust parameters that affect how other sensors observe the platforms. You can use an `rcsSignature` to model what the radar mounted on the tower would see.

The following code creates a helicopter and sets its radar cross section omnidirectionally to a value of 40 dBsm.

```
helicopter = platform(scenario);
helicopter.Dimensions = struct( ...
    'Length',30, ...
    'Width', .1, ...
    'Height', 7, ...
    'OriginOffset',[0 8 -3.2]);

helicopter.Signatures = rcsSignature( ...
    'Pattern',[40 40; 40 40], ...
    'Elevation',[-90; 90], ...
    'Azimuth',[-180,180]);
```

You can mount more than one sensor to any platform by placing the sensors in a cell array before assigning to the Sensors property.

```
helicopter.Sensors = { ...
    radarDataGenerator('SensorIndex', 2, ...
                        'UpdateRate', 20, ...
                        'MountingLocation', [22 0 0], ...
                        'MountingAngles', [-5 0 0], ...
                        'ScanMode', 'No scanning', ...
                        'HasINS', true, ...
                        'TargetReportFormat', 'Detections', ...
                        'DetectionCoordinates', 'Scenario'), ...
    radarDataGenerator('SensorIndex', 3, ...
                        'UpdateRate', 30, ...
                        'MountingLocation', [22 0 0], ...
                        'MountingAngles', [5 0 0], ...
                        'ScanMode', 'No scanning', ...
                        'HasINS', true, ...
                        'TargetReportFormat', 'Detections', ...
                        'DetectionCoordinates', 'Scenario')};
```

Platform Motion and Animation

You can arrange for the helicopter to cross the path of the radar beam. This shows how to make the helicopter follow a straight 100-meter path at a constant velocity with an elevation of 250 meters for seven seconds:

```
helicopter.Trajectory = waypointTrajectory([2000 50 -250; 2000 -50 -250],[0 7]);
```

Platform motion across time is performed by using a while-loop and calling the scenario's advance method.

You can plot all the platforms positions, orientations and dimensions in the loop:

```
profiles = platformProfiles(scenario);
dimensions = vertcat(profiles.Dimensions);

while advance(scenario)
    poses = platformPoses(scenario);
    positions = vertcat(poses.Position);
    orientations = vertcat(poses.Orientation);
    plotPlatform(pPlotter, positions, dimensions, orientations);
    plotCoverage(covPlotter, coverageConfig(scenario));
    % to animate more slowly uncomment the following line
    % pause(0.01)
end
```
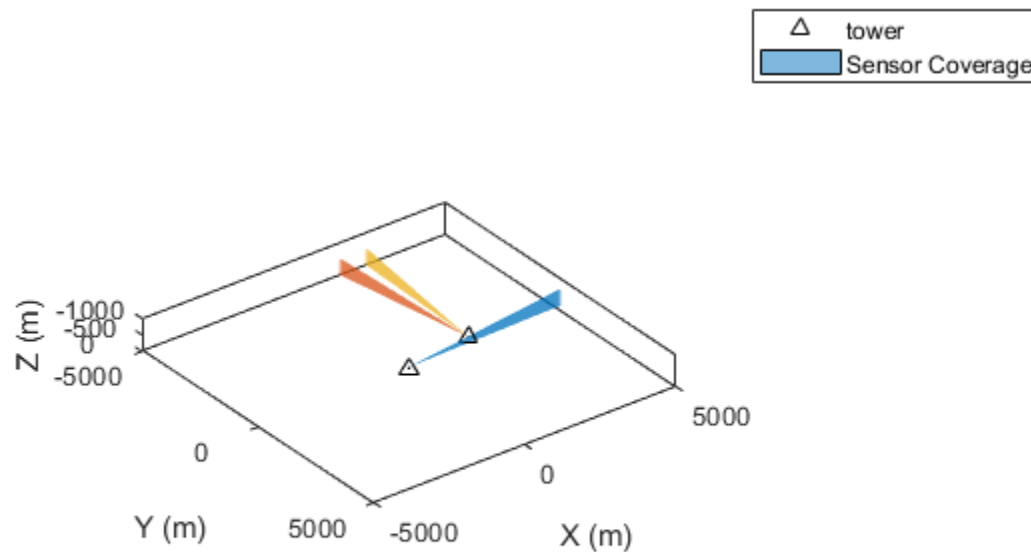
Detecting platforms

In the example above, you added three radars with different update rates: the tower has an update rate of 10 Hz, and the helicopter had two radars with update rates of 20 Hz and 30 Hz, respectively.

The scenario can be placed into a mode in which the call to `advance` updates the time of simulation as needed to update each of the sensors it contains. You can achieve this by setting the `UpdateRate` of the scenario to zero.

```
scenario.UpdateRate = 0;
```

To show the simulation time, add a UI control to the figure.

```
fig = ancestor(tPlot.Parent,'Figure');
timeReadout = uicontrol(fig,'Style','text','HorizontalAlignment','left','Position',[0 0 200 20])
timeReadout.String = "SimulationTime: " + scenario.SimulationTime;
```
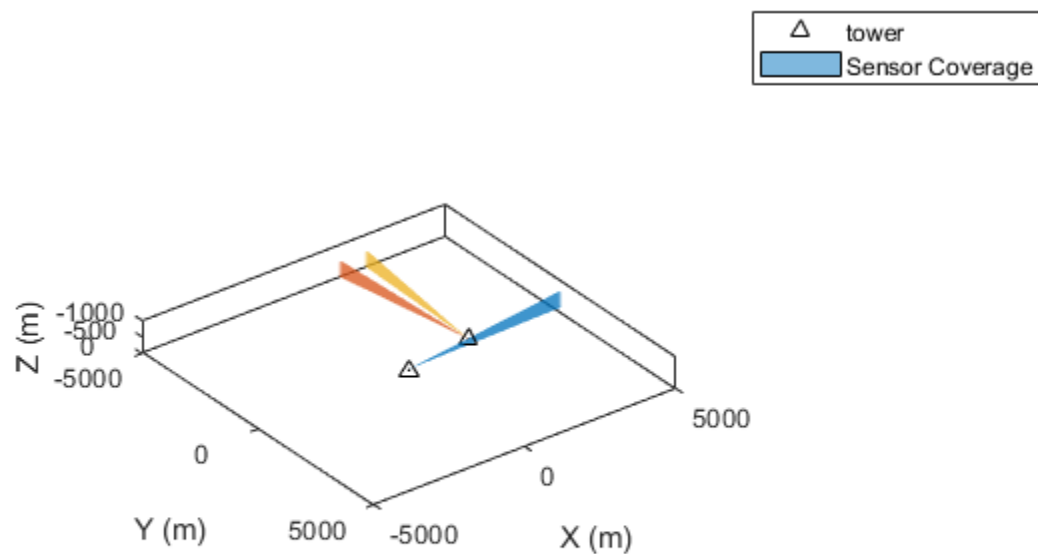
SimulationTime: 7.1

Now the proper sensor times can be reached. You can use `detect` to get the detections available by each sensor within the loop. Detections can be shown by constructing a `detectionPlotter` object.

```
dPlotter = detectionPlotter(tPlot,'DisplayName','detections');
```
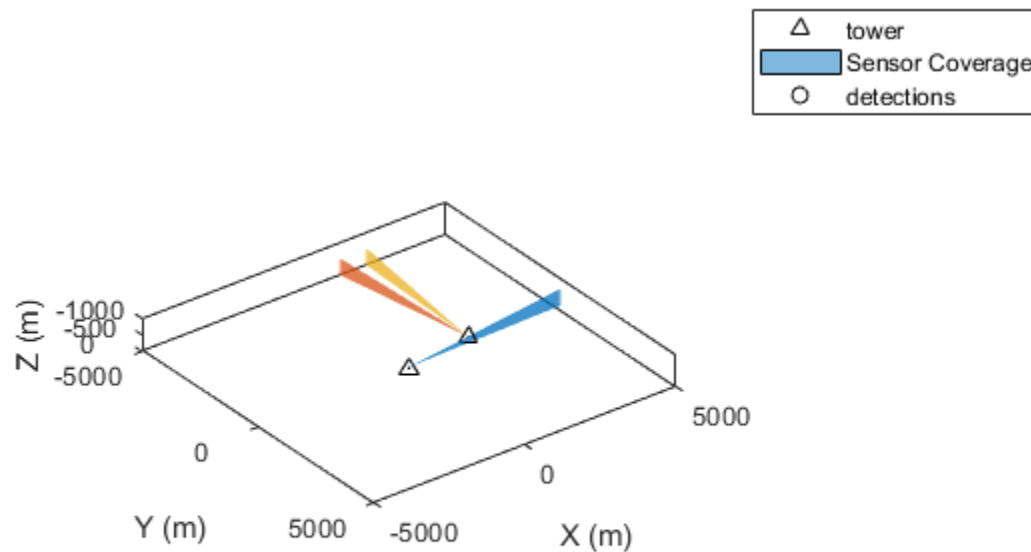
SimulationTime: 7.1

You can run the same simulation again by restarting it and modifying it to report detections.

The detected positions for a `radarDataGenerator` can be extracted from the detection `Measurement` field:

```
restart(scenario);

while advance(scenario)
    timeReadout.String = "SimulationTime: " + scenario.SimulationTime;
    detections = detect(scenario);

    % extract column vector of measurement positions
    allDetections = [detections{:}];
    if ~isempty(allDetections)
        measurement = cat(2,allDetections.Measurement)';
    else
        measurement = zeros(0,3);
    end

    plotDetection(dPlotter, measurement);

    poses = platformPoses(scenario);
    positions = vertcat(poses.Position);
    orientations = vertcat(poses.Orientation);
    plotPlatform(pPlotter, positions, dimensions, orientations);
    plotCoverage(covPlotter, coverageConfig(scenario));
end
```
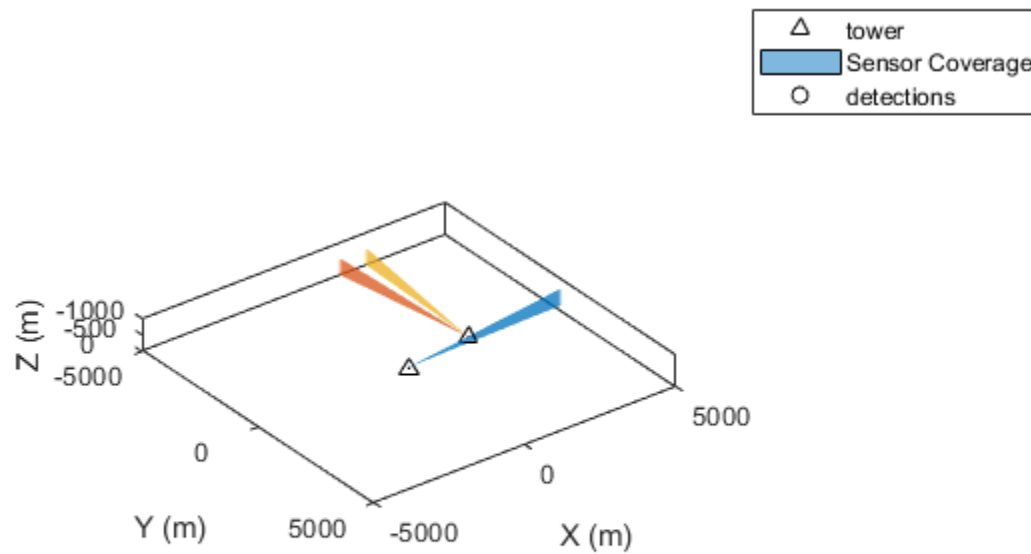
SimulationTime: 7

Notice that the update time of simulation increments non-uniformly to the times required by each of the sensors.

Summary

In this example, you learned how to construct and visualize a simple scenario and obtain detections generated by a radar data generator.

Use theaterPlot to Visualize Radar Scenario

This example shows how to use the `theaterPlot` object to visualize various aspects of a radar scenario.

Introduction

`theaterPlot` is an efficient tool for visualizing various aspects of a radar scenario. It is composed of a primary object, which hosts the plotting environment based on a parent axes, and plotters to plot the desired aspects of features from the radar scenario.

This figure shows a structural representation of a `theaterPlot` object.
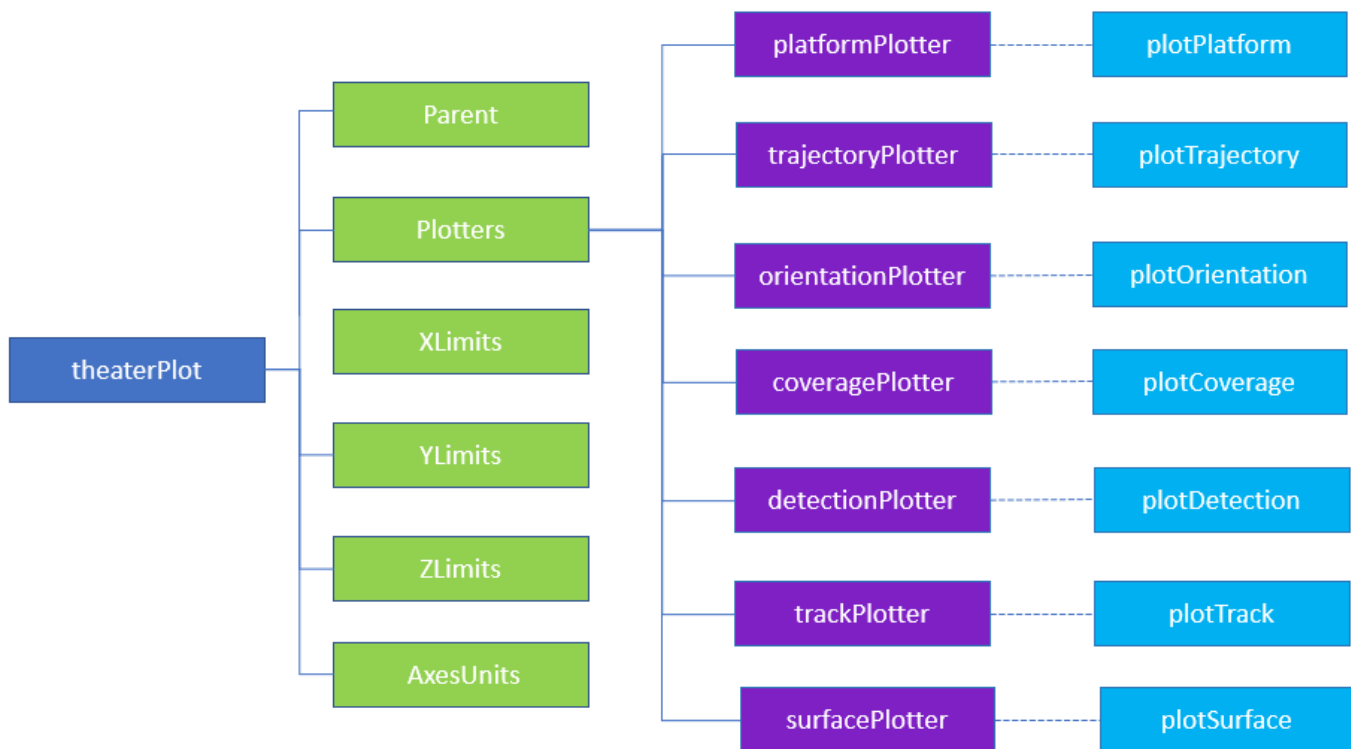
The `Parent` property specifies the axes on which the theater plot is enabled. You can specify the parent axes of a theater plot during object creation. If you do not specify a parent axes, `theaterPlot` creates a new figure and uses the current axes of the created figure as its `Parent` property. You can also set the axes limits of the parent axes using the `XLimits`, `YLimits`, and `Zlimits` properties by using name-value arguments during object creation. Set the units of measurement for each axes using the `AxesUnits` property.

The `Plotters` property holds plotters that you added to the `theaterPlot` object.

- `platformPlotter` — Plot platforms in a radar scenario
- `trajectoryPlotter` — Plot trajectories in a radar scenario
- `orientationPlotter` — Plot orientation of platforms in a radar scenario

- coveragePlotter — Plot sensor coverage and sensor beams in a radar scenario
- detectionPlotter — Plot sensor detections in a radar scenario
- trackPlotter — Plot tracks in a radar scenario
- surfacePlotter — Plot surfaces in a radar scenario

You can specify visual elements and effects for each plotter during the creation of the plotter. Each plotter is also paired with a theaterPlot object function, which you need to call to plot the results. For example, a coveragePlotter is paired with a plotCoverage object function that shows the sensor coverage.

This example showcases a few plotters for visualizing a radar scenario. theaterPlot can work efficiently with a radarScenario object even though you do not necessarily need a radarScenario object to use the theaterPlot object.

Create theaterPlot and radarScenario Objects

Create a radarScenario object and a theaterPlot object.

```
simulationDuration = 100;
scene = radarScenario('StopTime',simulationDuration);
tp = theaterPlot('XLimits',[-250 250],'YLimits',[-250 250],'ZLimits',[0 120]);
view(3);grid on;
```

Create Trajectory Plotter and Platform Plotter for Target

Create a waypoint trajectory for a target platform.

```
timeOfArrival = [0  simulationDuration];
waypoints = [100 -100 10; 100 100 80];
trajectory = waypointTrajectory(waypoints,timeOfArrival);
```

Add a cuboid target platform that follows the specified trajectory. First add a target platform to the radar scenario.

```
target = platform(scene,'Trajectory',trajectory,'Dimensions', ...
    struct('Length',35,'Width',15,'Height',5.5,'OriginOffset',[0 0 0]));
```

Then add a trajectoryPlotter object to the theaterPlot object, and use the plotTrajectory function to plot the waypoint trajectory.

```
trajPlotter = trajectoryPlotter(tp,'DisplayName','Trajectory','Color','k','LineWidth',1.2);
plotTrajectory(trajPlotter,{trajectory.Waypoints})
```

Tip You can plot multiple same-type features (platforms, trajectories, orientations, coverages, detections, or tracks) together using one plotter. For example, you can plot multiple trajectories together by specifying a cell array of waypoints as the second argument of the plotTrajectory function. See the syntax description of plotTrajectory for more details.

Define a plotter for the target platform.

```
targetPlotter = platformPlotter(tp,'DisplayName','Target', ...
    'Marker','s','MarkerEdgeColor','g','MarkerSize',2);
plotPlatform(targetPlotter,target.Position, ...
    target.Dimensions,quaternion(target.Orientation,'rotvecd'))
```

You can add graphical objects other than the plotter objects on the `theaterPlot` by directly plotting on the parent axes of the `theaterPlot` object. Put a circle marker at the origin.

```
hold on
plot3(tp.Parent,0,0,0,'Color','k','Marker','o','MarkerSize',4)
```

Create Platform with Mounted Radar Sensor

Add a tower platform to the scenario.

```
tower = platform(scene,'Position',[-100,0,0],'Dimensions', ...
    struct('Length',5,'Width',5,'Height',30,'OriginOffset',[0 0 -15]));
```

Display the tower using a platform plotter.

```
towerPlotter = platformPlotter(tp,'DisplayName','Tower','Marker','s','MarkerSize',2);
plotPlatform(towerPlotter,tower.Position,tower.Dimensions,quaternion(tower.Orientation,'rotvecd'
```

Mount a monostatic radar to the top of the tower.

```
radar = radarDataGenerator(1,'DetectionMode','Monostatic', ...
    'UpdateRate',5, ...
    'MountingLocation',[0, 0, 30], ...
    'FieldOfView',[4, 30],...
    'MechanicalAzimuthLimits',[-60 60], ...
    'MechanicalElevationLimits',[0 0], ...
    'HasElevation',true, ...
    'RangeResolution',200, ...
    'AzimuthResolution',20, ...
    'ElevationResolution',20);
tower.Sensors = radar;
```

Add a `coveragePlotter` and plot the coverage and initial beam for the monostatic radar. When plotting the coverage, the `plotCoverage` object function requires a second argument that specifies the configuration of the sensor coverage. Obtain the configuration by using the `coverageConfig` function on the radar scenario `scene`.

```
radarPlotter = coveragePlotter(tp,'Color','b','DisplayName','radar beam');
plotCoverage(radarPlotter,coverageConfig(scene))
```

Create a detection plotter to plot the detections that the radar generates.

```
detPlotter = detectionPlotter(tp,'DisplayName','Detection','MarkerFaceColor','r','MarkerSize',4)
```

Run Scenario and Update Theater Plot

Iterate through the radar scenario and generate radar detections. Plot the platform, radar coverage, and detections.

```
rng(2019) % for repeatable results
while advance(scene)
    % Plot target.
    plotPlatform(targetPlotter,target.Position, ...
        target.Dimensions,quaternion(target.Orientation,'rotvecd'))

    % Plot sensor coverage.
    plotCoverage(radarPlotter,coverageConfig(scene))
```

```
% Extract target pose from the view of the tower and use the extracted
% pose to generate detections.
poseInTower = targetPoses(tower);
[detections, numDets] = radar(poseInTower,scene.SimulationTime);
detPos = zeros(numDets,3);
detNoise = zeros(3,3,numDets);

% Obtain detection pose relative to the scenario frame. Also, obtain
% the covariance of the detection.
for i=1:numDets
    a = detections;
    detPos(i,:) = tower.Trajectory.Position + detections{i}.Measurement';
    detNoise(:,:,i) = detections{i}.MeasurementNoise;
end

% Plot any generated detections with the covariance ellipses.
if ~isempty(detPos)
    plotDetection(detPlotter,detPos,detNoise)
end
end
```
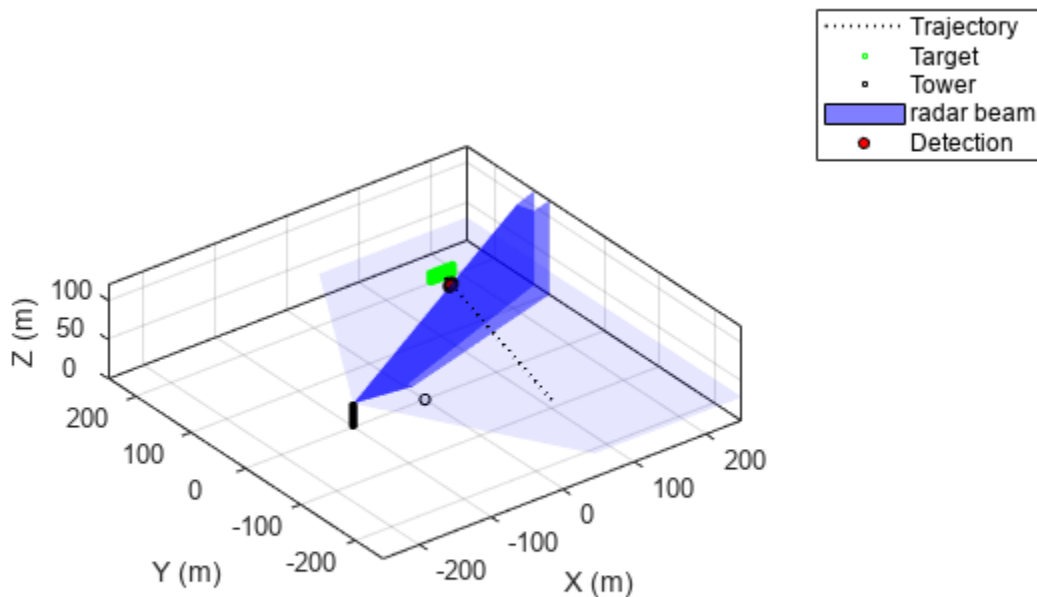

You can zoom in on the detection in the figure to visualize the plotted covariance ellipses of the generated detections.

Summary

In this example, you learned about the organization of a `theaterPlot` object. You also learned how to visualize a simple radar scenario using the `theaterPlot` object.

Simulate a Scanning Radar

This example shows how to simulate detection and tracking with a scanning monostatic radar. This example shows you how to configure a statistical radar model with mechanical and electronic scanning capabilities, how to set up a scenario management tool to handle platform dynamics and timing, and how to inspect the detections and tracks that are generated.

Introduction

Statistical Radar Model

Statistical radar models such as `radarDataGenerator` offer valuable data for early-stage development of radar systems. The system is modeled from the point of view of its performance characteristics, and a minimal set of parameters is used along with established theory to determine how this system will interact with its environment. This includes specification of fundamental system parameters such as operating frequency, bandwidth, and pulse repetition frequency (PRF), along with important metrics like angle and range resolution, false alarm rate, and detection probability. The radar model has dedicated modes for monostatic, bistatic, and passive operation. The primary output of this model is raw detection data or a sequence of track updates. While time-domain signals are not generated, an emitter can specify a numeric ID for the waveform it is using so that appropriate signal gain and rejection can be considered. Detections may be output in the radar frame, or in the scenario frame if the receiving radar has knowledge of the location and orientation of the transmitter. Tracks are generated using one of a variety of tracking and track management algorithms. Data rate is handled by a system update rate specification, so that detections are generated at the appropriate rate.

Scenario Management

The model works in tandem with a `radarScenario` to generate detection or track data over time in a dynamic environment. This scenario object not only manages the simulation time, but can handle the passing of data structures between objects in the scene as required when a frame of detection or track update is requested. The radar object is mounted on a platform which handles dynamics, such as position, velocity, and orientation in the scenario frame. These scenario platforms may also contain signature information, which describes how the platform appears to various types of sensors. For our purposes, the `rcsSignature` class will be used to endow target platforms with a radar cross section.

Scanning

Properties to control scanning can be found with the radar object definition. Scanning can be mechanical or electronic. With mechanical scanning, the pointing direction cannot change instantaneously, but is constrained by a maximum rotation rate in azimuth and elevation. With electronic scanning there is no such constraint, and the scan direction resets at the beginning of each loop. The radar is also allowed to perform mechanical and electronic scanning simultaneously.

This figure shows the difference in scan pattern between these two modes. With the electronic mode the scan angle is always increasing, and with the mechanical mode the scan position always changes to an adjacent position. The radar scans in azimuth first because that is the *primary scan direction*, which is the direction with the greatest extent of the scan limit.
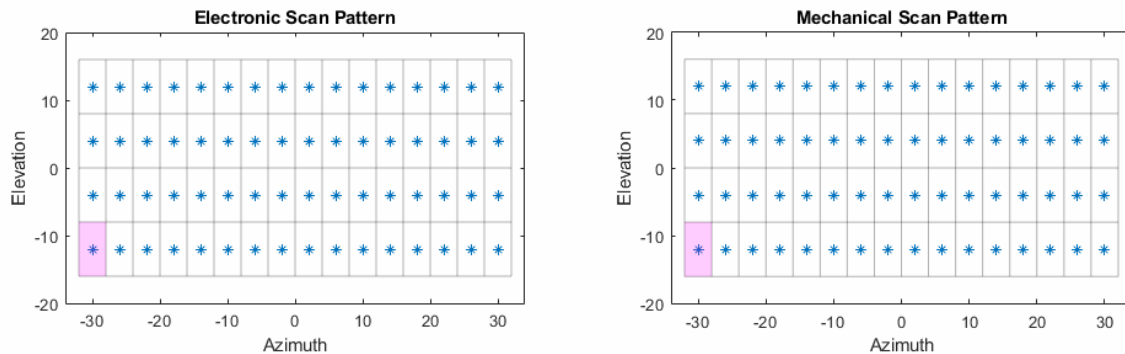
Scenario: Air Surveillance

For this scenario, consider a scanning radar system with its antenna mounted on a tower, as might be used for the tracking of aircraft. You will simulate a couple of inbound platforms with different RCS profiles, and inspect detection and track data. You can see the effect of SNR and range on the ability to make detections.

Radar System

Use a 1 GHz center frequency and a bandwidth of 1.5 MHz, to yield a range resolution of about 100 meters. To model a single pulse per scan position, set the update rate to the desired PRF. Set the size of the range ambiguity to reflect the PRF choice, and set the upper range limit to twice the size of the range ambiguity to enable detections of objects in the first and second range ambiguity.

```
% Set the seed for repeatability
rng('default');

% System parameters
c = physconst('lightspeed');
fc = 1e9; % Hz
bandwidth = 1.5e6; % Hz
prf = 4e3; % Hz
updateRate = prf;
rangeRes = c/(2*bandwidth); % m
rangeAmb = c/(2*prf); % m
rangeLims = [0 2*rangeAmb];
```

Besides choosing between mechanical and electronic scan modes, the scanning behavior is controlled by the specification of the scan limits and field of view (FoV) in azimuth and elevation. The FoV consists of the azimuth and elevation extent of the observable region at each scan position, similar to a two-sided half-power beamwidth specification. The scan points are chosen so that the total angular extent defined by the scan limit is covered by nonoverlapping segments with size equal to the FoV. Scan limits are specified in the radar's frame, which may be rotated from its platform's frame via the `MountingAngles` property.

Scan +/-20 degrees in azimuth and +/-10 degrees in elevation. Set the FoV to 4 degrees azimuth and 8 degrees elevation. Finally, specify the total number of complete scans to simulate, which will dictate the total simulation time.

```
% Scanning parameters
azLimits = [-20 20];
elLimits = [-10 10];
fov = [4;8]; % deg
numFullScans = 24; % number of full scans to simulate
```

The total number of scan points can be found as follows. The scan limits denote the minimum and maximum scan angles referenced to the antenna boresight vector, so the total area scanned in one direction is greater than the extent of the scan limits by up to half the FoV in that direction. The pattern is always a rectangular grid over az/el.

```
numScanPointsAz = floor(diff(azLimits)/fov(1)) + 1;
numScanPointsEl = floor(diff(elLimits)/fov(2)) + 1;
numScanPoints = numScanPointsAz*numScanPointsEl;
```

A specification of angular resolution separate from FoV is used, which allows for the modeling of angle estimation algorithms such as monopulse. This angular resolution determines the ability to discriminate between targets, and is used to derive the actual angle accuracy from the Cramer-Rao lower bound (CRLB) for monopulse angle estimation with that resolution. Similarly, range resolution determines the minimum spacing in range required to discriminate two targets, but the actual range measurement accuracy comes from the CRLB for range estimation. Exact measurables (range, range-rate, angle, etc.) can be used by turning measurement noise off with the `HasNoise` property. Typically, measurement error does not go to zero as SNR increases without bound. This can be captured by the bias properties (RangeBiasFraction, etc.), of which there is one for each type of measurable. For our purposes we will leave these at their defaults.

Use an angular resolution equal to 1/4th of the FoV in each direction, which allows discriminating between up to 16 point targets in the FoV at the same range.

```
angRes = fov/4;
```

Reference Target and Radar Loop Gain

Instead of directly specifying things like transmit power, noise power, and specifics of the detection algorithm used, this radar model uses the concept of *radar loop gain* to translate target RCS and range into an SNR, which is translated directly into a detection probability. Radar loop gain is an important property of any radar system, and is computed with a *reference target,* which consists of a reference range and RCS, and a receiver operating characteristic (ROC) for given detection and false alarm probabilities.

Use a detection probability of 90% for a reference target at 20 km and 0 dBsm. Use the default false alarm rate of 1e-6 false alarms per resolution cell per update.

```
refTgtRange = 20e3;
refTgtRCS = 0;
detProb = 0.9;
faRate = 1e-6;
```

Constructing the Radar

Construct the radar using the parameters above. Use the lower scan limit in elevation to set the pitch mounting angle of the radar from its platform. This points the boresight vector up so that the lower edge of the scanning area is parallel to the ground (that is, none of the scan points point the radar towards the ground).

```
pitch = elLimits(1); % mount the antenna rotated upwards
```

Enable elevation angle measurements, false alarm generation, and range ambiguities with the corresponding `Has...` property. To model a radar that only measures range and angle, set `HasRangeRate` to false. To get detections in scenario coordinates, the radar needs to be aware of the orientation of the transmitter. Because this example uses a monostatic radar, this can be achieved by enabling inertial navigation system (INS) functionality with the `HasINS` property.

```
% Create radar object
sensorIndex = 1; % a unique identifier is required
radar = radarDataGenerator(sensorIndex,'UpdateRate',updateRate,...
    'DetectionMode','Monostatic','ScanMode','Electronic','TargetReportFormat','Detections','Dete
    'HasElevation',true,'HasINS',true,'HasRangeRate',false,'HasRangeAmbiguities',true,'HasFalseA
    'CenterFrequency',fc,'Bandwidth',bandwidth,...
    'RangeResolution',rangeRes,'AzimuthResolution',angRes(1),'ElevationResolution',angRes(2),...
    'ReferenceRange',refTgtRange,'ReferenceRCS',refTgtRCS,'DetectionProbability',detProb,'FalseA
    'RangeLimits',rangeLims,'MaxUnambiguousRange',rangeAmb,'ElectronicAzimuthLimits',azLimits,'E
    'FieldOfView',fov,'MountingAngles',[0 pitch 0]);
```

Use the helper class to visualize the scan pattern, and animate it in the simulation loop.

```
scanplt = helperScanPatternDisplay(radar);
scanplt.makeOverviewPlot;
```
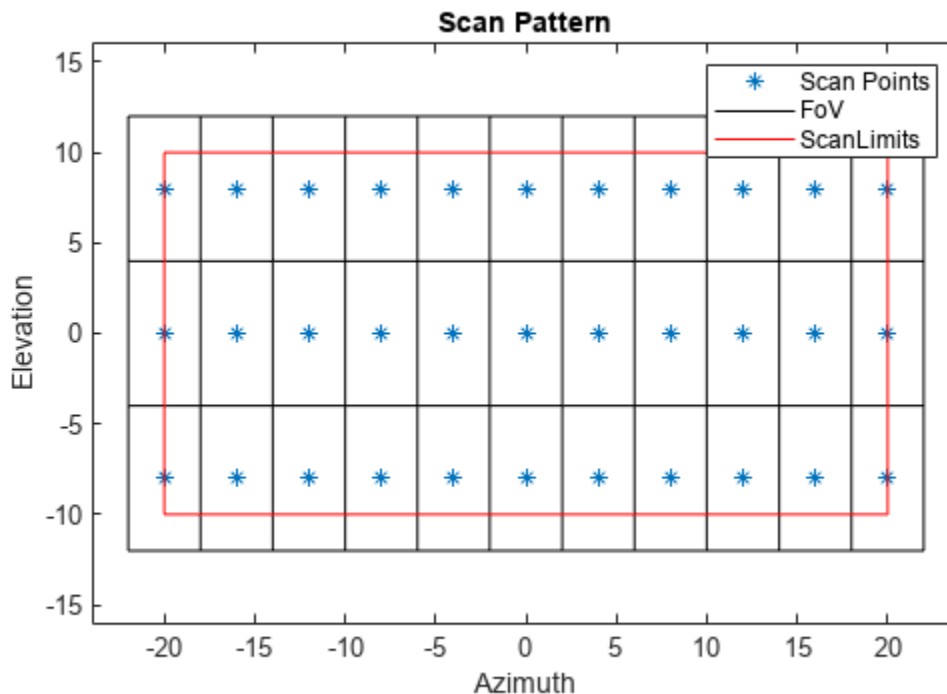

The scan points in azimuth extend exactly to the specified azimuth scan limits, while the elevation scan points are pulled in slightly as needed to avoid overlapping or redundant scan points.

Calculate the total number of range-angle resolution cells for the system, and the total number of frames to be used for detection. Then compute the expected number of false alarms by multiplying the false alarm rate by the total number of resolution cells interrogated throughout the simulation.

```
numResolutionCells = diff(rangeLims)*prod(fov)/(rangeRes*prod(angRes));
numFrames = numFullScans*numScanPoints; % total number of frames for detection
expNumFAs = faRate*numFrames*numResolutionCells % expected number of FAs
```

```
expNumFAs = 9.5040
```

Scenario and Target

Use the `radarScenario` object to manage the top-level simulation flow. This object provides functionality to add new platforms and quickly generate detections from all sensor objects in the scenario. It also manages the flow of time in the simulation. Use the total number of frames calculated in the previous step to find the required `StopTime`. To adaptively update simulation time based on the update rate of objects in the scene, set the scenario update rate to zero.

```
% Create scenario
stopTime = numFullScans*numScanPoints/prf;
scene = radarScenario('StopTime',stopTime,'UpdateRate',0);
```

Use the `platform` method to generate new platform objects, add them to the scene, and return a handle for accessing the platform properties. Each platform has a unique ID, which is assigned by the scenario upon construction. For a static platform, only the `Position` property needs to be set. To simulate a standard constant-velocity kinematic trajectory, use the `kinematicTrajectory` object and specify the position and velocity.

Create two target platforms at different ranges, both inbound at different speeds and angles.

```
% Create platforms
rdrPlat = platform(scene,'Position',[0 0 12]); % place 12 m above the origin
tgtPlat(1) = platform(scene,'Trajectory',kinematicTrajectory('Position',[38e3 6e3 10e3],'Velocity
tgtPlat(2) = platform(scene,'Trajectory',kinematicTrajectory('Position',[25e3 -3e3 1e3],'Velocity
```

To enable convenience detection methods and automatic advancement of simulation time, the scenario needs to be aware of the radar object constructed earlier. To do this, mount the radar to the platform by populating the `Sensors` property of the platform. To mount more than one sensor on a platform, the `Sensors` property may be a cell array of sensor objects.

```
% Mount radar to platform
rdrPlat.Sensors = radar;
```

The `rcsSignature` class can be used to specify platform RCS as a function of aspect angle. For a simple constant-RCS signature, set the `Pattern` property to a scalar value. Let one platform be 20 dBsm and the other 4 dBsm to demonstrate the difference in detectability.

```
% Set target platform RCS profiles
tgtRCS = [20, 4]; % dBsm
tgtPlat(1).Signatures = rcsSignature('Pattern',tgtRCS(1));
tgtPlat(2).Signatures = rcsSignature('Pattern',tgtRCS(2));
```

Detectability

Calculate the theoretical SNR that can be expected for out targets. Because a simple kinematic trajectory is used, the truth position and range is calculated as such.

```
time = (0:numFrames-1).'/updateRate;
tgtPosTruth(:,:,1) = tgtPlat(1).Position + time*tgtPlat(1).Trajectory.Velocity;
tgtPosTruth(:,:,2) = tgtPlat(2).Position + time*tgtPlat(2).Trajectory.Velocity;
tgtRangeTruth(:,1) = sqrt(sum((tgtPosTruth(:,:,1)-rdrPlat.Position).^2,2));
tgtRangeTruth(:,2) = sqrt(sum((tgtPosTruth(:,:,2)-rdrPlat.Position).^2,2));
```

Then use the known target RCS and the radar loop gain computed by the radar model to get the truth SNR for each target.

```
tgtSnr = tgtRCS - 40*log10(tgtRangeTruth) + radar.RadarLoopGain;
```

For a magnitude-only detection scheme, such as CFAR, with the specified probability of detection and false alarm rate, the minimum SNR required for detection is given by

```
minSnr = 20*log10(erfcinv(2*radar.FalseAlarmRate) - erfcinv(2*radar.DetectionProbability)); % dB
```

Compute the minimum range for detection.

```
Rd = 10.^((tgtRCS - minSnr + radar.RadarLoopGain)/40); % meters
```

Compare the mean range and SNR of the targets to the minimum range and SNR thresholds. The total travel distance of the targets is small enough that the range-based SNR does not change significantly.

```
snrMargin = mean(tgtSnr,1) - minSnr
```

snrMargin = *1×2*

```
    8.6781    0.5978
```

```
rangeMargin = (Rd - mean(tgtRangeTruth,1))/1e3 % km
```

rangeMargin = *1×2*

```
   25.7310    0.8812
```

The first target is about 8 dB brighter than needed for detection with the given CFAR parameters, while the second target is considered barely detectable.

Simulate Detections

The primary simulation loop starts by calling `advance` on the scenario object. This method steps the scenario forward to the next time at which an object in the scene requires an update, and returns false when the specified stop time is reached, exiting the loop. Note that the first call to `advance` does not step the simulation time, so that the first frame of data collection may occur at 0 s. As an alternative to using `advance`, `scene.SimulationStatus` can be checked to determine if the simulation has run to completion.

Inside the loop, the detection and track generation methods can be called for any sensor object individually (see the sensor `step` method), but convenience methods exist to generate detections and tracks for all sensors and from all targets in the scene with one function call. Because we only have one sensor, `detect(scene)` is a good solution to get a list of detections for all targets in the scene. By initiating the generation of detections at the top level like this, the scene itself handles the passing of required timing, INS, and other configuration data to the sensor. The visualization helper class is used here as well to animate the scan pattern.

```matlab
allDets = []; % initialize list of all detections generated
lookAngs = zeros(2,numFrames);
simTime = zeros(1,numFrames);
frame = 0; % current frame
while advance(scene) % advance until StopTime is reached

    % Increment frame index and record simulation time
    frame = frame + 1;
    simTime(frame) = scene.SimulationTime;

    % Generate new detections
    dets = detect(scene);

    % Record look angle for inspection
    lookAngs(:,frame) = radar.LookAngle;

    % Update coverage plot
    scanplt.updatePlot(radar.LookAngle);

    % Compile any new detections
    allDets = [allDets;dets];

end
```
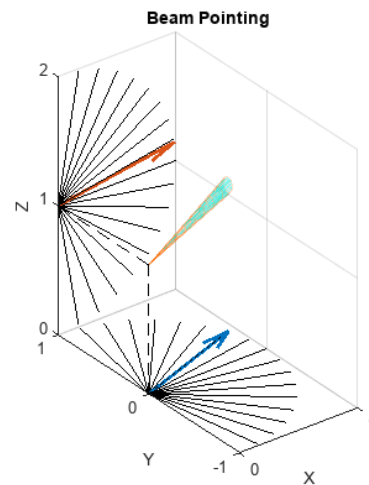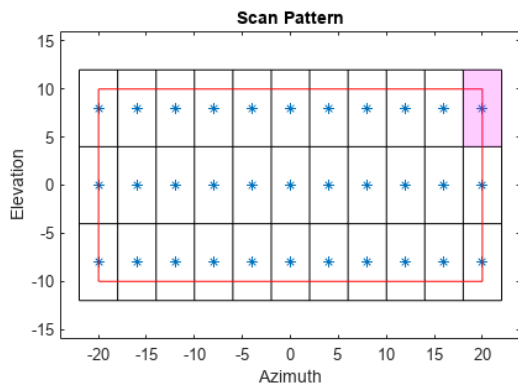

Scan Behavior

Plot the recorded look angles for the first two complete scans to inspect the pattern. The radar scans first in azimuth, and steps up in elevation at the end of each azimuth scan, instantaneously resetting the look angle at the start of each new scan.

```matlab
figure;
plot(simTime(1:2*numScanPoints)*1e3,lookAngs(:,1:2*numScanPoints));
xlabel('Time (ms)');
ylabel('Look Angle (deg)');
legend('Azimuth','Elevation');
title('Look Angles');
```
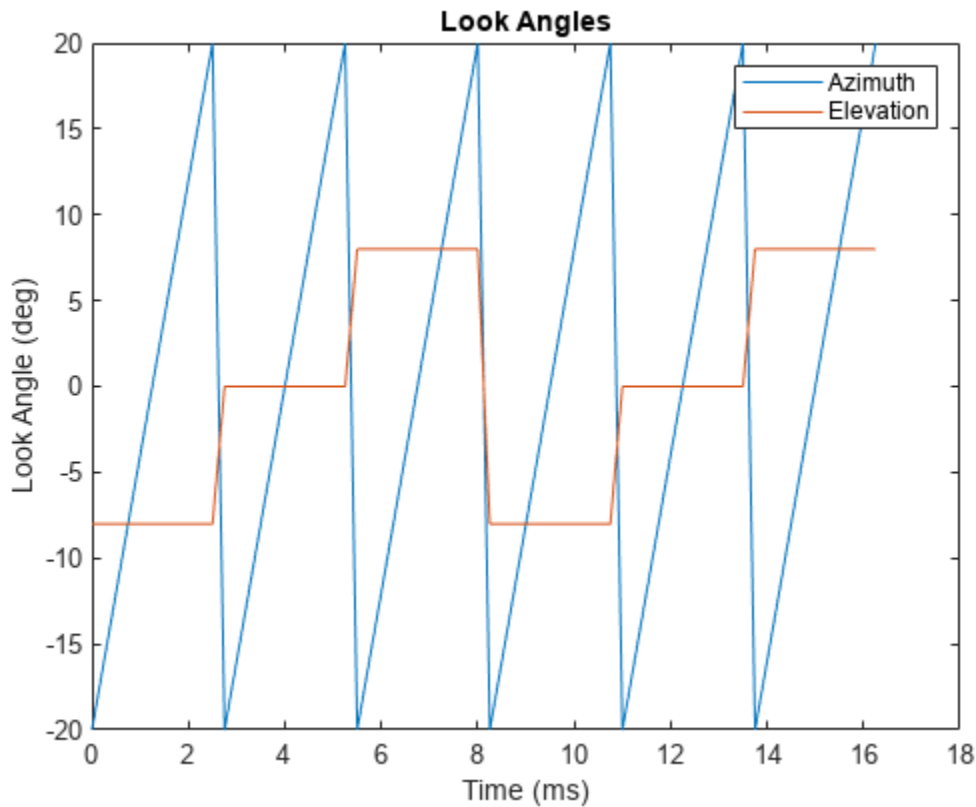
Detections

Inspect the contents of an `objectDetection` as output by our radar.

`allDets{1}`

```
ans =
  objectDetection with properties:

                   Time: 1.0000e-03
            Measurement: [3×1 double]
       MeasurementNoise: [3×3 double]
            SensorIndex: 1
          ObjectClassID: 0
    ObjectClassParameters: []
    MeasurementParameters: [1×1 struct]
        ObjectAttributes: {[1×1 struct]}
```

`Time` is the simulation time at which the radar generated the detection. `SensorIndex` shows which sensor this detection was generated by (which is important when there are multiple sensors and you are using the scene-level detect method). `Measurement` is the measurable value associated with the detection. The format depends on the choice of the detection mode and output coordinates. `MeasurementNoise` gives the variance or covariance of the measurement and is used by tracker models. When outputting detections in scenario coordinates, the measurement field is simply the estimated position vector of the target, and the measurement noise gives the covariance of that position estimate.

The `ObjectAttributes` field contains some useful metadata. Inspect the contents of the metadata for the first detection.

```
allDets{1}.ObjectAttributes{1}
```

ans = struct with fields:
*   TargetIndex: 3*
*   SNR: 13.1821*

`TargetIndex` shows the index of the platform that generated the detection, which is the unique identifier assigned by the scenario in the order the platforms are constructed. The SNR of this detection is about what we expected from the corresponding target platform. When the detection is a false alarm, `TargetIndex` will be -1, which is an invalid platform ID.

Collect the target index, SNR, and time of each detection into vectors.

```
detTgtIdx = cellfun(@(t) t.ObjectAttributes{1}.TargetIndex, allDets);
detTgtSnr = cellfun(@(t) t.ObjectAttributes{1}.SNR, allDets);
detTime   = cellfun(@(t) t.Time, allDets);
firstTgtDet  = find(detTgtIdx == tgtPlat(1).PlatformID);
secondTgtDet = find(detTgtIdx == tgtPlat(2).PlatformID);
```

Find the total number of FA detections.

```
numFAs = numel(find(detTgtIdx==-1))
```

numFAs = 9

This is close to the expected number of FAs computed earlier.

Collect the measurement data across detections and plot along with truth positions for the two target platforms.

```
tgtPosObs = cell2mat(cellfun(@(t) t.Measurement,allDets,'UniformOutput',0).');
subplot(1,2,1);
helperPlotPositionError( tgtPosTruth(:,:,1),time,tgtPosObs(:,firstTgtDet),detTime(firstTgtDet),s
title('Target 1 Position Error');
subplot(1,2,2);
helperPlotPositionError( tgtPosTruth(:,:,2),time,tgtPosObs(:,secondTgtDet),detTime(secondTgtDet)
title('Target 2 Position Error');
set(gcf,'Position',get(gcf,'Position')+[0 0 600 0]);
```
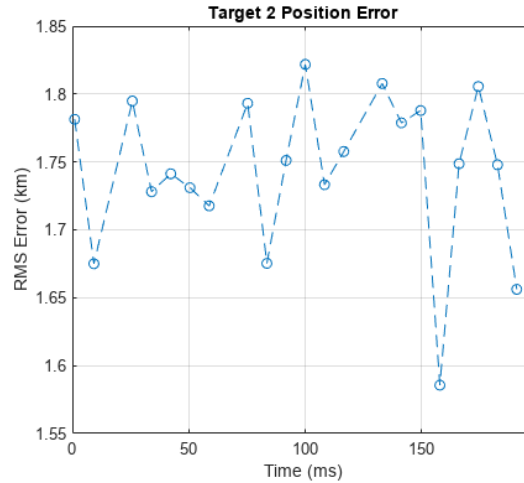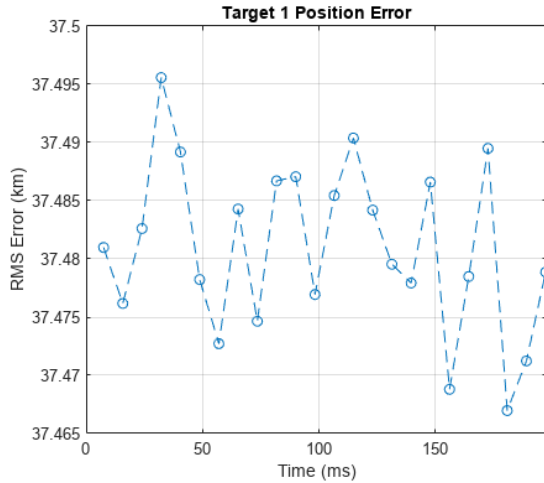
Most of the detections originate from the first target. However, the first target was not detected on every frame, and the second target has generated many detections despite having a low SNR. Though the first target is easily detected, the measurement error is large because the target is in the second range ambiguity and no disambiguation has been performed. The second target, when detected, shows position error that is consistent with 100 meter range resolution and relatively-poor angle resolution.

Look at the total variance of the detections against SNR. The total variance is the sum of the marginal variances in each Cartesian direction (X, Y, and Z). This variance includes the effects of estimation in range-angle space along with the transformation of those statistics to scenario coordinates.

```
detTotVar = cellfun(@(t) trace(t.MeasurementNoise),allDets);
figure;
subplot(1,2,1);
helperPlotPositionTotVar( detTgtSnr(firstTgtDet),detTotVar(firstTgtDet) );
title('First Target');
subplot(1,2,2);
helperPlotPositionTotVar( detTgtSnr(secondTgtDet),detTotVar(secondTgtDet) );
title('Second Target');
set(gcf,'Position',get(gcf,'Position')+[0 0 600 0]);
```
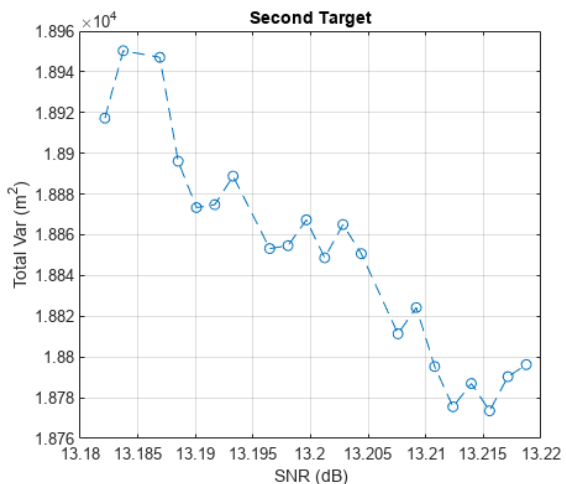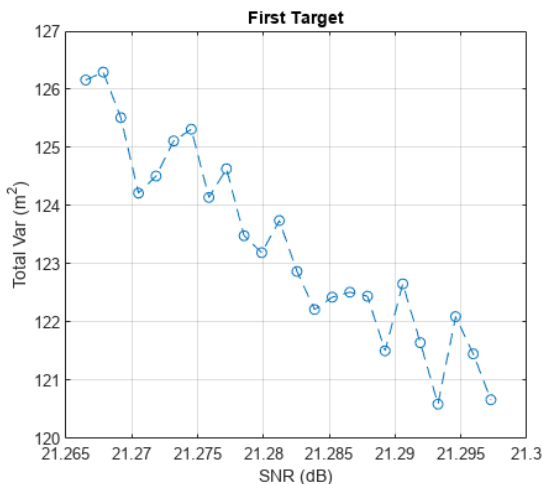
Although the absolute position error for the first target was greater due to the range ambiguity, the lower measurement variance reflects both the greater SNR of the first target and its apparently short range.

Simulate Tracks

The `radarDataGenerator` is also capable of performing open-loop tracking. Instead of outputting detections, it can output track updates.

Run the simulation again, but this time let configure the radar model to output tracks directly. Firstly, set some properties of the radar model to generate tracks in the scenario frame.

```
release(radar); % unlock for re-simulation
radar.TargetReportFormat = 'Tracks';
radar.TrackCoordinates = 'Scenario';
```

Tracking can be performed by a number of different algorithms. There are simple alpha-beta filters and Kalman filters of the linear, extended, and unscented variety, along with constant-velocity, constant-acceleration, and constant-turnrate motion models. Specify which algorithm to use with the `FilterInitializationFcn` property. This is a function handle or the name of a function in a character vector, for example `@initcvekf` or `'initcvekf'`. If using a function handle, it must take an initial detection as input and return an initialized tracker object. Any user-defined function with the same signature may be used. For this example, use the constant-velocity extended Kalman filter.

```
radar.FilterInitializationFcn = @initcvekf;
```

The overall structure of the simulation loop is the same, but in this case the radar's `step` function must be called manually. The required inputs are a target poses structure, acquired by calling `targetPoses` on the radar platform, the INS structure, acquired by calling `pose` on the radar platform, and the current simulation time.

```
restart(scene); % restart scenario

allTracks = []; % initialize list of all track data
while advance(scene)

    % Generate new track data
    tgtPoses = targetPoses(rdrPlat);
    ins = pose(rdrPlat);
    tracks = radar(tgtPoses,ins,scene.SimulationTime);

    % Compile any new track data
    allTracks = [allTracks;tracks];

end
```

Inspect a track object.

```
allTracks(1) % look at first track update

ans =
  objectTrack with properties:

                 TrackID: 1
                BranchID: 0
             SourceIndex: 1
              UpdateTime: 0.0093
```

```
                     Age: 34
                   State: [6×1 double]
          StateCovariance: [6×6 double]
          StateParameters: [1×1 struct]
             ObjectClassID: 0
   ObjectClassProbabilities: 1
               TrackLogic: 'History'
           TrackLogicState: [1 1 0 0 0]
               IsConfirmed: 1
                 IsCoasted: 0
            IsSelfReported: 1
          ObjectAttributes: [1×1 struct]
```

`TrackID` is the unique identifier of the track file, and `SourceIndex` is the ID of the tracker object reporting the track, which is useful when you have multiple trackers in the scene. When a multi-hypothesis tracker is used, `BranchID` gives the index of the hypothesis used. The `State` field contains the state information on this update. Because a constant-velocity model and scenario-frame track coordinates are used, `State` consists of the position and velocity vectors. `UpdateTime` gives the time at which this track update was generated. Another important property, `IsCoasted`, tells you if this track update used a fresh target detection to update the filter, or if it was simply propagated forward in time from the last target detection (the track is "coasted").

The ObjectAttributes structure gives the ID of the platform whose signature is being detected, and the SNR of that signature during the update.

`allTracks(1).ObjectAttributes`

ans = *struct with fields:*
    TargetIndex: 3
            SNR: 19.1837

Collect the target index and update time. Also check how many detections were used to update our tracks.

```
trackTgtIdx = arrayfun(@(t) t.TargetIndex,[allTracks.ObjectAttributes]);
updateTime = [allTracks.UpdateTime];
firstTgtTrack = find(trackTgtIdx == tgtPlat(1).PlatformID);
secondTgtTrack = find(trackTgtIdx == tgtPlat(2).PlatformID);
```

`numUpdates = numel(find(~[allTracks.IsCoasted]))` % total number of track updates with new detect

numUpdates = 33

This is roughly the number of non-FA detections generated in the first part.

Look at the reported positions in the XY plane and their covariances, along with truth position data. Because the first target is in the second range ambiguity and no disambiguation was performed, find the "truth" ambiguous position for the first target.

```
los = tgtPosTruth(:,:,1) - rdrPlat.Position;
R = sqrt(sum(los.^2,2));
los = los./R;
Ramb = mod(R,rangeAmb);
tgtPosTruthAmb = rdrPlat.Position + los.*Ramb;
```

Use the helper function to plot a visualization of the position track.

```
figure;
subplot(1,2,1);
helperPlotPositionTrack( tgtPosTruthAmb,allTracks(firstTgtTrack) );
title('First Target Track');
subplot(1,2,2);
helperPlotPositionTrack( tgtPosTruth(:,:,2),allTracks(secondTgtTrack) );
title('Second Target Track');
set(gcf,'Position',get(gcf,'Position')+[0 0 600 0]);
```
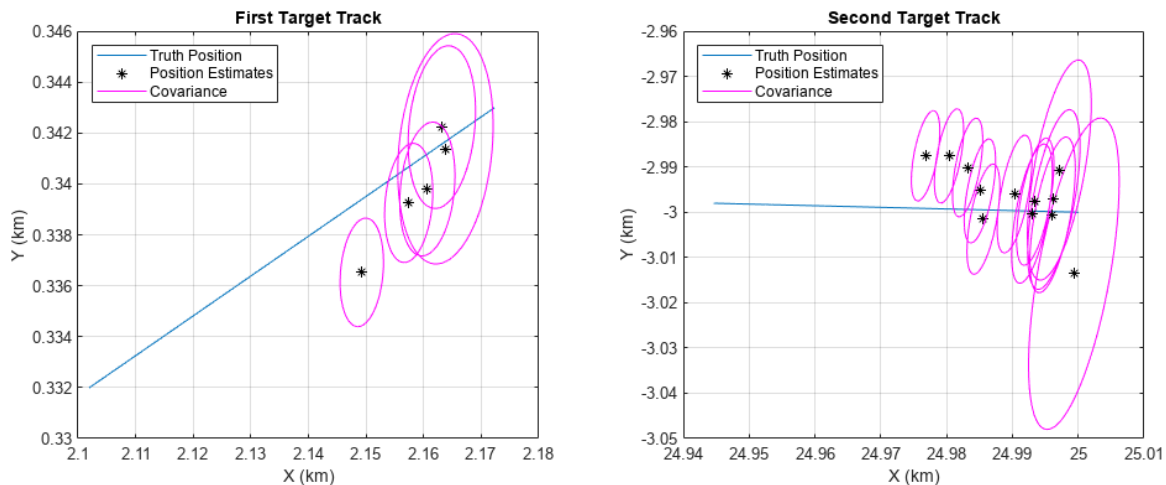

In both cases, the position estimates start by converging on the truth (or range-ambiguous truth) trajectory. For the first target we have more track updates and the final state appears closer to steady-state convergence. Looking at the covariance contours, notice the expected initial decrease in size as the tracking filter warms up.

To take a closer look at the tracking performance, plot the magnitude of the position estimation error for both targets, and indicate which samples were updated with target detections.

```
figure;
subplot(1,2,1);
helperPlotPositionTrackError( tgtPosTruthAmb,time,allTracks(firstTgtTrack),updateTime(firstTgtTra
title('First Target Ambiguous Position Error');
subplot(1,2,2);
helperPlotPositionTrackError( tgtPosTruth(:,:,2),time,allTracks(secondTgtTrack),updateTime(second
title('Second Target Position Error');
set(gcf,'Position',get(gcf,'Position')+[0 0 600 0]);
```
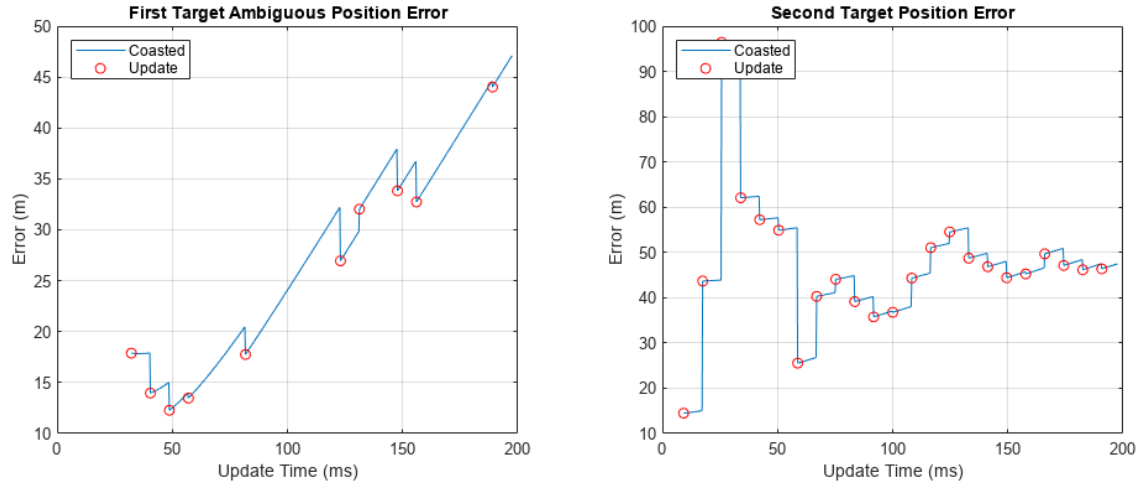
For the first target, which has a larger SNR, the position error decreases on samples that had a new detection to update the track file (indicated by a red circle). Since the second target had much lower SNR, the error in some individual detections was great enough to increase the position error of the track. Despite this, the second target track position error has an initial downward trend, and the estimated state will likely converge.

There is an issue with the first target track. Because it is in the second range ambiguity and has a component of motion perpendicular to the line of sight, the apparent velocity is changing, and the constant-velocity Kalman filter is unable to converge.

Conclusion

In this example you configured a statistical radar system for use in the simulation and analysis of target detectability and trackability. You saw how to use the scan limits and field of view properties to define a scan pattern, how to run the radar model with a scenario management tool, and how to inspect the generated detections and tracks.

Helper Functions

```
function helperPlotPositionError( tgtPosTruth,time,tgtPosObs,detTime,T )

tgtPosTruthX = interp1(time,tgtPosTruth(:,1),detTime).';
tgtPosTruthY = interp1(time,tgtPosTruth(:,2),detTime).';
tgtPosTruthZ = interp1(time,tgtPosTruth(:,3),detTime).';

err = sqrt((tgtPosTruthX - tgtPosObs(1,:)).^2+(tgtPosTruthY - tgtPosObs(2,:)).^2+(tgtPosTruthZ -

plot(detTime*1e3,err/1e3,'--o');
grid on;
xlim([0 T*1e3]);
ylabel('RMS Error (km)');
xlabel('Time (ms)');

end

function helperPlotPositionTotVar( detTgtSnr,detTotVar )

plot(detTgtSnr,detTotVar,'--o');
```

```matlab
    grid on;
    xlabel('SNR (dB)');
    ylabel('Total Var (m^2)');

end

function helperPlotPositionTrack( tgtPosTruth,tracks )

plot(tgtPosTruth(:,1)/1e3,tgtPosTruth(:,2)/1e3);
hold on;

updateIdx = find(~[tracks.IsCoasted]);
theta = 0:0.01:2*pi;

for ind = 1:2:numel(updateIdx)

    % plot update
    T = tracks(updateIdx(ind));
    plot(T.State(1)/1e3,T.State(3)/1e3,'*black');
    sigma = T.StateCovariance([1 3],[1 3]);
    [evec,eval] = eigs(sigma);
    mag = sqrt(diag(eval)).';
    u = evec(:,1)*mag(1);
    v = evec(:,2)*mag(2);
    x = cos(theta)*u(1) + sin(theta)*v(1);
    y = cos(theta)*u(2) + sin(theta)*v(2);
    plot(x/1e3 + T.State(1)/1e3,y/1e3 + T.State(3)/1e3,'magenta');

end

hold off;
grid on;
xlabel('X (km)');
ylabel('Y (km)');
legend('Truth Position','Position Estimates','Covariance','Location','northwest');

end

function helperPlotPositionTrackError( tgtPosTruth,time,tracks,updateTime )

state = [tracks.State];

sx = state(1,:);
sy = state(3,:);
sz = state(5,:);

x = interp1(time,tgtPosTruth(:,1),updateTime);
y = interp1(time,tgtPosTruth(:,2),updateTime);
z = interp1(time,tgtPosTruth(:,3),updateTime);

err = sqrt((x-sx).^2+(y-sy).^2+(z-sz).^2);

isUpdate = ~[tracks.IsCoasted];

plot(updateTime*1e3,err);
hold on;
plot(updateTime(isUpdate)*1e3,err(isUpdate),'or');
hold off;
```

```
grid on;
xlabel('Update Time (ms)');
ylabel('Error (m)')
legend('Coasted','Update','Location','northwest');

end
```

Simulate Passive Radar Sensors and Radar Interferences

This example shows how to model and simulate the output of active and passive radar sensors using `radarDataGenerator`. In this example, you observe how radio frequency (RF) interference impacts the detection performance of a radar. In addition, you use passive radar sensors to estimate the location and type of the RF interference.

Create Scenario

Assessing tracking performance for radars requires modeling a radio frequency (RF) scenario. The modeling workflow is as follows:

- Generate RF emissions.
- Propagate the emissions and reflect these emissions from platforms.
- Receive the emissions, calculate interference losses, and generate detections.

In this example you perform each of these steps using a scenario consisting of three platforms:

1. Airborne platform flying north at 500 km/h from the bottom of the scenario
2. Airborne platform flying south at 600 km/h from the top of the scenario
3. Airborne platform flying east at 700 km/h in the middle of the scenario

First, create the scenario and set the scenario duration, then create the three airborne platforms.

```
scene = radarScenario;
scene.StopTime = 10; % s

% Platform 1: Airborne and northbound at 500 km/h
spd = 500*1e3/3600; % m/s
wp1 = [0 0 -6000];
wp2 = [spd*scene.StopTime 0 -6000];
toa = [0; scene.StopTime];
platform(scene,'Trajectory',waypointTrajectory('Waypoints',[wp1; wp2],'TimeOfArrival',toa));

% Platform 2: Airborne and southbound at 600 km/h
spd = 600*1e3/3600; % m/s
wp1 = [30e3+spd*scene.StopTime 0 -6000];
wp2 = [30e3 0 -6000];
toa = [0; scene.StopTime];
platform(scene,'Trajectory',waypointTrajectory('Waypoints',[wp1; wp2],'TimeOfArrival',toa));

% Platform 3: Airborne and eastbound at 700 km/h
spd = 700*1e3/3600; % m/s
wp1 = [10e3 1e3 -6000];
wp2 = [10e3 1e3+spd*scene.StopTime -6000];
toa = [0; scene.StopTime];
platform(scene,'Trajectory',waypointTrajectory('Waypoints',[wp1; wp2],'TimeOfArrival',toa));
```

Use `theaterPlot` to create a display showing the platforms in the scenario and their trajectories.

```
ax = axes;
theaterDisplay = theaterPlot('Parent',ax,'AxesUnit',["km" "km" "km"], 'XLim',[-10000 40000] , 'Yl
view([90 -90]) % swap X and Y axis
patch('XData',[-10000 -10000 40000 40000],'YData',[-20000 20000 20000 -20000],'EdgeColor','none'
```

```
platPlotter = platformPlotter(theaterDisplay,'DisplayName','Platforms','MarkerFaceColor','k');
plotPlatform(platPlotter,vertcat(scene.platformPoses.Position));

trajPlotter = trajectoryPlotter(theaterDisplay,'DisplayName','Trajectories','LineStyle','-');
allTrajectories = cellfun(@(x) x.Trajectory.lookupPose(linspace(0,scene.StopTime,10)), scene.Pla
plotTrajectory(trajPlotter,allTrajectories);
```
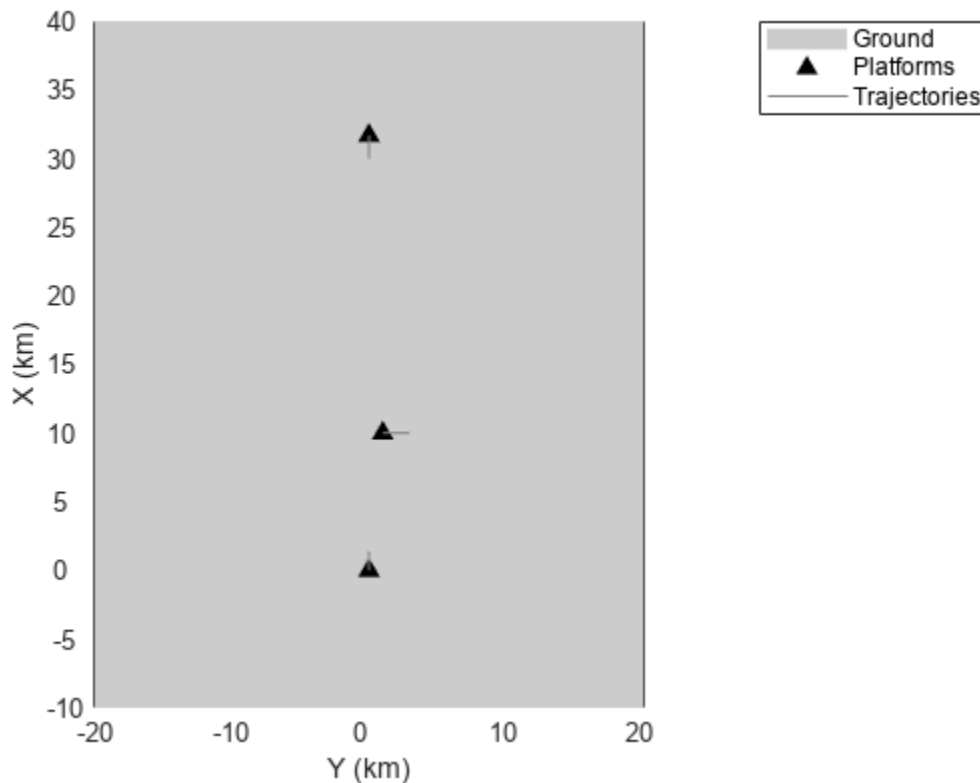

Radar Detection in the Presence of Interfering Emissions

Often, a radar operates in an environment where other undesirable RF emissions interfere with the waveforms emitted by the radar. When this occurs, the radar experiences a degradation in detection performance in the direction of the interfering signals. Attach an RF emitter to the platform at the bottom of the scenario (the first platform) and a radar to the platform at the top of the scenario (the second platform).

Create RF Emitter

Model an RF emission using a `radarEmitter` object. The emitter is in a forward-looking configuration with an azimuthal field of view of 20 degrees to include the two other platforms in the scenario. The effective isotropic radiated power (EIRP) sets the strength of the interfering signal. The waveform type is a user-defined value used to enumerate the various waveform types that are present in the scenario. For this scenario, use the value 0 to indicate a noise waveform type.

```
% Create the interference emitter.
rfEmitter = radarEmitter(1,'No scanning', ...
    'FieldOfView',[20 5], ...        % [az el] deg
```

```
    'EIRP',200, ...                    % dBi
    'CenterFrequency',300e6, ...       % Hz
    'Bandwidth',30e6, ...              % Hz
    'WaveformType',0)                  % Use 0 for noise-like emissions

rfEmitter =
  radarEmitter with properties:

        EmitterIndex: 1
          UpdateRate: 1
            ScanMode: 'No scanning'

    MountingLocation: [0 0 0]
      MountingAngles: [0 0 0]

         FieldOfView: [2x1 double]
     MechanicalAngle: 0
           LookAngle: 0
        HasElevation: 0

                EIRP: 200
     CenterFrequency: 300000000
           Bandwidth: 30000000
        WaveformType: 0
      ProcessingGain: 0
```

Attach the emitter to the first platform.

```
platEmit = scene.Platforms{1};
platEmit.Emitters = rfEmitter

platEmit =
  Platform with properties:

         PlatformID: 1
            ClassID: 0
           Position: [0 0 -6000]
        Orientation: [0 0 0]
         Dimensions: [1x1 struct]
         Trajectory: [1x1 waypointTrajectory]
      PoseEstimator: [1x1 insSensor]
           Emitters: {[1x1 radarEmitter]}
            Sensors: {}
         Signatures: {[1x1 rcsSignature]}
```

Create Monostatic Radar

Equip the second platform with a monostatic radar. Use `radarDataGenerator` to model this type of radar. First, create a monostatic radar using `radarDataGenerator`. Configure the radar's mounting orientation so that it scans the azimuth sector in front of its platform. Enable the INS input so that the radar can use the platform's pose estimator to output detections in scenario coordinates. Enable the interference input port so the interference signal created by the emitter above can be passed to the radar.

```
radar = radarDataGenerator(2, 'Sector', ...
    'DetectionMode', 'monostatic', ...
```

```
    'UpdateRate', 12.5, ...          % Hz
    'FieldOfView', [2 10]);          % [az el] deg

radar.MountingAngles = [0 0 0];      % [Z Y X] deg
radar.HasINS = true;
radar.InterferenceInputPort = true;
radar.DetectionCoordinates = 'scenario'

radar =
  radarDataGenerator with properties:

               SensorIndex: 2
                UpdateRate: 12.5000
             DetectionMode: 'Monostatic'
                  ScanMode: 'Mechanical'
      InterferenceInputPort: 1
        EmissionsInputPort: 0

           MountingLocation: [0 0 0]
             MountingAngles: [0 0 0]

                FieldOfView: [2 10]
                  LookAngle: [0 0]
                RangeLimits: [0 100000]

       DetectionProbability: 0.9000
             FalseAlarmRate: 1.0000e-06
             ReferenceRange: 100000

         TargetReportFormat: 'Clustered detections'

  Show all properties
```

Attach the radar to the second platform.

```
platRadar = scene.Platforms{2};
platRadar.Sensors = radar;
```

Update the display to show the platforms, the radar, and the emitter in the scenario.

```
emitterColor =  [0.9290 0.6940 0.1250];
radarColor = [0 0.4470 0.7410];
platEmitPlotter = platformPlotter(theaterDisplay,'DisplayName', 'RF emitter','Marker','d','Marker
platRadarPlotter = platformPlotter(theaterDisplay,'DisplayName','Monostatic radar','Marker','d',
platPlotter.DisplayName = 'Targets';
clearData(platPlotter);
covPlotter = coveragePlotter(theaterDisplay,'Alpha',[0.2 0]);
detPlotter = detectionPlotter(theaterDisplay,'DisplayName','Radar detections','MarkerFaceColor',
title('Radar Detection With an Interfering Emitter');

plotPlatform(platRadarPlotter, platRadar.pose.Position);
plotPlatform(platEmitPlotter, platEmit.pose.Position);
plotPlatform(platPlotter, scene.Platforms{3}.pose.Position);
plotCoverage(covPlotter, coverageConfig(scene), [-1 2], {emitterColor, radarColor});
```

1-495

Radar Detection With an Interfering Emitter
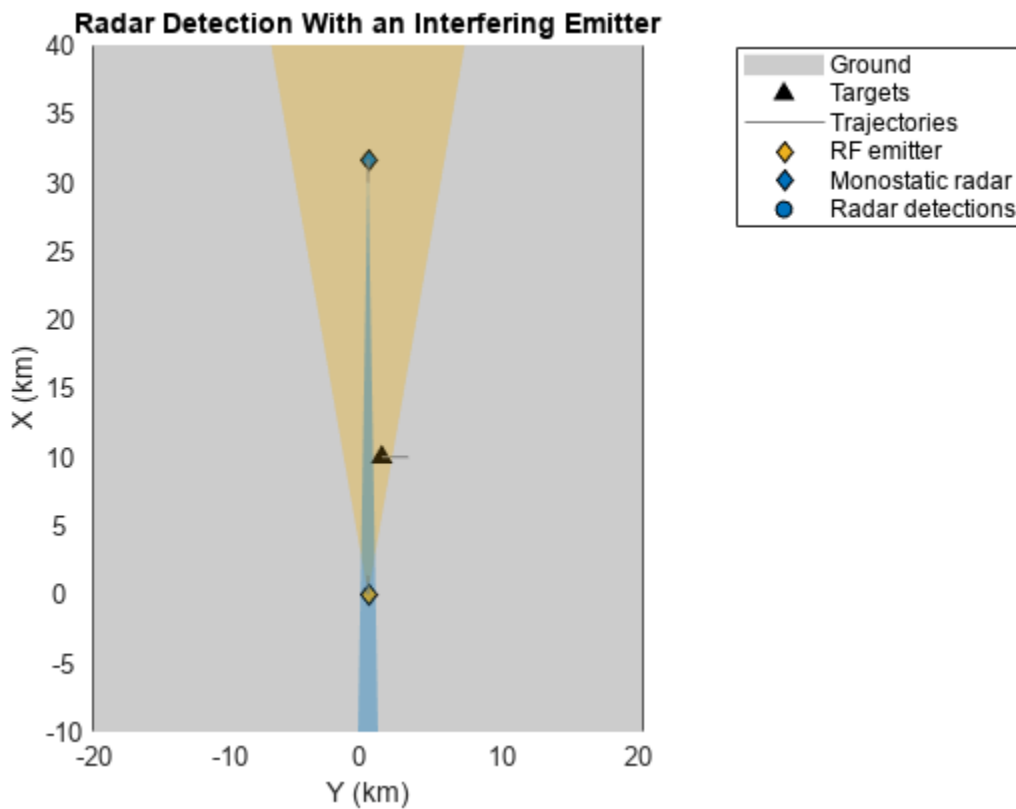
In the preceding figure, the platform carrying the forward-looking radar is shown as a blue diamond, and the radar's current field of view is a blue region originating from the platform. At the bottom of the figure, the platform carrying the interfering RF emission is shown as a yellow diamond, and the emitter's current field of view is a corresponding yellow region. Platforms without any emitters or sensors attached to them are referred to as *targets* and are displayed as black triangles.

Simulate Monostatic Detections

In multirate scenarios, you can either find an update rate that is a common divider of all the sensors and emitter rates defined in the scenario or you can use the continuous update, which automatically advances the scenario to the next valid update time when you call `advance`.

```
scene.UpdateRate = 0
```

```
scene =
  radarScenario with properties:

       IsEarthCentered: 0
            UpdateRate: 0
        SimulationTime: 0
              StopTime: 10
      SimulationStatus: NotStarted
             Platforms: {[1x1 radar.scenario.Platform]  [1x1 radar.scenario.Platform]  [1x1 radar
         SurfaceManager: [1x1 radar.scenario.SurfaceManager]
      AtmosphereManager: [1x1 radar.scenario.AtmosphereManager]
```

For each step in the following loop, use:

1. `advance` to move all of the platforms according to their trajectories.

2. `emit` to update the transmit direction of emissions from `platEmit`.

3. `propagate` to propagate the emissions directly to each platform in the scenario that lies within the emitter's field of view. Each platform that receives a direct path emission generates a single-bounce reflection that is also propagated to every other platform as a reflected emission.

4. `detect` to generate detections from the emissions received at `platRadar`.

The following figure shows the propagation of emissions from the emitter to the radar sensor.
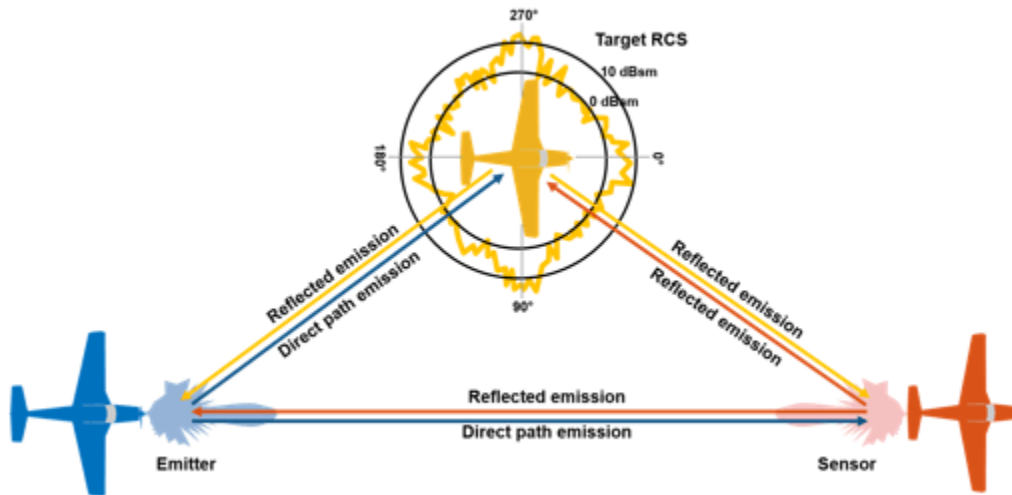

```matlab
% Set the random seed for repeatable results.
rng(2018);

plotDets = {};
while advance(scene)

    % Emit the RF signal.
    txEmiss = emit(scene);

    % Reflect the emitted signal off the platforms in the scenario.
    reflEmiss = propagate(scene, txEmiss);

    % Generate detections from the monostatic radar sensor.
    [dets, config] = detect(scene, reflEmiss);

    % Reset detections every time the radar completes a sector scan.
    if config.IsScanDone
        % Reset
        plotDets = dets;
    else
        % Buffer
        plotDets = [plotDets;dets]; %#ok<AGROW>
    end

    % Update display with current platform positions, beam positions and detections.
    plotPlatform(platRadarPlotter, platRadar.pose.Position);
```

```
    plotPlatform(platEmitPlotter, platEmit.pose.Position);
    plotPlatform(platPlotter, scene.Platforms{3}.pose.Position);
    plotCoverage(covPlotter, coverageConfig(scene), [-1 2], {emitterColor, radarColor});
    if ~isempty(plotDets)
        allDets = [plotDets{:}];
        % Extract column vector of measurement positions
        meas = [allDets.Measurement]';
        plotDetection(detPlotter,meas);
    end
end
```


Radar Detection With an Interfering Emitter
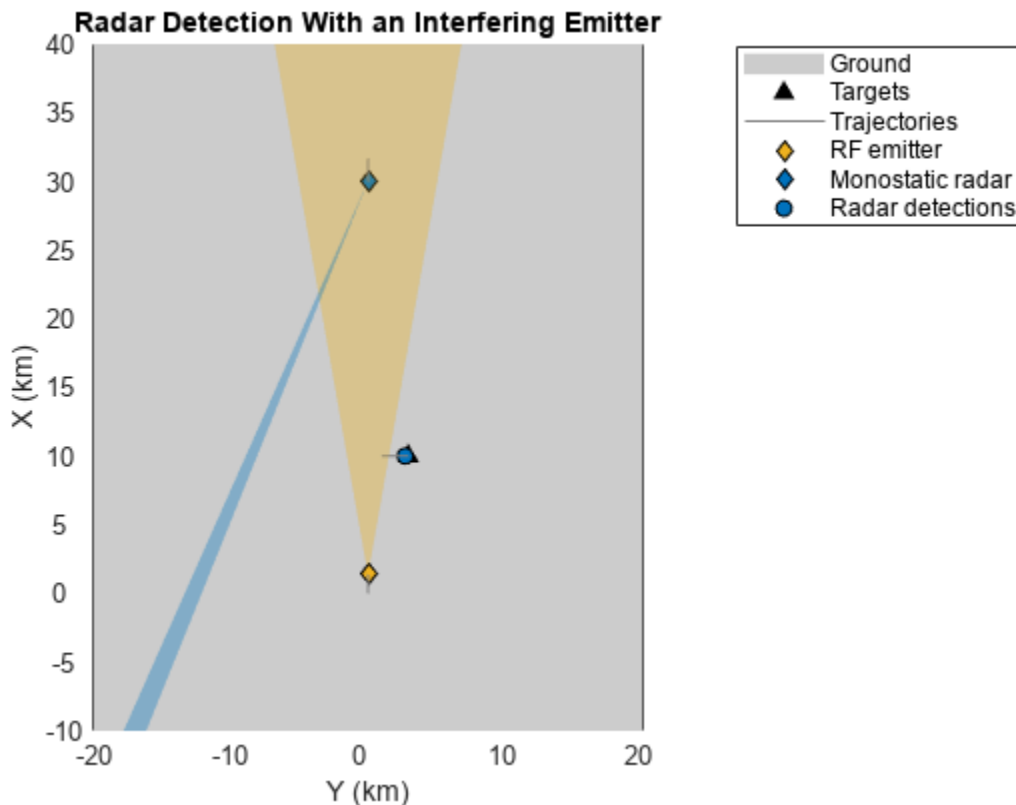
The preceding figure shows that the radar (shown in blue) is only able to detect the target in the middle of the scenario. The detections are shown as blue, filled circles, and are made whenever the radar's field of view (that is, the beamwidth) sweeps across the target. However, when the radar's beam sweeps across the emitting platform (shown in yellow), no detections are generated, since the interference generated by this platform prevents detection by the radar.

Passive Detection of RF Emissions

In the preceding section, the radar is unable to detect the location of the emitting platform because the emissions from that platform mask the radar's own emissions. However, such strong emissions can be detected and identified by passive sensors that listen for RF emissions. These sensors are often referred to as electronic support measures (ESM). These sensors typically listen across a broad range of frequencies and attempt to identify unique emitters, the direction of arrival of the emissions from those emitters, and whenever possible, the type of waveform used by the emitter.

Create an ESM Sensor

Reuse the scenario from the previous section, but replace the monostatic radar on the first platform with an ESM sensor. Use `radarDataGenerator` to model the ESM sensor and ensure that the sensor is configured so that its center frequency and bandwidth includes the RF spectrum of the emitter. Otherwise, it will be unable to detect the emitter.

```
restart(scene);
esm = radarDataGenerator(1,'No scanning', ...
    'DetectionMode','ESM', ...
    'UpdateRate',12.5, ...          % Hz
    'MountingAngles',[0 0 0], ...   % [Z Y X] deg
    'FieldOfView',[30 10], ...      % [az el] deg
    'CenterFrequency',300e6, ...    % Hz
    'Bandwidth',30e6, ...           % Hz
    'WaveformTypes',0, ...          % Detect the interference waveform type
    'HasINS',true)

esm =
  radarDataGenerator with properties:

         SensorIndex: 1
        EmitterIndex: 1
          UpdateRate: 12.5000
       DetectionMode: 'ESM'
            ScanMode: 'No scanning'

    MountingLocation: [0 0 0]
      MountingAngles: [0 0 0]

          FieldOfView: [30 10]

       FalseAlarmRate: 1.0000e-06

  Show all properties
```

Replace the radar on the second platform with the ESM sensor.

```
platESM = scene.Platforms{2};
platESM.Sensors = esm;
```

Update the visualization accordingly.

```
platRadarPlotter.DisplayName = "ESM sensor";
esmColor = [0.4940 0.1840 0.5560];
platRadarPlotter.MarkerFaceColor = esmColor;
% use a helper to add an angle-only detection plotter
delete(detPlotter);
esmDetPlotter = helperAngleOnlyDetectionPlotter(theaterDisplay,'DisplayName','ESM detections','Co
clearData(covPlotter);

plotCoverage(covPlotter, coverageConfig(scene), [-1 1], {emitterColor, esmColor});
title('Passive detection of RF emissions');
```

Passive detection of RF emissions
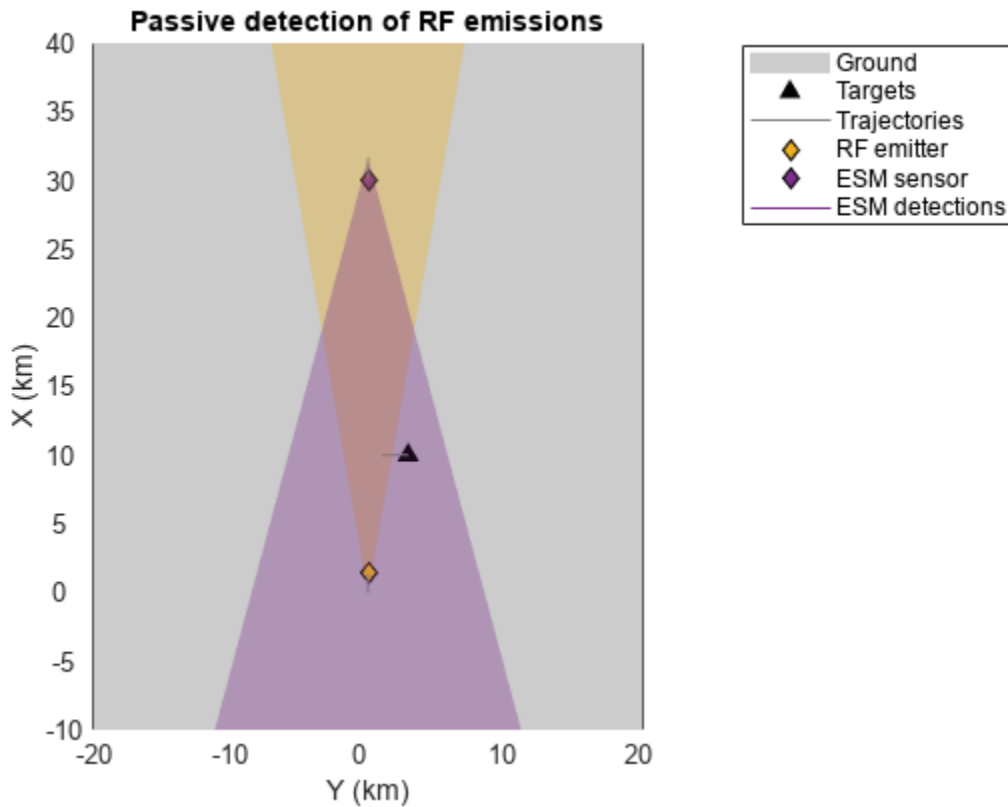
In the preceding figure, the radar is replaced by an ESM sensor mounted on the second platform. The ESM sensor's field of view is shown in magenta and includes both the emitting and target platforms.

Simulate ESM Detections

Now simulate detections using the ESM sensor instead of the radar. Notice that because the ESM sensor is a passive sensor, it is unable to localize the emitting platform, but indicates the direction of arrival of the platform's emissions. These angle-only detections are shown as rays originating from the ESM sensor toward the emitting platform.

```
% Set the random seed for repeatable results.
rng(2018);

plotDets = {};
snap = [];
while advance(scene)

    % Emit the RF signal.
    txEmiss = emit(scene);

    % Reflect the emitted signal off the platforms in the scenario.
    reflEmiss = propagate(scene, txEmiss);

    % Generate detections from the ESM sensor.
    [dets, config] = detect(scene, reflEmiss);

    % Reset detections every time the radar completes a sector scan.
```

```
        if config.IsScanDone
            % Reset
            plotDets = dets;
        else
            % Buffer
            plotDets = [plotDets;dets]; %#ok<AGROW>
        end

        % Update display with current platform positions, beam positions, and detections.
        plotPlatform(platRadarPlotter, platRadar.pose.Position);
        plotPlatform(platEmitPlotter, platEmit.pose.Position);
        plotPlatform(platPlotter, scene.Platforms{3}.pose.Position);
        plotCoverage(covPlotter, coverageConfig(scene), [-1 1], {emitterColor, esmColor});
        plotDetection(esmDetPlotter,plotDets);

        % Record the reflected detection at t = 2 seconds.
        snap = getSnap(ax, scene.SimulationTime, 2, snap);
        drawnow
    end
title('RF emitter detected by an ESM sensor');
```
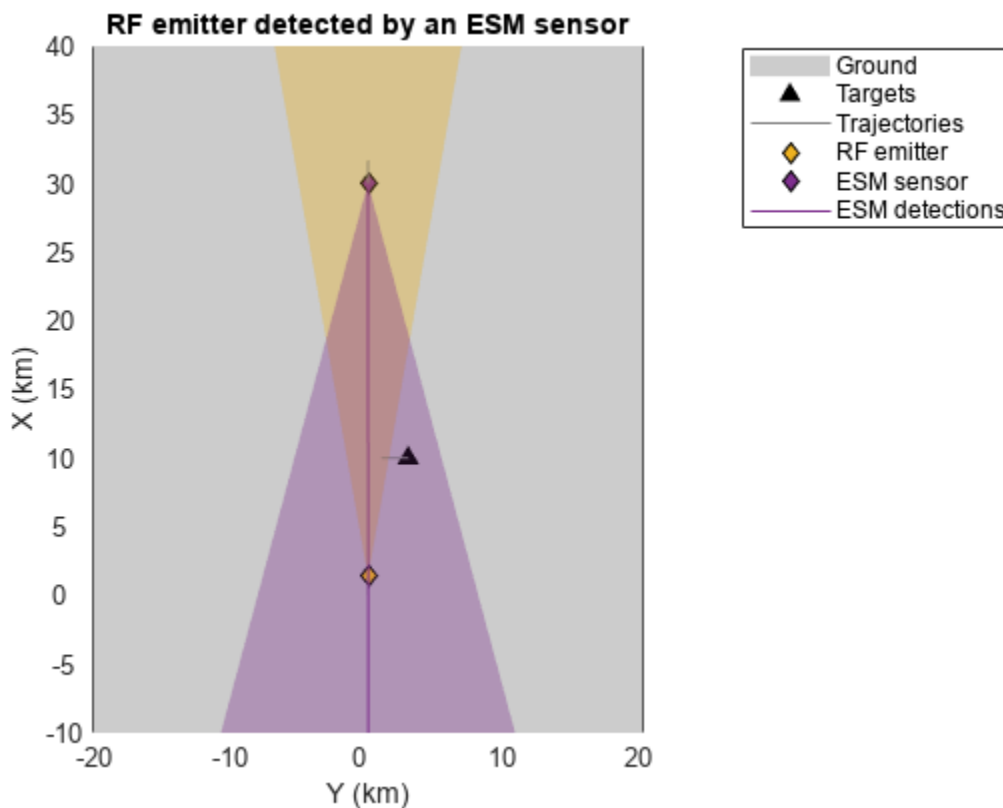

The ESM sensor detects the RF emissions and estimates their direction of arrival. This estimate is shown by the magenta line originating from the sensor and closely passing by the emitter. The angle estimate is noisy, which is why the line does not pass directly through the location of the emitter.

The ESM sensor classifies the waveform types in its reported detections. For this emitter, it reports the noise waveform type used by the emitter: 0.

```
dets{1}.ObjectAttributes{1}
```

```
ans = struct with fields:
    TargetIndex: 1
   EmitterIndex: 1
   WaveformType: 0
           SNR: 173.8303
```

Notice that the signal-to-noise ratio (SNR) of the emitted signal detected by the sensor is very large, 185 dB. Because the RF emitter has high power, reflections of the emitted waveform off the target are also detected by the ESM sensor. This is seen at 2 seconds into the simulation, when the target lies within the field of view of the emitter.

```
figure; imshow(snap.cdata);
title('Emitter and Target Detected by an ESM Sensor');
```
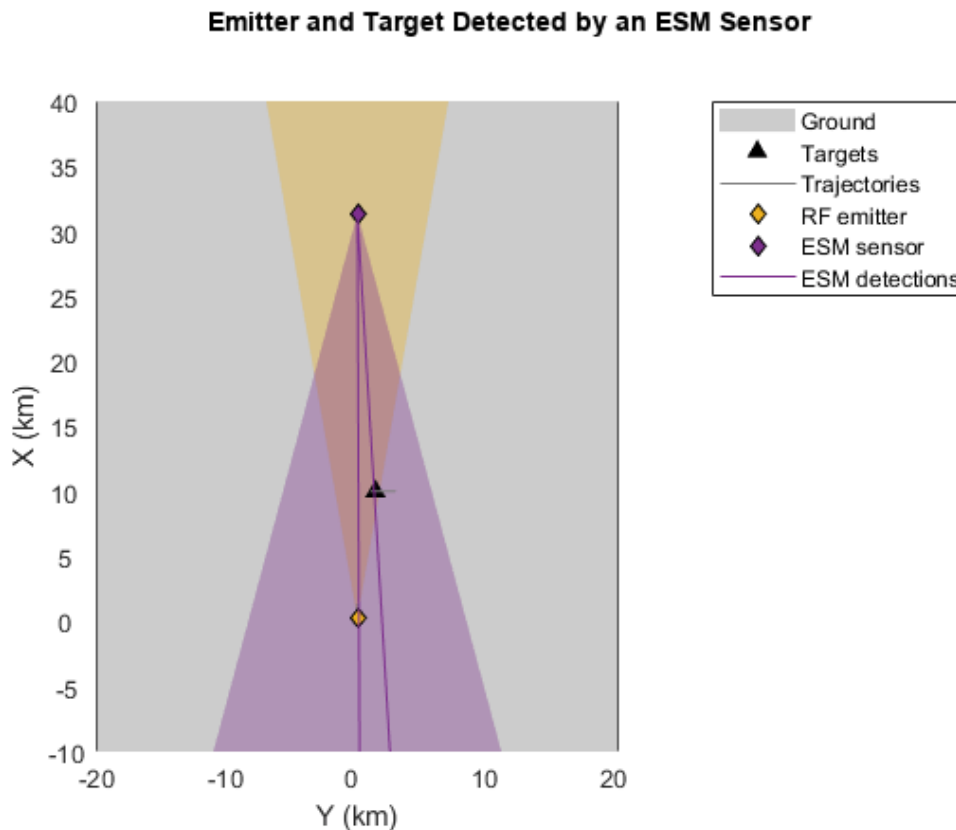

The preceding figure shows emissions that are detected from both the emitter and the target, as the target receives energy from the emitter and re-emits that waveform back into the scenario, causing it to be detected by the ESM sensor as well.

Passive Detection of Monostatic Radars

Monostatic radars also emit waveforms into the scenario. Passive detection of these emissions is sometimes desirable. To do so, you must model both the emitting and sensing portions of the radar separately. The emitter generates the waveforms that become part of the scenario's RF emissions. These waveforms can then be detected by other sensors, such as an ESM sensor.

Reuse the same scenario from before. For this scenario, attach a monostatic radar to the platform at the top of the scenario (the second platform), and attach an ESM sensor to the platform at the bottom of the scenario (the first platform). The middle platform remains a target with no emitters or sensors attached to it.

```
restart(scene);
```

Create a monostatic radar by modeling both the emitting and sensing portions of the sensor. Use `radarEmitter` to model the monostatic radar emitter. For this scenario, use 1 to indicate the waveform type used by this radar. The waveform type is an enumeration defined by the user to represent the different kinds of waveforms simulated in the scenario. The waveform enumeration enables emitters and sensors to know how to process these waveforms to generate detections. For example, if an emitter has a waveform type of 1 and a sensor includes this in its waveform list, then the sensor knows how to process the emitter's emissions (for example, using a matched filter) and realizes the processing gain associated with that waveform.

```
% Create the emitter for the monostatic radar.
radarTx = radarEmitter(2,'Sector', ...
    'UpdateRate',12.5, ...          % Hz
    'MountingAngles',[0 0 0], ...   % [Z Y X] deg
    'FieldOfView',[2 10], ...       % [az el] deg
    'CenterFrequency',300e6, ...    % Hz
    'Bandwidth',3e6, ...            % Hz
    'ProcessingGain',50, ...        % dB
    'WaveformType',1)               % Use 1 to indicate this radar's waveform.

radarTx =
  radarEmitter with properties:

            EmitterIndex: 2
              UpdateRate: 12.5000
                ScanMode: 'Mechanical'

        MountingLocation: [0 0 0]
          MountingAngles: [0 0 0]

              FieldOfView: [2x1 double]
    MaxMechanicalScanRate: 75
     MechanicalScanLimits: [-45 45]
          MechanicalAngle: 0
                LookAngle: 0
             HasElevation: 0

                     EIRP: 100
          CenterFrequency: 300000000
                Bandwidth: 3000000
             WaveformType: 1
           ProcessingGain: 50
```

Use `radarDataGenerator` to model the sensing portion of the radar that receives the RF emissions in the scenario, identifies the emissions that correspond to the monostatic emitter's waveform type, and generates detections from these received emissions. Emissions that do not match the emitter's waveform type are treated as interference.

When using `radarDataGenerator` to model the sensing portion of a monostatic radar, set the `DetectionMode` property of the sensor to `Monostatic`. This tells the sensor to use the emitter's configuration when processing the received RF emissions. The `EmissionsInputPort` property must also be set true to enable detections on `radarEmission` objects.

```
radarRx = radarDataGenerator(2, ...
    'DetectionMode','Monostatic', ...
    'EmissionsInputPort',true, ...
    'EmitterIndex',radarTx.EmitterIndex, ...
    'HasINS',true, ...
    'DetectionCoordinates','Scenario')

radarRx =
  radarDataGenerator with properties:

              SensorIndex: 2
             EmitterIndex: 2
            DetectionMode: 'Monostatic'
    InterferenceInputPort: 0
       EmissionsInputPort: 1

               RangeLimits: [0 100000]

            FalseAlarmRate: 1.0000e-06

  Show all properties
```

Attach the radar emitter and sensor to the second platform.

```
platRadar = scene.Platforms{2};
platRadar.Emitters = radarTx;
platRadar.Sensors = radarRx;
```

Reuse the ESM sensor from before, but set the list of known waveform types for the ESM sensor to include the waveform emitted by the radar. If the radar's waveform type is not known to the ESM sensor, it will not be detected.

```
% Add the radar's waveform to the list of known waveform types for the ESM sensor.
esm.WaveformTypes = [0 1];

% Attach the ESM sensor to the first platform.
platESM = scene.Platforms{1};
platESM.Emitters = {}; % Remove the emitter.
platESM.Sensors = esm;
```

Update the display to show both monostatic detections and ESM detections.

```
detPlotter = detectionPlotter(theaterDisplay,'DisplayName','Radar detections','MarkerFaceColor',
platRadarPlotter.DisplayName = 'Monostatic radar';
platRadarPlotter.MarkerFaceColor = radarColor;
platEmitPlotter.DisplayName = 'ESM sensor';
platEmitPlotter.MarkerFaceColor = esmColor;
```

```
clearData(esmDetPlotter);
clearData(covPlotter);
covcon = coverageConfig(scene);
plotCoverage(covPlotter, covcon([1 3]) , [1 -2], {esmColor, radarColor});
title(ax,'Passive detection of a monostatic radar');
```
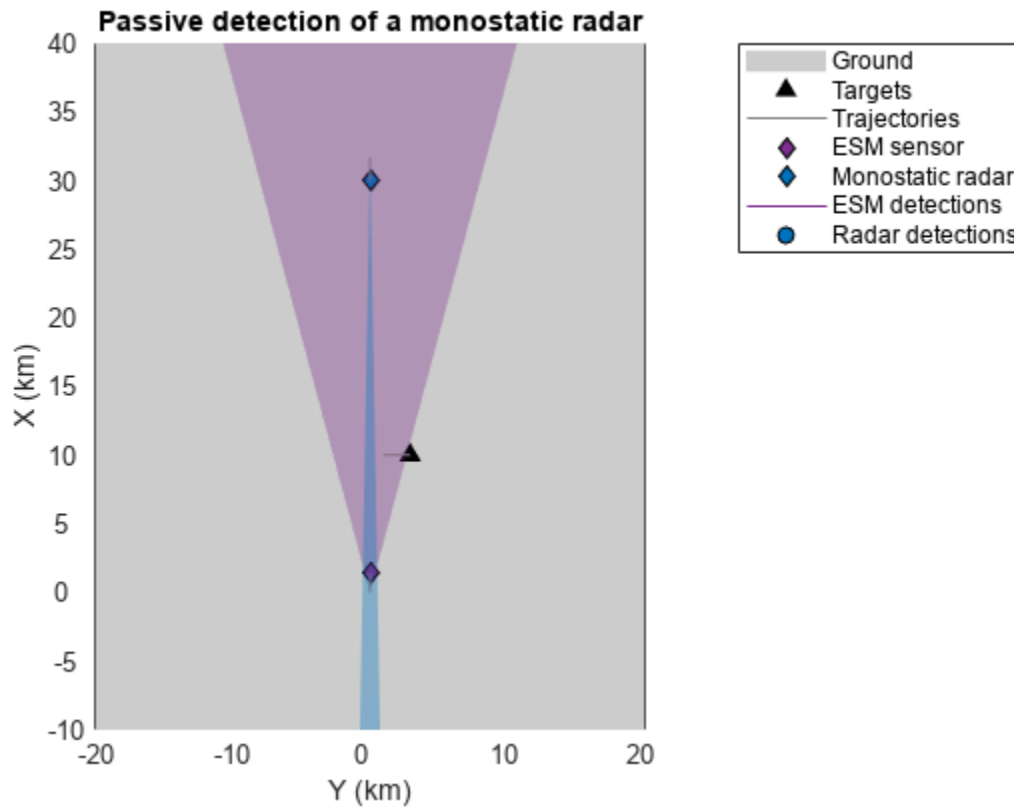

The preceding figure shows the radar scanning an azimuth sector in front of its platform, which includes both the target platform as well as the platform carrying the ESM sensor. The radar generates detections for both of these platforms when its field of view (shown in blue) sweeps over their location. However, when the radar's beam passes over the location of the ESM sensor, the ESM sensor detects the radar and indicates the estimated position by drawing a line originating from the sensor.

```
% Set the random seed for repeatable results.
rng(2018);

platforms = scene.Platforms;
numPlat = numel(platforms);


plotDets = {};
snap = [];
while advance(scene)

    % Emit the RF signal.
    [txEmiss, txConfigs] = emit(scene);
```

```matlab
    % Reflect the emitted signal off the platforms in the scenario.
    reflEmiss = propagate(scene, txEmiss);

    % Generate detections from the sensors.
    [dets, config] = detect(scene, reflEmiss, txConfigs);

    % Reset detections every time the radar completes a sector scan.
    if txConfigs(end).IsScanDone
        % Reset
        plotDets = dets;
    else
        % Buffer
        plotDets = [plotDets;dets];%#ok<AGROW>
    end

    % Update display with current platform positions, beam positions, and detections.
    plotPlatform(platRadarPlotter, platRadar.pose.Position);
    plotPlatform(platEmitPlotter, platEmit.pose.Position);
    plotPlatform(platPlotter, scene.Platforms{3}.pose.Position);
    covcon = coverageConfig(scene);
    plotCoverage(covPlotter, covcon([1 3]) , [1 -2], {esmColor, radarColor});
    plotDetection(esmDetPlotter,plotDets);
    plotMonostaticDetection(detPlotter,plotDets);

    % Record the reflected detection at t = 5.6 seconds.
    snap = getSnap(ax, scene.SimulationTime, 5.6, snap);
    drawnow
end
```
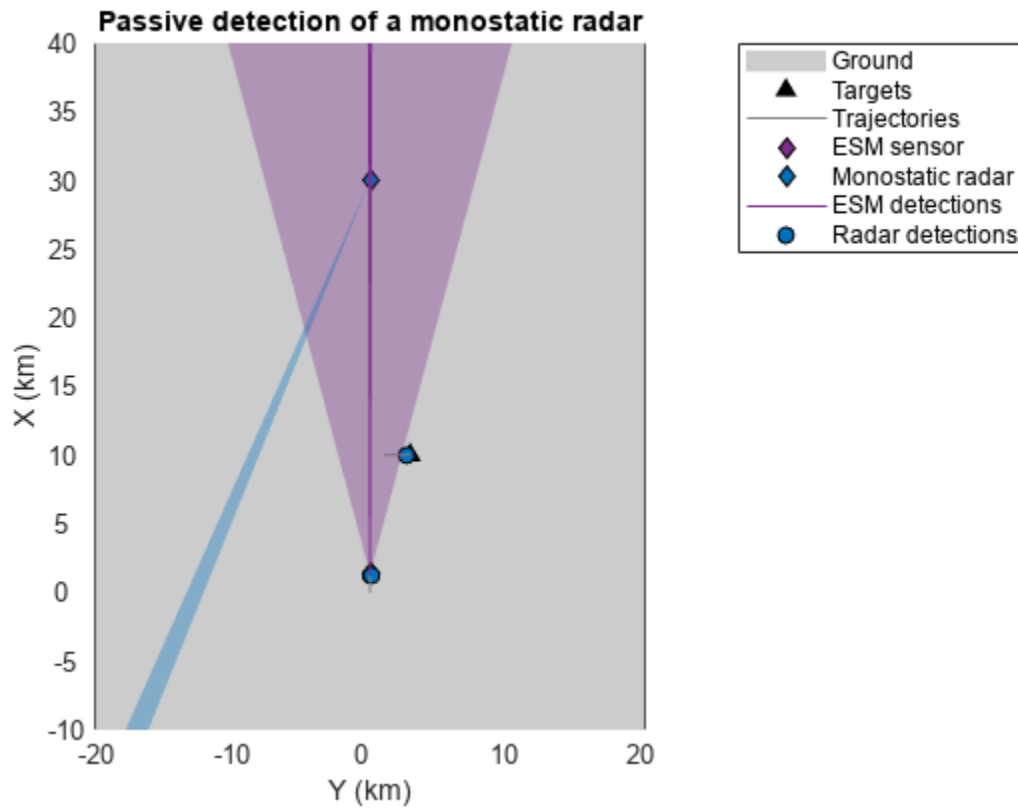
Detections from the monostatic radar modeled using `radarEmitter` and `radarDataGenerator` are shown as filled, blue circles near the target and the platform equipped with the ESM sensor. The ESM sensor is also able to detect the radar, as is indicated by the angle-only detection shown as a line originating from the ESM sensor and passing near the radar platform.

```
figure; imshow(snap.cdata);
title('Radar and Target Detected by an ESM Sensor');
```

Radar and Target Detected by an ESM Sensor
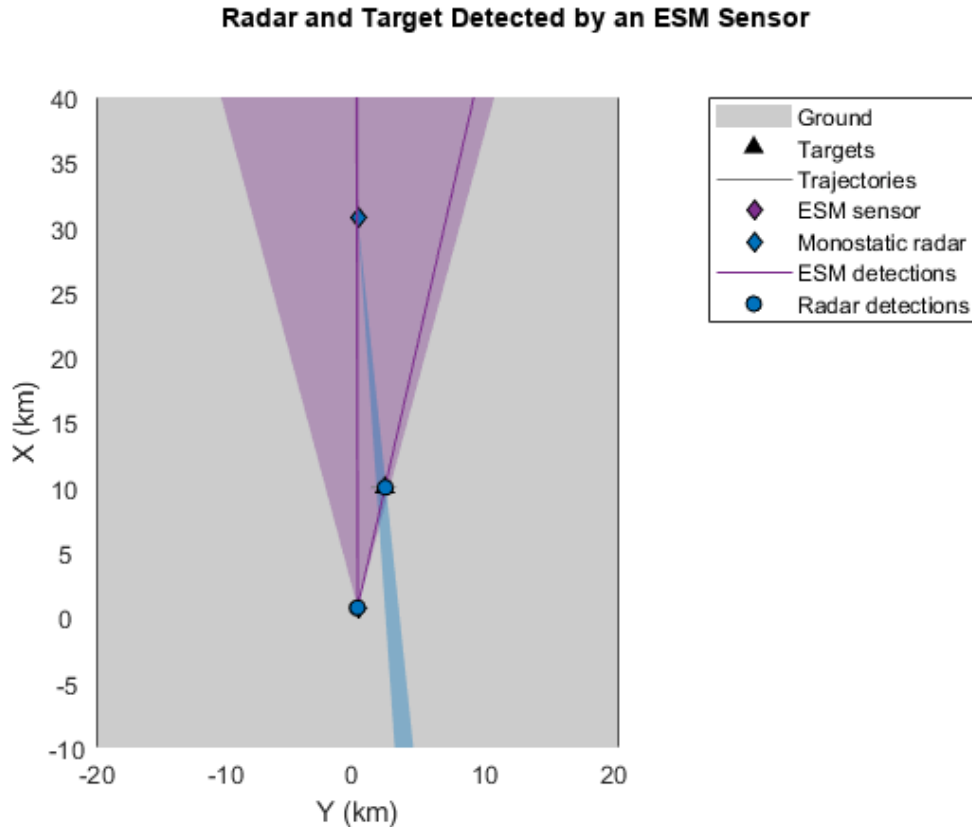
Because of the high power (EIRP) of the radar's emissions, the emitted energy is reflected off the target toward the ESM platform. Consequently, the ESM sensor detects the target platform when the radar emitter's field of view sweeps past the target platform while the target is still inside of the ESM sensor's field of view.

Supporting Functions

getSnap records a snapshot of an axis at a given snap time.

```
function snap = getSnap(hAx, curTime, snapTime, prevSnap)
if ~isempty(prevSnap)
    snap = prevSnap;
elseif curTime >= snapTime && curTime < snapTime + 0.05
    hAx.Title.Visible = 'off';
    snap = getframe(hAx.Parent);
    hAx.Title.Visible = 'on';
else
    snap = [];
end
end
```

plotMonostaticDetection parses detections to plot only monostatic detections with a detectionPlotter.

```
function plotMonostaticDetection(plotter, dets)
if ~isempty(dets)
    % Pass only monostatic detections to the detectionPlotter
    radarDetId = cellfun(@(x) x.SensorIndex == 2, dets);
    if any(radarDetId)
        % Extract measurement positions for the monostatic radar
        radarDets = [dets{radarDetId}];
        meas = [radarDets.Measurement]';
        plotDetection(plotter,meas);
    end
end
end
```

Introduction to Micro-Doppler Effects

This example introduces the basic concept of a micro-Doppler effect in the radar return of a target due to the rotation of that target. You can use the micro-Doppler signature to help identify the target.

Introduction

A moving target introduces a frequency shift in the radar return due to Doppler effect. However, because most targets are not rigid bodies, there are often other vibrations and rotations in different parts of the target in addition to the platform movement. For example, when a helicopter flies, its blades rotate, or when a person walks, their arms swing naturally. These micro scale movements produce additional Doppler shifts, referred to as micro-Doppler effects, which are useful in identifying target features. This example shows two applications where micro-Doppler effects can be helpful. In the first application, micro-Doppler signatures are used to determine the blade speed of a helicopter. In the second application, the micro-Doppler signatures are used to identify a pedestrian in an automotive radar return.

Estimating Blade Speed of A Helicopter

Consider a helicopter with four rotor blades. Assume the radar is located at the origin. Specify the location of the helicopter as (500, 0, 500), which sets its distance away from the radar in meters and a velocity of (60, 0, 0) m/s.

```
radarpos = [0;0;0];
radarvel = [0;0;0];

tgtinitpos = [500;0;500];
tgtvel     = [60;0;0];
tgtmotion  = phased.Platform('InitialPosition',tgtinitpos,'Velocity',tgtvel);
```

In this simulation, the helicopter is modeled by five scatterers: the rotation center and the tips of four blades. The rotation center moves with the helicopter body. Each blade tip is 90 degrees apart from the tip of its neighboring blades. The blades are rotating at a constant speed of 4 revolutions per second. The arm length of each blade is 6.5 meters.

```
Nblades   = 4;
bladeang  = (0:Nblades-1)*2*pi/Nblades;
bladelen  = 6.5;
bladerate = deg2rad(4*360);   % rps -> rad/sec
```

All four blade tips are assumed to have identical reflectivities while the reflectivity for the rotation center is stronger.

```
c  = 3e8;
fc = 5e9;
helicop = phased.RadarTarget('MeanRCS',[10 .1 .1 .1 .1],'PropagationSpeed',c,...
    'OperatingFrequency',fc);
```

Helicopter Echo Simulation

Assume the radar operates at 5 GHz with a simple pulse. The pulse repetition frequency is 20 kHz. For simplicity, assume the signal propagates in free space.

```
fs     = 1e6;
prf    = 2e4;
lambda = c/fc;

wav = phased.RectangularWaveform('SampleRate',fs,'PulseWidth',2e-6,'PRF',prf);
ura = phased.URA('Size',4,'ElementSpacing',lambda/2);
tx  = phased.Transmitter;
rx  = phased.ReceiverPreamp;
env = phased.FreeSpace('PropagationSpeed',c,'OperatingFrequency',fc,...
    'TwoWayPropagation',true,'SampleRate',fs);
txant = phased.Radiator('Sensor',ura,'PropagationSpeed',c,'OperatingFrequency',fc);
rxant = phased.Collector('Sensor',ura,'PropagationSpeed',c,'OperatingFrequency',fc);
```

At each pulse, the helicopter moves along its trajectory. Meanwhile, the blades keep rotating, and the tips of the blades introduce additional displacement and angular speed.

```
NSampPerPulse = round(fs/prf);
Niter = 1e4;
y     = complex(zeros(NSampPerPulse,Niter));
rng(2018);
for m = 1:Niter
    % update helicopter motion
    t = (m-1)/prf;
    [scatterpos,scattervel,scatterang] = helicopmotion(t,tgtmotion,bladeang,bladelen,bladerate);

    % simulate echo
    x  = txant(tx(wav()),scatterang);                      % transmit
    xt = env(x,radarpos,scatterpos,radarvel,scattervel);   % propagates to/from scatterers
    xt = helicop(xt);                                       % reflect
    xr = rx(rxant(xt,scatterang));                         % receive
    y(:,m) = sum(xr,2);                                    % beamform
end
```

This figure shows the range-Doppler response using the first 128 pulses of the received signal. You can see the display of three returns at the target range of approximately 700 meters.

```
rdresp  = phased.RangeDopplerResponse('PropagationSpeed',c,'SampleRate',fs,...
    'DopplerFFTLengthSource','Property','DopplerFFTLength',128,'DopplerOutput','Speed',...
    'OperatingFrequency',fc);
mfcoeff = getMatchedFilter(wav);
plotResponse(rdresp,y(:,1:128),mfcoeff);
ylim([0 3000])
```
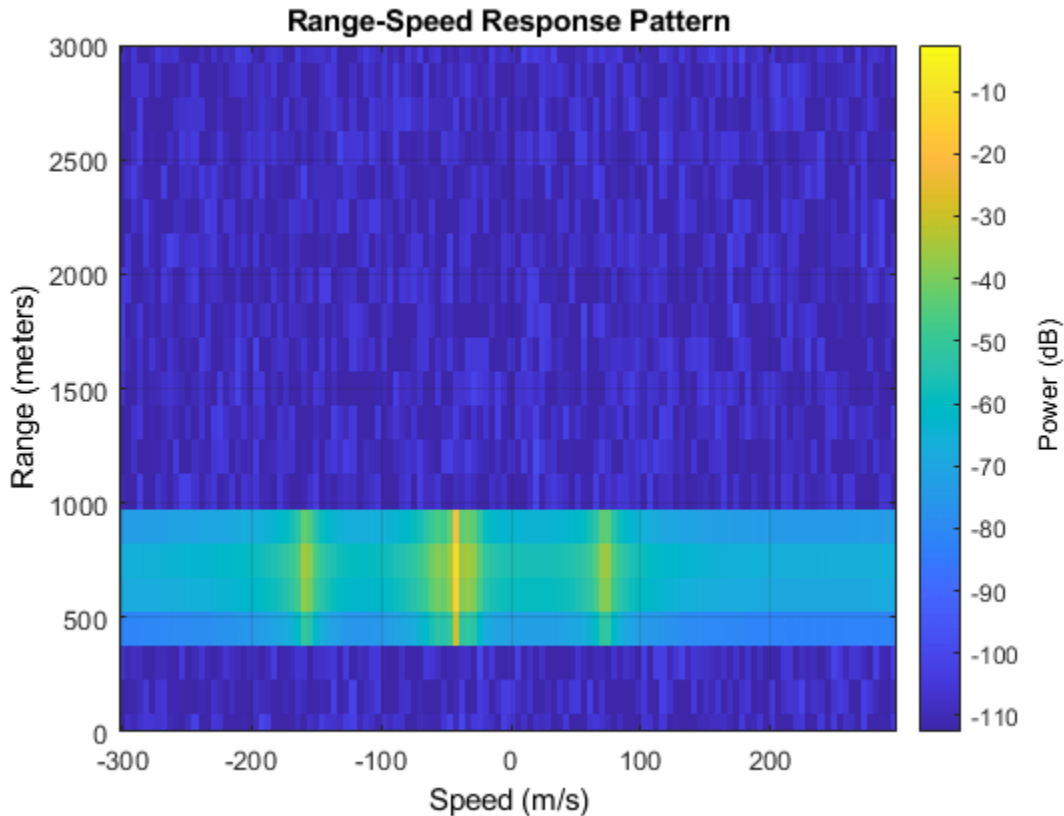

While the returns look as though they are from different targets, they are actually all from the same target. The center return is from the rotation center, and is much stronger compared to the other two returns. This intensity is because the reflection is stronger from the helicopter body when compared to the blade tips. The plot shows a speed of -40 m/s for the rotation center. This value matches the truth of the target radial speed.

```
tgtpos = scatterpos(:,1);
tgtvel = scattervel(:,1);
tgtvel_truth = radialspeed(tgtpos,tgtvel,radarpos,radarvel)
```

```
tgtvel_truth =

  -43.6435
```

The other two returns are from the tips of the blades when they approach or depart the target at maximum speed. From the plot, the speeds corresponding to these two approach and depart detections are about 75 m/s and -160 m/s, respectively.

```
maxbladetipvel = [bladelen*bladerate;0;0];
vtp = radialspeed(tgtpos,-maxbladetipvel+tgtvel,radarpos,radarvel)
vtn = radialspeed(tgtpos,maxbladetipvel+tgtvel,radarpos,radarvel)
```

```
vtp =

   75.1853
```

```
vtn =

 -162.4723
```

You can associate all three detections to the same target via further processing, but that topic is beyond the scope of this example.

Blade Return Micro-Doppler Analysis

The time-frequency representation of micro-Doppler effects can reveal more information. This code constructs a time-frequency representation in the detected target range bin.

```
mf  = phased.MatchedFilter('Coefficients',mfcoeff);
ymf = mf(y);
[~,ridx] = max(sum(abs(ymf),2)); % detection via peak finding along range
pspectrum(ymf(ridx,:),prf,'spectrogram')
```
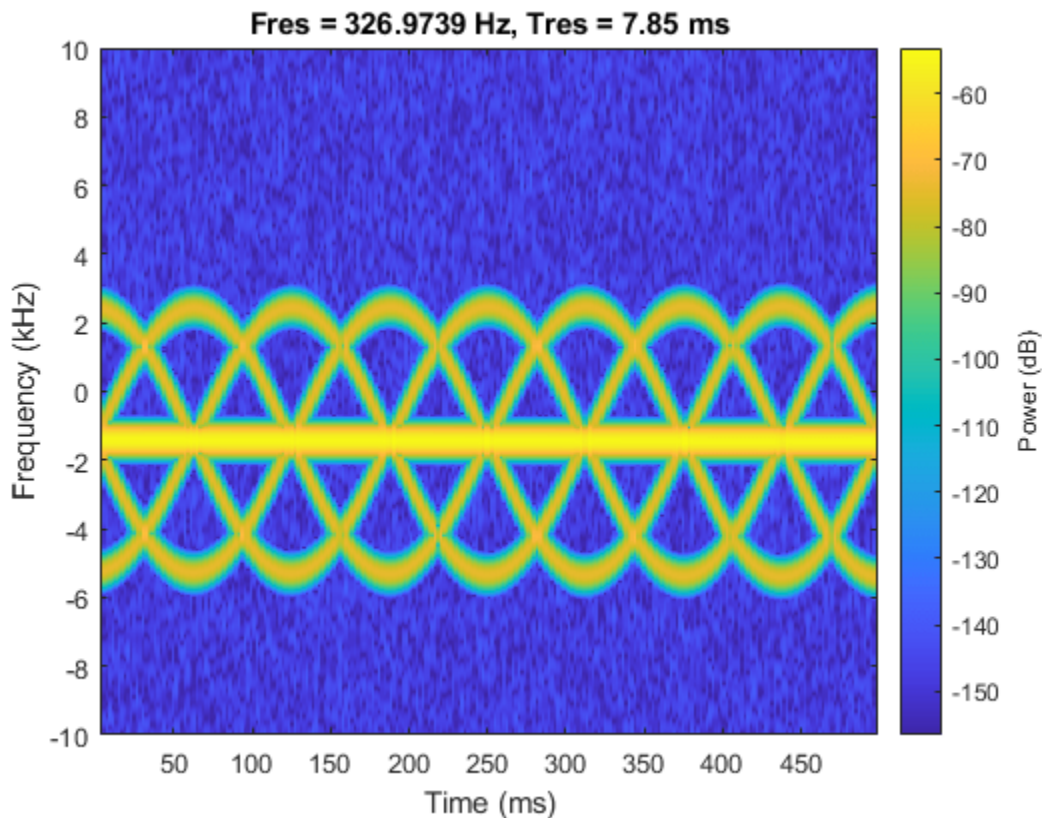
The figure shows the micro-Doppler modulation caused by blade tips around a constant Doppler shift. The image suggests that each blade tip introduces a sinusoid-like Doppler modulation. As noted in the figure below, within each period of the sinusoid, there are three extra sinusoids appearing at equal distance. This appearance suggests that the helicopter is equipped with four equally spaced blades.

```
hanno = helperAnnotateMicroDopplerSpectrogram(gcf);
```
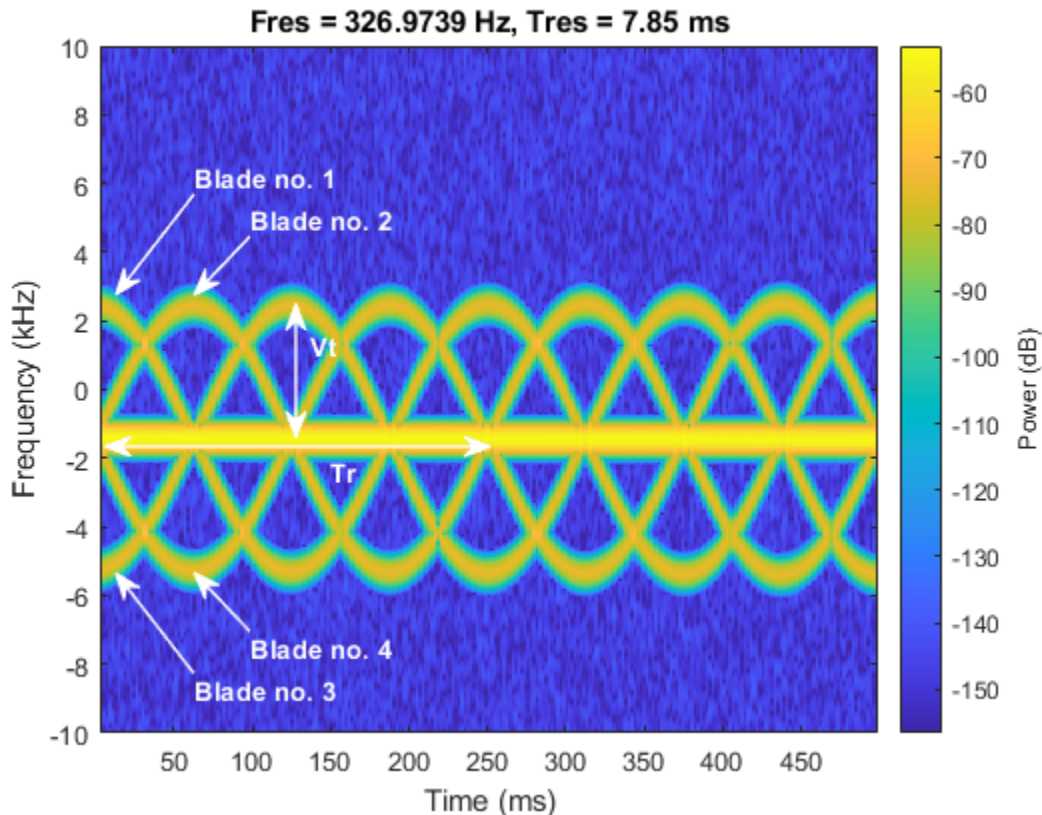

In addition to the number of blades, the image also shows that the period of each sinusoid, Tr, is about 250 ms. This value means that a blade returns to its original position after 250 ms. In this case, the angular speed of the helicopter is about 4 revolutions per second, which matches the simulation parameter.

```
Tp = 250e-3;
bladerate_est = 1/Tp
```

bladerate_est =

    4

This image also shows the tip velocity Vt, which can be derived from the maximum Doppler. The maximum Doppler is about 4 kHz away from the constant Doppler introduced by the bulk movement. Calculate the detected maximum tip velocity.

```
Vt_detect = dop2speed(4e3,c/fc)/2
```

```
Vt_detect =

   120
```

This value is the maximum tip velocity along the radial direction. To obtain the correct maximum tip velocity, the relative orientation must be taken into consideration. Because the blades are spinning in a circle, the detection is not affected by the azimuth angle. Correct only the elevation angle for the maximum tip velocity result.
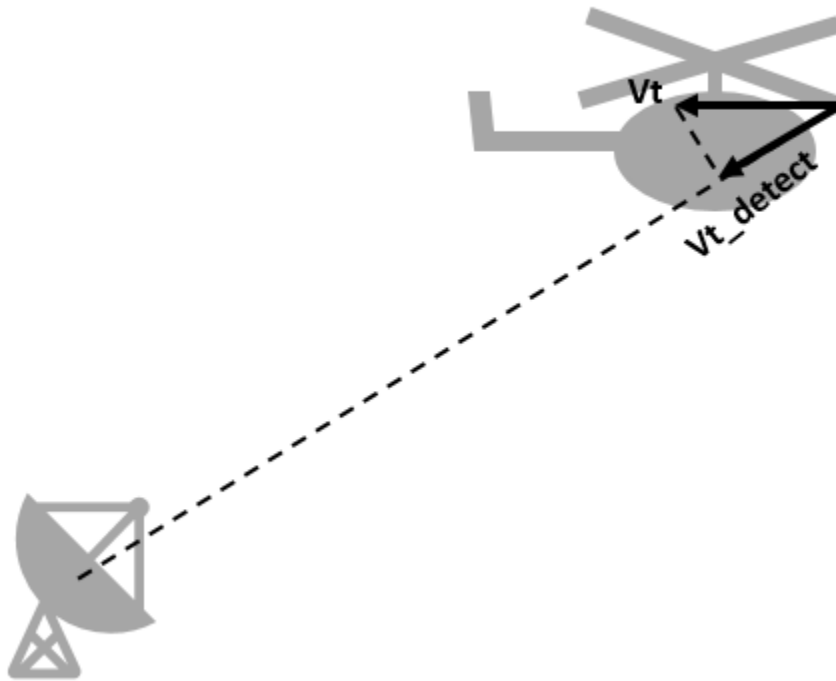

```
doa = phased.MUSICEstimator2D('SensorArray',ura,'OperatingFrequency',fc,...
    'PropagationSpeed',c,'DOAOutputPort',true,'ElevationScanAngles',-90:90);
[~,ang_est] = doa(xr);
Vt_est = Vt_detect/cosd(ang_est(2))
```

```
Vt_est =

  164.0793
```

Based on the corrected maximum tip velocity and the blade-spinning rate, calculate the blade length.

```
bladelen_est = Vt_est/(bladerate_est*2*pi)
```

```
bladelen_est =
```

```
6.5285
```

Note that the result matches the simulation parameter of 6.5 meters. Information such as number of blades, blade length, and blade rotation rate can help identify the model of the helicopter.

Pedestrian Identification in Automotive Radar

Considering an ego car with an FMCW automotive radar system whose bandwidth is 250 MHz and operates at 24 GHz.

```
bw = 250e6;
fs = bw;
fc = 24e9;
tm = 1e-6;
wav = phased.FMCWWaveform('SampleRate',fs,'SweepTime',tm,...
    'SweepBandwidth',bw);
```

The ego car is traveling along the road. Along the way, there is a car parked on the side of street and a human is walking out behind the car. The scene is illustrated in the following diagram
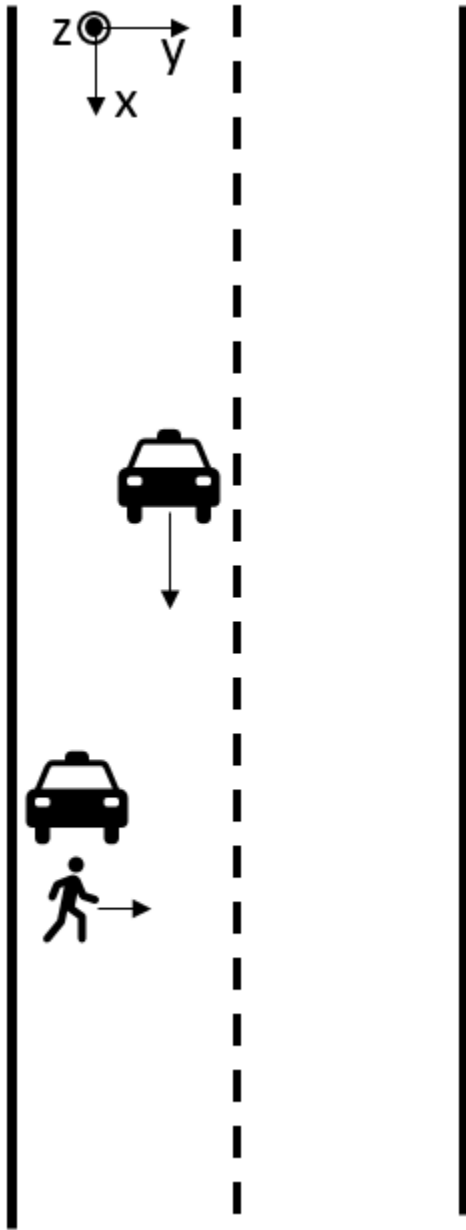
Based on this setup, if the ego car cannot identify that a pedestrian is present, an accident may occur.

```
egocar_pos = [0;0;0];
egocar_vel = [30*1600/3600;0;0];
egocar = phased.Platform('InitialPosition',egocar_pos,'Velocity',egocar_vel,...
    'OrientationAxesOutputPort',true);

parkedcar_pos = [39;-4;0];
parkedcar_vel = [0;0;0];
parkedcar = phased.Platform('InitialPosition',parkedcar_pos,'Velocity',parkedcar_vel,...
    'OrientationAxesOutputPort',true);
```

```
parkedcar_tgt = phased.RadarTarget('PropagationSpeed',c,'OperatingFrequency',fc,'MeanRCS',10);

ped_pos = [40;-3;0];
ped_vel = [0;1;0];
ped_heading = 90;
ped_height = 1.8;
ped = phased.BackscatterPedestrian('InitialPosition',ped_pos,'InitialHeading',ped_heading,...
    'PropagationSpeed',c,'OperatingFrequency',fc,'Height',1.6,'WalkingSpeed',1);

chan_ped = phased.FreeSpace('PropagationSpeed',c,'OperatingFrequency',fc,...
    'TwoWayPropagation',true,'SampleRate',fs);
chan_pcar = phased.FreeSpace('PropagationSpeed',c,'OperatingFrequency',fc,...
    'TwoWayPropagation',true,'SampleRate',fs);

tx = phased.Transmitter('PeakPower',1,'Gain',25);
rx = phased.ReceiverPreamp('Gain',25,'NoiseFigure',10);
```

Pedestrian Micro-Doppler Extraction

The following figure shows the range-Doppler map generated from the ego car's radar over time. Because the parked car is a much stronger target than the pedestrian, the pedestrian is easily shadowed by the parked car in the range-Doppler map. As a result, the map always shows a single target.
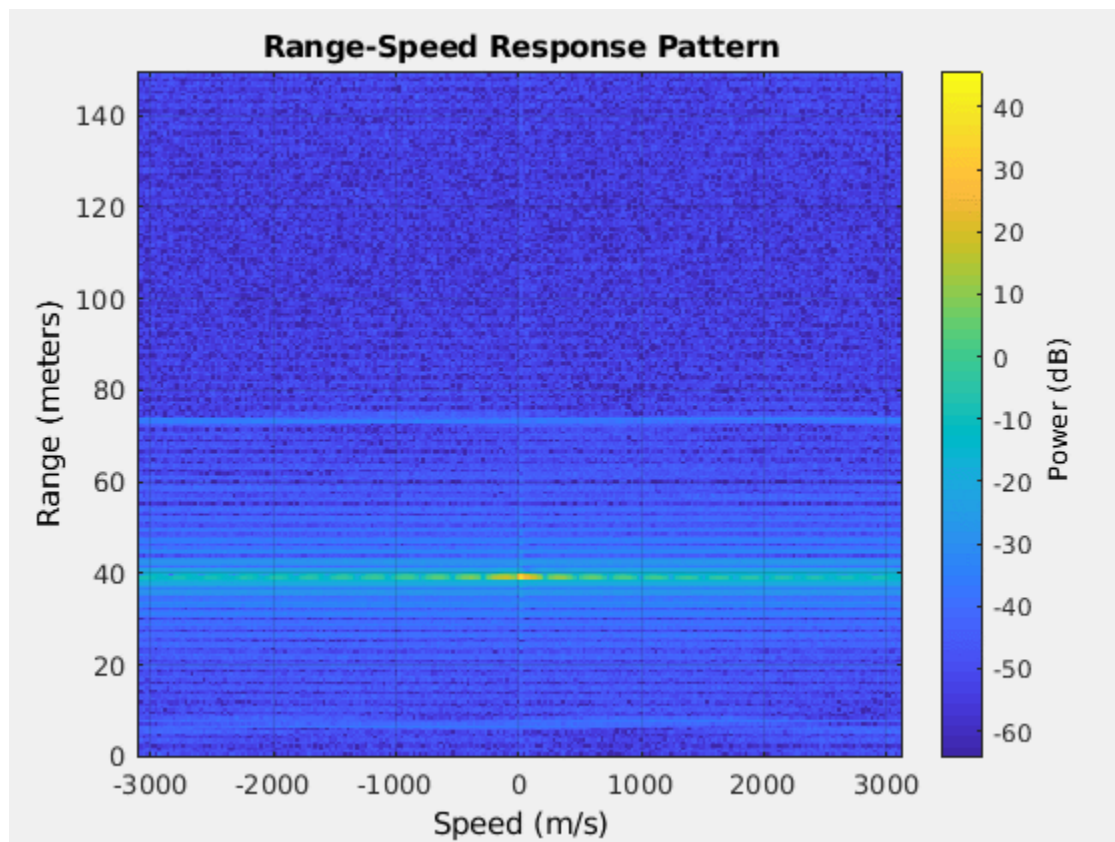
This means that conventional processing cannot satisfy our needs under this situation.

Micro-Doppler effect in time frequency domain can be a good candidate to identify if there is pedestrian signature embedded in the radar signal. As an example, the following section simulates the radar return for 2.5 seconds.

```
Tsamp = 0.001;
npulse = 2500;
xr = complex(zeros(round(fs*tm),npulse));
xr_ped = complex(zeros(round(fs*tm),npulse));
for m = 1:npulse
    [pos_ego,vel_ego,ax_ego] = egocar(Tsamp);
    [pos_pcar,vel_pcar,ax_pcar] = parkedcar(Tsamp);
    [pos_ped,vel_ped,ax_ped] = move(ped,Tsamp,ped_heading);
    [~,angrt_ped] = rangeangle(pos_ego,pos_ped,ax_ped);
    [~,angrt_pcar] = rangeangle(pos_ego,pos_pcar,ax_pcar);

    x = tx(wav());
    xt_ped = chan_ped(repmat(x,1,size(pos_ped,2)),pos_ego,pos_ped,vel_ego,vel_ped);
    xt_pcar = chan_pcar(x,pos_ego,pos_pcar,vel_ego,vel_pcar);
    xt_ped = reflect(ped,xt_ped,angrt_ped);
    xt_pcar = parkedcar_tgt(xt_pcar);
    xr_ped(:,m) = rx(xt_ped);
    xr(:,m) = rx(xt_ped+xt_pcar);
end

xd_ped = conj(dechirp(xr_ped,x));
xd = conj(dechirp(xr,x));
```

In the simulated signal, `xd_ped` contains only the pedestrian's return while `xd` contains the return from both the pedestrian and the parked car. If we generate a spectrogram using only the return of the pedestrian, we obtain a plot shown below.

```
clf;
spectrogram(sum(xd_ped),kaiser(128,10),120,256,1/Tsamp,'centered','yaxis');
clim = get(gca,'CLim');
set(gca,'CLim',clim(2)+[-50 0])
```
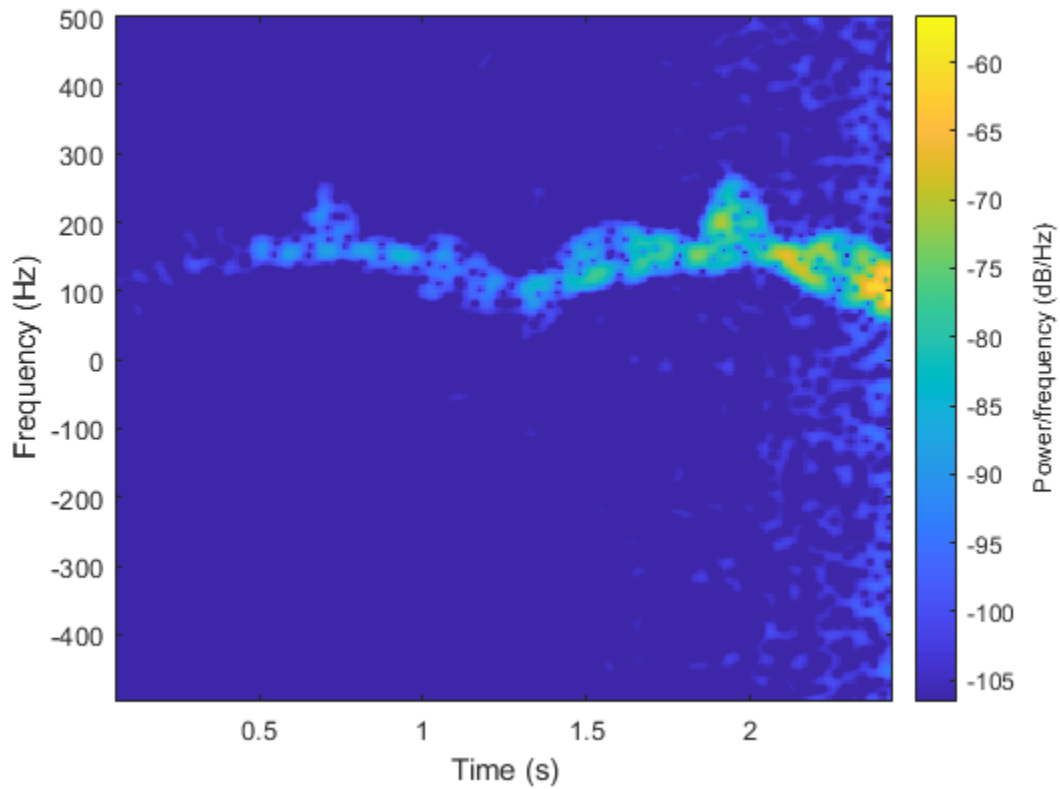
Note that the swing of arms and legs produces many parabolic curves in the time frequency domain along the way. Therefore such features can be used to determine whether a pedestrian exists in the scene.

However, when we generate a spectrogram directly from the total return, we get the following plot.

```
spectrogram(sum(xd),kaiser(128,10),120,256,1/Tsamp,'centered','yaxis');
clim = get(gca,'CLim');
set(gca,'CLim',clim(2)+[-50 0])
```
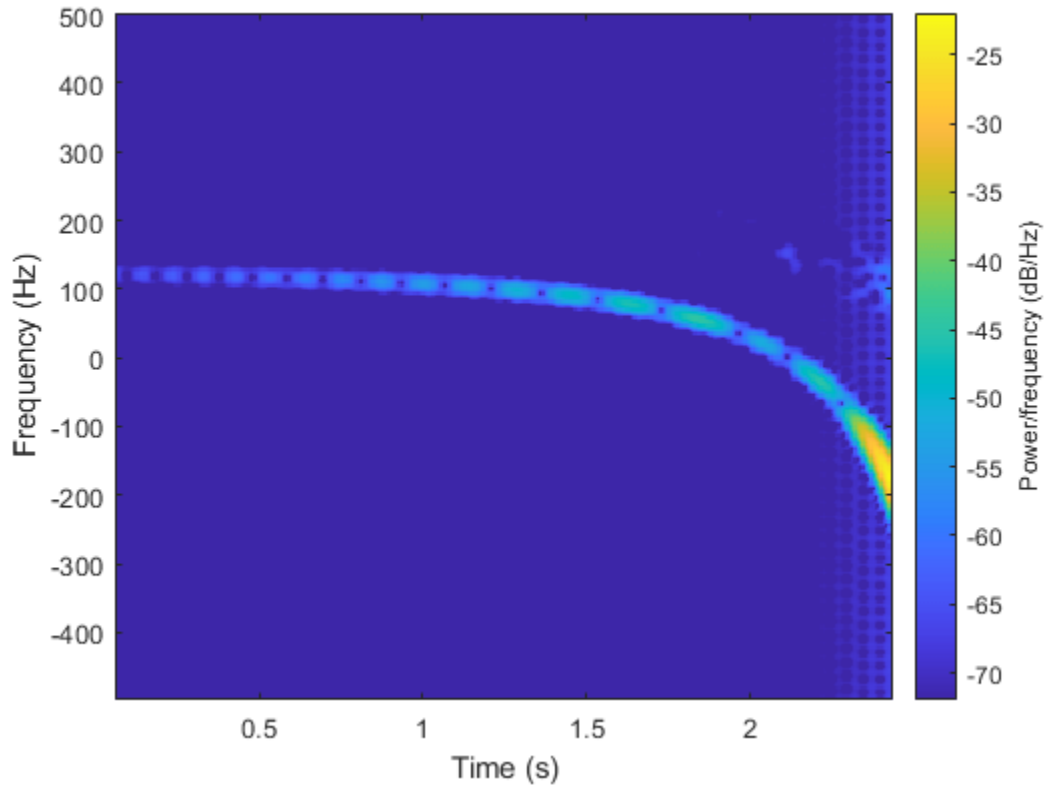
What we observe is that the parked car's return continue dominating the return, even in the time frequency domain. Therefore the time frequency response shows only the Doppler relative to the parked car. The drop of the Doppler frequency is due to the ego car getting closer to the parked car and the relative speed drops towards 0.

To see if there is a return hidden behind the strong return, we can use the singular value decomposition. The following plot shows the distribution of the singular value of the dechirped pulses.

```
[uxd,sxd,vxd] = svd(xd);
clf
plot(10*log10(diag(sxd)));
xlabel('Rank');
ylabel('Singular Values');
hold on;
plot([56 56],[-40 10],'r--');
plot([100 100],[-40 10],'r--');
plot([110 110],[-40 10],'r--');
text(25,-10,'A');
text(75,-10,'B');
text(105,-10,'C');
text(175,-10,'D');
```
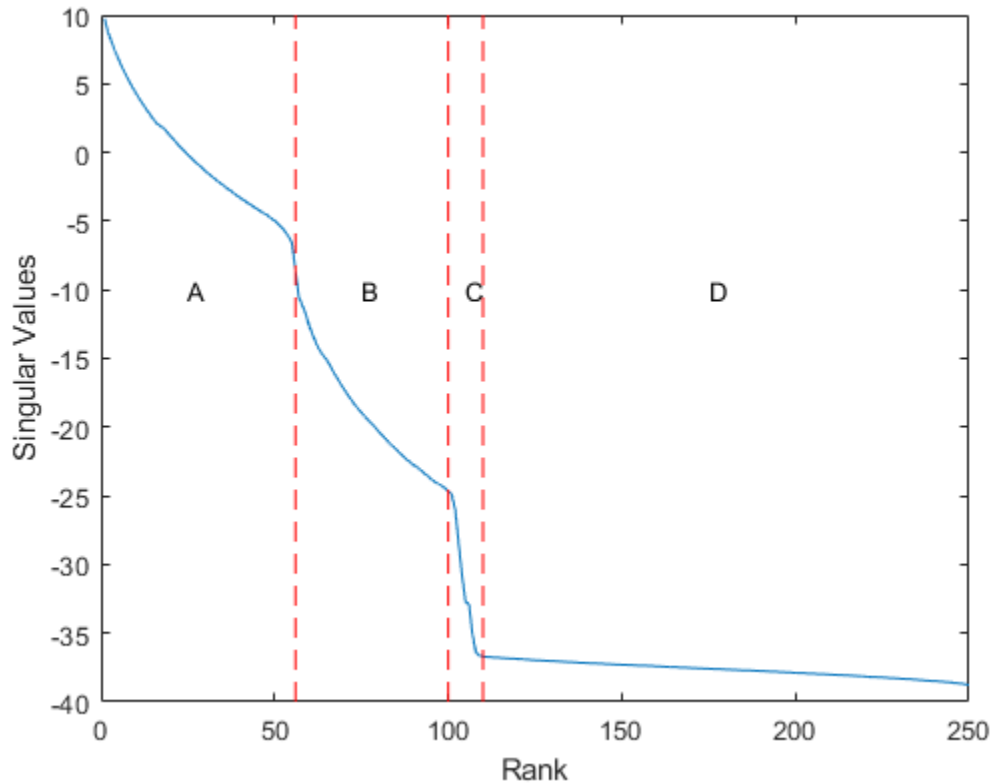
From the curve, it is clear that there are approximately four regions. The region A represents the most significant contribution to the signal, which is the parked car. The region D represents the noise. Therefore, the region B and C are due to the mix of parked car return and the pedestrian return. Because the return from the pedestrian is much weaker than the return from the parked car. In region B, it can still be masked by the residue of the return from the parked car. Therefore, we pick the region C to reconstruct the signal, and then plot the time frequency response again.

```
rk = 100:110;
xdr = uxd(:,rk)*sxd(rk,:)*vxd';
clf
spectrogram(sum(xdr),kaiser(128,10),120,256,1/Tsamp,'centered','yaxis');
clim = get(gca,'CLim');
set(gca,'CLim',clim(2)+[-50 0])
```
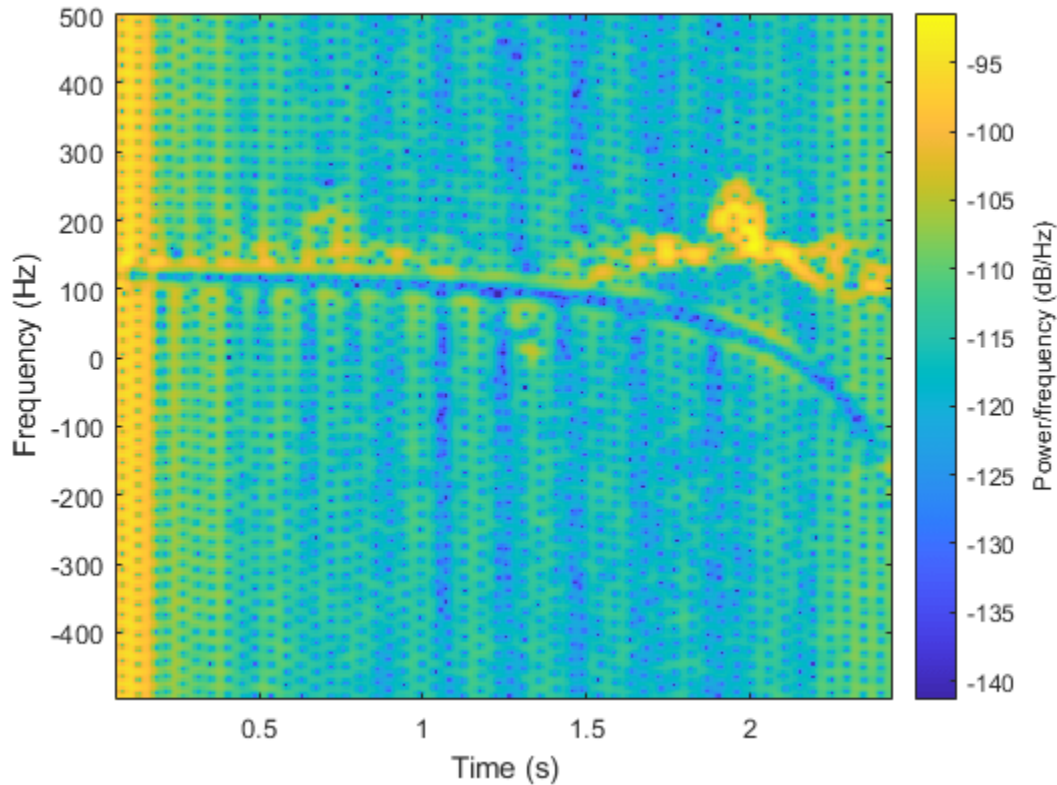
With the return from the car successfully filtered, the micro-Doppler signature from the pedestrian appears. Therefore, we can conclude that there is pedestrian in the scene and act accordingly to avoid an accident.

Summary

This example introduces the basic concept of a micro-Doppler effect and shows its impact on the target return. It also shows how to extract a micro-Doppler signature from the received I/Q signal and then derive relevant target parameters from the micro-Doppler information.

References

[1] Chen, V. C., *The Micro-Doppler Effect in Radar*, Artech House, 2011

[2] Chen, V. C., F. Li, S.-S. Ho, and H. Wechsler, "Micro-Doppler Effect in Radar: Phenomenon, Model, and Simulation Study", *IEEE Transactions on Aerospace and Electronic Systems*, Vol 42, No. 1, January 2006

Utility Functions

Function `helicopmotion` models the motion of multiple scatterers of the helicopter.

```
function [scatterpos,scattervel,scatterang] = helicopmotion(...
    t,tgtmotion,BladeAng,ArmLength,BladeRate)

prf = 2e4;
radarpos = [0;0;0];
```

```
Nblades  = size(BladeAng,2);

[tgtpos,tgtvel] = tgtmotion(1/prf);

RotAng     = BladeRate*t;
scatterpos = [0 ArmLength*cos(RotAng+BladeAng);0 ArmLength*sin(RotAng+BladeAng);zeros(1,Nblades+1
scattervel = [0 -BladeRate*ArmLength*sin(RotAng+BladeAng);...
    0 BladeRate*ArmLength*cos(RotAng+BladeAng);zeros(1,Nblades+1)]+tgtvel;

[~,scatterang] = rangeangle(scatterpos,radarpos);

end
```

Ground Clutter Mitigation with Moving Target Indication (MTI) Radar

This example shows the design of a moving target indication (MTI) radar to mitigate clutter and identify moving targets. For a radar system, *clutter* refers to the received echoes from environmental scatters other than targets, such as land, sea or rain. Clutter echoes can be many orders of magnitude larger than target echoes. An MTI radar exploits the relatively high Doppler frequencies of moving targets to suppress clutter echoes, which usually have zero or very low Doppler frequencies.

A typical MTI radar uses a high-pass filter to remove energy at low Doppler frequencies. Since the frequency response of an FIR high-pass filter is periodic, some energy at high Doppler frequencies is also removed. Targets at those high Doppler frequencies thus will not be detectable by the radar. This issue is called the *blind speed problem*. This example shows how to use a technique, called staggered pulse-repetition frequencies (PRFs), to address the blind speed problem.

Construct a Radar System

First, define the components of a radar system. The focus of this example is on MTI processing, so the example uses the radar system built in the example "Simulating Test Signals for a Radar Receiver". You can explore the details of the radar system design through that example. Change the antenna height to 100 meters to simulate a radar mounted on top of a building. Notice that the PRF in the system is approximately 30 kHz, which corresponds to a maximum unambiguous range of 5 km.

```
load BasicMonostaticRadarExampleData;
sensorheight = 100;
sensormotion.InitialPosition = [0 0 sensorheight]';
prf = waveform.PRF;
```

Retrieve the sampling frequency, the operating frequency, and the propagation speed.

```
fs = waveform.SampleRate;
fc = radiator.OperatingFrequency;
wavespeed = radiator.PropagationSpeed;
```

In many MTI systems, especially low-end ones, the power source for the transmitter is a magnetron. Thus, the transmitter adds a random phase to each transmitted pulse. Hence, it is often necessary to restore the coherence at the receiver. Such setup is referred to as *coherent on receive*. In these systems, the receiver locks onto the random phases added by transmitter for each pulse. Then, the receiver removes the phase impact from the samples received within the corresponding pulse interval. Simulate a coherent on receive system by setting the transmitter and receiver as follows.

```
transmitter.CoherentOnTransmit = false;
transmitter.PhaseNoiseOutputPort = true;
receiver.PhaseNoiseInputPort = true;
```

Define Targets

Next, define two moving targets.

The first target is located at the position [1600 0 1300]. Given the radar position shown in the preceding sections, it has a range of 2 km from the radar. The velocity of the target is [100 80 0], corresponding to a radial speed of -80 m/s relative to the radar. The target has a radar cross section of 25 square meters.

The second target is located at position [2900 0 800], corresponding to a range of 3 km from the radar. Set the speed of this target to a blind speed, where the Doppler signature is aliased to the PRF. This setting prevents the MTI radar from detecting the target. Use the `dop2speed` function to calculate the blind speed which has a corresponding Doppler frequency equal to the PRF.

```
wavelength = wavespeed/fc;
blindspd = dop2speed(prf,wavelength)/2; % half to compensate round trip

tgtpos = [[1600 0 1300]',[2900 0 800]'];
tgtvel = [[100 80 0]',[-blindspd 0 0]'];
tgtmotion = phased.Platform('InitialPosition',tgtpos,'Velocity',tgtvel);

tgtrcs = [25 25];
target = phased.RadarTarget('MeanRCS',tgtrcs,'OperatingFrequency',fc);
```

Clutter

The clutter signal was generated using the simplest clutter model, the constant gamma model, with the gamma value set to -20 dB. Such a gamma value is typical for flatland clutter. Assume that the clutter patches exist at all ranges, and that each patch has an azimuth width of 10 degrees. Also assume that the main beam of the radar points horizontally. Note that the radar is not moving.

```
trgamma = surfacegamma('flatland');

clutter = constantGammaClutter('Sensor',antenna,...
    'PropagationSpeed',radiator.PropagationSpeed,...
    'OperatingFrequency',radiator.OperatingFrequency,...
    'SampleRate',waveform.SampleRate,'TransmitSignalInputPort',true,...
    'PRF',waveform.PRF,'Gamma',trgamma,'PlatformHeight',sensorheight,...
    'PlatformSpeed',0,'PlatformDirection',[0;0],...
    'MountingAngles',[0 0 0],'ClutterMaxRange',5000,...
    'ClutterAzimuthSpan',360,'PatchAzimuthSpan',10,...
    'SeedSource','Property','Seed',2011);
```

Simulate the Received Pulses and Matched Filter

Simulate 10 received pulses for the radar and targets defined earlier.

```
pulsenum = 10;

% Set the seed of the receiver to duplicate results
receiver.SeedSource = 'Property';
receiver.Seed = 2010;

rxPulse = helperMTISimulate(waveform,transmitter,receiver,...
    radiator,collector,sensormotion,...
    target,tgtmotion,clutter,pulsenum);
```

Then pass the received signal through a matched filter.

```
matchingcoeff = getMatchedFilter(waveform);
matchedfilter = phased.MatchedFilter('Coefficients',matchingcoeff);
mfiltOut = matchedfilter(rxPulse);

matchingdelay = size(matchingcoeff,1)-1;
mfiltOut = buffer(mfiltOut(matchingdelay+1:end),size(mfiltOut,1));
```

Perform MTI Processing Using a Three-Pulse Canceler

MTI processing uses MTI filters to remove low-frequency components in slow-time sequences. Because land clutter usually is not moving, removing low-frequency components can effectively suppress it. The three-pulse canceler is a popular and simple MTI filter. The canceler is an all-zero FIR filter with filter coefficients [1 -2 1].

```
h = [1 -2 1];
mtiseq = filter(h,1,mfiltOut,[],2);
```

Use noncoherent pulse integration to combine the slow time sequences. Exclude the first two pulses because they are in the transient period of the MTI filter.

```
mtiseq = pulsint(mtiseq(:,3:end));
% For comparison, also integrate the matched filter output
mfiltOut = pulsint(mfiltOut(:,3:end));

% Calculate the range for each fast time sample
fast_time_grid = (0:size(mfiltOut,1)-1)/fs;
rangeidx = wavespeed*fast_time_grid/2;

% Plot the received pulse energy again range
plot(rangeidx,pow2db(mfiltOut.^2),'r--',...
    rangeidx,pow2db(mtiseq.^2),'b-' ); grid on;
title('Fast Time Sequences Using a Uniform PRF');
xlabel('Range (m)'); ylabel('Power (dB)');
legend('Before MTI filter','After MTI filter');
```
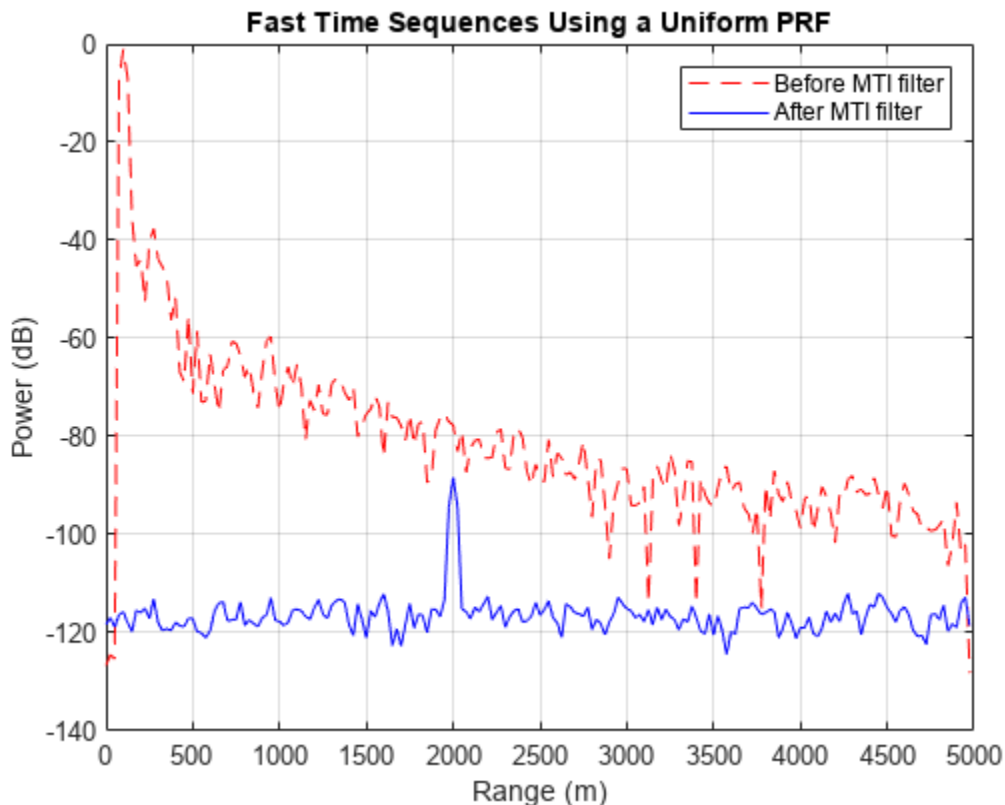
Recall that there are two targets (at 2 km and 3 km). In the case before MTI filtering, both targets are buried in clutter returns. The peak at 100 m is the direct path return from the ground right below the radar. Notice that the power is decreasing as the range increases, which is due to the signal propagation loss.

After MTI filtering, most clutter returns are removed except for the direct path peak. The noise floor is now no longer a function of range, so the noise is now receiver noise rather than clutter noise. This change shows the clutter suppression capability of the three-pulse canceler. At the 2 km range, you can see a peak representing the first target. However, there is no peak at 3 km range to represent the second target. The peak disappears because the three-pulse canceler suppresses the second target that travels at the blind speed of the canceler.

To better understand the blind speed problem, look at the frequency response of the three-pulse canceler.

```
f = linspace(0,prf*9,1000);
hresp = freqz(h,1,f,prf);
plot(f/1000,20*log10(abs(hresp)));
grid on; xlabel('Doppler Frequency (kHz)'); ylabel('Magnitude (dB)');
title('Frequency Response of the Three-Pulse Canceler');
```
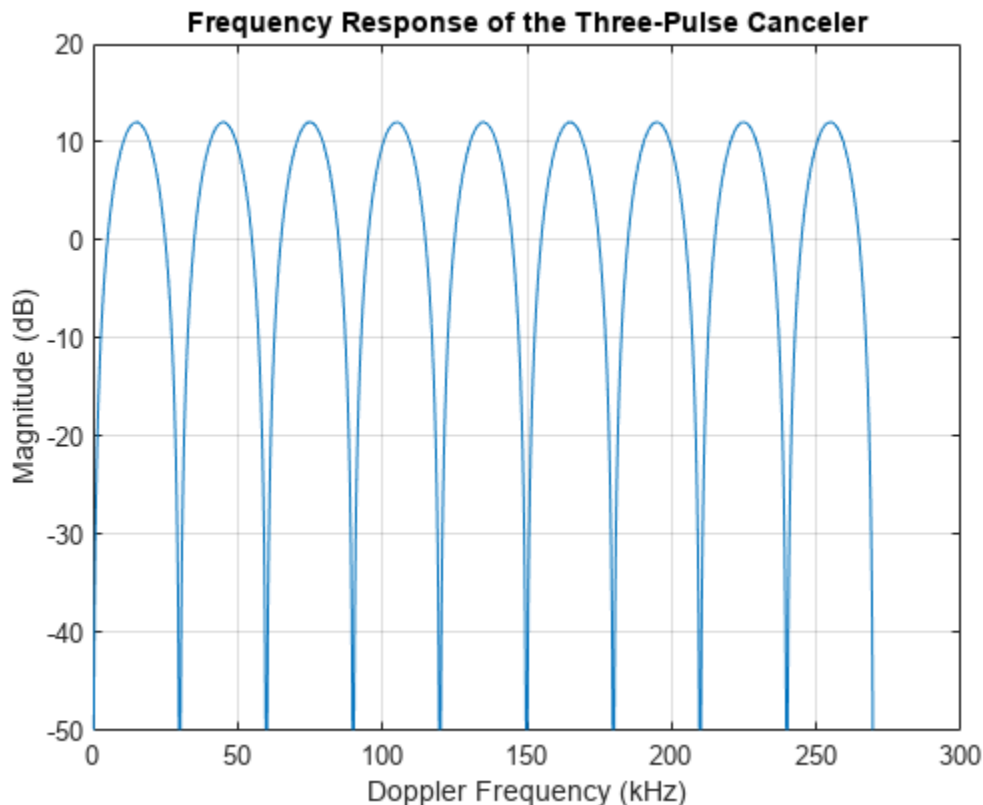

Notice the recurring nulls in the frequency response. The nulls correspond to the Doppler frequencies of the blind speeds. Targets with these Doppler frequencies are canceled by the three-pulse canceler. The plot shows that the nulls occur at integer multiples of the PRF (approximately 30kHz, 60kHz, and so on). If these nulls can be removed or pushed away from the Doppler frequency region of the radar specifications, the blind speed problem can be avoided.

Simulate the Received Pulses Using Staggered PRFs

One solution to the blind speed problem is to use nonuniform PRFs (staggered PRFs). Adjacent pulses are transmitted at different PRFs. Such a configuration pushes the lower bound of blind speeds to a higher value. To illustrate this idea, this example uses a two-staggered PRF, and then plots the frequency response of the three-pulse canceler.

Choose a second PRF at around 25kHz, which corresponds to a maximum unambiguous range of 6 km.

```
prf = wavespeed./(2*[6000 5000]);
```

```
% Calculate the magnitude frequency response for the three-pulse canceler
pf1 = @(f)(1-2*exp(1j*2*pi/prf(1)*f)+exp(1j*2*pi*2/prf(1)*f));
pf2 = @(f)(1-2*exp(1j*2*pi/prf(2)*f)+exp(1j*2*pi*2/prf(2)*f));
sfq = (abs(pf1(f)).^2 + abs(pf2(f)).^2)/2;
```

```
% Plot the frequency response
hold on;
plot(f/1000,pow2db(sfq),'r--');
ylim([-50, 30]);
legend('Uniform PRF','Two-Staggered PRF');
```
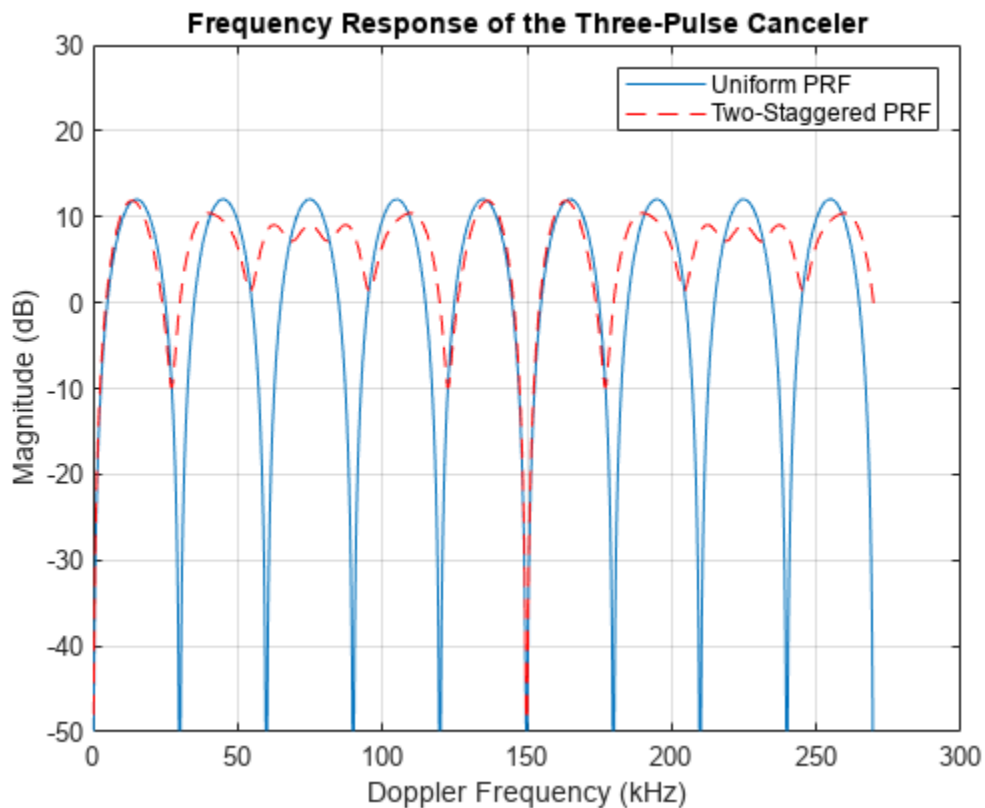

From the plot of the staggered PRFs you can see that the first blind speed corresponds to a Doppler frequency of 150 kHz, five times larger than the uniform-PRF case. Thus the target with the 30 kHz Doppler frequency will not be suppressed.

Now, simulate the reflected signals from the targets using the staggered PRFs.

```
% Assign the new PRF
release(waveform);
waveform.PRF = prf;

release(clutter);
clutter.PRF = prf;

% Reset noise seed
release(receiver);
receiver.Seed = 2010;

% Reset platform position
reset(sensormotion);
reset(tgtmotion);

% Simulate target return
rxPulse = helperMTISimulate(waveform,transmitter,receiver,...
    radiator,collector,sensormotion,...
    target,tgtmotion,clutter,pulsenum);
```

Perform MTI Processing for Staggered PRFs

Process the pulses as before by first passing them through a matched filter and then integrating the pulses noncoherently.

```
mfiltOut = matchedfilter(rxPulse);
% Use the same three-pulse canceler to suppress the clutter.
mtiseq = filter(h,1,mfiltOut,[],2);

% Noncoherent integration
mtiseq = pulsint(mtiseq(:,3:end));
mfiltOut = pulsint(mfiltOut(:,3:end));

% Calculate the range for each fast time sample
fast_time_grid = (0:size(mfiltOut,1)-1)/fs;
rangeidx = wavespeed*fast_time_grid/2;

% Plot the fast time sequences against range.
clf;
plot(rangeidx,pow2db(mfiltOut.^2),'r--',...
    rangeidx,pow2db(mtiseq.^2),'b-' ); grid on;
title('Fast Time Sequences Using Staggered PRFs');
xlabel('Range (m)'); ylabel('Power (dB)');
legend('Before MTI filter','After MTI filter');
```
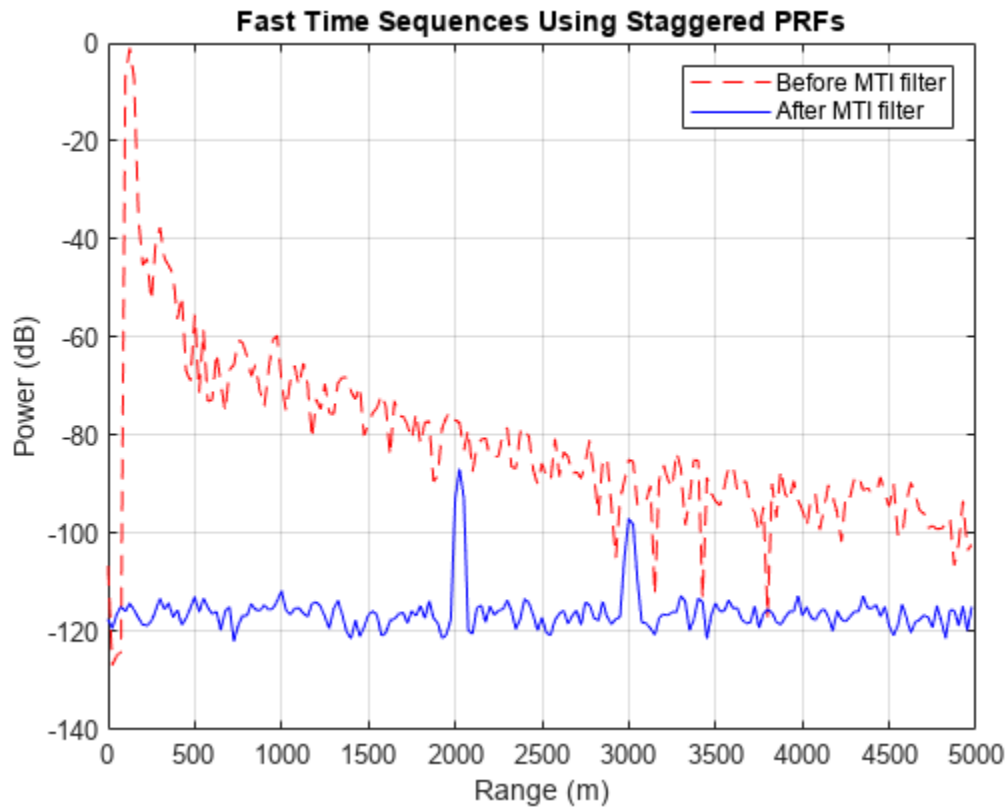
The plot shows both targets are now detectable after MTI filtering, and the clutter is also removed.

Summary

With very simple operations, MTI processing can be effective at suppressing low-speed clutter. A uniform-PRF waveform will miss targets at blind speeds, but this issue can be addressed by using staggered PRFs. For clutter that has a large Doppler bandwidth, MTI performance could be poor. Such clutter can be suppressed using space-time adaptive processing. For more details, see the example "Introduction to Space-Time Adaptive Processing".

References

[1] Richards, M. A. *Fundamentals of Radar Signal Processing*. New York: McGraw-Hill, 2005.

Simulating a Polarimetric Radar Return for Weather Observation

This example shows how to simulate a polarimetric Doppler radar return that meets the requirements of weather observations. Radar plays a critical role in weather observation, detection of hazards, classification and quantification of precipitation, and forecasting. In addition, polarimetric radar provides multiparameter measurements with unprecedented quality and information. This example shows how to simulate a polarimetric Doppler radar that scans an area of distributed weather targets. The simulation derives the radar parameters according to the well-known NEXRAD radar specifications. After synthesizing the received pulses, radar spectral moment estimation and polarimetric moment estimation are performed. The estimates are compared with NEXRAD ground truth, from which error statistics are obtained and data quality is evaluated.

Radar Definition

A well-known weather radar is the Weather Surveillance Radar, 1988 Doppler (WSR-88D), also known as NEXRAD, which is operated by the US National Weather Service, FAA and DoD. For more information, see the NEXRAD Radar Operations Center website.

Radar system specifications are designed as follows.

```matlab
max_range = 100e3;                       % Maximum unambiguous range (m)
range_res = 250;                         % Required range resolution (m)
pulnum = 32;                             % Number of pulses to process in an azimuth
fc = 2800e6;                             % Frequency (Hz)
prop_speed = physconst('LightSpeed');    % Propagation speed (m/s)
lambda = prop_speed/fc;                  % Wavelength (m)
```

To translate these requirements to radar parameters, we follow the process within the example "Simulating Test Signals for a Radar Receiver". In this example, for the sake of simplicity, load precalculated radar parameters.

```matlab
load NEXRAD_Parameters.mat
```

Antenna Pattern

As NEXRAD is polarimetric, modeling the polarimetric characteristics of the antenna and weather targets is important. According to NEXRAD specifications, the antenna pattern has a beamwidth of about 1 degree and first sidelobe below -30 dB.

```matlab
azang = [-180:0.5:180];
elang = [-90:0.5:90];
% We synthesize a pattern using isotropic antenna elements and tapering the
% amplitude distribution to make it follow NEXRAD specifications.
magpattern = load('NEXRAD_pattern.mat');
phasepattern = zeros(size(magpattern.pat));
% The polarimetric antenna is assumed to have ideally matched horizontal
% and vertical polarization pattern.
antenna = phased.CustomAntennaElement('AzimuthAngles',azang,...
    'ElevationAngles',elang,...
    'HorizontalMagnitudePattern',magpattern.pat,...
    'VerticalMagnitudePattern',magpattern.pat,...
    'HorizontalPhasePattern',phasepattern,...
    'VerticalPhasePattern',phasepattern,...
```

```
    'SpecifyPolarizationPattern',true);

clear magpattern
clear phasepattern
```

Plot the azimuth cut of the antenna pattern.

```
D = pattern(antenna,fc,azang,0);
P = polarpattern(azang,D,'TitleTop','Polar Pattern for Azimuth Cut (elevation angle = 0 degree)')
P.AntennaMetrics = 1;
removeAllCursors(P);
```


Polar Pattern for Azimuth Cut (elevation angle = 0 degree)
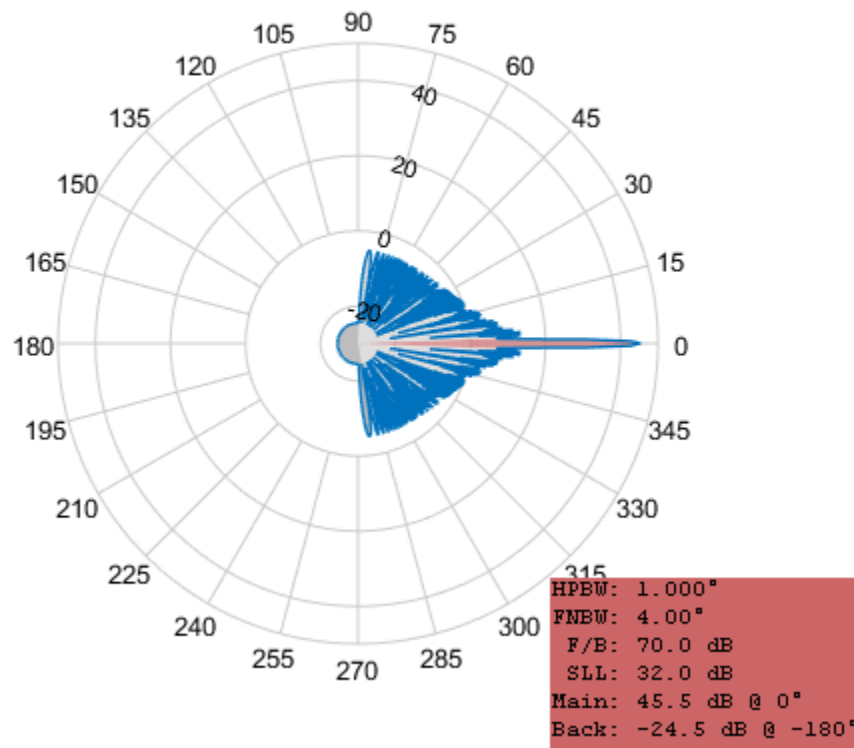
Associate the array with the radiator and collector.

```
radiator = phased.Radiator(...
    'Sensor',antenna,'Polarization','Dual',...
    'OperatingFrequency',fc);

collector = phased.Collector(...
    'Sensor',antenna,'Polarization','Dual',...
    'OperatingFrequency',fc);
```

Weather Target

Generally, weather radar data is categorized into three levels. Level-I data is raw time series I/Q data as input to the signal processor in the Radar Data Acquisition unit. Level-II data consists of the radar spectral moments (reflectivity, mean radial velocity, and spectrum width) and polarimetric moments (differential reflectivity, correlation coefficient, and differential phase) output from the signal

1-533

processor. Level-III data is the output product data of the radar product generator, such as hydrometeor classification, storm total precipitation, and tornadic vortex signature.

In this example, Level-II data from KTLX NEXRAD radar at 20:08:11 UTC on May 20th, 2013 is used. This data comes from an intense tornado that occurred in Moore, Oklahoma and is used to generate mean radar cross section (RCS) of *equivalent scattering centers*. The data is available via FTP download. It represents a volume scan that includes a series of 360-degree sweeps of the antenna at predetermined elevation angles completed in a specified period of time. In the data file name *KTLX20130520_200811_V06*, *KTLX* refers to the radar site name, *20130520_200811* refers to the date and time when the data was collected, and *V06* refers to the data format of version 6. In this simulation, the lowest elevation cut (0.5 degree) is extracted from the volume scan data as an example.

Read the Level-II data into the workspace. Store it in the *nexrad* structure array, which contains all the radar moments as well as an azimuth field that specifies the azimuth angle for each radial data point in the Cartesian coordinate system. For simplicity, load NEXRAD data that was transformed from a compressed file to a MAT-file.

```
load NEXRAD_data.mat;
```

Define an area of interest (AOI) in terms of azimuth and range in Cartesian coordinates.

```
az1 = 96;       % Starting azimuth angle (degree)
az2 = 105;      % Ending azimuth angle (degree)
rg1 = 22750;    % Starting range (m)
rg2 = 38750;    % Ending range (m)
% NEXRAD has a blind range of 2000m.
blind_rg = 2000;
% We define the number of azimuth angles that weather targets occupy in the
% two dimensional plane.
num_az = 40;
% We define the number of range bins that weather targets occupy in each
% azimuth radial of the two dimensional plane.
num_bin = 150;
% Select AOI data and store it in _nexrad_aoi_ structure array, which
% contains all the radar moments, as well as starting and ending azimuth
% and range indices. And the number of available weather targets in space
% is returned as Ns.
[nexrad_aoi,Ns] = helperSelectAOI(nexrad,az1,az2,rg1,rg2,blind_rg,range_res,num_az,num_bin);
```

Because weather targets are polarimetric and distributed in a plane, they can be represented by specifying scattering matrices at discrete azimuth angles. Weather target reflectivity is defined as the mean backscattering cross section per unit volume. Based on the weather radar equation, weather targets can be considered as a collection of small targets within each resolution volume. The overall reflectivity can be transformed to the mean RCS and regarded as an equivalent scattering center. As a result, each element in the scattering matrix is the square root of RCS in relevant polarization.

```
% Preallocate target position, velocity, RCS, azimuth, elevation, and radar
% scattering matrices.
tgtpos = zeros(3, Ns);
tgtvel = zeros(3, Ns);
RCSH = zeros(1,Ns);
RCSV = zeros(1,Ns);
azpatangs = [-180 180];
elpatangs = [-90 90];
shhpat = zeros(2,2,Ns);
```

```
svvpat = zeros(2,2,Ns);
shvpat = zeros(2,2,Ns);
zz = 0;
% NEXRAD beamwidth is about 1 degree.
beamwidth = 1.0;
for ii = nexrad_aoi.rlow:nexrad_aoi.rup
    theta = nexrad.azimuth(ii);
    for jj = 1:num_bin
        if isnan(nexrad.ZH(ii,jj))==0
            zz = zz+1;
            rpos = (jj-1)*range_res + blind_rg;
            tpos = [rpos*cosd(theta);rpos*sind(theta);0];
            tgtpos(:,zz) = tpos;
            RCSH(zz) = helperdBZ2RCS(beamwidth,rpos,lambda,pulse_width,nexrad.ZH(ii,jj),prop_spee
            shhpat(:,:,zz) = sqrt(RCSH(zz))*ones(2,2);
            RCSV(zz) = helperdBZ2RCS(beamwidth,rpos,lambda,pulse_width,nexrad.ZV(ii,jj),prop_spee
            svvpat(:,:,zz) = sqrt(RCSV(zz))*ones(2,2);
        end
    end
end

tgtmotion = phased.Platform('InitialPosition',tgtpos,'Velocity',tgtvel);
target = phased.BackscatterRadarTarget('EnablePolarization',true,...
    'Model','Nonfluctuating','AzimuthAngles',azpatangs,...
    'ElevationAngles',elpatangs,'ShhPattern',shhpat,'ShvPattern',shvpat,...
    'SvvPattern',svvpat,'OperatingFrequency',fc);
```

Radar Pulse Synthesis

Generate a radar data cube using the defined radar system parameters. Within each resolution volume, include the appropriate correlation to ensure the resulting I/Q data presents proper weather signal statistical properties.

```
rxh_aoi = complex(zeros(nexrad_aoi.rgnum,nexrad_aoi.aznum));
rxv_aoi = complex(zeros(nexrad_aoi.rgnum,nexrad_aoi.aznum));
% The number of realization sequences
realiznum = 1000;
% The number of unusable range bins due to NEXRAD blind range
i0 = blind_rg/range_res;
% Rotate sensor platform to simulate NEXRAD scanning in azimuth
for kk = 1:nexrad_aoi.aznum

    axes = rotz(nexrad.azimuth(kk+nexrad_aoi.r1-1));
    % Update sensor and target positions
    [sensorpos,sensorvel] = sensormotion(1/prf);
    [tgtpos,tgtvel] = tgtmotion(1/prf);

    % Calculate the target angles as seen by the sensor
    [tgtrng,tgtang] = rangeangle(tgtpos,sensorpos,axes);

    % Form transmit beam for this scan angle and simulate propagation
    pulse = waveform();
    [txsig,txstatus] = transmitter(pulse);
    % Adopt simultaneous transmission and reception mode as NEXRAD
    txsig = radiator(txsig,txsig,tgtang,axes);
    txsig = channel(txsig,sensorpos,tgtpos,sensorvel,tgtvel);

    % Reflect pulse off of targets
```

```
    ang_az = tgtang(1:2:end);
    ang_az = ang_az+(-1).^(double(ang_az>0))*180;
    tgtsig = target(txsig,[ang_az;zeros(size(ang_az))],axes);

    % Collect the target returns received at the sensor
    [rxsig_h,rxsig_v] = collector(tgtsig,tgtang,axes);
    rxh = receiver(rxsig_h,~(txstatus>0));
    rxv = receiver(rxsig_v,~(txstatus>0));

    % Matched filtering
    [rxh, mfgainh] = matchedfilter(rxh);
    [rxv, mfgainv] = matchedfilter(rxv);
    rxh = [rxh(matchingdelay+1:end);zeros(matchingdelay,1)];
    rxv = [rxv(matchingdelay+1:end);zeros(matchingdelay,1)];

    % Decimation
    rxh = rxh(1:2:end);
    rxv = rxv(1:2:end);

    % Discard blind range data and select AOI data
    rxh_aoi(:,kk) = rxh(nexrad_aoi.b1+i0:nexrad_aoi.b2+i0);
    rxv_aoi(:,kk) = rxv(nexrad_aoi.b1+i0:nexrad_aoi.b2+i0);
end

clear txsig
clear tgtsig
```

Weather Radar Moment Estimation

Using pulse pair processing, calculate all the radar moments from estimates of correlations, including reflectivity, mean radial velocity, spectrum width, differential reflectivity, correlation coefficient, and differential phase.

```
moment = helperWeatherMoment(rxh_aoi,rxv_aoi,nexrad_aoi,pulnum,realiznum,prt,lambda);
```

Simulation Result

Compare the simulation result with the NEXRAD ground truth. Evaluate the simulated data quality using error statistics, a sector image, a range profile, and a scatter plot. Error statistics are expressed as the bias and standard deviation of the estimated radar moments compared to the NEXRAD Level-II data (truth fields).

Define the azimuth and range for plotting.

```
azimuth = nexrad.azimuth(nexrad_aoi.r1:nexrad_aoi.r2);
range = (nexrad_aoi.b1-1:nexrad_aoi.b2-1)*250 + 2000;
```

Reflectivity

Reflectivity, Z, is the zeroth moment of the Doppler spectrum and is related to liquid water content or precipitation rate in the resolution volume. Because values of Z that are commonly encountered in weather observations span many orders of magnitude, radar meteorologists use a logarithmic scale given by $10log_{10}Z$ as dBZ, where Z is in units of mm^6/m^3.

```
[Z_bias,Z_std] = helperDataQuality(nexrad_aoi,moment,range,azimuth,'Z');
```
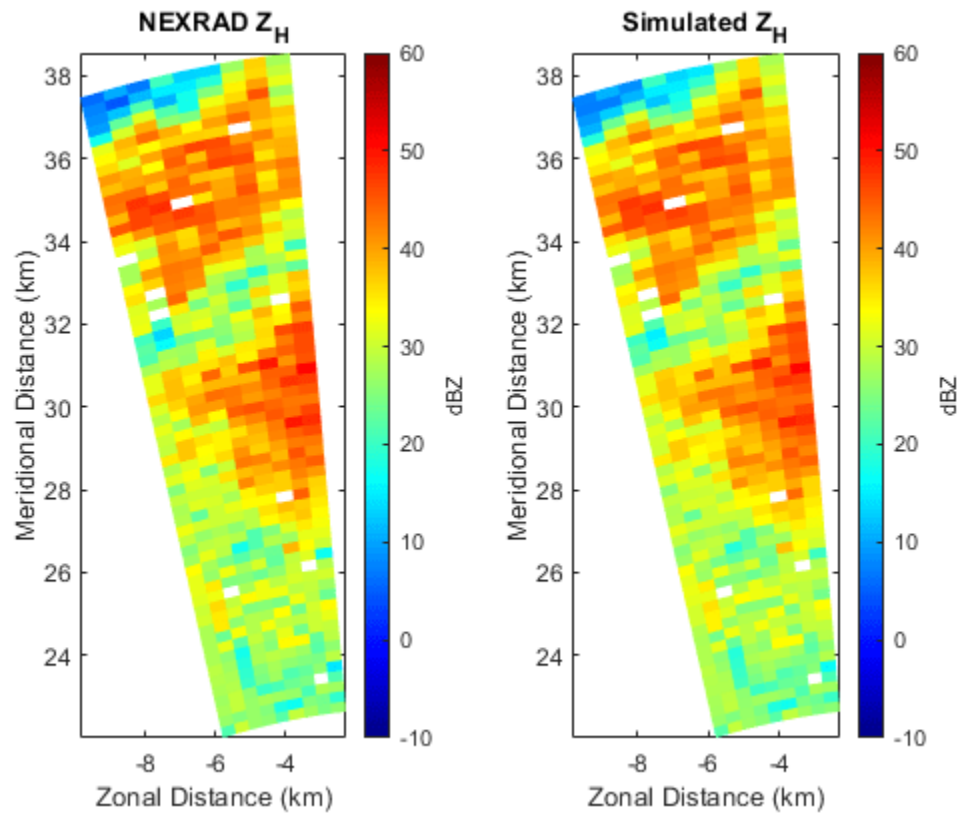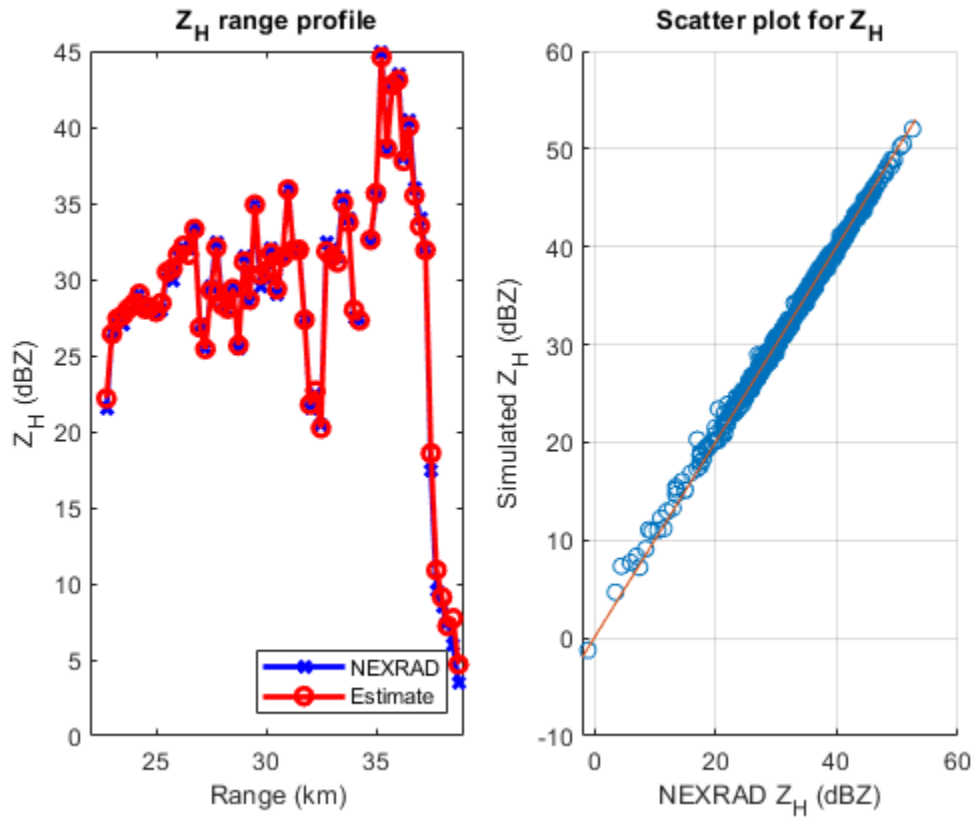
Radial Velocity

Radial velocity, V_r, is the first moment of the power-normalized spectra, which reflects the air motion toward or away from the radar.

```
[Vr_bias,Vr_std] = helperDataQuality(nexrad_aoi,moment,range,azimuth,'Vr');
```
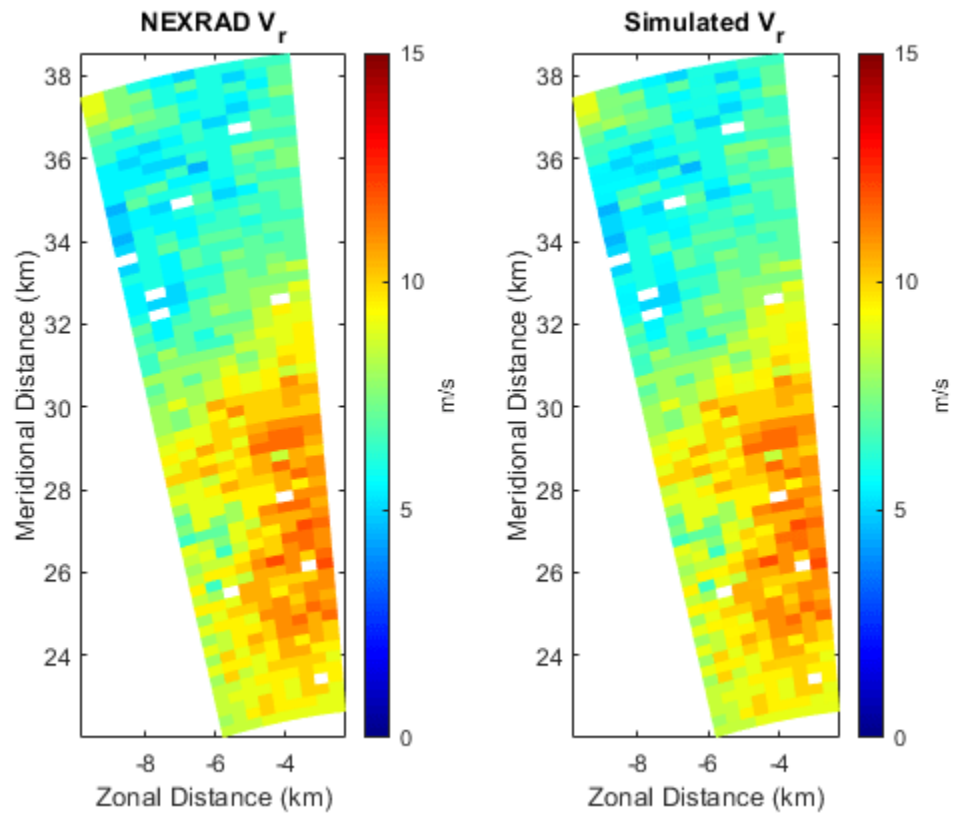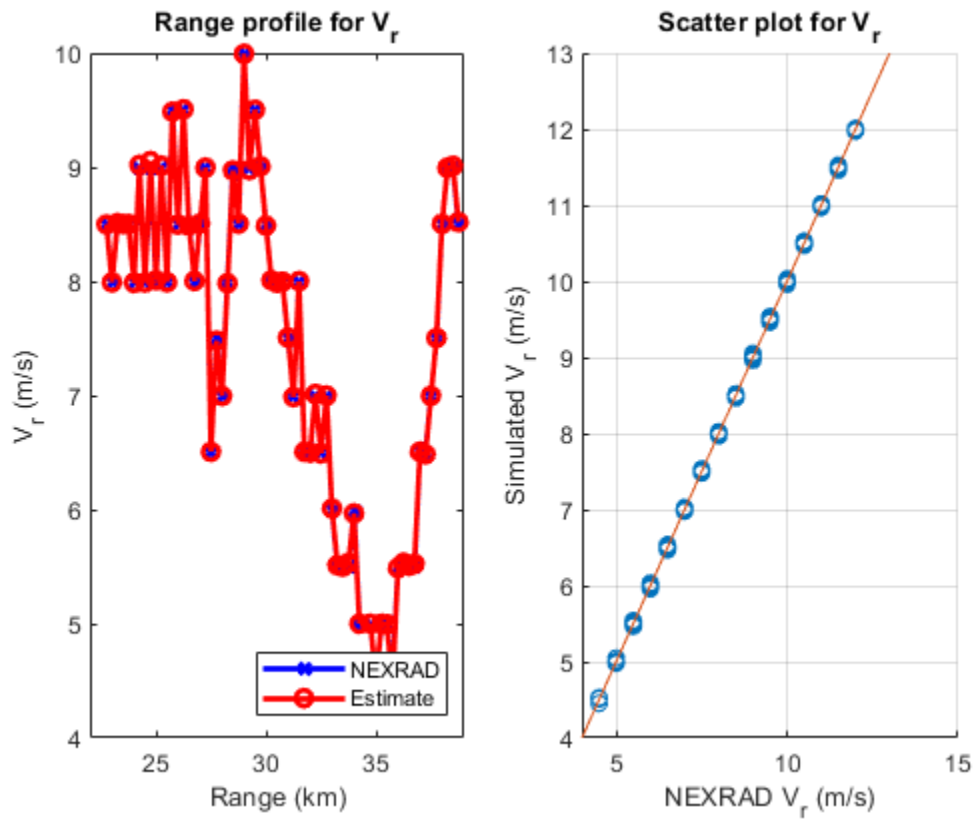
Spectrum Width

Spectrum width, σ_v, is the square root of the second moment of the normalized spectrum. The spectrum width is a measure of the velocity dispersion, that is, shear or turbulence within the resolution volume.

```
[sigmav_bias,sigmav_std] = helperDataQuality(nexrad_aoi,moment,range,azimuth,'sigmav');
```
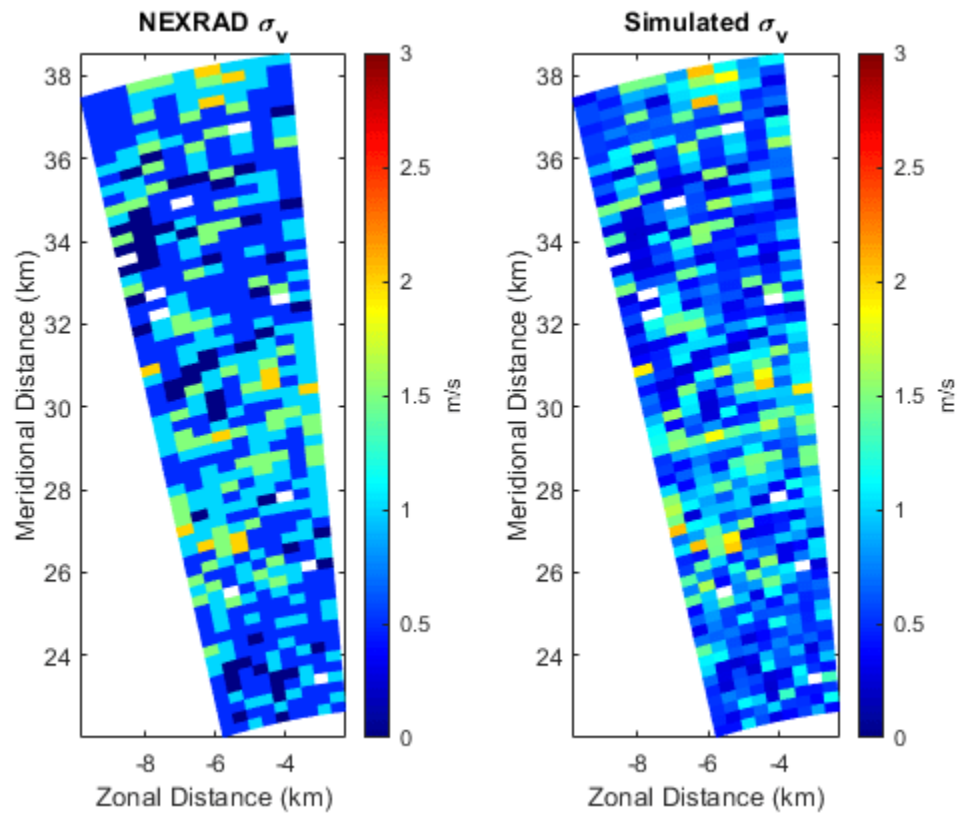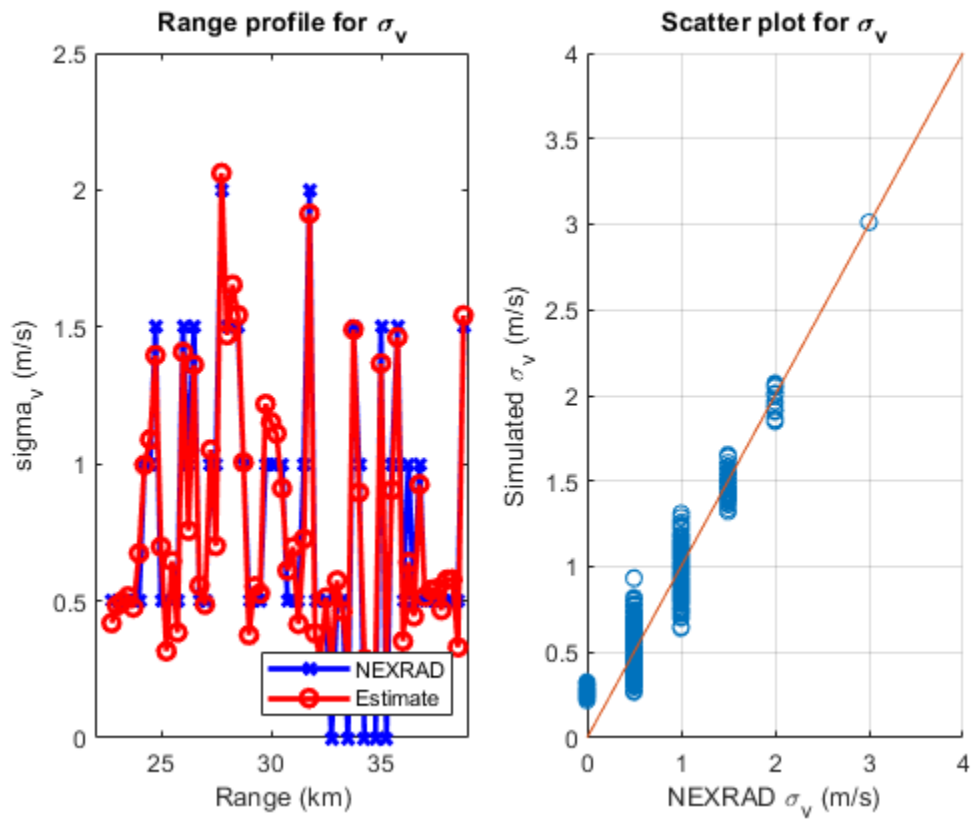
Differential Reflectivity

Differential reflectivity, Z_{DR}, is estimated from the ratio of the power estimates for the horizontal and vertical polarization signals. The differential reflectivity is useful in hydrometeor classification.

```
[ZDR_bias,ZDR_std] = helperDataQuality(nexrad_aoi,moment,range,azimuth,'ZDR');
```
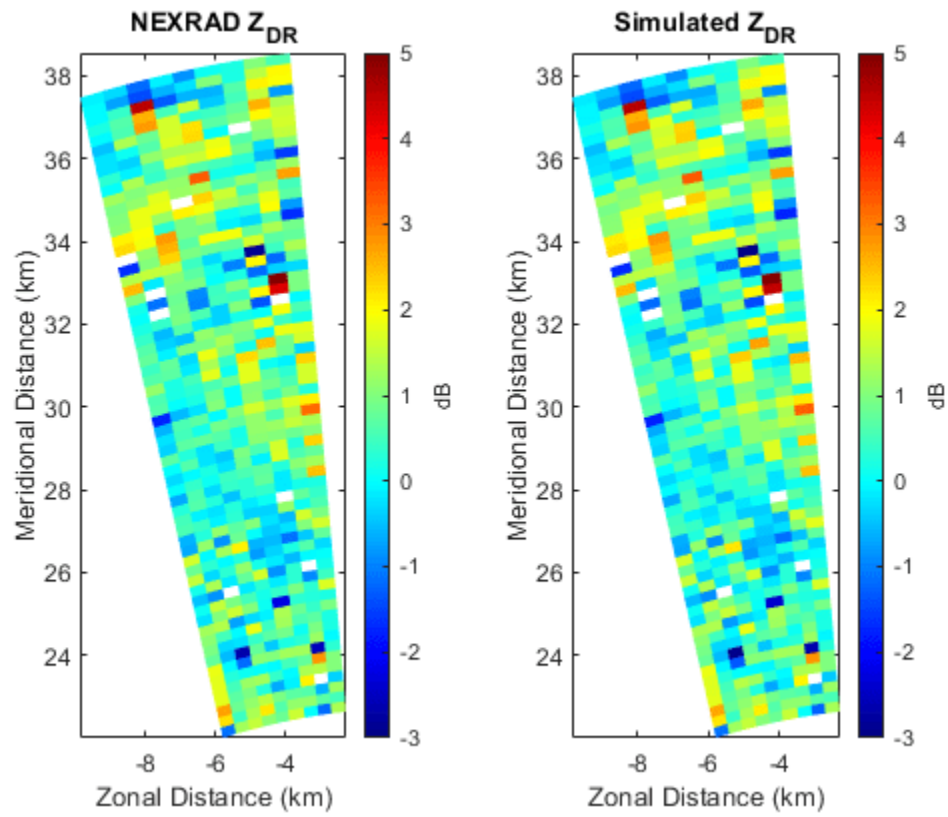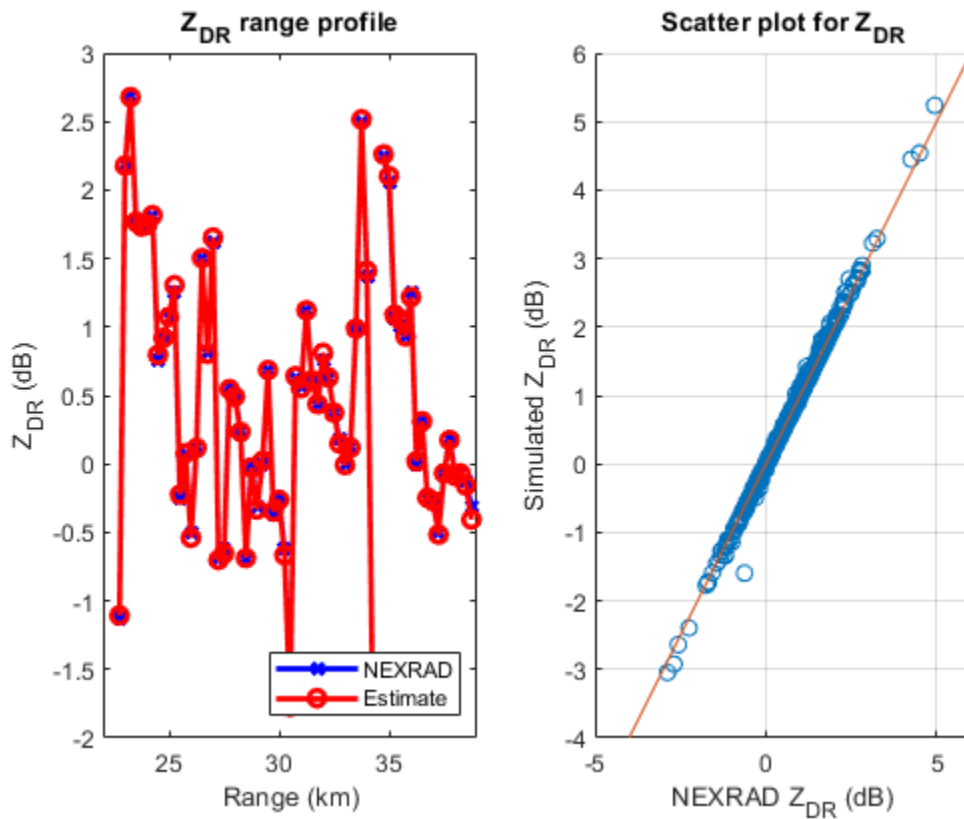
Correlation Coefficient

The correlation coefficient, ρ_{hv}, represents the consistency of the horizontal and vertical returned power and phase for each pulse. The correlation coefficient plays an important role in determining system performance and classifying radar echo types.

```
[Rhohv_bias,Rhohv_std] = helperDataQuality(nexrad_aoi,moment,range,azimuth,'Rhohv');
```
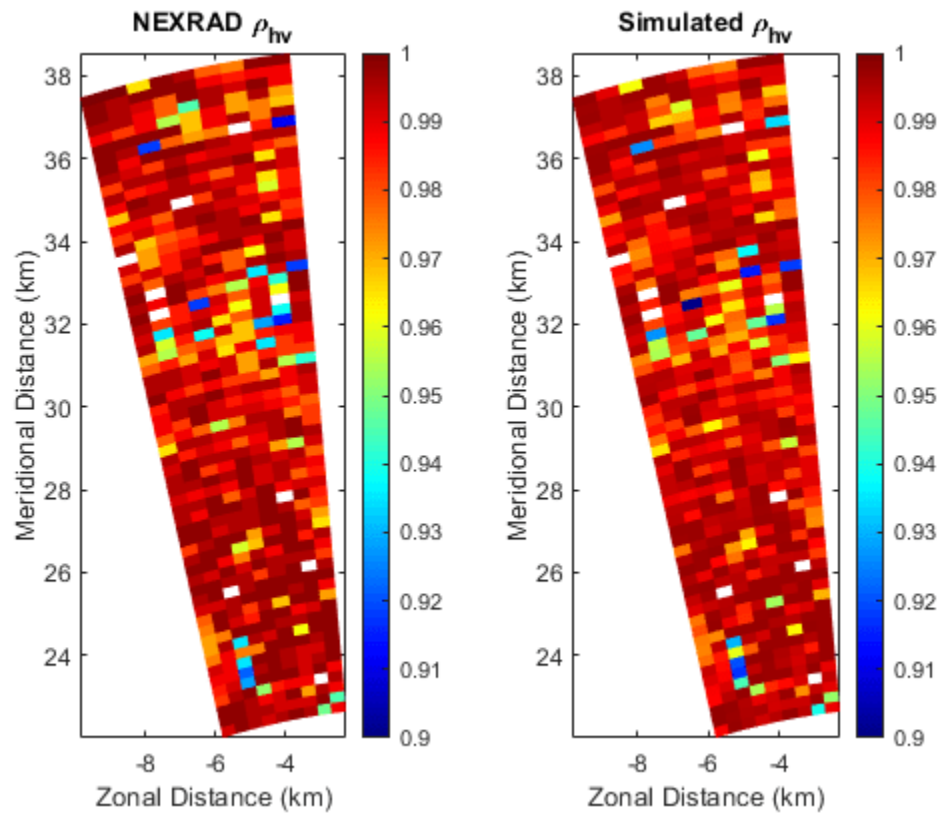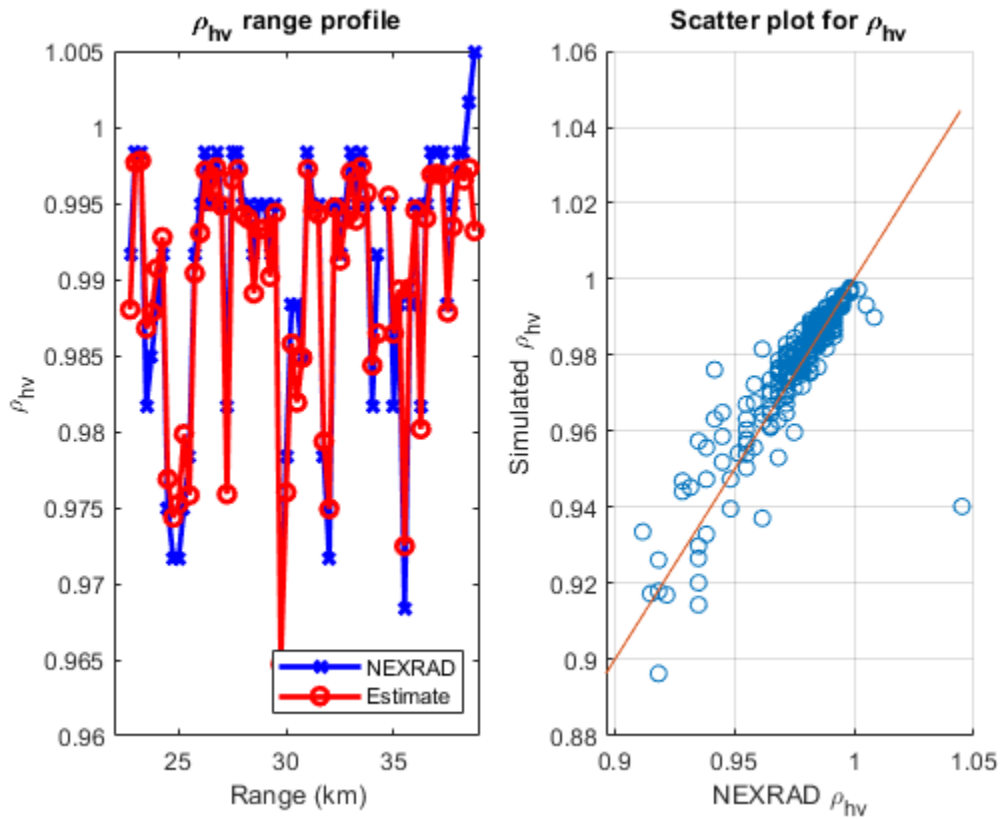
Differential Phase

The differential phase, ϕ_{DP}, is the difference in the phase delay of the returned pulse from the horizontal and vertical polarizations. The differential phase provides information on the nature of the scatterers that are being sampled.

```
[Phidp_bias,Phidp_std] = helperDataQuality(nexrad_aoi,moment,range,azimuth,'Phidp');
```
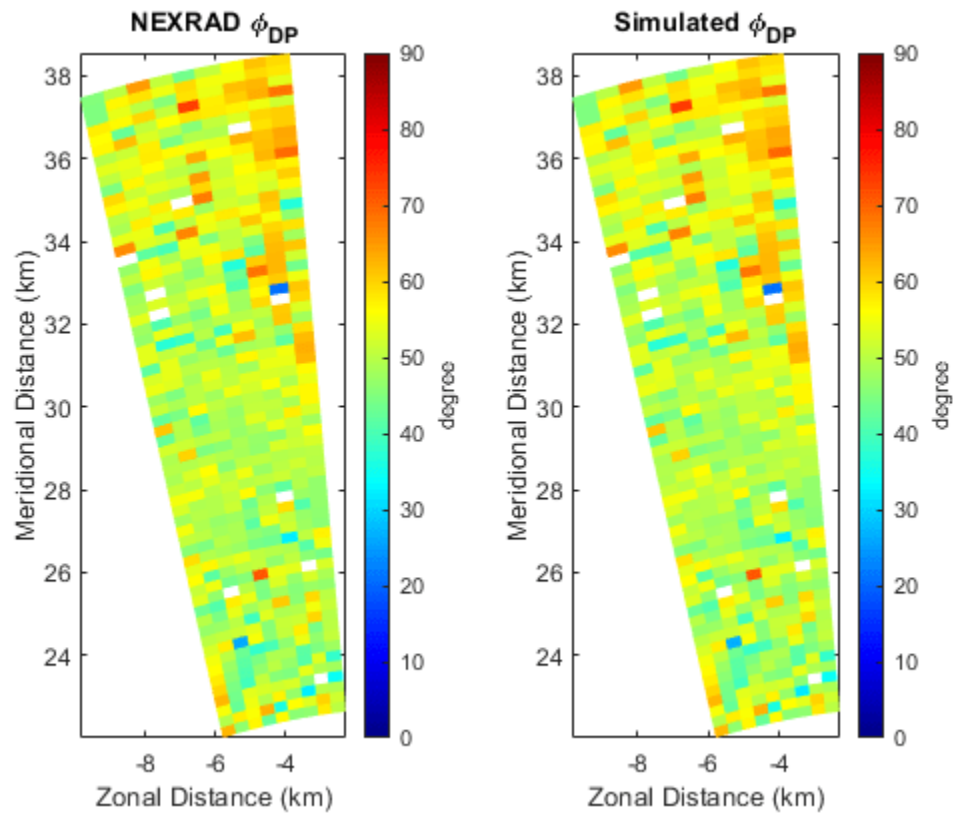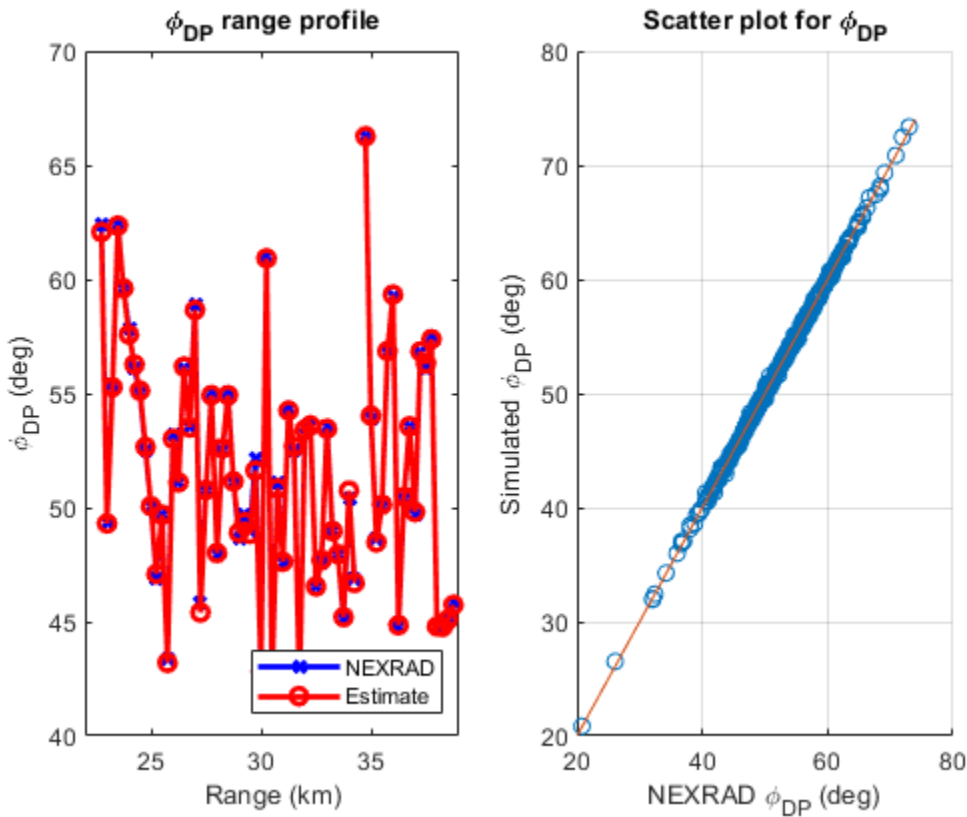
Error Statistics

Figures in previous section provide a visual qualitative measure of the simulation quality. This section of the example shows the quantitative comparison of the estimates with NEXRAD specifications as error statistics.

```
MomentName = {'Z';'Vr';'sigmav';'ZDR';'Rhohv';'Phidp'};
STDEV = [round(Z_std,2);round(Vr_std,2);round(sigmav_std,2);round(ZDR_std,2);round(Rhohv_std,3);
Specs = [1;1;1;0.2;0.01;2];
Unit = {'dB';'m/s';'m/s';'dB';'';'degree'};
T = table(MomentName,STDEV,Specs,Unit);
disp(T);
```

MomentName	STDEV	Specs	Unit
{'Z' }	0.5	1	{'dB' }
{'Vr' }	0.01	1	{'m/s' }
{'sigmav' }	0.14	1	{'m/s' }
{'ZDR' }	0.06	0.2	{'dB' }
{'Rhohv' }	0.006	0.01	{0x0 char}
{'Phidp' }	0.16	2	{'degree'}

By comparison, all the radar moment estimation meets NEXRAD specifications, which indicates good data quality.

Summary

This example showed how to simulate the polarimetric Doppler radar return from an area of distributed weather targets. Visual comparison and error statistics showed the estimated radar moments met the NEXRAD ground truth specifications. With this example, you can further explore the simulated time series data in other applications such as waveform design, system performance study, and data quality evaluation for weather radar.

References

[1] Doviak, R and D. Zrnic. *Doppler Radar and Weather Observations*, 2nd Ed. New York: Dover, 2006.

[2] Zhang, G. Weather Radar Polarimetry. Boca Raton: CRC Press, 2016.

[3] Li, Z, S. Perera, Y. Zhang, G. Zhang, and R. Doviak. "Time-Domain System Modeling and Applications for Multi-Function Array Radar Weather Measurements." *2018 IEEE Radar Conference (RadarConf18)*, Oklahoma city, OK, 2018, pp. 1049-1054.

Clutter and Jammer Mitigation with STAP

This example shows how to use Simulink® to suppress clutter and jammer interference from the received pulses of a monostatic radar. It illustrates how to model clutter and jammer interference as well as how to use the adaptive displaced phase center array (ADPCA) pulse canceller block in order to suppress the interference. The ADPCA Canceller block is one of several space-time adaptive processing (STAP) blocks provided in the Phased Array System Toolbox™. For more information on modeling an end-to-end monostatic radar in Simulink® see the "Simulating Test Signals for a Radar Receiver in Simulink" example. For an introduction to STAP see the "Introduction to Space-Time Adaptive Processing" example.

Structure of the Model

This example models a monostatic radar with a moving target and a stationary barrage jammer. The jammer transmits interfering signals through free space to the radar. A six-element uniform linear antenna array (ULA) with back baffled elements then receives the reflected pulse from the target as well as the jammer's interference. A clutter simulator's output is also added to the received signal before being processed. After adding noise, the signal is buffered into a data cube. In this example the cube is processed by the ADPCA Canceller block at the target's estimated range, azimuth angle, and doppler shift. In practice the ADPCA Canceller will scan several ranges, azimuth angles and doppler shifts since the speed and position of the target is unknown.

Several blocks in this example need to share the same sensor array configuration. This is done by assigning a sensor array configuration object to a MATLAB™ variable and sharing this variable in the **Sensor Array** tab of the Block Paramter dialog box.

In addition to the blocks listed in the "Simulating Test Signals for a Radar Receiver in Simulink" example, there are:

- FreeSpace -- Performs two-way propagation of the signal when you select two-way propagation is selected in the block parameters. This mode allows the use of one block instead of two to model the transmitted and reflected propagation paths of the signal.

- Jammer - Generates a barrage jamming signal. This subsystem also includes a `Platform` to model the speed and position of the jammer which are needed by the `Freespace` blocks. The position is also needed to calculate the angle between the target and the jammer.

- Selector - Selects the target's angle from the `Range Angle` block. This angle is used by the `Narrowband Tx Array` block.

- Constant Gamma Clutter - Generates clutter with a gamma value of -15 dB. Such a gamma value can be used to model terrain covered by woods.

- Radar Platform - Updates the position and velocity of the radar.

STAP
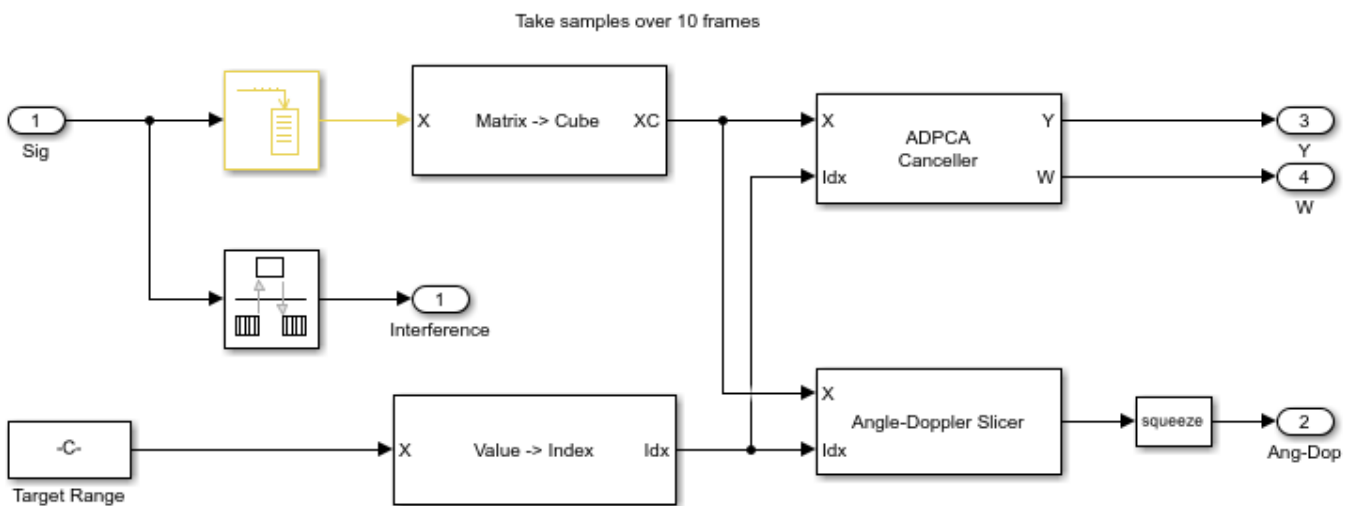
- Buffer - Buffers 10 pulses of the received signal.
- Matrix to Cube - Reshapes the buffered signal into an M-by-Q-by-N data cube. M is the number of range bins in fast time (the number of samples in one pulse), Q is the number of antenna elements, and N is the number of buffered pulses. The dimensions of the cube this example is 200-by-6-by-10 dimensions.
- Value to Index - Calculates the index of the estimated target's range bin from the range value.
- ADPCA Canceller - Perform ADPCA pulse cancelling along the specified range bin. The antenna array configuration of the radar is shared using a variable in the **Sensor Array** tab of the Block Parameter dialog box. The output is the received pulse with the clutter and jammer interference suppressed. The adaptive weights of the filter are also produced, enabling them is optional.
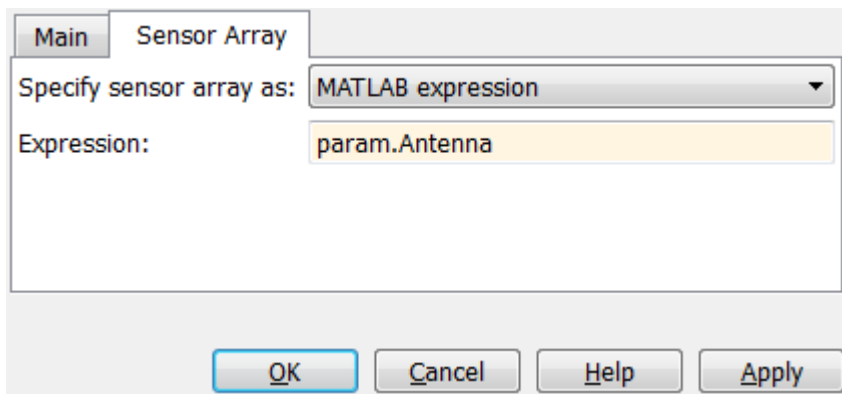
- Angle Doppler Slicer - Slices the data cube along the dimension specified in the block parameters. This example examines the angle-Doppler slice of the cube at the estimated range.

- Visualization - This subsystems displays the clutter interference in the time domain, the angle-Doppler response of the received data, the output of the ADPCA Canceller block, as well as the weights.

Exploring the Example

Several parameters of the model are calculated by the helper function helperslexSTAPParam. To open the function from the model, click the Modify Simulation Parameters block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the block parameters. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

Displays from different stages of the simulation are shown here. The first figure shows how the signal received at the antenna array is dominated by the clutter return. Because the radar is located 1000 meters above the surface, the clutter returns from the ground start at 1000 meters.
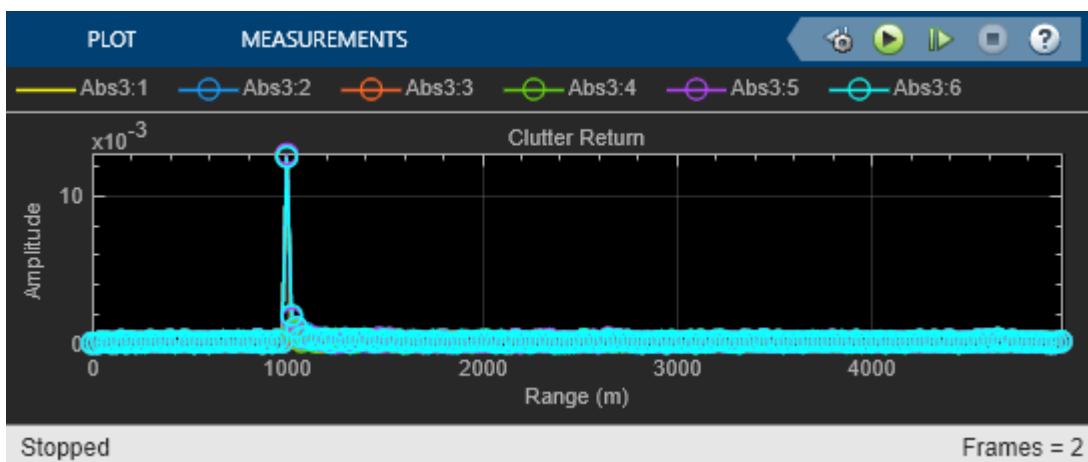
This figure shows the angle-Doppler response of the return for the estimated range bin. It presents the clutter as a function of angle and Doppler shift. The clutter return looks like a diagonal line in angle-Doppler space. Such a line is often referred to as a *clutter ridge*. The received jammer signal is white noise, spread over the entire Doppler spectrum at approximately 60 degrees.
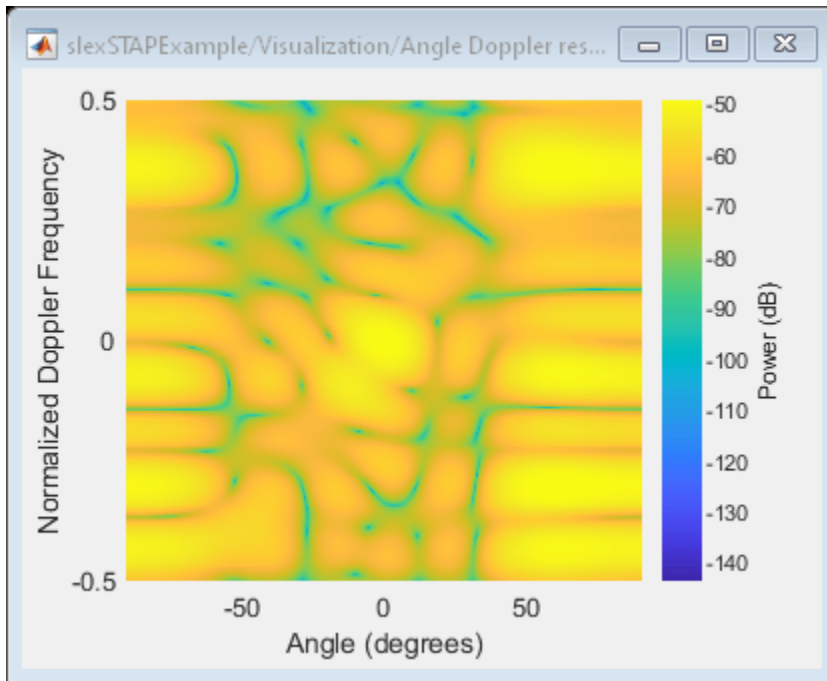
As you can see in the here, the weights of the ADPCA Canceller produce a deep null along the clutter ridge and also in the direction of the jammer.
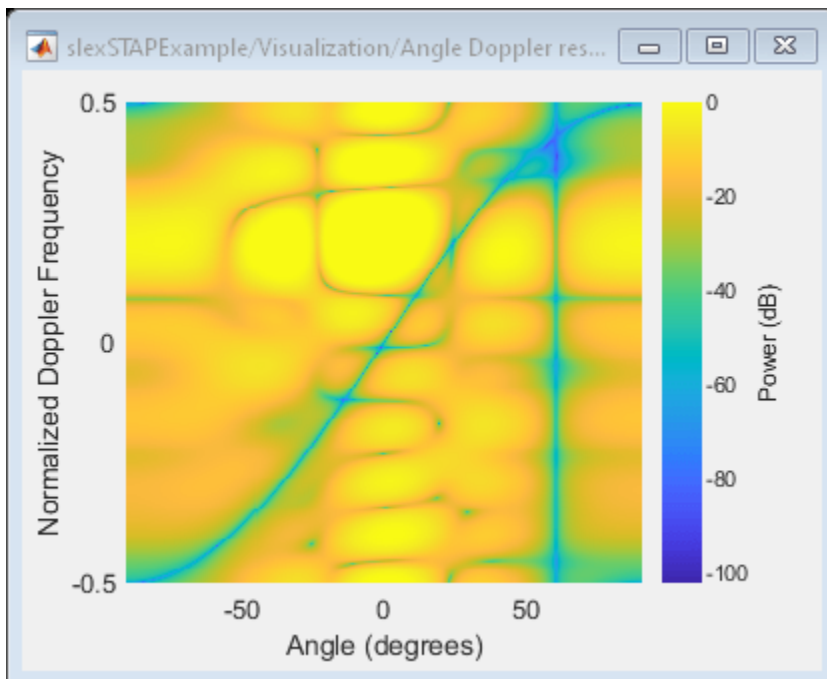
The next figure displays the return at the output of the ADPCA Canceller, clearly showing the target's range at 1750 meters. The barrage jammer and clutter have been filtered out.
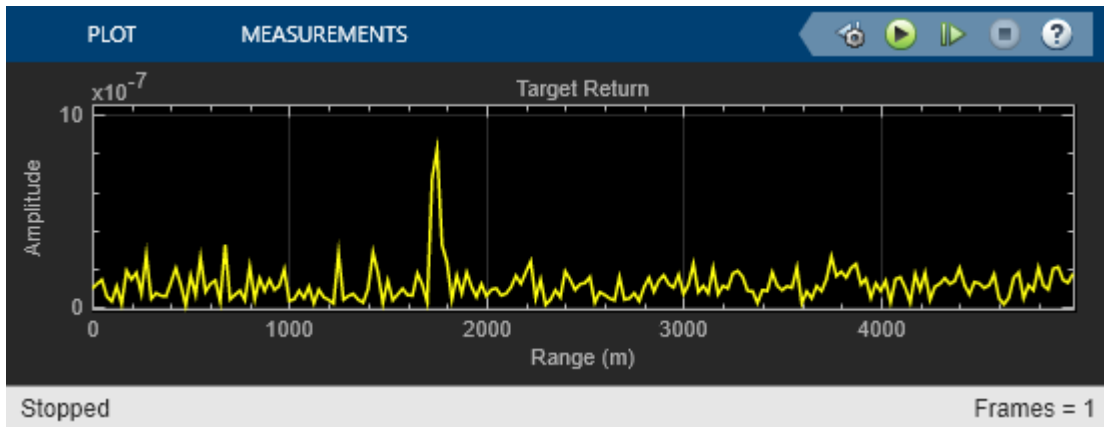
Introduction to Space-Time Adaptive Processing

This example gives a brief introduction to space-time adaptive processing (STAP) techniques and illustrates how to use Phased Array System Toolbox™ to apply STAP algorithms to the received pulses. STAP is a technique used in airborne radar systems to suppress clutter and jammer interference.

Introduction

In a ground moving target indicator (GMTI) system, an airborne radar collects the returned echo from the moving target on the ground. However, the received signal contains not only the reflected echo from the target, but also the returns from the illuminated ground surface. The return from the ground is generally referred to as *clutter*.

The clutter return comes from all the areas illuminated by the radar beam, so it occupies all range bins and all directions. The total clutter return is often much stronger than the returned signal echo, which poses a great challenge to target detection. Clutter filtering, therefore, is a critical part of a GMTI system.

In traditional MTI systems, clutter filtering often takes advantage of the fact that the ground does not move. Thus, the clutter occupies the zero Doppler bin in the Doppler spectrum. This principle leads to many Doppler-based clutter filtering techniques, such as pulse canceller. Interested readers can refer to "Ground Clutter Mitigation with Moving Target Indication (MTI) Radar" on page 1-525 for a detailed example of the pulse canceller. When the radar platform itself is also moving, such as in a plane, the Doppler component from the ground return is no longer zero. In addition, the Doppler components of clutter returns are angle dependent. In this case, the clutter return is likely to have energy across the Doppler spectrum. Therefore, the clutter cannot be filtered only with respect to Doppler frequency.

Jamming is another significant interference source that is often present in the received signal. The simplest form of jamming is a barrage jammer, which is strong, continuous white noise directed toward the radar receiver so that the receiver cannot easily detect the target return. The jammer is usually at a specific location, and the jamming signal is therefore associated with a specific direction. However, because of the white noise nature of the jammer, the received jamming signal occupies the entire Doppler band.

STAP techniques filter the signal in both the angular and Doppler domains (thus, the name space-time adaptive processing) to suppress the clutter and jammer returns. In the following sections, you will simulate returns from target, clutter, and jammer and illustrate how STAP techniques filter the interference from the received signal.

System Setup

First define a radar system, starting from the system built in the example "Simulating Test Signals for a Radar Receiver".

```
load BasicMonostaticRadarExampleData.mat;     % Load monostatic pulse radar
```

Antenna Definition

Assume that the antenna element has an isotropic response in the front hemisphere and all zeros in the back hemisphere. The operating frequency range is set to 8 to 12 GHz to match the 10 GHz operating frequency of the system.

```
antenna = phased.IsotropicAntennaElement...
    ('FrequencyRange',[8e9 12e9],'BackBaffled',true); % Baffled Isotropic
```

Define a 6-element uniform linear array (ULA) with a custom element pattern. Assume the element spacing to be one half the wavelength of the waveform.

```
fc = radiator.OperatingFrequency;
c = radiator.PropagationSpeed;
lambda = c/fc;
ula = phased.ULA('Element',antenna,'NumElements',6,...
    'ElementSpacing',lambda/2);
```

```
pattern(ula,fc,'PropagationSpeed',c,'Type','powerdb')
title('Six-Element Baffled ULA Response Pattern')
view(60,50)
```
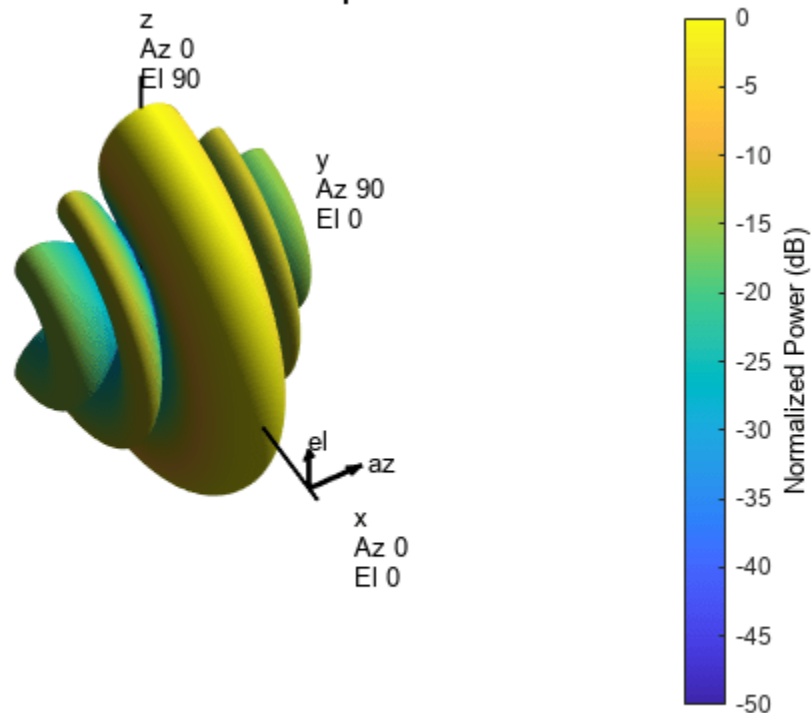

Radar Setup

Next, mount the antenna array on the radiator and collector. Then, define the radar motion. The radar system is mounted on a plane that flies 1000 meters above the ground. The plane is flying along the array axis of the ULA at a speed such that it travels a half element spacing of the array during one pulse interval. (An explanation of such a setting is provided in the DPCA technique section that follows.)

```
radiator.Sensor = ula;
collector.Sensor = ula;
sensormotion = phased.Platform('InitialPosition',[0; 0; 1000]);
```

```
arrayAxis = [0; 1; 0];
prf = waveform.PRF;
vr = ula.ElementSpacing*prf;  % in [m/s]
sensormotion.Velocity = vr/2*arrayAxis;
```

Target

Next, define a nonfluctuating target with a radar cross section of 1 square meter moving on the ground.

```
target = phased.RadarTarget('Model','Nonfluctuating','MeanRCS',1, ...
    'OperatingFrequency', fc);
tgtmotion = phased.Platform('InitialPosition',[1000; 1000; 0],...
    'Velocity',[30; 30; 0]);
```

Jammer

The target returns the desired signal; however, several interferences are also present in the received signal. If you have a Radar Toolbox license, set the variable `hasRadarToolbox` to true to define a simple barrage jammer with an effective radiated power of 100 watts. Otherwise, the simulation uses a saved jammer signal.

```
hasRadarToolbox = false;
Fs = waveform.SampleRate;
rngbin = c/2*(0:1/Fs:1/prf-1/Fs).';
if hasRadarToolbox
    jammer = barrageJammer('ERP',100);
    jammer.SamplesPerFrame = numel(rngbin);
    jammermotion = phased.Platform('InitialPosition',[1000; 1732; 1000]);
end
```

Clutter

This example simulates clutter using the constant-gamma model with a gamma value of -15 dB. Literature shows that such a gamma value can be used to model terrain covered by woods. For each range, the clutter return can be thought of as a combination of the returns from many small clutter patches on that range ring. Since the antenna is back baffled, the clutter contribution is only from the front. To simplify the computation, use an azimuth width of 10 degrees for each patch. Again, if you do not have a Radar Toolbox license, the simulation uses a saved clutter signal.

```
if hasRadarToolbox
    Rmax = 5000;
    Azcov = 120;
    clutter = constantGammaClutter('Sensor',ula,'SampleRate',Fs,...
        'Gamma',-15,'PlatformHeight',1000,...
        'OperatingFrequency',fc,...
        'PropagationSpeed',c,...
        'PRF',prf,...
        'TransmitERP',transmitter.PeakPower*db2pow(transmitter.Gain),...
        'PlatformSpeed',norm(sensormotion.Velocity),...
        'PlatformDirection',[90;0],...
        'ClutterMaxRange', Rmax,...
        'ClutterAzimuthSpan',Azcov,...
        'PatchAzimuthSpan',10,...
        'OutputFormat','Pulses');
end
```

Propagation Paths

Finally, create a free space environment to represent the target and jammer paths. Because the example uses a monostatic radar system, the target channel is set to simulate two-way propagation delays. The jammer path computes only one-way propagation delays.

```
tgtchannel = phased.FreeSpace('TwoWayPropagation',true,'SampleRate',Fs,...
    'OperatingFrequency', fc);
jammerchannel = phased.FreeSpace('TwoWayPropagation',false,...
    'SampleRate',Fs,'OperatingFrequency', fc);
```

Simulation Loop

You can now simulate the returns. Collect 10 pulses. The seed of the random number generator from the jammer model is set to a constant to get reproducible results.

```
numpulse = 10; % Number of pulses
tsig = zeros(size(rngbin,1), ula.NumElements, numpulse);
jsig = tsig; tjcsig = tsig; tcsig = tsig; csig = tsig;

if hasRadarToolbox
    jammer.SeedSource = 'Property';
    jammer.Seed = 5;
    clutter.SeedSource = 'Property';
    clutter.Seed = 5;
else
    load STAPIntroExampleData;
end

for m = 1:numpulse

    % Update sensor, target and calculate target angle as seen by the sensor
    [sensorpos,sensorvel] = sensormotion(1/prf);
    [tgtpos,tgtvel] = tgtmotion(1/prf);
    [~,tgtang] = rangeangle(tgtpos,sensorpos);

    % Update jammer and calculate the jammer angles as seen by the sensor
    if hasRadarToolbox
        [jampos,jamvel] = jammermotion(1/prf);
        [~,jamang] = rangeangle(jampos,sensorpos);
    end

    % Simulate propagation of pulse in direction of targets
    pulse = waveform();
    [pulse,txstatus] = transmitter(pulse);
    pulse = radiator(pulse,tgtang);
    pulse = tgtchannel(pulse,sensorpos,tgtpos,sensorvel,tgtvel);

    % Collect target returns at sensor
    pulse = target(pulse);
    tsig(:,:,m) = collector(pulse,tgtang);

    % Collect jammer and clutter signal at sensor
    if hasRadarToolbox
        jamsig = jammer();
        jamsig = jammerchannel(jamsig,jampos,sensorpos,jamvel,sensorvel);
        jsig(:,:,m) = collector(jamsig,jamang);

        csig(:,:,m) = clutter();
    end
```

```
    % Receive collected signals
    tjcsig(:,:,m) = receiver(tsig(:,:,m)+jsig(:,:,m)+csig(:,:,m),...
        ~(txstatus>0));                          % Target + jammer + clutter
    tcsig(:,:,m) = receiver(tsig(:,:,m)+csig(:,:,m),...
        ~(txstatus>0));                          % Target + clutter
    tsig(:,:,m) = receiver(tsig(:,:,m),...
        ~(txstatus>0));                          % Target echo only
end
```

True Target Range, Angle and Doppler

The target azimuth angle is 45 degrees, and the elevation angle is about -35.27 degrees.

```
tgtLocation = global2localcoord(tgtpos,'rs',sensorpos);
tgtAzAngle = tgtLocation(1)
```

```
tgtAzAngle = 44.9981
```

```
tgtElAngle = tgtLocation(2)
```

```
tgtElAngle = -35.2651
```

The target range is 1732 m.

```
tgtRng = tgtLocation(3)
```

```
tgtRng = 1.7320e+03
```

The target Doppler normalized frequency is about 0.21.

```
sp = radialspeed(tgtpos, tgtmotion.Velocity, ...
            sensorpos, sensormotion.Velocity);
tgtDp = 2*speed2dop(sp,lambda);   % Round trip Doppler
tgtDp/prf
```

```
ans = 0.2116
```

The total received signal contains returns from the target, clutter and jammer combined. The signal is a data cube with three dimensions (range bins-by-number of elements-by-number of pulses). Notice that the clutter return dominates the total return and masks the target return. The target (blue vertical line) cannot be detected without further processing.

```
ReceivePulse = tjcsig;
plot([tgtRng tgtRng],[0 0.01],rngbin,abs(ReceivePulse(:,:,1)));
xlabel('Range (m)'), ylabel('Magnitude');
title('Signals Collected by the ULA Within the First Pulse Interval')
```

Signals Collected by the ULA Within the First Pulse Interval
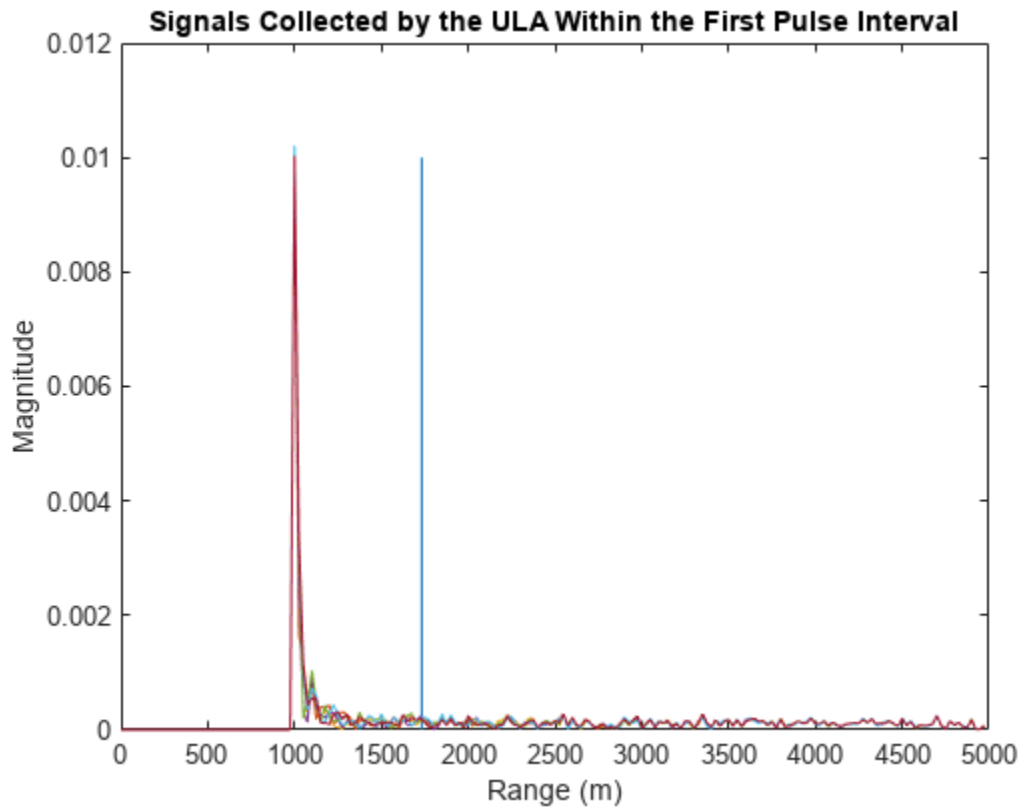
Now, examine the returns in 2-D angle Doppler (or space-time) domain. In general, the response is generated by scanning all ranges and azimuth angles for a given elevation angle. Because you know exactly where the target is, you can calculate its range and elevation angle with respect to the antenna array.

```
tgtCellIdx = val2ind(tgtRng,c/(2*Fs));
snapshot = shiftdim(ReceivePulse(tgtCellIdx,:,:));   % Remove singleton dim
angdopresp = phased.AngleDopplerResponse('SensorArray',ula,...
                'OperatingFrequency',fc, 'PropagationSpeed',c,...
                'PRF',prf, 'ElevationAngle',tgtElAngle);
plotResponse(angdopresp,snapshot,'NormalizeDoppler',true);
text(tgtAzAngle,tgtDp/prf,'+ Target')
```

Angle-Doppler Response Pattern
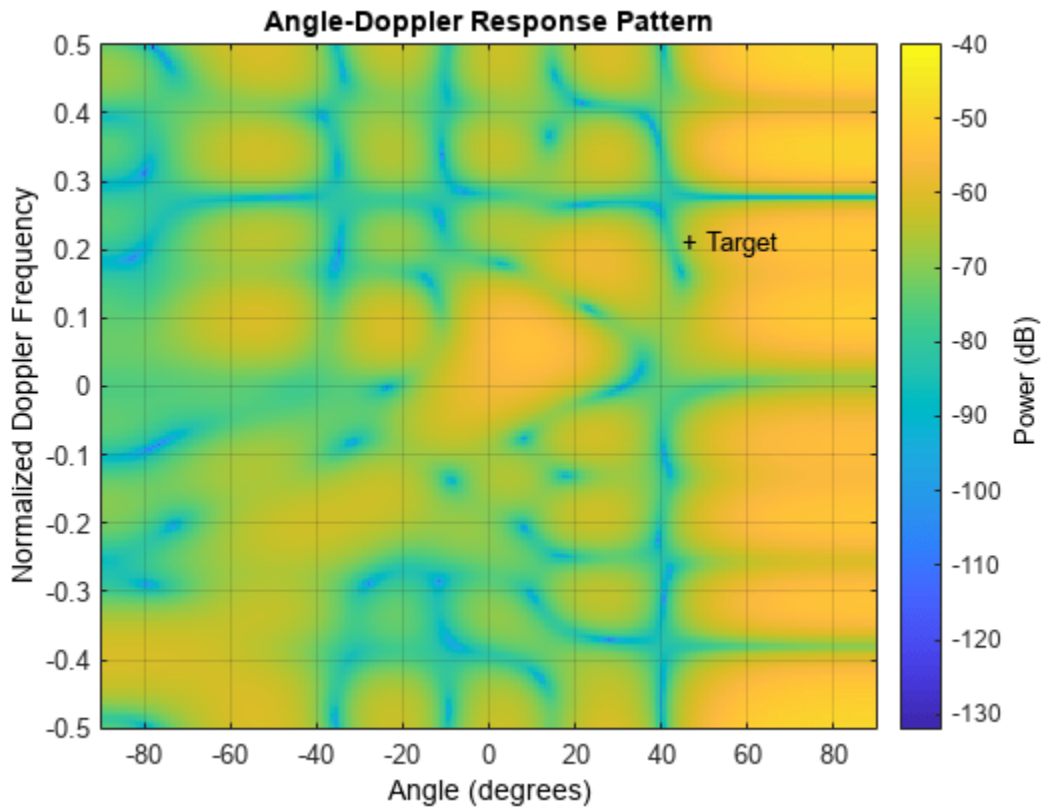
If you look at the angle-Doppler response, which is dominated by the clutter return, you see that the clutter return occupies not only the zero Doppler, but also other Doppler bins. The Doppler of the clutter return is also a function of the angle. The clutter return looks like a diagonal line in the entire angle Doppler space. Such a line is often referred to as *clutter ridge*. The received jammer signal is white noise, which spreads over the entire Doppler spectrum at a particular angle, around 60 degrees.

Clutter Suppression with a DPCA Canceller

The displaced phase center antenna (DPCA) algorithm is often considered to be the first STAP algorithm. This algorithm uses the shifted aperture to compensate for the platform motion, so that the clutter return does not change from pulse to pulse. Thus, this algorithm can remove the clutter via a simple subtraction of two consecutive pulses.

A DPCA canceller is often used on ULAs but requires special platform motion conditions. The platform must move along the array axis of the antenna and at a speed such that during one pulse interval, the platform travels exactly half of the element spacing. The system used here is set up, as described in earlier sections, to meet these conditions.

Assume that N is the number of ULA elements. The clutter return received at antenna 1 through antenna N-1 during the first pulse is the same as the clutter return received at antenna 2 through antenna N during the second pulse. By subtracting the pulses received at these two subarrays during the two pulse intervals, the clutter can be cancelled out. The cost of this method is an aperture that is one element smaller than the original array.

Now, define a DPCA canceller. The algorithm might need to search through all combinations of angle and Doppler to locate a target, but for the example, because you know exactly where the target is, you can direct the processor to that point.

```
rxmainlobedir = [0; 0];
stapdpca = phased.DPCACanceller('SensorArray',ula,'PRF',prf,...
    'PropagationSpeed',c,'OperatingFrequency',fc,...
    'Direction',rxmainlobedir,'Doppler',tgtDp,...
    'WeightsOutputPort',true)
```

```
stapdpca =
  phased.DPCACanceller with properties:

            SensorArray: [1x1 phased.ULA]
       PropagationSpeed: 299792458
     OperatingFrequency: 1.0000e+10
              PRFSource: 'Property'
                    PRF: 2.9979e+04
        DirectionSource: 'Property'
              Direction: [2x1 double]
     NumPhaseShifterBits: 0
          DopplerSource: 'Property'
                Doppler: 6.3429e+03
      WeightsOutputPort: true
        PreDopplerOutput: false
```

First, apply the DPCA canceller to both the target return and the clutter return.

```
ReceivePulse = tcsig;
[y,w] = stapdpca(ReceivePulse,tgtCellIdx);
```

The processed data combines all information in space and across the pulses to become a single pulse. Next, examine the processed signal in the time domain.

```
plot([tgtRng tgtRng],[0 1.2e-5],rngbin,abs(y));
xlabel('Range (m)'), ylabel('Magnitude');
title('DPCA Canceller Output (no Jammer)')
```
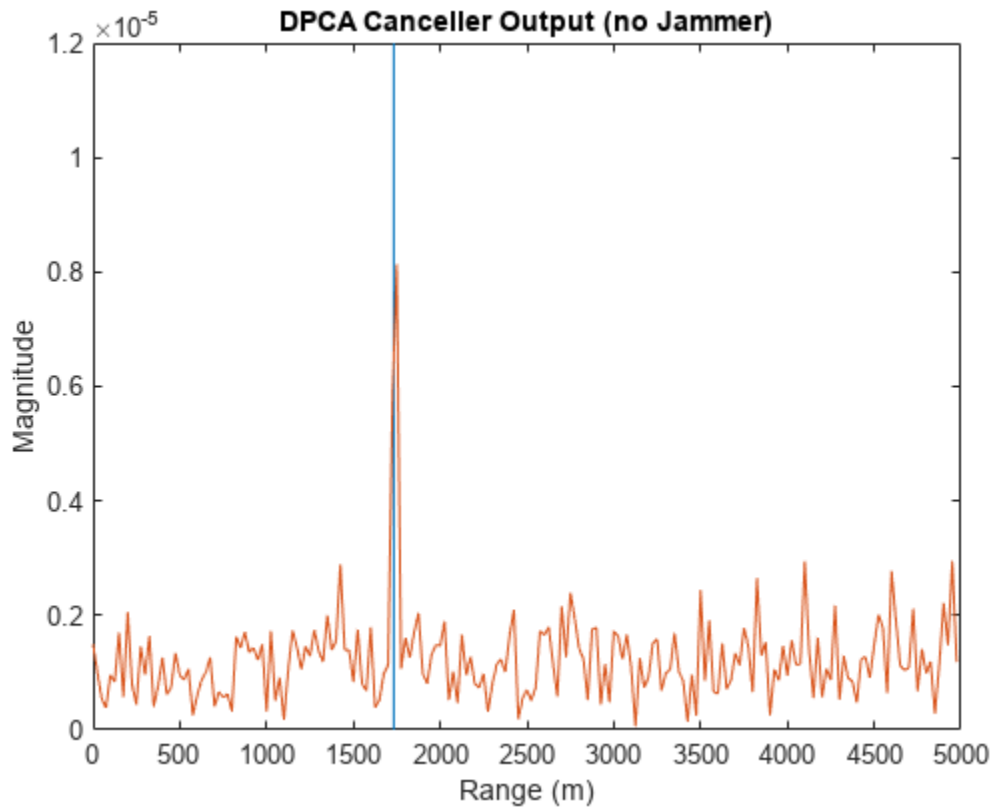
The signal now is clearly distinguishable from the noise and the clutter has been filtered out. From the angle-Doppler response of the DPCA processor weights below, you can also see that the weights produce a deep null along the clutter ridge.

```
angdopresp.ElevationAngle = 0;
plotResponse(angdopresp,w,'NormalizeDoppler',true);
title('DPCA Weights Angle Doppler Response at 0 degrees Elevation')
```

DPCA Weights Angle Doppler Response at 0 degrees Elevation
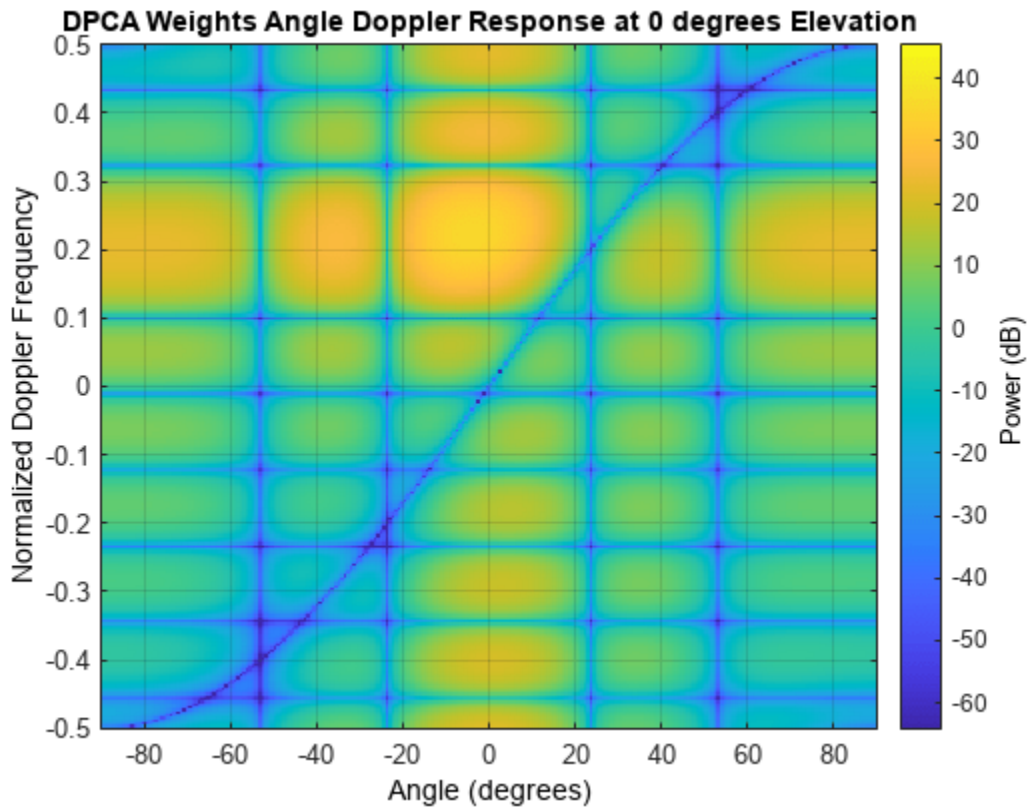
Although the results obtained by DPCA are very good, the radar platform has to satisfy very strict requirements in its movement to use this technique. Also, the DPCA technique cannot suppress the jammer interference.

Applying DPCA processing to the total signal produces the result shown in the following figure. You can see that DPCA cannot filter the jammer from the signal. The resulting angle Doppler pattern of the weights is the same as before. Thus, the processor cannot adapt to the newly added jammer interference.

```
ReceivePulse = tjcsig;
[y,w] = stapdpca(ReceivePulse,tgtCellIdx);
plot([tgtRng tgtRng],[0 8e-4],rngbin,abs(y));
xlabel('Range (m)'), ylabel('Magnitude');
title('DPCA Canceller Output (with Jammer)')
```
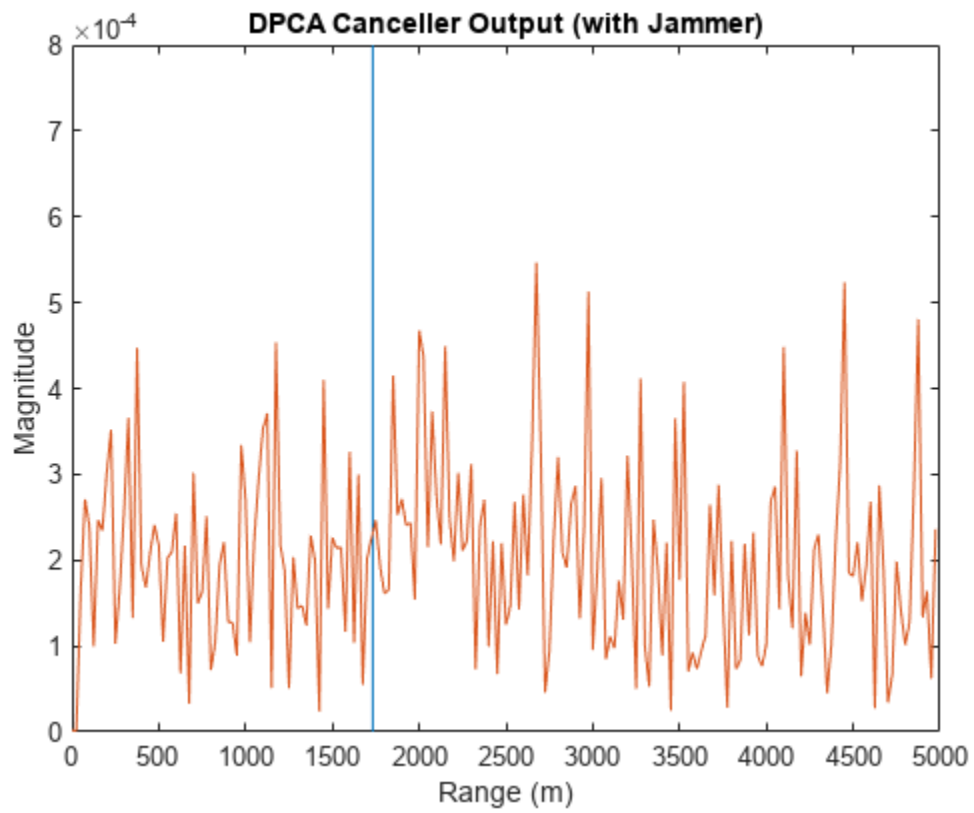
```
plotResponse(angdopresp,w,'NormalizeDoppler',true);
title('DPCA Weights Angle Doppler Response at 0 degrees Elevation')
```

DPCA Weights Angle Doppler Response at 0 degrees Elevation
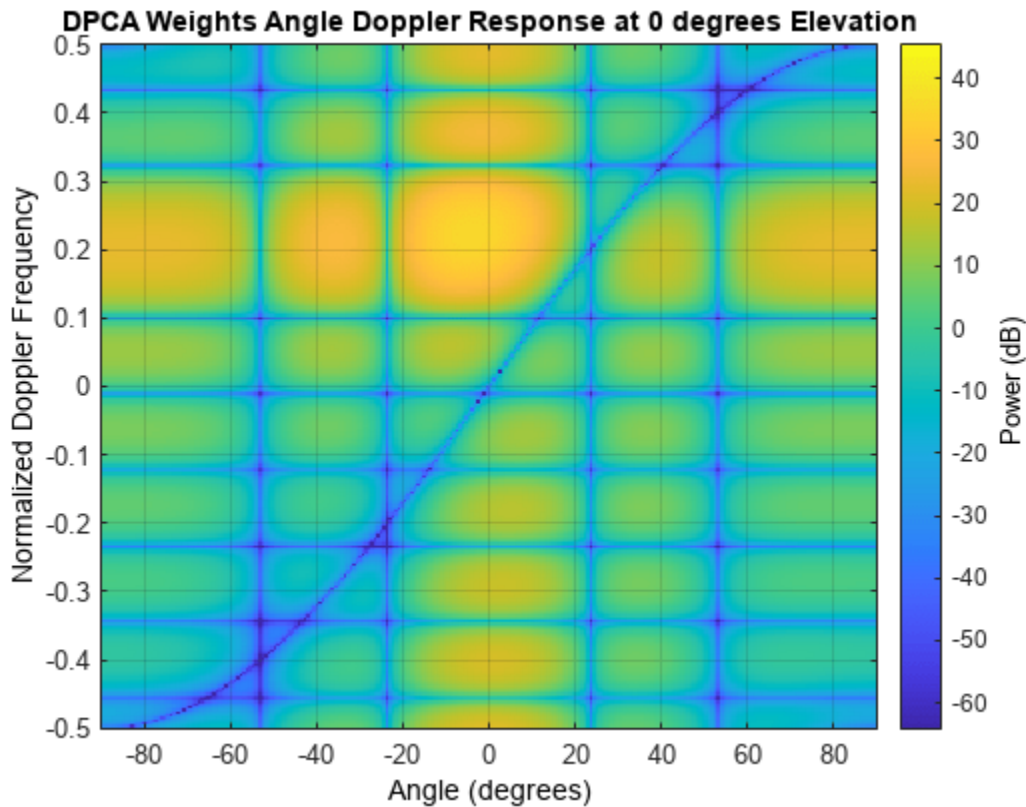
Clutter and Jammer Suppression with an SMI Beamformer

To suppress the clutter and jammer simultaneously, you need a more sophisticated algorithm. The optimum receiver weights, when the interference is Gaussian-distributed, are given by [1]

$$\mathbf{w} = k\mathbf{R}^{-1}\mathbf{s}$$

where k is a scalar factor, $\mathbf{R}$ is the space-time covariance matrix of the interference signal, and $\mathbf{s}$ is the desired space-time steering vector. The exact information of $\mathbf{R}$ is often unavailable, so this example uses the sample matrix inversion (SMI) algorithm. The algorithm estimates $\mathbf{R}$ from training-cell samples and then uses it in the aforementioned equation.

Now, define an SMI beamformer and apply it to the signal. In addition to the information needed in DPCA, the SMI beamformer needs to know the number of guard cells and the number of training cells. The algorithm uses the samples in the training cells to estimate the interference. Hence do not use the cells that are close to the target cell for the estimates because they can contain some target information, that is, you need to define guard cells. The number of guard cells must be an even number to be split equally in front of and behind the target cell. The number of training cells also must be an even number and split equally in front of and behind the target. Normally, the larger the number of training cells, the better the interference estimate.

```
tgtAngle = [tgtAzAngle; tgtElAngle];
stapsmi = phased.STAPSMIBeamformer('SensorArray', ula, 'PRF', prf, ...
    'PropagationSpeed', c, 'OperatingFrequency', fc, ...
    'Direction', tgtAngle, 'Doppler', tgtDp, ...
```

```
    'WeightsOutputPort', true,...
    'NumGuardCells', 4, 'NumTrainingCells', 100)

stapsmi =
  phased.STAPSMIBeamformer with properties:

           SensorArray: [1x1 phased.ULA]
      PropagationSpeed: 299792458
     OperatingFrequency: 1.0000e+10
             PRFSource: 'Property'
                   PRF: 2.9979e+04
        DirectionSource: 'Property'
             Direction: [2x1 double]
    NumPhaseShifterBits: 0
         DopplerSource: 'Property'
               Doppler: 6.3429e+03
          NumGuardCells: 4
       NumTrainingCells: 100
      WeightsOutputPort: true
```

```
[y,w] = stapsmi(ReceivePulse,tgtCellIdx);
plot([tgtRng tgtRng],[0 2e-6],rngbin,abs(y));
xlabel('Range (m)'), ylabel('Magnitude');
title('SMI Beamformer Output (with Jammer)')
```
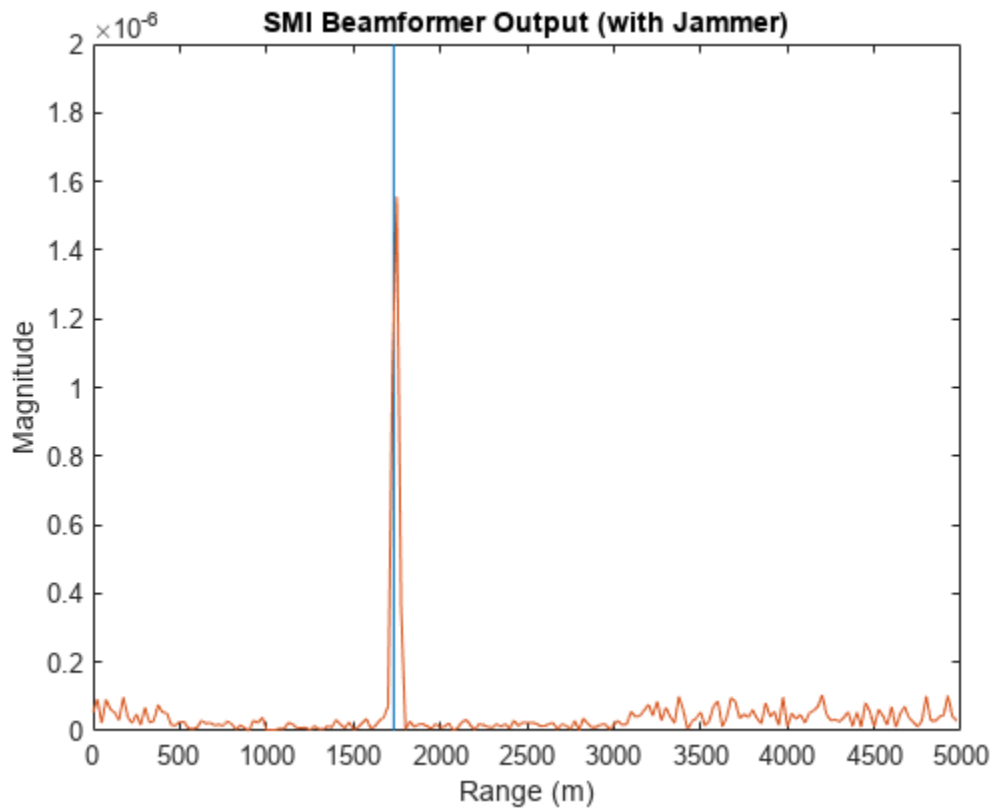


```
plotResponse(angdoppresp,w,'NormalizeDoppler',true);
title('SMI Weights Angle Doppler Response at 0 degrees Elevation')
```
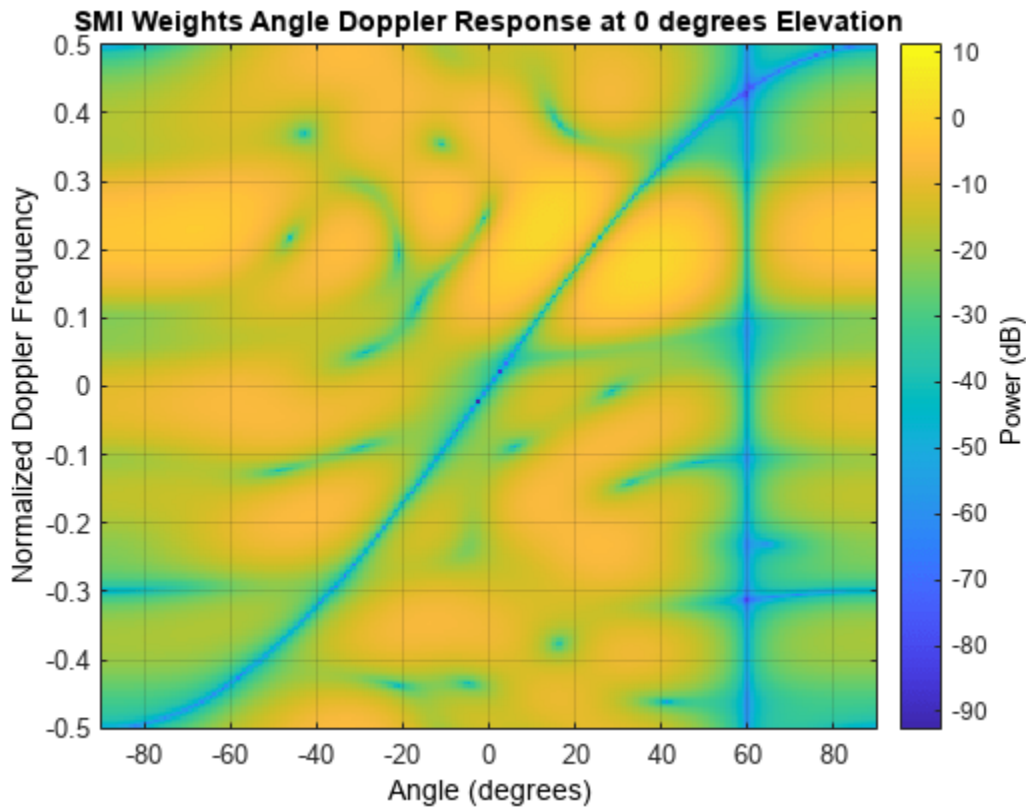
The result shows that an SMI beamformer can distinguish signals from both the clutter and the jammer. The angle Doppler pattern of the SMI weights shows a deep null along the jammer direction.

SMI provides the maximum degrees of freedom, and hence, the maximum gain among all STAP algorithms. It is often used as a baseline for comparing different STAP algorithms.

Reducing the Computation Cost with an ADPCA Canceller

Although SMI is the optimum STAP algorithm, it has several innate drawbacks, including a high computation cost because it uses the full dimension data of each cell. More importantly, SMI requires a stationary environment across many pulses. This kind of environment is not often found in real applications. Therefore, many reduced dimension STAP algorithms have been proposed.

An adaptive DPCA (ADPCA) canceller filters out the clutter in the same manner as DPCA, but it also has the capability to suppress the jammer as it estimates the interference covariance matrix using two consecutive pulses. Because there are only two pulses involved, the computation is greatly reduced. In addition, because the algorithm is adapted to the interference, it can also tolerate some motion disturbance.

Now, define an ADPCA canceller, and then apply it to the received signal.

```
stapadpca = phased.ADPCACanceller('SensorArray', ula, 'PRF', prf, ...
    'PropagationSpeed', c, 'OperatingFrequency', fc, ...
    'Direction', rxmainlobedir, 'Doppler', tgtDp, ...
    'WeightsOutputPort', true,...
    'NumGuardCells', 4, 'NumTrainingCells', 100)
```

```
stapadpca =
  phased.ADPCACanceller with properties:

             SensorArray: [1x1 phased.ULA]
        PropagationSpeed: 299792458
      OperatingFrequency: 1.0000e+10
               PRFSource: 'Property'
                     PRF: 2.9979e+04
          DirectionSource: 'Property'
               Direction: [2x1 double]
      NumPhaseShifterBits: 0
           DopplerSource: 'Property'
                 Doppler: 6.3429e+03
           NumGuardCells: 4
        NumTrainingCells: 100
       WeightsOutputPort: true
        PreDopplerOutput: false
```

```matlab
[y,w] = stapadpca(ReceivePulse,tgtCellIdx);
plot([tgtRng tgtRng],[0 2e-6],rngbin,abs(y));
xlabel('Range (m)'), ylabel('Magnitude');
title('ADPCA Canceller Output (with Jammer)')
```
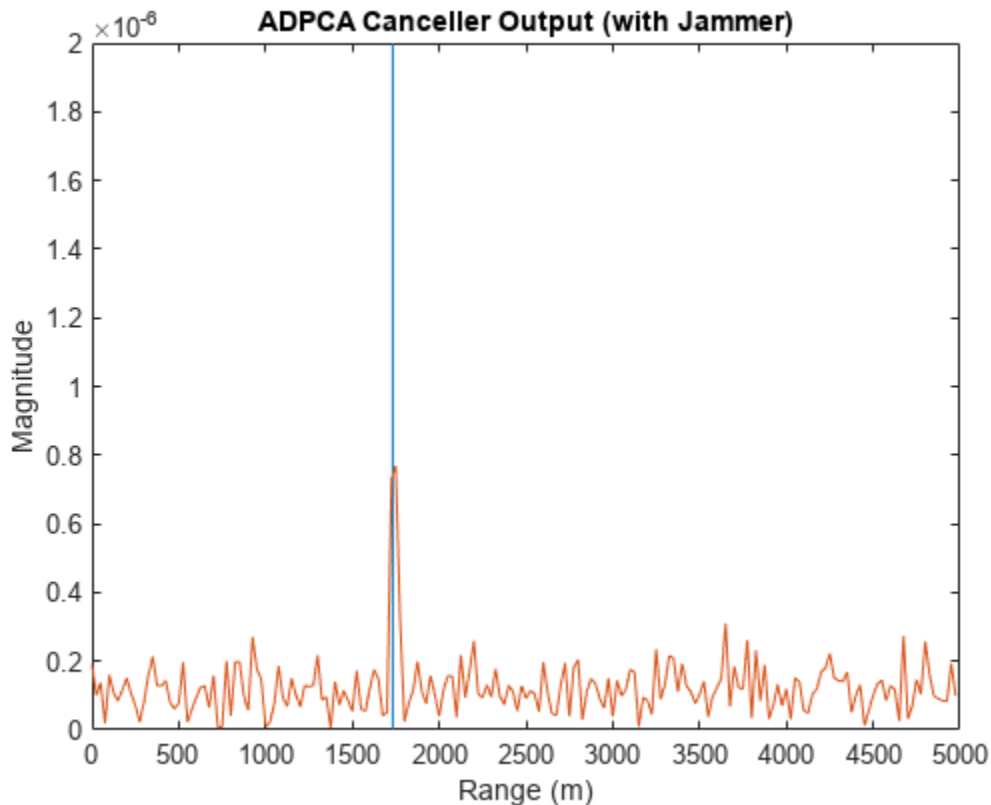


```matlab
plotResponse(angdoporesp,w,'NormalizeDoppler',true);
title('ADPCA Weights Angle Doppler Response at 0 degrees Elevation')
```
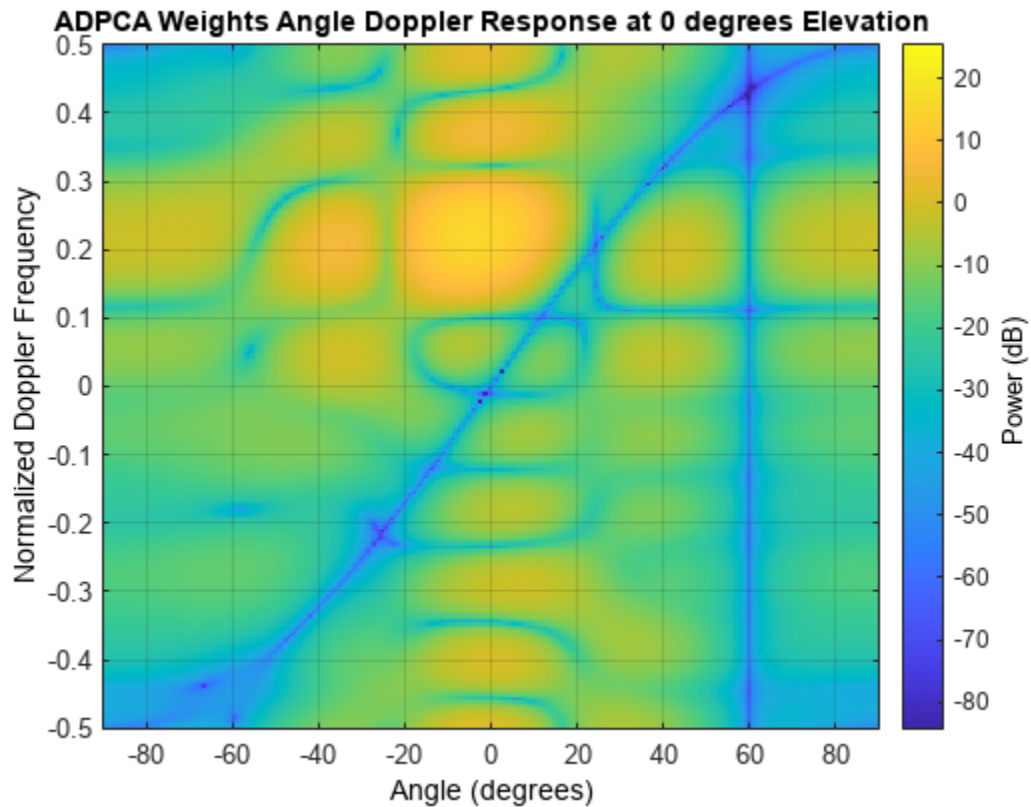
The time domain plot shows that the signal is successfully recovered. The angle Doppler response of the ADPCA weights is similar to the one produced by the SMI weights.

Summary

This example presented a brief introduction to STAP and illustrated how to use different STAP algorithms, namely, SMI, DPCA, and ADPCA, to suppress clutter and jammer interference in the received pulses.

Reference

[1] Guerci, J. R. *Space-Time Adaptive Processing for Radar*. Artech House Radar Library. Boston: Artech House, 2003.

Acceleration of Clutter Simulation Using GPU and Code Generation

This example shows how to simulate clutter on a graphical processing unit (GPU) or through code generation (MEX) instead of the MATLAB™ interpreter. The example applies the sample matrix inversion (SMI) algorithm, one of the popular space-time adaptive processing (STAP) techniques, to the signal received by an airborne radar with a six-element uniform linear array (ULA). The example focuses on comparing the performance of clutter simulation between GPU, code generation, and the MATLAB interpreter. You can find details of the simulation and the algorithm in the example "Introduction to Space-Time Adaptive Processing".

The full functionality of this example requires Parallel Computing Toolbox™ and MATLAB Coder™.

Clutter Simulation

Radar system engineers often need to simulate the clutter return to test signal processing algorithms, such as STAP algorithms. However, generating a high fidelity clutter return involves many steps and therefore is often computationally expensive. For example, `constantGammaClutter` simulates the clutter using the following steps:

1   Divide the entire terrain into small clutter patches. The size of the patch depends on the azimuth patch span and the range resolution.

2   For each patch, calculate its corresponding parameters, such as the random return, the grazing angle, and the antenna array gain.

3   Combine returns from all clutter patches to generate the total clutter return.

The number of clutter patches depends on the terrain coverage, but it is usually in the range of thousands to millions. In addition, all steps above need to be performed for each pulse (assuming a pulsed radar is used). Therefore, clutter simulation is often the tall pole in a system simulation.

To improve the speed of the clutter simulation, one can take advantage of parallel computing. Note that the clutter return from later pulses could depend on the signal generated in earlier pulses, so certain parallel solutions offered by MATLAB, such as `parfor`, are not always applicable. However, because the computation at each patch is independent of the computations at other patches, it is suitable for GPU acceleration.

If you have a supported GPU and have access to Parallel Computing Toolbox, then you can take advantage of the GPU in generating the clutter return by using `gpuConstantGammaClutter` instead of `constantGammaClutter`. In most cases, using a `gpuConstantGammaClutter` System object™ is the only change you need to make.

If you have access to MATLAB Coder, you can also speed up clutter simulation by generating C code for `constantGammaClutter`, compiling it, and running the compiled version. When running in code generation mode, this example compiles stapclutter using the codegen command:

```
codegen('stapclutter','-args',...
          {coder.Constant(maxRange),...
           coder.Constant(patchAzSpan)});
```

All property values of `constantGammaClutter` have to be passed as constant values. The codegen command will generate the MEX file, stapclutter_mex, which will be called in a loop.

Comparing Clutter Simulation Times

To compare the clutter simulation performance between the MATLAB interpreter, code generation, and a GPU, launch the following GUI by typing `stapcpugpu` in the MATLAB command line. The launched GUI is shown in the following figure.
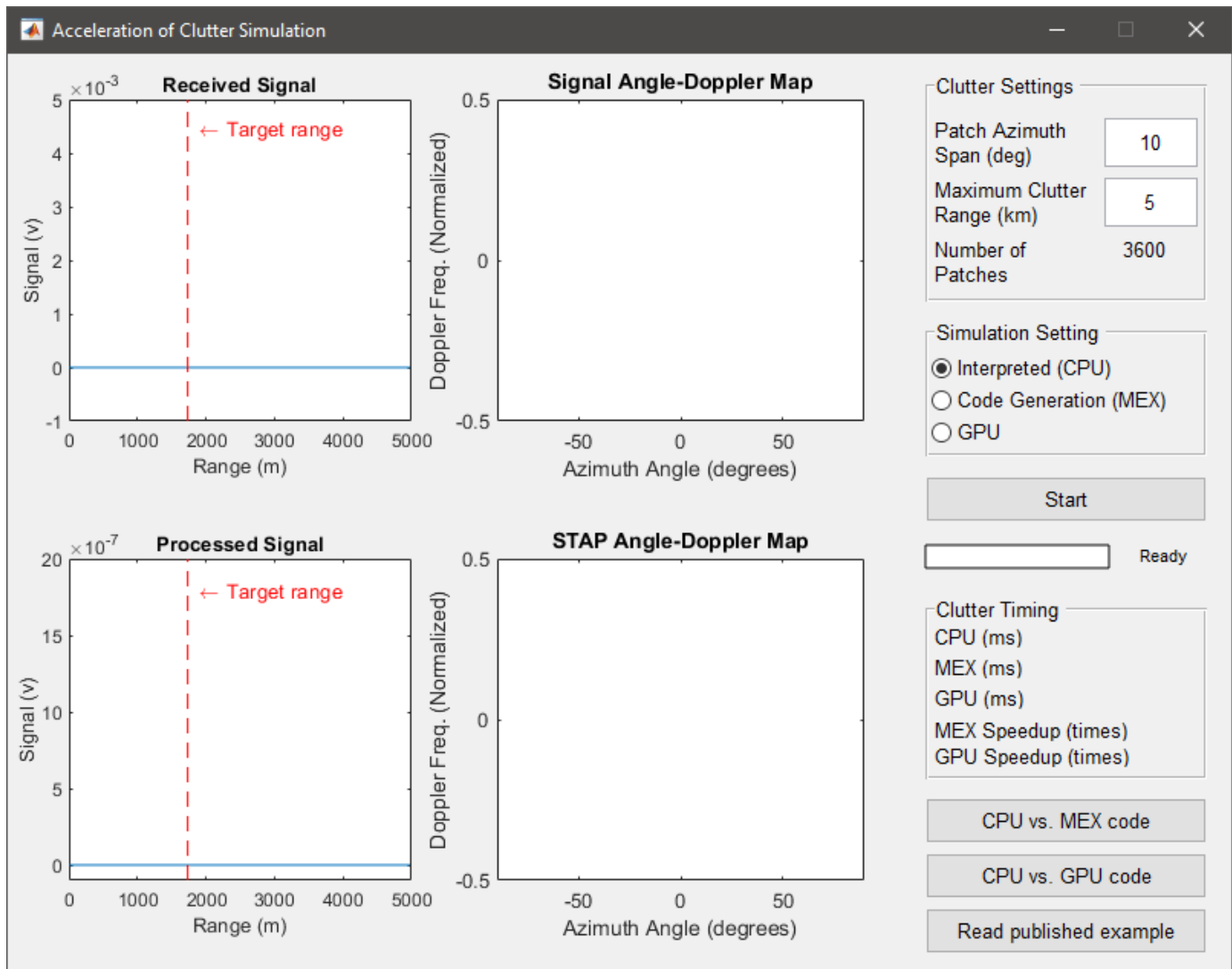
The left side of the GUI contains four plots, showing the raw received signal, the angle-Doppler response of the received signal, the processed signal, and the angle-Doppler response of the STAP processing weights. The details of the processing involved can be found in the example "Introduction to Space-Time Adaptive Processing". On the right side of the GUI, you control the number of clutter patches by modifying the clutter patch span in the azimuth direction (in degrees) and maximum clutter range (in km). You can then click the Start button to start the simulation, which simulates 5 coherent processing intervals (CPI) where each CPI contains 10 pulses. The processed signal and the angle-Doppler responses are updated once every CPI.

The next section shows the timing for different simulation runs. In these simulations, each pulse consists of 200 range samples with a range resolution of 50 m. Combinations of the clutter patch span and the maximum clutter range result in various number of total clutter patches. For example, a

clutter patch span of 10 degrees and a maximum clutter range of 5 km implies 3600 clutter patches. The simulations are carried out on the following system configurations:

- CPU: Xeon X5650, 2.66 GHz, 24 GB memory
- GPU: Tesla C2075, 6 GB memory

The timing results are shown in the following figure.

`helperCPUGPUResultPlot`
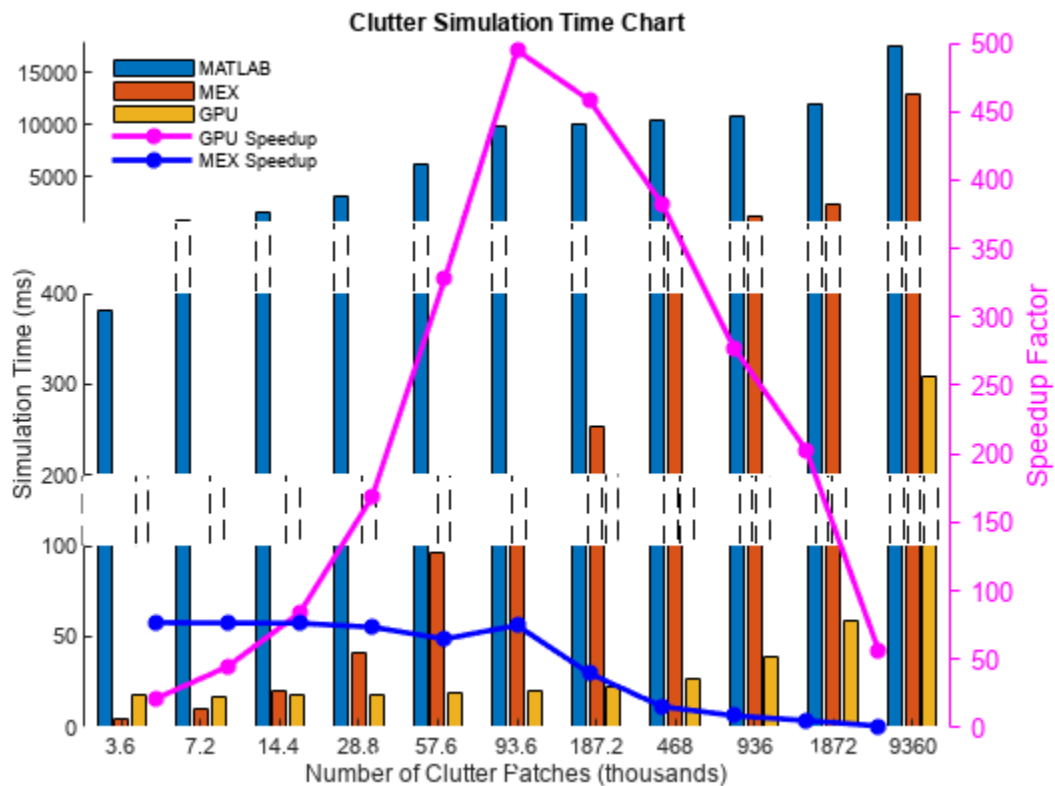
From the figure, you can see that in general the GPU improves the simulation speed by dozens of times, sometimes even hundred of times. Two interesting observations are:

- When the number of clutter patches are small, as long as the data can be fit into the GPU memory, the GPU's performance is almost constant. The same is not true for the MATLAB interpreter.
- Once the number of clutter patches gets large, the data can no longer be fit into the GPU memory. Therefore, the speed up provided by GPU over the MATLAB interpreter starts to decrease. However, for close to ten millions of clutter patches, the GPU still provides an acceleration of over 50 times.

Simulation speed improvement due to code generation is less than the GPU speed improvement, but is still significant. Code generation for `constantGammaClutter` precalculates the collected clutter as an array of constant values. For a larger number of clutter patches, the size of the array becomes too big, thus reducing the speed improvement due to the overhead of memory management. Code generation requires access to MATLAB Coder but requires no special hardware.

Other Simulation Timing Results

Even though the simulation used in this example calculates millions of clutter patches, the resulting data cube has a size of 200-by-6-by-10, indicating only 200 range samples within each pulse, 6 channels, and 10 pulses. This data cube is small compared to real problems. This example chooses these parameters to show the benefit you can get from using a GPU or code generation while ensuring that the example runs within a reasonable time in the MATLAB interpreter. Some simulations with larger data cube size yield the following results:

- 45-fold acceleration using a GPU for a simulation that generates 50 pulses for a 50-element ULA with 5000 range samples in each pulse, i.e., a 5000-by-50-by-50 data cube. The range resolution is 10 m. The radar covers a total azimuth of 60 degrees, with 1 degree in each clutter patch. The maximum clutter range is 50 km. The total number of clutter patches is 305,000.

- 60-fold acceleration using a GPU for a simulation like the one above, except with 180-degree azimuth coverage and a maximum clutter range equal to the horizon range (about 130 km). In this case, the total number of clutter patches is 2,356,801.

Summary

This example compares the performance achieved by simulating clutter return using either the MATLAB interpreter, a GPU, or code generation. The result indicates that the GPU and code generation offer big speed improvements over the MATLAB interpreter.

Modeling Target Radar Cross Section

This example shows how to model radar targets with increasing levels of fidelity. The example introduces the concept of radar cross sections (RCS) for simple point targets and extends it to more complicated cases of targets with multiple scattering centers. It also discusses how to model fluctuations of the RCS over time and briefly considers the case of polarized signals.

Introduction

A radar system relies on target reflection or scattering to detect and identify targets. The more strongly a target reflects, the greater the returned echo at the radar receiver, resulting in a higher signal-to-noise ratio (SNR) and likelier detection. In radar systems, the amount of energy reflected from a target is determined by the radar cross section (RCS), defined as

$$\sigma = \lim_{R \to \infty} 4\pi R^2 \frac{|E_s|^2}{|E_i|^2}$$

where σ represents the RCS, R is the distance between the radar and the target, E_s is the field strength of the signal reflected from the target, and E_i is the field strength of the signal incident on the target. In general, targets scatter energy in all directions and the RCS is a function of the incident angle, the scattering angle, and the signal frequency. RCS depends on the shape of the target and the materials from which it is constructed. Common units used for RCS include square meters or dBsm.

This example focuses on narrowband monostatic radar systems, when the transmitter and receiver are co-located. The incident and scattered angles are equal and the RCS is a function only of the incident angle. This is the *backscattered* case. For a narrowband radar, the signal bandwidth is small compared to the operating frequency and is therefore considered to be constant.

RCS of a Simple Point Target

The simplest target model is an isotropic scatterer. An example of an isotropic scatterer is a metallic sphere of uniform density. In this case, the reflected energy is independent of the incident angle. An isotropic scatterer can often serve as a first order approximation of a more complex point target that is distant from the radar. For example, a pedestrian can be approximated by an isotropic scatterer with a 1 square meter RCS.

```
c = 3e8;
fc = 3e8;
pedestrian = phased.RadarTarget('MeanRCS',1,'PropagationSpeed',c,...
    'OperatingFrequency',fc)
```

```
pedestrian =

  phased.RadarTarget with properties:

    EnablePolarization: false
        MeanRCSSource: 'Property'
              MeanRCS: 1
                Model: 'Nonfluctuating'
     PropagationSpeed: 300000000
    OperatingFrequency: 300000000
```

where c is the propagation speed and fc is the operating frequency of the radar system. The scattered signal from a unit input signal can then be computed as

```
x = 1;
ped_echo = pedestrian(x)
```

```
ped_echo =

    3.5449
```

where x is the incident signal. The relation between the incident and the reflected signal can be expressed as $y = \sqrt{G} * x$ where

$$G = \frac{4\pi\sigma}{\lambda^2}$$

G represents the dimensionless gain that results from the target reflection. λ is the wavelength corresponding to the system's operating frequency.

RCS of Complex Targets

For targets with more complex shapes, reflections can non longer be considered the same across all directions. The RCS varies with the incident angles (also known as aspect angles). Aspect-dependent RCS patterns can be measured or modeled just as you would antenna radiation patterns. The result of such measurements or models is a table of RCS values as a function of azimuth and elevation angles in the target's local coordinate system.

The example below first computes the RCS pattern of a cylindrical target, with a radius of 1 meter and a height of 10 meters, as a function of azimuth and elevation angles.
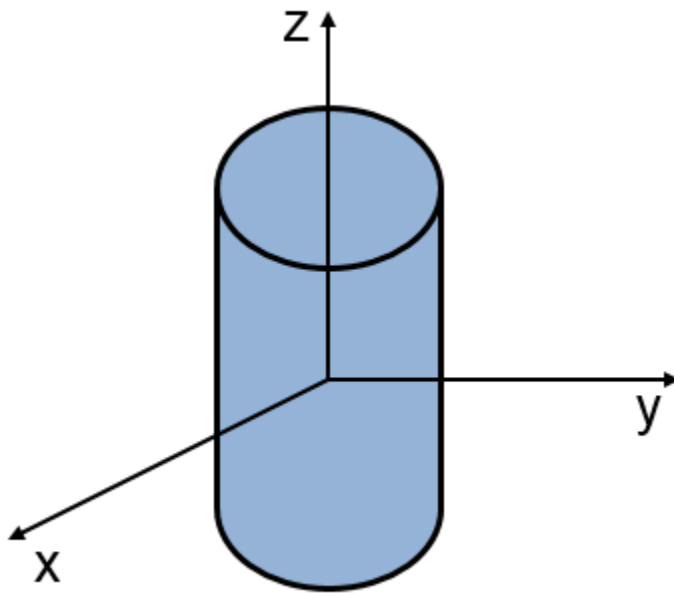
```
[cylrcs,az,el] = rcscylinder(1,1,10,c,fc);
```

Because the cylinder is symmetric around the z axis, there is no azimuth-angle dependency. RCS values vary only with elevation angle.

```
helperTargetRCSPatternPlot(az,el,cylrcs);
title('RCS Pattern of Cylinder');
```

RCS Pattern of Cylinder
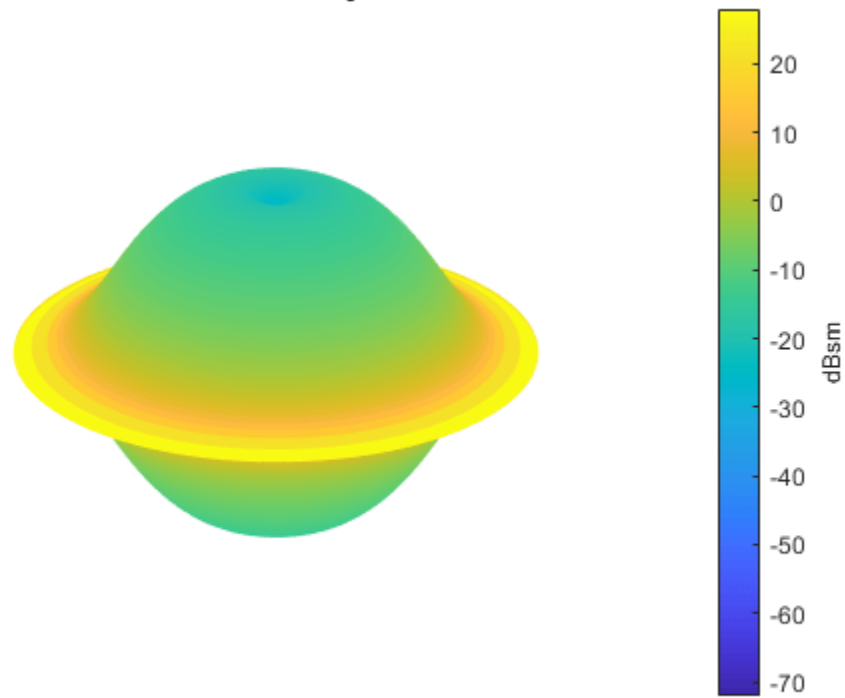
The pattern in the elevation cut looks like

```
plot(el,pow2db(cylrcs));
grid; axis tight; ylim([-30 30]);
xlabel('Elevation Angles (degrees)');
ylabel('RCS (dBsm)');
title('RCS Pattern for Cylinder');
```

RCS Pattern for Cylinder
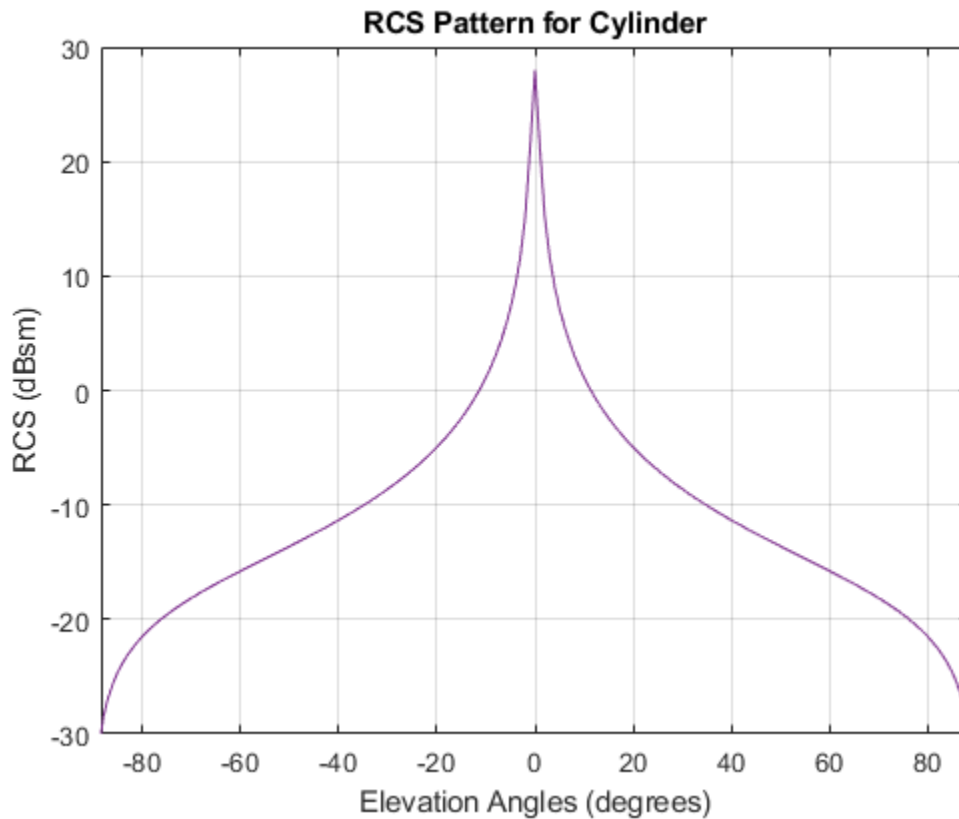
The aspect-dependent RCS patterns can then imported into a `phased.BackscatterRadarTarget` object.

```
cylindricalTarget = phased.BackscatterRadarTarget('PropagationSpeed',c,...
    'OperatingFrequency',fc,'AzimuthAngles',az,'ElevationAngles',el,...
    'RCSPattern',cylrcs)
```

```
cylindricalTarget =

  phased.BackscatterRadarTarget with properties:

    EnablePolarization: false
        AzimuthAngles: [-180 -179 -178 -177 -176 -175 -174 -173 -172 ... ]
      ElevationAngles: [-90 -89 -88 -87 -86 -85 -84 -83 -82 -81 -80 -79 ... ]
           RCSPattern: [181x361 double]
                Model: 'Nonfluctuating'
     PropagationSpeed: 300000000
    OperatingFrequency: 300000000
```

Finally, generate the target reflection. Assume three equal signals are reflected from the target at three different aspect angles. The first two angles have the same elevation angle but with different azimuth angles. The last has a different elevation angle from the first two.

```
x = [1 1 1];              % 3 unit signals
ang = [0 30 30;0 0 30]; % 3 directions
cyl_echo = cylindricalTarget(x,ang)
```

```
cyl_echo =

    88.8577    88.8577    1.3161
```

One can verify that there is no azimuth angle dependence because the first two outputs are the same.

The number of target shapes for which analytically-derived RCS patterns exist are few. For more complicated shapes and materials, computational electromagnetics approaches, such as method of moments (MoM), or finite element analysis (FEM), can be used to accurately predict the RCS pattern. A more detailed discussion of these techniques is available in [1]. You can use the output of these computations as input to the `phased.BackscatterRadarTarget` System object™ as was done in the cylinder example before.

RCS of Extended Targets with Multiple Scatterers

Although computational electromagnetic approaches can provide accurate RCS predictions, they often require a significant amount of computation and are not suitable for real-time simulations. An alternative approach for describing a complex targets is to model it as a collection of simple scatterers. The RCS pattern of the complex target can then be derived from the RCS patterns of the simple scatterer as [1]

$$\sigma = |\sum_p \sqrt{\sigma_p} e^{i\phi_p}|^2$$

where σ is the RCS of the target, σ_p is the RCS of the p th scatterer, and ϕ_p is the relative phase of the p th scatterer. A multi-scatterer target behaves much like an antenna array.

The next section shows how to model a target consisting of four scatterers. The scatterers are located at the four vertices of a square. Each scatterer is a cylindrical point target as derived in the previous section. Without loss of generality, the square is placed in the xy -plane. The side length of the square is 0.5 meter.
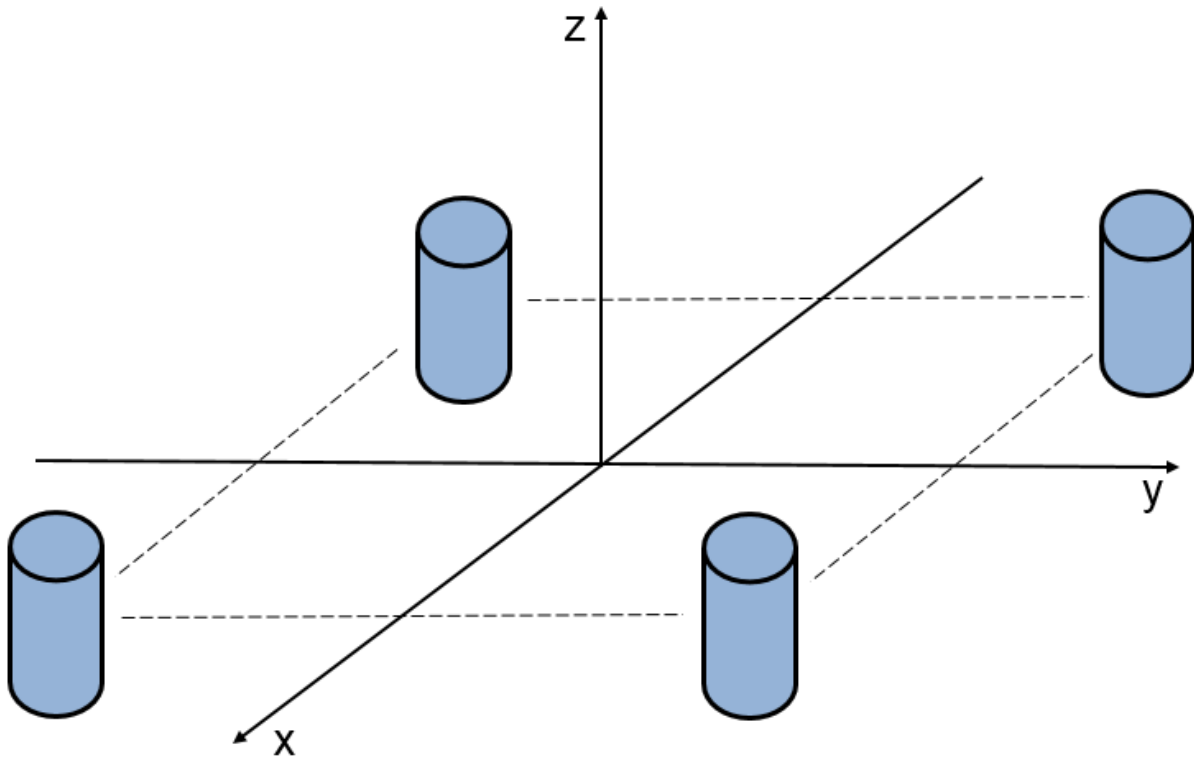
First, define the positions of the scatterers.

```
scatpos = [-0.5 -0.5 0.5 0.5;0.5 -0.5 0.5 -0.5;0 0 0 0];
```

If the target is in the far field of the transmitter, the incident angle for each component scatterer is the same. Then, the total RCS pattern can be computed as

```
naz = numel(az);
nel = numel(el);
extrcs = zeros(nel,naz);
for m = 1:nel
    sv = steervec(scatpos,[az;el(m)*ones(1,naz)]);
    % sv is squared due to round trip in a monostatic scenario
    extrcs(m,:) = abs(sqrt(cylrcs(m,:)).*sum(sv.^2)).^2;
end
```

The total RCS pattern then looks like

```
helperTargetRCSPatternPlot(az,el,extrcs);
title('RCS Pattern of Extended Target with 4 Scatterers');
```

RCS Pattern of Extended Target with 4 Scatterers
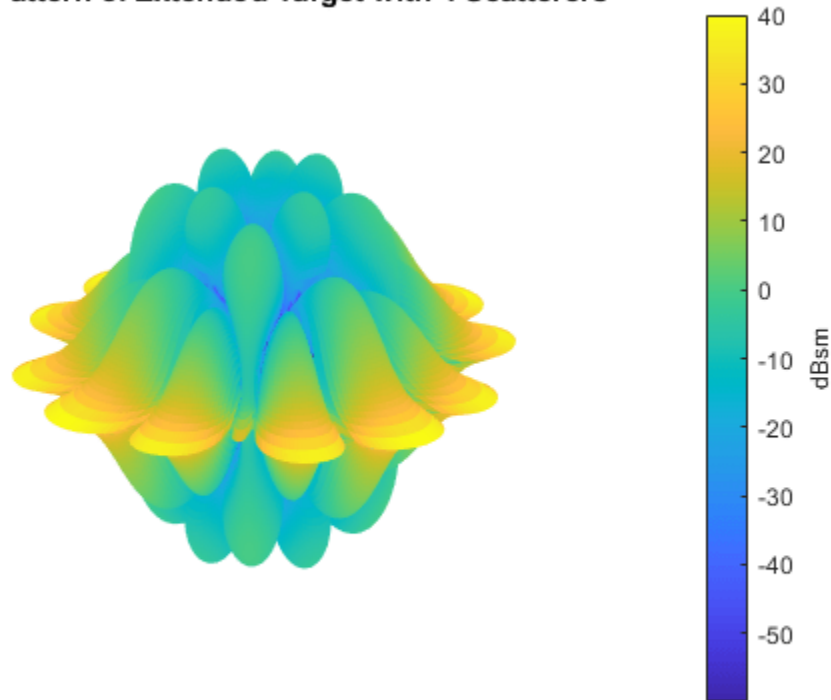
This pattern can then be used in a `phased.BackscatterRadarTarget` object to compute the reflected signal. The results verify that the reflected signal depends on both azimuth and elevation angles.

```
extendedTarget = phased.BackscatterRadarTarget('PropagationSpeed',c,...
    'OperatingFrequency',fc,'AzimuthAngles',az,'ElevationAngles',el,...
    'RCSPattern',extrcs);

ext_echo = extendedTarget(x,ang)
```

```
ext_echo =

  355.4306  236.7633    0.0000
```

Wideband RCS of Extended Targets with Multiple Scatterers

Wideband radar systems are typically defined as having a bandwidth greater than 5% of their center frequency. In addition to improved range resolution, wideband systems also offer improved target detection. One way in which wideband systems improve detection performance is by filling in fades in a target's RCS pattern. This can be demonstrated by revisiting the extended target comprised of 4 cylindrical scatterers used in the preceding section. The modeled narrowband RCS swept across various target aspects is shown as

```
sweepaz = -90:90; % Azimuthal sweep across target
sweepel = 0;
```

```matlab
[elg,azg] = meshgrid(sweepel,sweepaz);
sweepang = [azg(:)';elg(:)'];
x = ones(1,size(sweepang,2)); % unit signals

release(extendedTarget);
extNarrowbandSweep = extendedTarget(x,sweepang);

clf;
plot(sweepaz,pow2db(extNarrowbandSweep));
grid on; axis tight;
xlabel('Azimuth Angles (degrees)');
ylabel('RCS (dBsm)');
title(['RCS Pattern at 0^o Elevation ',...
    'for Extended Target with 4 Scatterers']);
```
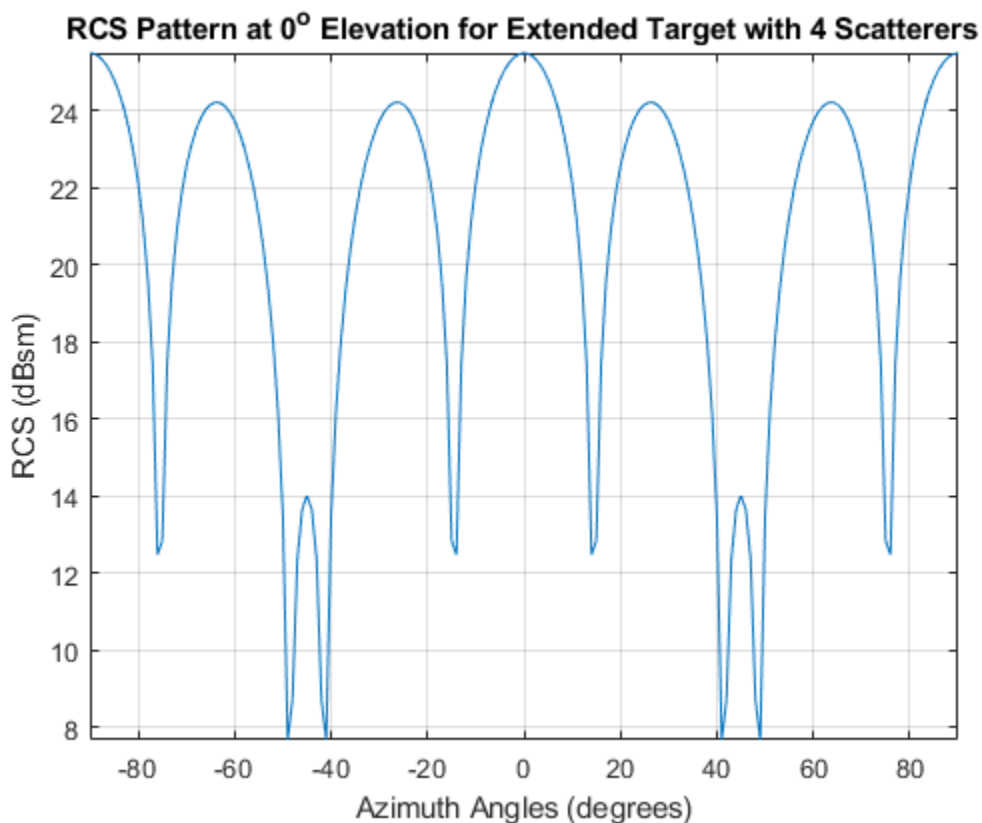

Returns from the multiple cylinders in the extended target model coherently combine, creating deep fades between 40 and 50 degrees. These fades can cause the target to not be detected by the radar sensor.

Next, the RCS pattern for a wideband system operating at the same center frequency will be examined. The bandwidth for this system will be set to 10% of the center frequency

```matlab
bw = 0.10*fc; % Bandwidth is greater-than 5% of center frequency
fs = 2*bw;
```

A wideband RCS model is created as was previously done for the narrowband extended target. Often, RCS models are generated offline using either simulation tools or range measurements and are then

provided to radar engineers for use in their system models. Here, it is assumed that the provided RCS model has been sampled at 1MHz intervals on either side of the radar's center frequency.

```
modelFreq = (-80e6:1e6:80e6)+fc;
[modelCylRCS,modelAz,modelEl] = helperCylinderRCSPattern(c,modelFreq);
```

The contributions from the various scattering centers are modeled as before. It is important to note that this approximation assumes that all of the target's scattering centers fall within the same range resolution bin, which is true for this example.

```
nf = numel(modelFreq);
naz = numel(modelAz);
nel = numel(modelEl);
modelExtRCS = zeros(nel,naz,nf);
for k = 1:nf
    for m = 1:nel
        pos = scatpos*modelFreq(k)/fc;
        sv = steervec(pos,[modelAz;modelEl(m)*ones(1,naz)]);
        % sv is squared due to round trip in a monostatic scenario
        modelExtRCS(m,:,k) = abs(sqrt(modelCylRCS(m,:,k)).*sum(sv.^2)).^2;
    end
end
```

The wideband RCS target model is now generated, using the RCS patterns that were just computed.

```
widebandExtendedTarget = phased.WidebandBackscatterRadarTarget(...
    'PropagationSpeed',c,'OperatingFrequency',fc,'SampleRate',fs,...
    'AzimuthAngles',modelAz,'ElevationAngles',modelEl,...
    'FrequencyVector',modelFreq,'RCSPattern',modelExtRCS);
```

The modeled wideband RCS can now be compared to the narrowband system

```
extWidebandSweep = widebandExtendedTarget(x,sweepang);
```

```
hold on;
plot(sweepaz,pow2db(extWidebandSweep));
hold off;
legend('Narrowband','Wideband');
```
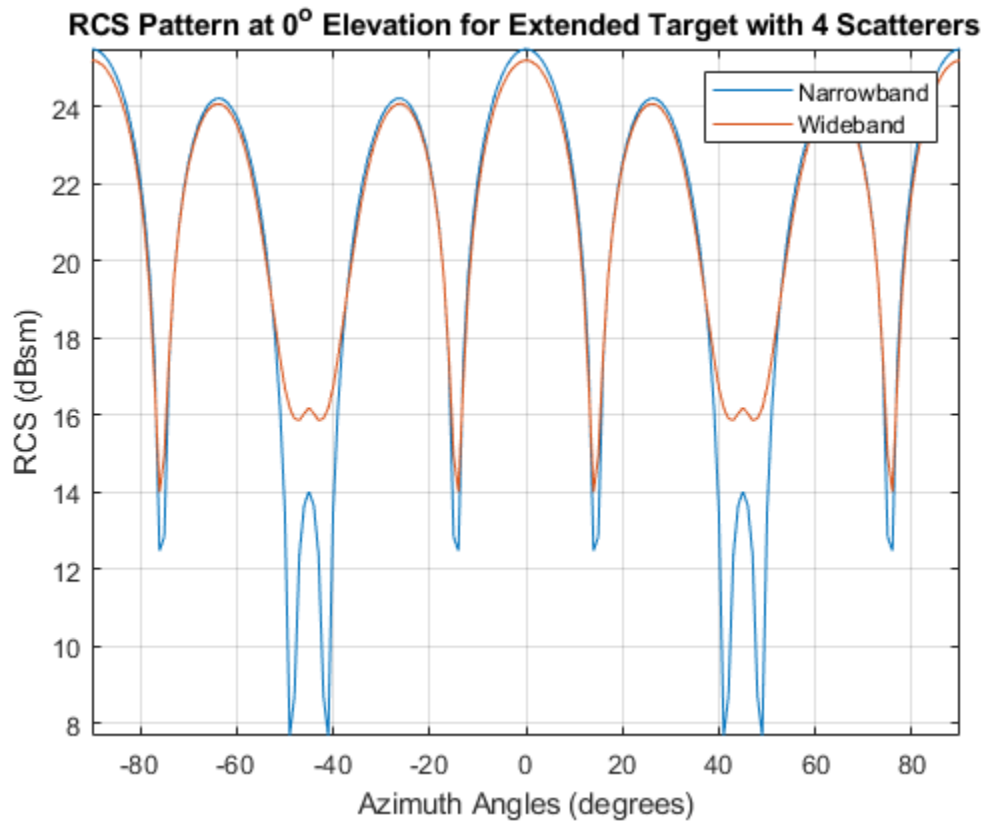
The target's RCS pattern now has much shallower nulls between 40 and 50 degrees azimuth. The deep nulls in the narrowband pattern occur when signals combine destructively at a specific frequency and azimuth combination. The wideband waveform fills in these fades because, while a few frequencies may experience nulls for a given aspect, the majority of the bandwidth does not lie within the null at that azimuth angle.

RCS of Fluctuating Targets

The discussion so far assumes that the target RCS value is constant over time. This is the nonfluctuating target case. In reality, because both the radar system and the target are moving, the RCS value changes over time. This case is a fluctuating target. To simulate fluctuating targets, Peter Swerling developed four statistical models, referred to as Swerling 1 through Swerling 4, that are widely adopted in practice. The Swerling models divide fluctuating targets into two probability distributions and two time varying behaviors as shown in the following table:

```
                        Slow Fluctuating     Fast Fluctuating
      ------------------------------------------------------------
            Exponential     Swerling 1           Swerling 2
  4th Degree Chi-square     Swerling 3           Swerling 4
```

The RCS of a slow-fluctuating target remains constant during a dwell but varies from scan to scan. In contrast, the RCS for a fast fluctuating target changes with each pulse within a dwell.

The Swerling 1 and 2 models obey an exponential density function (pdf) given by

$$p(\sigma) = \frac{1}{\mu_\sigma} e^{-sigma/\mu_\sigma}$$,

These models are useful in simulating a target consisting of a collection of equal strength scatterers.

The Swerling 3 and 4 models obey a 4th degree Chi-square pdf, given by

$$p(\sigma) = \frac{4\sigma}{\mu_\sigma^2} e^{-2\sigma/\mu_\sigma}$$,

These models apply when the target contains a dominant scattering component. In both pdf definitions, μ_σ represents the mean RCS value, which is the RCS value of the same target under the nonfluctuating assumption.

The next section shows how to apply a Swerling 1 statistical model when generating the radar echo from the previously described cylindrical target.

```
cylindricalTargetSwerling1 = ...
    phased.BackscatterRadarTarget('PropagationSpeed',c,...
    'OperatingFrequency',fc,'AzimuthAngles',az,'ElevationAngles',el,...
    'RCSPattern',cylrcs,'Model','Swerling1')


cylindricalTargetSwerling1 =

  phased.BackscatterRadarTarget with properties:

    EnablePolarization: false
         AzimuthAngles: [-180 -179 -178 -177 -176 -175 -174 -173 -172 ... ]
       ElevationAngles: [-90 -89 -88 -87 -86 -85 -84 -83 -82 -81 -80 -79 ... ]
            RCSPattern: [181x361 double]
                 Model: 'Swerling1'
      PropagationSpeed: 300000000
    OperatingFrequency: 300000000
            SeedSource: 'Auto'
```

In the Swerling 1 case, the reflection is no longer constant. The RCS value varies from scan to scan. Assuming that the target is illuminated by the signal only once per dwell, the following code simulates the reflected signal power for 10,000 scans for a unit incident signal.

```
N = 10000;
tgt_echo = zeros(1,N);
x = 1;
for m = 1:N
    tgt_echo(m) = cylindricalTargetSwerling1(x,[0;0],true);
end
p_echo = tgt_echo.^2; % Reflected power
```

Plot the histogram of returns from all scans and verify that the distribution of the returns match the theoretical prediction. The theoretical prediction uses the nonfluctuating RCS derived before. For the cylindrical target, the reflected signal power at normal incidence for unit power input signal is

```
p_n = cyl_echo(1)^2;
```

```
helperTargetRCSReturnHistogramPlot(p_echo,p_n)
```
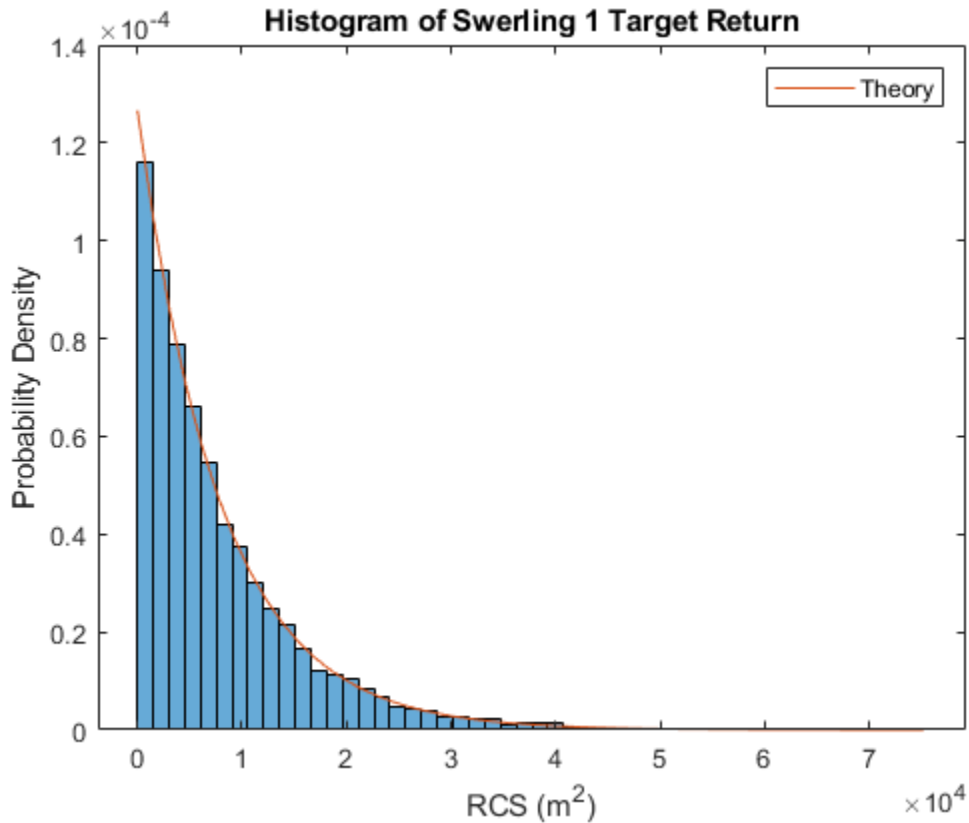
RCS of Polarized Targets

The target RCS is also a function of polarization. To describe the polarization signature of a target, a single RCS value is no longer sufficient. Instead, for each frequency and incident angle, a scattering matrix is used to describe the interaction of the target with the incoming signal's polarization components. This example will not go into further details because this topic is covered in the "Modeling and Analyzing Polarization" example.

Conclusion

This example gave a brief introduction to radar target modeling for a radar system simulation. It showed how to model point targets, targets with measured patterns, and extended targets. It also described how to take statistical fluctuations into account when generating target echoes.

Reference

[1] Merrill Skolnik, Radar Handbook, 2nd Ed. Chapter 11, McGraw-Hill, 1990

[2] Bassem Mahafza, Radar Systems Analysis and Design Using MATLAB, 2nd Ed. Chapman & Hall/ CRC, 2005

Simulating Bistatic Polarimetric Radar

This example shows how to simulate a polarimetric bistatic radar system to estimate the range and speed of targets. Transmitter, receiver, and target kinematics are taken into account. For more information regarding polarization modeling capabilities, see "Modeling and Analyzing Polarization".

System Setup

The system operates at 300 MHz, using a linear FM waveform whose maximum unambiguous range is 48 km. The range resolution is 50 meters and the time-bandwidth product is 20.

```
maxrng = 48e3;          % Maximum range
rngres = 50;            % Range resolution
tbprod = 20;            % Time-bandwidth product
```

The transmitter has a peak power of 2 kw and a gain of 20 dB. The receiver also provides a gain of 20 dB and the noise bandwidth is the same as the sweep bandwidth of the waveform.

The transmit antenna array is a stationary four-element Uniform Linear Arra (ULA) located at the origin. The array is made of vertical dipoles.

```
txAntenna = phased.ShortDipoleAntennaElement('AxisDirection','Z');
[waveform,transmitter,txmotion,radiator] = ...
    helperBistatTxSetup(maxrng,rngres,tbprod,txAntenna);
```

The receive antenna array is also a four-element ULA. It is located [20000;1000;100] meters away from the transmit antenna and is moving at a velocity of [0;20;0] m/s. Assume the elements in the receive array are also vertical dipoles. The received antenna array is oriented so that its broadside points back to the transmit antenna.

```
rxAntenna = phased.ShortDipoleAntennaElement('AxisDirection','Z');
[collector,receiver,rxmotion,rngdopresp,beamformer] = ...
    helperBistatRxSetup(rngres,rxAntenna);
```

There are two targets present in space. The first one is a point target modeled as a sphere; it preserves the polarization state of the incident signal. It is located at [15000;1000;500] meters away from the transmit array and is moving at a velocity of [100;100;0] m/s.

The second target is located [35000;-1000;1000] meters away from the transmit array and is approaching at a velocity of [-160;0;-50] m/s. Unlike the first target, the second target flips the polarization state of the incident signal, which means that the horizontal polarization component of the input signal becomes the vertical polarization components of the output signal and vice versa.

```
[target,tgtmotion,txchannel,rxchannel] = ...
    helperBistatTargetSetup(waveform.SampleRate);
```

A single scattering matrix is a fairly simple polarimetric model for a target. It assumes that no matter what the incident and reflecting directions are, the power distribution between the H and V components is fixed. However, even such a simple model can reveal complicated target behavior in the simulation because, firstly the H and V directions vary for different incident and reflecting directions; and secondly, the orientation of the targets, defined by the local coordinate system, also affects the polarization matching.

System Simulation

The next section simulates 256 received pulses. The receiving array is beamformed toward the two targets. The first figure shows the system setting and how the receive array and the targets move. The second figure shows a range-Doppler map generated for every 64 pulses received at the receiver array.

```matlab
Nblock = 64; % Burst size
dt = 1/waveform.PRF;
y = complex(zeros(round(waveform.SampleRate*dt),Nblock));

hPlots = helperBistatViewSetup(txmotion,rxmotion,tgtmotion,waveform,...
    rngdopresp,y);
Npulse = Nblock*4;
for m = 1:Npulse

    % Update positions of transmitter, receiver, and targets
    [tpos,tvel,txax] = txmotion(dt);
    [rpos,rvel,rxax] = rxmotion(dt);
    [tgtp,tgtv,tgtax] = tgtmotion(dt);

    % Calculate the target angles as seen by the transmitter
    [txrng,radang] = rangeangle(tgtp,tpos,txax);

    % Simulate propagation of pulse in direction of targets
    wav = waveform();
    wav = transmitter(wav);
    sigtx = radiator(wav,radang,txax);
    sigtx = txchannel(sigtx,tpos,tgtp,tvel,tgtv);

    % Reflect pulse off of targets
    for n = 2:-1:1
        % Calculate bistatic forward and backward angles for each target
        [~,fwang] = rangeangle(tpos,tgtp(:,n),tgtax(:,:,n));
        [rxrng(n),bckang] = rangeangle(rpos,tgtp(:,n),tgtax(:,:,n));

        sigtgt(n) = target{n}(sigtx(n),fwang,bckang,tgtax(:,:,n));
    end

    % Receive path propagation
    sigrx = rxchannel(sigtgt,tgtp,rpos,tgtv,rvel);
    [~,inang] = rangeangle(tgtp,rpos,rxax);

    rspeed_t = radialspeed(tgtp,tgtv,tpos,tvel);
    rspeed_r = radialspeed(tgtp,tgtv,rpos,rvel);

    % Receive target returns at bistatic receiver
    sigrx = collector(sigrx,inang,rxax);
    yc = beamformer(sigrx,inang);
    y(:,mod(m-1,Nblock)+1) = receiver(sum(yc,2));

    helperBistatViewTrajectory(hPlots,tpos,rpos,tgtp);

    if ~rem(m,Nblock)
        rd_rng = (txrng+rxrng)/2;
        rd_speed = rspeed_t+rspeed_r;
        helperBistatViewSignal(hPlots,waveform,rngdopresp,y,rd_rng,...
            rd_speed)
```

```
        end
    end
```
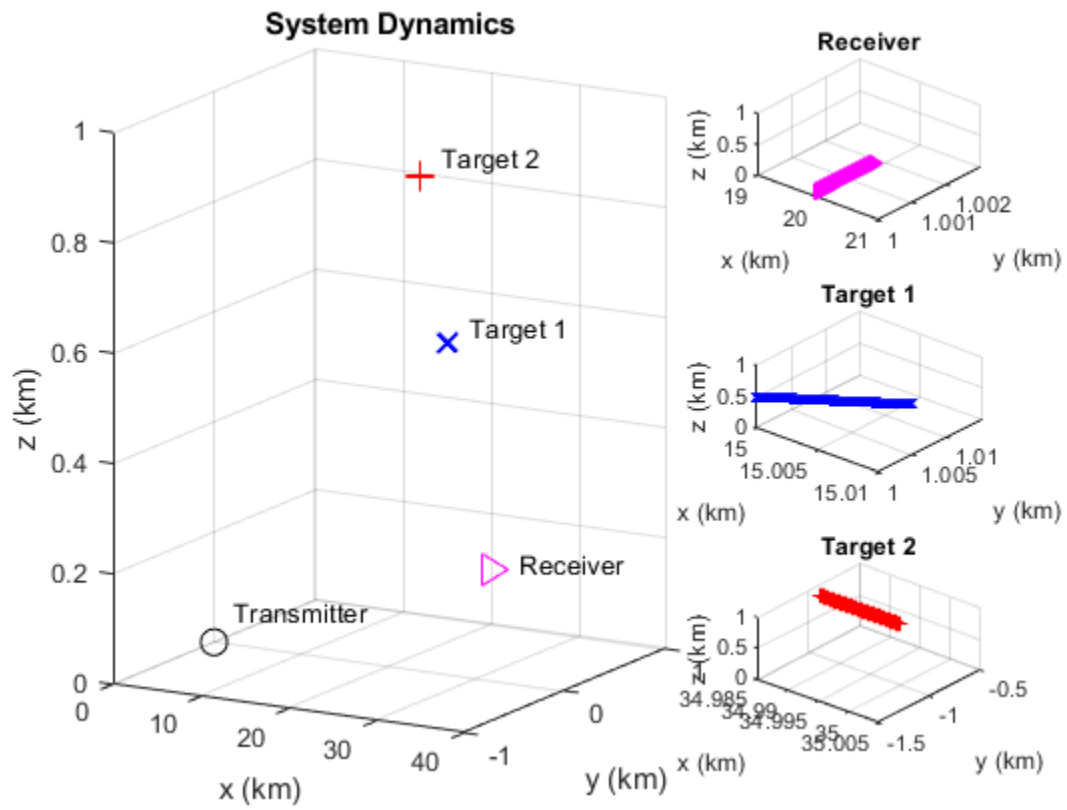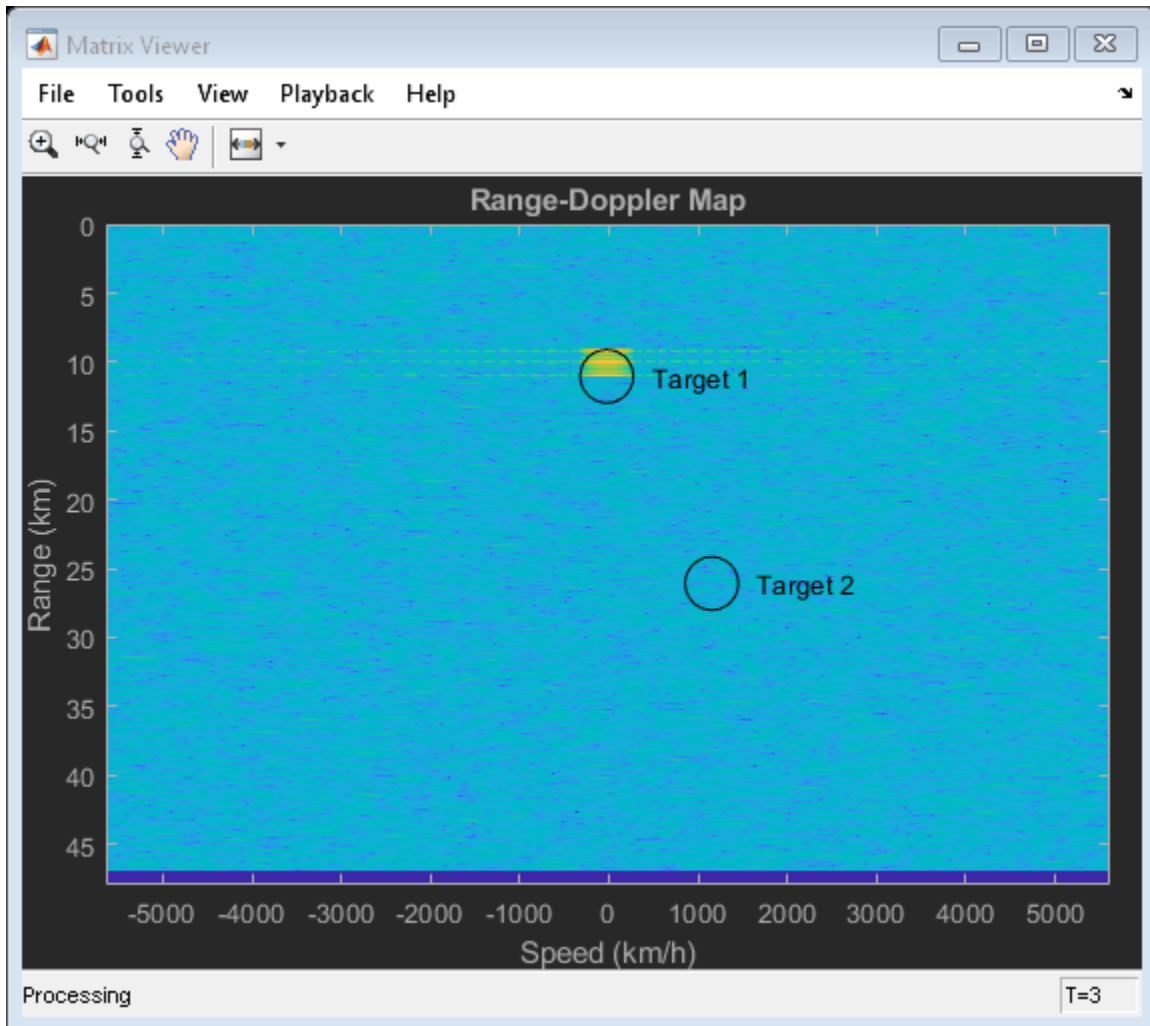
The range-Doppler map only shows the return from the first target. This is probably no surprise since both the transmit and receive array are vertically polarized and the second target maps the vertically polarized wave to horizontally polarized wave. The received signal from the second target is mostly orthogonal to the polarization of the receive array resulting in significant polarization loss.

You might also notice that the resulting range and radial speed do not agree with the range and radial speed of the target relative to the transmitter. This is because in a bistatic configuration, the estimated range is actually the geometric mean of the target range relative to the transmitter and the receiver. Similarly, the estimated radial speed is the sum of the target radial speed relative to the transmitter and the receiver. The circle in the map shows where the targets should appear in the range-Doppler map. Further processing is required to identify the exact location of the target, but that is beyond the scope of this example.

Using Circularly Polarized Receive Array

Vertical dipole is a very popular choice of transmit antenna in real applications because it is low cost and has a omnidirectional pattern. However, the previous simulation shows that if the same antenna is used in the receiver, there is a risk that the system will miss certain targets. Therefore, a linear polarized antenna is often not the best choice as the receive antenna in such a configuration because no matter how the linear polarization is aligned, there always exists an orthogonal polarization. In

case the reflected signal bears a polarization state close to that direction, the polarization loss becomes large.

One way to solve this issue is to use a circularly polarized antenna at the receive end. A circularly polarized antenna cannot fully match any linear polarization. But on the other hand, the polarization loss between a circular polarized antenna and a linearly polarized signal is 3 dB, regardless what direction the linear polarization is in. Therefore, although it never gives the maximum return, it never misses a target. A frequently used antenna with circular polarization is a crossed dipole antenna.

The next section shows what happens when crossed dipole antennas are used to form the receive array.

```
rxAntenna = phased.CrossedDipoleAntennaElement;
collector = clone(collector);
collector.Sensor.Element = rxAntenna;

helperBistatSystemRun(waveform,transmitter,txmotion,radiator,collector,...
    receiver,rxmotion,rngdopresp,beamformer,target,tgtmotion,txchannel,...
    rxchannel,hPlots,Nblock,Npulse);
```
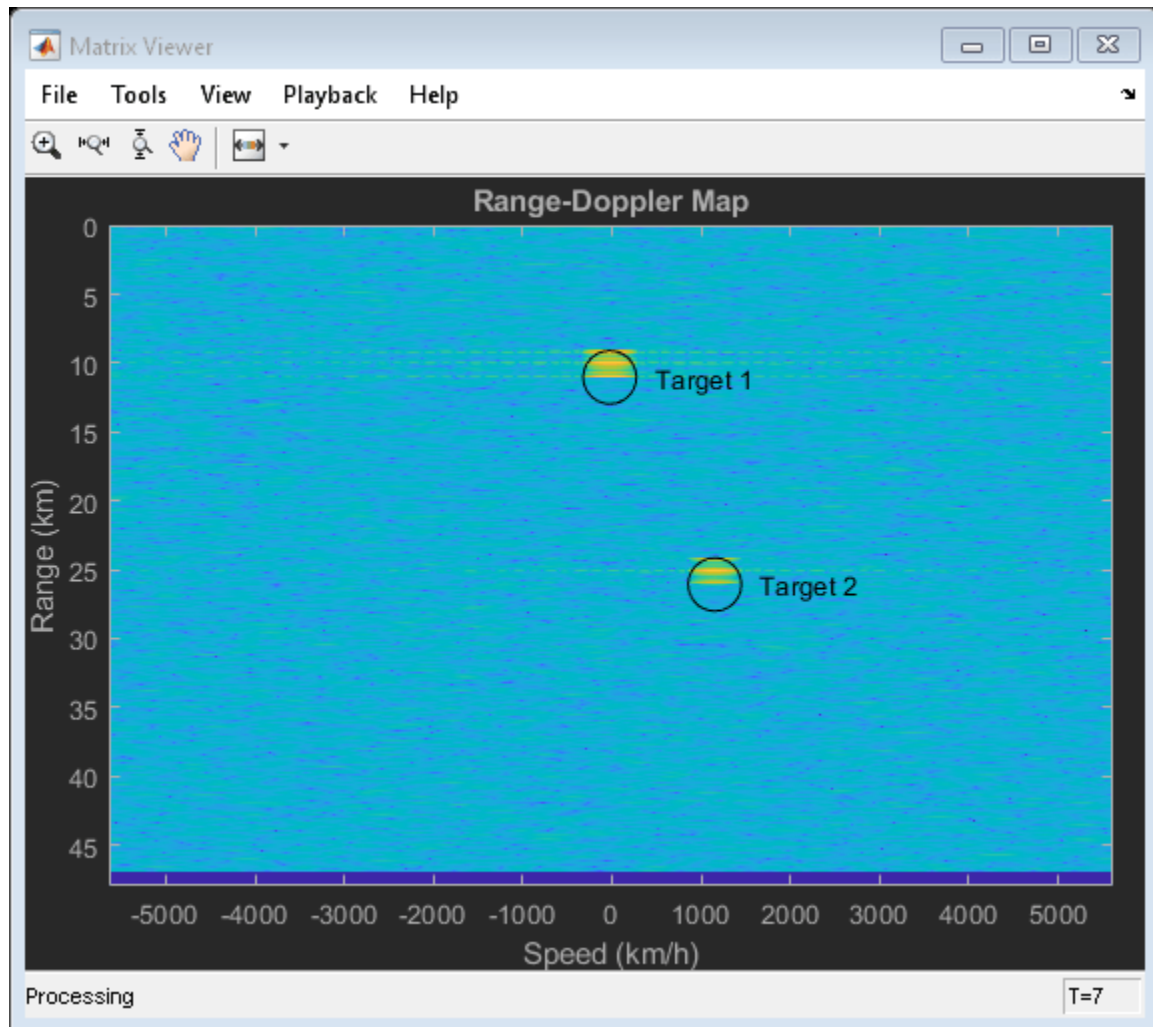

The range-Doppler map now shows both targets at their correct locations.

Summary

This example shows a system level simulation of a bistatic polarimetric radar. The example generates range-Doppler maps of the received signal for different transmit and receive array polarization configurations and shows how a circularly polarized antenna can be used to avoid losing linear polarized signals due to the polarization scattering property of a target.

Simulating a Bistatic Radar with Two Targets

This example shows how to simulate a bistatic radar system with two targets. The transmitter and the receiver of a bistatic radar are not co-located and move along different paths.

Exploring the Example

The following model shows an end-to-end simulation of a bistatic radar system. The system is divided into three parts: the transmitter subsystem, the receiver subsystem, and the targets and their propagation channels. The model shows the signal flowing from the transmitter, through the channels to the targets and reflected back to the receiver. Range-Doppler processing is then performed at the receiver to generate the range-Doppler map of the received echoes.
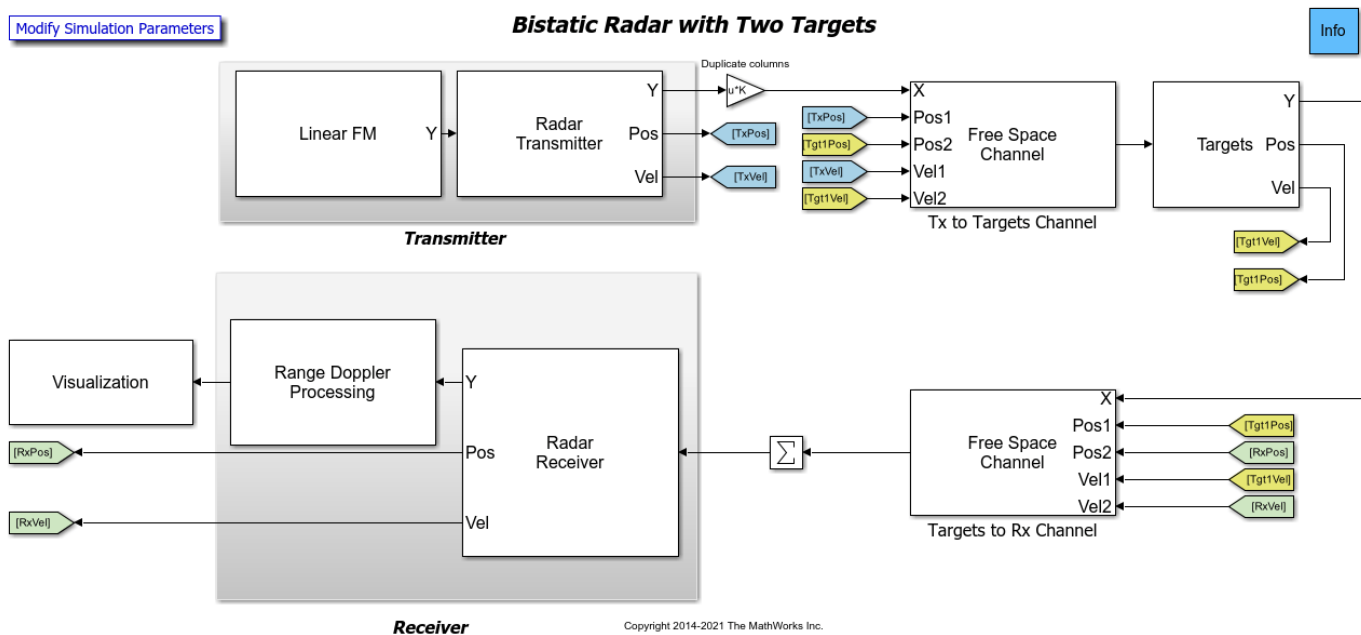
Transmitter

- `Linear FM` - Creates linear FM pulse as the transmitter waveform. The signal sweeps a 3 MHz bandwidth, corresponding to a 50-meter range resolution.
- `Radar Transmitter` - Amplifies the pulse and simulates the transmitter motion. In this case, the transmitter is mounted on a stationary platform located at the origin. The operating frequency of the transmitter is 300 MHz.
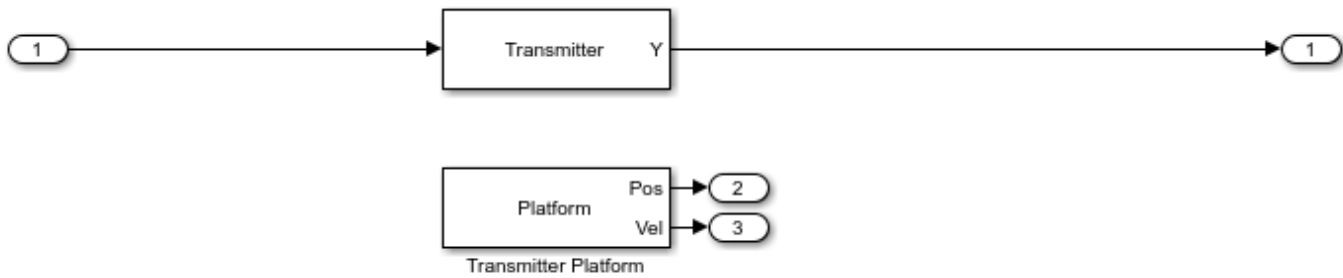
Targets

This example includes two targets with similar configurations. The targets are mounted on the moving platforms.

- `Tx to Targets Channel` - Propagates signal from the transmitter to the targets. The signal inputs and outputs of the channel block have two columns, one column for the propagation path to each target.

- `Targets to Rx Channel` - Propagates signal from the targets to the receiver. The signal inputs and outputs of the channel block have two columns, one column for the propagation path from each target.

- `Targets` - Reflects the incident signal and simulates both targets motion. This first target with an RCS of 2.5 square meters is approximately 15 km from the transmitter and is moving at a speed of 141 m/s. The second target with an RCS of 4 square meters is approximately 35 km from the transmitter and is moving at a speed of 168 m/s. The RCS of both targets are specified as a vector of two elements in the Mean radar cross section parameter of the underlying Target block.
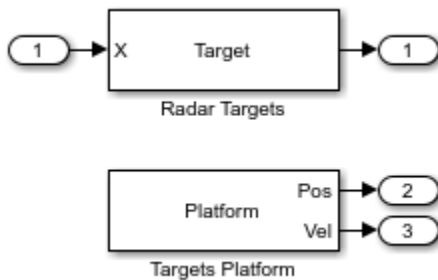
Receiver

- `Radar Receiver` - Receives the target echo, adds receiver noise, and simulates the receiver motion. The distance between the transmitter and the receiver is 20 km, and the receiver is moving at a speed of 20 m/s. The distance between the receiver and the two targets are approximately 5 km and 15 km, respectively.

1-595
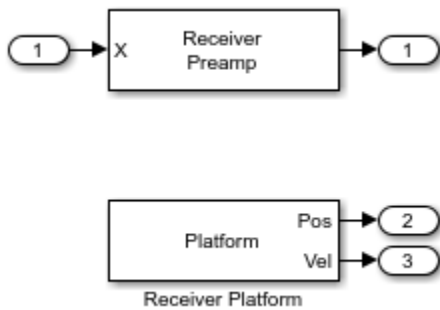
- `Range-Doppler Processing` - Computes the range-Doppler map of the received signal. The received signal is buffered to form a 64-pulse burst which is then passed to a range-Doppler processor. The processor performs a matched filter operation along the range dimension and an FFT along the Doppler dimension.
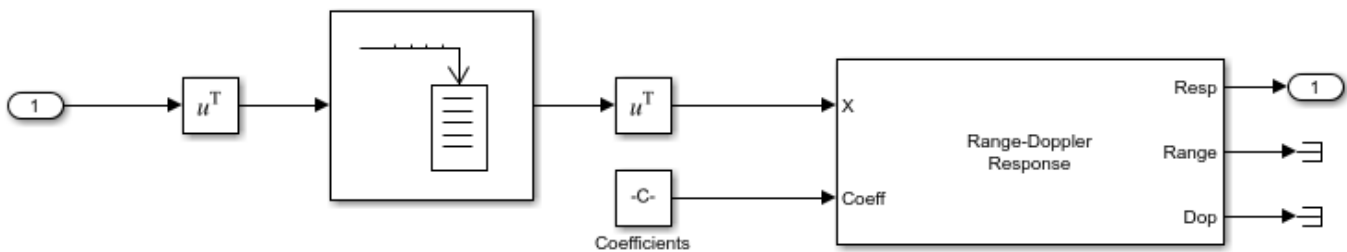
Exploring the Model

Several dialog parameters of the model are calculated by the helper function helperslexBistaticParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

The figure below shows the two targets in the range-Doppler map, range-time intensity plot and Doppler-time intensity plot.
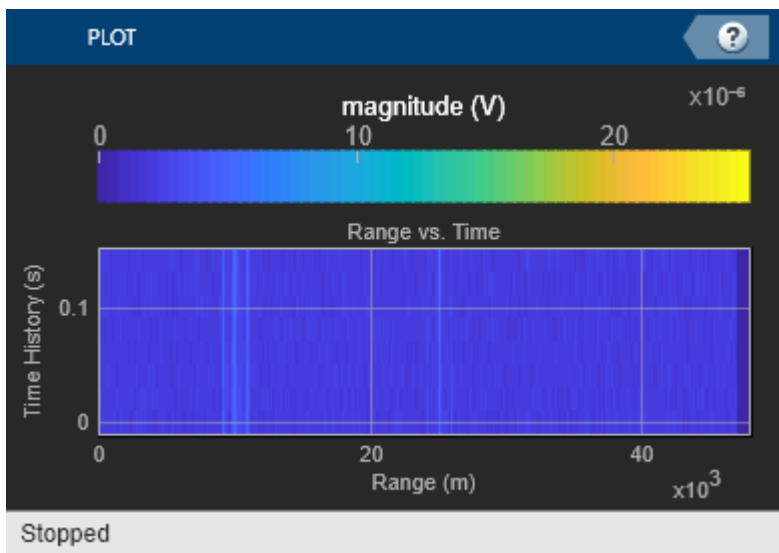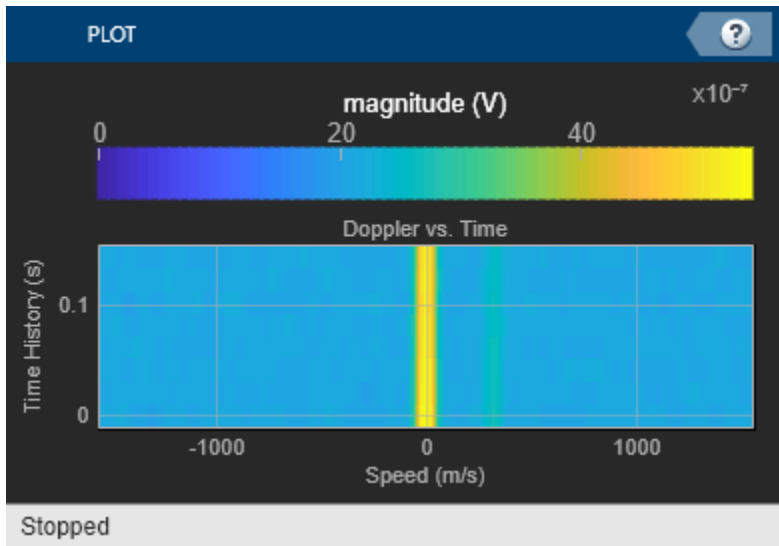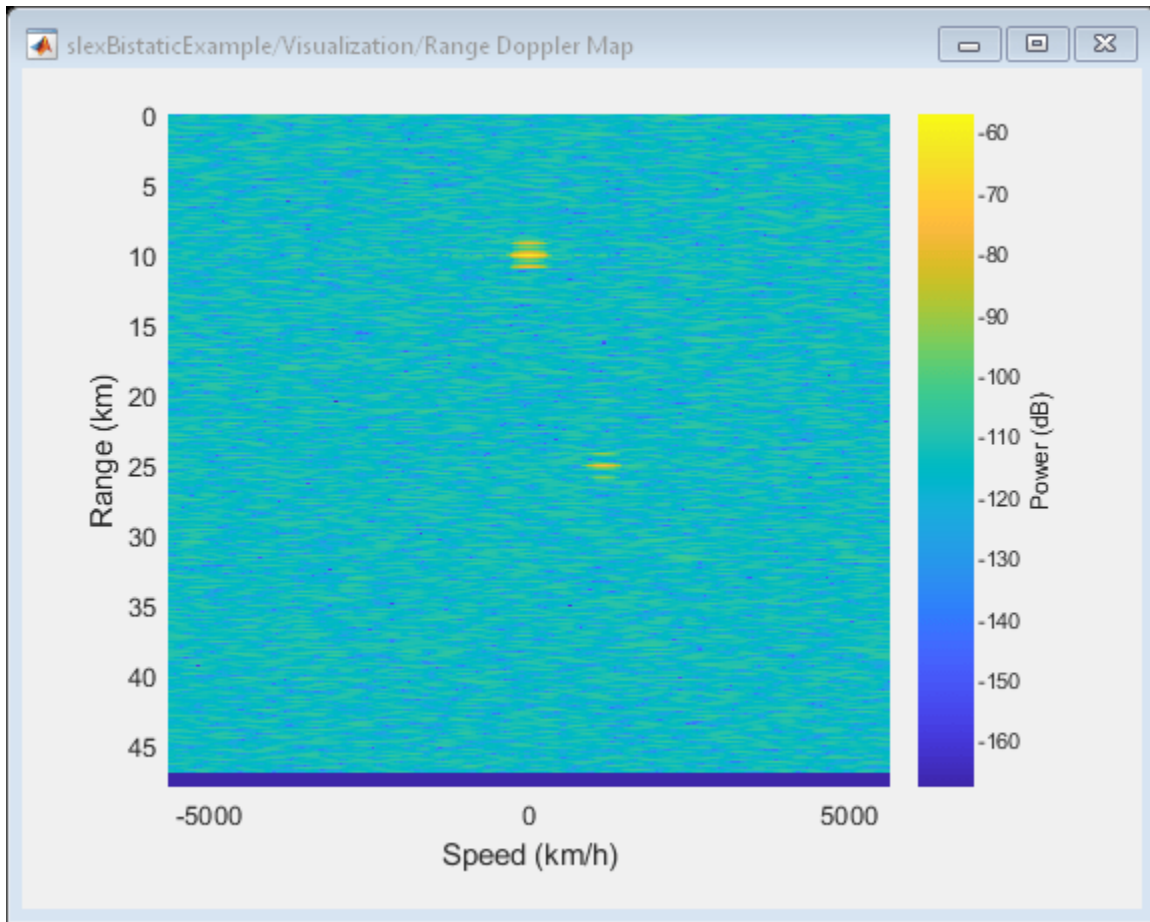
Because this is a bistatic radar, the range-Doppler map above actually shows the target range as the arithmetic mean of the distances from the transmitter to the target and from the target to the receiver. Therefore, the expected range of the first target is approximately 10 km, ((15+5)/2) and for second target approximately 25 km, ((35+15)/2). The range-Doppler map shows these two values as the measured values.

Similarly, the Doppler shift of a target in a bistatic configuration is the sum of the target's Doppler shifts relative to the transmitter and the receiver. The relative speeds to the transmitter are -106.4 m/s for the first target and 161.3 m/s for the second target while the relative speeds to the receiver are 99.7 m/s for the first target and 158.6 m/s for second target. Thus, the range-Doppler map shows the overall relative speeds as -6.7 m/s (-24 km/h) and 319.9 m/s (1152 km/h) for the first target and the second target, respectively, which agree with the expected sum values.

The Range-Time Intensity scope and Doppler-Time Intensity scope shows the scrolling range and Doppler response intensities over time which matches the measured values range and speed values.
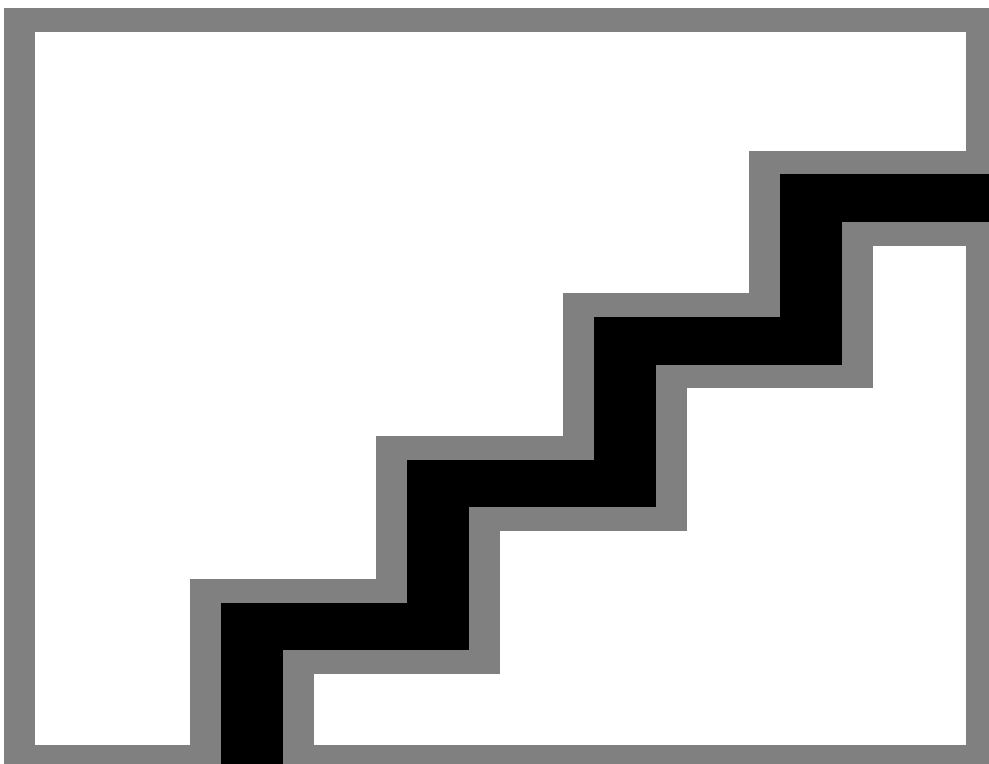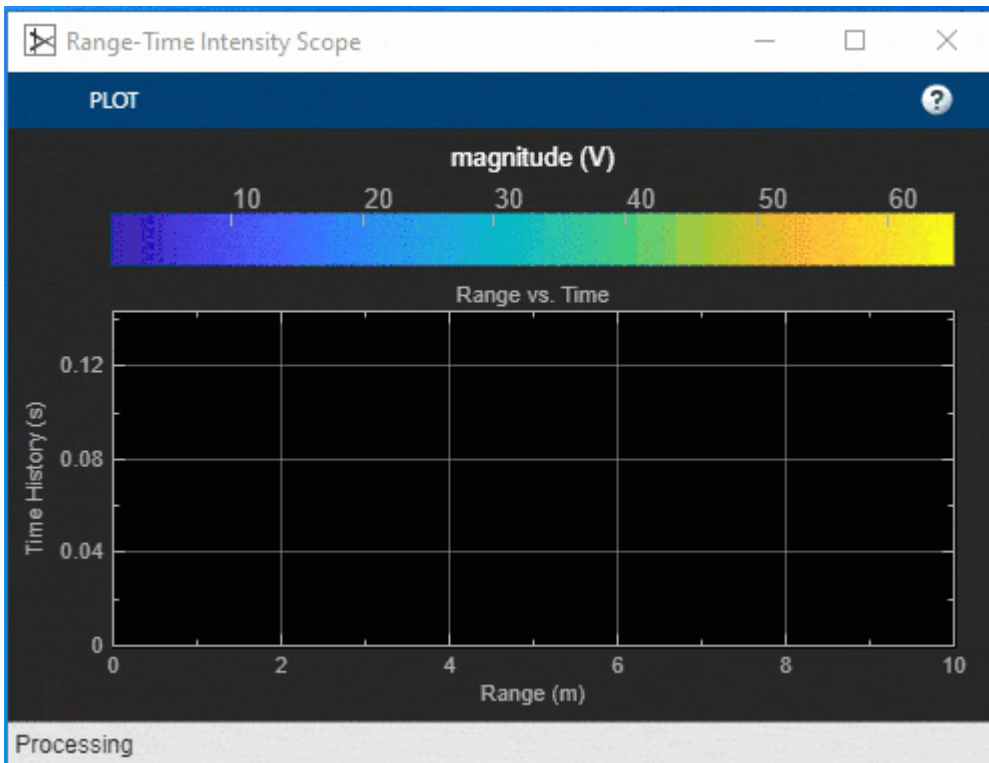
Summary

This example shows an end-to-end bistatic radar system simulation with two targets. It explains how to analyze the target return by plotting a range-Doppler map.

Modeling a Wideband Monostatic Radar in a Multipath Environment

This example shows how to simulate a wideband radar system. A radar system is typically considered to be wideband when its bandwidth exceeds 5% of the system's center frequency. For this example, a bandwidth of 10% will be used.

Exploring the Example

This example expands on the narrowband monostatic radar system explored in the "Simulating Test Signals for a Radar Receiver in Simulink" example by modifying it for wideband radar simulation. For wideband signals, both propagation losses and target RCS can vary considerably across the system's bandwidth. It is for this reason that narrowband models cannot be used, as they only model propagation and target reflections at a single frequency. Instead, wideband models divide the system's bandwidth into multiple subbands. Each subband is then modeled as a narrowband signal and the received signals within each subband are recombined to determine the response across the entire system's bandwidth.
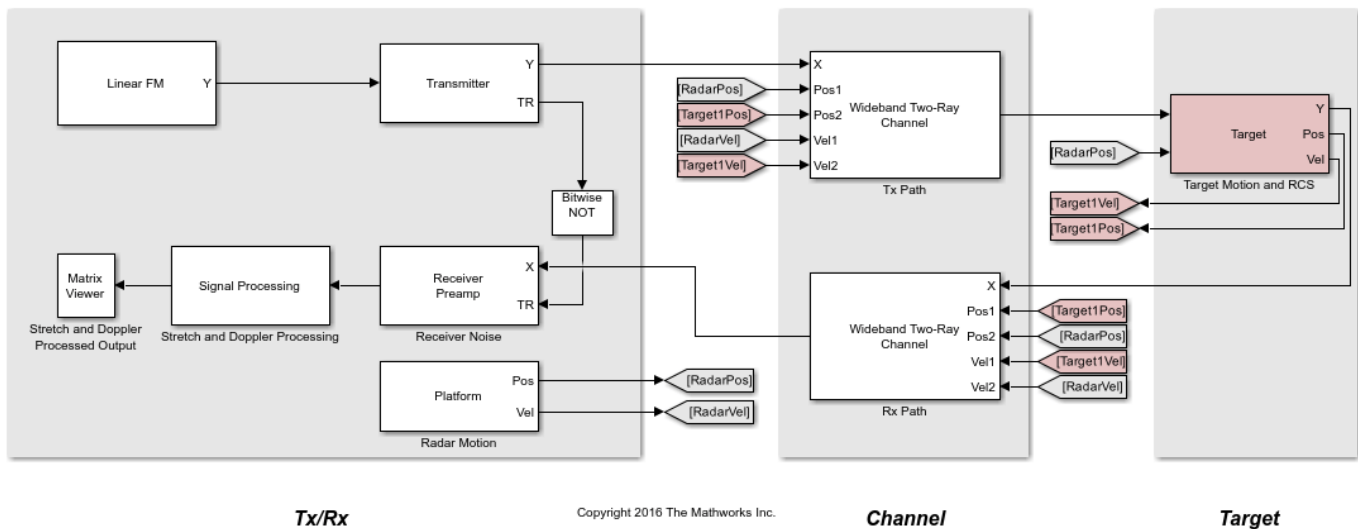
The model consists of a transceiver, a channel, and a target. The blocks that correspond to each section of the model are:

Transceiver

- `Linear FM` - Creates linear FM pulses.
- `Transmitter` - Amplifies the pulses and sends a Transmit/Receive status to the `Receiver Preamp` block to indicate if it is transmitting.
- `Receiver Preamp` - Receives the propagated pulses when the transmitter is off. This block also adds noise to the signal.

1-601

- `Platform` - Used to model the radar's motion.
- `Signal Processing` - Subsystem performs stretch processing, Doppler processing, and noise floor estimation.
- `Matrix Viewer` - Displays the processed pulses as a function of the measured range, radial speed, and estimated signal power to interference plus noise power ratio (SINR).

Signal Processing Subsystem
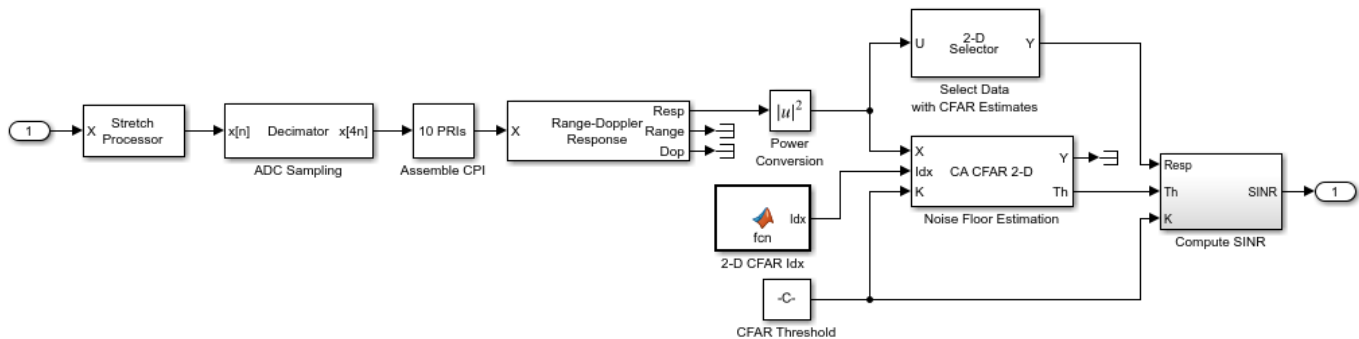
- `Stretch Processor` - Dechirps the received signal by mixing it in the analog domain with the transmitted linear FM waveform delayed to a selected reference range. A more detailed discussion on stretch processing is available in the "FMCW Range Estimation" example.
- `Decimator` - Subsystem models the sample rate of the analog-to-digital converter (ADC) by reducing the simulation's sample rate according to the bandwidth required by the range span selected in the stretch processor.
- `Buffer CPI` - Subsystem collects multiple pulse repetition intervals (PRIs) to form a coherent processing interval (CPI), enabling radial speed estimation through Doppler processing.
- `Range-Doppler Response` - Computes DFTs along the range and Doppler dimensions to estimate the range and radial speed of the received pulses.
- `CA CFAR 2-D` - Estimates the noise floor of the received signals using the cell-averaging (CA) method in both range and Doppler.
- `Compute SINR` - Subsystem normalizes the received signal using the CFAR detector's computed threshold, returning the estimated SINR in decibels (dB).

Channel

- `Wideband Two-Ray` - Applies propagation delays, losses, Doppler shifts and multipath reflections off of a flat ground to the pulses. One block is used for the transmitted pulses and another one for the reflected pulses. The `Wideband Two-Ray` blocks require the positions and velocities of the radar and the target. Those are supplied using the `Goto` and `From` blocks.

Target Subsystem

The `Target` subsystem models the target's motion and reflects the pulses according to the wideband RCS model and the target's aspect angle presented to the radar. In this example, the target is positioned 3000 meters from the wideband radar and is moving away from the radar at 100 m/s.
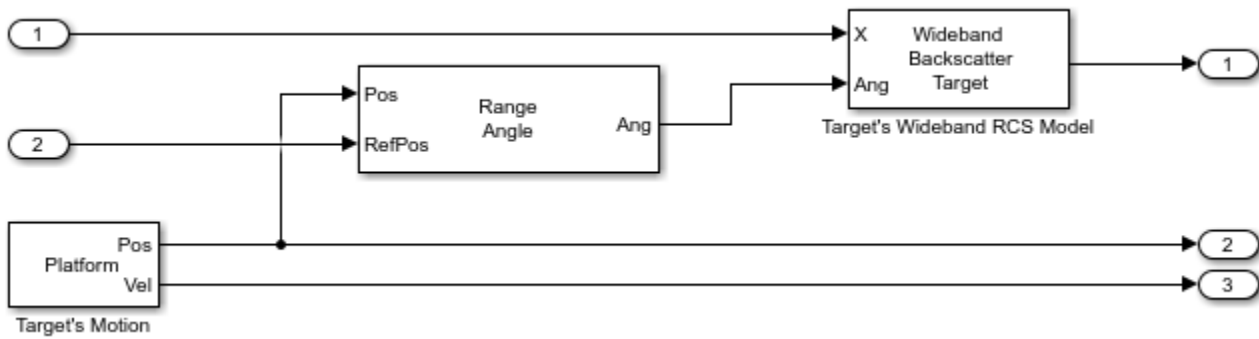
- **Platform** - Used to model the target's motion. The target's position and velocity values are used by the `Wideband Two-Ray Channel` blocks to model propagation and by the `Range Angle` block to compute the signal's incident angles at the target's location.
- **Range Angle** - Computes the propagated signal's incident angles in azimuth and elevation at the target's location.
- **Wideband Backscatter Target** - Models the wideband reflections of the target to the incident pulses. The extended wideband target model introduced in the Modeling Target Radar Cross Section example is used for this simulation.

Exploring the Model

Several dialog parameters of the model are calculated by the helper function helperslexWidebandMonostaticRadarParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs. To modify any parameters, either change the values in the structure at the command prompt or edit the helper function and rerun it to update the parameter structure.

Results and Displays

The figure below shows the range and radial speed of the target. Target range is computed from the round-trip delay of the reflected pulses. The target's radial speed is estimated by using the DFT to compare the phase progression of the received target returns across the coherent pulse interval (CPI). The target's range and radial speed are measured from the peak of the stretch and Doppler processed output.
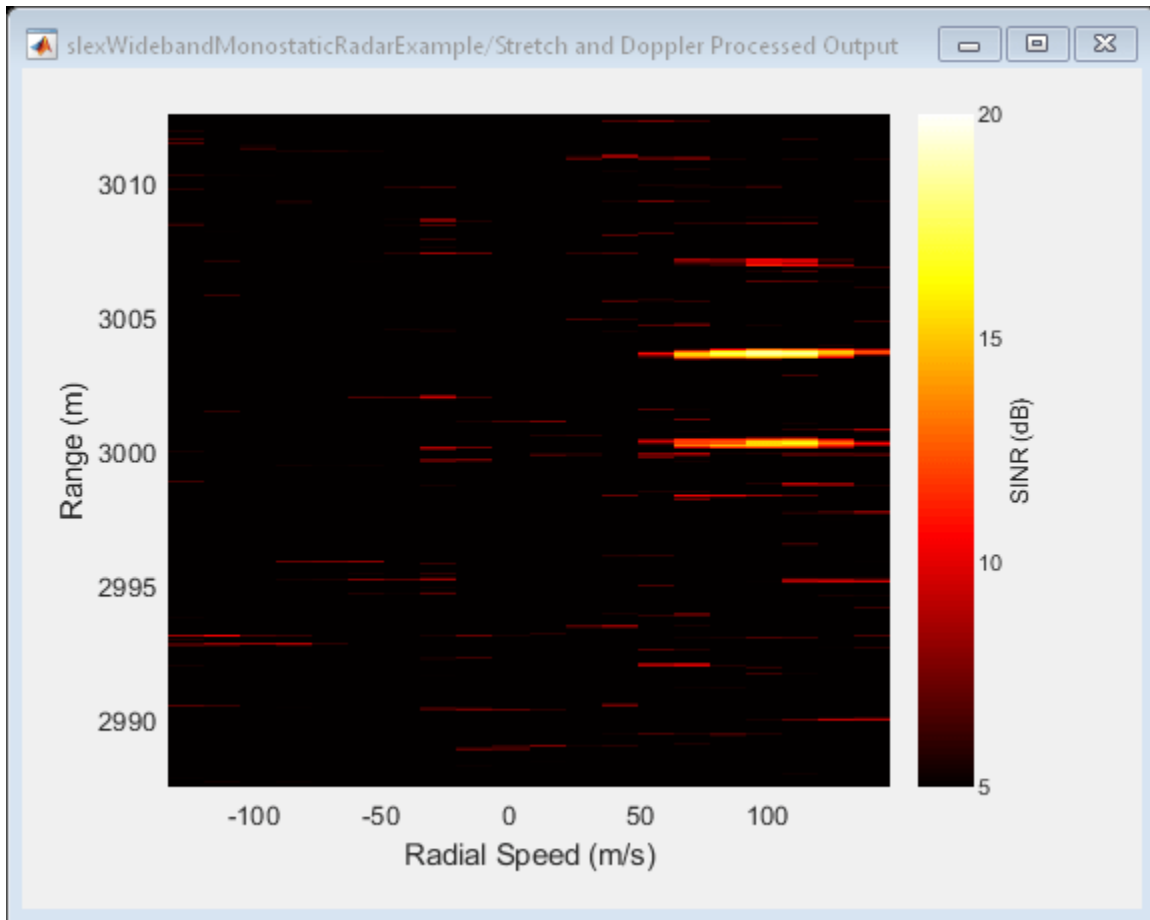
Although only a single target was modeled in this example, three target returns are observed in the upper right-hand portion of the figure. The multipath reflections along the transmit and receive paths give rise to the second and third target returns, often referred to as the single- and double-bounce returns respectively. The expected range and radial speed for the target is computed from the simulation parameters.

```
tgtRange = rangeangle(paramWidebandRadar.targetPos,...
  paramWidebandRadar.sensorPos)
```

```
tgtRange =

       3000
```

```
tgtSpeed = radialspeed(...
  paramWidebandRadar.targetPos,paramWidebandRadar.targetVel,...
  paramWidebandRadar.sensorPos,paramWidebandRadar.sensorVel)
```

```
tgtSpeed =

   -100
```

This expected range and radial speed are consistent with the simulated results in the figure above.

The expected separation between the multipath returns can also be found. The figure below illustrates the line-of-sight (R_{los}) and reflected path (R_{rp}) geometries.
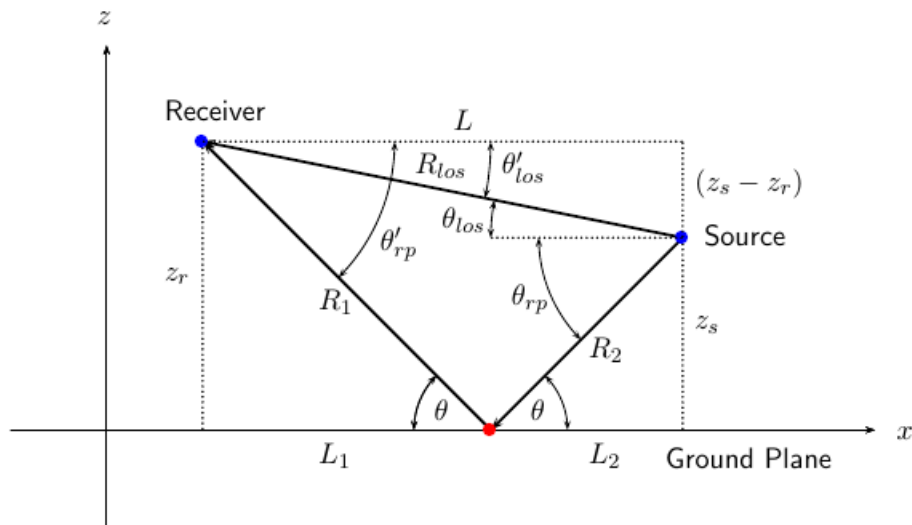
The modeled geometric parameters for this simulation are defined as follows.

```
zr = paramWidebandRadar.targetPos(3);
zs = paramWidebandRadar.sensorPos(3);
Rlos = tgtRange;
```

The length of each of these paths is easily derived.

$$L = \sqrt{R_{los}^2 - (z_r - z_s)^2}$$

$$R_1 = \frac{z_r}{z_r + z_s}\sqrt{(z_r + z_s)^2 + L^2}$$

$$R_2 = \frac{z_s}{z_r + z_s}\sqrt{(z_r + z_s)^2 + L^2}$$

$$R_{rp} = R_1 + R_2 = \sqrt{(z_r + z_s)^2 + L^2}$$

Using these results, the reflected path range can be computed.

```
L = sqrt(Rlos^2-(zr-zs)^2);
Rrp = sqrt((zs+zr)^2+L^2)
```

Rrp =

```
3.0067e+03
```

For a monostatic system, the single bounce return can traverse two different paths.

1 Radar $\xrightarrow{R_{rp}}$ Target $\xrightarrow{R_{los}}$ Radar

2 Radar $\xrightarrow{R_{los}}$ Target $\xrightarrow{R_{rp}}$ Radar

In both cases, the same range will be observed by the radar.

$$R_{obs} = \frac{R_{los} + R_{rp}}{2}$$

The range separation between all of the multipath returns is then found to be the difference between the observed and line-of-sight ranges.

$$R_\Delta = R_{obs} - R_{los} = \frac{R_{rp} - R_{los}}{2}$$

```
Rdelta = (Rrp-Rlos)/2
```

```
Rdelta =

    3.3296
```

Which matches the multipath range separation observed in the simulated results.

Summary

This example demonstrated how an end-to-end wideband radar system can be modeled within Simulink®. The variation of propagation losses and target RCS across the system's bandwidth required wideband propagation and target models to be used.

The signal to interference plus noise ratio (SINR) of the received target returns was estimated using the `CA CFAR 2-D` block. The CFAR estimator used cell-averaging to estimate the noise and interference power in the vicinity of the target returns which enabled calculation of the received signal's SINR.

The target was modeled in a multipath environment using the `Wideband Two-Ray Channel`, which gave rise to three target returns observed by the radar. These returns correspond to the line-of-sight, single-bounce, and double-bounce paths associated with the two-way signal propagation between the monostatic radar and the target. The simulated separation of the multipath returns in range was shown to match the expected separation computed from the modeled geometry.

Extended Target Tracking with Multipath Radar Reflections in Simulink

This example shows how to model and mitigate multipath radar reflections in a highway driving scenario in Simulink®. It closely follows the "Highway Vehicle Tracking with Multipath Radar Reflections" on page 1-64 MATLAB® example.

Introduction

While automotive radars provide robust detection performance across the diverse array of environmental conditions encountered in autonomous driving scenarios, interpreting the detections reported by the radar can prove challenging. Sensor fusion algorithms processing the radar detections will need to be able to identify the desired target detections returned along with detections arising from road (often referred to as clutter) and multipath between the various objects in the driving scenario like guardrails and other vehicles on the road. Detections generated by multiple reflections between the radar and a particular target are often referred to as *ghost detections* because they seem to originate in regions where no targets exist. This example shows you the impact of these multipath reflections on designing and configuring an object tracking strategy using radar detections. For more details regarding the multipath phenomenon and simulation of ghost detections, see the "Simulate Radar Ghosts Due to Multipath Return" on page 1-45 example.

Load Scenario and Radars

This example uses the same scenario and radars defined by the `helperCreateMultipathDrivingScenario` function used in the "Highway Vehicle Tracking with Multipath Radar Reflections" on page 1-64 example. Opening the model loads this scenario into the workspace for use by the Scenario Reader (Automated Driving Toolbox) block.

```
open_system('MultipathRadarDetectionsTrackingModel')
```
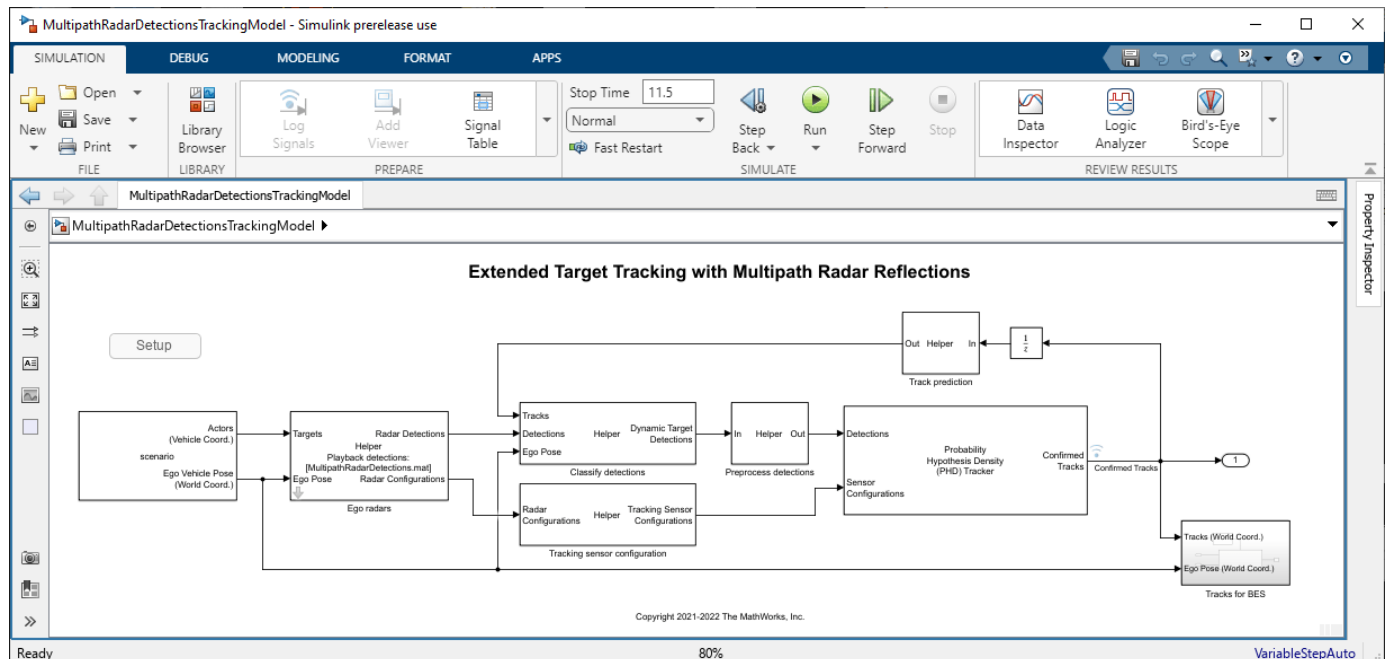
Use the **Ego radars** helper block to play back detections recorded from four radars providing full 360 degree coverage around the ego vehicle. To record a new set of detections, clear the **Playback radar recording** check box.

```
open_system('MultipathRadarDetectionsTrackingModel/Ego radars')
```
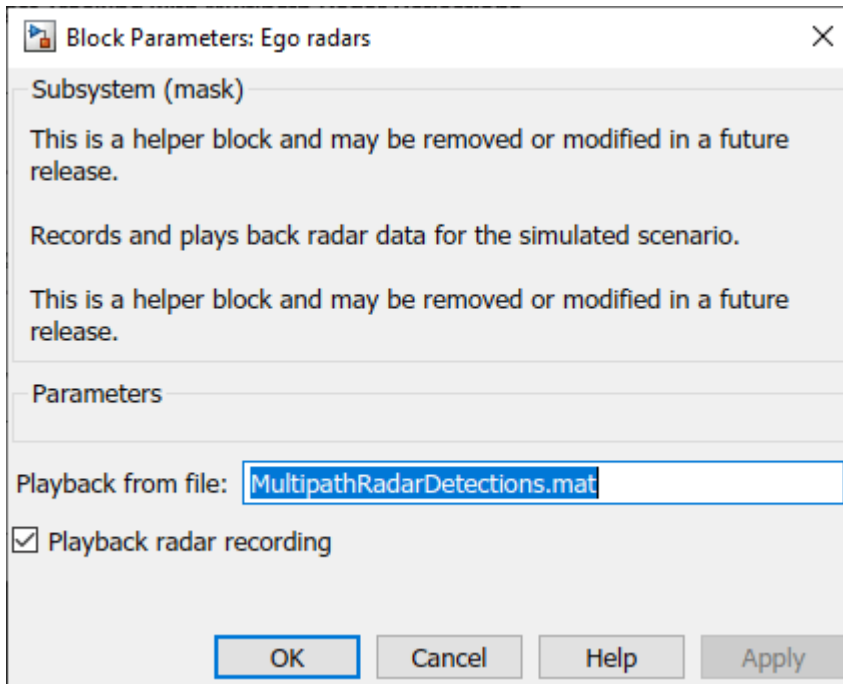


```
close_system('MultipathRadarDetectionsTrackingModel/Ego radars')
```

The four sensor models are configured in the **Record radars** block.

```
open_system('MultipathRadarDetectionsTrackingModel/Ego radars/Record radars')
```
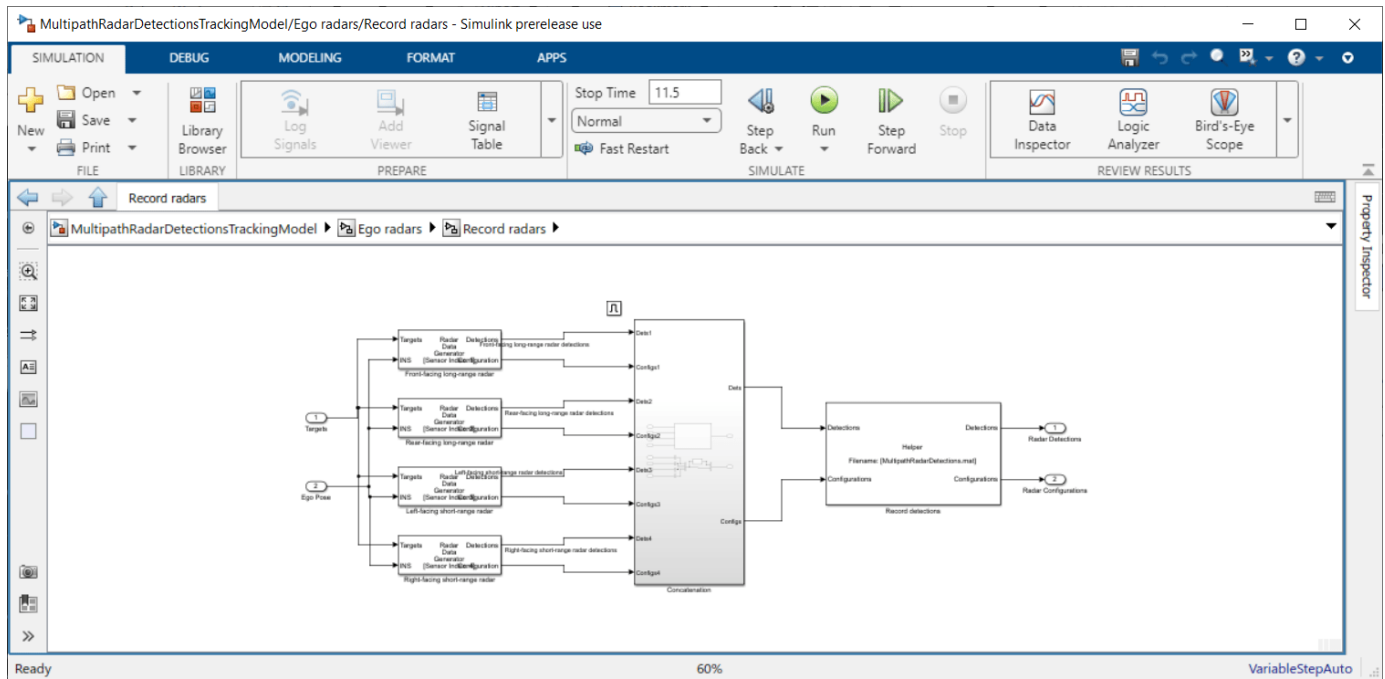
Use Bird's-Eye Scope (Automated Driving Toolbox) to visualize the scenario and sensor coverage in this model.
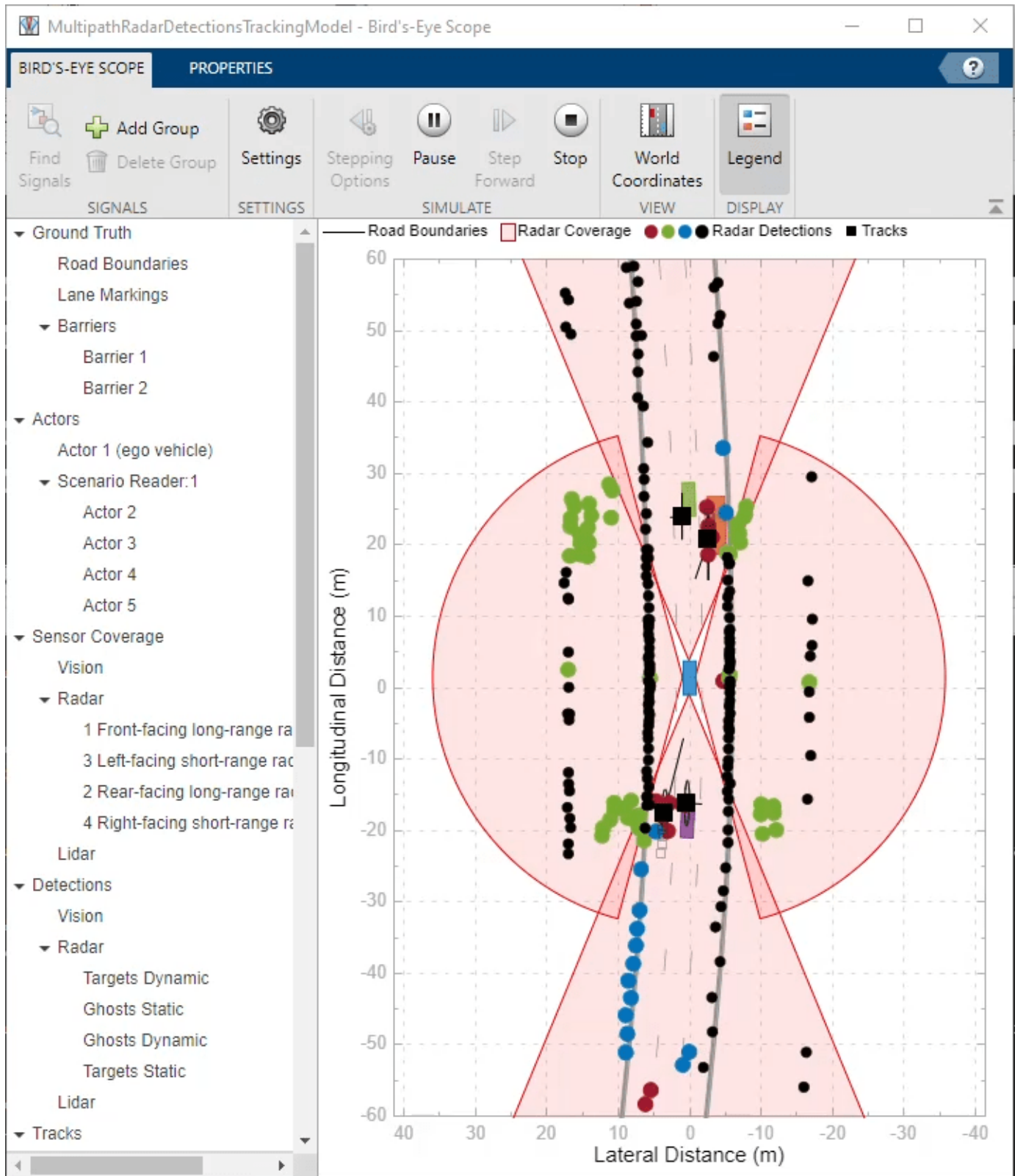
The **Classify detections** helper block classifies the detections generated by the four radars by comparing their measurements to the confirmed tracks from the Probability Hypothesis Density (PHD) Tracker (Sensor Fusion and Tracking Toolbox) block. The detection classification utilizes the measured radial velocity from the targets to determine if the target generating the detection was static or dynamic [1]. The detections are classified into four categories:

1 Dynamic targets — These (red) detections are classified to originate from real dynamic targets in the scene.

2 Static ghosts — These (green) detections are classified to originate from dynamic targets but reflected via the static environment.

3 Dynamic ghosts — These (blue) detections are classified to originate from dynamic targets but reflected via other dynamic objects.

4 Static targets — These (black) detections are classified to originate from the static environment.

Configure GGIW-PHD Extended Object Tracker

Configure the Probability Hypothesis Density (PHD) Tracker (Sensor Fusion and Tracking Toolbox) block with the same parameters as used by the "Highway Vehicle Tracking with Multipath Radar Reflections" on page 1-64 example. Reuse the `helperMultipathExamplePartitionFcn` function to define the detection partitions used within the tracker.

```
open_system('MultipathRadarDetectionsTrackingModel/Probability Hypothesis Density Tracker')
```

Block Parameters: Probability Hypothesis Density Tracker ✕

Probability Hypothesis Density (PHD) Tracker

The PHD tracker block creates and manages the tracks of stationary and moving objects in a multi-sensor environment. The tracker uses a multi-target probability hypothesis density filter to estimate the states of point targets and extended objects. The PHD is represented by a weighted summation of probability density functions, and peaks in the PHD are extracted to represent possible targets.

Source code

| Tracker Configuration | Track Management | Port Setting |

Tracker identifier: `1`

Detection partition function: `@(x)helperMultipathExamplePartitionFcn(x,2,5)`

Detection selection threshold: `450`

Maximum number of sensors: `20`

Maximum number of tracks: `1000`

Sensor configurations: `2}) , toStruct(sensorConfigs{3}) , toStruct(sensorConfigs{4}) ]`

☑ Update sensor configurations with time

Track state parameters: `struct`

☐ Update track state parameters with time

Simulate using: `Interpreted execution ▾`

OK   Cancel   Help   Apply

```
close_system('MultipathRadarDetectionsTrackingModel/Probability Hypothesis Density Tracker',0)
```

Run Simulation

Use the following command to play back the recorded detections and generate tracks.

```
simout = sim('MultipathRadarDetectionsTrackingModel')
```
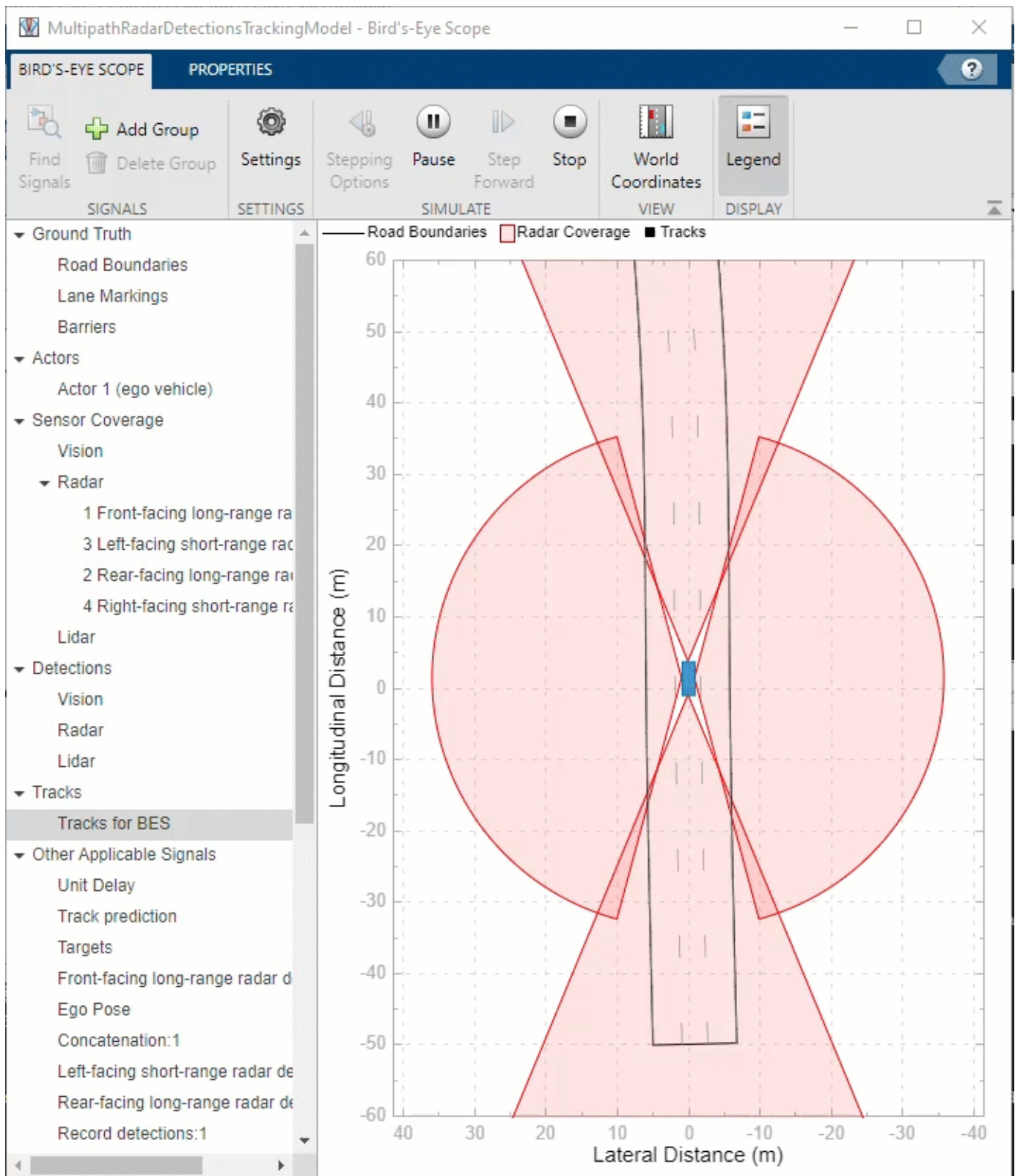
Use `helperSaveSimulationLogs` to save the logged tracks and classified detections for offline analysis.

```
helperSaveSimulationLogs('MultipathRadarDetectionsTrackingModel',simout);
```

Analyze Performance

Load the logged tracks and detections to assess the performance of the tracking algorithm by using the GOSPA metric and its associated components.

```
[confirmedTracks,confusionMatrix] = helperLoadSimulationLogs('MultipathRadarDetectionsTrackingMod
```

Use `trackGOSPAMetric` to calculate GOSPA metrics from the logged tracks.

```
gospaMetric = trackGOSPAMetric('Distance','custom', ...
    'DistanceFcn',@helperGOSPADistance, ...
    'CutoffDistance',35);

% Number of simulated track updates
numSteps = numel(confirmedTracks.Time);

% GOSPA metric
gospa = NaN(4,numSteps);

restart(scenario);
groundTruth = scenario.Actors(2:end);
iStep = 1;
tol = seconds(scenario.SampleTime/4);
while scenario.SimulationTime<=seconds(confirmedTracks.Time(end))
    % Select data from time table for current simulation time
    tsim = scenario.SimulationTime;
    wt = withtol(seconds(tsim),tol);

    % Select tracks from time table and compute GOSPA metrics
    theseTracks = confirmedTracks{wt,'Tracks'}{1};
    [gospa(1,iStep),~,~,gospa(2,iStep),gospa(3,iStep),gospa(4,iStep)] = gospaMetric(theseTracks,g

    if scenario.IsRunning
        advance(scenario);
    else
        break
    end
    iStep = iStep+1;
end
```

Quantitatively assess the performance of the tracking algorithm by using the GOSPA metric and its associated components. A lower value of the metric denotes better tracking performance. In the following figure, the Missed-target component of the metric remains zero after a few steps in the beginning, representing establishment delay of the tracker. This component shows that no targets were missed by the tracker. The False-tracks component of the metric is zero for most of the simulation, indicating that no false tracks were confirmed by the tracker during those times.

```
% Plot GOSPA metrics
plot(seconds(confirmedTracks.Time),gospa','LineWidth',2);
xlabel('Time (s)');
title('GOSPA Metrics');
grid on;
legend('GOSPA','Localization GOSPA','Missed-target GOSPA','False-tracks GOSPA');
```
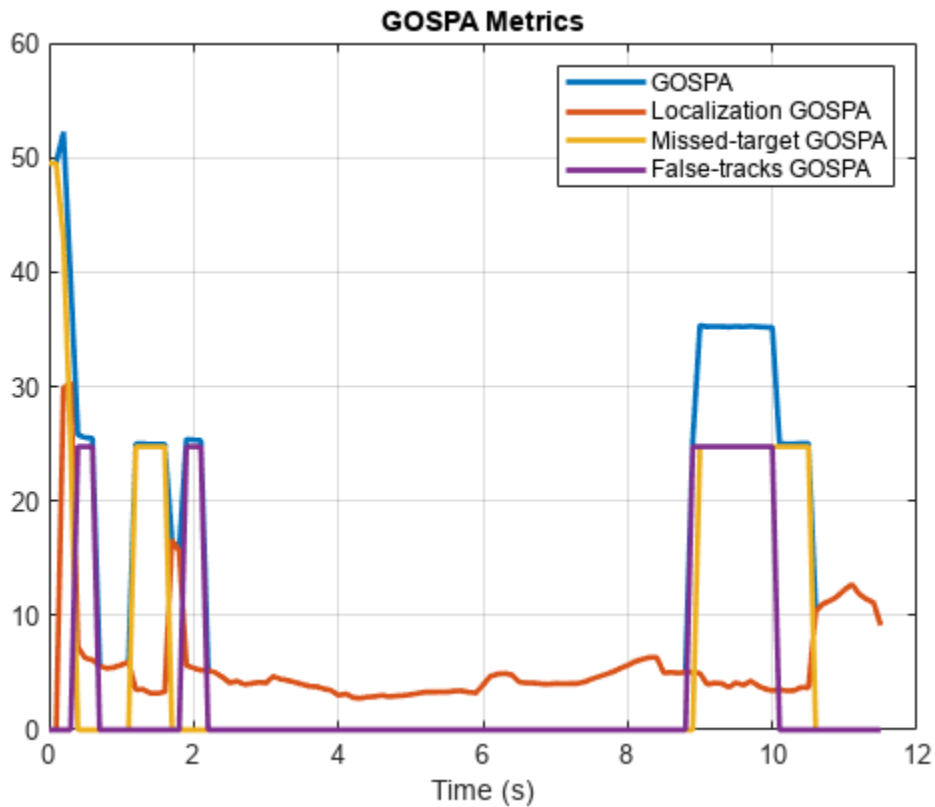
Similar to the tracking algorithm, you also quantitatively analyze the performance of the radar detection classification algorithm by using a confusion matrix [2]. The rows shown in the table denote the true classification information of the radar detections and the columns represent the predicted classification information. For example, the second element of the first row defines the percentage of target detections predicted as ghosts from static object reflections.

91% of the target detections are classified correctly. However, a small percentage of the target detections are misclassified as ghosts from dynamic reflections. Also, approximately 4% of ghosts from static object reflections and 22% of ghosts from dynamic object reflections are misclassified as targets and sent to the tracker for processing. A common situation when this occurs in this example is when the detections from two-bounce reflections lie inside the estimated extent of the vehicle. Further, the classification algorithm used in this example is not designed to find false alarms or clutter in the scene. Therefore, the fifth column of the confusion matrix is zero. Due to spatial distribution of the false alarms inside the field of view, the majority of false alarm detections are either classified as reflections from static objects or dynamic objects.

```
% Accumulate confusion matrix over all steps
confMat = shiftdim(reshape([confusionMatrix{:,'Confusion Matrix'}],numSteps,5,5),1);
confMat = sum(confMat,3);

% Number of detections for each target type
numDetections = sum(confMat,2);

numDetsTable = array2table(numDetections,'RowNames',{'Targets','Ghost (S)','Ghost (D)','Environme
    'VariableNames',{'Number of Detections'});
```

```
disp('True Information');disp(numDetsTable);
```

```
True Information
                      Number of Detections
                      _____

      Targets              1990
      Ghost (S)            3242
      Ghost (D)             848
      Environment         27451
      Clutter               139
```

```
% Calculate classification percentages
percentMatrix = confMat./numDetections*100;

percentMatrixTable = array2table(round(percentMatrix,2),'RowNames',{'Targets','Ghost (S)','Ghost
    "VariableNames",{'Targets','Ghost (S)','Ghost (D)', 'Environment','Clutter'});

disp('True vs Predicted Confusion Matrix (%)');disp(percentMatrixTable);
```

```
True vs Predicted Confusion Matrix (%)
                   Targets     Ghost (S)     Ghost (D)     Environment     Clutter

                   _____     _____     _____     _____     _____

      Targets       90.9         0.75          7.94            0.4            0
      Ghost (S)     3.52        84.86         11.32           0.31            0
      Ghost (D)    22.29         0.24         77.48              0            0
      Environment   1.53         2.93          3.42          92.12            0
      Clutter      19.42        66.19         13.67           0.72            0
```

Summary

In this example, you simulated radar detections due to multipath propagation in an urban highway driving scenario using Simulink. You configured a data processing algorithm to simultaneously filter ghost detections and track vehicles on the highway. You also analyzed the performance of the tracking algorithm and the classification algorithm using the GOSPA metric and confusion matrix.

References

[1] Prophet, Robert, et al. "Instantaneous Ghost Detection Identification in Automotive Scenarios." *2019 IEEE Radar Conference (RadarConf)*. IEEE, 2019.

[2] Kraus, Florian, et al. "Using machine learning to detect ghost images in automotive radar." *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2020.

Radar Vertical Coverage over Terrain

The propagation path of a radar system is modified by the environment in which it operates. When overlooking a surface, the free space radiation pattern is modified by the reflected wave, which results in an interference pattern with lobes and nulls. In a lobe maximum, a radar return can increase by as much as 12 dB, whereas in a minimum the return can theoretically go to 0 dB.

The vertical coverage pattern is a detection contour. It offers a look into the performance of a radar system at a constant signal level as specified by the free space range. The vertical coverage considers the interference between the direct and ground-reflected rays, as well as refraction. This example defines an L-band radar system in the presence of heavy clutter and shows you how to visualize its three-dimensional vertical coverage over terrain.

Define the Radar

To start, specify an L-band surveillance radar with the following parameters:

- Peak power: 3 kW
- Operating frequency: 1 GHz
- Transmit and receive antenna beamwidth: 20 degrees in azimuth and elevation
- Pulse width: 2 μs

```
% Radar parameters
rfs        = 50e3;             % Free-space range (m)
rdrppower  = 3e3;              % Peak power (W)
fc         = 1e9;             % Operating frequency (Hz)
hpbw       = 20;              % Half-power beamwidth (deg)
rdrpulsew  = 2e-6;            % Pulse width (s)
tiltAng    = 1;              % Radar tilt (elevation) angle (deg)
azRotation = 80;             % Radar azimuth rotation angle (deg)

% Vertical coverage parameters
minEl      = 0;              % Minimum vertical coverage angle (deg)
maxEl      = 90;             % Maximum vertical coverage angle (deg)
elStepSize = 1;             % Elevation step size (deg)
azAng      = -60:2:60;       % Azimuth angles for elevation cuts (deg)
```

Convert the transmit antenna's half-power beamwidth (HPBW) values to gain using the `beamwidth2gain` function. Assume a cosine rectangular aperture, which is a good approximation for a real-world antenna.

```
rdrgain    = beamwidth2gain(hpbw,'CosineRectangular'); % Transmit and receive antenna gain (dB)
```

Define Location

The next section defines the radar location. The radar is mounted on a tall tower 100 meters above the ground. The radar altitude is the sum of the ground elevation and the radar tower height referenced to the mean sea level (MSL). Visualize the location using `geoplot3`. The radar will appear as a red circular point in the upper right-hand corner of the image.

```
% Radar location
rdrlat = 39.5;             % Radar latitude (deg)
rdrlon = -105.5;           % Radar longitude (deg)
rdrtowerht = 100;          % Antenna height (m)
```

```
surfaceHtAtRadar = 2812; % Surface height at radar location (m)
rdralt = surfaceHtAtRadar + rdrtowerht; % Radar altitude (m)

% Import the relevant terrain data from the United States Geological
% Survey (USGS)
dtedfile = 'n39_w106_3arc_v2.dt1';
attribution = 'SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey.';
[Z,R] = readgeoraster(dtedfile,'OutputType','double');

% Visualize the location including terrain using the geographic globe plot
addCustomTerrain('southboulder',dtedfile,'Attribution',attribution);
hTerrain = uifigure;
g = geoglobe(hTerrain,'Terrain','southboulder','Basemap','streets');
hold(g,'on')
h_rdrtraj = geoplot3(g,rdrlat,rdrlon,rdralt,'o','Color',[0.6350 0.0780 0.1840],'LineWidth',6,'Ma
campos(g,40.0194565714502,-105.564712896622,13117.1497971643);
camheading(g,150.8665488888147);
campitch(g,-16.023558618352187);
```


Investigate Free Space Range

For this example, the free space range `rfs` is the range at which a target has the specified signal-to-noise ratio (SNR). Assume a large target with a 10 dBsm radar cross section (RCS). At this range, the free space target SNR can be calculated as follows.

```
trcs      = db2pow(10); % RCS (m^2)
lambda    = freq2wavelen(fc); % Wavelength (m)
```

```
tsnr       = radareqsnr(lambda,rfs,rdrppower,rdrpulsew, ...
    'RCS',trcs,'Gain',rdrgain)
```

```
tsnr = -0.7449
```

For our surveillance radar, the desired performance index is a probability of detection (Pd) of 0.8 and probability of false alarm (Pfa) below 1e-3. To make the radar system design more feasible, you can use a pulse integration technique to reduce the required SNR. For this system, assume noncoherent integration of 64 pulses. A good approximation of the minimum SNR needed for a detection at the specified Pd and Pfa can be computed by the `detectability` function as follows. Note that the free space SNR satisfies the detectability requirement.

```
Pd = 0.8;
Pfa = 1e-3;
minsnr = detectability(Pd,Pfa,64);
isdetectable = tsnr >= minsnr
```

```
isdetectable = logical
   1
```

The `toccgh` function allows you to translate receiver detection probabilities into track probabilities. Assuming the default tracker, the required Pd and Pfa translate to the following probability of target track (Pdt) and probability of false track (Pft).

```
[Pdt,Pft]  = toccgh(Pd,Pfa)
```

```
Pdt = 0.9608
```

```
Pft = 1.0405e-06
```

The Pd and Pfa requirements enable a track probability of about 96% and a false track probability on the order of 1e-6.

Plot Vertical Coverage

The vertical coverage pattern is visualized using a Blake chart, also known as a range-height-angle chart. The range along the *x*-axis is the propagated range and the height along the *y*-axis is relative to the origin of the ray.

The vertical coverage contour is calculated using the `radarvcd` function. The default permittivity model in `radarvcd` is based on a sea surface. Such a model is not applicable for the defined scenario, which is over land. Thus, the first step in simulating more realistic propagation is to select a more appropriate permittivity. Use the `earthSurfacePermittivity` function with the vegetation flag. Assume an ambient temperature of 21.1 degrees Celsius, which is about 70 degrees Fahrenheit. Assume a gravimetric water content of 0.3.

```
temp = 21.1; % Ambient temperature (degrees Celsius)
gwc = 0.3; % Gravimetric water content
[~,~,epsc] = earthSurfacePermittivity('vegetation',fc,temp,gwc);
```

Next, specify the antenna. Simulate the antenna as a theoretical sinc antenna pattern and plot.

```
hBeam = phased.SincAntennaElement('Beamwidth',hpbw);
pattern(hBeam,fc);
```

3D Directivity Pattern
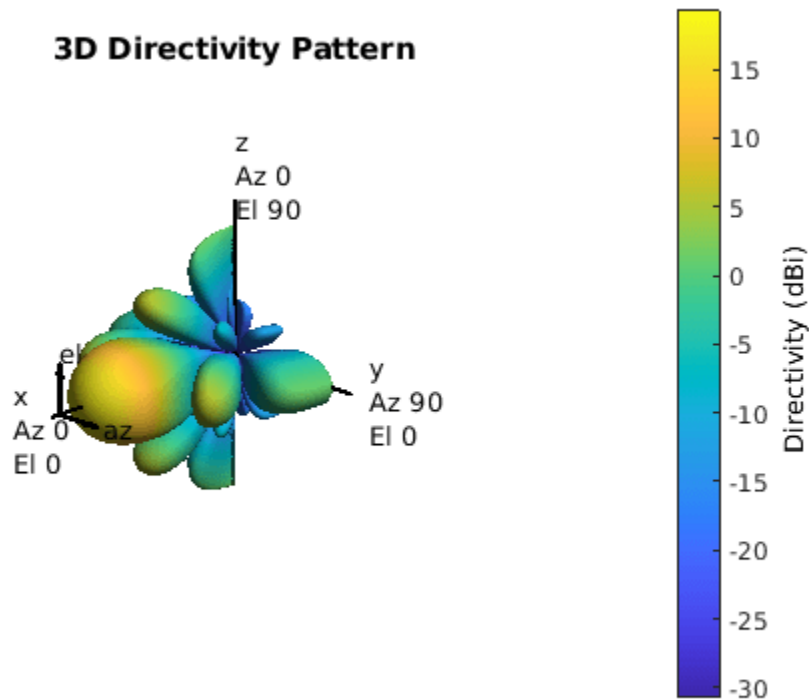
Get the elevation pattern at 0 degrees azimuth.

```
elAng = -90:0.01:90;
pat = getVoltagePattern(hBeam,fc,0,elAng);
```

Specify the atmosphere and calculate the corresponding effective Earth radius. Since the latitude of the radar in this example is 39.5 degrees, a mid-latitude atmosphere model would be most appropriate. Assume that the time period is during the summer.

```
% Set effective Earth radius using the mid latitude atmosphere model
% in summer
[nidx,N] = refractiveidx([0 1e3], ...
    'LatitudeModel','Mid','Season','Summer');
RGradient = (nidx(2) - nidx(1))/1e3;
Re = effearthradius(RGradient); % m
```

Next, calculate the vertical coverage pattern using the `radarvcd` function.

```
[vcpkm,vcpang] = radarvcd(fc,rfs.*1e-3,rdrtowerht, ...
    'SurfaceRelativePermittivity',epsc, ...
    'SurfaceHeightStandardDeviation',30, ...
    'Vegetation','Trees', ...
    'AntennaPattern',pat,'PatternAngles',elAng, ...
    'TiltAngle',tiltAng, ...
    'EffectiveEarthRadius',Re, ...
    'MinElevation',minEl, ...
    'ElevationStepSize',elStepSize, ...
    'MaxElevation',maxEl);
```

Use the surface height at the site to obtain the surface refractivity from `refractiveidx`.

```
% Calculate an appropriate surface
[~,Ns] = refractiveidx(surfaceHtAtRadar,'LatitudeModel','Mid','Season','Summer');
```

Plot the vertical coverage using the `blakechart` function. The `blakechart` axes are formed using the Central Radio Propagation Laboratory (CRPL) reference atmosphere. The CRPL model is widely used and models the refractivity profile as an exponential decay versus height, which has been shown to be an excellent approximation for a normal atmosphere (an atmosphere that is not undergoing anomalous propagation like ducting). The CRPL model is tailored by setting the surface height refractivity and the refraction exponent to a value that is appropriate for the location in which the system is operating and the time of year.

Update the surface refractivity and the refraction exponent inputs in the function call. In the subsequent plot, the constant-signal-level of the contour is given by the previously calculated target SNR.

```
rexp = refractionexp(Ns);
blakechart(vcpkm,vcpang,'ScalePower',1,'SurfaceRefractivity',Ns,'RefractionExponent',rexp)
```

Blake Chart
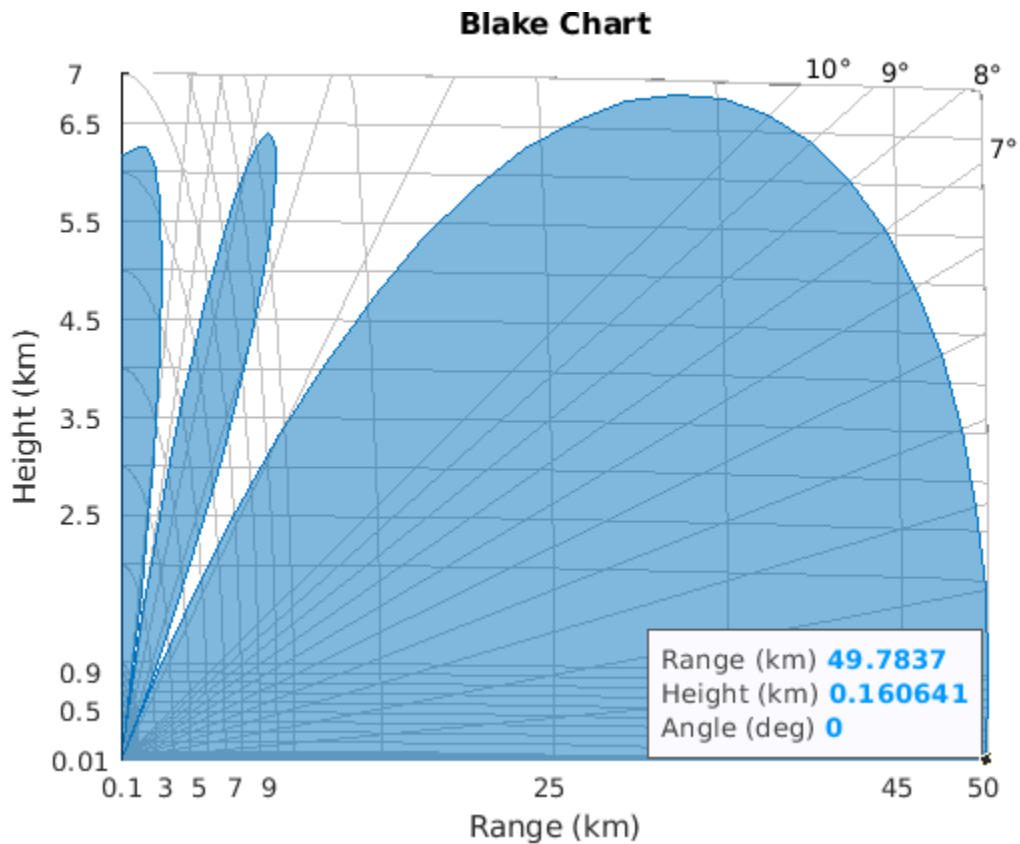
Visualize 3-D Vertical Coverage over Terrain

The next section calculates the vertical coverage at the specified azimuth intervals. The vertical coverage range and angle is converted to height using the `range2height` function using the CRPL method. The range-height-angle values are then converted to Cartesian.

```
% Initialize Cartesian X, Y, Z outputs
numAz = numel(azAng);
```

```
numRows = numel(minEl:elStepSize:maxEl)+1;
vcpX = zeros(numRows,numAz);
vcpY = zeros(numRows,numAz);
vcpZ = zeros(numRows,numAz);
wgs84 = wgs84Ellipsoid;

% Obtain vertical coverage contour in Cartesian coordinates
for idx = 1:numAz
    % Get elevation pattern at this azimuth
    pat = getVoltagePattern(hBeam,fc,azAng(idx),elAng);

    % Get terrain height information
    [xdir,ydir,zdir] = sph2cart(deg2rad(azAng(idx)+azRotation),0,1e3);
    [xlat,ylon,zlon] = enu2geodetic(xdir,ydir,zdir,rdrlat,rdrlon,rdralt,wgs84);
    [~,~,~,htsurf] = los2(Z,R,rdrlat,rdrlon, ...
        xlat,ylon,rdralt,zlon,'MSL','MSL');
    htstdSurf = std(htsurf(~isnan(htsurf)));

    % Calculate vertical coverage pattern
    [vcp,vcpang] = radarvcd(fc,rfs,rdrtowerht, ...
        'RangeUnit','m', ...
        'HeightUnit','m', ...
        'SurfaceRelativePermittivity',epsc, ...
        'SurfaceHeightStandardDeviation',htstdSurf, ...
        'Vegetation','Trees', ...
        'AntennaPattern',pat,'PatternAngles',elAng, ...
        'TiltAngle',tiltAng, ...
        'EffectiveEarthRadius',Re, ...
        'MinElevation',1, ...
        'ElevationStepSize',1, ...
        'MaxElevation',90);

    % Calculate associated heights
    vcpht = range2height(vcp,rdrtowerht,vcpang, ...
        'Method','CRPL','MaxNumIterations',2,'Tolerance',1e-2, ...
        'SurfaceRefractivity',Ns,'RefractionExponent',rexp);

    % Calculate true slant range and elevation angle to target
    [~,vcpgeomrt,vcpelt] = ...
        height2range(vcpht,rdrtowerht,vcpang, ...
        'Method','CRPL', ...
        'SurfaceRefractivity',Ns,'RefractionExponent',rexp);

    % Set values for this iteration
    vcpelt = [0;vcpelt(:);vcpang(end)];
    vcpgeomrt = [0;vcpgeomrt(:);0]; % km

    % Convert range, angle, height to Cartesian X, Y, Z
    [vcpX(:,idx),vcpY(:,idx),vcpZ(:,idx)] = sph2cart(...
        deg2rad(azAng(idx).*ones(numRows,1)+azRotation), ...
        deg2rad(vcpelt),vcpgeomrt);
end
```

Convert the Cartesian vertical coverage values to geodetic and plot on the globe. The three-dimensional vertical coverage appears as a blue mesh.

```
[vcpLat,vcpLon,vcpAlt] = enu2geodetic(vcpX,vcpY,vcpZ,rdrlat,rdrlon,rdralt,wgs84);
[xMesh,yMesh,zMesh] = helperCreateMesh(vcpLat,vcpLon,vcpAlt);
geoplot3(g,xMesh,yMesh,zMesh,'LineWidth',1,'Color',[0 0.4470 0.7410]);
```
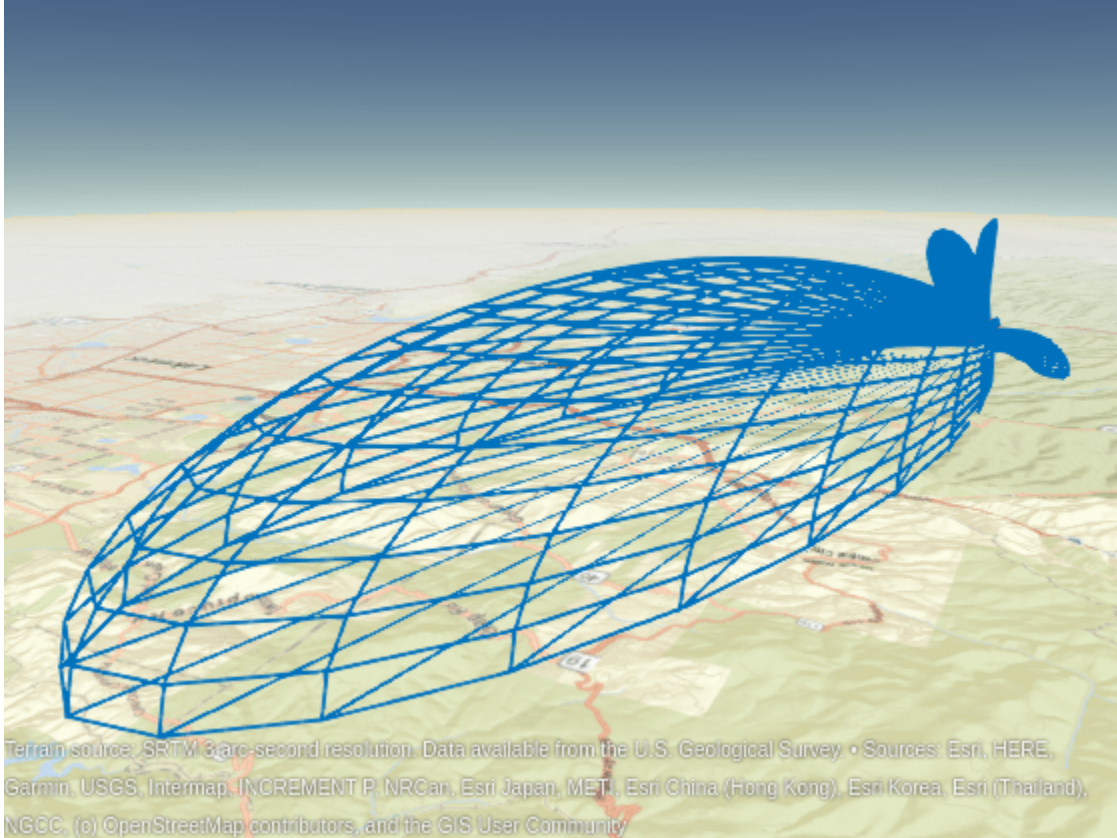

Summary

This example discussed a method to calculate a three-dimensional vertical coverage, discussing ways to tailor the analysis to the local atmospheric conditions. The visualization created gives insight into the vertical performance of the radar system, considering the interference from the surface reflections and refraction.

Helper Functions

```
% Clean up by closing the geographic globe and removing the imported
% terrain data.
if isvalid(hTerrain)
    close(hTerrain)
end
removeCustomTerrain("southboulder")

function pat = getVoltagePattern(hBeam,fc,azAng,elAng)
% Obtain voltage pattern from antenna element
numAz = numel(azAng);
numEl = numel(elAng);
pat = zeros(numAz,numEl);
for ia = 1:numAz
    for ie = 1:numEl
        pat(ia,ie) = step(hBeam,fc,[azAng(ia);elAng(ie)]);
```

```
        end
    end
end

function [xMesh,yMesh,zMesh] = helperCreateMesh(x,y,z)
% Organizes the inputs into a mesh, which can be plotted using geoplot3
numPts = numel(x);
xMesh = zeros(2*numPts,1);
yMesh = zeros(2*numPts,1);
zMesh = zeros(2*numPts,1);
[nrows,ncols] = size(x);
idxStart = 1;
idxEnd = nrows;
for ii = 1:ncols
    if mod(ii,2) == 0
        xMesh(idxStart:idxEnd) = x(:,ii);
        yMesh(idxStart:idxEnd) = y(:,ii);
        zMesh(idxStart:idxEnd) = z(:,ii);
    else
        xMesh(idxStart:idxEnd) = flipud(x(:,ii));
        yMesh(idxStart:idxEnd) = flipud(y(:,ii));
        zMesh(idxStart:idxEnd) = flipud(z(:,ii));
    end

    idxStart = idxEnd + 1;
    if ii ~= ncols
        idxEnd = idxStart + nrows -1;
    else
        idxEnd = idxStart + ncols -1;
    end
end

for ii = 1:nrows
    if mod(ii,2) == 0
        xMesh(idxStart:idxEnd) = x(ii,:);
        yMesh(idxStart:idxEnd) = y(ii,:);
        zMesh(idxStart:idxEnd) = z(ii,:);
    else
        xMesh(idxStart:idxEnd) = fliplr(x(ii,:));
        yMesh(idxStart:idxEnd) = fliplr(y(ii,:));
        zMesh(idxStart:idxEnd) = fliplr(z(ii,:));
    end
    idxStart = idxEnd + 1;
    idxEnd = idxStart + ncols -1;
end
end
```

References

1 Barton, D. K. *Radar Equations for Modern Radar*. Artech House Radar Series. Boston, Mass: Artech House, 2013.

2 Bean, B. R., and G. D. Thayer. "Central Radio Propagation Laboratory Exponential Reference Atmosphere." Journal of Research of the National Bureau of Standards, Section D: Radio Propagation 63D, no. 3 (November 1959): 315. https://doi.org/10.6028/jres.063D.031.

3 Blake, L. V. "Radio Ray (Radar) Range-Height-Angle Charts." Naval Research Laboratory, January 22, 1968.

4   Blake, L. V. "Ray Height Computation for a Continuous Nonlinear Atmospheric Refractive-Index Profile." Radio Science 3, no. 1 (January 1968): 85-92. https://doi.org/10.1002/rds19683185.

See Also

blakechart | detectability | effearthradius | height2range | radareqsnr | radarvcd | range2height | refractiveidx | toccgh

Related Examples

- "Modeling Target Position Errors Due to Refraction" on page 1-671

SAR Target Classification using Deep Learning

This example shows how to create and train a simple convolution neural network to classify SAR targets using deep learning.

Deep learning is a powerful technique that can be used to train robust classifier. It has shown its effectiveness in diverse areas ranging from image analysis to natural language processing. These developments have huge potential for SAR data analysis and SAR technology in general, slowly being realized. A major task for SAR-related algorithms has long been object detection and classification, which is called automatic target recognition (ATR). Here we used a simple convolution neural network to train and classify SAR targets using Deep Learning Toolbox™.

The Deep Learning Toolbox provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and apps.

This example demonstrates how to:

- Download Data set.
- Load and analyze image data.
- Splitting and augmentation of the data.
- Define the network architecture.
- Train the network.
- Predict the labels of new data and calculate the classification accuracy.

To illustrate this workflow, we will use the Moving and Stationary Target Acquisition and Recognition (MSTAR) Mixed Targets dataset published by the Air Force Research Laboratory [1 on page 1-633]. Our goal is to develop a model to classify ground targets based on SAR imagery.

Download Data set

This example uses MSTAR target dataset contains 8688 SAR images from 7 ground vehicle and a calibration target. The data was collected using an X-band sensor in spotlight mode, with a 1-foot resolution. The type of target we used are BMP2 (Infantry Fighting Vehicle), BTR70 (armored car), and T72 (tank). The images were captured at two different depression angles 15 degree and 17 degree with 190 ~ 300 different aspect versions, which are full aspect coverage over 360 degree. Optical images and SAR images of these three types of targets and their replicate targets are shown below in figure.

Download the dataset from the given URL using the `helperDownloadMSTARTargetData` helper function, defined at the end of this example. The size of data set is 28MiB.

```
outputFolder = pwd;
dataURL = ['https://ssd.mathworks.com/supportfiles/radar/data/' ...
    'MSTAR_TargetData.tar.gz'];
helperDownloadMSTARTargetData(outputFolder,dataURL);
```

Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser and extract the file. If you do so, change the outputFolder variable in the code to the location of the downloaded file.

Load and analyze Image Data

Load the SAR image data as an image datastore. `imageDatastore` automatically labels the images based on folder names and stores the data as an `imageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network.

```
sarDatasetPath = fullfile(pwd,'Data');
imds = imageDatastore(sarDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');
```

The MSTAR dataset contains SAR returns from 7 ground vehicles and a calibration target. Optical images and SAR images for these 8 targets are shown below
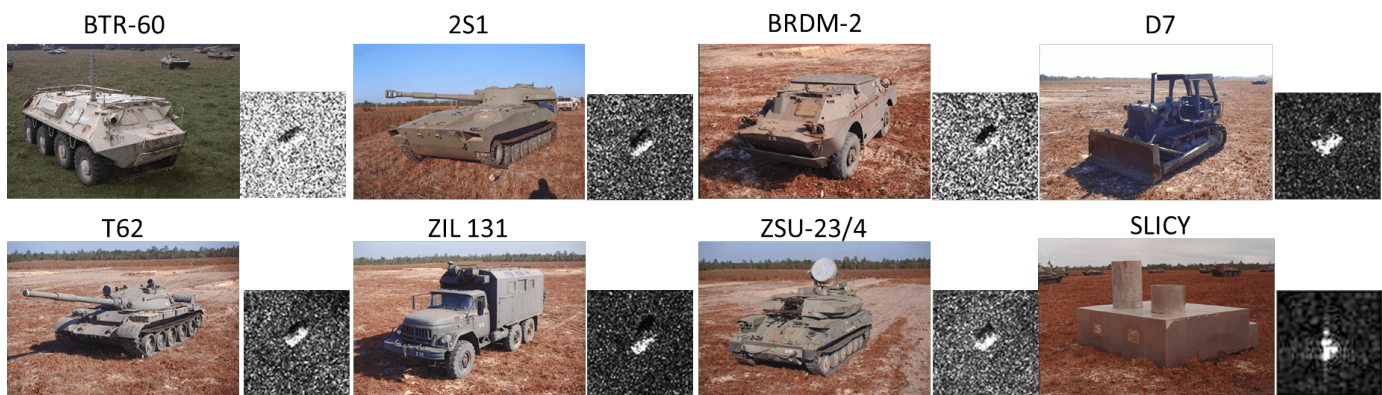
Let's explore the datastore by randomly displaying some chip images.

```
rng(0)
figure
% Shuffle the datastore.
imds = shuffle(imds);
for i = 1:20
    subplot(4,5,i)
    img = read(imds);
    imshow(img)
    title(imds.Labels(i))
    sgtitle('Sample training images')
end
```

Sample training images
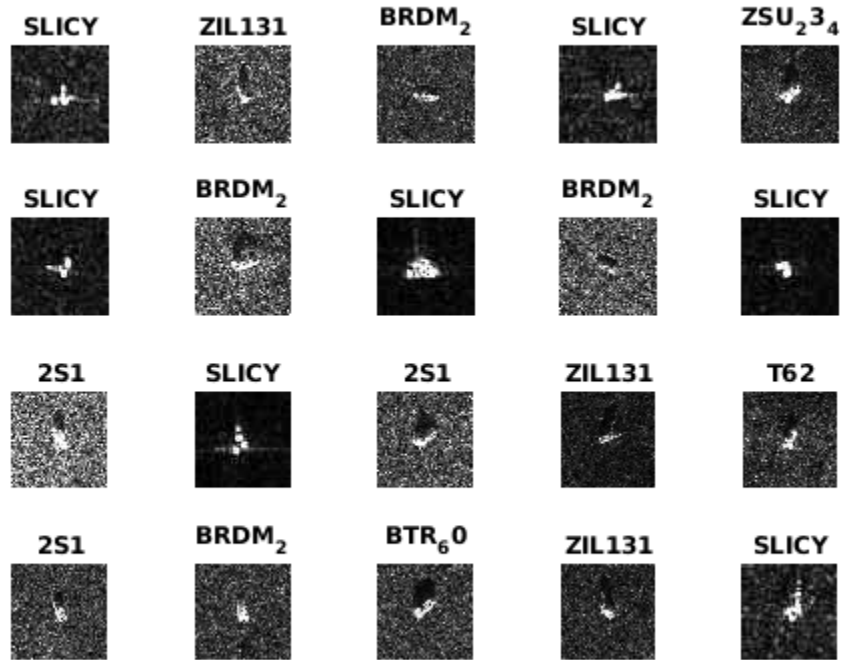
The `imds` variable now contains the images and the category labels associated with each image. The labels are automatically assigned from the folder names of the image files. Use `countEachLabel` to summarize the number of images per category.

```
labelCount = countEachLabel(imds)
```

```
labelCount=8×2 table
    Label      Count
    _____   _____

    2S1        1164
    BRDM_2     1415
    BTR_60      451
    D7          573
    SLICY      2539
    T62         572
    ZIL131      573
    ZSU_23_4   1401
```

First, specify the network input size. When choosing the network input size, consider the memory constraint of your system and the computation cost incurred in training.

```
imgSize = [128,128,1];
```

Create Datastore Object for Training, Validation and Testing

Divide the data into training, validation and test sets. We will use 80% of our dataset for training, 10% for model validation during training, and 10% for testing after training. `splitEachLabel` splits the datastore `imds` into three new datastores, `imdsTrain`, `imdsValidation`, and `imdsTest`. In doing so, it considers the varying number of images of different classes, so that the training, validation, and test sets have the same proportion of each class.

```
trainingPct = 0.8;
validationPct = 0.1;
[imdsTrain,imdsValidation,imdsTest] = splitEachLabel(imds,...
    trainingPct,validationPct,'randomize');
```

Data Augmentation

The images in the datastore do not have a consistent size. To train the images with our network, the image size must match the size of the network's input layer. Instead of resizing the images ourselves, we can use an `augmentedImageDatastore`, which will automatically resize the images before passing them into the network. The `augmentedImageDatastore` can also be used to apply transformations, such as rotation, reflection, or scaling, to the input images. This is useful to keep the network from overfitting to our data.

```
auimdsTrain = augmentedImageDatastore(imgSize, imdsTrain);
auimdsValidation = augmentedImageDatastore(imgSize, imdsValidation);
auimdsTest = augmentedImageDatastore(imgSize, imdsTest);
```

Define Network Architecture

Define the convolutional neural network architecture.

```
layers = createNetwork(imgSize);
```

Train Network

After defining the network structure, use `trainingOptions` (Deep Learning Toolbox) to specify the training options. Train the network using stochastic gradient descent with momentum (SGDM) with an initial learning rate of 0.001. We set the maximum number of epochs to 3. An epoch is a full training cycle on the entire training data set. Monitor the network accuracy during training by specifying validation data and validation frequency. Shuffle the data every epoch. The software trains the network on the training data and calculates the accuracy on the validation data at regular intervals during training. The validation data is not used to update the network weights. We set `'CheckpointPath'` to a temporary location. This enables the saving of partially trained detectors during the training process. If training is interrupted, such as by a power outage or system failure, you can resume training from the saved checkpoint.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.001, ...
    'MaxEpochs',3, ...
    'Shuffle','every-epoch', ...
    'MiniBatchSize',48,...
    'ValidationData',auimdsValidation, ...
    'ValidationFrequency',15, ...
    'Verbose',false, ...
    'CheckpointPath',tempdir,...
    'Plots','training-progress');
```

Train the network using the architecture defined by `layers`, the training data, and the training options. By default, `trainNetwork` uses a GPU if one is available (requires Parallel Computing

Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). For information about the supported compute capabilities, see GPU Support by Release (Parallel Computing Toolbox). Otherwise, it uses a CPU. You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value pair argument of `trainingOptions`.

The training progress plot shows the mini-batch loss and accuracy and the validation loss and accuracy. For more information on the training progress plot, see "Monitor Deep Learning Training Progress" (Deep Learning Toolbox). The loss is the cross-entropy loss. The accuracy is the percentage of images that the network classifies correctly.

```
net = trainNetwork(auimdsTrain,layers,options);
```
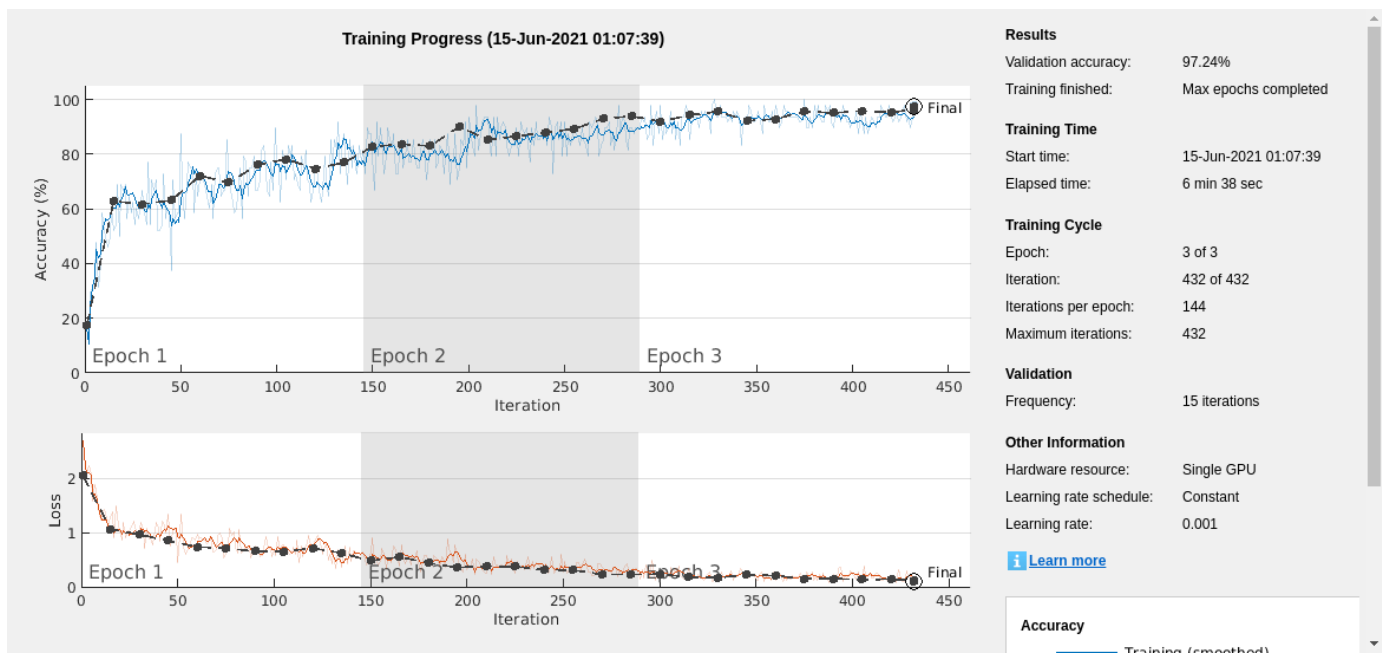

The training process is displayed in the image above. The dark blue line on the upper plot indicates the model's accuracy on the training data, while the black dashed line indicates the model's accuracy on the validation data (separate from training). The validation accuracy is more than 97%, which is quite good for an eight-class classifier. Furthermore, note that the validation accuracy and training accuracy are similar. This indicates that we have a robust classifier. When the training accuracy is much higher than the validation accuracy, the model is overfitting to (i.e. memorizing) the training data.

Classify Test Images and Compute Accuracy

Predict the labels of the validation data using the trained network and calculate the final accuracy. Accuracy is the fraction of labels that the network predicts correctly.

```
YPred = classify(net,auimdsTest);
YTest = imdsTest.Labels;

accuracy = sum(YPred == YTest)/numel(YTest)

accuracy = 0.9827
```

The test accuracy is very close to the validation accuracy, giving us confidence in the model's predictive ability.

We can use a confusion matrix to study the model's classification behavior in greater detail. A strong center diagonal represents accurate predictions. Ideally, we would like to see small, randomly located values off the diagonal. Large values off the diagonal could indicate specific scenarios where the model struggles.

```
figure
cm = confusionchart(YPred, YTest);
cm.RowSummary = 'row-normalized';
cm.Title = 'SAR Target Classification Confusion Matrix';
```

SAR Target Classification Confusion Matrix

True Class \ Predicted Class	2S1	BRDM_2	BTR_60	D7	SLICY	T62	ZIL131	ZSU_23_4
2S1	109							
BRDM_2		139						
BTR_60		1	45					
D7				54			1	
SLICY					254			
T62	1					55		
ZIL131	1						56	
ZSU_23_4	5	1		3		2		140
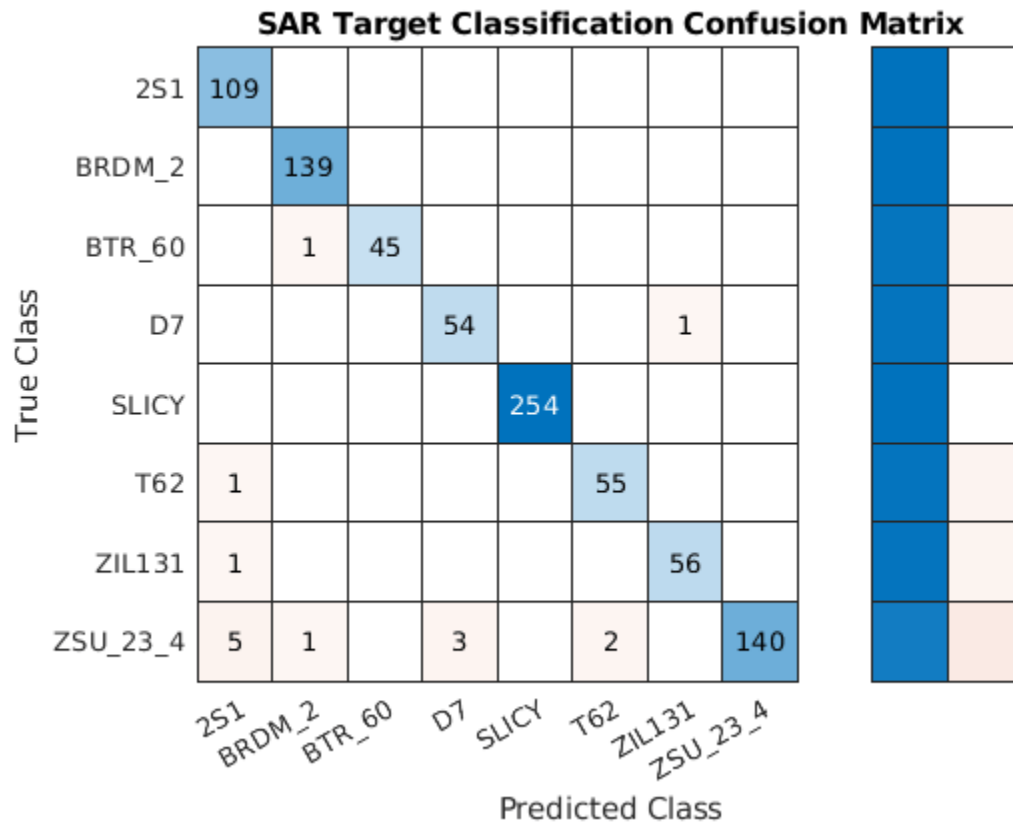
Out of the eight classes, the model appears to struggle the most with correctly classifying the ZSU-23/4. The ZSU-23/4 and 2S1 have very similar SAR images and hence we observe some misclassification by our trained model. However, it is still able to achieve greater than 90% accuracy for the class.

Helper Function

The function `createNetwork` takes as input the image input size `imgSize` and returns a convolution neural network. See below for a description of what each layer type does.

Image Input Layer — An `imageInputLayer` (Deep Learning Toolbox) is where you specify the image size. These numbers correspond to the height, width, and the channel size. The SAR image data consists of grayscale images, so the channel size (color channel) is 1. For a color image, the channel size is 3, corresponding to the RGB values. You do not need to shuffle the data because `trainNetwork`, by default, shuffles the data at the beginning of training. `trainNetwork` can also automatically shuffle the data at the beginning of every epoch during training.

Convolutional Layer — In the convolutional layer, the first argument is `filterSize`, which is the height and width of the filters the training function uses while scanning along the images. In this example, the number 3 indicates that the filter size is 3-by-3. You can specify different sizes for the height and width of the filter. The second argument is the number of filters, `numFilters`, which is the number of neurons that connect to the same region of the input. This parameter determines the number of feature maps. Use the `'Padding'` name-value pair to add padding to the input feature map. For a convolutional layer with a default stride of 1, `'same'` padding ensures that the spatial output size is the same as the input size. You can also define the stride and learning rates for this layer using name-value pair arguments of `convolution2dLayer` (Deep Learning Toolbox).

Batch Normalization Layer — Batch normalization layers normalize the activations and gradients propagating through a network, making network training an easier optimization problem. Use batch normalization layers between convolutional layers and nonlinearities, such as ReLU layers, to speed up network training and reduce the sensitivity to network initialization. Use `batchNormalizationLayer` (Deep Learning Toolbox) to create a batch normalization layer.

ReLU Layer — The batch normalization layer is followed by a nonlinear activation function. The most common activation function is the rectified linear unit (ReLU). Use `reluLayer` (Deep Learning Toolbox) to create a ReLU layer.

Max Pooling Layer — Convolutional layers (with activation functions) are sometimes followed by a down-sampling operation that reduces the spatial size of the feature map and removes redundant spatial information. Down-sampling makes it possible to increase the number of filters in deeper convolutional layers without increasing the required amount of computation per layer. One way of down-sampling is using a max pooling, which you create using `maxPooling2dLayer` (Deep Learning Toolbox). The max pooling layer returns the maximum values of rectangular regions of inputs, specified by the first argument, `poolSize`. In this example, the size of the rectangular region is [2,2]. The `'Stride'` name-value pair argument specifies the step size that the training function takes as it scans along the input.

Fully Connected Layer — The convolutional and down-sampling layers are followed by one or more fully connected layers. As its name suggests, a fully connected layer is a layer in which the neurons connect to all the neurons in the preceding layer. This layer combines all the features learned by the previous layers across the image to identify the larger patterns. The last fully connected layer combines the features to classify the images. Therefore, the `OutputSize` parameter in the last fully connected layer is equal to the number of classes in the target data. In this example, the output size is 10, corresponding to the 10 classes. Use `fullyConnectedLayer` (Deep Learning Toolbox) to create a fully connected layer.

Softmax Layer — The softmax activation function normalizes the output of the fully connected layer. The output of the softmax layer consists of positive numbers that sum to one, which can then be used as classification probabilities by the classification layer. Create a softmax layer using the `softmaxLayer` (Deep Learning Toolbox) function after the last fully connected layer.

Classification Layer — The final layer is the classification layer. This layer uses the probabilities returned by the softmax activation function for each input to assign the input to one of the mutually exclusive classes and compute the loss. To create a classification layer, use `classificationLayer` (Deep Learning Toolbox).

```matlab
function layers = createNetwork(imgSize)
    layers = [
        imageInputLayer([imgSize(1) imgSize(2) 1])      % Input Layer
        convolution2dLayer(3,32,'Padding','same')       % Convolution Layer
        reluLayer                                        % Relu Layer
```

```matlab
        convolution2dLayer(3,32,'Padding','same')
        batchNormalizationLayer                         % Batch normalization Layer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)                 % Max Pooling Layer

        convolution2dLayer(3,64,'Padding','same')
        reluLayer
        convolution2dLayer(3,64,'Padding','same')
        batchNormalizationLayer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)

        convolution2dLayer(3,128,'Padding','same')
        reluLayer
        convolution2dLayer(3,128,'Padding','same')
        batchNormalizationLayer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)

        convolution2dLayer(3,256,'Padding','same')
        reluLayer
        convolution2dLayer(3,256,'Padding','same')
        batchNormalizationLayer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)

        convolution2dLayer(6,512)
        reluLayer

        dropoutLayer(0.5)                               % Dropout Layer
        fullyConnectedLayer(512)                        % Fully connected Layer.
        reluLayer
        fullyConnectedLayer(8)
        softmaxLayer                                    % Softmax Layer
        classificationLayer                             % Classification Layer
        ];
end

function helperDownloadMSTARTargetData(outputFolder,DataURL)
% Download the data set from the given URL to the output folder.

    radarDataTarFile = fullfile(outputFolder,'MSTAR_TargetData.tar.gz');

    if ~exist(radarDataTarFile,'file')

        disp('Downloading MSTAR Target data (28 MiB)...');
        websave(radarDataTarFile,DataURL);
        untar(radarDataTarFile,outputFolder);
    end
end
```

References

[1] MSTAR Dataset. https://www.sdms.afrl.af.mil/index.php?collection=mstar

Automatic Target Recognition (ATR) in SAR Images

This example shows how to train a region-based convolutional neural network (R-CNN) for target recognition in large-scene synthetic aperture radar (SAR) images using Deep Learning Toolbox™ and Parallel Computing Toolbox™.

Deep Learning Toolbox provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and apps.

Parallel Computing Toolbox lets you solve computationally and data-intensive problems using multicore processors, GPUs, and computer clusters. It enables you to use GPUs directly from MATLAB® and accelerate the computation capabilities needed in deep learning algorithms.

Neural network based algorithms have shown remarkable achievement in diverse areas ranging from natural scene detection to medical imaging. They have shown huge improvement over the standard detection algorithms. Inspired by these advancements, researchers have put efforts to apply deep learning based solutions to the field of SAR imaging. In this example, the solution has been applied to solve the problem of target detection and recognition. The R-CNN network employed here not only solves problem of integrating detection and recognition but also provides an effective and efficient performance solution that scales to large scene SAR images as well.

This example demonstrates how to:

- Download the dataset and the pretrained model
- Load and analyze the image data
- Define the network architecture
- Specify training options
- Train the network
- Evaluate the network

To illustrate this workflow, the example uses the Moving and Stationary Target Acquisition and Recognition (MSTAR) clutter dataset published by the Air Force Research Laboratory. The dataset is available for download here. Alternatively, the example also includes a subset of the data used to showcase the workflow. The goal is to develop a model that can detect and recognize the targets.

Download the Dataset

This example uses a subset of the MSTAR clutter dataset that contains 300 training and 50 testing clutter images with five different targets. The data was collected using an X-band sensor in the spotlight mode with a one-foot resolution. The data contains rural and urban types of clutters. The types of targets used are BTR-60 (armoured car), BRDM-2 (fighting vehicle), ZSU-23/4 (tank), T62 (tank), and SLICY (multiple simple geometric shaped static target). The images were captured at a depression angle of 15 degrees. The clutter data is stored in the PNG image format and the corresponding ground truth data is stored in the `groundTruthMSTARClutterDataset.mat` file. The file contains 2-D bounding box information for five classes, which are SLICY, BTR-60, BRDM-2, ZSU-23/4, and T62 for training and testing data. The size of the dataset is 1.6 GB.

Download the dataset using the `helperDownloadMSTARClutterData` helper function, defined at the end of this example.

```
outputFolder = pwd;
dataURL = ('https://ssd.mathworks.com/supportfiles/radar/data/MSTAR_ClutterDataset.tar.gz');
helperDownloadMSTARClutterData(outputFolder,dataURL);
```

Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, download the dataset to a local disk using your web browser and extract the file. When using this approach, change the <outputFolder> variable in the example to the location of the downloaded file.

Download the Pretrained Network

Download the pretrained network from the link here using the `helperDownloadPretrainedSARDetectorNet` helper function, defined at the end of this example. The pretrained model allows you to run the entire example without having to wait for the training to complete. To train the network, set the `doTrain` variable to `true`.

```
pretrainedNetURL = ('https://ssd.mathworks.com/supportfiles/radar/data/TrainedSARDetectorNet.tar

doTrain = ▆▆;
if ~doTrain
    helperDownloadPretrainedSARDetectorNet(outputFolder,pretrainedNetURL);
end
```

Load the Dataset

Load the ground truth data (training set and test set). These images are generated in such a way that it places target chips at random locations on a background clutter image. The clutter image is constructed from the downloaded raw data. The generated target will be used as ground truth targets to train and test the network.

```
load('groundTruthMSTARClutterDataset.mat', "trainingData", "testData");
```

The ground truth data is stored in a six-column table, where the first column contains the image file paths and the second to the sixth columns contain the different target bounding boxes.

```
% Display the first few rows of the data set
trainingData(1:4,:)
```

ans=4×6 table

imageFilename	SLICY	BTR_60	BRDM_2
"./TrainingImages/Img0001.png"	{[285 468 28 28]}	{[135 331 65 65]}	{[597 739 65 6
"./TrainingImages/Img0002.png"	{[595 1585 28 28]}	{[880 162 65 65]}	{[308 1683 65 6
"./TrainingImages/Img0003.png"	{[200 1140 28 28]}	{[961 1055 65 65]}	{[306 1256 65 6
"./TrainingImages/Img0004.png"	{[623 186 28 28]}	{[536 946 65 65]}	{[131 245 65 6

Display one of the training images and box labels to visualize the data.

```
img = imread(trainingData.imageFilename(1));
bbox = reshape(cell2mat(trainingData{1,2:end}),[4,5])';
labels = {'SLICY', 'BTR_60', 'BRDM_2',  'ZSU_23_4', 'T62'};
annotatedImage = insertObjectAnnotation(img,'rectangle',bbox,labels,...
    'TextBoxOpacity',0.9,'FontSize',50);
figure
imshow(annotatedImage);
title('Sample Training Image With Bounding Boxes and Labels')
```

Sample Training image with bounding boxes and labels
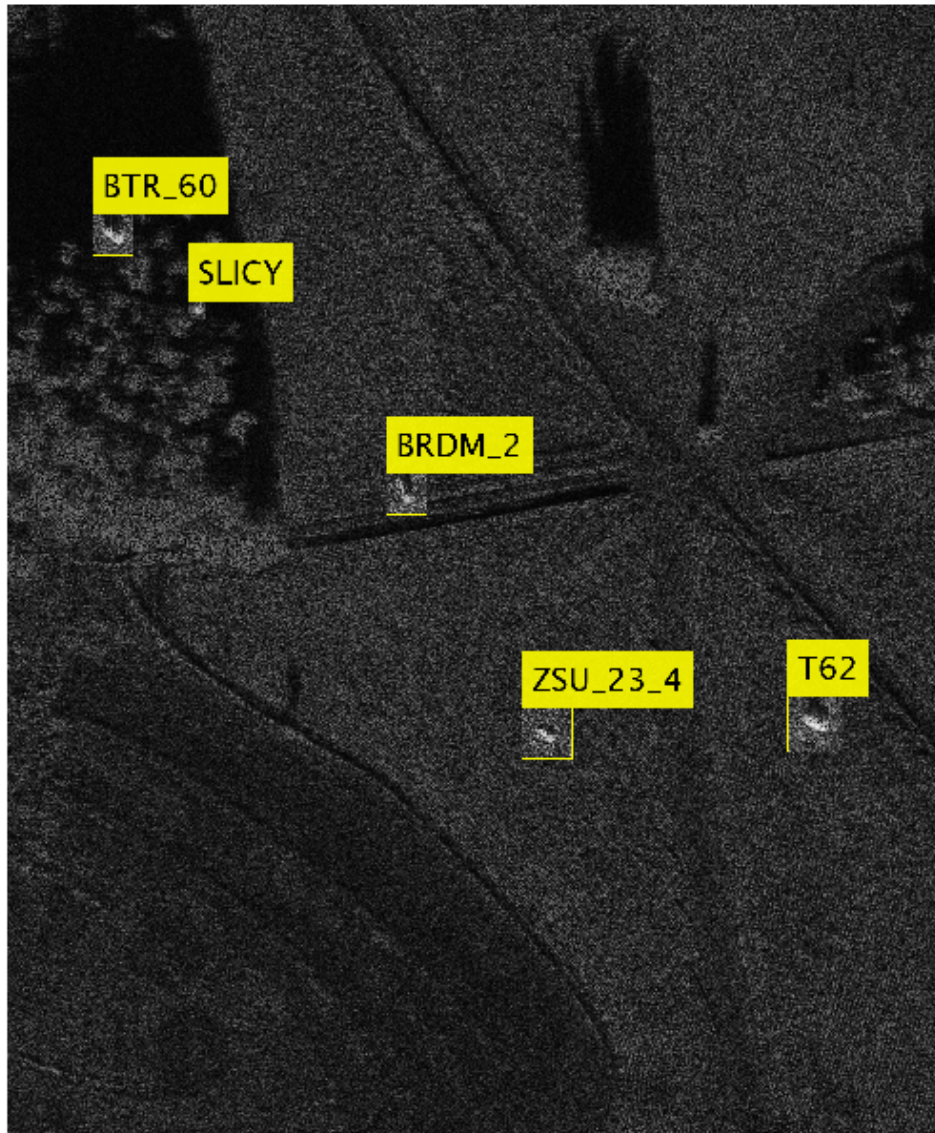
Define the Network Architecture

Create an R-CNN object detector for five targets: SLICY, BTR_60, BRDM_2, ZSU_23_4, T62.

```
objectClasses = {'SLICY', 'BTR_60', 'BRDM_2', 'ZSU_23_4', 'T62'};
```

The network must be able to classify the five targets and a background class in order to be trained using the `trainRCNNObjectDetector` function available in Deep Learning Toolbox™. 1 is added in the code below to include the background class.

```
numClassesPlusBackground = numel(objectClasses) + 1;
```

The final fully connected layer of the network defines the number of classes that it can classify. Set the final fully connected layer to have an output size equal to `numClassesPlusBackground`.

```matlab
% Define input size
inputSize = [128,128,1];

% Define network
layers = createNetwork(inputSize,numClassesPlusBackground);
```

Now, these network layers can be used to train an R-CNN based five-class object detector.

Train Faster R-CNN

Use `trainingOptions` (Deep Learning Toolbox) to specify network training options. `trainingOptions` by default uses a GPU if one is available (requires Parallel Computing Toolbox™ and a CUDA® enabled GPU with compute capability 3.0 or higher). Otherwise, it uses a CPU. You can also specify the execution environment by using the `ExecutionEnvironment` name-value argument of `trainingOptions`. To detect automatically if you have a GPU available, set `ExecutionEnvironment` to `auto`. If you do not have a GPU, or do not want to use one for training, set `ExecutionEnvironment` to `cpu`. To ensure the use of a GPU for training, set `ExecutionEnvironment` to `gpu`.

```matlab
% Set training options
options = trainingOptions('sgdm', ...
    'MiniBatchSize', 128, ...
    'InitialLearnRate', 1e-3, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.1, ...
    'LearnRateDropPeriod', 100, ...
    'MaxEpochs', 10, ...
    'Verbose', true, ...
    'CheckpointPath',tempdir,...
    'ExecutionEnvironment','auto');
```

Use `trainRCNNObjectDetector` to train R-CNN object detector if `doTrain` is true. Otherwise, load the pretrained network. If training, adjust `NegativeOverlapRange` and `PositiveOverlapRange` to ensure that training samples tightly overlap with ground truth.

```matlab
if doTrain
    % Train an R-CNN object detector. This will take several minutes
    detector = trainRCNNObjectDetector(trainingData, layers, options,'PositiveOverlapRange',[0.5
else
    % Load a previously trained detector
    preTrainedMATFile = fullfile(outputFolder,'TrainedSARDetectorNet.mat');
    load(preTrainedMATFile);
end
```

Evaluate Detector on a Test Image

To get a qualitative idea of the functioning of the detector, pick a random image from the test set and run it through the detector. The detector is expected to return a collection of bounding boxes where it thinks the detected targets are, along with scores indicating confidence in each detection.

```
% Read test image
imgIdx = randi(height(testData));
testImage = imread(testData.imageFilename(imgIdx));

% Detect SAR targets in the test image
[bboxes,score,label] = detect(detector,testImage,'MiniBatchSize',16);
```

To understand the results achieved, overlay the results with the test image. A key parameter is the detection threshold, the score above which the detector detected a target. A higher threshold will result in fewer false positives; however, it also results in more false negatives.

```
scoreThreshold =    0.8  ⟋  ;
```

```
% Display the detection results
outputImage = testImage;
for idx = 1:length(score)
    bbox = bboxes(idx, :);
    thisScore = score(idx);

    if thisScore > scoreThreshold
        annotation = sprintf('%s: (Confidence = %0.2f)', label(idx),...
            round(thisScore,2));
        outputImage = insertObjectAnnotation(outputImage, 'rectangle', bbox,...
            annotation,'TextBoxOpacity',0.9,'FontSize',45,'LineWidth',2);
    end
end
f = figure;
f.Position(3:4) = [860,740];
imshow(outputImage)
title('Predicted Boxes and Labels on Test Image')
```

Predicted boxes and labels on test image
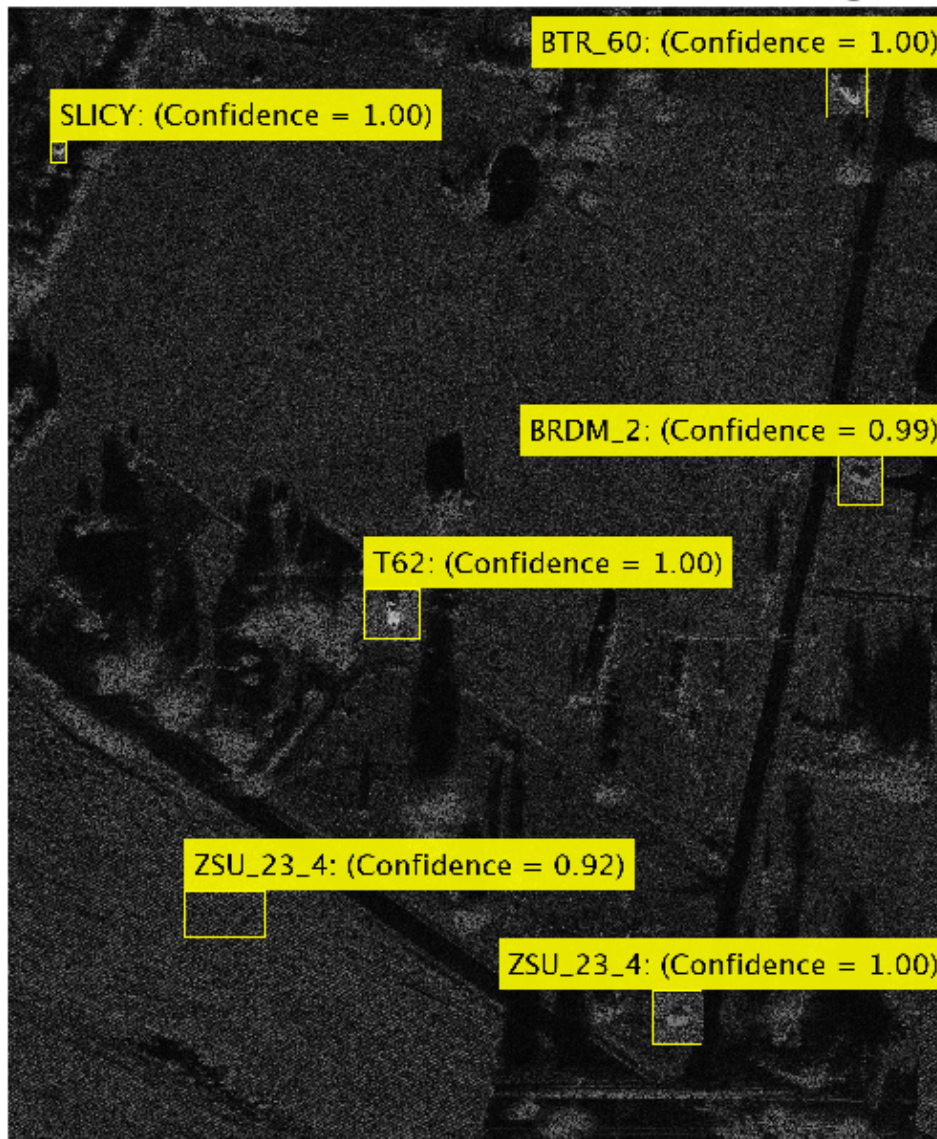
Evaluate Model

By looking at the images sequentially, you can understand the detector performance. To perform more rigorous analysis using the entire test set, run the test set through the detector.

```
% Create a table to hold the bounding boxes, scores and labels output by the detector
numImages = height(testData);
results = table('Size',[numImages 3],...
```

```
    'VariableTypes',{'cell','cell','cell'},...
    'VariableNames',{'Boxes','Scores','Labels'});

% Run detector on each image in the test set and collect results
for i = 1:numImages
    imgFilename = testData.imageFilename{i};

    % Read the image
    I = imread(imgFilename);

    % Run the detector
    [bboxes, scores, labels] = detect(detector, I,'MiniBatchSize',16);

    % Collect the results
    results.Boxes{i} = bboxes;
    results.Scores{i} = scores;
    results.Labels{i} = labels;
end
```

The possible detections and their bounding boxes for all images in the test set can be used to calculate the detector's average precision (AP) for each class. The AP is the average of the detector's precision at different levels of recall, so let us define precision and recall.

- Precision = $\frac{tp}{tp + fp}$

- Recall = $\frac{tp}{tp + fn}$

where

- tp - Number of true positives (the detector predicts a target when it is present)
- fp - Number of false positives (the detector predicts a target when it is not present)
- fn - Number of false negatives (the detector fails to detect a target when it is present)

A detector with a precision of 1 is considered good at detecting targets that are present, while a detector with a recall of 1 is good at avoiding false detections. Precision and recall have an inverse relationship.

Plot the relationship between precision and recall for each class. The average value of each curve is the AP. Plot curves for detection thresholds with the value of 0.5.

For more details, see `evaluateDetectionPrecision` (Computer Vision Toolbox).

```
% Extract expected bounding box locations from test data
expectedResults = testData(:, 2:end);

threshold = 0.5;
% Evaluate the object detector using average precision metric
[ap, recall, precision] = evaluateDetectionPrecision(results, expectedResults,threshold);

% Plot precision recall curve
f = figure; ax = gca; f.Position(3:4) = [860,740];
xlabel('Recall')
ylabel('Precision')
grid on; hold on; legend('Location', 'southeast');
title('Precision Vs Recall Curve for Threshold Value 0.5 for Different Classes');
```

```
for i = 1:length(ap)
% Plot precision/recall curve
    plot(ax,recall{i},precision{i},'DisplayName',['Average Precision for Class ' trainingData.Pro
end
```
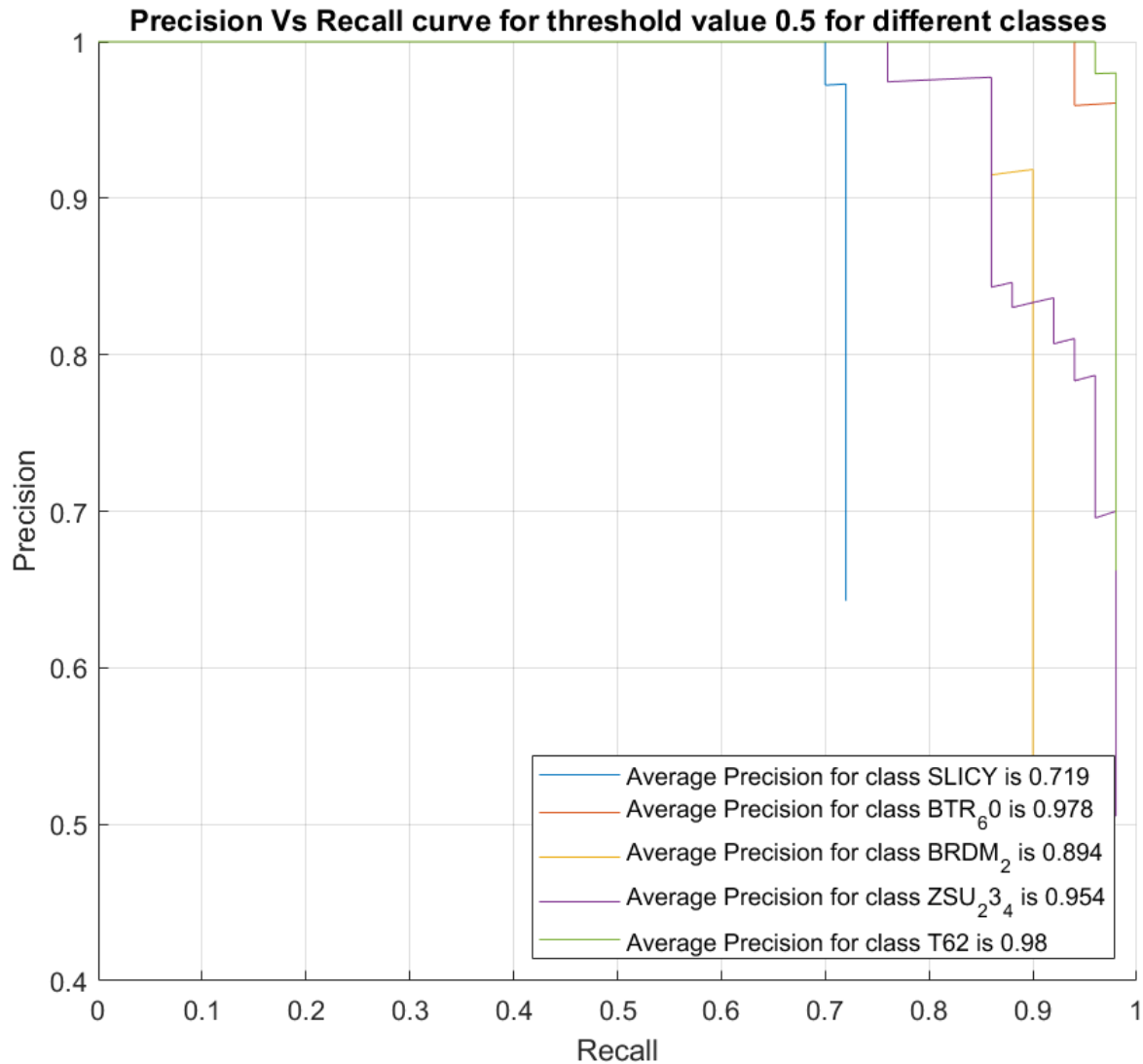

The AP for most of the classes is more than 0.9. Out of these, the trained model appears to struggle the most in detecting the SLICY targets. However, it is still able to achieve an AP of 0.7 for the class.

Helper Function

The function `createNetwork` takes as input the image size `inputSize` and number of classes `numClassesPlusBackground`. The function returns a CNN.

```
function layers = createNetwork(inputSize,numClassesPlusBackground)
    layers = [
        imageInputLayer(inputSize)                          % Input Layer
```

```matlab
        convolution2dLayer(3,32,'Padding','same')        % Convolution Layer
        reluLayer                                         % Relu Layer
        convolution2dLayer(3,32,'Padding','same')
        batchNormalizationLayer                           % Batch normalization Layer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)                   % Max Pooling Layer

        convolution2dLayer(3,64,'Padding','same')
        reluLayer
        convolution2dLayer(3,64,'Padding','same')
        batchNormalizationLayer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)

        convolution2dLayer(3,128,'Padding','same')
        reluLayer
        convolution2dLayer(3,128,'Padding','same')
        batchNormalizationLayer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)

        convolution2dLayer(3,256,'Padding','same')
        reluLayer
        convolution2dLayer(3,256,'Padding','same')
        batchNormalizationLayer
        reluLayer
        maxPooling2dLayer(2,'Stride',2)

        convolution2dLayer(6,512)
        reluLayer

        dropoutLayer(0.5)                                 % Dropout Layer
        fullyConnectedLayer(512)                          % Fully connected Layer.
        reluLayer
        fullyConnectedLayer(numClassesPlusBackground)
        softmaxLayer                                      % Softmax Layer
        classificationLayer                               % Classification Layer
        ];

end

function helperDownloadMSTARClutterData(outputFolder,DataURL)
% Download the data set from the given URL to the output folder.

    radarDataTarFile = fullfile(outputFolder,'MSTAR_ClutterDataset.tar.gz');

    if ~exist(radarDataTarFile,'file')

        disp('Downloading MSTAR Clutter data (1.6 GB)...');
        websave(radarDataTarFile,DataURL);
        untar(radarDataTarFile,outputFolder);
    end
end

function helperDownloadPretrainedSARDetectorNet(outputFolder,pretrainedNetURL)
% Download the pretrained network.

    preTrainedMATFile = fullfile(outputFolder,'TrainedSARDetectorNet.mat');
```

```
    preTrainedZipFile = fullfile(outputFolder,'TrainedSARDetectorNet.tar.gz');

    if ~exist(preTrainedMATFile,'file')
        if ~exist(preTrainedZipFile,'file')
            disp('Downloading pretrained detector (29.4 MB)...');
            websave(preTrainedZipFile,pretrainedNetURL);
        end
        untar(preTrainedZipFile,outputFolder);
    end
end
```

Summary

This example shows how to train an R-CNN for target recognition in SAR images. The pretrained network attained an accuracy of more than 0.9.

References

[1] *MSTAR Overview*. https://www.sdms.afrl.af.mil/index.php?collection=mstar

Introduction to Pulse Integration and Fluctuation Loss in Radar

The radar detectability factor is the minimum signal-to-noise ratio (SNR) required to declare a detection with the specified probabilities of detection, P_d, and false alarm, P_{fa}. The "Modeling Radar Detectability Factors" on page 1-388 example discusses in detail the computation of the detectability factor for a radar system given a set of performance requirements. It shows how to use the detectability factor in the radar equation to evaluate the maximum detection range. It also shows how to compute the effective probability of detection at a given range.

Typically, when a radar detectability factor is computed for N pulses received from a Swerling target, it already includes the pulse integration gain assuming noncoherent integration and a fluctuation loss due to the target radar cross section (RCS) fluctuation. However, if other pulse integration techniques are used, such as binary or cumulative, an additional integration loss must be added to the detectability factor. Similarly, if the system employs diversity, the target RCS fluctuation can be exploited to achieve a diversity gain.
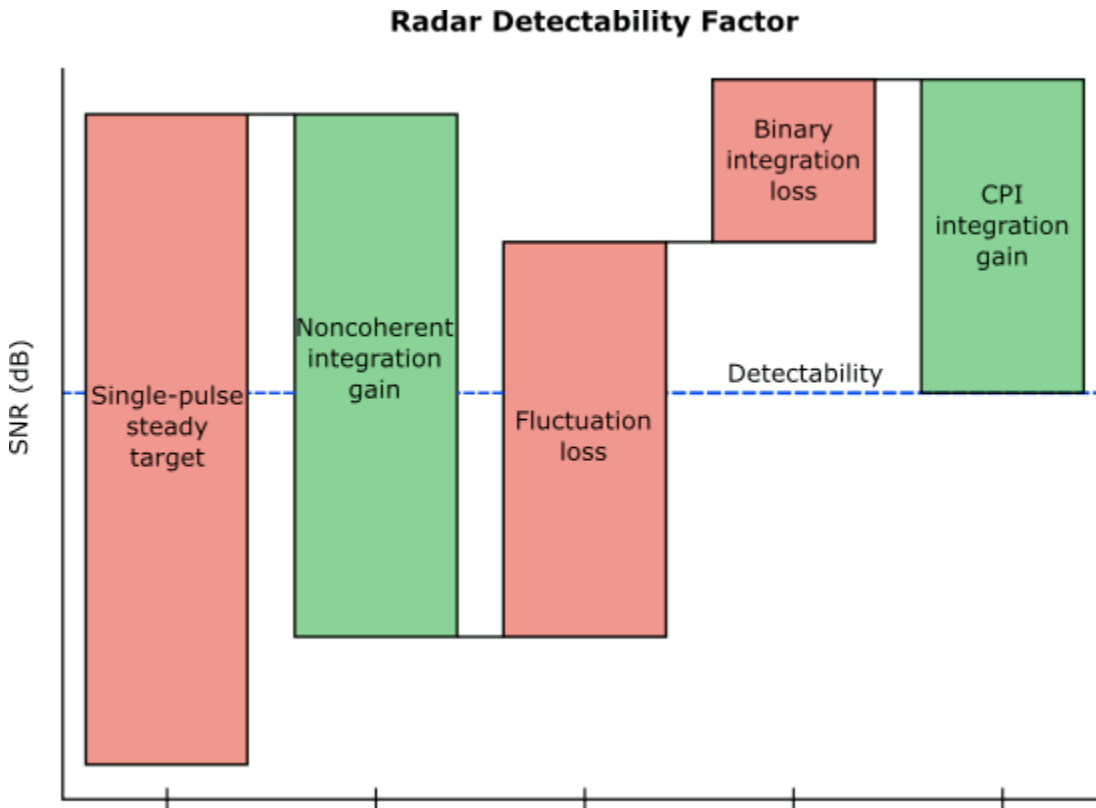
This example illustrates how to compute the pulse integration loss for several pulse integration techniques. It also shows computation of the losses due to the target's RCS fluctuation.

Pulse Integration

Typically, in a pulsed radar the required detection performance cannot be achieved with a single pulse. Instead, pulse integration is used to improve the SNR by adding signal samples together while averaging out the noise and interference.

Coherent and Noncoherent Integration

In coherent integration the complex signal samples are combined after coherent demodulation. Given that the samples are added in phase, the coherent integration process increases the available SNR by the number of integrated pulses, N. However, coherent integration might not always be possible due to the target's RCS fluctuation, which can make the coherent processing interval (CPI) too short to collect enough samples.

Noncoherent integration discards the phase information and combines the squared magnitudes (assuming the square-law detector) of the signal samples instead. Since the phase information is lost in this case, the integration gain of noncoherent integration is lower than that of coherent integration given the same number of received pulses. In addition, unlike the coherent integration gain, the gain from the noncoherent integration is also a function of P_d and P_{fa}. Examine how sensitive is the noncoherent integration gain to these parameters and compare it to the coherent integration gain.

Compute the integration gain as a function of the number of pulses N for several different values of P_d and P_{fa}.

```
% Number of received pulses
N = 1:100;
```

```
% Detection probability
Pd = [0.5 0.95];
```

```
% Probability of false alarm
Pfa = [1e-4 1e-8];
```

The noncoherent integration gain, G_i, can be defined as a difference (in dB) between a single-pulse detectability factor, $D_0(P_d, P_{fa}, 1)$, and an N-pulse detectability factor, $D_0(P_d, P_{fa}, N)$, both computed for a steady target ("0" subscript indicates the steady target i.e. the Swerling 0 case)

$G_i = D_0(P_d, P_{fa}, 1) - D_0(P_d, P_{fa}, N)$ [1].

```
% Single-pulse detectability factor
D0 = detectability(Pd,Pfa,1);
```

```
numPdPfa = numel(Pd)*numel(Pfa);
```

```
% N-pulse detectability
D0n = zeros(numPdPfa,numel(N));
```

```
for i = 1:numel(N)
    D0n(:,i) = reshape(detectability(Pd,Pfa,N(i)),[],1);
end
```

```
% Noncoherent integration gain
Gnc = reshape(D0,[],1)-D0n;
```

The coherent integration gain does not depend on P_d and P_{fa}, and simply equals the number of integrated pulses.

```
% Coherent integration gain
Gc = pow2db(N);
```

We plot the computed gains on a single plot together with a $N^{1/2}$ line, which is a commonly used approximation for the noncoherent gain.

```
figure
semilogx(N,Gnc(1:2,:),'LineWidth',2)
hold on
semilogx(N,Gnc(3:4,:),'--','LineWidth',2)
semilogx(N,Gc,'k','LineWidth',2)
semilogx(N,pow2db(sqrt(N)),'k:','LineWidth',2)

xlabel('Number of Integrated Pulses')
ylabel('Gain (dB)')
title('Pulse Integration Gain')

labels = helperLegendLabels('Noncoherent P_d=%.2f','P_{fa}=%.0e',Pd,Pfa);
labels{end + 1} = 'Coherent';
labels{end + 1} = 'N^{1/2}';

xticks = [1 2 5 10 20 50 100];
set(gca(),'xtick',xticks,'xticklabel',num2cell(xticks))
legend(labels,'location','best')
grid on

yyaxis(gca(),'right')
set(gca(),'ycolor','k')
ylabel('Exponent of N')
```
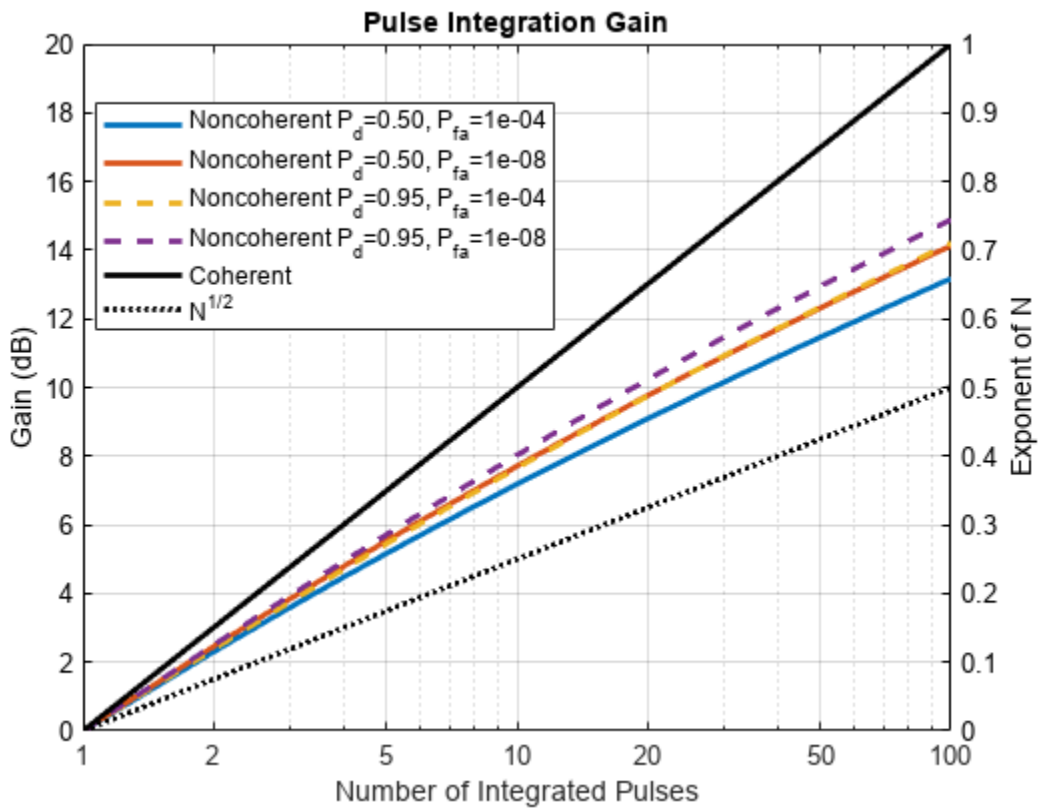
From this result you can observe that the noncoherent integration gain is not very sensitive to P_d and P_{fa}. It also appears to be significantly higher than the $N^{1/2}$ approximation with the actual values being between $N^{0.6}$ and $N^{0.8}$ [1, 2].

Binary Integration

Binary integration, also known as the M-of-N integration, is a double-threshold system. The first threshold is applied to each pulse resulting in N detection outcomes each with a probability of detection, p_d, and false alarm, p_{fa}. These outcomes are then combined and compared to the second threshold. If at least M out of N pulses cross the first threshold, a target is declared to be present. Given the detection and the false alarm probabilities of the individual outcomes, the resultant P_d and P_{fa} after the binary detection are

$$P_d = \sum_{k = M}^{N} \binom{N}{k} p_d^k (1 - p_d)^{N - k}$$

$$P_{fa} = \sum_{k = M}^{N} \binom{N}{k} p_{fa}^k (1 - p_{fa})^{N - k}$$

Thus, the desired P_d can be achieved starting with a significantly lower detection probability in a single detection. Similarly, the single pulse p_{fa} could be set to a higher value than the required P_{fa}. For $P_d = 0.95, P_{fa} = 10^{-8}$, $N = 4$, and $M = 2$ determine the probabilities of detection and false alarm for each individual detection.

```
[~,pd,pfa] = binaryintloss(0.95,1e-8,4,2)
```

```
pd = 0.7514
```

```
pfa = 4.0826e-05
```

Since binary integration is suboptimal, it results in a binary integration loss compared to the optimal noncoherent integration. For a given set of P_d, P_{fa}, and N this loss depends on the choice of M. However, the optimal value of M is not a sensitive selection and it can be different from the optimum without significant penalty. For a nonfluctuating target, a good choice of M was shown to be $M = 0.955 N^{0.8}$ [3].

Compute the binary integration loss as a function of N for several values of P_d and P_{fa}.

```
% Binary integration loss
Lb = zeros(numPdPfa,numel(N));

for i = 1:numel(N)
    Lb(:,i) = reshape(binaryintloss(Pd,Pfa,N(i)),[],1);
end

figure
semilogx(N,Lb,'LineWidth',2)

xlabel('Number of Integrated Pulses')
ylabel('Loss (dB)')
title({'Binary Integration Loss','M=0.955N^{0.8}'})
```

```
set(gca(),'xtick',xticks,'xticklabel',num2cell(xticks))
labels = helperLegendLabels('P_d=%.2f','P_{fa}=%.0e',Pd,Pfa);
legend(labels,'location','best')
grid on
```
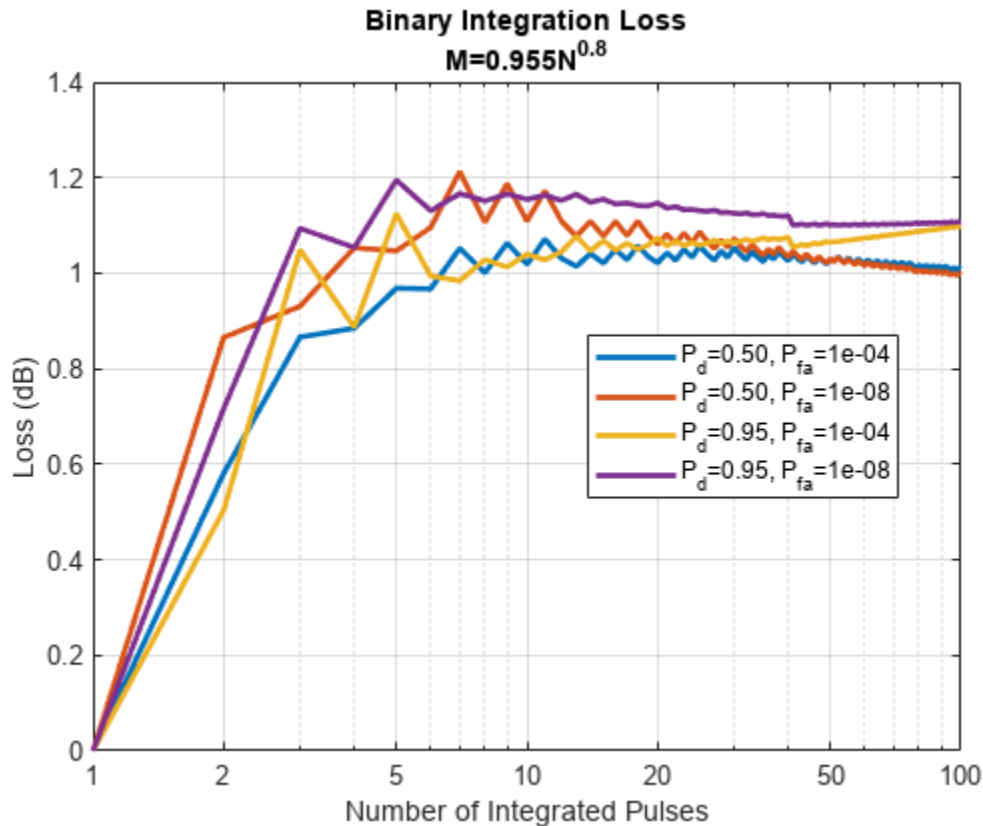

The resultant loss is around 1 to 1.2 dB and is not very sensitive to P_d, P_{fa}, and N. The binary integrator is a relatively simple automatic detector. It is also robust to non-Gaussian background noise and clutter.

M-of-N Integration Over Multiple CPIs

M-of-N integration scheme can also be applied to multiple CPIs or multiple scans. This might be necessary when detecting high-velocity targets that can move across several resolution cells during the integration time. In this case, N pulses are divided into n groups, where n is the number of CPIs. Within each CPI, N/n pulses can be coherently or noncoherently integrated resulting in probabilities of detection, p_d, and false alarm, p_{fa}. The target is declared to be present if it was detected in at least m CPIs out of n. The detectability factor for M-of-N integration assuming required $P_d = 0.95$ and $P_{fa} = 10^{-8}$ is computed here as a function of the total number of pulses, N, for several choices of the M-of-N threshold.

```
% M-of-N detection thresholds: 1-of-2, 2-of-3, and 1-of-3
n = [2 3 3];
m = [1 2 1];

% Detectability factor for M-of-N CPI integration
Dmn = zeros(numel(n),numel(N));
```

```
for i = 1:numel(n)
    % Detection and false alarm probabilities required in a single CPI
    [~,pd,pfa] = binaryintloss(Pd(2),Pfa(2),n(i),m(i));

    for j = 1:numel(N)
        Dmn(i,j) = detectability(pd,pfa,N(j)/n(i));
    end
end
```

The resultant loss is the difference (in dB) between the SNR required when M-of-N integration is performed over *n* CPIs and the SNR required when all *N* pulses are processed within a single CPI.

$L_{mn} = D_0(p_d, p_{fa}, N/n) - D_0(P_d, P_{fa}, N).$

```
% M-of-N integration loss with respect to the optimal noncoherent
% integration
Lmn = Dmn-D0n(4,:);

figure
semilogx(N,Lmn,'LineWidth',2)

xlabel('Number of Integrated Pulses')
ylabel('Loss (dB)')
title({'Loss for M-of-N Integration over Multiple CPIs',sprintf('P_d=%.2f, P_{fa}=%.0e',Pd(2),Pfa

set(gca(),'xtick',xticks,'xticklabel',num2cell(xticks))

labels = arrayfun(@(i)sprintf('%d-of-%d',m(i),n(i)), 1:numel(n),'UniformOutput',false);
legend(labels,'location','south')
grid on
```
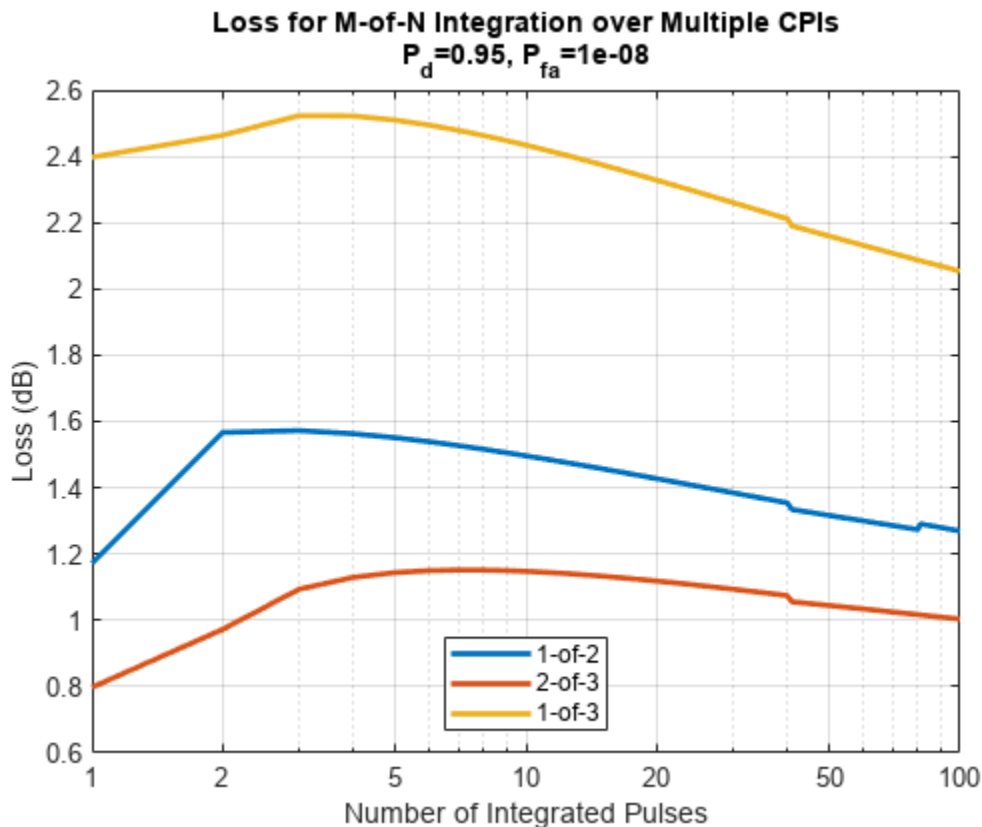
This result shows that the loss is sensitive to the chosen M-of-N threshold, and that it tends to decrease with N, although the variation with the number of pulses is not very strong.

The results above compute the integration loss with respect to optimal noncoherent integration. Alternatively, different integration methods can be compared with respect to coherent integration. Compare noncoherent integration, binary integration, and a special case of the M-of-N integration with $M = 1$ (called cumulative integration) for $P_d = 0.95$ and $P_{fa} = 10^{-8}$.

Compute the detectability factor for the coherent case assuming a single pulse is detected in the envelope detector after integrating N pulses coherently.

```
% Detectability factor for coherent integration
Dc = detectability(Pd(2),Pfa(2),1) - 10*log10(N);
```

Then compute the losses for different integration types with respect to coherent integration.

```
% Noncoherent integration loss
Li = D0n(4,:)-Dc;
```

```
%  Binary integration loss
Lbi = D0n(4,:)-Dc+Lb(4,:);
```

```
% Cumulative integration loss with 1-of-3 detection
Lcum = D0n(4,:)-Dc+Lmn(3,:);
```

Plot and compare these integration losses for different values of the total number of integrated pulses.

```
figure
semilogx(N,zeros(size(Dc)),'LineWidth',2)
hold on
semilogx(N,Li,'LineWidth',2)
semilogx(N,Lbi,'LineWidth',2)
semilogx(N,Lcum,'LineWidth',2)

xlabel('Number of Integrated Pulses')
ylabel('Loss (dB)')
title({'Integration Loss for Different Pulse Integration Methods',sprintf('P_d=%.1f, P_{fa}=%.0e

set(gca(),'xtick',xticks,'xticklabel',num2cell(xticks))
legend({'Coherent','Noncoherent','Binary','Cumulative'},'location','northwest')
ylim([-1 8])
grid on
```
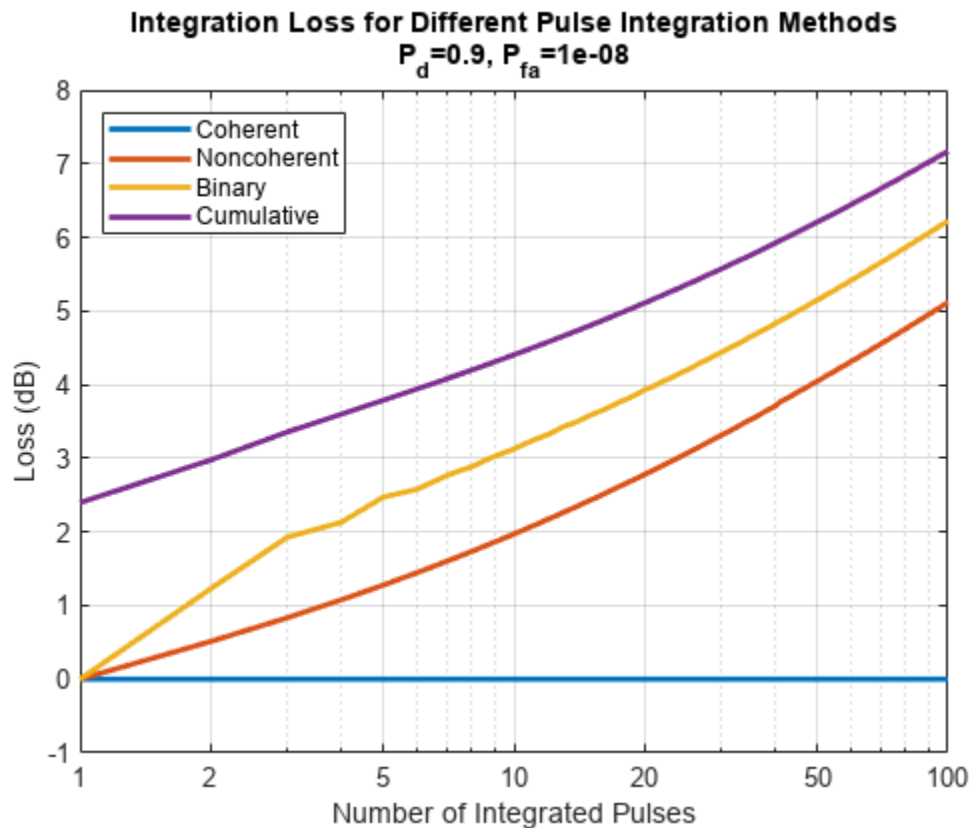

Fluctuation Loss

Since the RCS of a real-world target fluctuates, the signal energy required to achieve a given probability of detection is higher compared to that required for a steady target. This increase in the energy is the fluctuation loss. Evaluate the fluctuation loss as a function of the probability of detection for different values of P_{fa} and N.

```
% Detection probability
Pd = linspace(0.01,0.995,100);

% Probability of false alarm
```

```
Pfa = [1e-8 1e-4];

% Number of received pulses
n = [1 10 50];
```

The fluctuation loss can be computed as a difference (in dB) between the detectability factor for the fluctuating target, $D_k(P_d, P_{fa}, N)$, and the detectability factor for the steady target, $D_0(P_d, P_{fa}, N)$

$$L_{kf} = D_k(P_d, P_{fa}, N) - D_0(P_d, P_{fa}, N)$$

where the subscript $k = 1, 2, 3, 4$ indicates the Swerling case [1]. This example performs the computation for Swerling 1 and Swerling 2 cases that model slow and fast fluctuating targets, respectively.

```
% Fluctuation loss for Swerling 1 case
L1f = zeros(numel(Pd),numel(Pfa),numel(n));

% Fluctuation loss for Swerling 2 case
L2f = zeros(numel(Pd),numel(Pfa),numel(n));

for i = 1:numel(n)
    % Detectability factor for a steady target
    D0n = detectability(Pd,Pfa,n(i),'Swerling0');

    % Detectability factor for Swerling 1 case fluctuating target
    D1n = detectability(Pd,Pfa,n(i),'Swerling1');

    % Detectability factor for Swerling 2 case fluctuating target
    D2n = detectability(Pd,Pfa,n(i),'Swerling2');

    L1f(:,:,i) = D1n-D0n;
    L2f(:,:,i) = D2n-D0n;
end
```

The computed fluctuation loss is plotted here against the probability of detection. This result shows that in the Swerling 1 case, when there is no pulse-to-pulse RCS fluctuation, the loss is not very sensitive to the number of pulses. However, it is very sensitive to the probability of detection. High values of the required P_d result in a large fluctuation loss. In the case of the Swerling 2 model, the RCS fluctuates from pulse-to-pulse, therefore the fluctuation loss is very sensitive to the number of pulses and decreases rapidly with N. The fluctuation loss is not a strong function of P_{fa} in both Swerling 1 and 2 cases.

```
figure
ax1 = subplot(1,2,1);
helperPlotFluctuationLoss(ax1,Pd,L1f)

title(ax1,{'Fluctuation Loss','Swerling 1 Case'});
labels = helperLegendLabels('P_{fa}=%.0e','N=%d',Pfa,n);
legend(labels)

ax2 = subplot(1,2,2);
helperPlotFluctuationLoss(ax2,Pd,L2f)

title(ax2,{'Fluctuation Loss','Swerling 2 Case'})
legend(labels)

set(gcf,'Position',[100 100 800 600])
```
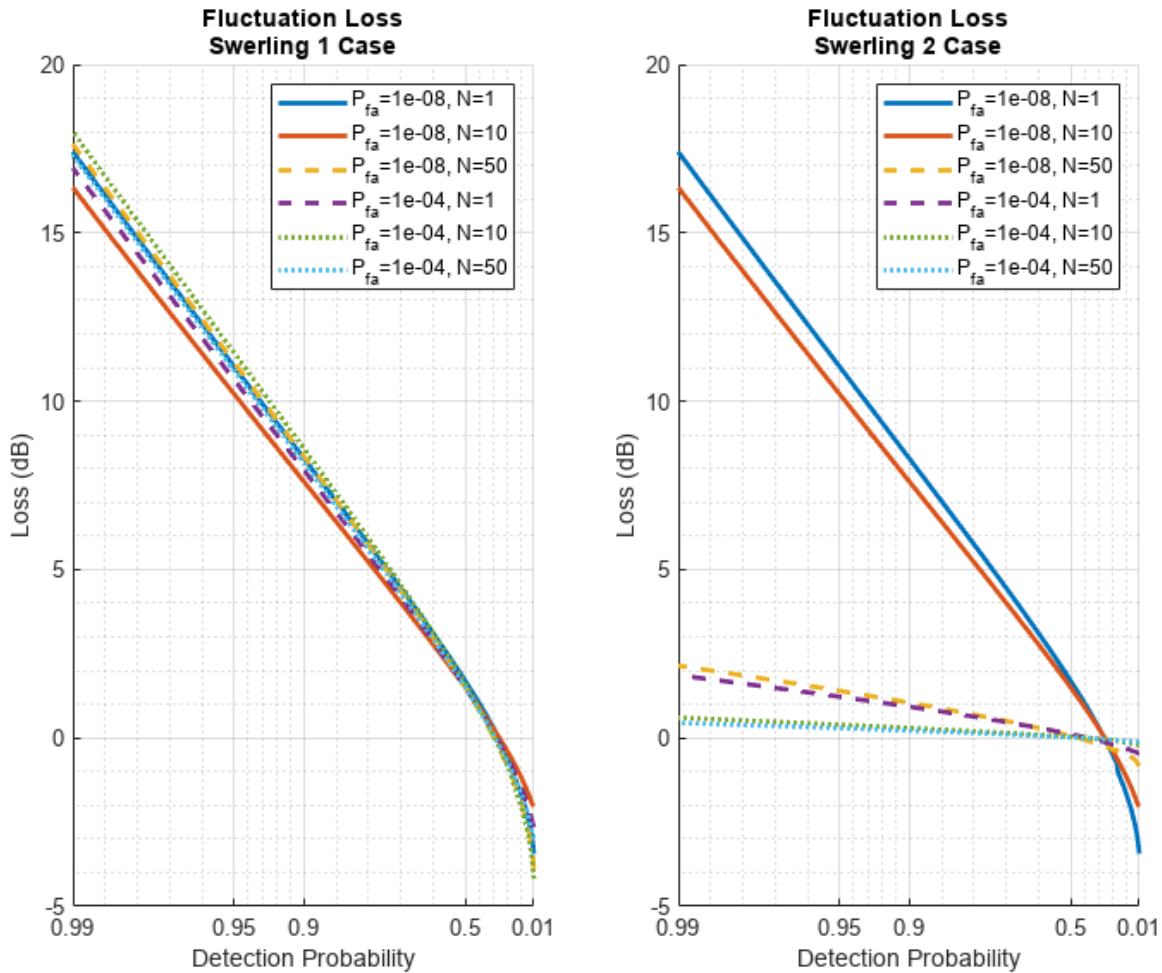
Diversity Gain

The Swerling 2 case can also be used to represent target samples obtained by a radar system with diversity (for example, frequency, space, polarization, among others). As was shown in the earlier section, the fluctuation loss decreases with the number of received pulses for a Swerling 2 target. This indicates that obtaining more samples with diversity can reduce the fluctuation loss. Consider N pulses received during a dwell time. These pulses might be integrated coherently to provide a single pulse at the input to the envelope detector. However, if the radar system can transmit at M different frequencies, N pulses could be transmitted in M groups of N/M. For example, if a radar system transmits a total of $N = 16$ pulses, they could be transmitted in two groups of eight, four groups of four, eight groups of two, or all 16 pulses at different frequencies.

```
% Total number of transmitted pulses
N = 16;

% Number of diversity samples
M = [1 2 4 8 16];
```

Within each frequency group coherent integration can be applied to the received pulses providing M diverse samples, which then can be aggregated by noncoherent integration. If the frequencies are

chosen such that the echoes at different frequencies are decorrelated, there is going to be an optimal number of diversity samples that minimizes the required signal energy. Assume the following values for the required P_d and P_{fa}.

```
% Detection probability
Pd = [0.9 0.7 0.5];

% Probability of false alarm
Pfa = 1e-6;
```

To provide the required detection performance, the energy in each diversity sample must be equal to $D_2(P_d, P_{fa}, M)$. Since the pulses within each group are coherently integrated, the required SNR for each pulse equals $D = (M/N)D_2(P_d, P_{fa}, M)$.

```
% Detectability factor
D = zeros(numel(Pd),numel(M));

for i = 1:numel(M)
    D(:,i) = detectability(Pd,Pfa,M(i),'Swerling2') - 10*log10(N/M(i));
end
```

The following plot shows the detectability factor as a function of the number of diversity samples. From this result you can see that for each value of P_d there is an optimal value of M. For example, when P_d is 0.9, the best result is achieved when transmitting eight groups of two pulses. The detectability factor in this case is almost 5 dB lower than when all 16 pulses are integrated coherently. This is a diversity gain due to utilizing multiple frequencies.

```
figure
semilogx(M,D,'LineWidth',2)

ylabel('Detectability (dB)')
xlabel('Number of Diversity Samples')
title({'Detectability vs Number of Diversity Samples',sprintf('P_{fa}=%.0e',Pfa)})

set(gca(),'xtick',M,'xticklabel',num2cell(M))
legend(helperLegendLabels('P_{d}=%.1f',Pd))
grid on
```

Detectability vs Number of Diversity Samples
$P_{fa}=1e-06$
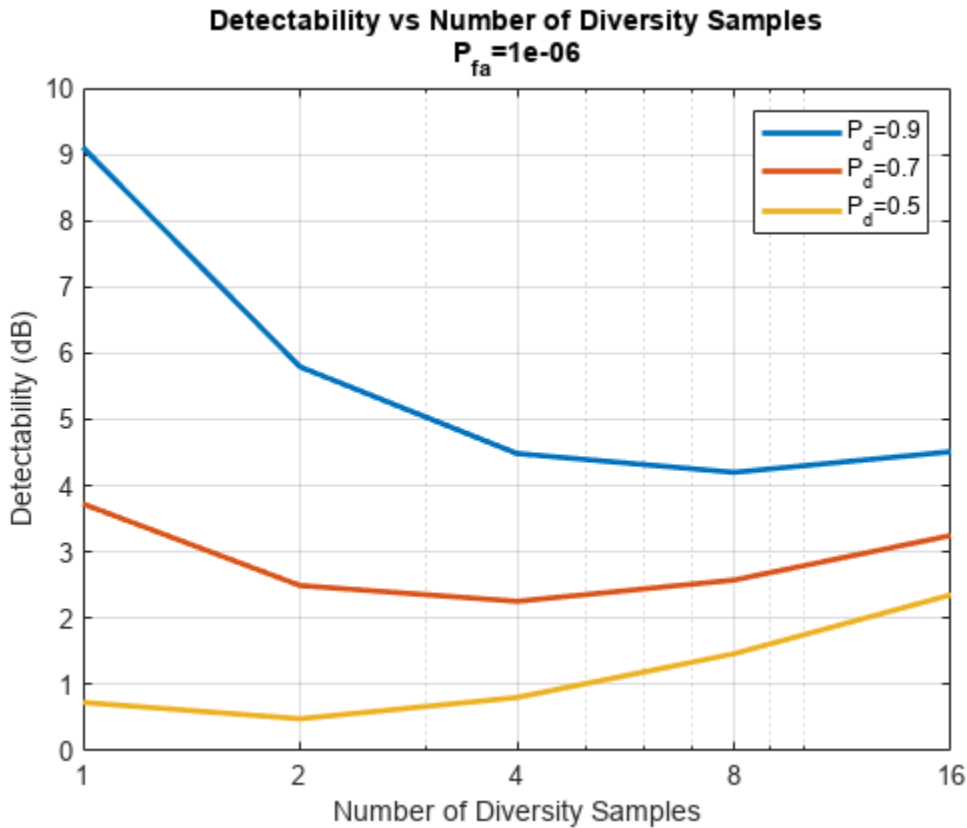
Conclusion

This example discusses pulse integration and fluctuation losses included in the effective detectability factor. It starts by introducing coherent, noncoherent, binary, and cumulative pulse integration techniques and examines how the resulting integration loss depends on the detection parameters, that is, the probability of detection, probability of false alarm, and the number of received pulses. The fluctuation loss is discussed next. The example shows that in the case of a slowly fluctuating target the required high detection probability results in the large fluctuation loss, while changes in the false alarm and the number of received pulses have a much lesser impact. On the other hand, the fluctuation loss in the case of a rapidly fluctuating target decreases rapidly with the number of received pulses. It is then shown that the RCS fluctuation together with the frequency diversity can be employed to achieve a diversity gain.

References

1   Barton, David Knox. *Radar Equations for Modern Radar*. Artech House, 2013.
2   Richards, M. A. "Noncoherent integration gain, and its approximation." *Georgia Institute of Technology, Technical Memo* (2010).
3   Richards, M. A. *Fundamentals of Radar Signal Processing*. Second edition, McGraw-Hill Education, 2014.

Introduction to Scanning and Processing Losses in Pulse Radar

In an idealized radar system with no losses and a nonfluctuating target, the detectability factor is a function of only three parameters — the desired probability of detection P_d, the required probability of false alarm P_{fa}, and the number of received pulses N. Practical systems, however, employ suboptimal processing resulting in a series of losses that must be added to the detectability factor. These losses increase the required signal energy needed to satisfy the stated detection requirements. Thus, the effective detectability factor becomes dependent on the components of the signal processing chain, the type of the pulse integration, the target fluctuation model, and several other factors. This example shows how various parameters influence the losses that must be included in the radar detectability factor when evaluating the radar equation. It discusses losses caused by the pulse eclipsing effect, off-broadside scanning with an electronic beam, and moving target indication (MTI) processing. It also addresses constant false alarm rate (CFAR) loss and filter matching loss.
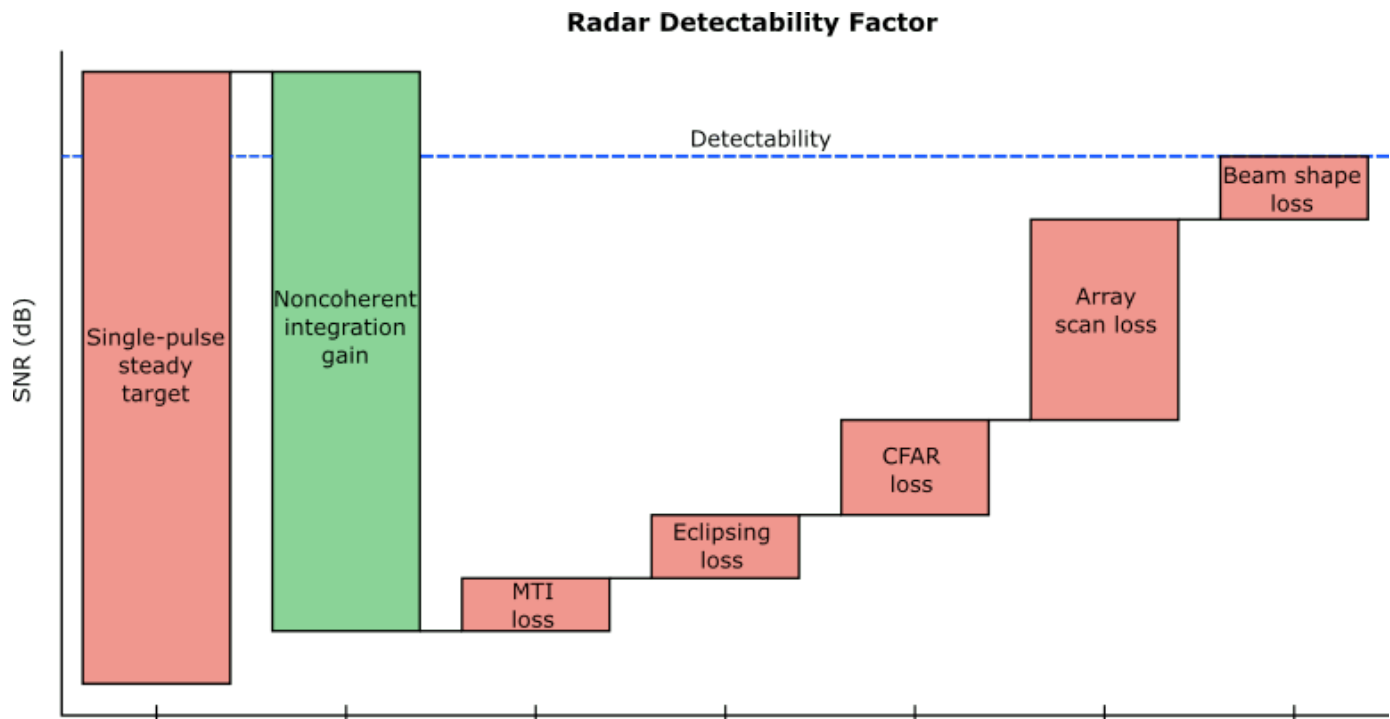
Radar Detectability Factor

The "Modeling Radar Detectability Factors" on page 1-388 example discusses in detail the computation of the detectability factor for a radar system given a set of performance requirements. It shows how to use the detectability factor in the radar equation to evaluate the maximum detection range. It also shows how to compute the effective probability of detection at a given range.

Statistical Losses

The effective probability of detection, p, is a function of the available SNR at the receiver, $\bar{\chi}$

$$p = p(\bar{\chi}).$$

For a specific target position, the available SNR, $\bar{\chi}_0$, can be computed from the radar equation, which typically assumes that the observed target is in the middle of a range-Doppler cell and lies directly

along the antenna axis. However, since the target's position varies in the four-dimensional radar space, the probability of detection will also vary

$$p = p(\bar{\chi}_0 H(x))$$

where $H(x)$ is a scaling factor indicating the change in the available SNR when the target is located at a position x in the range-Doppler-angle space. Thus, it makes sense to introduce the average probability of detection for a region bounded by the target positions x_1 and x_2

$$\bar{P} = \frac{1}{x_2 - x_1} \int_{x_1}^{x_2} p(\bar{\chi}_0 H(x)) dx$$

Here the probability that a target echo arrives from the position x is assumed to be uniformly distributed between the boundaries x_1 and x_2. Thus, the SNR $\bar{\chi}_s$ needed to achieve the required P_d on average can be found by solving the following equation

$$P_d - \frac{1}{x_2 - x_1} \int_{x_1}^{x_2} p(\bar{\chi} H(x)) dx = 0$$

The increase in the SNR, $\bar{\chi}_s$, compared to the idealized case when there are no probability of detection variations with x, is called a statistical loss [1]

$$L_s = \frac{\bar{\chi}_s}{\bar{\chi}_0}$$

This example considers three statistical losses:

- Eclipsing loss — Accounts for P_d variations with range due to the pulse eclipsing effect.
- Array scan sector loss — Accounts for P_d variations with a scan angle in electronically scanned arrays due the reduced projected array area in the beam direction and a reduction of the effective aperture area of the individual array elements at off-broadside angles.
- MTI velocity response loss — Accounts for P_d variations due to a target lying close or in a null of the MTI filter.

Eclipsing Loss

Typically, pulse radar systems turn off their receivers during the pulse transmission. Thus, the target echoes arriving from the ranges within one pulse length from the radar or within one pulse length around the unambiguous range will be eclipsed by the transmitted pulse resulting in only a fraction of the pulse being received and processed. For different values of the duty cycle, the fraction of the received signal energy due to eclipsing is shown here as a function of the target range assuming the pulse repetition frequency (PRF) is 1kHz.

```
% Duty cycle
Du = [0.02 0.05 0.1 0.2];

% Pulse repetition frequency
PRF = 1e3;

% Compute eclipsing factor at 1 km intervals between zero and the
% unambiguous range
R = 0:1000:time2range(1/PRF);
Fecl = zeros(numel(R),numel(Du));
```

```
for i = 1:numel(Du)
    Fecl(:,i) = eclipsingfactor(R,Du(i),PRF);
end

% Plot the eclipsing factor in linear units
figure
plot(R*1e-3,db2pow(Fecl),'LineWidth',2)

xlabel('Range (km)')
ylabel('Received Energy Fraction')
title({'Eclipsing Effect',sprintf("PRF=%d Hz",PRF)})

legend(helperLegendLabels('Duty cycle = %.2f',Du),'location','south')
grid
ylim([0 1.2])
```

Eclipsing Effect
PRF=1000 Hz
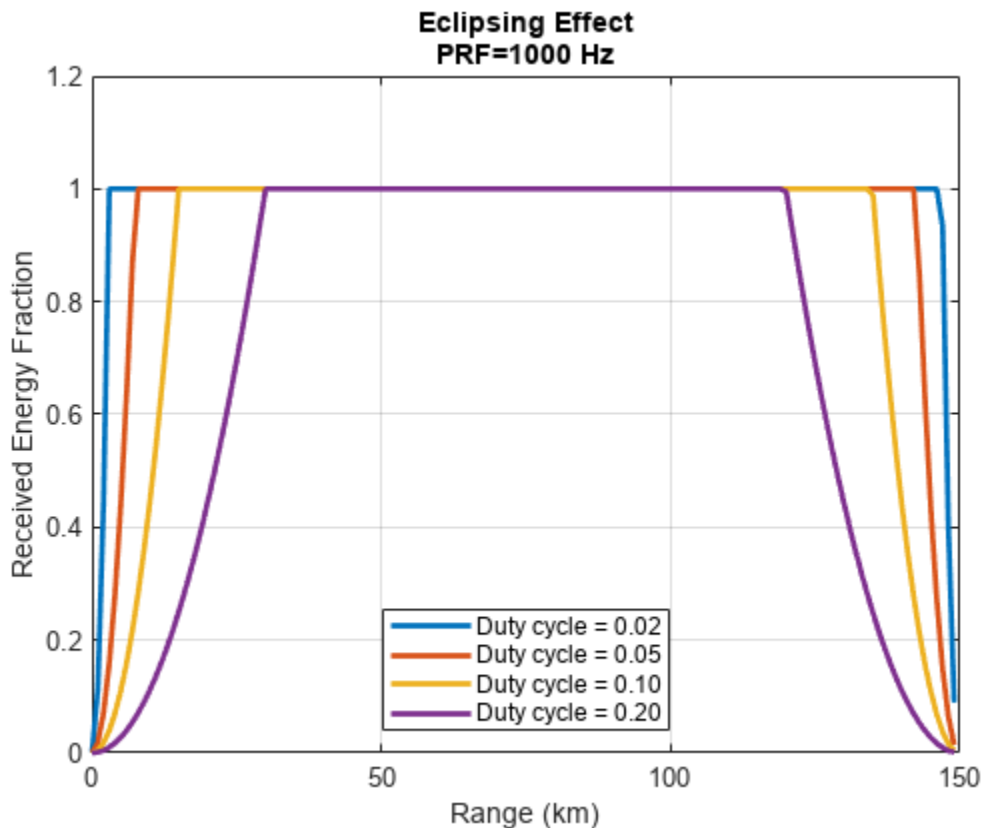
The variations in the received signal energy due to the eclipsing effect causes variations in the probability of detection. Specifically, P_d decreases rapidly as a larger and larger fraction of the pulse is eclipsed when the target range approaches zero or the unambiguous range. The statistical eclipsing loss is computed as an increase in the signal energy required to compensate for these variations and make the probability of detection, averaged over the target ranges, equal to the desired probability of detection. Compute the statistical eclipsing loss as a function of P_d for $P_{fa} = 10^{-6}$ and different values of the duty cycle.

```
% Detection probability
Pd = linspace(0.1,0.99,100);
```

```matlab
% Probability of false alarm
Pfa = 1e-6;

% Eclipsing loss
Lecl = zeros(numel(Pd),numel(Du));

for i = 1:numel(Du)
    % Assume a single pulse is received from a Swerling 1 target
    Lecl(:,i) = eclipsingloss(Pd,Pfa,1,Du(i),'Swerling1');
end

figure
plot(Pd,Lecl,'LineWidth',2)

xlabel('Detection Probability')
ylabel('Loss (dB)')
title({'Eclipsing Loss',sprintf('Swerling 1 Case, P_{fa}=%.0e',Pfa)})

legend(helperLegendLabels('Duty cycle = %.2f',Du),'location','northwest')
grid on
```
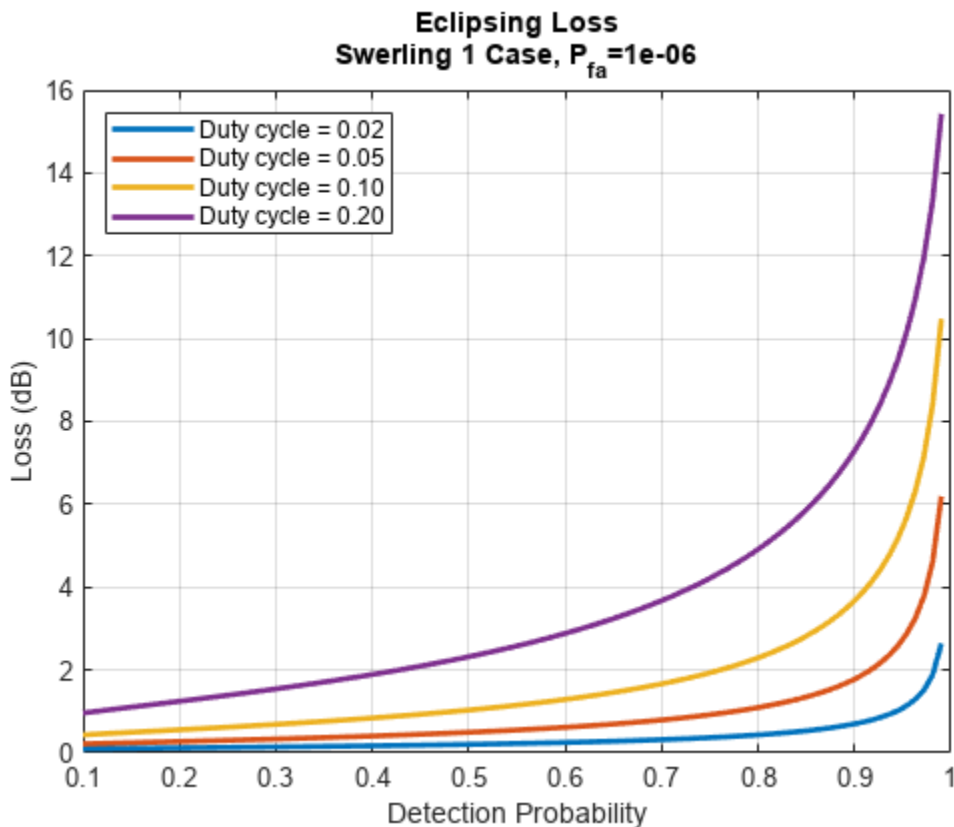

The statistical eclipsing loss increases with the duty cycle and can become very large for high values of P_d. This loss should be included in the radar equation when analyzing high-PRF radar systems in which a target can pass rapidly through the eclipsing regions. In such systems, the PRF diversity is typically used to mitigate the eclipsing effects by shifting the eclipsing regions in range.

Scan Sector Loss

In radar systems with electronically scanned phased array antennas, the gain of the antenna array scanned off broadside is reduced due to the reduction of the projected array area in the beam direction and the reduction of the effective aperture area of the individual array elements at off-broadside angles. Assuming both the transmitter and the receiver are using the same antenna array, at an off-broadside angle θ the received energy is reduced by approximately a factor of $\cos^3\theta$.

```
theta = linspace(-90,90,500);

figure
plot(theta,cosd(theta).^3,'LineWidth',2)

xlabel('Scan Angle (deg)')
ylabel('cos^3(\theta)')
title('Reduction in Antenna Gain for a Phased Array Scanning off Broadside')
grid on
```
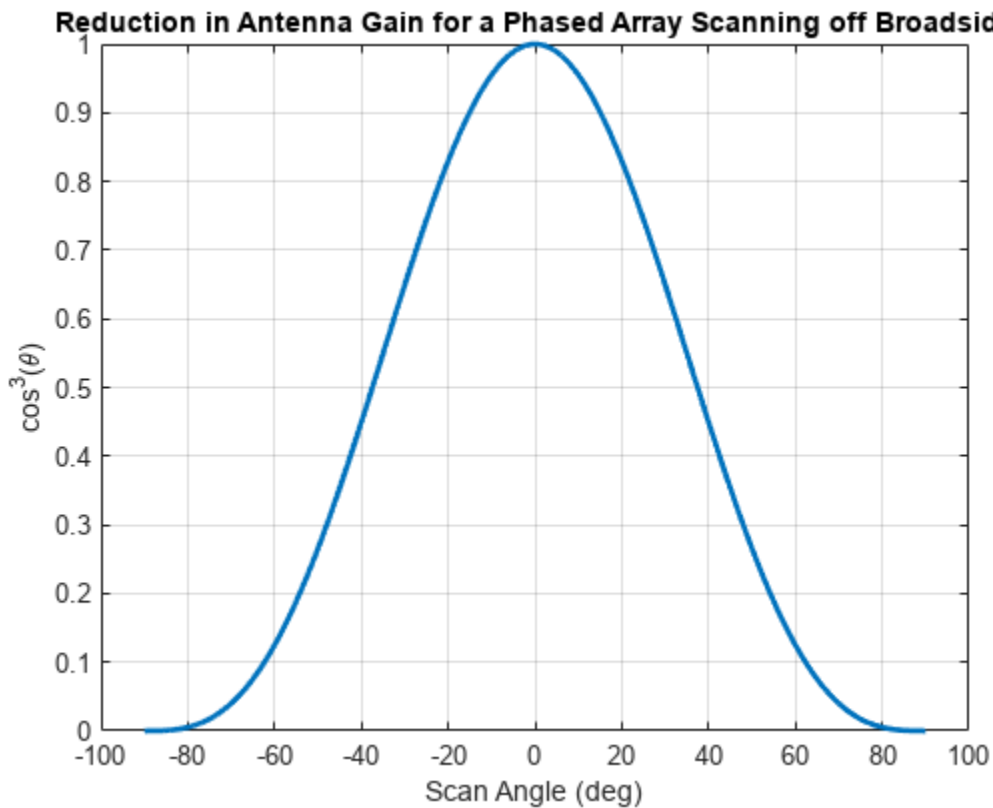

This figure shows that the received energy is reduced by half when the beam is pointed to about 37 degrees off broadsided.

The statistical scan sector loss defined for a scan sector $[\theta_1, \theta_2]$ is an increase in the signal energy required to obtain a specific value of the average P_d over this scan sector compared to the signal energy required to obtain the same P_d when the target lies along the antenna axis.

Consider scan sectors of different sizes and compute the statistical scan sector loss as a function of P_d for $P_{fa} = 10^{-6}$. The value of θ shown in the legend is the maximum scan angle specified about the broadside direction, that is the array scans from $-\theta$ to $+\theta$.

```
% Maximum scan angle about the broadside direction (deg)
Theta = [15 30 45 60];

% Array scan sector loss
Lscan = zeros(numel(Pd),numel(Theta));

for i = 1:numel(Theta)
    % Assume a single pulse is received from a Swerling 1 target
    Lscan(:,i) = arrayscanloss(Pd,Pfa,1,Theta(i),'Swerling1','CosinePower',3);
end

figure
plot(Pd,Lscan,'LineWidth',2)

xlabel('Detection Probability')
ylabel('Loss (dB)')
title({'Array Scan Sector Loss',sprintf('Swerling 1 Case, P_{fa}=%.0e',Pfa)})

legend(helperLegendLabels('\\theta = %.0f deg',Theta),'location','northwest')
grid on
```
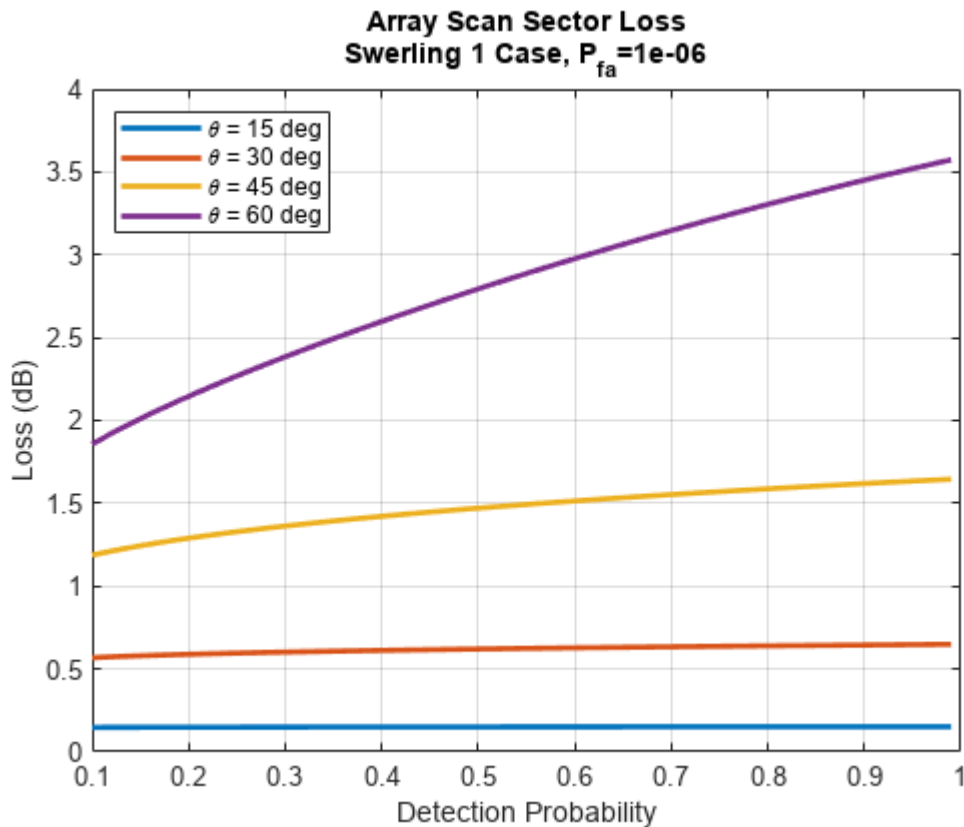

For small scan sectors the statistical scan sector loss is relatively small and does not significantly vary with the probability of detection. As the size of the scan sector increases, the reduction in antenna

gain becomes larger resulting in more rapid increase in the scan sector loss for larger values of the detection probability.

MTI Velocity Response Loss

In moving target indicator (MTI) radar systems, the MTI filter is used to reject clutter components at or near zero Doppler frequency or near the center frequency of the clutter spectrum while passing the target signal spectrum as much as possible. Because the MTI filter is not ideal, it can significantly suppress or even cancel the targets that are close to the null of its frequency response. To illustrate this, the frequency responses of the two, three, and four-pulse MTI cancellers are shown here.

```
m = [2 3 4];
fdnorm = linspace(-0.5,0.5,100);

% Frequency response of an m-pulse MTI canceller
Hmti = (2*sin(pi*fdnorm)) .^ (m'-1);

figure
plot(fdnorm,abs(Hmti),'LineWidth',2)

xlabel('Normalized Doppler Frequency')
ylabel('Frequency Response')
title('2, 3, and 4-pulse Canceller Frequency Response')

legend(helperLegendLabels('m = %d',m),'location','north')
grid on
```
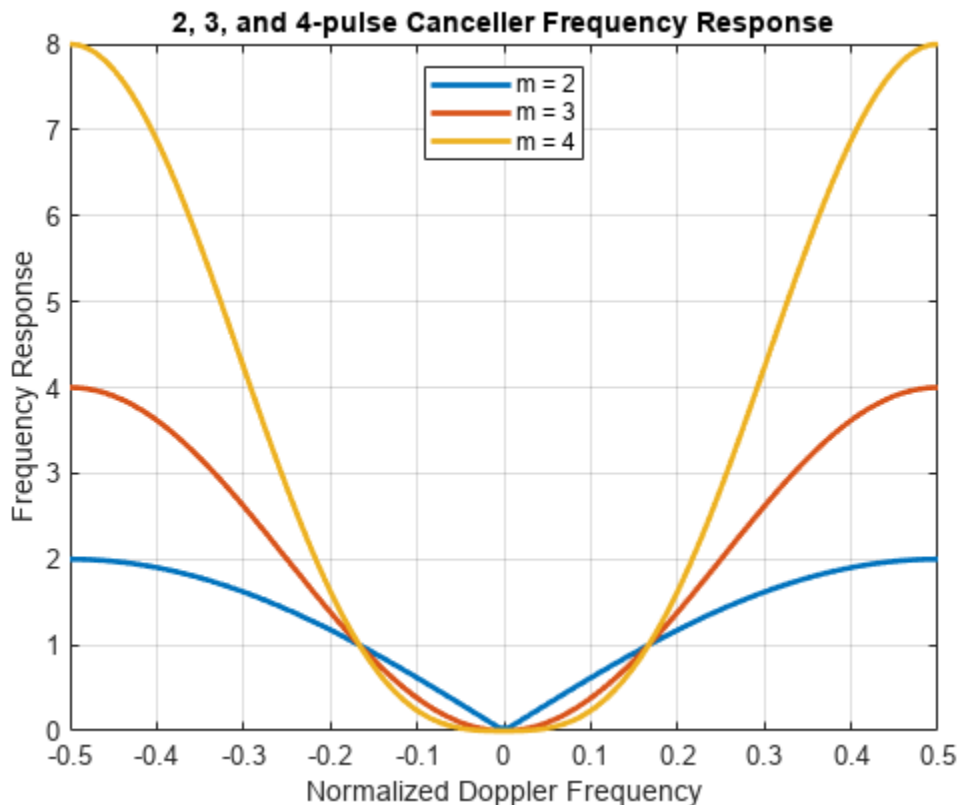
To guarantee that the detection probability averaged over the Doppler frequencies of interest equals the required P_d, the average received signal energy must be increased. This increase compared to the energy required in a system without the MTI (with an all-pass filter) is called a statistical MTI velocity response loss. Plot the MTI velocity response loss as a function of P_d for $P_{fa} = 10^{-6}$.

```
Lmti_velocity_responce = zeros(numel(Pd),numel(m));

for i = 1:numel(m)
    [~,Lmti_velocity_responce(:,i)] = mtiloss(Pd,Pfa,m(i)+1,m(i),'Swerling1');
end

figure
plot(Pd,Lmti_velocity_responce,'LineWidth',2)

xlabel('Detection Probability')
ylabel('Loss (dB)')
title({'MTI Velocity Response Loss',sprintf('Swerling 1 Case, P_{fa}=%.0e',Pfa)})

legend(helperLegendLabels('m = %d',m),'location','northwest')
grid on
ylim([-1 10])
```
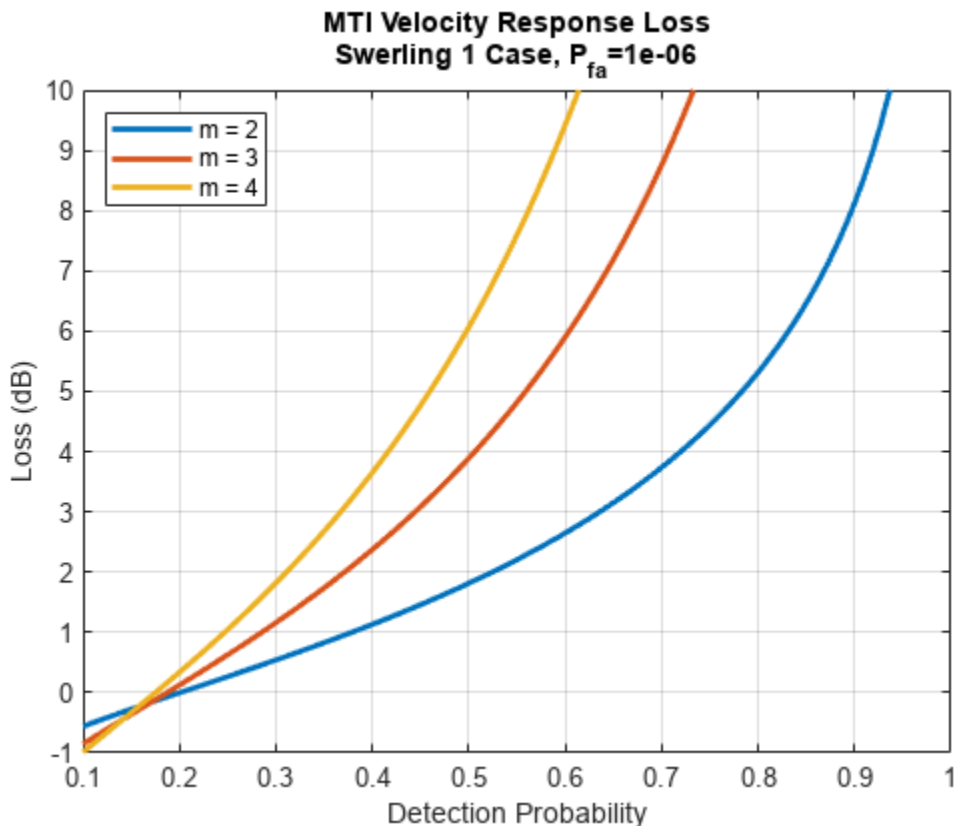

This result shows that in the MTI system with a single PRF the statistical velocity response loss can become very large. The loss grows with the probability of detection. It also increases as the order of the MTI filter increases since the stop band of the filter becomes broader. In practical systems, staggered PRF is used to prevent large values of the MTI velocity response loss.

MTI Integration Loss

In addition to the MTI velocity response loss, the MTI system has another category of losses that must be taken into account. These losses could be viewed as a reduction in the effective number of pulses available for integration after the MTI. Depending on the type of the MTI processing these losses can include [1]:

- MTI noise correlation loss. This loss is a result of the partial correlation introduced by the MTI processing to the received pulses that pass through an *m*-pulse MTI canceller.
- MTI batch processing loss. The result of using a batch processing MTI instead of a sequential MTI. A batch MTI processes *N* received pulses in batches of size *m* resulting in only *N/m* pulses available for integration at the output of the MTI.
- MTI blind phase loss. If only a single channel is available for the MTI processing, the number of available independent target samples is reduced by a factor of two. Additionally, the fluctuation loss also increases. This loss must be added to the noise correlation loss.

To compare different components of the MTI integration loss, consider two, three, and four-pulse MTI cancellers. The probability of detection is set to $P_d = 0.9$ and the probability of false alarm to $P_{fa} = 10^{-6}$.

```matlab
% Number of pulses in MTI canceller
m = [2 3 4];

% Detection probability
Pd = 0.9;

% Probability of false alarm
Pfa = 1e-6;
```

The losses are computed for Swerling 1 case and are plotted as a function of the number of received pulses *N*. Consider a quadrature MTI with sequential processing, a quadrature MTI with batch processing, and a single-channel MTI with sequential processing.

```matlab
% Number of received pulses
N = 1:100;

% MTI noise correlation loss
Lmti_noise_correlation = NaN(numel(m),numel(N));

% MTI blind phase loss
Lmti_blind_phase = NaN(numel(m),numel(N));

% MTI batch processing loss
Lmti_batch_processing = NaN(numel(m),numel(N));

% Compute losses
for i = 1:numel(m)
    for j = 1:numel(N)
        if N(j) > m(i)
            [Lnc,~,Lbp] = mtiloss(Pd,Pfa,N(j),m(i),'Swerling1','IsQuadrature',false);
            Lmti_noise_correlation(i,j) = Lnc;
            Lmti_blind_phase(i,j) = Lbp + Lnc;

            Lnc = mtiloss(Pd,Pfa,N(j),m(i),'Swerling1','Method','batch');
            Lmti_batch_processing(i,j) = Lnc;
```

```matlab
        end
    end
end

% Plot results
figure
for i = 1:numel(m)
    subplot(3,1,i)
    semilogx(N,Lmti_noise_correlation(i,:),'LineWidth',2)
    hold on
    semilogx(N,Lmti_batch_processing(i,:),'LineWidth',2)
    semilogx(N,Lmti_blind_phase(i,:),'LineWidth',2)

    ylabel('Loss (dB)')
    title(sprintf('%d-pulse Canceller',m(i)))

    set(gca(),'xtick',xticks,'xticklabel',num2cell(xticks))
    legend({'Quadrature','Batch','Single-channel'},'location','northeast')
    grid on
    ylim([0 8])
end

xlabel('Number of Received Pulses')
set(gcf,'Position',[100 100 800 600])
```
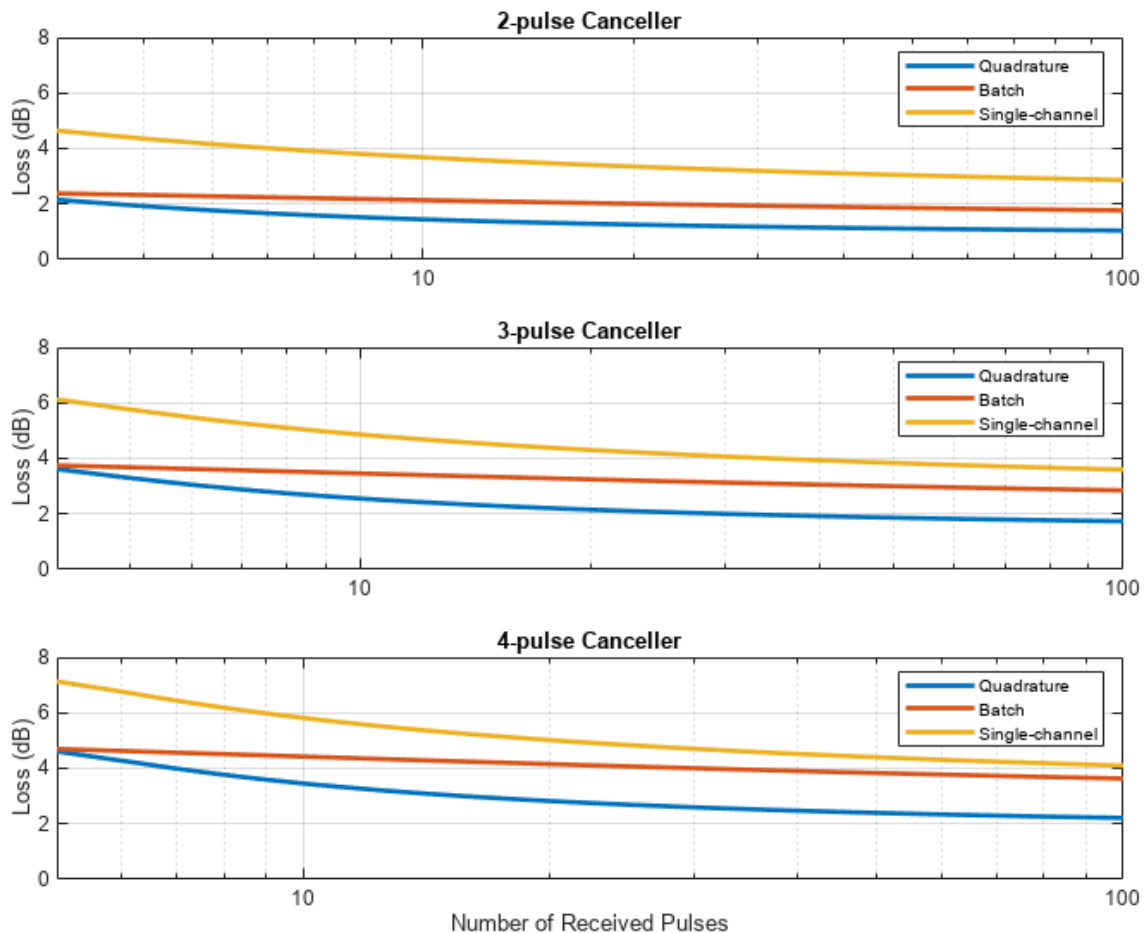
From these results we can see that the quadrature MTI with sequential processing is the most efficient. The noise correlation loss decreases as more pulses become available for integration. For the batch processing MTI, the loss is close to the sequential case when the number of received pulses is small. It also decreases with N but not as fast. Finally, the blind phase loss due to a single-channel MTI is the largest because in this case the number of available samples is further decreased by two in addition to the noise correlation loss.

CFAR Loss

CFAR is used to estimate the value of the detection threshold when the levels of the noise or clutter are variable. In practice, however, this estimate is subject to an error due to the finite number of reference cells, n_{cells}, and the rapid changes in the interference levels. To compensate for this error a higher received signal energy is needed. The increase in the signal energy required to achieve the desired detection performance using CFAR compared to a system with a perfectly known detection threshold is called a CFAR loss. For a one-dimensional case, a convenient approximation for CFAR loss in cell-averaging (CA) and greatest-of cell-averaging (GOCA) CFAR was developed in [3] and described in [1].

```
% Number of CFAR reference cells
numCells = 4:4:64;
```

```
% Probability of false alarm
Pfa = 1e-8;

% CFAR ratio
x = -log10(Pfa)./numCells;

% Compute CFAR loss
Lcfar_ca = cfarloss(Pfa,numCells);
Lcfar_goca = cfarloss(Pfa,numCells,'Method','GOCA');

% Plot
figure
hold on
plot(x,Lcfar_ca,'LineWidth',2)
plot(x,Lcfar_goca,'LineWidth',2)

xlabel('-log10(P_{fa})/n_{cells}')
ylabel('Loss (dB)')
title({'Universal Curve for CFAR Loss'})

legend({'Cell-averaging','Greatest-of cell-averaging'},'location','northwest')
grid on
```
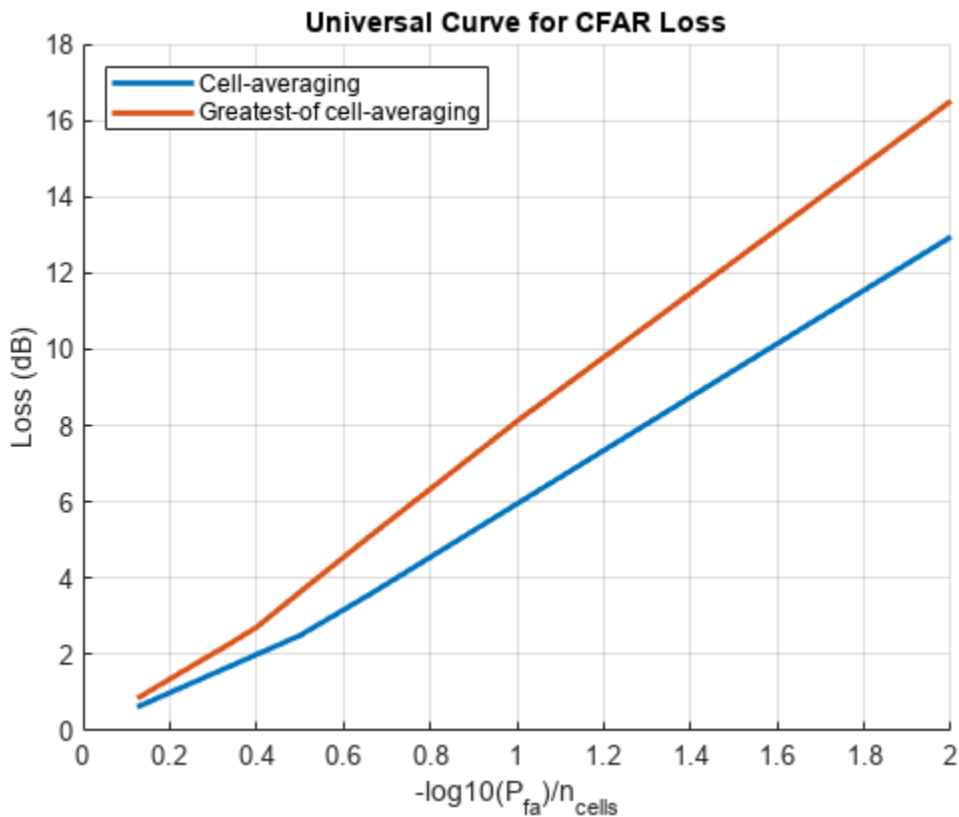

For a given CFAR method, the approximated loss depends only on the ratio $-log_{10}(P_{fa})/n_{cells}$, called a CFAR ratio. This allows for finding a trade-off between the required probability of false alarm and the number of reference cells to achieve a desired level of CFAR loss. The approximation shows that

CFAR loss increases with the CFAR ratio. The loss of about 2 dB or smaller can be achieved if the CFAR ratio is kept under 0.4 for both CA and GOCA CFAR methods. If n_{cells} is kept constant, reducing P_{fa} would result in an increased loss. Thus, to achieve a lower required probability of false alarm while keeping the loss small, the number of reference cells must be increased. These loss curves are referred to as universal curves for CFAR loss since they can be used both for nonfluctuating and Rayleigh targets and need only P_{fa} and n_{cells} to compute the loss. This makes it easy to use these results when analyzing a wide variety of radar systems [1].

Matching Loss

When the spectrum of the received signal is different from the spectrum of the matched filter, the system will incur a matching loss. This loss can be defined as a ratio of the received output SNR to the SNR available from a filter perfectly matched to the received signal. This example considers several filter types: rectangular, Sinc, Gaussian, and single-pole. For ease of comparison, the filter bandwidth is assumed to be equal to one. The shapes of these filters are shown in the following figure.

```matlab
% Normalized bandwidth of the matched filter
B = 1;

% Max normalized pulse duration
taumax = 10;

% Sampling period
dt = 0.01;

% Number of frequency bins
nf = 2 ^ (nextpow2(taumax/dt) + 1);
m = -nf/2:nf/2 - 1;

% Normalized frequency
f = m / (nf*dt);
H = zeros(4,numel(f));

% Frequency response of a rectangular filter
H(1,abs(f) <= B/2) = 1;

% Frequency response of a sinc filter
H(2,:) = sinc(f/B);

% Frequency response of a Gaussian filter
H(3,:) = exp(-(pi/2) * (f/B).^2);

% Frequency response of a single-pole filter
H(4,:) = 1./(1 + (pi*f / (2*B)).^2);

figure
plot(f,H,'LineWidth',2)

xlabel('Frequency')
ylabel('H(f)')
title('Filter Frequency Responses')

labels = {'Rectangular filter','Sinc filter','Gaussian filter','Single-pole filter'};
legend(labels)
```

```
grid on;
xlim([-10 10])
```

Filter Frequency Responses
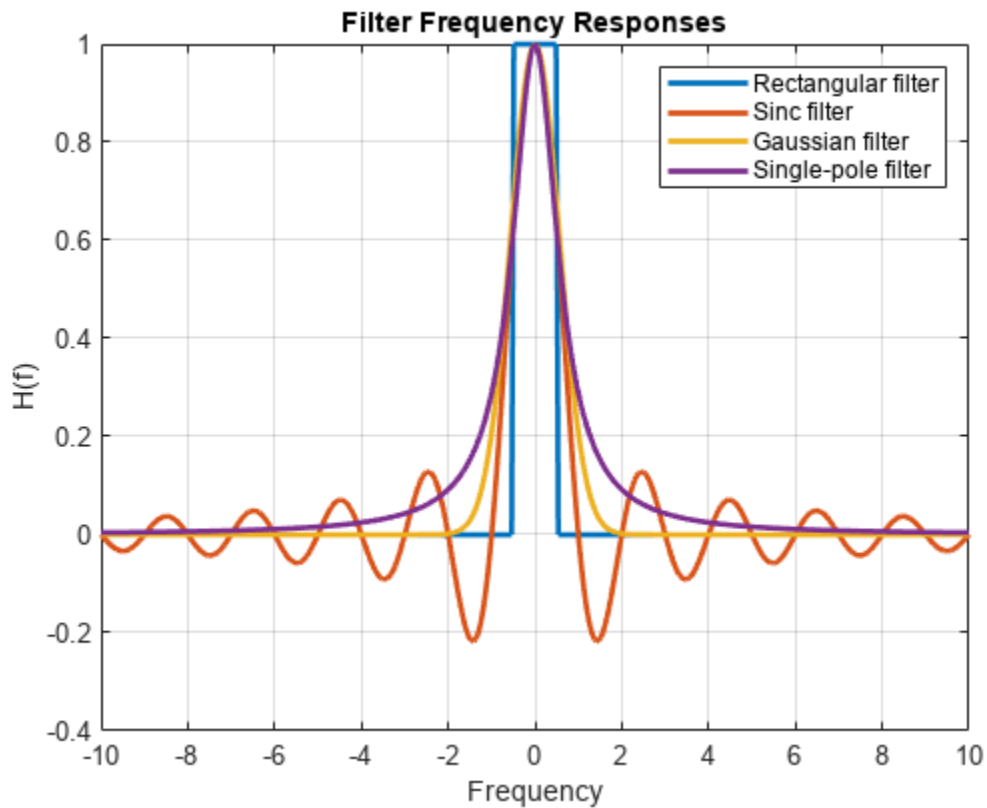
The received signal in this example is an unmodulated ideal rectangular pulse. The filter matching loss for the four filter types is shown here as a function of the time-bandwidth product.

```
% Pulse duration normalized by (1/B)
tau = 0.1:0.1:taumax;

% Spectrum of an ideal pulse with no phase modulation
N = ceil(tau.'/dt);
S = sin(pi*N*m/nf) ./ sin(pi*m/nf);
S(:,nf/2 + 1) = N;

% Matching loss
L = matchingloss(S,H);

figure
semilogx(tau*B,L,'LineWidth',2)

xlabel('Time-Bandwidth Product')
ylabel('Loss (dB)')
title({'Matching Loss for an Unmodulated Rectangular Pulse'})

legend(labels,'location','North')
xticks = [0.1 0.2 0.5 1 2 5 10];
set(gca(),'xtick',xticks,'xticklabel',num2cell(xticks))
grid on
```

Matching Loss for an Unmodulated Rectangular Pulse
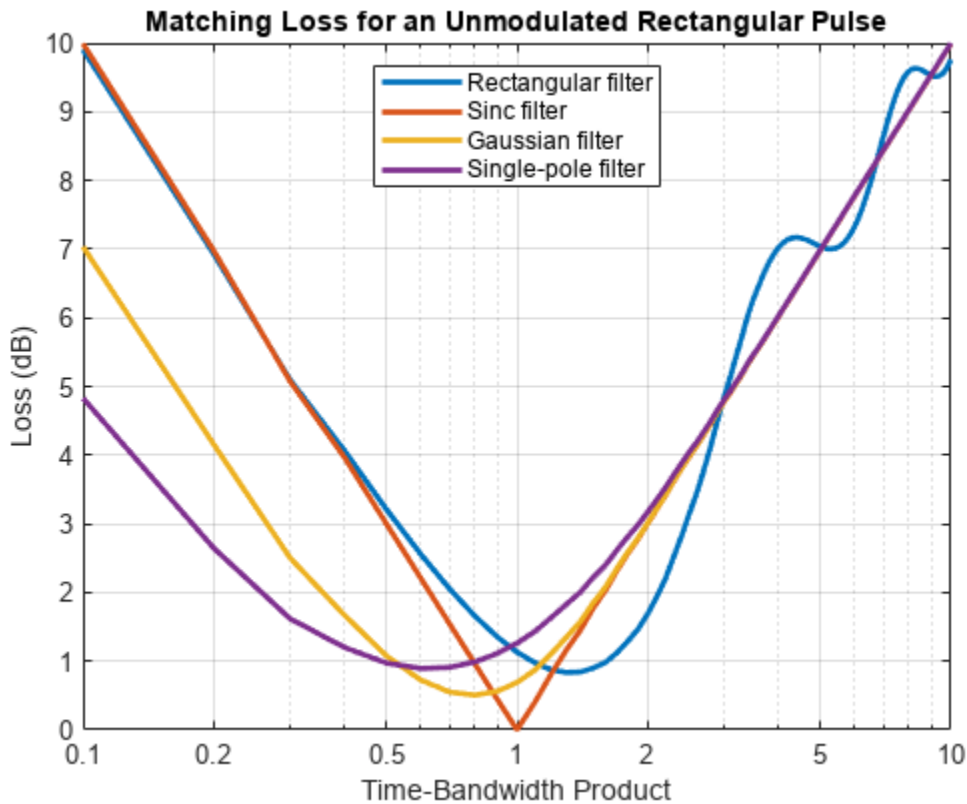
As expected, the filter with the Sinc spectrum has the lowest matching loss because it ideally matches the spectrum of the unmodulated rectangular pulse. The Gaussian filter that can represent a cascade of several filters, results in 0.5 dB matching loss when the time-bandwidth product is close to 0.8.

Conclusion

This example discusses several losses that must be included in the effective detectability factor when evaluating the radar range equation. It starts by considering a category of statistical losses. These losses are incurred by a radar system because of a dependance of the effective detection probability on the target position in the range-Doppler-angle space. Evaluation of statistical losses due to the eclipsing effect, the off-broadside electronic scanning, and a nonideal MTI filter is shown in this example. It is also shown that these losses increase rapidly with the probability of detection. The example further explores losses in an MTI system by considering different components of the MTI integration loss and its dependence on the order of the MTI pulse canceller. The example also illustrates a convenient way to assess the CFAR loss based on the universal curve for CFAR loss. Finally, it shows the computation of the filter matching loss.

References

1   Barton, David Knox. *Radar Equations for Modern Radar*. Artech House, 2013.

2   Richards, M. A. *Fundamentals of Radar Signal Processing*. Second edition, McGraw-Hill Education, 2014.

3   Gregers-Hansen, V., "Constant false alarm rate processing in search radars." In *IEE Conf. Publ. no. 105,*" *Radar-Present and Future*", pp. 325-332. 1973.

Modeling Target Position Errors Due to Refraction

A radar system does not operate in isolation. The performance of a radar system is closely tied to the environment in which it operates. This example will discuss some of the environmental factors that are outside of the system designer's control. These environmental factors can result in losses, as well as errors in target parameter estimation.

This example first reviews several atmospheric models. Next, it presents an approximation of the standard atmospheric model using a simple exponential decay versus height. It then discusses these models within the context of a maximum range assessment and visualization of the refracted path. Lastly, it ends with a discussion on how the atmosphere causes errors in determining target height, slant range, and angle.

Atmospheric Models

Calculating tropospheric losses and the refraction phenomenon requires models of the atmospheric temperature, pressure, and water vapor density, which are dependent on height. The function `atmositu` offers six ITU reference atmospheric models, namely:

- Standard atmospheric model, also known as the Mean Annual Global Reference Atmosphere (MAGRA)
- Summer high latitude model (higher than 45 degrees)
- Winter high latitude model (higher than 45 degrees)
- Summer mid latitude model (between 22 and 45 degrees)
- Winter mid latitude model (between 22 and 45 degrees)
- Low latitude model (smaller than 22 degrees)

Note that for low latitudes, seasonal variations are not significant. Plot the temperature, pressure, and water vapor density profiles over a range of heights from the surface to 100 km for the default standard atmospheric model.

```
% Standard atmosphere model
hKm = 0:100;
h = hKm.*1e3;
[T,P,wvden] = atmositu(h);

% Plot temperature profile
figure
subplot(1,3,1)
plot(T,hKm,'LineWidth',2)
grid on
xlabel('Temperature (K)')
ylabel('Altitude (km)')

% Plot pressure profile
subplot(1,3,2)
semilogx(P,hKm,'LineWidth',2)
grid on
xlabel('Pressure (hPa)')
ylabel('Altitude (km)')

% Plot water vapor density profile
subplot(1,3,3)
```

```
semilogx(wvden,hKm,'LineWidth',2)
grid on
xlabel('Water Vapor Density (g/m^3)')
ylabel('Altitude (km)')

% Add title
sgtitle('Standard: ITU Reference Atmosphere')
```


Standard: ITU Reference Atmosphere
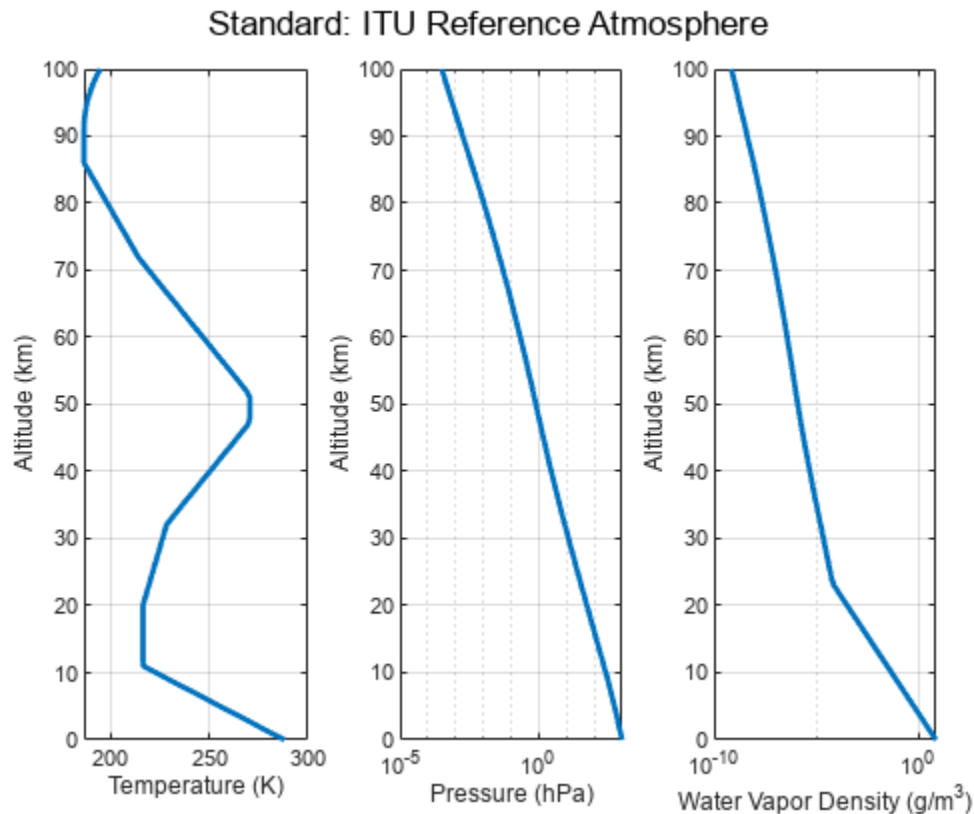
Once you have the temperature T(K), pressure P (hPa), and water vapor density ρ (g/m3) profiles, convert these profiles to refractivity values N as

$$N = 77.6\frac{P}{T} - 5.6\frac{e}{T} + 3.75 \times 10^5\frac{e}{T^2},$$

where e is the water vapor pressure (hPa) calculated as

$$e = \frac{\rho T}{216.7}.$$

The refractivity profile for the standard atmosphere is shown here and is obtained from the function `refractiveidx`. `refractiveidx` allows you to specify one of the ITU reference models listed earlier or import custom temperature, pressure, and water vapor density profiles.

```
% Standard atmosphere model refractivity
[refidxStandard,refractivityStandard] = refractiveidx(h);

% Plot standard atmosphere refractivity profile
```

```
figure
semilogy(hKm,refractivityStandard,'LineWidth',2)
grid on
xlabel('Height (km)')
ylabel('Refractivity (N-units)')
title('Standard: Refractivity versus Height')
```
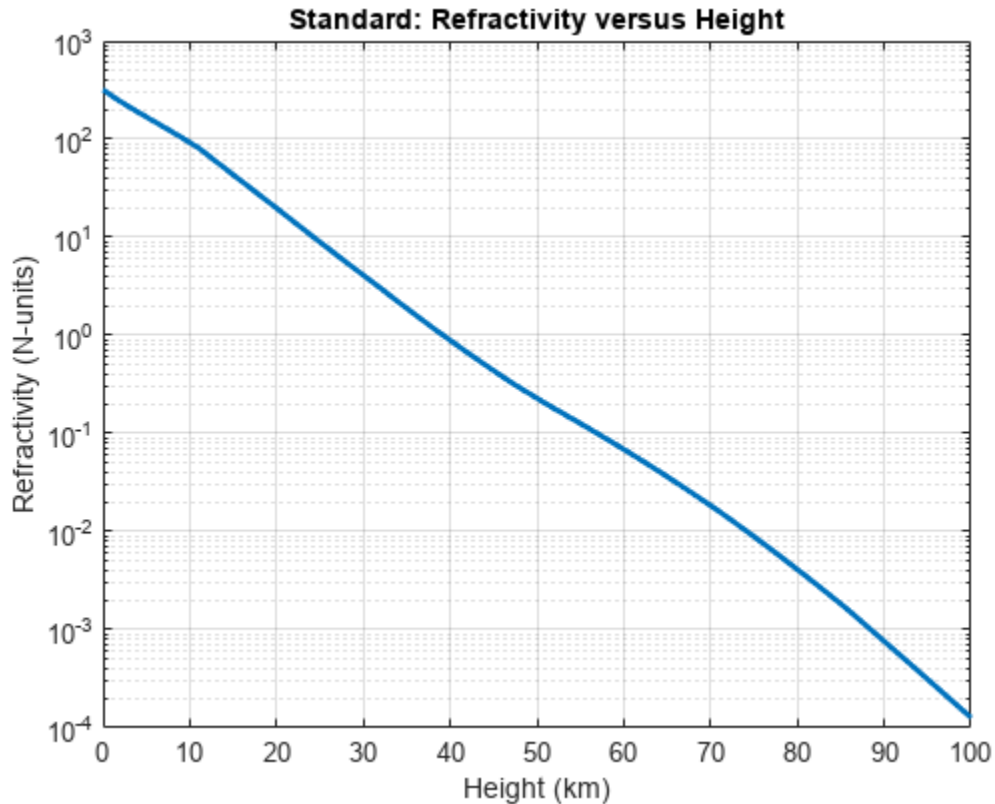

Exponential Approximation

The Central Radio Propagation Laboratory (CRPL) developed a widely used reference atmosphere model that approximates the refractivity profile as an exponential decay versus height. This exponential decay is modeled as

$$N = N_S \exp(-c_e h),$$

where N_S is the surface refractivity, h is the height, and c_e is a decay constant calculated as

$$c_e = \ln\frac{N_S}{N_S - \Delta N}.$$

In the above equation, ΔN is the difference between the surface refractivity N_S and the refractivity at a height of 1 km. ΔN can be approximated by an exponential function expressed as

$$\Delta N = 7.32 \exp(0.005577 N_S).$$

Assume the surface refractivity is 313. Calculate the decay constant using the `refractionexp` function. Compare the CRPL exponential model with the ITU standard atmosphere model. Note how well the two models match.

```matlab
% CRPL reference atmosphere
Ns = 313; % Surface refractivity (N-units)
rexp = refractionexp(Ns); % 1/km
refractivityCRPL = Ns*exp(-rexp*hKm);

% Compare ITU and CRPL reference atmosphere models
figure
semilogy(hKm,refractivityStandard,'LineWidth',2)
grid on
hold on
semilogy(hKm,refractivityCRPL,'--r','LineWidth',2)
legend('Standard','CRPL')
xlabel('Height (km)')
ylabel('Refractivity (N-units)')
title('Refractivity versus Height')
```
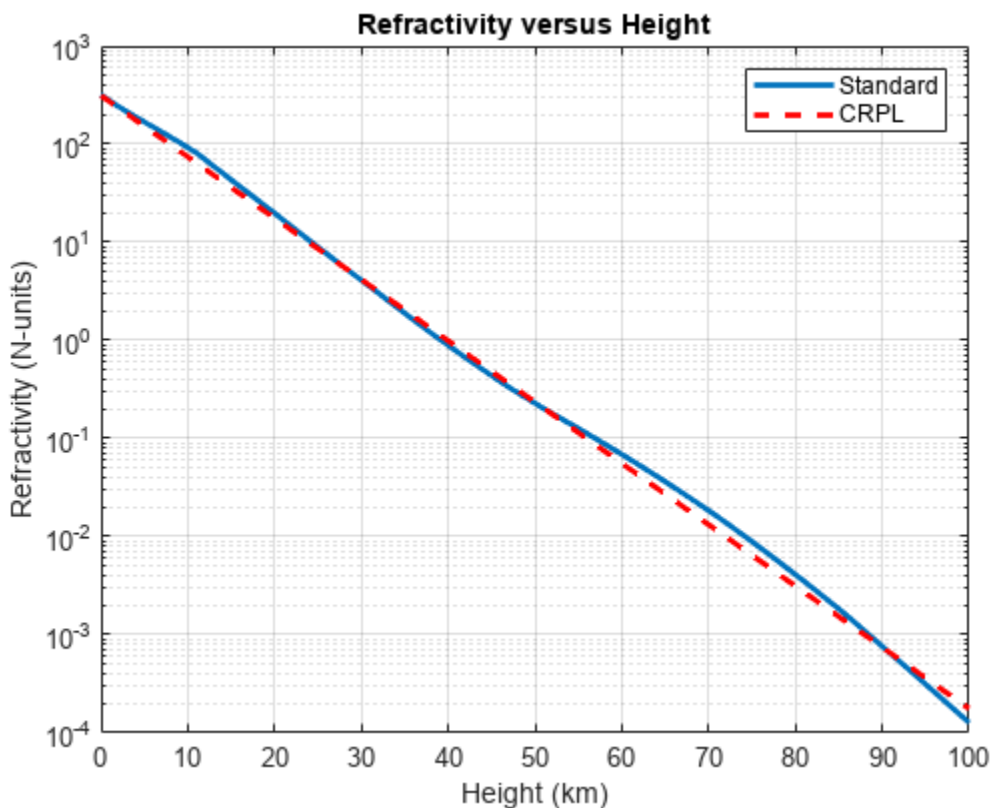

The CRPL reference exponential model forms the basis for the `blakechart` function, as well as its supporting functions `height2range`, `height2grndrange`, and `range2height`.

Vertical Coverage

A vertical coverage pattern, also known as a Blake chart or range-height-angle chart, is a detection or constant-signal-level contour that is a visualization of refraction and the interference between the

direct and ground-reflected rays. Normal atmospheric refraction is included by using an effective Earth radius and the axes of the Blake chart are built using the CRPL exponential reference atmosphere. Scattering and ducting are assumed to be negligible. The propagated range exists along the x-axis, and the height relative to the origin of the ray is along the y-axis.

To create the Blake chart, compute an appropriate free space range. Consider the case of an X-band radar system operating in an urban environment.

```
% Radar parameters
freq = 10e9;        % X-band frequency (Hz)
anht = 20;          % Height (m)
ppow = 100e3;       % Peak power (W)
tau  = 200e-6;      % Pulse width (sec)
elbw = 10;          % Half-power elevation beamwidth (deg)
Gt   = 30;          % Transmit gain (dB)
Gr   = 20;          % Receive gain (dB)
nf   = 2;           % Noise figure (dB)
Ts   = systemp(nf); % System temperature (K)
L    = 1;           % System losses
el0  = 0.2;         % Initial elevation angle (deg)
N    = 10;          % Number of pulses coherently integrated
pd   = 0.9;         % Probability of detection
pfa  = 1e-6;        % Probability of false alarm

% Surface
[hgtsd,beta0] = landroughness('Urban');
epsc = 5.3100 - 0.3779i; % Permittivity for concrete at 10 GHz
```

Assume a Swerling 1 target with a 1 m^2 radar cross section (RCS). Determine the minimum detectable signal-to-noise ratio (SNR) using the `detectability` function, and then calculate the maximum detectable range using the `radareqrng` function. The calculated range will be used as the free space range input for `radarvcd`.

```
% Calculate coherent integration gain
Gc = pow2db(N);

% Calculate minimum detectable SNR
minsnr = detectability(pd,pfa,10,'Swerling1') - Gc;
fprintf('Minimum detectable SNR = %.1f dB\n',minsnr);

Minimum detectable SNR = 3.5 dB

% Calculate the maximum free space range
lambda = freq2wavelen(freq);
rcs = 1;
maxRngKm = radareqrng(lambda,minsnr,ppow,tau, ...
        'gain',[Gt Gr],'rcs',rcs,'Ts',Ts,'Loss',L)*1e-3;
fprintf('Maximum detectable range in free space = %.1f km\n',maxRngKm);

Maximum detectable range in free space = 84.3 km
```

Use the `radarvcd` and `blakechart` functions in conjunction to visualize the vertical coverage of the radar system in the presence of refraction and interference between the direct and ground reflected rays. The Blake chart's constant signal level in this case is the previously calculated minimum SNR. The lobing that is seen in the following figure is the interference pattern created from the interaction of the direct and ground-reflected rays.

```
% Obtain the vertical coverage contour
[vcpkm,vcpang] = radarvcd(freq,maxRngKm,anht, ...
```

```
    'SurfaceHeightStandardDeviation',hgtsd,...
    'SurfaceSlope',beta0,...
    'SurfaceRelativePermittivity',epsc,...
    'TiltAngle',el0,...
    'ElevationBeamwidth',elbw);

% Plot the vertical coverage contour on a Blake chart
figure
blakechart(vcpkm,vcpang)
```
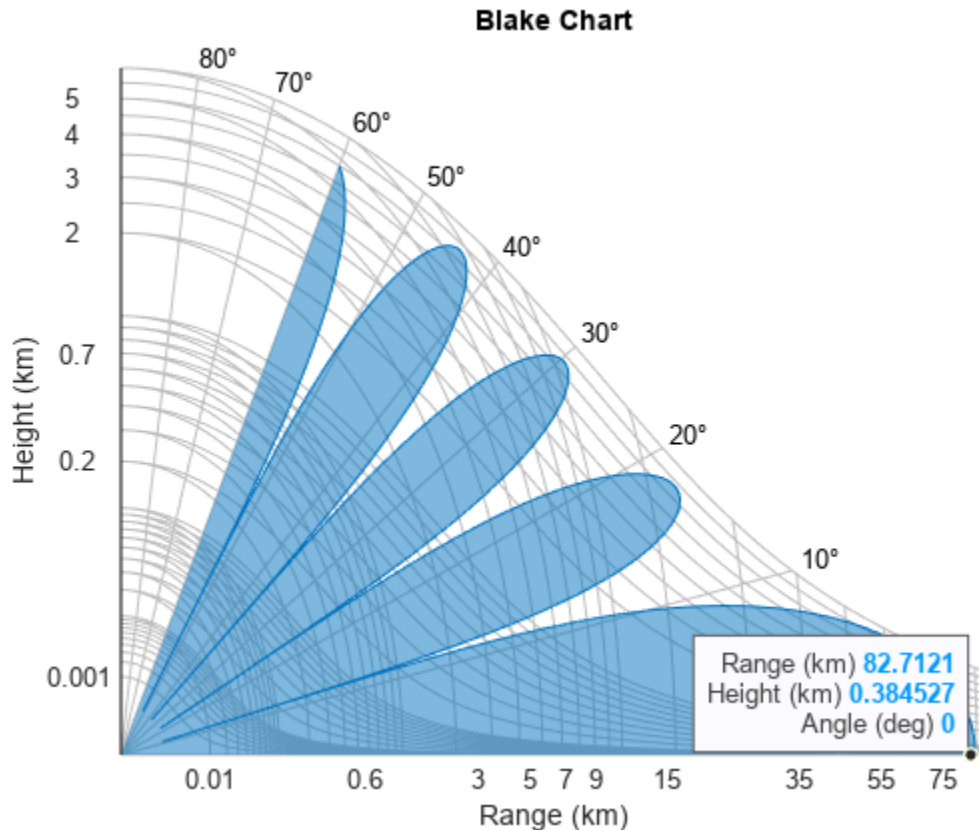

Understanding Different Atmospheric Assumptions

The following section shows you how to estimate the target height under the following assumptions:

- No refraction
- Effective Earth approximation
- CRPL reference atmosphere

No Refraction

In this case, the Earth's radius is the true radius. The refractive index in this case would be equal to 1. This case presents an upper bound on the estimated target height.

```
% Target height assuming no refraction
Rm = linspace(0.1,maxRngKm,1000)*1e3;
Rkm = Rm*1e-3;
Rearth = physconst('EarthRadius'); % Earth radius (m)
```

```
tgthtNoRef = range2height(Rm,anht,el0,'EffectiveEarthRadius',Rearth);
fprintf('Target height for no refraction case %.1f m\n',tgthtNoRef(end));
```

Target height for no refraction case 872.7 m

Effective Earth

The effective Earth model approximates the refraction phenomena by performing calculations with an effective radius rather than the true radius. Typically, the effective Earth radius is set to 4/3 the true Earth radius.

The 4/3 Earth model suffers from two major shortcomings:

- The 4/3 Earth value is only applicable for certain areas and at certain times of the year.
- The gradient of the refractive index implied by the 4/3 Earth model is nearly constant and decreases with height at a uniform rate. Thus, values can reach unrealistically low values.

To improve the accuracy of the effective Earth calculations, you can avoid the aforementioned pitfalls by setting the effective Earth radius to a value that is more consistent with the local atmospheric conditions. For instance, if a location has a typical gradient of refractivity of −100 N-units/km, the effective Earth's radius factor is about 11/4 and the radius is 17,500 km.

The `effearthradius` function facilitates the calculation of the effective Earth radius and the corresponding fractional factor using two methods based on either the:

- Refractivity gradient, or
- An average radius of curvature calculation that considers the range, the antenna height, and the target height.

All functions that accept an effective Earth radius input can be updated with the `effearthradius` output to be more consistent with local conditions or the target geometry. For this example, continue with the standard 4/3 Earth.

```
tgthtEffEarth = range2height(Rm,anht,el0);
fprintf('Target height for 4/3 effective Earth case %.1f m\n',tgthtEffEarth(end));
```

Target height for 4/3 effective Earth case 734.0 m

CRPL Reference Atmosphere

Lastly is the CRPL reference atmosphere. This is a refractivity profile that approximates the atmosphere with a decaying exponential.

```
tgthtCRPL = range2height(Rm,anht,el0,'Method','CRPL');
fprintf('Target height for CRPL case %.1f m\n',tgthtCRPL(end));
```

Target height for CRPL case 716.4 m

As was seen, the resulting target heights can vary greatly depending on the atmospheric assumptions.

Assuming the CRPL atmosphere is a close representation of the true atmosphere, the errors between the effective Earth radius and the CRPL method are as follows.

```
% Calculate true slant range and true elevation angle
[~,trueSR,trueEl] = height2range(tgthtCRPL,anht,el0,'Method','CRPL');
```

```
% Display errors
fprintf('Target height error = %.4f m\n',tgthtEffEarth(end) - tgthtCRPL(end));
```

Target height error = 17.5602 m

```
fprintf('Target slant range error = %.4f m\n',Rm(end) - trueSR(end));
```

Target slant range error = 25.3316 m

```
fprintf('Target angle error = %.4f deg\n',el0 - trueEl(end));
```

Target angle error = 0.1059 deg

Visualizing Refraction

Next, visualize the refraction geometry including the:

- Initial ray: This is the ray as it leaves the antenna at the initial elevation angle. This would be the path of the ray if refraction was not present.
- Refracted ray: The refracted ray is the actual path of the ray that bends as it propagates through the atmosphere.
- True slant range: This is the true slant range from the antenna to the target.
- Horizontal: This is the horizontal line at the origin, coming from the antenna.

```
% Plot initial ray
anhtKm = anht*1e-3;
[Xir,Yir] = pol2cart(deg2rad(el0),Rkm);
Yir = Yir + anhtKm;
figure
plot(Xir,Yir,'-.k','LineWidth',1)
grid on
hold on

% Plot refracted ray
[X,Y] = pol2cart(deg2rad(trueEl),trueSR*1e-3);
Y = Y + anhtKm;
co = colororder;
plot(X,Y,'Color',co(1,:),'LineWidth',2)

% Plot true slant range
X = [0 X(end)];
Y = [anhtKm Y(end)];
plot(X,Y,'Color',co(1,:),'LineStyle','--','LineWidth',2)

% Plot horizontal line
yline(anhtKm,'Color','k','LineWidth',1,'LineStyle','-')

% Add labels
title('Refraction Geometry')
xlabel('X (km)')
ylabel('Y (km)')
legend('Initial Ray','Refracted Ray','True Slant Range', ...
    'Horizontal at Origin','Location','Best')
```
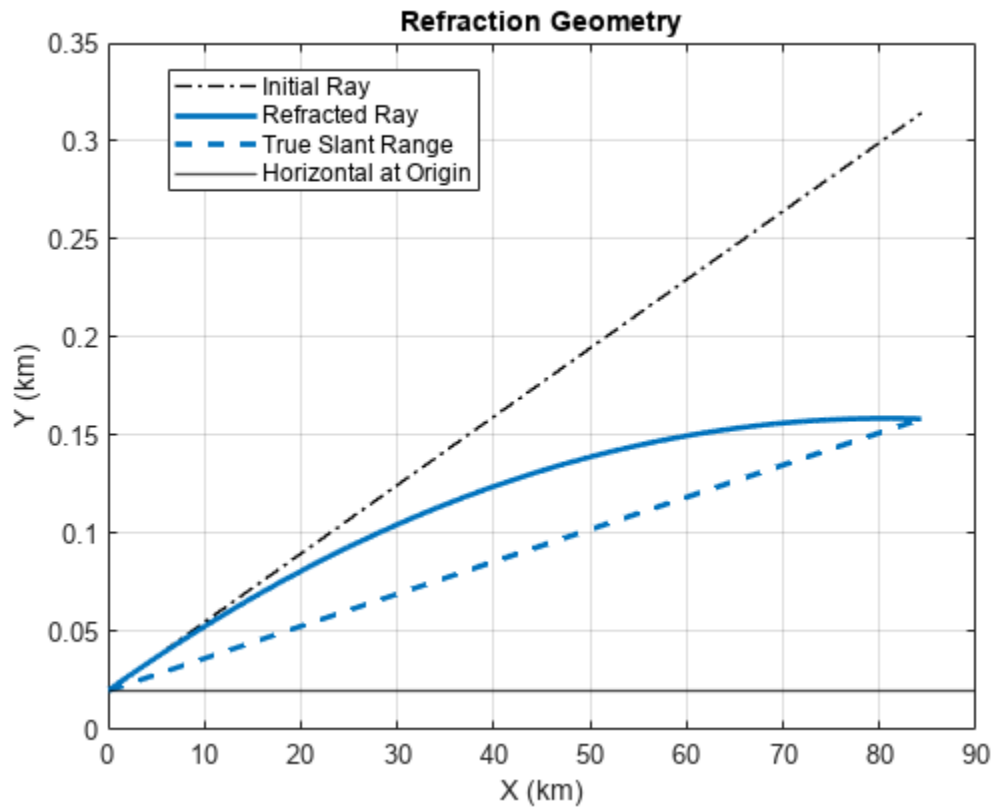
Target Parameter Estimation Errors

Continuing further with concepts investigated in the previous section, this section will investigate target parameter estimation errors within the context of a ground-based radar system. As was done previously, proceed from the assumption that the CRPL atmosphere is representative of the true atmosphere. Calculate the errors in height, slant range, and angle when using the 4/3 effective Earth approximation.

```
% Analysis parameters
tgthtTrue = 1e3;      % True target height (m)
el        = 90:-1:1;  % Initial elevation angle (degrees)
anht      = 10;       % Antenna height (m)
```

First, calculate the propagated range, R. This is the actual refracted path of the ray through the atmosphere. The radar system would assume that the R values are the true straight-line ranges to the target and that the initial elevation angles `el` are the true elevation angles. However, the true slant range and elevation angles are given in the outputs `trueSR` and `trueEl`.

```
% Calculate the propagated range
[R,trueSR,trueEl] = height2range(tgthtTrue,anht,el,'Method','CRPL');
```

Next, calculate the target height under the 4/3 effective Earth radius. Plot the errors.

```
% Calculate target height assuming 4/3 effective Earth radius
tgtht = range2height(R,anht,el);
```

```
% This is the difference between the target height under an assumption of a
```

```
% 4/3 Earth versus what the target height truly is.
figure
subplot(3,1,1);
plot(el,tgtht - tgthtTrue,'LineWidth',1.5)
grid on
xlabel('Initial Elevation Angle (deg)')
ylabel(sprintf('Height\nError (m)'))

% This is the difference between the propagated range (what the radar
% detects as the range) versus what the actual true slant range is.
subplot(3,1,2)
plot(el,(R - trueSR),'LineWidth',1.5)
grid on
xlabel('Initial Elevation Angle (deg)')
ylabel(sprintf('Range\nError (m)'))

% This is the difference between the local elevation angle (i.e., the radar
% transmitter angle) and the actual angle to the target.
subplot(3,1,3)
plot(el,el - trueEl,'LineWidth',1.5)
grid on
xlabel('Initial Elevation Angle (deg)')
ylabel(sprintf('Angle\nError (deg)'))
sgtitle('Errors')
```
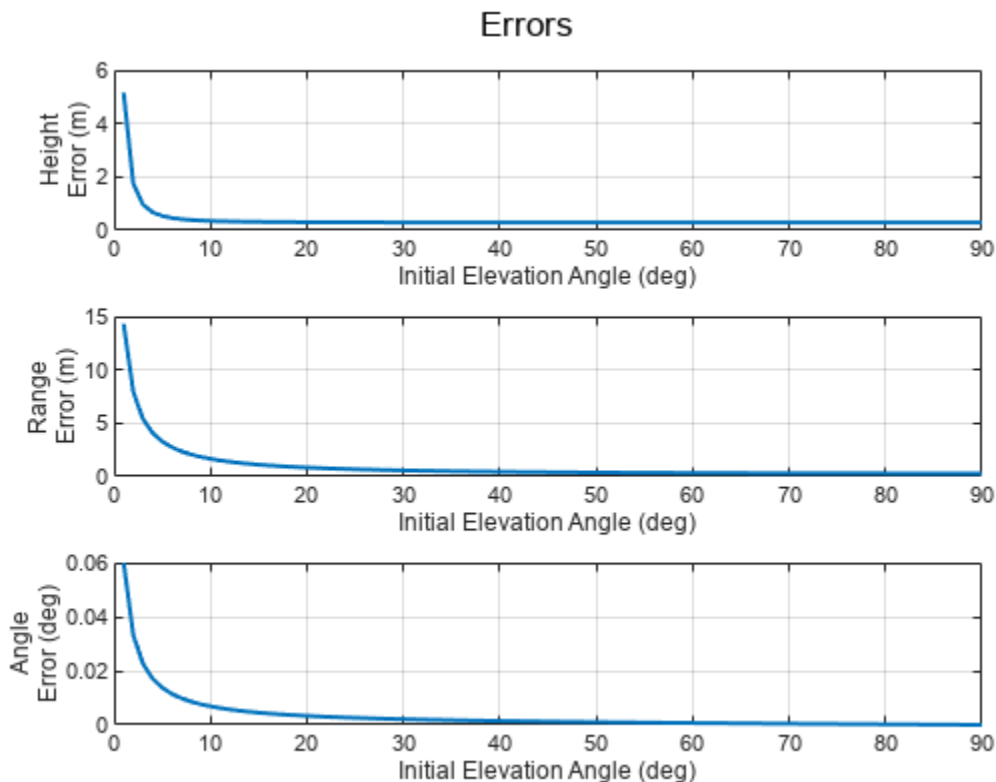
Note that the errors are at their worst when the initial elevation angle is small. In this case, however, since the heights of the platforms are still rather low, the effective Earth radius is a good approximation when the elevation angles are larger.

Investigating General Error Trends Due to Refraction

Calculate the height errors for different slant ranges and target heights. Consider target heights in the range from 1 m to 30 km and slant ranges from 1 m to 100 km. Convert the slant ranges to propagated range using the `slant2range` function. Continue to use the default CRPL parameters. After the propagated range and initial elevation angle of the propagated path is calculated, use the `range2height` function with the true Earth radius to calculate the apparent target heights given the initial elevation angle.

```matlab
% Inputs
numPts  = 50;                             % Number of points
tgtHgts = linspace(1e-3,30,numPts)*1e3;   % Target heights (m)
tgtSR   = linspace(1e-3,100,numPts)*1e3;  % Range to target (m)
Rearth  = physconst('EarthRadius');       % Earth radius (m)
anht    = 10;                             % Antenna height (m)

% Initialize calculation
tgtPropPath = nan(numPts,numPts);         % Propagated target path (m)
tgtEl = nan(numPts,numPts);               % Target elevation angles (deg)
tgtHgtsApparent = nan(numPts,numPts);     % Apparent target heights (m)
k = ones(numPts,numPts);                  % Effective Earth k

% Estimate height errors using slant2range and range2height
[tgtSRGrid,tgtHgtsGrid] = meshgrid(tgtSR,tgtHgts);
idx = tgtSRGrid(:) >= abs(anht - tgtHgtsGrid(:));
[tgtPropPath(idx),tgtEl(idx),k(idx)] = slant2range(tgtSRGrid(idx),anht,tgtHgtsGrid(idx), ...
    'Method','CRPL');
tgtHgtsApparent(k~=1) = range2height(tgtPropPath(k~=1),anht,tgtEl(k~=1), ...
    'Method','Curved','EffectiveEarthRadius',Rearth);
tgtHgtsErr = tgtHgtsApparent - tgtHgtsGrid;
```

Draw contour lines to display the height error as a function of true target height and slant range. The plot indicates that at the farthest ranges and the lower target altitudes, height errors up to about 200 meters can be seen.

```matlab
% Plot height error contours
figure('Name','Height Error Contours')
tgthgtErrorLevels = [2 5 10 20 30 50:25:300]; % Contour levels (m)
[~,hCon1] = contour(tgtSR*1e-3,tgtHgts*1e-3, ...
    tgtHgtsErr,tgthgtErrorLevels,'ShowText','on');
hCon1.LineWidth = 1.5;
grid on
xlabel('Slant Range (km)')
ylabel('True Target Height (km)')
hCb = colorbar;
hCb.Label.String = 'Height Errors (m)';
axis equal
title('Height Error Contours (m)')
```

Height Error Contours (m)
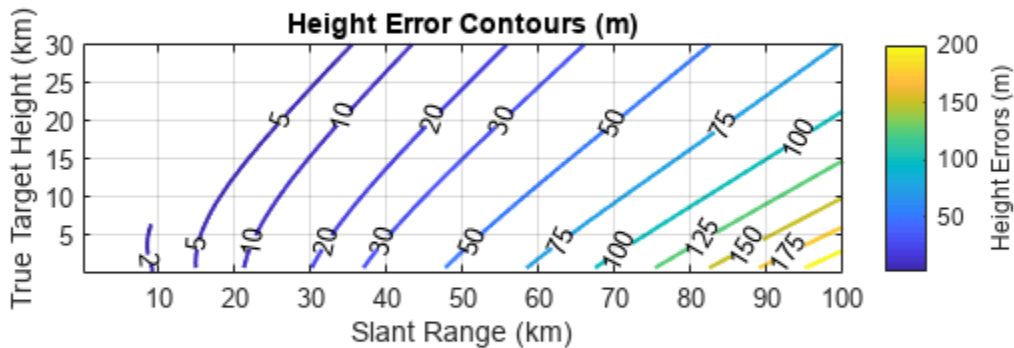
Summary

This example discussed atmospheric models and investigated their effect on target range, height, and angle estimation.

References

1 Barton, David K. *Radar Equations for Modern Radar*. Norwood, MA: Artech House, 2013.

2 Blake, L.V. "Radio Ray (Radar) Range-Height-Angle Charts." Naval Research Laboratory, NRL Report 6650, Jan. 22, 1968.

3 Blake, L.V. "Ray Height Computation for a Continuous Nonlinear Atmospheric Refractive-Index Profile." *RADIO SCIENCE*, Vol. 3 (New Series), No. 1, Jan. 1968, pp. 85-92.

4 Bean, B. R., and G. D. Thayer. *CRPL Exponential Reference Atmosphere*. Washington, DC: US Gov. Print. Off., 1959.

5 International Telecommunication Union (ITU). "Attenuation by Atmospheric Gases". Recommendation ITU-R P.676-12, P Series, *Radiowave Propagation*, August 2019.

6 Weil, T. A. "Atmospheric Lens Effect: Another Loss for the Radar Range Equation." *IEEE Transactions on Aerospace and Electronic Systems*, Vol. AES-9, No. 1, Jan. 1973.

See Also

atmositu | blakechart | detectability | height2range | landroughness | radareqrng | radarvcd | range2height | refractionexp | refractiveidx

Related Examples

- "Radar Vertical Coverage over Terrain" on page 1-617

Modeling the Propagation of Radar Signals

This example shows how to model several RF propagation effects. These include free space path loss, atmospheric attenuation due to rain, fog and gas, and multipath propagation due to bounces on the ground. The discussion in this example is based on the ITU-R P series recommendations of the International Telecommunication Union. ITU-R is the radio communication section and the P series focuses on radio wave propagation.

Introduction

To properly evaluate the performance of radar and wireless communication systems, it is critical to understand the propagation environment. The received signal power of a monostatic radar is given by the radar range equation:

$$P_r = \frac{P_t G^2 \sigma \lambda^2}{(4\pi)^3 R^4 L}$$

where P_t is the transmitted power, G is the antenna gain, σ is the target radar cross section (RCS), λ is the wavelength, and R is the propagation distance. All propagation losses other than free space path loss are included in the L term. The rest of the example shows how to estimate this L term in different scenarios.

Free Space Path Loss

Free space path loss is computed as a function of propagation distance and frequency. In free space, RF signals propagate at the speed of light in all directions. At a far enough distance, the radiating source looks like a point in space and the wavefront forms a sphere whose radius is equal to R. The power density at the wavefront is inversely proportional to R^2:

$$\frac{P_t}{4\pi R^2}$$

where P_t is the transmitted signal power. For a monostatic radar where the signal has to travel both directions (from the source to the target and back), the dependency is actually inversely proportional to R^4, as shown previously in the radar equation. The loss related to this propagation mechanism is referred to as free space path loss, sometimes also called the spreading loss. Quantitatively, free space path loss is also a function of frequency, given by [5]:

$$L_{fs} = 20 * \log_{10}(\frac{4\pi R}{\lambda}) \quad dB$$

As a convention, propagation losses are often expressed in dB. This convention makes it much easier to derive the two-way free space path loss by simply doubling the one-way free space loss.

Use the `fspl` function to calculate the free-space path loss, and plot the loss for frequencies between 1 and 1000 GHz, for different ranges.

```
c = physconst('lightspeed');
R0 = [100 1e3 10e3];
freq = (1:1000).'*1e9;
apathloss = fspl(R0,c./freq);
loglog(freq/1e9,apathloss);
```

```
grid on;
ylim([90 200]);
legend('Range: 100 m', 'Range: 1 km', 'Range: 10 km','Location','northwest');
xlabel('Frequency (GHz)');
ylabel('Path Loss (dB)');
title('Free Space Path Loss');
```
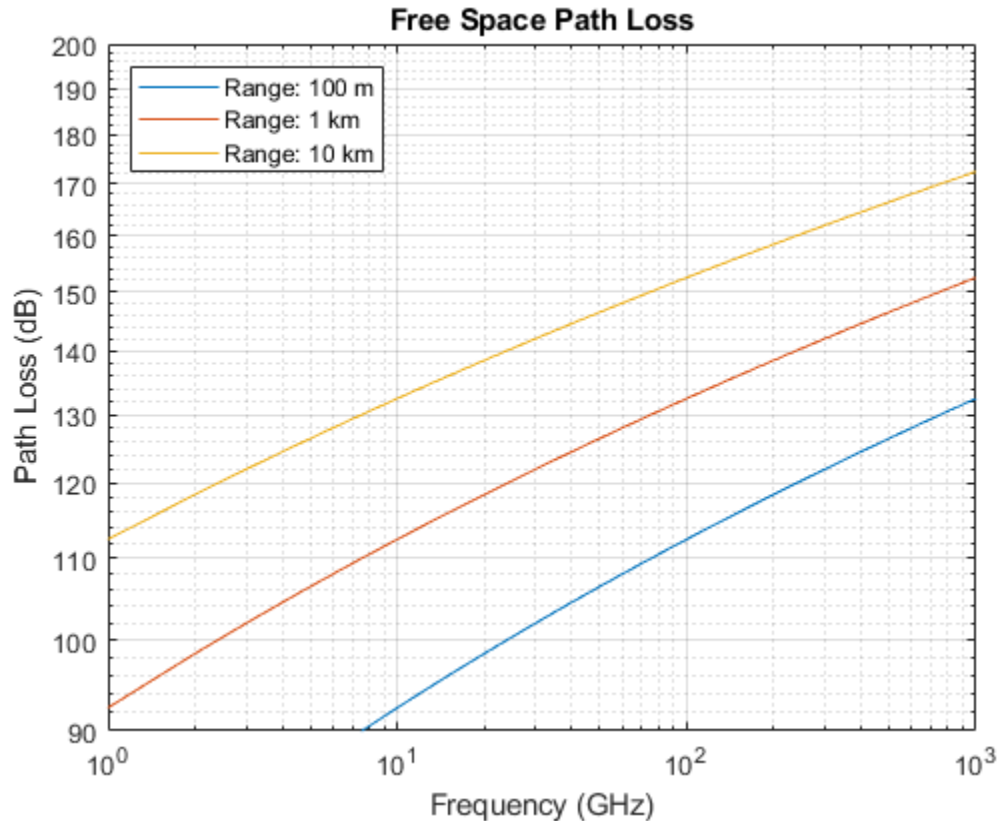

The figure shows that the propagation loss increases with range and frequency.

Propagation Loss Due to Precipitation and Atmosphere

In reality, signals do not always travel in a vacuum, so free space path loss describes only part of the signal attenuation. Signals interact with particles in the air and lose energy along the propagation path. The loss varies with different factors such as pressure, temperature, and water density.

Loss Due to Rain and Snow

Rain can be a major limiting factor for radar systems, especially when operating above 5 GHz. In the ITU model in [2], rain is characterized by the rain rate (in mm/h). According to [6], the rain rate can range from less than 0.25 mm/h for very light rain to over 50 mm/h for extreme rains. In addition, because of the rain drop's shape and its relative size compared to the RF signal wavelength, the propagation loss due to rain is also a function of signal polarization. In general, horizontal polarization represents the worst case for propagation loss due to rain.

The functions `rainpl` and `cranerainpl` can be used to compute losses due to rain according to the ITU and Crane models, respectively. Both models are valid between 1 GHz and 1 THz. Let the

polarization be horizontal, so the tilt angle is 0, and let the signal propagate parallel to the ground, so the elevation angle is 0. Plot losses computed with both models and compare.

```matlab
R0 = 5e3;                    % 5 km range
rainrate = [1 4 20];         % rain rate in mm/h
el = 0;                      % 0 degree elevation
tau = 0;                     % horizontal polarization

for m = 1:numel(rainrate)
    rainloss_itu(:,m) = rainpl(R0,freq,rainrate(m),el,tau)';
    rainloss_crane(:,m) = cranerainpl(R0,freq,rainrate(m),el,tau)';
end
loglog(freq/1e9,rainloss_itu);
hold on;
set(gca,'ColorOrderIndex',1); % reset color index for better comparison
loglog(freq/1e9,rainloss_crane,'--');
hold off;
grid on;
legend('Light Rain (ITU)','Moderate Rain (ITU)','Heavy Rain (ITU)',...
    'Light Rain (Crane)','Moderate Rain (Crane)','Heavy Rain (Crane)', ...
    'Location','SouthEast');
xlabel('Frequency (GHz)');
ylabel('Attenuation at 5 km (dB)')
title('Rain Attenuation for Horizontal Polarization');
```
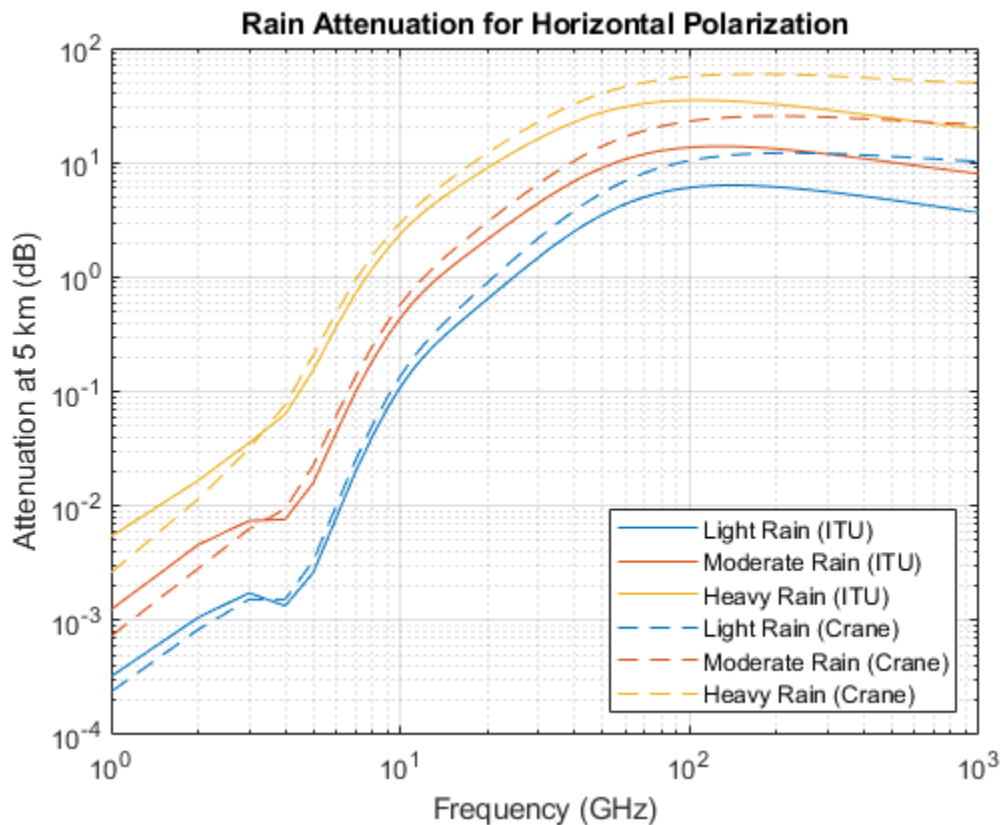

The losses computed with the Crane model are mostly larger than the losses computed with the ITU model at this propagation range. At smaller propagation ranges and lower frequencies, the ITU model

may output a smaller attenuation value than Crane. Note that the models differ greatly enough that at higher frequencies, light rainfall for one model may have the same attenuation as moderate rainfall for the other model.

Similar to rainfall, snow can also have a significant impact on the propagation of RF signals. A common practice is to treat snow as rainfall and compute the propagation loss based on the rain model, even though this approach tends to overestimate the loss a bit. Attenuation due to propagation through snow is not considered dependent on polarization, but is highly dependent on frequency. The model for losses due to snow is parameterized by the equivalent liquid content instead of volume. For a given water content, snow requires about 10 times as much volume as rain.

Use the `snowpl` function to compute losses due to snow, and plot the losses against frequency. By default, this function uses the Gunn-East attenuation model, which is generally valid up to about 20 GHz.

```
freq = (1:20)*1e9;
R0 = 1e3;                % 1 km range
snowrate = [0.1 1.5 4]; % equivalent liquid water content in mm/h

for m = 1:numel(snowrate)
    snowloss(:,m) = snowpl(R0,freq,snowrate(m));
end
loglog(freq/1e9,snowloss);
grid on;
legend('Light Snow','Moderate Snow','Heavy Snow', ...
    'Location','SouthEast');
xlabel('Frequency (GHz)');
ylabel('Attenuation at 1 km (dB)')
title('Snow Attenuation');
```
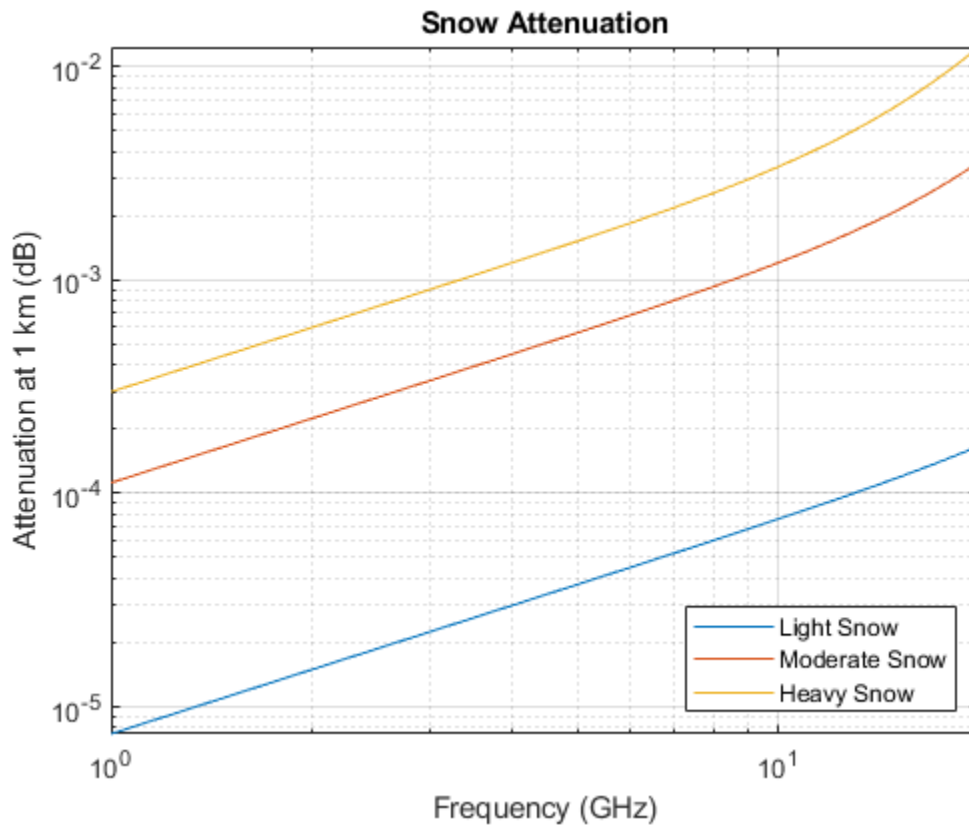
Loss Due to Fog and Cloud

Fog and cloud are formed with water droplets too, although much smaller compared to rain drops. The size of fog droplets is generally less than 0.01 cm. Fog is often characterized by the liquid water density. A medium fog with a visibility of roughly 300 meters, has a liquid water density of 0.05 g/m^3. For heavy fog where the visibility drops to 50 meters, the liquid water density is about 0.5 g/m^3. The atmosphere temperature (in Celsius) is also present in the ITU model for propagation loss due to fog and cloud [3].

Use the `fogpl` function to compute losses due to fog, and plot the losses against frequency. The ITU model for attenuation due to fog is valid between 10 GHz and 1 THz.

```
freq = (10:1000)*1e9;
T = 15;                          % 15 degree Celsius
waterdensity = [0.01 0.05 0.5];  % liquid water density in g/m^3
for m = 1: numel(waterdensity)
    fogloss(:,m) = fogpl(R0,freq,T,waterdensity(m))';
end
loglog(freq/1e9,fogloss);
grid on;
legend('Light Fog','Medium Fog','Heavy Fog','Location','southeast');
xlabel('Frequency (GHz)');
ylabel('Attenuation at 1 km (dB)')
title('Fog Attenuation');
```

Fog Attenuation
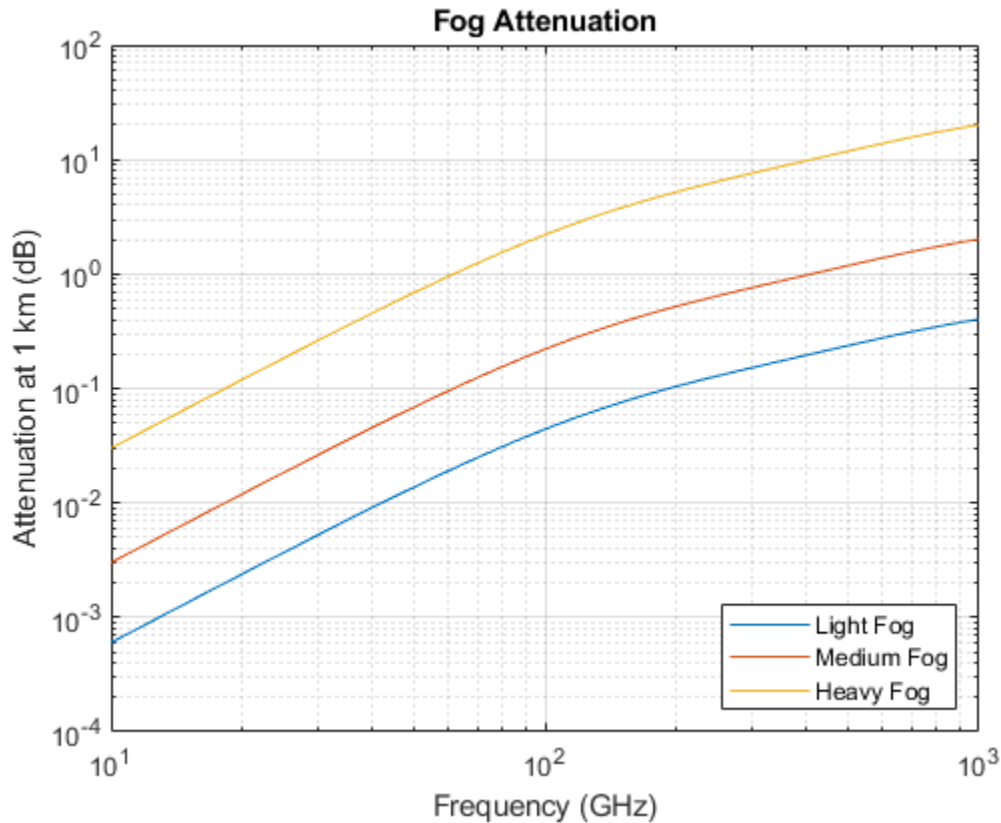
Note that in general fog is not present when it is raining.

Loss Due to Atmospheric Absorption and Lensing

Even when there is no fog or rain, the atmosphere is full of gases that still affect the signal propagation. The ITU model [4] describes atmospheric gas attenuation as a function of both dry air pressure, like oxygen, measured in hPa, and water vapour density, measured in g/m^3.

Use the `tropopl` function to compute losses due to atmospheric absorption, and plot the losses against frequency. By default, this function uses the Mean Annual Global Reference Atmosphere (MAGRA) model to get typical values of temperature, pressure, and water vapor density for a given altitude. We can also specify a latitude model to use a model tailored for a specific range of latitudes. Some latitude models also allow for specification of a season. Let our altitude be 2 km (note that the troposphere, for which this model is valid, extends up to 10 km) and our propagation path be depressed by 5 degrees. This function returns the total loss due to atmospheric absorption over the slanted propagation path, but does not include dissipation due to refraction (lensing). Compare losses between the low, mid, and high latitude models.

```
height = 2e3;
el = -5; % elevation angle
atmloss_low = tropopl(R0,freq,height,el,'LatitudeModel','Low');
atmloss_mid = tropopl(R0,freq,height,el,'LatitudeModel','Mid');
atmloss_high = tropopl(R0,freq,height,el,'LatitudeModel','High');
loglog(freq/1e9,atmloss_low);
hold on;
loglog(freq/1e9,atmloss_mid);
```

```
loglog(freq/1e9,atmloss_high);
hold off;
grid on;
legend('Low Latitudes','Mid Latitudes','High Latitudes','Location','northwest');
xlabel('Frequency (GHz)');
ylabel('Attenuation at 1 km (dB)')
title('Atmospheric Gas Attenuation');
```
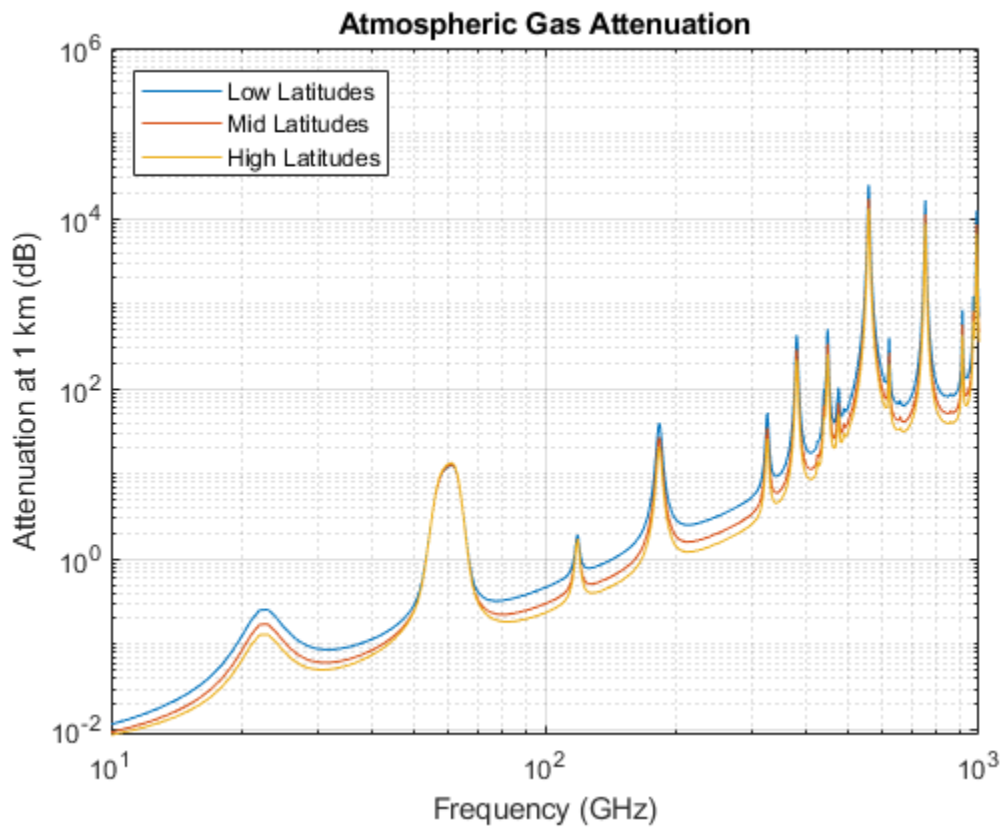

The plot suggests that there is a strong absorption due to atmospheric gases at around 60 GHz.

Another source of losses due to atmosphere is from atmospheric lensing. This is a phenomenon whereby the angular extent of a transmission is increased with range due to a refractivity gradient. This spreading of energy decreases the energy density along the nominal (straight) propagation path, independent of frequency.

Atmospheric pressure, and thus refractivity, changes with altitude. So for a given height, the elevation angle of the propagation path is enough to determine losses due to this effect.
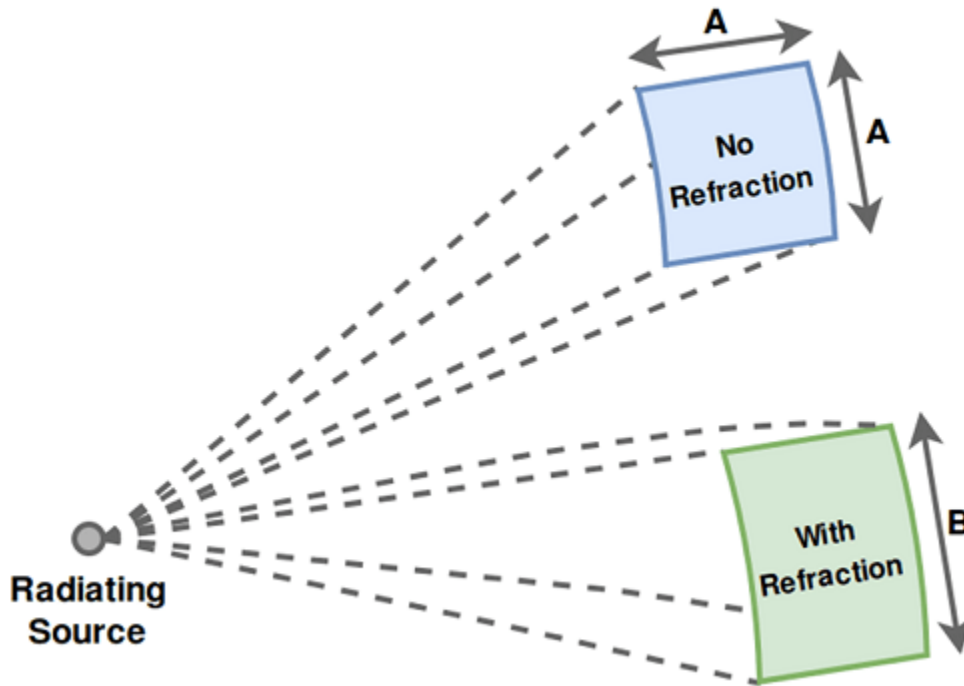
Use the `lenspl` function to compute these losses and plot against frequency. Because this loss is independent of frequency, plot the loss against propagation range for a set of heights. Use an elevation angle of 0.05 degrees for a slanted propagation path.

```
R = 1e3:1e3:100e3;        % propagation range
el = 0.05;                % elevation angle
heights = [10 100 200];   % radar platform heights
for m = 1:numel(heights)
    lenloss(:,m) = lenspl(R,heights(m),el);
end
semilogy(R/1e3,lenloss);
grid on;
legend('Height: 10 m','Height: 100 m','Height: 200 m','Location',...
    'southeast');
xlabel('Propagation Range (km)');
ylabel('Attenuation (dB)')
title('Atmospheric Lensing Attenuation');
```
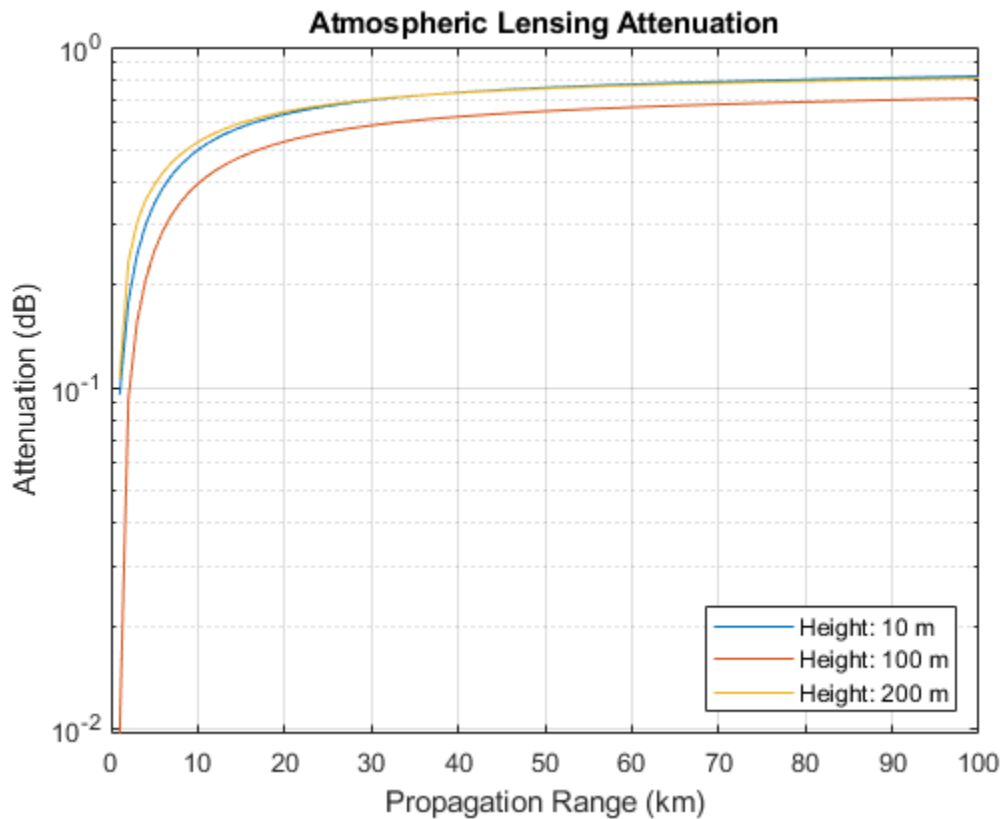
Attenuation due to lensing decreases as altitude increases. For convenience, attenuation due to lensing is also provided as a secondary output from `tropopl`.

Loss Due to Polarization Mismatch

Some types of propagation loss are dependent on the polarization of the transmitted radiation, such as with rain loss. This is a result of the chemical and structural properties of the medium. However, even in free space there may be losses due to a mismatch of the propagated polarization vector and the polarization of the receiving antenna. For example, if the propagated polarization vector is orthogonal to the polarization of the receiving antenna, almost no direct signal energy would be received. Note that the propagated polarization vector is not in general the same as the transmitted polarization vector, as the direction of propagation must be taken into account. Note also that the other loss functions which take polarization as an input do not compute losses due to this mismatch. Polarization-dependent losses due to properties of the propagation medium can be handled separately from losses due to polarization mismatch, as the latter is heavily dependent on the transmitter/ receiver orientation.

Use the `polloss` function to compute the loss due to polarization mismatch for a given transmit/ receive polarization, platform positions, and platform orientations. Place the transmit platform at the origin with no rotation from inertial. Place the receive platform along the X axis and compute polarization loss for a range of roll angles. Let the antenna polarizations both be vertical.

```
poltx = [0;1];   % [H;V] polarization
polrx = [0;1];
postx = [0;0;0];
posrx = [100;0;0];
```

```
frmtx = eye(3); % transmit frame aligned with inertial
rolls = 0:180;

for m = 1:numel(rolls)
    frm_r = rotx(rolls(m));
    rho(m) = polloss(poltx,polrx,posrx,frm_r,postx,frmtx);
end

semilogy(rolls,rho);
grid on;
xlabel('Roll Angle (deg)');
ylabel('Attenuation (dB)');
title('Attenuation Due to Polarization Mismatch');
```
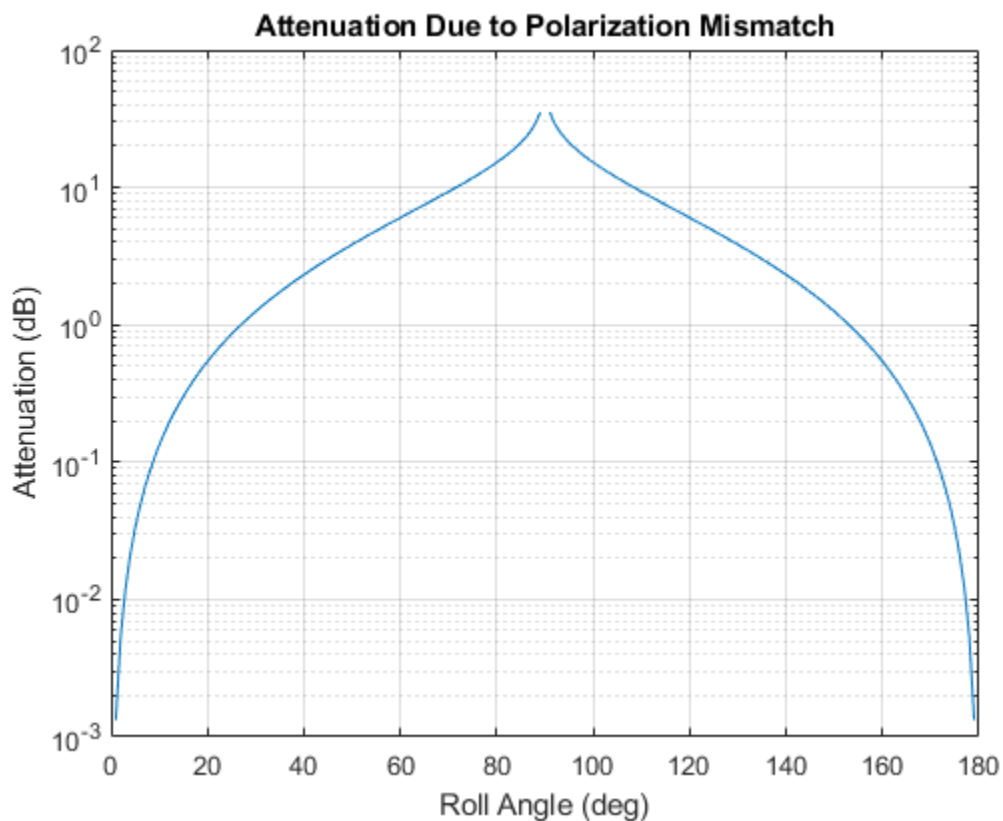

The attenuation approaches infinity at a 90 degree roll angle.

Radar Propagation Factor and Vertical Coverage Diagram

When transmitting over a wide angle or from an antenna close to the ground, multipath from ground bounce, along with refraction from atmosphere, yields a radiation pattern at a given range that can be quite different from the nominal transmit pattern. This is captured by the radar propagation factor, which is the ratio of the actual field strength relative to what the field strength would be in free space. The propagation factor can vary greatly as the relative phase between the direct and indirect path signals changes.

A vertical coverage diagram (Blake chart) is a compact way of displaying contours of fixed signal energy (such as a minimum signal power for detection) as a function of propagation range and

elevation angle. Only the vertical plane in which both the direct and indirect path signals propagate is considered.

The function `radarvcd` takes a reference range as input and returns the range at which the received power in the multipath environment equals what it would be in free space. This effective range is plotted on a range-height-angle chart. This can quickly give, for example, the actual detection range given a free-space detection range, as a function of range, height, or elevation angle.

Use a free-space detection range of 100 km, transmit frequencies in L-Band and C-Band, and an antenna height of 12 m. A sinc transmit pattern is used by default.

```
freq = [1.06 5.7]*1e9; % L-Band and C-Band transmit frequencies (Hz)
antht = 12;             % height of antenna (m)
rngfs = 100;            % free-space detection range (km)
for m = 1:numel(freq)
    [vcp{m}, vcpang{m}] = radarvcd(freq(m),rngfs,antht);
end
```

`blakechart` takes these detection ranges and angles, along with additional atmospheric properties to create the Blake chart. Use the `refractiveidx` function to compute the corresponding refraction exponent for input to `blakechart`.

```
[~,N] = refractiveidx(0); % atmospheric refractivity at the surface
helperPlotBlakeChart(vcp,vcpang,N)
```
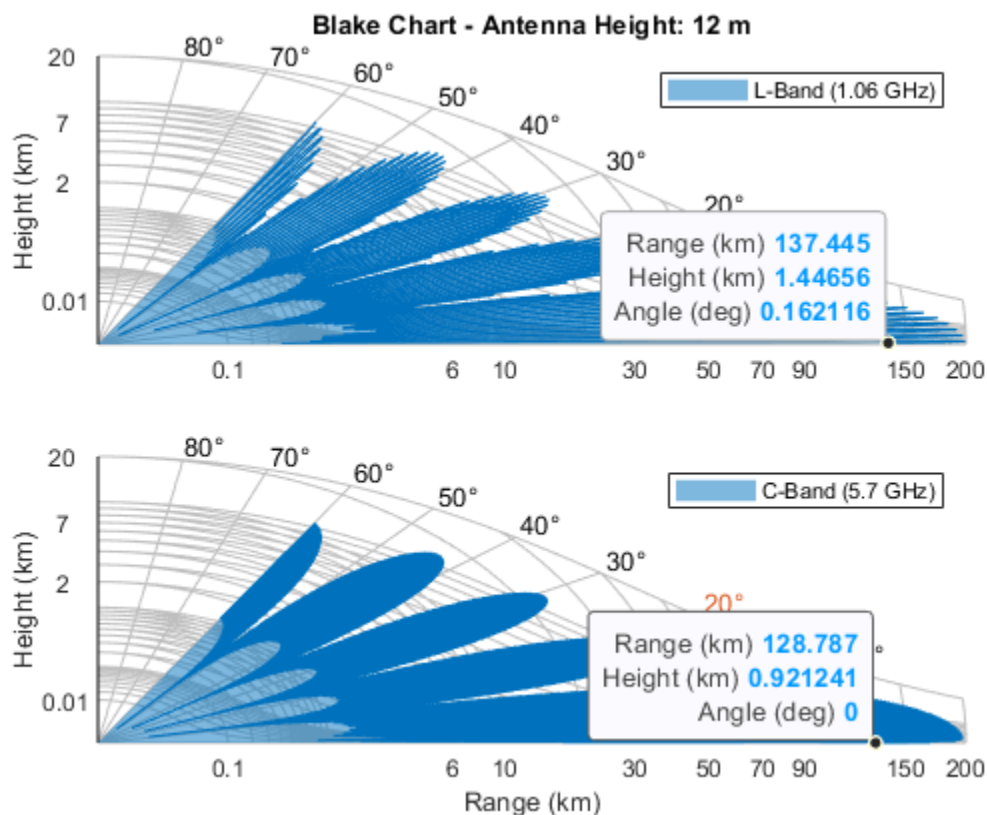

Ground-bounce interference dominates the propagation factor for shorter ranges, in the so-called interference region, but at longer ranges and low elevation angles the propagation factor is

dominated by diffraction over the horizon, the diffraction region. Use the `radarpropfactor` function to compute the propagation factor for an interval of ranges and observe the difference between these two regions.

Compute the propagation factor for a fixed height above the surface of 1 km and propagation ranges between 50 and 200 km. Set the surface slope and height standard deviation to 0 to represent a smooth surface. Perform the analysis for the two frequency bands.

```
tgtht = 1e3;                % target height (m)
R = (50:200)*1e3;           % propagation range (m)
Re = effearthradius;        % effective Earth radius (m)
Rd = sqrt(2*Re)*(sqrt(antht) + sqrt(tgtht)); % diffraction range
F = zeros(numel(freq),numel(R));
for m = 1:numel(freq)
    F(m,:) = radarpropfactor(R,freq(m),antht,tgtht,'SurfaceHeightStandardDeviation',0,'SurfaceSl
end
helperPlotPropagationFactor(R,F,Rd)
```
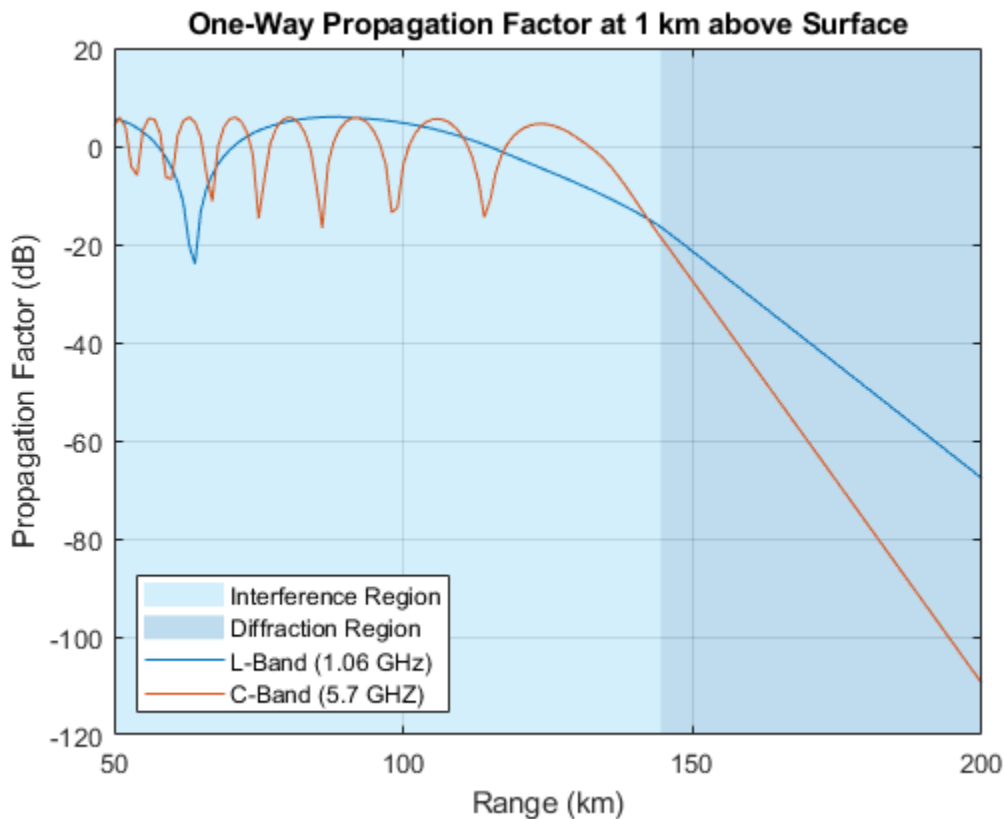

The propagation factor oscillates in the interference region, then decreases quickly in the diffraction region.

Combine the ground-bounce interference and atmospheric absorption losses. Assume in this calculation that a 3.3 GHz S-band surface ship radar is 20 m above the water and has an elevation beamwidth of 30 deg.

```
freq = 3.3e9;                     % Frequency (Hz)
elbw = 30;                        % Elevation beamwidth (deg)
```

```
Rkm = 1:0.1:120;                  % Range (km)
R = Rkm.*1e3;                     % Range (m)
[htsd,beta0] = searoughness(1);   % Sea surface
anht = 20 + 2*htsd;               % Radar height (m)
tgtht = (anht+1):1:300;           % Target height (m)

% Calculate combined environment losses for different heights and ranges
[PLdB, PLdBNorm] = helperCombineEnvLosses(R,freq,anht,tgtht,htsd,beta0,elbw);

% Plot combined losses for different heights and ranges
helperPlotCombinedEnvLosses(Rkm,freq,anht,tgtht,PLdBNorm)
```
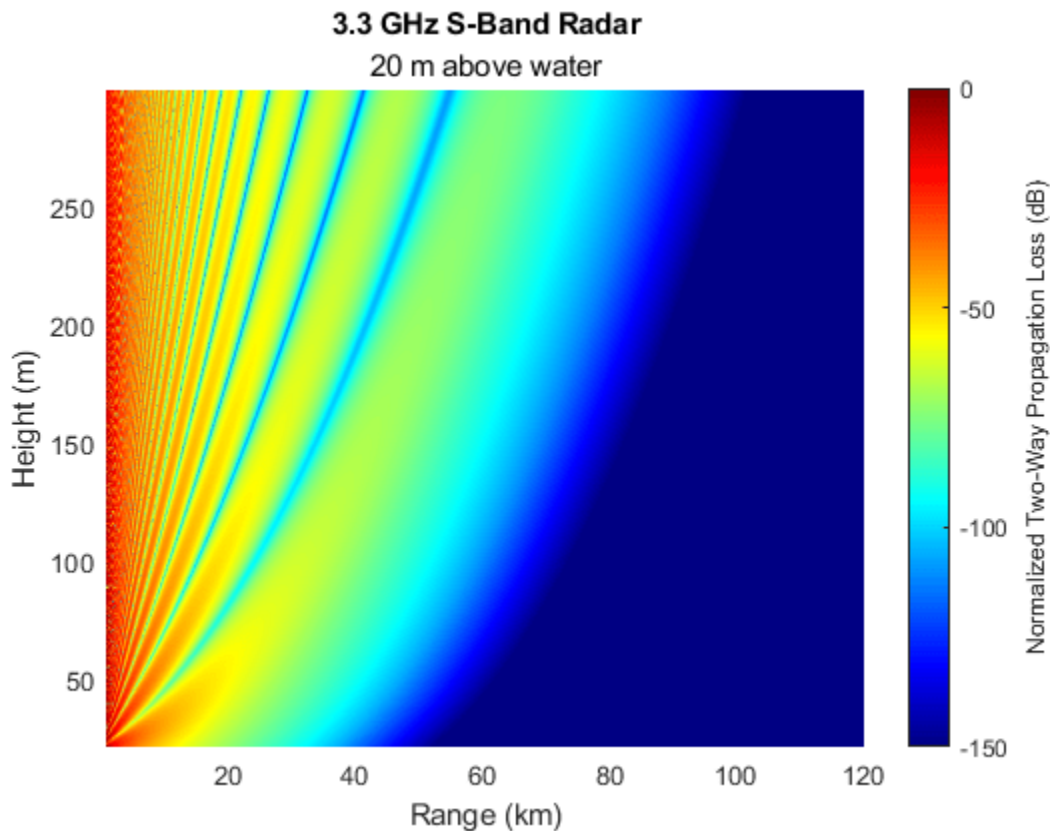

Multipath Propagation, Time Delays, and Doppler Shifts

Signals may not always propagate along the line of sight, but arrive at the destination via different paths and may add up either constructively or destructively. This multipath effect can cause significant fluctuations in the received signal power.

The functions mentioned in the previous sections for computing propagation losses are useful to establish budget links, but to simulate the propagation of arbitrary signals, you also need to apply range-dependent time delays, gains, and phase shifts. Various channel objects are available to model multipath propagation. For a simple line-of-sight path, use the `phased.LOSChannel` object to model the propagation subject to any of the loss types described previously.

Ground reflection is a common phenomenon for many radar or wireless communication systems. For example, when a ground-based or sea-based radar illuminates a target, the signal not only propagates

along the direct line of sight but is also reflected from the ground. Use the `twoRayChannel` object to model the combination of a direct path and single-bounce path, such as for ground reflection.

Time Delays and Doppler Shifts

First, define the transmitted signal. Use a rectangular waveform.

```
waveform = phased.RectangularWaveform('PRF',250);
wav = waveform();
```

Assume an L-band operating frequency of 1.9 GHz. Model the channel.

```
fc = 1.9e9;
channel = twoRayChannel('PropagationSpeed',c,'OperatingFrequency',fc);
```

Assume the target unit is 1.65 km above the ground, the radar antenna is 12 meters above the ground at a 50 km distance. Simulate the signal as it reaches the target.

```
pos_radar = [0;0;12];
pos_target = [50e3;0;1.65e3];
vel_radar = [0;0;0];
vel_target = [-200;0;0];
y2ray = channel(wav,pos_radar,pos_target,vel_radar,vel_target);
```

Visualize the transmitted and propagated pulses and their normalized spectra. The channel introduced a delay of 167 μs which corresponds to the 50 km range of the target divided by the speed of light.

```
[delay, dop] = helperPlotDelayAndDopplerShift(wav,y2ray,waveform.SampleRate);
```
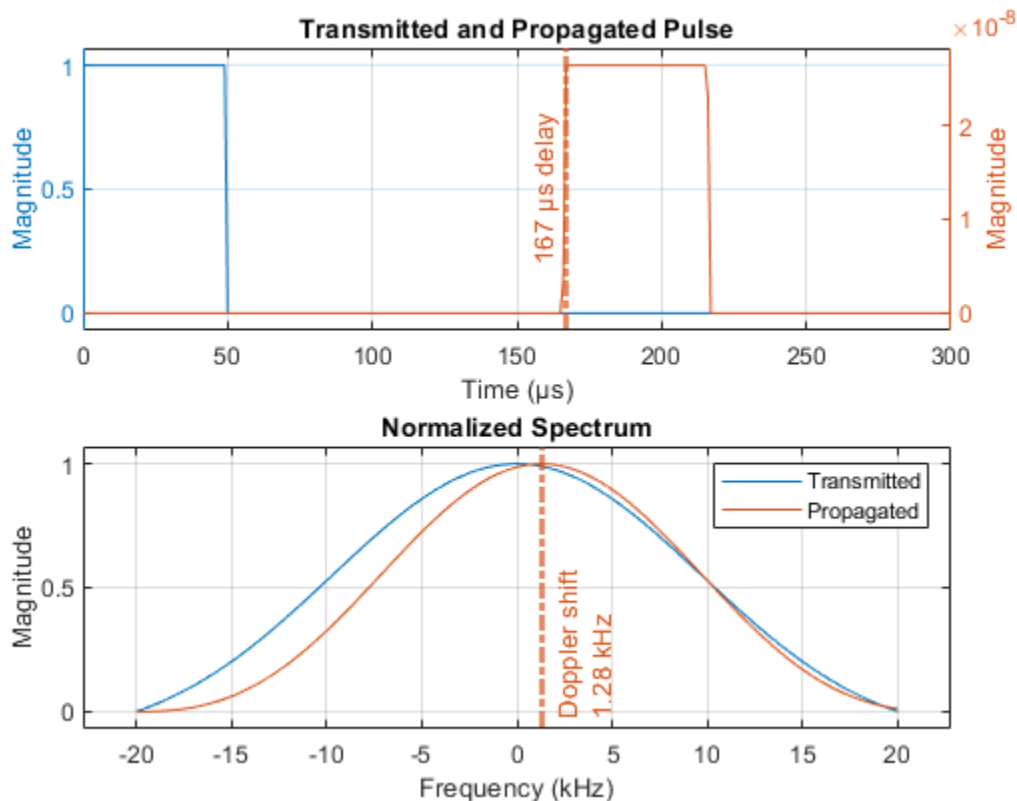
```
estRange = delay*c*1e-3 % km
```

```
estRange =

   49.9954
```

The channel also applied a Doppler shift that corresponds to the range rate of the target. Compare the estimated value to the -200 m/s ground truth using the `dop2speed` and `freq2wavelen` functions.

```
estRangeRate = -dop2speed(dop,freq2wavelen(fc)) % m/s
```

```
estRangeRate =

 -201.9038
```

Multipath Fading

Calculate the signal loss suffered in this channel.

```
L_2ray = pow2db(bandpower(wav))-pow2db(bandpower(y2ray))
```

```
L_2ray =

  151.5888
```

Calculate the free-space path loss.

```
L_ref = fspl(norm(pos_target-pos_radar),c/fc)
```

```
L_ref =

  132.0069
```

The result suggests that in this configuration, the channel introduces an extra 19.6 dB loss to the received signal compared to the free space case. Now assume the target flies a bit higher at 1.8 km above the ground. Repeating the simulation above suggests that this time the ground reflection actually provides a 6 dB gain. Although free space path loss is essentially the same in the two scenarios, a 150 m move caused a 25.6 dB fluctuation in signal power.

```
pos_target = [50e3;0;1.8e3];
y2ray  = channel(wav,pos_radar,pos_target,vel_radar,vel_target);
L_2ray = pow2db(bandpower(wav))-pow2db(bandpower(y2ray))
L_ref  = fspl(norm(pos_target-pos_radar),c/fc)
```

```
L_2ray =

  126.0374
```

```
L_ref =

  132.0078
```

Increasing the bandwidth of a system increases the capacity of its channel. This enables higher data rates in communication systems and finer range resolutions for radar systems. The increased bandwidth can also improve robustness to multipath fading for both systems.

Typically, wideband systems operate with a bandwidth of greater than 5% of their center frequency. In contrast, narrowband systems operate with a bandwidth of 1% or less of the center frequency.

The narrowband channel in the preceding section was shown to be very sensitive to multipath fading. Slight changes in the target's height resulted in considerable signal losses.

Plot the fading loss for the channel by varying the height of the target across a span of operational heights for this radar system. Choose a span of heights from 1 km to 3 km.

```
% Simulate the signal fading at the target for heights from 1 km to 3 km
hTarget = linspace(1e3,3e3);
pos_target = repmat([50e3;0;1.6e3],[1 numel(hTarget)]);
pos_target(3,:) = hTarget;
vel_target = repmat(vel_target,[1 numel(hTarget)]);

release(channel);
y2ray = channel(repmat(wav,[1 numel(hTarget)]),pos_radar,pos_target,vel_radar,vel_target);
```

Plot the signal loss observed at the target.

```
L2ray = pow2db(bandpower(wav))-pow2db(bandpower(y2ray));

clf;
plot(hTarget,L2ray);
xlabel('Target Height (m)');
ylabel('One-Way Propagation Loss (dB)');
title('Multipath Fading Observed at the Target');
grid on;
```

Multipath Fading Observed at the Target
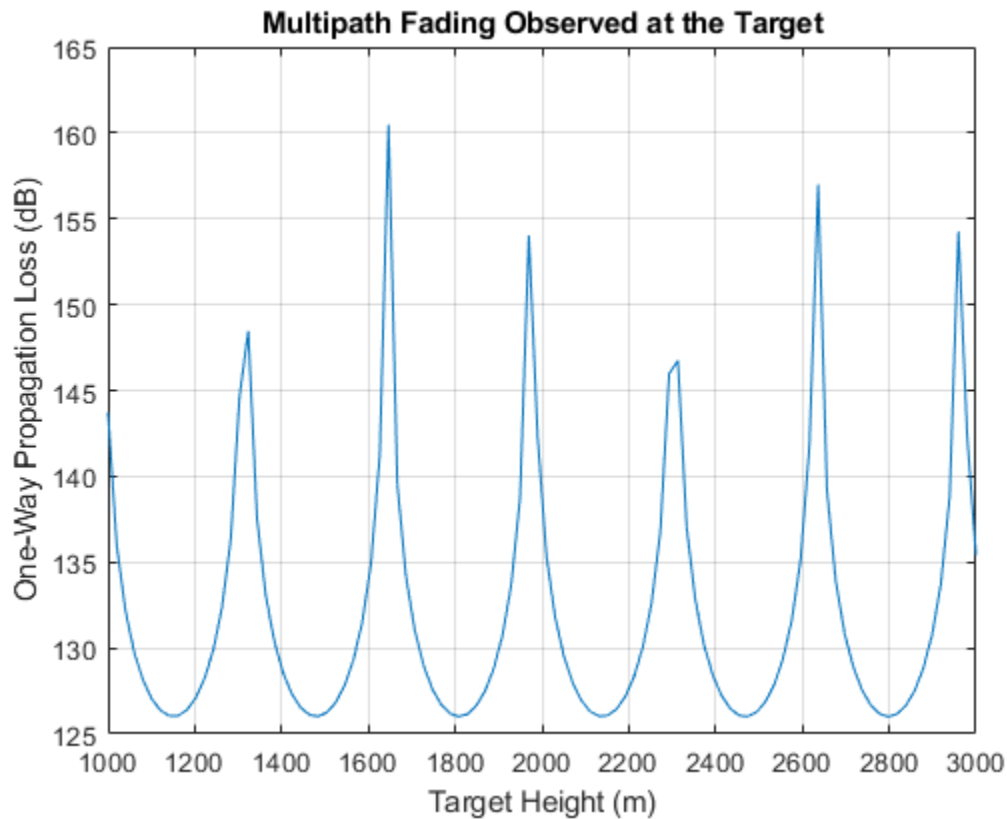
The sensitivity of the channel loss to target height for this narrowband system is clear. Deep signal fades occur at heights that are likely to be within the surveillance area of the radar.

Increasing the bandwidth of the channel can improve robustness to these multipath fades. To do this, use a wideband waveform with a bandwidth of 8% of the center frequency of the link.

```
bw = 0.08*fc;
pulse_width = 1/bw;
fs = 2*bw;

waveform = phased.RectangularWaveform('SampleRate',fs,'PRF',2000,'PulseWidth',pulse_width);
wav = waveform();
```

Use a wideband version of this channel model, widebandTwoRayChannel, to simulate multipath reflections of this wideband signal off of the ground between the radar and the target, and to compute the corresponding channel loss.

```
channel = widebandTwoRayChannel('PropagationSpeed',c,'OperatingFrequency',fc,'SampleRate',fs);
```

Simulate the signal at the target for various operational heights.

```
y2ray_wb = channel(repmat(wav,[1 numel(hTarget)]),pos_radar,pos_target,vel_radar,vel_target);
L2ray_wb = pow2db(bandpower(wav))-pow2db(bandpower(y2ray_wb));

hold on;
plot(hTarget,L2ray_wb);
hold off;
legend('Narrowband','Wideband');
```
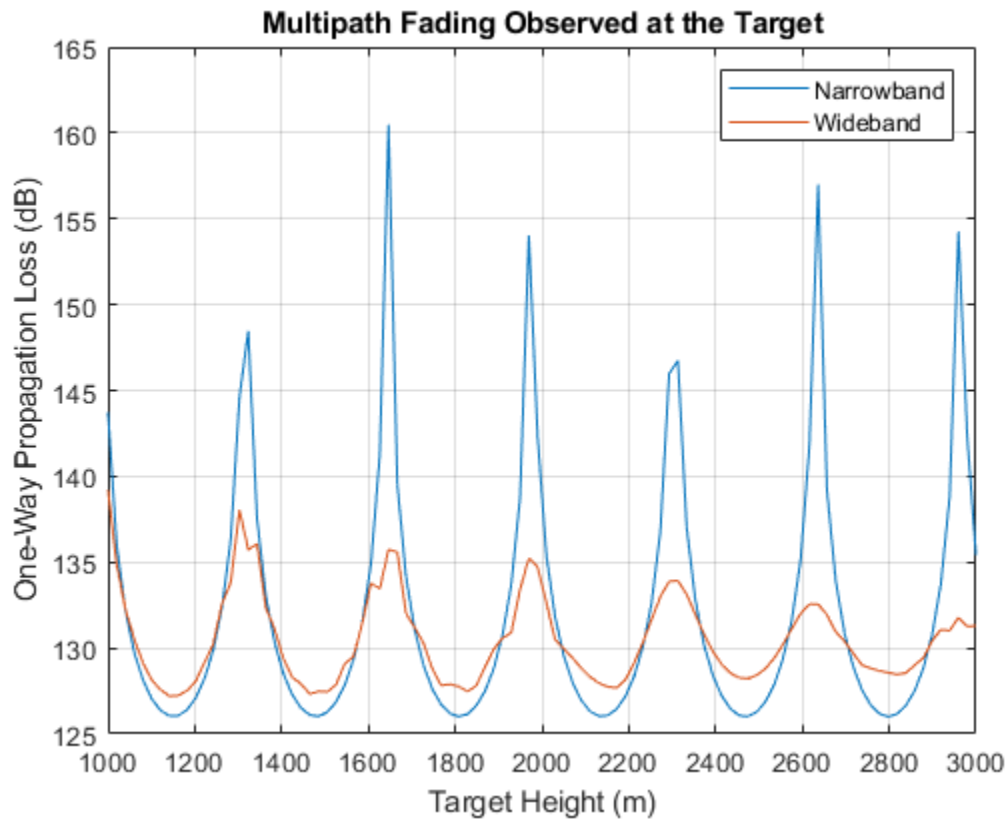
As expected, the wideband channel provides much better performance across a wide range of heights for the target. In fact, as the height of the target increases, the impact of multipath fading almost completely disappears. This is because the difference in propagation delay between the direct and bounce path signals is increasing, reducing the amount of coherence between the two signals when received at the target.

Conclusion

This example provides an overview of RF propagation losses due to atmospheric and weather effects. It also introduces multipath signal fluctuations due to ground bounces. It highlights functions and objects to simulate attenuation losses for narrowband and wideband single-bounce channels.

References

[1] Seybold, John S. Introduction to RF Propagation: Seybold/Introduction to RF Propagation. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2005. https://doi.org/10.1002/0471743690

[2] Recommendation ITU-R P.838-3, 2005

[3] Recommendation ITU-R P.840-3, 2013

[4] Recommendation ITU-R P.676-10, 2013

[5] Recommendation ITU-R P.525-2, 1994

[6] Rain, A Water Resource (Pamphlet), U.S. Geological Survey, 1988

Supporting Functions

helperPlotPropagationFactor

```matlab
function helperPlotPropagationFactor(R,F,Rd)

% Plot interference and diffraction region patches
[minF, maxF] = bounds(F(:));
maxF = ceil((maxF+10)/10)*10;
minF = floor((minF-10)/10)*10;
yPatch = [minF minF maxF maxF];
c1 = [0.3010 0.7450 0.9330];
c2 = [0 0.4470 0.7410];
clf % clear current figure
fill([R(1) Rd Rd R(1)]/1e3,yPatch,c1,'EdgeColor','none','FaceAlpha',0.25)
hold on
fill([Rd R(end) R(end) Rd]/1e3,yPatch,c2,'EdgeColor','none','FaceAlpha',0.25)

% Plot one-way propagation factor
set(gca,'ColorOrderIndex',1); % reset color index
plot(R/1e3,F);
ylim([minF maxF])
grid on;
xlabel('Range (km)');
ylabel('Propagation Factor (dB)');
title('One-Way Propagation Factor at 1 km above Surface');
legend('Interference Region', 'Diffraction Region',...
    'L-Band (1.06 GHz)', 'C-Band (5.7 GHZ)',...
    'Location','SouthWest')
hold off
end
```

helperPlotBlakeChart

```matlab
function helperPlotBlakeChart(vcp,vcpang,N)
% Calculate refraction exponent
DelN = -7.32*exp(0.005577*N);
rexp = log(N./(N + DelN));

subplot(211)
blakechart(vcp{1},vcpang{1},'SurfaceRefractivity',N,'RefractionExponent',rexp);
legend('L-Band (1.06 GHz)')
xlabel('')
title ('Blake Chart - Antenna Height: 12 m')
subplot(212)
blakechart(vcp{2},vcpang{2},'SurfaceRefractivity',N,'RefractionExponent',rexp);
allc = get(gca,'Children');
set(allc(11),'Color',[0.8500 0.3250 0.0980]) % Change line color
title('')
legend('C-Band (5.7 GHz)')
end
```

helperPlotDelayAndDopplerShift

```matlab
function [delay, dop] = helperPlotDelayAndDopplerShift(wav,y2ray,Fs)
% Plot transmitted and propagated pulse
t = 1e6*(0:numel(wav)-1)'/Fs;
subplot(211)
```

```matlab
yyaxis left
plot(t,abs(wav))
ylabel('Magnitude')
yyaxis right
plot(t,abs(y2ray))
grid on
axis padded
xlim([0 300])
xlabel(['Time (' char(0x00B5) 's)'])
ylabel('Magnitude')
title('Transmitted and Propagated Pulse')

% Annotation
delay = midcross(abs(y2ray),t/1e6,'MidPercentReferenceLevel',80); % seconds
delay = delay(1);
xl = xline(1e6*delay,'-.',... % Annotation
    {[num2str(round(1e6*delay)),' ',char(0x00B5) 's delay']},'Color',[0.8500 0.3250 0.0980]);
xl.LabelVerticalAlignment = 'middle';
xl.LabelHorizontalAlignment = 'left';
xl.LineWidth = 2;

% Plot power spectrum
subplot(212)
[p,f] = pspectrum([wav y2ray],Fs,'FrequencyLimits',[-20e3 20e3]);
p = abs(p);
plot(1e-3*f,rescale(p,'InputMin',min(p),'InputMax',max(p)));
axis padded
grid on
[~,idx]=max(p);
dop = f(idx(2))-f(idx(1));    % Hz
xlabel('Frequency (kHz)')
ylabel('Magnitude')
title('Normalized Spectrum')

xl = xline(1e-3*dop,'-.',... % Annotation
    {'Doppler shift',[num2str(round(dop)*1e-3) ' kHz']},'Color',[0.8500 0.3250 0.0980]);
xl.LabelVerticalAlignment = 'bottom';
xl.LineWidth = 2;
legend('Transmitted','Propagated')
end
```

helperCombineEnvLosses

```matlab
function [PLdB, PLdBNorm] = helperCombineEnvLosses(R,freq,anht,tgtht,htsd,beta0,elbw)
% Calculate the combined environment losses
numHt = numel(tgtht);
numR = numel(R);
F = zeros(numHt,numR);
for ih = 1:numHt
    F(ih,:) = radarpropfactor(R, freq, anht, tgtht(ih),...
    'SurfaceHeightStandardDeviation',htsd,'SurfaceSlope',beta0,...
    'ElevationBeamwidth', elbw);
end

% Free space spreading loss
Lspl_dB = 2*fspl(R,freq2wavelen(freq));  % Factor of 2 for two-way

% Perform tropospheric losses calculation for a subset of elevation angles,
```

```matlab
    % since the ray refracting can take a long time.
    numEl = 10;
    minEl = height2el(tgtht(1),anht,R(end)); % Min elevation angle (deg)
    maxEl = height2el(tgtht(end),anht,R(1)); % Max elevation angle (deg)
    elSubset = linspace(minEl,maxEl,numEl);
    LtropoSubset = zeros(numEl,numR);
    for ie = 1:numEl
        LtropoSubset(ie,:) = tropopl(R,freq,anht,elSubset(ie));
    end
    % Interpolate tropospheric losses for all elevation angles of interest
    Ltropo = zeros(numHt,numR);
    for ir = 1:numR
        el = height2el(tgtht,anht,R(ir));
        Ltropo(:,ir) = interp1(elSubset,LtropoSubset(:,ir),el);
    end

    PLdB = 2*F - Lspl_dB - Ltropo;          % Factor of 2 for two-way
    PLdBNorm = PLdB - max(PLdB(:));
    end
```

helperPlotCombinedEnvLosses

```matlab
function helperPlotCombinedEnvLosses(Rkm,freq,anht,tgtht,PLdBNorm)
% Plot combined losses for different heights and ranges
hP = pcolor(Rkm,tgtht,PLdBNorm);
set(hP, 'EdgeColor', 'none');
title([num2str(freq/1e9) ' GHz S-Band Radar'])
subtitle([num2str(round(anht)) ' m above water'])
xlabel('Range (km)')
ylabel('Height (m)')
colormap('jet');
caxis([-150 0])
hC = colorbar;
hC.Label.String = 'Normalized Two-Way Propagation Loss (dB)';
end
```

Display Micro-Doppler Shift of Moving Bicyclist

Create a Simulink™ model of a moving bicyclist.

```
open_system('BicyclistMicrospectrumExample');
```
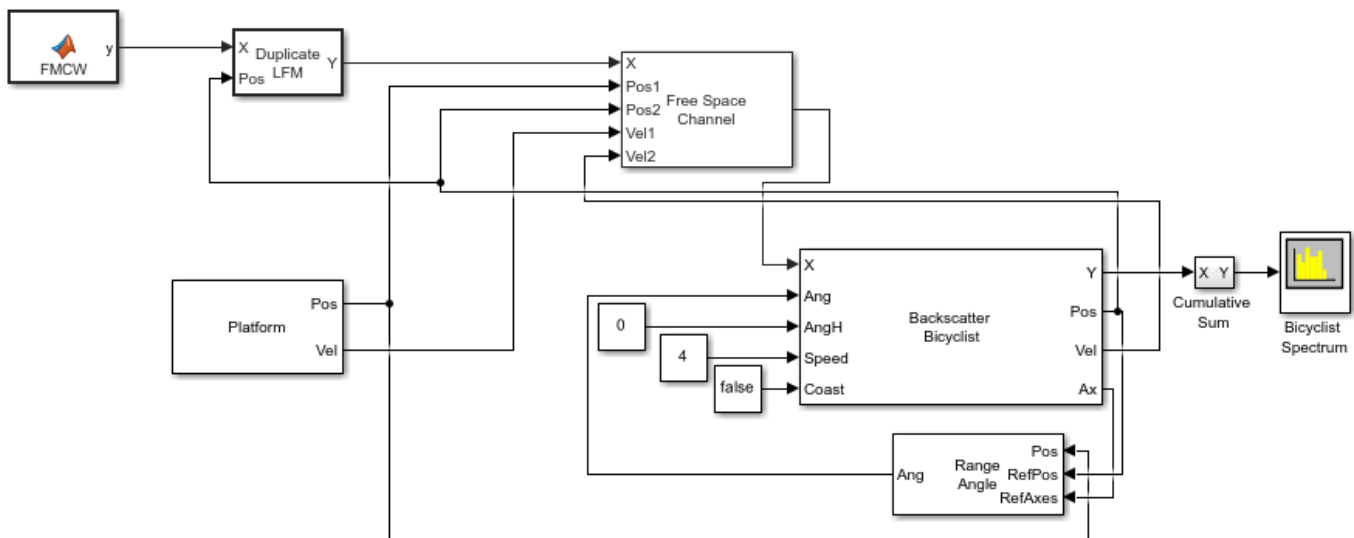

Copyright 2020-2021 The Mathworks Inc.
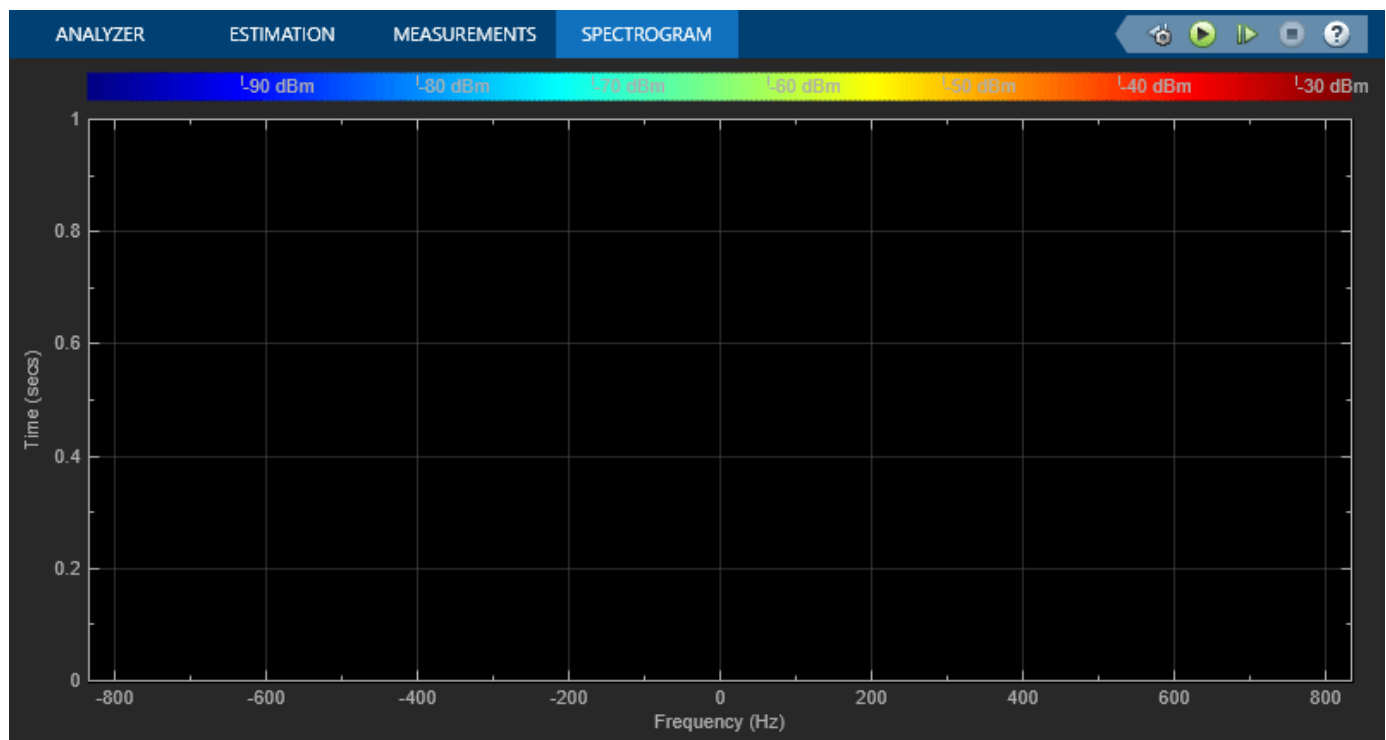
Display the micro-Doppler spectrum.

```
sim('BicyclistMicrospectrumExample');
```
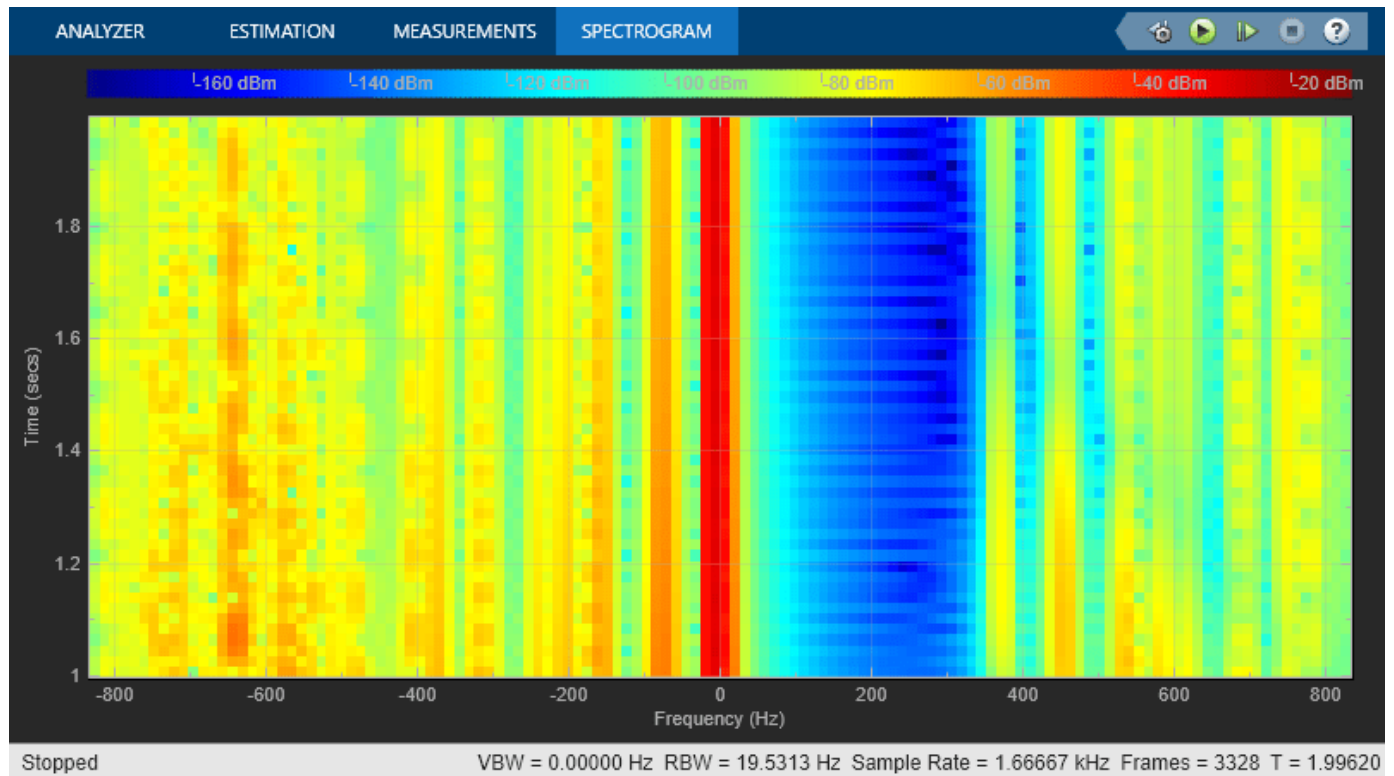


```
close_system('BicyclistMicrospectrumExample');
```

Radar Performance Analysis over Terrain

The performance of a radar system is highly dependent on the environment in which it operates. While the free space spreading loss might be such that the target signal-to-noise ratio (SNR) does not satisfy the minimum detectability threshold for the desired probability of detection and probability of false alarm, the detectability of a target might further suffer in some terrains as there might be no direct, unobstructed line-of-sight from the radar to the target. As you will see in this example, as the elevation of a target increases above the terrain, the radar has a better chance of detecting the target.

In this example, you will learn how to analyze the performance of a ground-based, long-range terminal airport surveillance radar tasked with detecting an aircraft in the presence of heavy, mountainous clutter. The example first defines the radar system operating characteristics and its global position. It then defines the target and its trajectory. Finally, the detectability of the target as it moves through its trajectory is presented along with detailed visualizations.

This example requires the Mapping Toolbox™.

Define Radar

To start, specify a C-band long-range, terminal airport surveillance radar with the following parameters:

- Peak power: 1 kW

- Operating frequency: 6 GHz

- Transmit and receive antenna: 2 degrees in azimuth, 5 degrees in elevation

- Pulse width: 1 µs

```
rdrppower = 1e3;              % Peak power (W)
fc = 6e9;                     % Operating frequency (Hz)
hpbw = [2; 5];                % Half-power beamwidth [azimuth; elevation] (deg)
rdrpulsew = 1e-6;             % Pulse width (s)
lambda = freq2wavelen(fc);   % Wavelength (m)
```

Convert the transmitter half-power beamwidth (HPBW) values to gain using the `beamwidth2gain` function. Assume a cosine rectangular aperture, which is a good approximation for a real-world antenna.

```
rdrgain = beamwidth2gain(hpbw,'CosineRectangular'); % Transmitter and receiver gain (dB)
```

Define the radar ground location as the Rocky Mountain Metropolitan Airport in Broomfield, Colorado, USA. The radar is mounted on a tower 10 meters above the ground. The radar altitude is the sum of the ground elevation and the radar tower height referenced to the mean sea level (MSL).

```
rdrlat = 39.913756;          % Radar latitude (deg)
rdrlon = -105.118062;        % Radar longitude (deg)
rdrtowerht = 10;             % Antenna height (m)
rdralt = 1717 + rdrtowerht;  % Radar altitude (m)
```

To visualize the radar location, import the relevant terrain data from the United States Geological Survey (USGS).

```
dtedfile = "n39_w106_3arc_v2.dt1";
attribution = "SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey.";
```

```matlab
[Zterrain,Rterrain] = readgeoraster(dtedfile,"OutputType","double");

% Visualize the location using the geographic globe plot.
addCustomTerrain("southboulder",dtedfile,"Attribution",attribution);
fig = uifigure;
g = geoglobe(fig,"Terrain","southboulder");
hold(g,"on")
h_rdrtraj = geoplot3(g,rdrlat,rdrlon,rdralt,"ro","LineWidth",6,"MarkerSize",10);
```
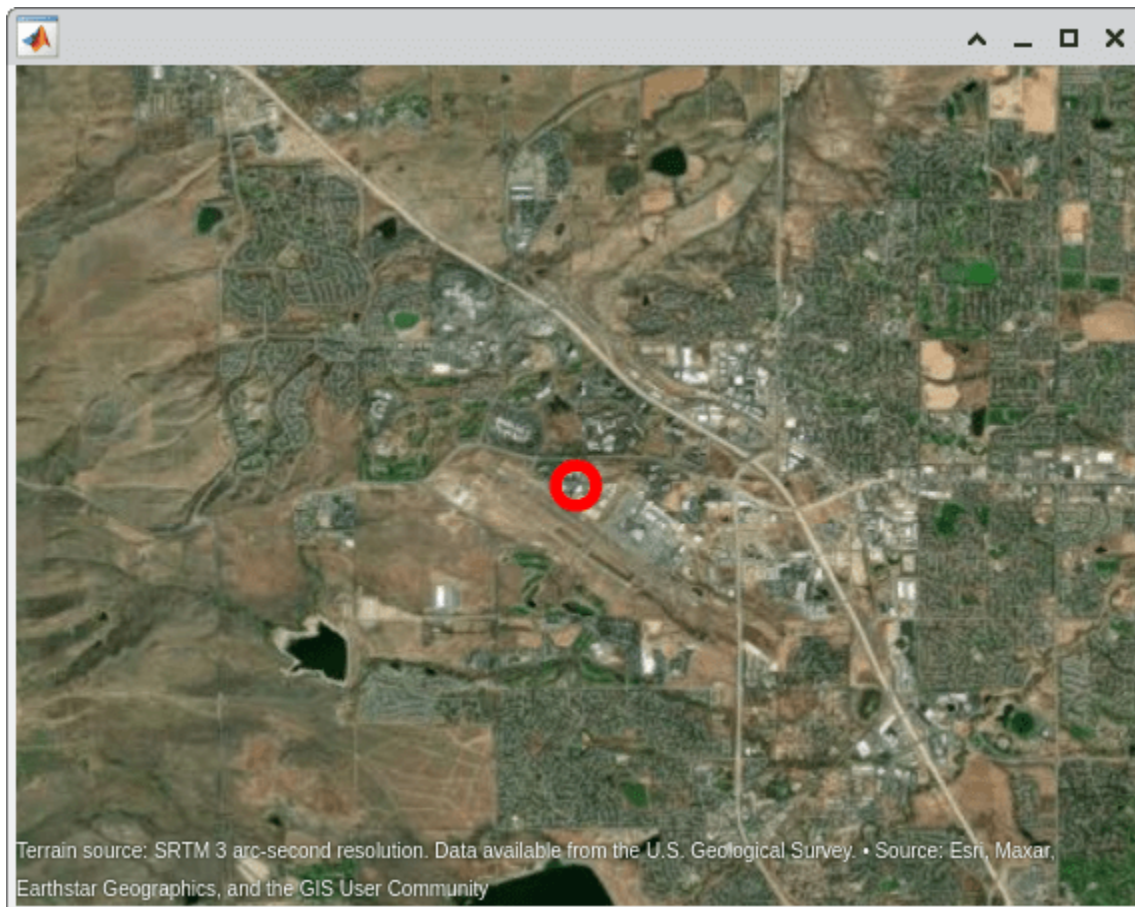

Note that the limits for the file correspond to the region around Boulder, Colorado, USA, and the resolution corresponds to DTED level-1, which has a sample resolution of 3 arc seconds or about 90 meters.

Define Target

Consider a large commercial aircraft as the target. Assume the aircraft trajectory is a corkscrew maneuver, where the aircraft descends rapidly in a spiral.

```matlab
tlat0 = 39.80384;        % Target initial latitude (deg)
tlon0 = -105.49916;      % Target initial longitude (deg)
tht0 = 3000;             % Target initial height (m)
azs = 1:2:540;           % Target azimuth (deg)
r = 5000;                % Target slant range (m)

% Convert from polar coordinates to Cartesian East, North, Up (ENU).
```

```
[X,Y] = pol2cart(deg2rad(azs),r);

% Convert ENU to geodetic.
Z = linspace(0,1000,numel(azs));
wgs84 = wgs84Ellipsoid;
[tlat,tlon,tht] = enu2geodetic(X,Y,Z,tlat0,tlon0,tht0,wgs84);

% Define the target altitude.
talt = tht - egm96geoid(tlat,tlon); % Target altitude (m)
```

For simplicity, assume the waypoints are obtained at a constant sampling rate of 0.1 Hz. The trajectory can be generated using `geoTrajectory` with positions specified as latitude, longitude, and altitude.

```
fs = 0.1;
t = (0:length(X)-1)/fs;
ttraj = geoTrajectory([tlat.' tlon.' talt.'],t,'SampleRate',fs);
```

Plot the ground truth trajectory over the terrain.

```
h_ttraj = geoplot3(g,tlat,tlon,talt,"yo","LineWidth",3);
campos(g,39.77114,-105.62662,6670)
camheading(g,70)
campitch(g,-12)
```
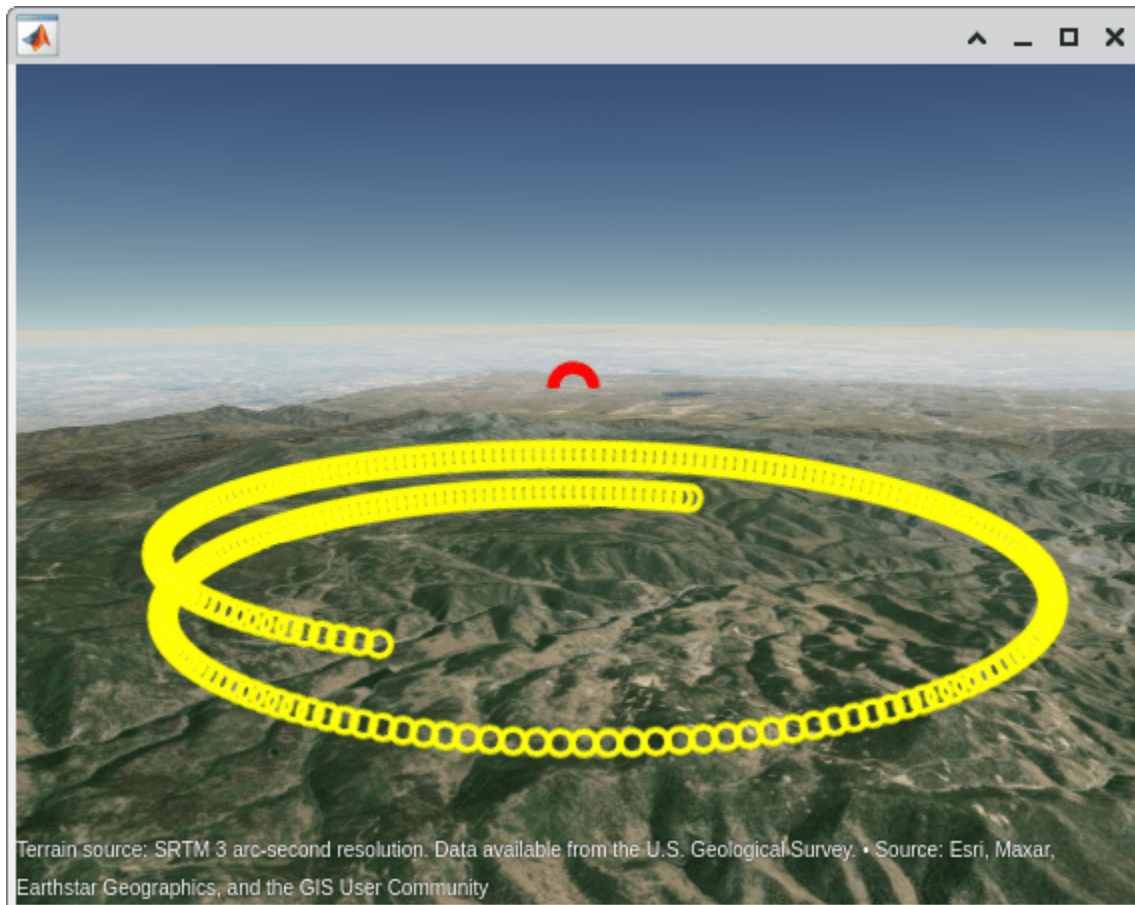
The radar cross section (RCS) for an aircraft is typically from 1 to 10 square meters. For this example, consider the aircraft as an isotropic point target with an RCS of 10 square meters.

```
trcs = pow2db(10);              % Target RCS (dBsm)
```

Simulate the Scenario

Now that the radar and target have been defined, build the scenario, which is composed of a terminal airport radar and large commercial aircraft in the presence of mountainous clutter. Run the simulated scenario for the duration of the aircraft trajectory.

```
scene = radarScenario('IsEarthCentered',true,'UpdateRate',fs,'StopTime',t(end));
rdrplatform = platform(scene,'Position',[rdrlat,rdrlon,rdralt],'Sensor',radarDataGenerator);
tplatform = platform(scene,'Trajectory',ttraj,'Signatures',...
    {rcsSignature('Azimuth',[-180 180],'Elevation',[-90 90],'Pattern',trcs)});
```

The line-of-sight path from the radar to the target is determined for each point in the target trajectory. For locations where the aircraft is not occluded by the terrain, an SNR value is calculated using the radar equation, including the propagation factor along the path.

The propagation factor is calculated using the `radarpropfactor` function. The default permittivity model in `radarpropfactor` is based on a sea permittivity model in Blake's *Machine Plotting of Radar Vertical-Plane Coverage Diagrams*. Such a model is not applicable in this example. Thus, the first step in simulating more realistic propagation is to select a more appropriate permittivity. Use the `earthSurfacePermittivity` function with the vegetation flag. Assume an ambient temperature of 21.1 degrees Celsius, which is about 70 degrees Fahrenheit. Assume a gravimetric water content of 0.3.

```
temp = 21.1;                    % Ambient temperature (degrees Celsius)
gwc = 0.3;                      % Gravimetric water content
[~,~,epsc] = earthSurfacePermittivity('vegetation',fc,temp,gwc);
```

Calculate the propagation factor using the `radarpropfactor` function. Include the following in the calculation:

- Surface permittivity
- Standard deviation of height along the path
- Elevation beamwidth

```
tsnr = -inf(size(t));
F = zeros(size(t));
trange = zeros(size(t));
isVisible = false(size(t));
idx = 1;

while advance(scene)
    tpose = pose(tplatform,'CoordinateSystem','Geodetic');
    tpos = tpose.Position;
    [isVisible(idx),~,~,h] = los2(Zterrain,Rterrain,rdrlat,rdrlon, ...
        tpos(1),tpos(2),rdralt,tpos(3),"MSL","MSL");
    hgtStdDev = std(h);
    if isVisible(idx)
        trange(idx) = llarangeangle([rdrlat rdrlon rdralt],tpos,1);
        F(idx) = radarpropfactor(trange(idx),fc,rdralt,tpos(3), ...
            'SurfaceRelativePermittivity',epsc,...
            'SurfaceHeightStandardDeviation',hgtStdDev, ...
```

```
            'ElevationBeamwidth',hpbw(2));
    end
    idx = idx+1;
end
```

Calculate the SNR along the trajectory.

```
tsnr(isVisible) = radareqsnr(lambda,trange(isVisible).',rdrppower,rdrpulsew,...
    'RCS',trcs,'Gain',rdrgain,'PropagationFactor',F(isVisible).');
```

Next, plot the SNR along the trajectory.

```
tsnr_finiteidx = ~isinf(tsnr);
tsnr_cidx = zeros(size(tsnr));
cmap = colormap(g);
numclvls = size(cmap,1);
tsnr_cidx(tsnr_finiteidx) = discretize(tsnr(tsnr_finiteidx),numclvls-1);
tsnr_cidx(~tsnr_finiteidx) = numclvls;

delete(h_ttraj);
hsnr = zeros(size(tsnr));
for m = 1:numel(tsnr)
    hsnr(m) = geoplot3(g,tlat(m),tlon(m),talt(m),'Marker','o','LineWidth',2,'MarkerSize',1);
    if tsnr_finiteidx(m)
        set(hsnr(m),'Color',cmap(tsnr_cidx(m),:));
    else
        set(hsnr(m),'Color','r');
    end
end
```

Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. • Source: Esri, Maxar, Earthstar Geographics, and the GIS User Community
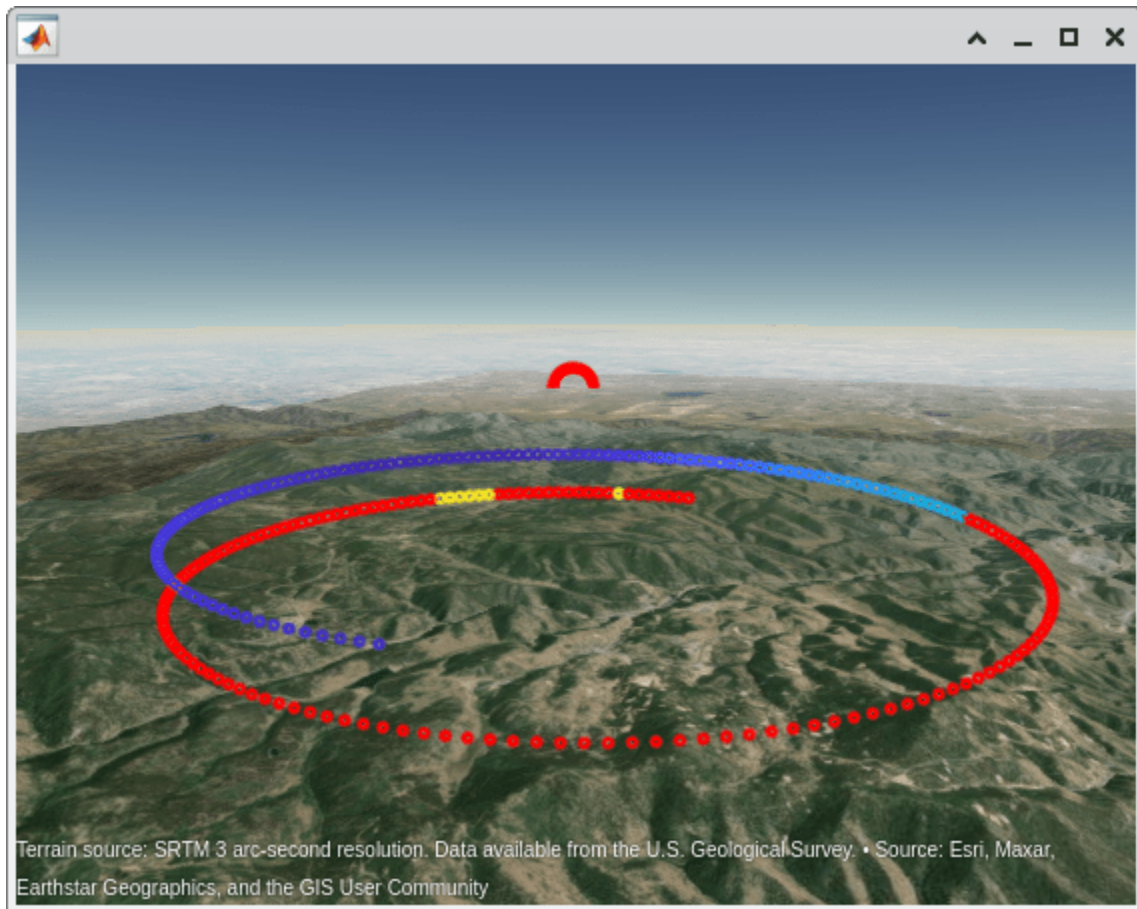
As the aircraft performs the corkscrew maneuver, the SNR of the received signal might vary, as shown in the figure. The radar has an unobstructed view of the aircraft if there is a line-of-sight path. The red portion of the trajectory indicates that there is no line-of-sight path between the aircraft and the radar.

For the surveillance radar, the desired performance index is a probability of detection (Pd) of 0.9 and probability of false alarm (Pfa) below 1e-6. To make the radar system design more feasible, you can use a pulse integration technique to reduce the required SNR. For this system, assume noncoherent integration of 32 pulses. A good approximation of the minimum SNR needed for a detection at the specified Pd and Pfa can be computed by the `detectability` function.

```
pd = 0.9;
pfa = 1e-6;
minsnr_32p = detectability(pd,pfa,32);
isdetectable_32p = tsnr >= minsnr_32p;
```

Observe in which part of the trajectory the target is detectable (shown in green) given the minimum SNR requirement. Note that the existence of a line-of-sight link does not guarantee that the target is detectable.

```
for m = 1:numel(tsnr)
    if isdetectable_32p(m)
        set(hsnr(m),'Color','g');
    else
```

```
        set(hsnr(m),'Color','r');
    end
end
```
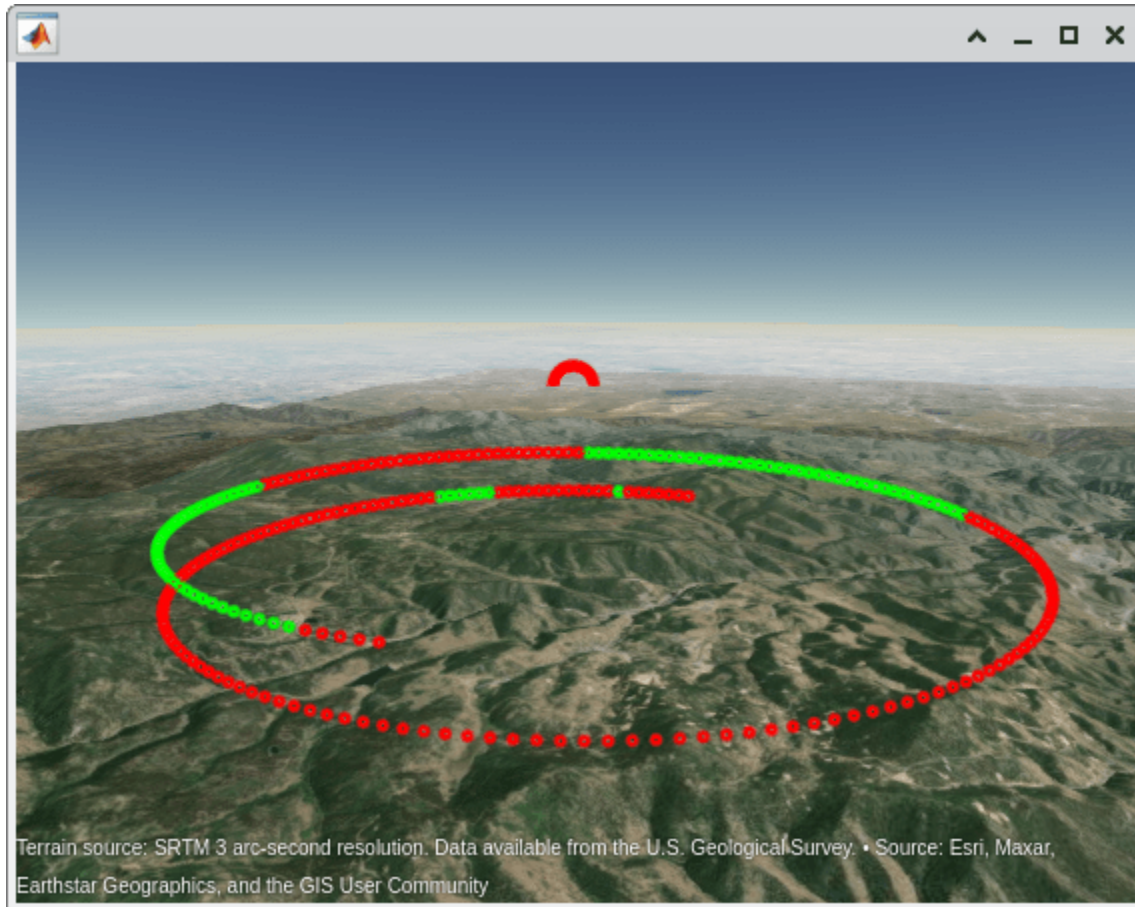

To improve detectability with surveillance radars, often radar engineers discuss maximizing the power-aperture product of the system. This generally translates to increasing the physical size or peak power of a system. It can also be considered in terms of illumination time (i.e., energy on target). Some methods to improve detectability include:

- Increasing the peak power: This can be difficult to achieve due to limitations on the power supply and on the radar platform location. Furthermore, if there is a requirement for low probability of intercept (LPI), increasing peak power is often undesirable.

- Increasing the physical size of the antenna aperture: Increasing the physical size of the antenna results in an increase in the associated gain and decrease in the half-power beamwidth. Limitations of the platform or location might make increasing the physical size of the antenna aperture infeasible. Additionally, with a finer beamwidth, it becomes more important that the antenna beam is steered towards the target under test.

- Increasing the number of pulses to be integrated: This will bring the detectability to a lower value. However, if the aircraft is maneuvering at a high speed, then it might take too long to collect all of the transmitted pulses under the assumption of target stationarity. If the target stationarity assumption is invalid, additional signal processing steps will need to be taken to mitigate the range walk of the target.

- Increasing the average power: Rather than increasing the peak power, one can increase the average power by increasing the duty cycle. Increasing the duty cycle means increasing either the pulse width or the pulse repetition frequency (PRF), which might put undue burden on the radar hardware. The downside to increasing the pulse width is an increase in the minimum range and potentially overlapping, inseparable target returns. On the other hand, increasing the pulse repetition frequency decreases the maximum unambiguous range, which might be undesirable for a long-range surveillance system, especially if it is not performing a disambiguation technique.

The above list, while in no way exhaustive, shows some of the tradeoffs in the design of an terminal airport surveillance system. For this example, increase the peak power. Since this is a ground-based system, increasing the power is not expected to be too burdensome. Additionally, other airport radars like the ASR-9 operate at a peak power of about 1 MW. Since this is an airport radar, there is no need for LPI requirements.

Consider the case where the peak power is increased to 10 kW.

```matlab
rdrppower = 10e3;              % Peak power (W)

% Recalculate the SNR along the trajectory.
tsnr(isVisible) = radareqsnr(lambda,trange(isVisible).',rdrppower,rdrpulsew,...
    'RCS',trcs,'Gain',rdrgain,'PropagationFactor',F(isVisible).');

% Determine the regions of the trajectory that are now detectable given the
% newly updated SNR.
isdetectable_32p = tsnr >= minsnr_32p;
```

Notice that by increasing the peak power, the regions at the end of the trajectory that were previously undetected now satisfy the minimum SNR threshold.

```matlab
for m = 1:numel(tsnr)
    if isdetectable_32p(m)
        set(hsnr(m),'Color','g');
    else
        set(hsnr(m),'Color','r');
    end
end
```
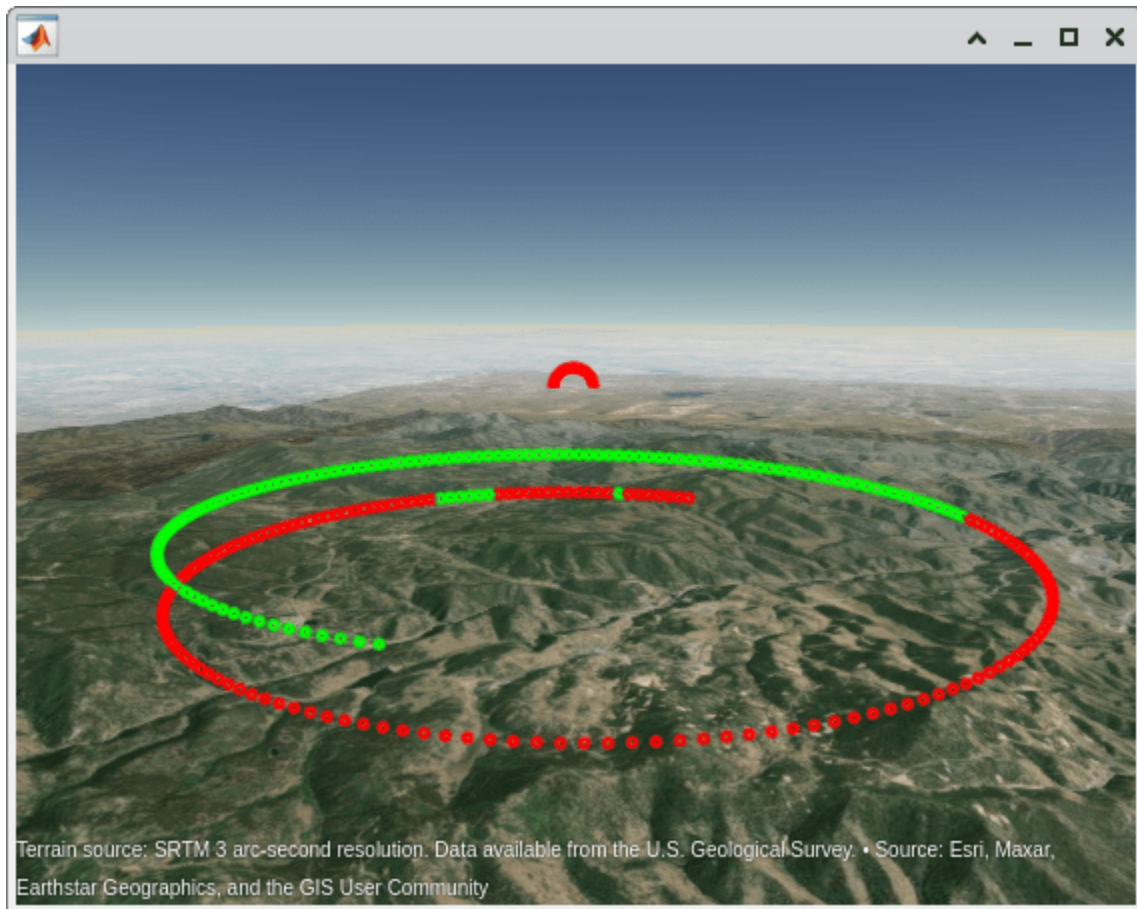
Summary

In this example, SNR is calculated and visualized for a ground-based, long-range terminal airport surveillance radar tasked with detecting an aircraft in the presence of heavy, mountainous clutter. The example shows how to calculate the line-of-sight given a target trajectory. It also shows that the existence of a line-of-sight link does not necessarily guarantee that the target is detectable. This example considers some design tradeoffs to improve target detectability, discussing how radar parameters can be modified to match other system requirements. The example can be easily extended for other aircraft, different trajectory sets, and different terrain maps.

```
% Clean up by closing the geographic globe and removing the imported
% terrain data.
if isvalid(fig)
    close(fig)
end
removeCustomTerrain("southboulder")
```

Create Physics-Based Radar Model from Statistical Model

This example shows how to programmatically create a physics-based radar model from a statistical radar model.

A radar is a perception system that uses an antenna or antenna array to capture RF energy, which is then downconverted and processed to provide information on objects in the radar's field of view. The received signal needs to pass through both a signal processing subsystem and a data processing subsystem.

The goal of the signal processing subsystem is to translate received IQ signals to target detections. The data processing subsystem takes those detections and produces tracks corresponding to the detected targets.

The signal processing subsystem helps to generate a snapshot of the scene at the current time and includes information on whether an object is in the coverage and, if so, where it is. The data processing subsystem links those snapshots together so operators can understand what happened over time. This helps obtain Doppler information in addition to predictions of where targets are heading.

Radar engineers that simulate and model algorithms and systems need to work across a range of abstraction levels that span the signal and data processing domains. The level of abstraction depends on the phase of the radar development life cycle, the length of the scene being simulated, and the type of engineering work being performed.

At early stages of a project, as design tradeoffs are being explored, modeling at the radar equation level may be adequate. As a project progresses, it will be necessary to increase the level of model fidelity, moving from a statistical level to signal level simulation. In addition, the length of a scenario can dictate which modeling abstraction level makes sense. For example, for longer scenario times (seconds, minutes, or longer), it may be better to generate statistical or probabilistic radar detections and tracks to cover a mission or to test tracking and sensor fusion algorithms. Alternatively, higher fidelity, physics-based simulations that include transmitted waveforms, signal propagation through the environment, reflections off targets, and received signals at a receive array are needed for events of interest or for when signal processing algorithms are being developed.

In this example, a scene is created with both radar and targets. First, detections are generated using a statistical model based on the radar equation. Next, an equivalent physics-based radar model is created from the statistical model. The physics-based radar model is then used to simulate the IQ signal and generate the detections. The example shows that the detections from the two models are consistent.

Define the Scene

To start, define a scenario with a fixed-location surveillance radar. The radar has three targets in its field of view. Plot the locations of the radar and targets.

```
% Create targets

tgt1 = struct( ...
    'PlatformID', 1, ...
    'Position', [0 -50e3 -1e3], ...
    'Velocity', [0 900*1e3/3600 0]);
```

```
tgt2 = struct( ...
    'PlatformID', 2, ...
    'Position', [20e3 0 -500], ...
    'Velocity', [700*1e3/3600 0 0]);

tgt3 = struct( ...
    'PlatformID', 3, ...
    'Position', [-20e3 0 -500], ...
    'Velocity', [300*1e3/3600 0 0]);

tp = theaterPlot('XLim',[-30e3 30e3],'YLim',[-60e3 10e3],'ZLim',[-10e3 1e3]);
gtplot = platformPlotter(tp,'DisplayName','Target Ground Truth',...
    'Marker','^','MarkerSize',8,'MarkerFaceColor','r');
plotPlatform(gtplot,[tgt1.Position;tgt2.Position;tgt3.Position],...
    [tgt1.Velocity;tgt2.Velocity;tgt3.Velocity],{'Target1','Target2','Target3'});
```
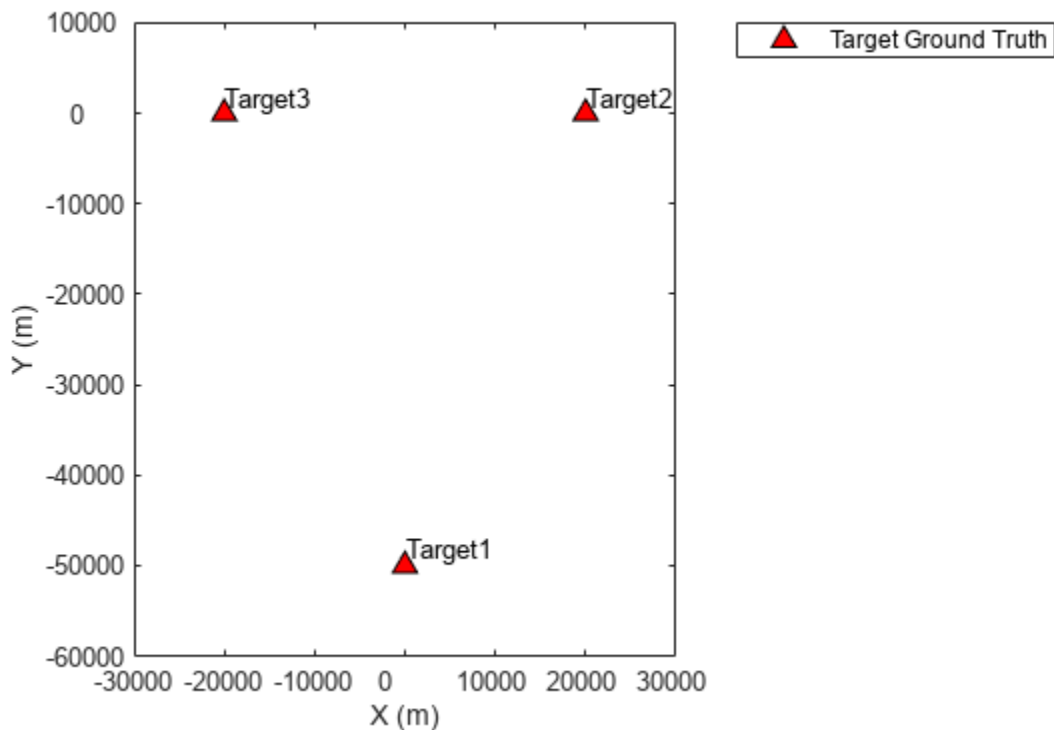

Define Radar for Detection Generation

Next, define an airport surveillance radar that generates detections from a statistical model. The airport surveillance radar is installed 15 meters above the ground. The radar sensor definition includes the key radar parameters such as the scanning type and field of view information.

`radarDataGenerator` generates detections statistically based on the radar equation.

```
rpm = 12.5;
fov = [1.4;5]; % [azimuth; elevation]

scanrate = rpm*360/60;  % deg/s
```

```
updaterate = scanrate/fov(1); % Hz

sensor = radarDataGenerator(1, 'Rotator', ...
    'DetectionProbability', 0.99, ...
    'UpdateRate', updaterate, ...
    'MountingLocation', [0 0 -15], ...
    'MaxAzimuthScanRate', scanrate, ...
    'FieldOfView', fov, ...
    'AzimuthResolution', fov(1));

radarPosition = [0 0 0];
radarVelocity = [0 0 0];
radarplot = platformPlotter(tp,'DisplayName','Radar',...
    'Marker','s','MarkerSize',8,'MarkerFaceColor','b');
plotPlatform(radarplot,radarPosition,radarVelocity,{'Radar'})
```
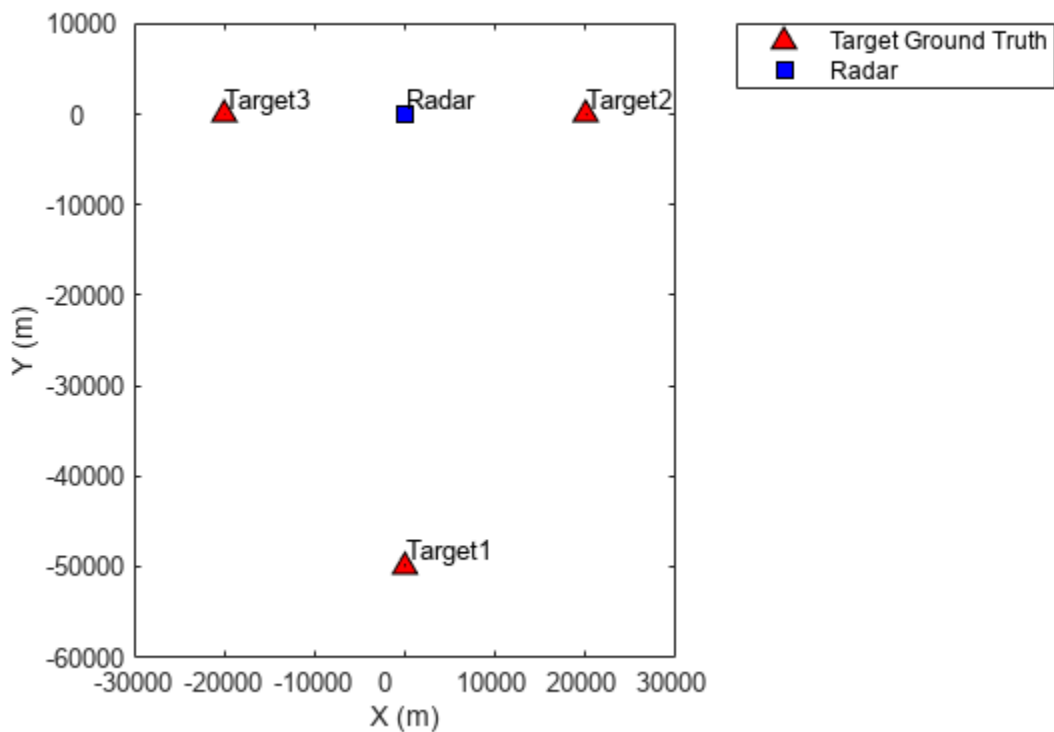

Generate Statistical Radar Detection

Generate detections from a full scan of the radar and plot the detections in the scene.

```
% Generate detections from a full scan of the radar
simTime = 0;
detBuffer = {};

rng(2020);
while true
    [dets, numDets, config] = sensor([tgt1 tgt2 tgt3], simTime);
    detBuffer = [detBuffer;dets]; %#ok<AGROW>
```

```
    % Is full scan complete?
    if config.IsScanDone
        break % yes
    end
    simTime = simTime+1/sensor.UpdateRate;
end

stadetpos = zeros(numel(detBuffer),3);
for m = 1:numel(detBuffer)
    stadetpos(m,:) = detBuffer{m}.Measurement.';
end
stadet = detectionPlotter(tp,'DisplayName','Statistical Detection',...
    'Marker','d','MarkerSize',6,'MarkerFaceColor','g');
plotDetection(stadet,stadetpos)
```
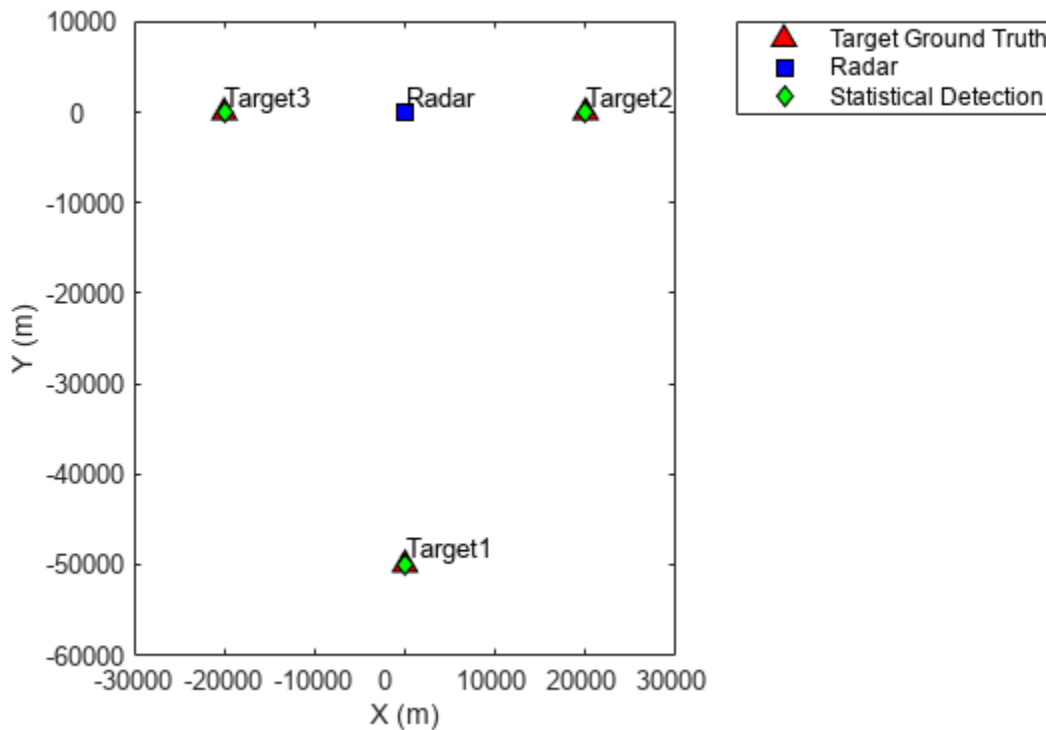

The plot shows that the generated detections match the ground truth target locations. The three targets all indicated by a truth marker have a detection that is shown as an overlay on the truth marker.

Define Radar for IQ Signal Generation and Processing

Because the statistical simulation is satisfactory, you can now perform IQ signal simulation to verify if the signal processing algorithms are working properly. Create a radar transceiver that produces the IQ signal based on the statistical sensor configured earlier.

```
sensor_iq = radarTransceiver(sensor)
```

```
sensor_iq =
  radarTransceiver with properties:

                          Waveform: [1x1 phased.RectangularWaveform]
                       Transmitter: [1x1 phased.Transmitter]
                   TransmitAntenna: [1x1 phased.Radiator]
                    ReceiveAntenna: [1x1 phased.Collector]
                          Receiver: [1x1 phased.ReceiverPreamp]
                 MechanicalScanMode: 'Circular'
          InitialMechanicalScanAngle: -0.1000
                 MechanicalScanRate: 75
                 ElectronicScanMode: 'None'
                   MountingLocation: [0 0 -15]
                     MountingAngles: [0 0 0]
               NumRepetitionsSource: 'Property'
                     NumRepetitions: 1
                  RangeLimitsSource: 'Property'
                        RangeLimits: [0 100000]
                     RangeOutputPort: false
                      TimeOutputPort: false
```

Notice that the configuration of the `sensor_iq` variable is closer to a physical system. `sensor_iq` produces IQ signals that you can then process. For this example, implement a simple threshold detector to generate detections.

```matlab
% configure signal processing component

coeff = getMatchedFilter(sensor_iq.Waveform);
mf = phased.MatchedFilter('Coefficients',coeff,'GainOutputPort',true);

npower = noisepow(1/sensor_iq.Waveform.PulseWidth,...
    sensor_iq.Receiver.NoiseFigure,sensor_iq.Receiver.ReferenceTemperature);
threshold = npower * db2pow(npwgnthresh(sensor.FalseAlarmRate));

fs = sensor_iq.Waveform.SampleRate;
prf = sensor_iq.Waveform.PRF;
c = physconst('lightspeed');
fc = sensor_iq.TransmitAntenna.OperatingFrequency;
lambda = c/fc;
Nsamp = round(fs/prf);
rgates = (0:Nsamp-1)/fs*c/2;
tvg = phased.TimeVaryingGain(...
    'RangeLoss',2*fspl(rgates,lambda),...
    'ReferenceLoss',2*fspl(Nsamp/fs*c/2,lambda));
```

IQ Signal and Processing Simulation

Next, perform IQ simulation and check if the processing algorithm yields a result similar to that of the statistical sensor. Notice that the simulation loop that generates the IQ signal is almost identical to the loop that generates the statistical detection. The loop also shows how to process the IQ signal to get the detection.

```matlab
simTime = 0;
detBuffer_iq = {};

while true
```

```
    [sig, config] = sensor_iq([tgt1 tgt2 tgt3], simTime);
    if config.IsScanDone
        break
    end

    % Processing
    [sigp,Gmf] = mf(sig);
    sigp = tvg(sigp);

    th = sqrt(threshold*db2pow(Gmf));
    ind = abs(sigp)>th;
    if any(ind)
        [~,idx] = max(abs(sigp));
        rng_est = rgates(idx);
        meas_sensor = [0;0;rng_est];
        meas_body = local2globalcoord(meas_sensor,'sr',...
            config.OriginPosition,config.Orientation);
        dets_iq = struct('Time',simTime,'Measurement',meas_body);
        detBuffer_iq = [detBuffer_iq;dets_iq]; %#ok<AGROW>
    end

    simTime = simTime+1/updaterate;
end

iqdetpos = zeros(numel(detBuffer_iq),3);
for m = 1:numel(detBuffer_iq)
    iqdetpos(m,:) = detBuffer_iq{m}.Measurement.';
end
iqdet = detectionPlotter(tp,'DisplayName','IQ Detection',...
    'Marker','o','MarkerSize',10,'MarkerEdgeColor','k');
plotDetection(iqdet,iqdetpos)
```
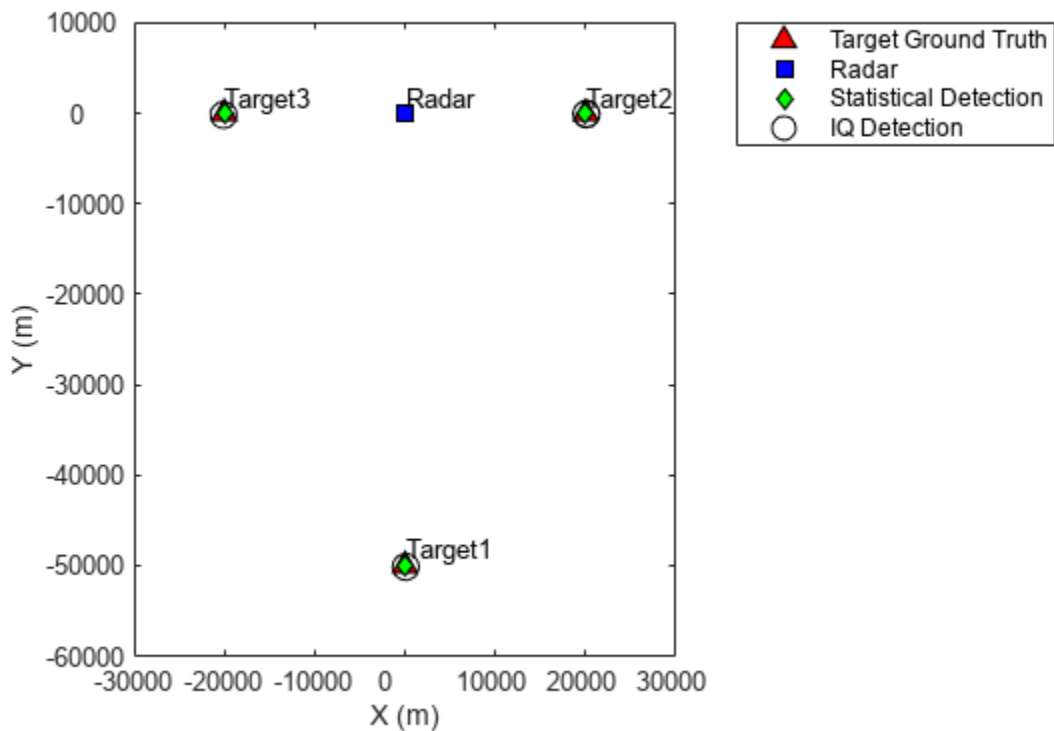
The plot clearly indicates that the result obtained from IQ signal generation is similar to the result generated from the statistical model.

Summary

In this example, a statistical model is used to generate radar detections based on the radar equation. Then a physics-based radar model is programmatically created from the statistical model, and a new set of detections are derived from the IQ signal generated from this physics-based radar model. The detections from both models match the ground truth well. This workflow is a very convenient way to get a signal level model up and running quickly. Once the basic signal model is in place, it can be extended as the project dictates.

Doppler Estimation

This example shows a monostatic pulse radar detecting the radial velocity of moving targets at specific ranges. The speed is derived from the Doppler shift caused by the moving targets. We first identify the existence of a target at a given range and then use Doppler processing to determine the radial velocity of the target at that range.

Radar System Setup

First, we define a radar system. Since the focus of this example is on Doppler processing, we use the radar system built in the example "Simulating Test Signals for a Radar Receiver". Readers are encouraged to explore the details of radar system design through that example.

```
load BasicMonostaticRadarExampleData;
```

System Simulation

Targets

Doppler processing exploits the Doppler shift caused by the moving target. We now define three targets by specifying their positions, radar cross sections (RCS), and velocities.

```
tgtpos = [[1200; 1600; 0],[3543.63; 0; 0],[1600; 0; 1200]];
tgtvel = [[60; 80; 0],[0;0;0],[0; 100; 0]];
tgtmotion = phased.Platform('InitialPosition',tgtpos,'Velocity',tgtvel);

tgtrcs = [1.3 1.7 2.1];
fc = radiator.OperatingFrequency;
target = phased.RadarTarget('MeanRCS',tgtrcs,'OperatingFrequency',fc);
```

Note that the first and third targets are both located at a range of 2000 m and are both traveling at a speed of 100 m/s. The difference is that the first target is moving along the radial direction, while the third target is moving in the tangential direction. The second target is not moving.

Environment

We also need to setup the propagation environment for each target. Since we are using a monostatic radar, we use the two way propagation model.

```
fs = waveform.SampleRate;

channel = phased.FreeSpace(...
    'SampleRate',fs,...
    'TwoWayPropagation',true,...
    'OperatingFrequency',fc);
```

Signal Synthesis

With the radar system, the environment, and the targets defined, we can now simulate the received signal as echoes reflected from the targets. The simulated data is a data matrix with the fast time (time within each pulse) along each column and the slow time (time between pulses) along each row.

We need to set the seed for noise generation at the receiver so that we can reproduce the same results.

```
receiver.SeedSource = 'Property';
receiver.Seed = 2009;

prf = waveform.PRF;
num_pulse_int = 10;

fast_time_grid = unigrid(0,1/fs,1/prf,'[)');
slow_time_grid = (0:num_pulse_int-1)/prf;

% Pre-allocate array for improved processing speed
rxpulses = zeros(numel(fast_time_grid),num_pulse_int);

for m = 1:num_pulse_int

    % Update sensor and target positions
    [sensorpos,sensorvel] = sensormotion(1/prf);
    [tgtpos,tgtvel] = tgtmotion(1/prf);

    % Calculate the target angles as seen by the sensor
    [tgtrng,tgtang] = rangeangle(tgtpos,sensorpos);

    % Simulate propagation of pulse in direction of targets
    pulse = waveform();
    [txsig,txstatus] = transmitter(pulse);
    txsig = radiator(txsig,tgtang);
    txsig = channel(txsig,sensorpos,tgtpos,sensorvel,tgtvel);

    % Reflect pulse off of targets
    tgtsig = target(txsig);

    % Receive target returns at sensor
    rxsig = collector(tgtsig,tgtang);
    rxpulses(:,m) = receiver(rxsig,~(txstatus>0));
end
```

Doppler Estimation

Range Detection

To be able to estimate the Doppler shift of the targets, we first need to locate the targets through range detection. Because the Doppler shift spreads the signal power into both I and Q channels, we need to rely on the signal energy to do the detection. This means that we use noncoherent detection schemes.

The detection process is described in detail in the aforementioned example so we simply perform the necessary steps here to estimate the target ranges.

```
% calculate initial threshold
pfa = 1e-6;
% in loaded system, noise bandwidth is half of the sample rate
noise_bw = receiver.SampleRate/2;
npower = noisepow(noise_bw,...
    receiver.NoiseFigure,receiver.ReferenceTemperature);
threshold = npower * db2pow(npwgnthresh(pfa,num_pulse_int,'noncoherent'));

% apply matched filter and update the threshold
matchingcoeff = getMatchedFilter(waveform);
matchedfilter = phased.MatchedFilter(...
```

```
    'Coefficients',matchingcoeff,...
    'GainOutputPort',true);
[rxpulses, mfgain] = matchedfilter(rxpulses);
threshold = threshold * db2pow(mfgain);

% compensate the matched filter delay
matchingdelay = size(matchingcoeff,1)-1;
rxpulses = buffer(rxpulses(matchingdelay+1:end),size(rxpulses,1));

% apply time varying gain to compensate the range dependent loss
prop_speed = radiator.PropagationSpeed;
range_gates = prop_speed*fast_time_grid/2;
lambda = prop_speed/fc;

tvg = phased.TimeVaryingGain(...
    'RangeLoss',2*fspl(range_gates,lambda),...
    'ReferenceLoss',2*fspl(prop_speed/(prf*2),lambda));

rxpulses = tvg(rxpulses);

% detect peaks from the integrated pulse
[~,range_detect] = findpeaks(pulsint(rxpulses,'noncoherent'),...
    'MinPeakHeight',sqrt(threshold));
range_estimates = round(range_gates(range_detect))
```

range_estimates = *1×2*

```
      2000        3550
```

These estimates suggest the presence of targets in the range of 2000 m and 3550 m.

Doppler Spectrum

Once we successfully estimated the ranges of the targets, we can then estimate the Doppler information for each target.

Doppler estimation is essentially a spectrum estimation process. Therefore, the first step in Doppler processing is to generate the Doppler spectrum from the received signal.

The received signal after the matched filter is a matrix whose columns correspond to received pulses. Unlike range estimation, Doppler processing processes the data across the pulses (slow time), which is along the rows of the data matrix. Since we are using 10 pulses, there are 10 samples available for Doppler processing. Because there is one sample from each pulse, the sampling frequency for the Doppler samples is the pulse repetition frequency (PRF).

As predicted by the Fourier theory, the maximum unambiguous Doppler shift a pulse radar system can detect is half of its PRF. This also translates to the maximum unambiguous speed a radar system can detect. In addition, the number of pulses determines the resolution in the Doppler spectrum, which determines the resolution of the speed estimates.

```
max_speed = dop2speed(prf/2,lambda)/2
```

max_speed = 224.6888

```
speed_res = 2*max_speed/num_pulse_int
```

speed_res = 44.9378

As shown in the calculation above, in this example, the maximum detectable speed is 225m/s, either approaching (-225) or departing (+225). The resulting Doppler resolution is about 45 m/s, which means that the two speeds must be at least 45 m/s apart to be separable in the Doppler spectrum. To improve the ability to discriminate between different target speeds, more pulses are needed. However, the number of pulses available is also limited by the radial velocity of the target. Since the Doppler processing is limited to a given range, all pulses used in the processing have to be collected before the target moves from one range bin to the next.

Because the number of Doppler samples are in general limited, it is common to zero pad the sequence to interpolate the resulting spectrum. This will not improve the resolution of the resulting spectrum, but can improve the estimation of the locations of the peaks in the spectrum.

The Doppler spectrum can be generated using a periodogram. We zero pad the slow time sequence to 256 points.

```
num_range_detected = numel(range_estimates);
[p1, f1] = periodogram(rxpulses(range_detect(1),:).',[],256,prf, ...
                'power','centered');
[p2, f2] = periodogram(rxpulses(range_detect(2),:).',[],256,prf, ...
                'power','centered');
```

The speed corresponding to each sample in the spectrum can then be calculated. Note that we need to take into consideration of the round trip effect.

```
speed_vec = dop2speed(f1,lambda)/2;
```

Doppler Estimation

To estimate the Doppler shift associated with each target, we need to find the locations of the peaks in each Doppler spectrum. In this example, the targets are present at two different ranges, so the estimation process needs to be repeated for each range.

Let's first plot the Doppler spectrum corresponding to the range of 2000 meters.

```
periodogram(rxpulses(range_detect(1),:).',[],256,prf,'power','centered');
```

Periodogram Power Spectrum Estimate
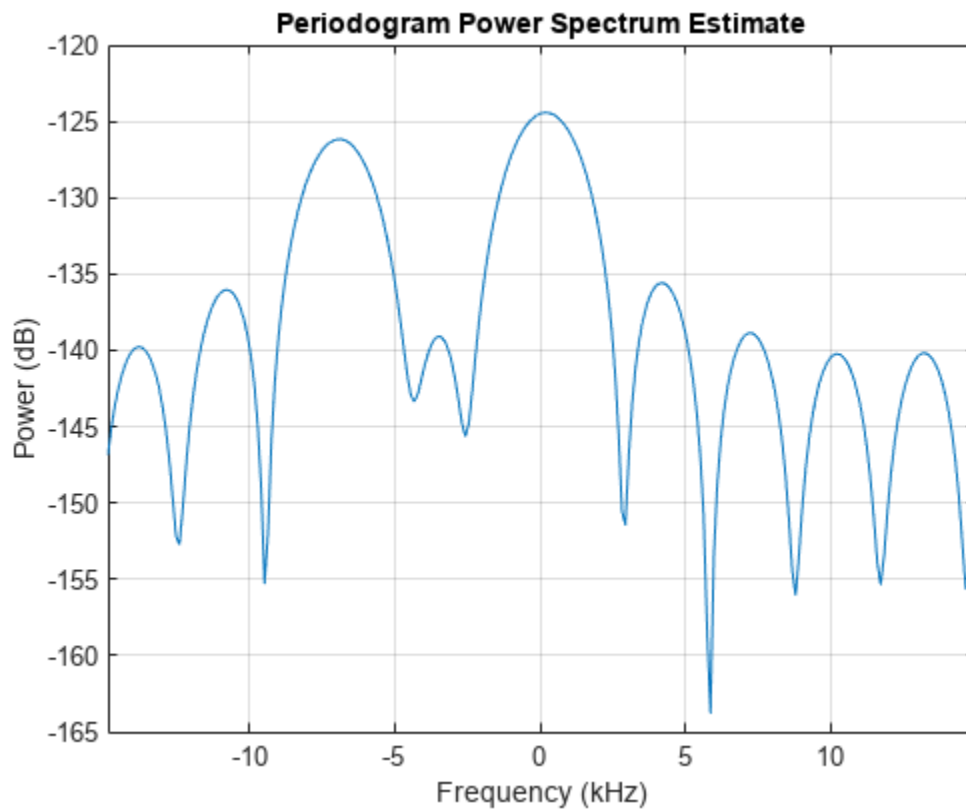
Note that we are only interested in detecting the peaks, so the spectrum values themselves are not critical. From the plot of Doppler spectrum, we notice that 5 dB below the maximum peak is a good threshold. Therefore, we use -5 as our threshold on the normalized Doppler spectrum.

```
spectrum_data = p1/max(p1);
[~,dop_detect1] = findpeaks(pow2db(spectrum_data),'MinPeakHeight',-5);
sp1 = speed_vec(dop_detect1)
```

sp1 = *2×1*

```
 -103.5675
    3.5108
```

The results show that there are two targets at the 2000 m range: one with a velocity of 3.5 m/s and the other with -104 m/s. The value -104 m/s can be easily associated with the first target, since the first target is departing at a radial velocity of 100 m/s, which, given the Doppler resolution of this example, is very close to the estimated value. The value 3.5 m/s requires more explanation. Since the third target is moving along the tangential direction, there is no velocity component in the radial direction. Therefore, the radar cannot detect the Doppler shift of the third target. The true radial velocity of the third target, hence, is 0 m/s and the estimate of 3.5 m/s is very close to the true value.

Note that these two targets cannot be discerned using only range estimation because their range values are the same.

The same operations are then applied to the data corresponding to the range of 3550 meters.

```
periodogram(rxpulses(range_detect(2),:).',[],256,prf,'power','centered');
```


Periodogram Power Spectrum Estimate
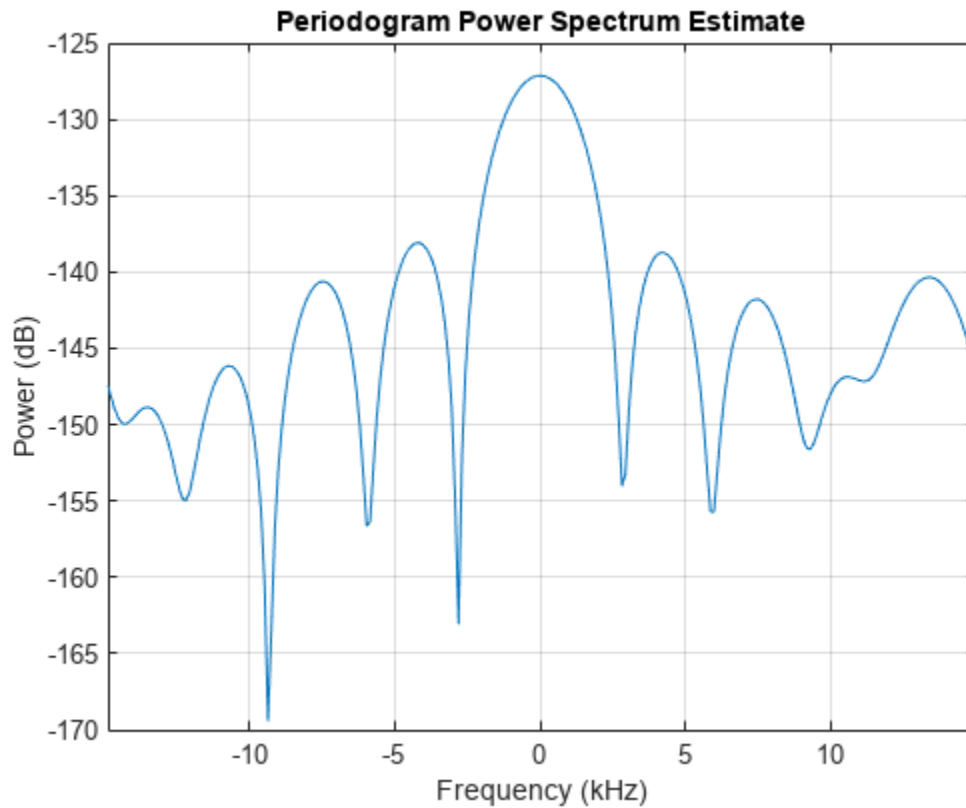
```
spectrum_data = p2/max(p2);
[~,dop_detect2] = findpeaks(pow2db(spectrum_data),'MinPeakHeight',-5);
sp2 = speed_vec(dop_detect2)
```

```
sp2 = 0
```

This result shows an estimated speed of 0 m/s, which matches the fact that the target at this range is not moving.

Summary

This example showed a simple way to estimate the radial speed of moving targets using a pulse radar system. We generated the Doppler spectrum from the received signal and estimated the peak locations from the spectrum. These peak locations correspond to the target's radial speed. The limitations of the Doppler processing are also discussed in the example.

Constant False Alarm Rate (CFAR) Detection

This example introduces constant false alarm rate (CFAR) detection and shows how to use CFARDetector and CFARDetector2D in the Phased Array System Toolbox™ to perform cell averaging CFAR detection.

Introduction

One important task a radar system performs is target detection. The detection itself is fairly straightforward. It compares the signal to a threshold. Therefore, the real work on detection is coming up with an appropriate threshold. In general, the threshold is a function of both the probability of detection and the probability of false alarm.

In many phased array systems, because of the cost associated with a false detection, it is desirable to have a detection threshold that not only maximizes the probability of detection but also keeps the probability of false alarm below a preset level.

There is extensive literature on how to determine the detection threshold. Readers might be interested in the "Signal Detection in White Gaussian Noise" and "Signal Detection Using Multiple Samples" examples for some well known results. However, all these classical results are based on theoretical probabilities and are limited to white Gaussian noise with known variance (power). In real applications, the noise is often colored and its power is unknown.

CFAR technology addresses these issues. In CFAR, when the detection is needed for a given cell, often termed as the cell under test (CUT), the noise power is estimated from neighboring cells. Then the detection threshold, T, is given by

$$T = \alpha P_n$$

where P_n is the noise power estimate and α is a scaling factor called the threshold factor.

From the equation, it is clear that the threshold adapts to the data. It can be shown that with the appropriate threshold factor, α, the resulting probability of false alarm can be kept at a constant, hence the name CFAR.

Cell Averaging CFAR Detection

The cell averaging CFAR detector is probably the most widely used CFAR detector. It is also used as a baseline comparison for other CFAR techniques. In a cell averaging CFAR detector, noise samples are extracted from both leading and lagging cells (called training cells) around the CUT. The noise estimate can be computed as [1]

$$P_n = \frac{1}{N} \sum_{m=1}^{N} x_m$$

where N is the number of training cells and x_m is the sample in each training cell. If x_m happens to be the output of a square law detector, then P_n represents the estimated noise power. In general, the number of leading and lagging training cells are the same. Guard cells are placed adjacent to the CUT, both leading and lagging it. The purpose of these guard cells is to avoid signal components from leaking into the training cell, which could adversely affect the noise estimate.

The following figure shows the relation among these cells for the 1-D case.
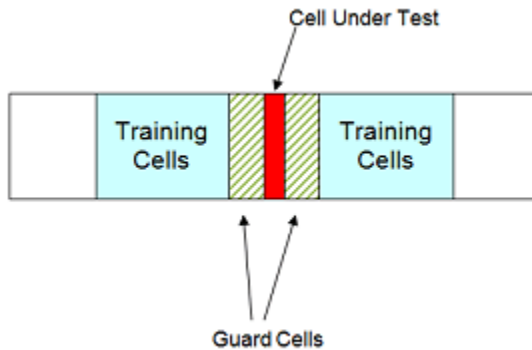
With the above cell averaging CFAR detector, assuming the data passed into the detector is from a single pulse, i.e., no pulse integration involved, the threshold factor can be written as [1]

$$\alpha = N(P_{fa}^{-1/N} - 1)$$

where P_{fa} is the desired false alarm rate.

CFAR Detection Using Automatic Threshold Factor

In the rest of this example, we show how to use Phased Array System Toolbox to perform a cell averaging CFAR detection. For simplicity and without losing any generality, we still assume that the noise is white Gaussian. This enables the comparison between the CFAR and classical detection theory.

We can instantiate a CFAR detector using the following command:

```
cfar = phased.CFARDetector('NumTrainingCells',20,'NumGuardCells',2);
```

In this detector we use 20 training cells and 2 guard cells in total. This means that there are 10 training cells and 1 guard cell on each side of the CUT. As mentioned above, if we assume that the signal is from a square law detector with no pulse integration, the threshold can be calculated based on the number of training cells and the desired probability of false alarm. Assuming the desired false alarm rate is 0.001, we can configure the CFAR detector as follows so that this calculation can be carried out.

```
exp_pfa = 1e-3;
cfar.ThresholdFactor = 'Auto';
cfar.ProbabilityFalseAlarm = exp_pfa;
```

The configured CFAR detector is shown below.

```
cfar
```

```
cfar =
  phased.CFARDetector with properties:

                   Method: 'CA'
            NumGuardCells: 2
         NumTrainingCells: 20
           ThresholdFactor: 'Auto'
    ProbabilityFalseAlarm: 1.0000e-03
             OutputFormat: 'CUT result'
```

```
    ThresholdOutputPort: false
  NoisePowerOutputPort: false
```

We now simulate the input data. Since the focus is to show that the CFAR detector can keep the false alarm rate under a certain value, we just simulate the noise samples in those cells. Here are the settings:

- The data sequence is 23 samples long, and the CUT is cell 12. This leaves 10 training cells and 1 guard cell on each side of the CUT.

- The false alarm rate is calculated using 100 thousand Monte Carlo trials.

```matlab
rs = RandStream('mt19937ar','Seed',2010);
npower = db2pow(-10);   % Assume 10dB SNR ratio

Ntrials = 1e5;
Ncells = 23;
CUTIdx = 12;

% Noise samples after a square law detector
rsamp = randn(rs,Ncells,Ntrials)+1i*randn(rs,Ncells,Ntrials);
x = abs(sqrt(npower/2)*rsamp).^2;
```

To perform the detection, pass the data through the detector. In this example, there is only one CUT, so the output is a logical vector containing the detection result for all the trials. If the result is true, it means that a target is present in the corresponding trial. In our example, all detections are false alarms because we are only passing in noise. The resulting false alarm rate can then be calculated based on the number of false alarms and the number of trials.

```matlab
x_detected = cfar(x,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

```
act_pfa = 9.4000e-04
```

The result shows that the resulting probability of false alarm is below 0.001, just as we specified.

CFAR Detection Using Custom Threshold Factor

As explained in the earlier part of this example, there are only a few cases in which the CFAR detector can automatically compute the appropriate threshold factor. For example, using the previous scenario, if we employ a 10-pulses noncoherent integration before the data goes into the detector, the automatic threshold can no longer provide the desired false alarm rate.

```matlab
npower = db2pow(-10);   % Assume 10dB SNR ratio
xn = 0;
for m = 1:10
    rsamp = randn(rs,Ncells,Ntrials)+1i*randn(rs,Ncells,Ntrials);
    xn = xn + abs(sqrt(npower/2)*rsamp).^2;   % noncoherent integration
end
x_detected = cfar(xn,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

```
act_pfa = 0
```

One may be puzzled why we think a resulting false alarm rate of 0 is worse than a false alarm rate of 0.001. After all, isn't a false alarm rate of 0 a great thing? The answer to this question lies in the fact that when the probability of false alarm is decreased, so is the probability of detection. In this case,

because the true false alarm rate is far below the allowed value, the detection threshold is set too high. The same probability of detection can be achieved with our desired probability of false alarm at lower cost; for example, with lower transmitter power.

In most cases, the threshold factor needs to be estimated based on the specific environment and system configuration. We can configure the CFAR detector to use a custom threshold factor, as shown below.

```
release(cfar);
cfar.ThresholdFactor = 'Custom';
```

Continuing with the pulse integration example and using empirical data, we found that we can use a custom threshold factor of 2.35 to achieve the desired false alarm rate. Using this threshold, we see that the resulting false alarm rate matches the expected value.

```
cfar.CustomThresholdFactor = 2.35;
x_detected = cfar(xn,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

```
act_pfa = 9.6000e-04
```

CFAR Detection Threshold

A CFAR detection occurs when the input signal level in a cell exceeds the threshold level. The threshold level for each cell depends on the threshold factor and the noise power in that derived from training cells. To maintain a constant false alarm rate, the detection threshold will increase or decrease in proportion to the noise power in the training cells. Configure the CFAR detector to output the threshold used for each detection using the `ThresholdOutputPort` property. Use an automatic threshold factor and 200 training cells.

```
release(cfar);
cfar.ThresholdOutputPort = true;
cfar.ThresholdFactor = 'Auto';
cfar.NumTrainingCells = 200;
```

Next, create a square-law input signal with increasing noise power.

```
rs = RandStream('mt19937ar','Seed',2010);
Npoints = 1e4;
rsamp = randn(rs,Npoints,1)+1i*randn(rs,Npoints,1);
ramp = linspace(1,10,Npoints)';
xRamp = abs(sqrt(npower*ramp./2).*rsamp).^2;
```

Compute detections and thresholds for all cells in the signal.

```
[x_detected,th] = cfar(xRamp,1:length(xRamp));
```

Next, compare the CFAR threshold to the input signal.

```
plot(1:length(xRamp),xRamp,1:length(xRamp),th,...
  find(x_detected),xRamp(x_detected),'o')
legend('Signal','Threshold','Detections','Location','Northwest')
xlabel('Time Index')
ylabel('Level')
```
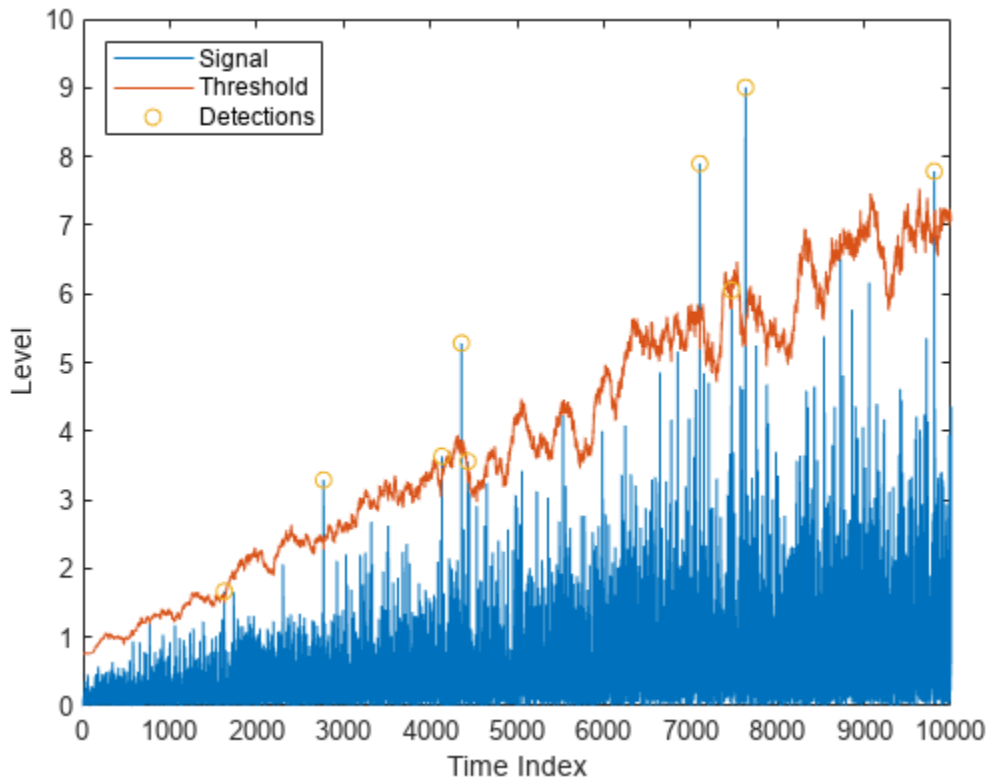
Here, the threshold increases with the noise power of the signal to maintain the constant false alarm rate. Detections occur where the signal level exceeds the threshold.

Comparison Between CFAR and Classical Neyman-Pearson Detector

In this section, we compare the performance of a CFAR detector with the classical detection theory using the Neyman-Pearson principle. Returning to the first example and assuming the true noise power is known, the theoretical threshold can be calculated as

```
T_ideal = npower*db2pow(npwgnthresh(exp_pfa));
```

The false alarm rate of this classical Neyman-Pearson detector can be calculated using this theoretical threshold.

```
act_Pfa_np = sum(x(CUTIdx,:)>T_ideal)/Ntrials
```

act_Pfa_np = 9.5000e-04

Because we know the noise power, classical detection theory also produces the desired false alarm rate. The false alarm rate achieved by the CFAR detector is similar.

```
release(cfar);
cfar.ThresholdOutputPort = false;
cfar.NumTrainingCells = 20;
x_detected = cfar(x,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

act_pfa = 9.4000e-04

Next, assume that both detectors are deployed to the field and that the noise power is 1 dB more than expected. In this case, if we use the theoretical threshold, the resulting probability of false alarm is four times more than what we desire.

```
npower = db2pow(-9);  % Assume 9dB SNR ratio
rsamp = randn(rs,Ncells,Ntrials)+1i*randn(rs,Ncells,Ntrials);
x = abs(sqrt(npower/2)*rsamp).^2;
act_Pfa_np = sum(x(CUTIdx,:)>T_ideal)/Ntrials
```

```
act_Pfa_np = 0.0041
```

On the contrary, the CFAR detector's performance is not affected.

```
x_detected = cfar(x,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

```
act_pfa = 0.0011
```

Hence, the CFAR detector is robust to noise power uncertainty and better suited to field applications.

Finally, use a CFAR detection in the presence of colored noise. We first apply the classical detection threshold to the data.

```
npower = db2pow(-10);
fcoeff = maxflat(10,'sym',0.2);
x = abs(sqrt(npower/2)*filter(fcoeff,1,rsamp)).^2;   % colored noise
act_Pfa_np = sum(x(CUTIdx,:)>T_ideal)/Ntrials
```

```
act_Pfa_np = 0
```

Note that the resulting false alarm rate cannot meet the requirement. However, using the CFAR detector with a custom threshold factor, we can obtain the desired false alarm rate.

```
release(cfar);
cfar.ThresholdFactor = 'Custom';
cfar.CustomThresholdFactor = 12.85;
x_detected = cfar(x,CUTIdx);
act_pfa = sum(x_detected)/Ntrials
```

```
act_pfa = 0.0010
```

CFAR Detection for Range-Doppler Images

In the previous sections, the noise estimate was computed from training cells leading and lagging the CUT in a single dimension. We can also perform CFAR detection on images. Cells correspond to pixels in the images, and guard cells and training cells are placed in bands around the CUT. The detection threshold is computed from cells in the rectangular training band around the CUT.
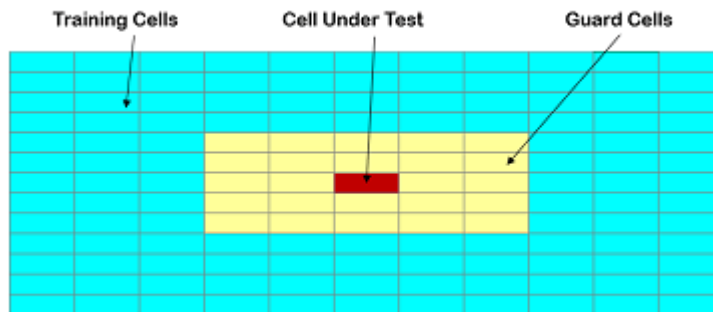
In the figure above, the guard band size is [2 2] and the training band size is [4 3]. The size indices refer to the number of cells on each side of the CUT in the row and columns dimensions, respectively. The guard band size can also be defined as 2, since the size is the same along row and column dimensions.

Next, create a two-dimensional CFAR detector. Use a probability of false alarm of 1e-5 and specify a guard band size of 5 cells and a training band size of 10 cells.

```
cfar2D = phased.CFARDetector2D('GuardBandSize',5,'TrainingBandSize',10,...
    'ProbabilityFalseAlarm',1e-5);
```

Next, load and plot a range-doppler image. The image includes returns from two stationary targets and one target moving away from the radar.

```
[resp,rngGrid,dopGrid] = helperRangeDoppler;
```
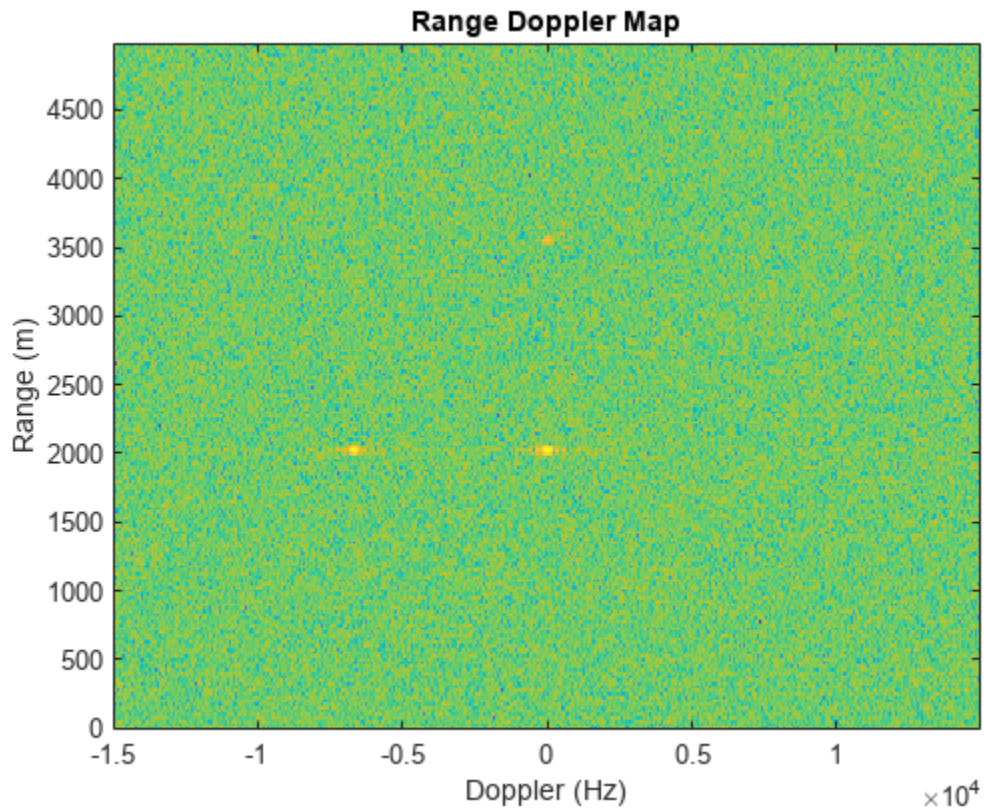
Range Doppler Map

Use CFAR to search the range-Doppler space for objects, and plot a map of the detections. Search from -10 to 10 kHz and from 1000 to 4000 m. First, define the cells under test for this region.

```
[~,rangeIndx] = min(abs(rngGrid-[1000 4000]));
[~,dopplerIndx] = min(abs(dopGrid-[-1e4 1e4]));
[columnInds,rowInds] = meshgrid(dopplerIndx(1):dopplerIndx(2),...
    rangeIndx(1):rangeIndx(2));
CUTIdx = [rowInds(:) columnInds(:)]';
```

Compute a detection result for each cell under test. Each pixel in the search region is a cell in this example. Plot a map of the detection results for the range-Doppler image.

```
detections = cfar2D(resp,CUTIdx);
helperDetectionsMap(resp,rngGrid,dopGrid,rangeIndx,dopplerIndx,detections)
```

Range Doppler CFAR Detections
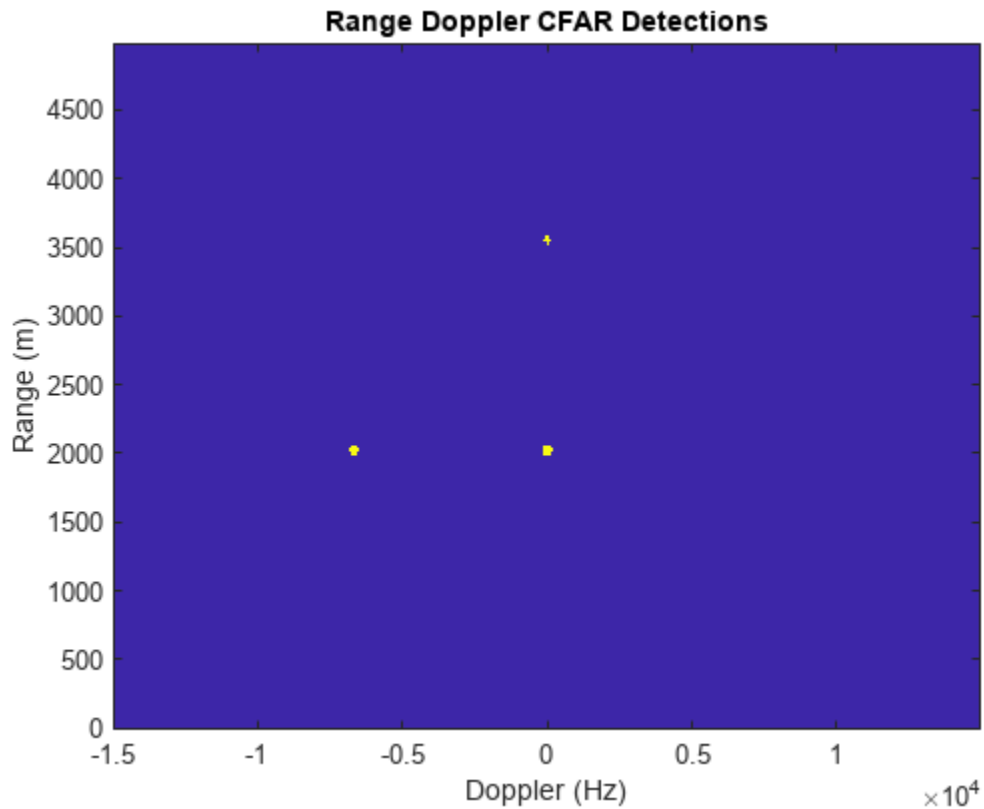
The three objects are detected. A data cube of range-Doppler images over time can likewise be provided as the input signal to `cfar2D`, and detections will be calculated in a single step.

Summary

In this example, we presented the basic concepts behind CFAR detectors. In particular, we explored how to use the Phased Array System Toolbox to perform cell averaging CFAR detection on signals and range-Doppler images. The comparison between the performance offered by a cell averaging CFAR detector and a detector equipped with the theoretically calculated threshold shows clearly that the CFAR detector is more suitable for real field applications.

Reference

[1] Mark Richards, *Fundamentals of Radar Signal Processing*, McGraw Hill, 2005

Waveform Parameter Extraction from Received Pulse

Modern aircraft often carry a radar warning receiver (RWR) with them. The RWR detects the radar emission and warns the pilot when the radar signal shines on the aircraft. An RWR can not only detect the radar emission, but also analyze the intercepted signal and catalog what kind of radar is the signal coming from. This example shows how an RWR can estimate the parameters of the intercepted pulse. The example simulates a scenario with a ground surveillance radar (emitter) and a flying aircraft (target) equipped with an RWR. The RWR intercepts radar signals, extracts the waveform parameters from the intercepted pulse, and estimates the location of the emitter. The extracted parameters can be utilized by the aircraft to take counter-measures.

This example requires Image Processing Toolbox™.

Introduction

An RWR is a passive electronic warfare support system [1] that provides timely information to the pilot about its RF signal environment. The RWR intercepts an impinging signal, and uses signal processing techniques to extract information about the intercepted waveform characteristics, as well as the location of the emitter. This information can be used to invoke counter-measures, such as jamming to avoid being detected by the radar. The interaction between the radar and the aircraft is depicted in the following diagram.
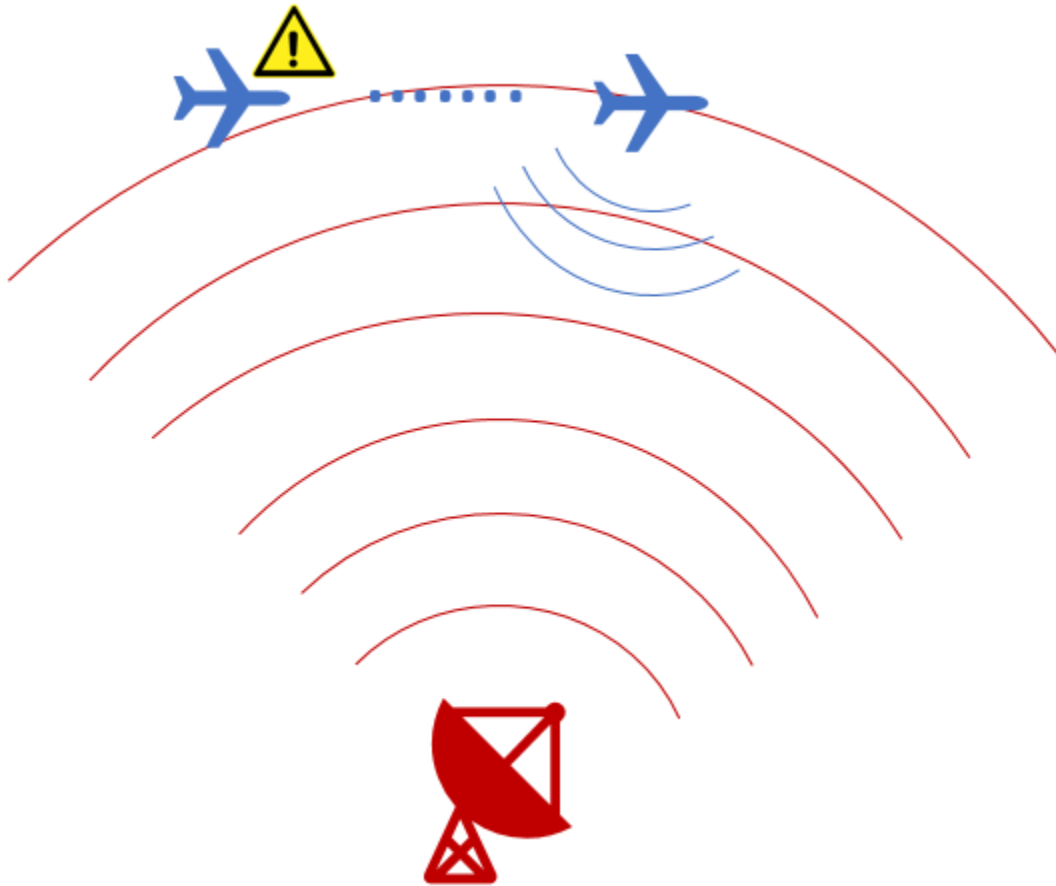
This example simulates a scenario with a ground surveillance radar and an airplane with an RWR. The RWR detects the radar signal and extracts the following waveform parameters from the intercepted signal:

1. Pulse repetition interval
2. Center frequency
3. Bandwidth
4. Pulse duration
5. Direction of arrival
6. Position of the emitter

The RWR chain consists of a phased array antenna, a channelized receiver, an envelope detector, and a signal processor. The frequency band of the intercepted signal is estimated by the channelized receiver and the envelope detector, following which the detected subbanded signal is fed to the signal processor. Beam steering is applied towards the direction of arrival of this subbanded signal, and the waveform parameters are estimated using pseudo Wigner-Ville transform in conjunction with Hough transform. Using angle of arrival and single-baseline approach, the location of the emitter is also estimated.

Scenario Setup

Assume the ground based surveillance radar operates in the L band, and transmits chirp signals of 3 μs duration at a pulse repetition interval of 15 μs. Bandwidth of the transmitted chirp is 30 MHz, and the carrier frequency is 1.8 GHz. The surveillance radar is located at the origin and is stationary, and the aircraft is flying at a constant speed of 200 m/s (~0.6 Mach).

```
% Define the transmitted waveform parameters
fs = 4e9;                       % Sampling frequency for the systems (Hz)
fc = 1.8e9;                     % Operating frequency of the surveillance radar (Hz)
T = 3e-6;                       % Chirp duration (s)
PRF = 1/(15e-6);                % Pulse repetition frequency (Hz)
BW = 30e6;                      % Chirp bandwidth (Hz)
c= physconst('LightSpeed');     % Speed of light in air (m/s)

% Assume the surveillance radar is at the origin and is stationary
radarPos= [0;0;0];          % Radar position (m)
radarVel= [0;0;0];          % Radar speed (m/s)

% Assume aircraft is moving with constant velocity
rwrPos= [-3000;1000;1000];    % Aircraft position (m)
rwrVel= [200; 0; 0];          % Aircraft speed (m/s)

% Configure objects to model ground radar and aircraft's relative motion
rwrPose = phased.Platform(rwrPos, rwrVel);
radarPose = phased.Platform(radarPos, radarVel);
```

The transmit antenna of the radar is a 8-by-8 uniform rectangular phased array, having a spacing of λ/2 between its elements. The signal propagates from the radar to the aircraft and is intercepted and analyzed by the RWR. For simplicity, the waveform is chosen as a linear FM waveform with a peak power of 100 W.

```
% Configure the LFM waveform using the waveform parameters defined above
wavGen= phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',T,'SweepBandwidth',BW,'PRF',PRF);

% Configure the Uniform Rectangular Array
antennaTx = phased.URA('ElementSpacing',repmat((c/fc)/2, 1, 2), 'Size', [8,8]);

% Configure objects for transmitting and propagating the radar signal
tx = phased.Transmitter('Gain', 5, 'PeakPower',100);
radiator = phased.Radiator( 'Sensor', antennaTx, 'OperatingFrequency', fc);
envIn = phased.FreeSpace('TwoWayPropagation',false,'SampleRate', fs,'OperatingFrequency',fc);
```

The ground surveillance radar is unaware of the direction of the target, therefore, it needs to scan the entire space to look for the aircraft. In general, the radar will transmit a series of pulses at each direction before moving to the next direction. Therefore, without losing generality, this example assumes that the radar is transmitting toward zero degrees azimuth and elevation. The following figure shows the time frequency representation of the arrival of a four-pulse train at the aircraft. Note that although the pulse train arrives at a specific delay, the time delay of the arrival of the first pulse is irrelevant for the RWR because it has no knowledge transmit time and has to constantly monitor its environment.

```
% Transmit a train of pulses
numPulses = 4;
txPulseTrain = helperRWR('simulateTransmission',numPulses, wavGen, rwrPos, ...
    radarPos, rwrVel, radarVel, rwrPose, radarPose, tx, radiator, envIn,fs,fc,PRF);
```

```
% Observe the signal arriving at the RWR
pspectrum(txPulseTrain,fs,'spectrogram','FrequencyLimits',[1.7e9 1.9e9], 'Leakage',0.65)
title('Transmitted Pulse Train Spectrogram')
caxis([-110 -90])
```
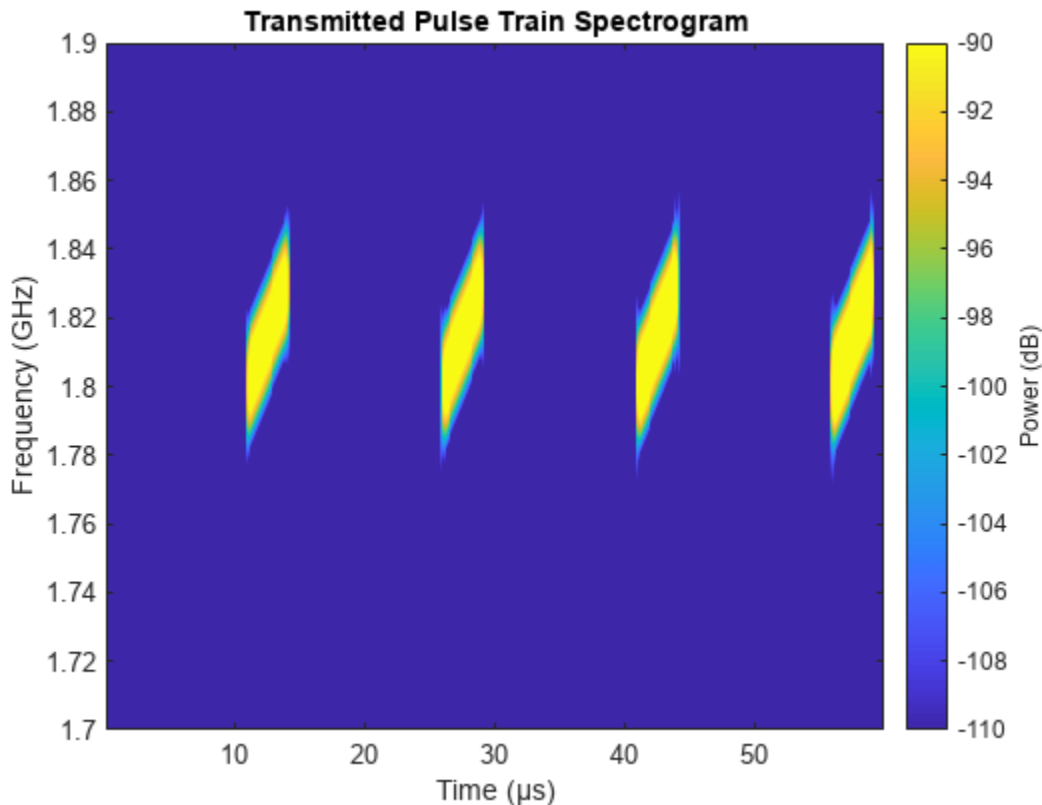

The RWR is equipped with a 10-by-10 uniform rectangular array with a spacing of $\lambda/2$ between its elements. It operates in the entire L-band, with a center frequency of 2 GHz. The RWR listens to the environment, and continuously feeds the collected data into the processing chain.

```
% Configure the receive antenna
dip = phased.IsotropicAntennaElement('BackBaffled',true);
antennaRx = phased.URA('ElementSpacing',repmat((c/2e9)/2,1,2),'Size', [10,10],'Element',dip);

% Model the radar receiver chain
collector = phased.Collector('Sensor', antennaRx,'OperatingFrequency',fc);
rx = phased.ReceiverPreamp('Gain',0,'NoiseMethod','Noise power', 'NoisePower',2.5e-6,'SeedSource

% Collect the waves at the receiver
[~, tgtAng] = rangeangle(radarPos,rwrPos);
yr = collector(txPulseTrain,tgtAng);
yr = rx(yr);
```

RWR Envelope Detector

The envelope detector in the RWR is responsible for detecting the presence of any signal. As the RWR is continuously receiving data, the receiver chain buffers and truncates the received data into 50 μs segments.

```
% Truncate the received data
truncTime = 50e-6;
truncInd = round(truncTime*fs);
yr = yr(1:truncInd,:);
```

Since the RWR has no knowledge about the exact center frequency used in the transmit waveform, it first uses a bank of filters, each tuned to a slightly different RF center frequency, to divide the received data into subbands. Then the envelope detector is applied in each band to check whether a signal is present. In this example, the signal is divided into subbands of 100 MHz bandwidth. An added benefit for such an operation is that instead of sampling the entire bandwidth covered by the RWR, the signal in each subband can be downsampled to a sampling frequency of 100 MHz.

```
% Define the bandwidth of each frequency subband
stepFreq = 100e6;
```

```
% Calculate number of subbands and configure dsp.Channelizer
numChan = fs/stepFreq;
channelizer = dsp.Channelizer('NumFrequencyBands', numChan, 'StopbandAttenuation', 80);
```

This plot shows the first four band created by the filter bank.

```
% Visualize the first four filters created in the filter bank of the
% channelizer
freqz(channelizer, 1:4)
title('Zoomed Channelizer Response for First Four Filters')
xlim([0 0.2])
```
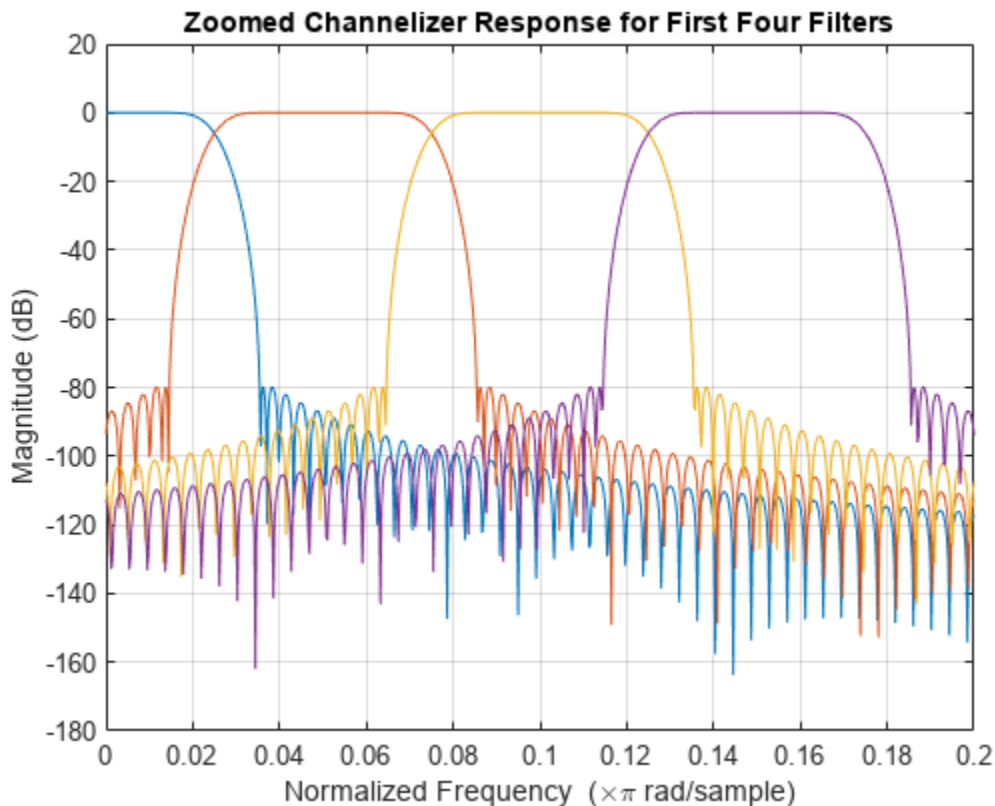
```
% Pass the received data through the channelizer
subData = channelizer(yr);
```

The received data, `subData`, has three dimensions. The first dimension represents the fast-time, the second dimension represents the subbands, and the third dimension corresponds to the receiving elements of the receiving array. For the RWR's 10-by-10 antenna configuration used in this example, you have 100 receiving elements. Because the transmit power is low and the receiver noise is high, the radar signal is indistinguishable from the noise. Therefore the received powers are summed across these elements to enhance the signal-to-nose ratio (SNR) and get a better estimate of the power in each subband. The band that has the maximum power is the one used by the radar.

```
% Rearrange the subData to combine the antenna array channels only
incohsubData = pulsint(permute(subData,[1,3,2]),'noncoherent');
incohsubData = squeeze(incohsubData);

% Plot power distribution
subbandPow = pow2db(rms(incohsubData,1).^2)+30;
plot(subbandPow);
xlabel('Band Index');
ylabel('Power (dBm)');
```
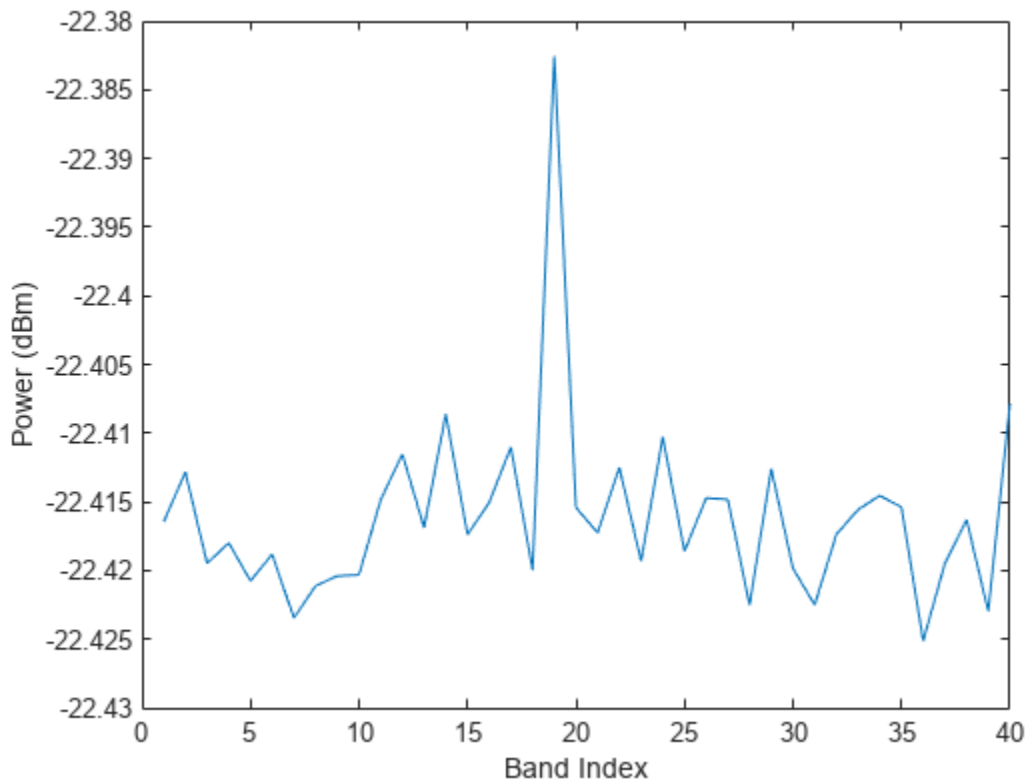


```
% Find the subband with maximum power
[~,detInd] = max(subbandPow);
```

1-743

RWR Signal Processor

Although the power in the selected band is higher compared to the neighboring band, the SNR within the band is still low, as shown in the following figure.

```
subData = (subData(:,detInd,:));
subData = squeeze(subData);  % adjust the data to 2-D matrix

% Visualize the detected sub-band data
plot(mag2db(abs(sum(subData,2)))+30)
ylabel('Power (dBm)')
title('Detected Subband from 100 Channels Combined Incoherently')
```
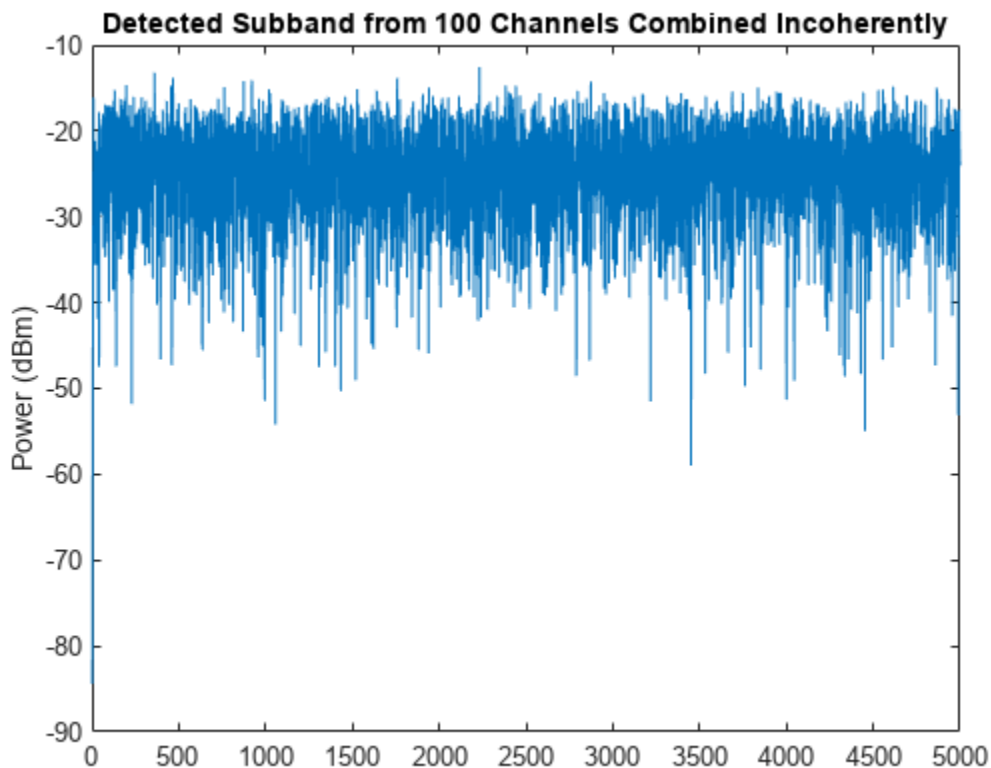


```
% Find the original starting frequency of the sub-band having the detected
% signal
detfBand = fs*(detInd-1)/(fs/stepFreq);

% Update the sampling frequency to the decimated frequency
fs = stepFreq;
```

The `subData` is now a two-dimensional matrix. The first dimension represents fast-time samples and the second dimension is the data across 100 receiving antenna channels. The detected subband starting frequency is calculated to find the carrier frequency of the detected signal.

The next step for the RWR is to find the direction from which the radio waves are arriving. This angle of arrival information will be used to steer the receive antenna beam in the direction of the emitter and locate the emitter on the ground using single baseline approach. The RWR estimates the

direction of arrival using a two-dimensional MUSIC estimator. Beam steering is done using phase-shift beamformer to achieve maximum SNR of the signal, and thus help in waveform parameter extraction.

Assume that the ground plane is flat and parallel to the xy-plane of the coordinate system. Then, the RWR can use the altitude information from its altimeter readings of the aircraft along with the direction of arrival to triangulate the location of the emitter.

```
% Configure the MUSIC Estimator to find the direction of arrival of the
% signal
doaEst = phased.MUSICEstimator2D('OperatingFrequency',fc,'PropagationSpeed',c,...
    'SensorArray',antennaRx,'DOAOutputPort',true,'AzimuthScanAngles',-50:.5:50,...
    'ElevationScanAngles',-50:.5:50, 'NumSignalsSource', 'Property','NumSignals', 1);

[mSpec,doa] = doaEst(subData);
plotSpectrum(doaEst,'Title','2-D MUSIC Spatial Spectrum Top View');
view(0,90); axis([-30 0 -30 0]);
```
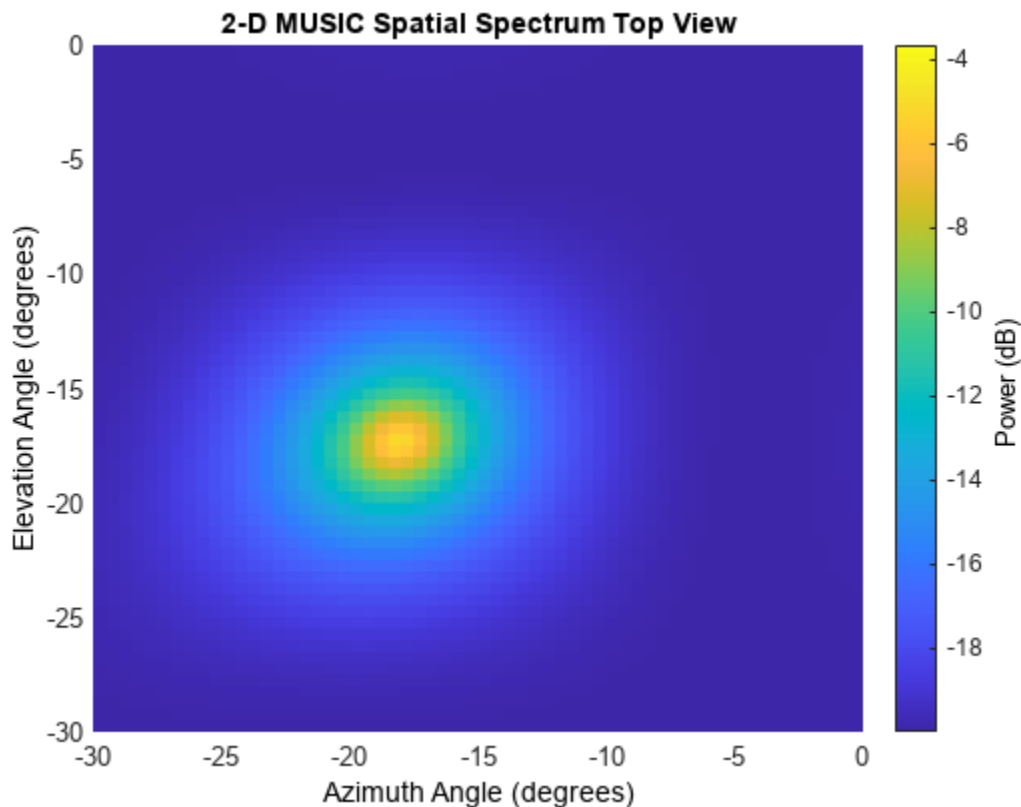

The figure clearly shows the location of the emitter.

```
% Configure the beamformer object to steer the beam before combining the
% channels
beamformer = phased.PhaseShiftBeamformer('SensorArray',antennaRx,...
    'OperatingFrequency',fc,'DirectionSource','Input port');

% Apply the beamforming, and visualize the beam steered radiation
% pattern
```

```
mBeamf = beamformer(subData, doa);

% Find the location of the emitter
altimeterElev = rwrPos(3);
d = abs(altimeterElev/sind(doa(2)));
```

After applying the beam steering, the antenna has the maximum gain in the azimuth and elevation angle of the arrival of the signal. This further improves the SNR of the intercepted signal. Next, the signal parameters are extracted in the signal processor using one of the time-frequency analysis techniques known as pseudo Wigner-Ville transform coupled with Hough transform as described in [2].

First, derive the time frequency representation of the intercepted signal using the Wigner-Ville transform.

```
% Compute the pseudo Wigner-Ville transform
[tpwv,t,f] = helperRWR('pWignerVille',mBeamf,fs);

% Plot the pseudo Wigner-Ville transform
imagesc(f*1e-6,t*1e6,pow2db(abs(tpwv./max(tpwv(:)))));
xlabel('Frequency (MHz)'); ylabel('Time(\mus)');
caxis([-50 0]); clb = colorbar; clb.Label.String = 'Normalized Power (dB)';
title ('Pseudo Wigner-Ville Transform')
```
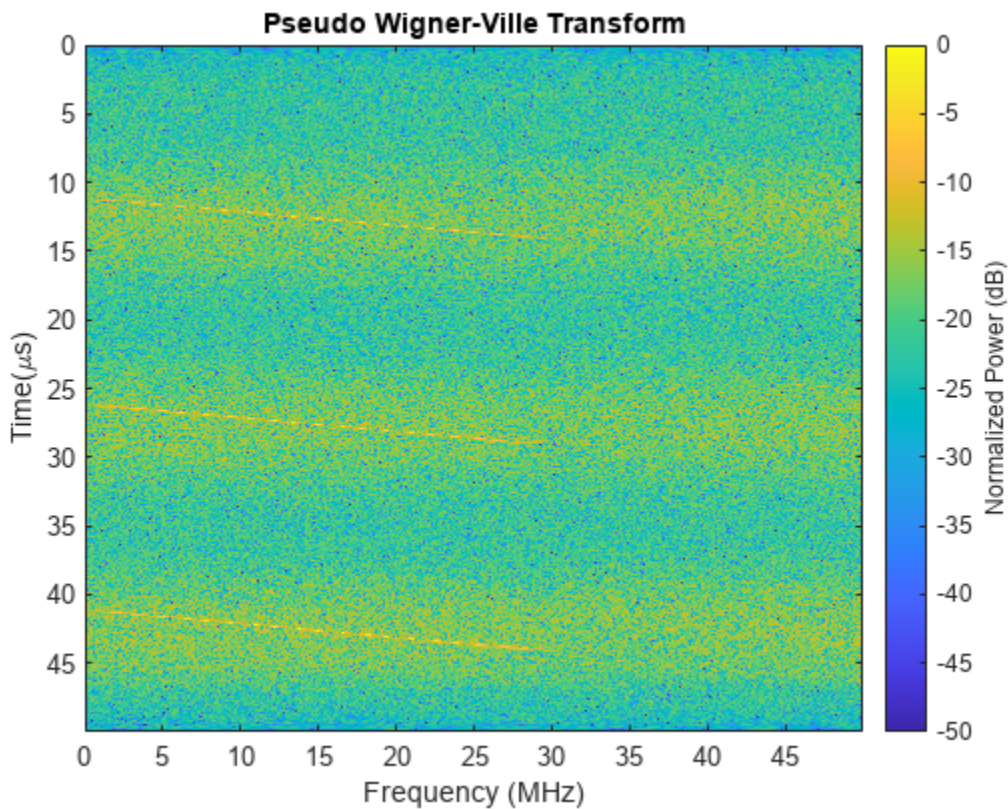

Using human eyes, even though the resulting time frequency representation is noisy, it is not too hard to separate the signal from the background. Each pulse appears as a line in the time frequency plane. Thus, using beginning and end of the time-frequency lines, you can derive the pulse width and the

bandwidth of the pulse. Similarly, the time between lines from different pulses gives you the pulse repetition interval.

To do this automatically without relying on human eyes, use Hough transform to identify those lines from the image. The Hough transform can perform well in the presence of noise, and the transform is an enhancement to the time-frequency signal analysis method.

To use the Hough transform, it is necessary to convert the time frequency image into a binary image. The following performs some data smoothing on the image and then uses the `imbinarize` function to do the conversion. The conversion threshold can be modified based on the signal-noise characteristics of the receiver and the operating environment.

```matlab
% Normalize the pseudo Wigner-Ville image
twvNorm = abs(tpwv)./max(abs(tpwv(:)));

% Implement a median filter to clear the noise
filImag = medfilt2(twvNorm,[7 7]);

% Use threshold to convert filtered image into binary image
BW = imbinarize(filImag./max(filImag(:)), 0.15);
imagesc(f*1e-6,t*1e6,BW); colormap('gray');
xlabel('Frequency (MHz)'); ylabel('Time(\mus)');
title ('Pseudo Wigner-Ville Transform - BW')
```
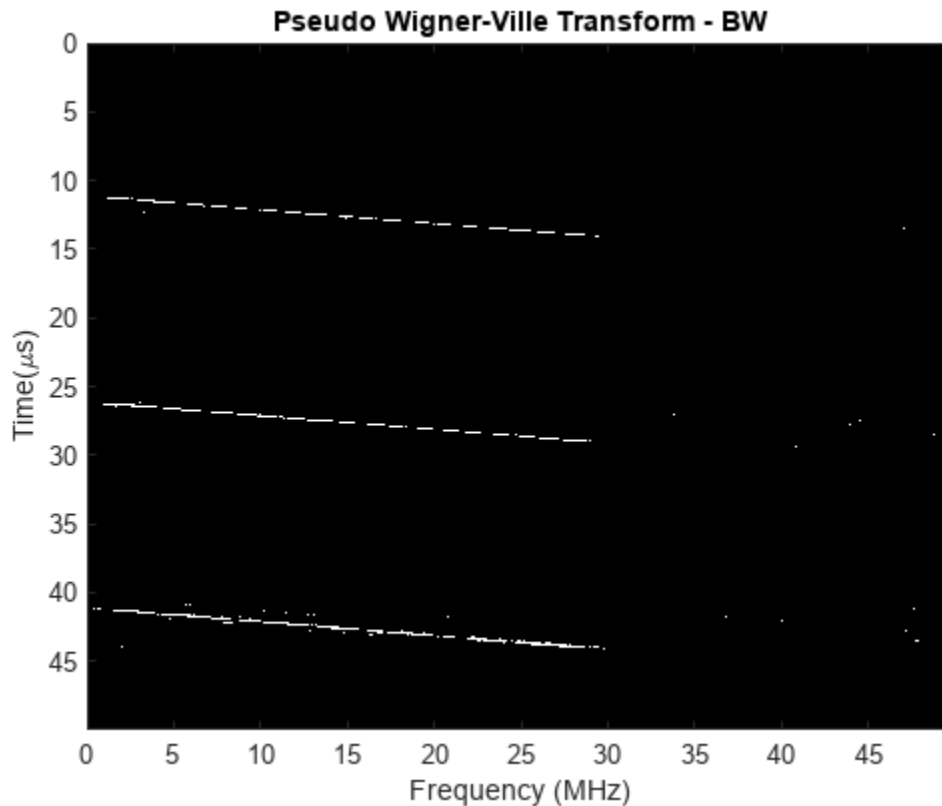

Using the Hough transform, the binary pseudo Wigner-Ville image is first transformed to peaks. This way, instead of detecting the line in an image, you just need to detect a peak in an image.

```
% Compute the Hough transform of the image and plot
[H,T,R] = hough(BW);
imshow(H,[],'XData',T,'YData',R,'InitialMagnification','fit');
xlabel('\theta'), ylabel('\rho');
axis on, axis normal, hold on;
title('Hough Transform of the Image')
```

The peak positions are extracted using the `houghpeaks` function.

```
% Compute peaks in the transform, up to 5 peaks
P  = houghpeaks(H,5);
x = T(P(:,2)); y = R(P(:,1));
plot(x,y,'s','color','g'); xlim([-90 -50]); ylim([-5000 0])
```
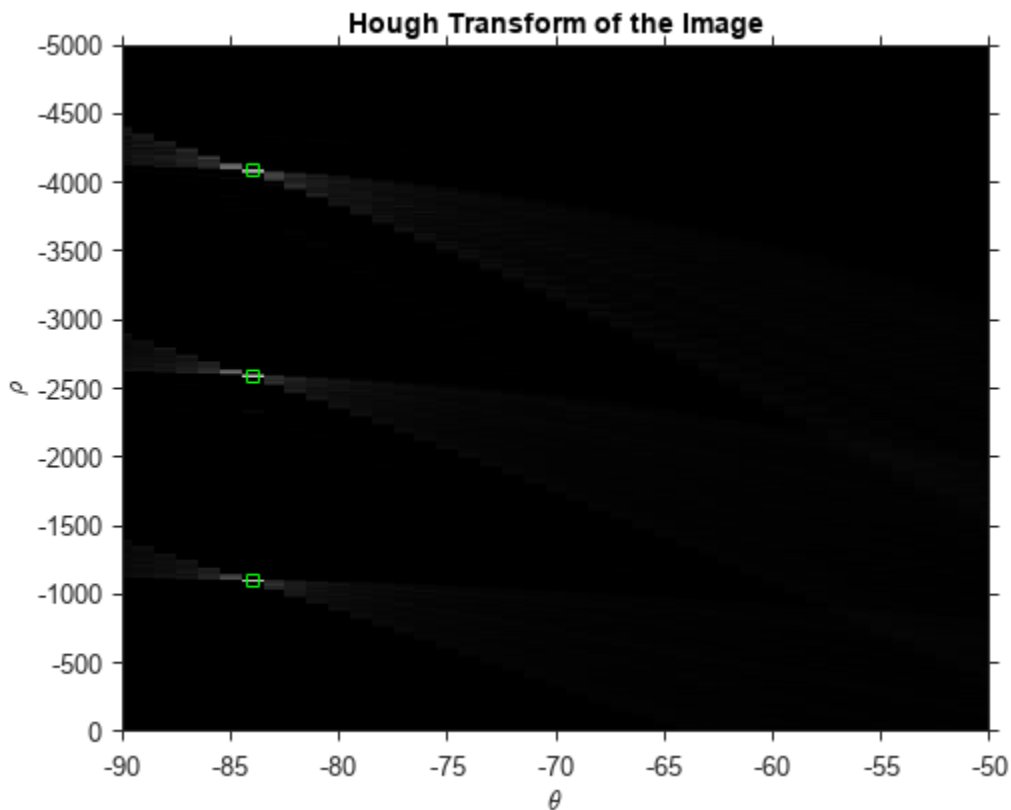

Using these positions, the `houghlines` function can reconstruct the lines in the original binary image. Then as discussed earlier, the beginning and the end of these lines help you estimate the waveform parameters.

```
lines = houghlines(BW,T,R,P,'FillGap',3e-6*fs,'MinLength',1e-6*fs);
coord = [lines(:).point1; lines(:).point2];
```

```
% Plot the detected lines superimposed on the binary image
clf;
imagesc(f*1e-6, t*1e6, BW); colormap(gray); hold on
xlabel('Frequency (MHz)')
ylabel('Time(\mus)')
title('Hough Transform - Detected Lines')
for ii = 1:2:2*size(lines,2)
```

```
    plot(f(coord(:,ii))*1e-6, t(coord(:,ii+1))*1e6,'LineWidth',2,'Color','green');
end
```

Hough Transform — Detected Lines
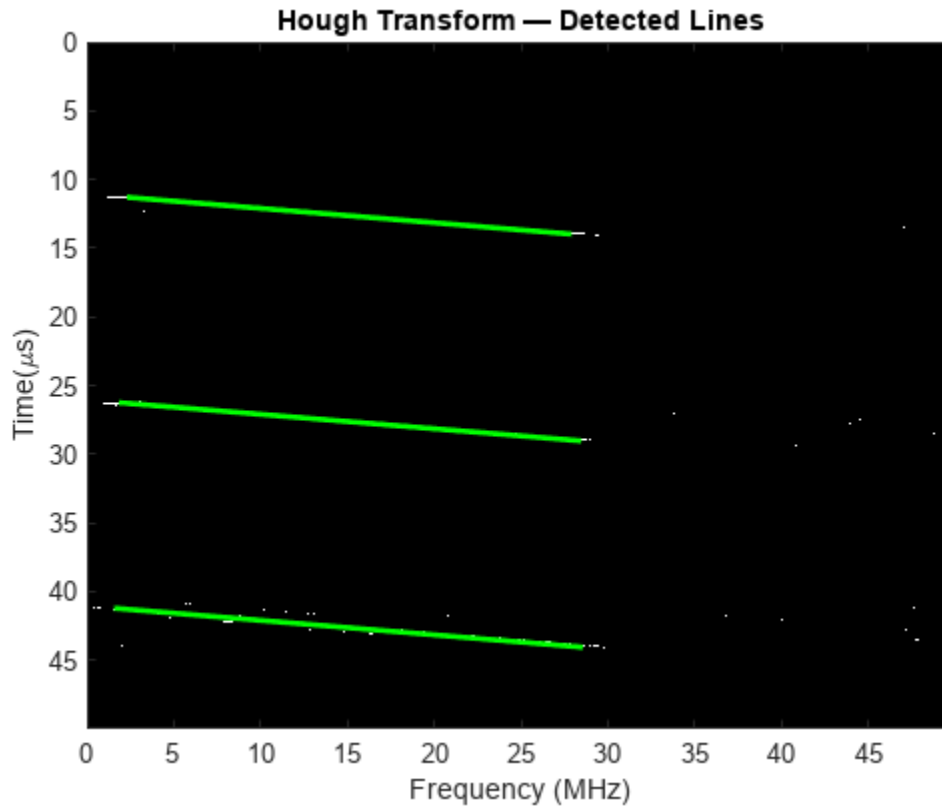

```
% Calculate the parameters using the line co-ordinates
pulDur = t(coord(2,2)) - t(coord(1,2));          % Pulse duration
bWidth = f(coord(2,1)) - f(coord(1,1));          % Pulse Bandwidth
pulRI = abs(t(coord(1,4)) - t(coord(1,2)));      % Pulse repetition interval
detFc = detfBand + f(coord(2,1));                % Center frequency
```

The extracted waveform characteristics are listed next. They match the truth very well. These estimates can then be used to catalog the radar and prepare for counter measures if necessary.

```
helperRWR('displayParameters',pulRI, pulDur, bWidth,detFc, doa,d);
```

```
Pulse repetition interval = 14.97 microseconds
Pulse duration = 2.84 microseconds
Pulse bandwidth = 27 MHz
Center frequency = 1.8286 GHz
Azimuth angle of emitter = -18.5 degrees
Elevation angle of emitter = -17.5 degrees
Distance of the emitter = 3325.5095 m
```

Summary

This example shows how an RWR can estimate the parameters of the intercepted radar pulse using signal processing and image processing techniques.

References

[1] *Electronic Warfare and Radar Systems Engineering Handbook* 2013, Naval Air Warfare Center Weapons Division, Point Mugu, California.

[2] Stevens, Daniel L., and Stephanie A. Schuckers. "Detection and Parameter Extraction of Low Probability of Intercept Radar Signals Using the Hough Transform." *Global Journal of Researches in Engineering*, Jan. 2016, pp. 9–25. *DOI.org (Crossref)*, https://doi.org/10.34257/GJREJVOL15IS6PG9.

Lidar and Radar Fusion in Urban Air Mobility Scenario

This example shows how to use multiobject trackers to track various unmanned aerial vehicles (UAVs) in an urban environment. You create a scene using the `uavScenario` object based on building and terrain data available online. You then 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 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 dataset available from the U.S. Geological Survey. The building data in `southboulder.osm` was downloaded from https://www.openstreetmap.org/, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

```
dtedfile = "n39_w106_3arc_v2.dt1";
buildingfile = "southboulder.osm";
scene = createScenario(dtedfile,buildingfile);
```

Next, add a few UAVs to the scenario.

To model a package delivery operation, define a trajectory leaving from the roof of a building and flying to a different 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", "AutoBank
uavA = uavPlatform("UAV", scene, "Trajectory", trajA, "ReferenceFrame", "ENU");
updateMesh(uavA, "quadrotor", {5}, [0.6350    0.0780    0.1840], eye(4));
```

Add another UAV to model an air taxi flying by. Its trajectory is linear and slightly descending. Use the `fixedwing` geometry to model a larger UAV that is suitable for transporting people.

```
waypointsB = [1940 120 50; 1800 50 20];
timeB = [0 41];
trajB = waypointTrajectory(waypointsB, "TimeOfArrival", timeB, "ReferenceFrame", "ENU", "AutoBank
uavB = uavPlatform("UAV2", scene, "Trajectory", trajB, "ReferenceFrame", "ENU");
updateMesh(uavB, "fixedwing", {10}, [0.6350    0.0780    0.1840], eye(4));
```

Then add a quadrotor with a trajectory following the street path. This represents 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", "AutoBank
uavC = uavPlatform("UAV3", scene, "Trajectory", trajC, "ReferenceFrame", "ENU");
updateMesh(uavC, "quadrotor", {5}, [0.6350    0.0780    0.1840], eye(4));
```

Finally, 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. Use a lidar puck that is commonly used in automotive applications [1]. The puck is a small sensor that can be attached on a quadrotor. 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, 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. For details on that, see the "Simulate Radar Sensor Mounted On UAV" (UAV Toolbox) 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. Based on the Echodyne EchoFlight UAV radar [2], set the radar configuration 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, 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, set the `TargetReportFormat` property to `Detections` so that the radar reports 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);

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 (ENU) 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

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 set up 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. 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. 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` at 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

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

                 OOSMHandling: 'Terminate'
```

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

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

                   NumTracks: 0
          NumConfirmedTracks: 0

       EnableMemoryManagement: false
```

Radar Tracker

In this example, you assume that the radar returns are preprocessed such that only returns from moving objects are preserved, that is, there are 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 of the approaches is that you can cluster the detections and augment the state with dimensions and orientation constants as done previously with the lidar cuboids. In the other approach, 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 3-D ellipse. This second approach is preferable because you do not have too many detections per object and clustering is less accurate than extended target tracking

To create a GGIW-PHD tracker, first define the tracking sensor configuration for each sensor reporting to the tracker. In this case, you need to define the configuration for only 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, define a filter initialization function based on the sensor configuration. Finally, 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
             MaxNumComponents: 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. Use the 13-dimensional state of lidar tracks as the fused state definition.

```matlab
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

Use a helper class to visualize the scenario. The helper class in this example utilizes the `uavScenario` visualization capabilities and the `theaterPlot` plotter to represent detection and track information.

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

```matlab
viewer = helperUAVDisplay(scene);
```
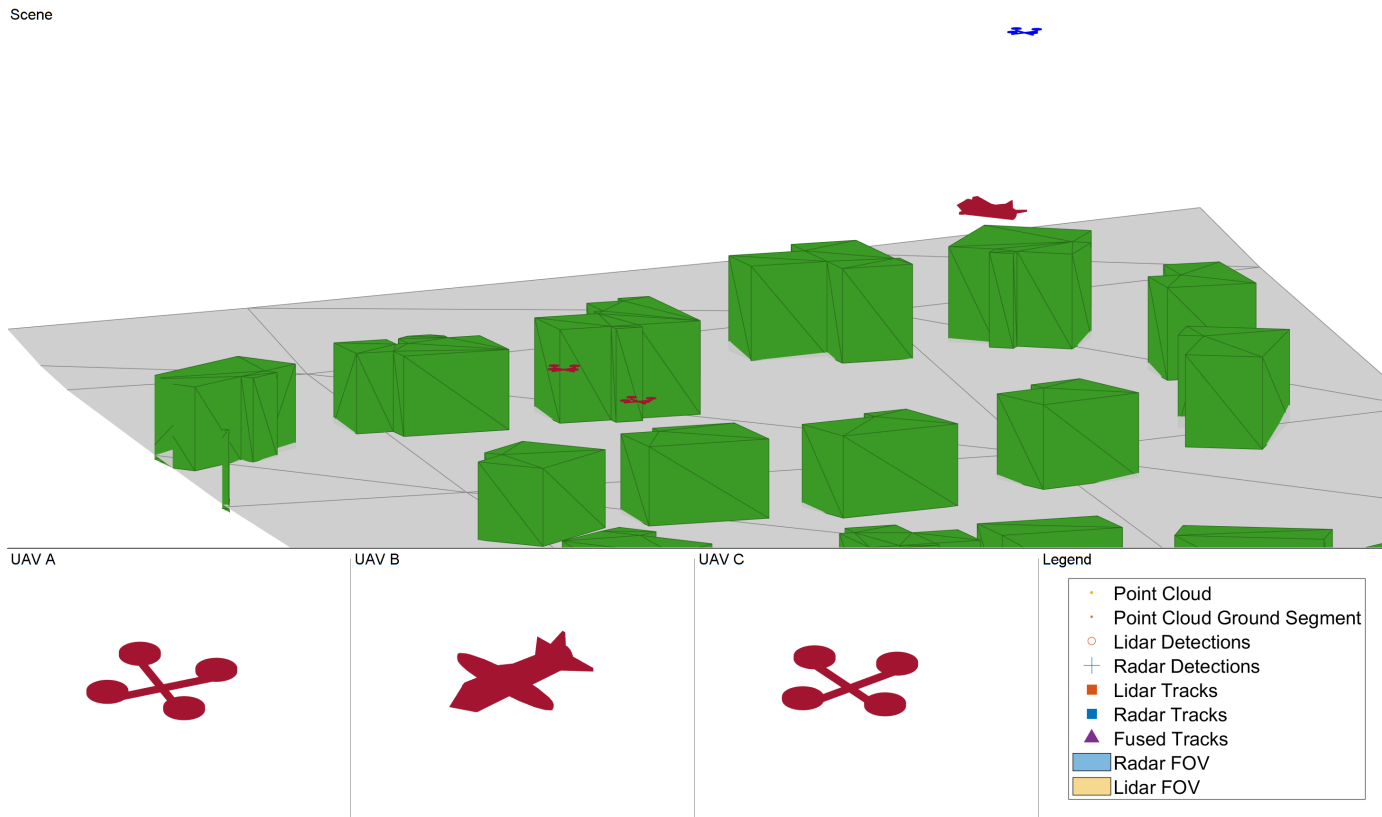
```matlab
% Radar and lidar coverages for display
[radarcov,lidarcov] = sensorCoverage(radarSensor, lidar);
```

Simulate Scenario

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 offline metric analysis.

```matlab
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, lidarcov, nonGroundCloud, groundCloud, lidardets, radardets, lidarTracks, ra
end
```
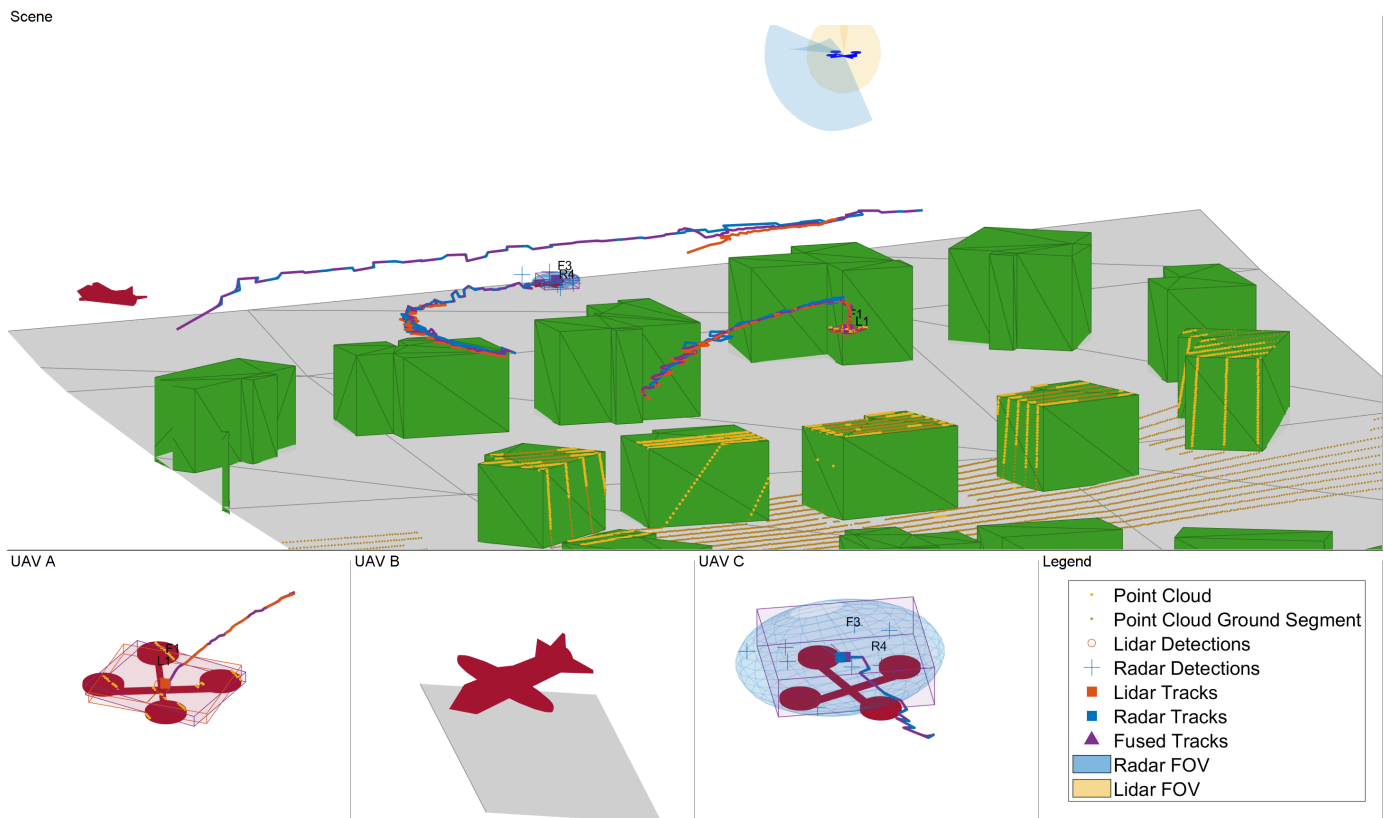

Based on the visualization results, perform an initial qualitative assessment of the tracking performance. The display at the end of the scenario shows that all three UAVs were well tracked by the ego. With the current sensor suite configuration, lidar tracks were only established partially due to the limited coverage of the lidar sensor. The wider field of view of the radar allowed establishing radar tracks more consistently in this scenario.

UAV A
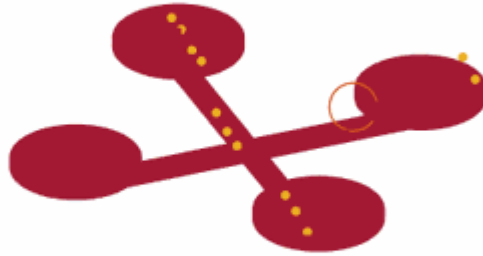
UAV B
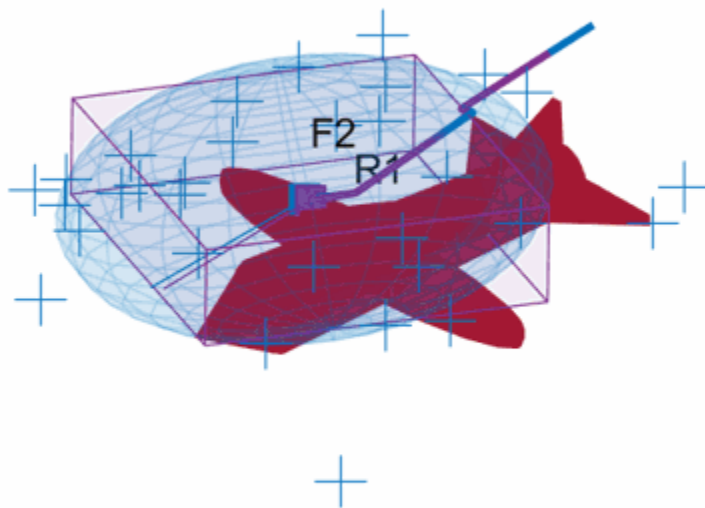
UAV C
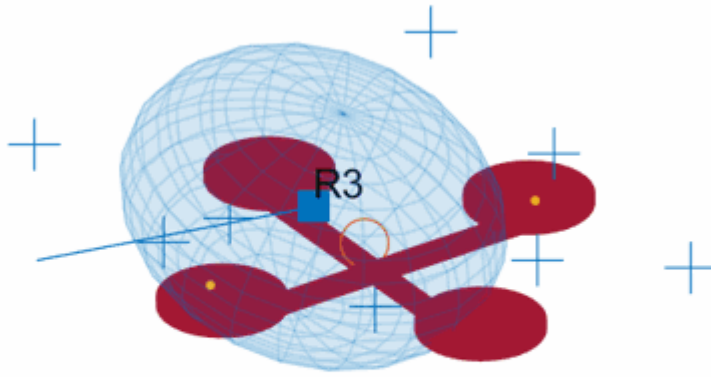
The three animated GIFs above show parts of the chase views. You can see that the quality of lidar tracks (orange box) is affected by the geometry of the scenario. UAV A (left) is illuminated by the lidar (shown in yellow) almost directly from above. This enables the tracker to capture the full extent 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 OSPA(2) tracking metric. First define the distance function which quantifies the error between track and truth using a scalar value. A lower OSPA value means an overall better performance.

```
ospaR = trackOSPAMetric('Metric','OSPA(2)','Distance','custom','DistanceFcn',@metricDistance);
ospaL = clone(ospaR);
ospaF = clone(ospaR);

ospaRadar = zeros(1,numSteps);
ospaLidar = zeros(1,numSteps);
ospaFused = zeros(1,numSteps);
```

```
for i=1:numSteps
    truth = truthlog{i};
    ospaRadar(i) = ospaR(radarlog{i},truth);
    ospaLidar(i) = ospaL(lidarlog{i},truth);
    ospaFused(i) = ospaF(fusedlog{i},truth);
end

figure
plot(ospaRadar,'Color',viewer.RadarColor,'LineWidth',2);
hold on
grid on
plot(ospaLidar,'Color',viewer.LidarColor,'LineWidth',2);
plot(ospaFused,'Color',viewer.FusedColor,'LineWidth',2);
legend('Radar','Lidar','Lidar + Radar');
xlabel('Steps')
ylabel('OSPA(2)')
```
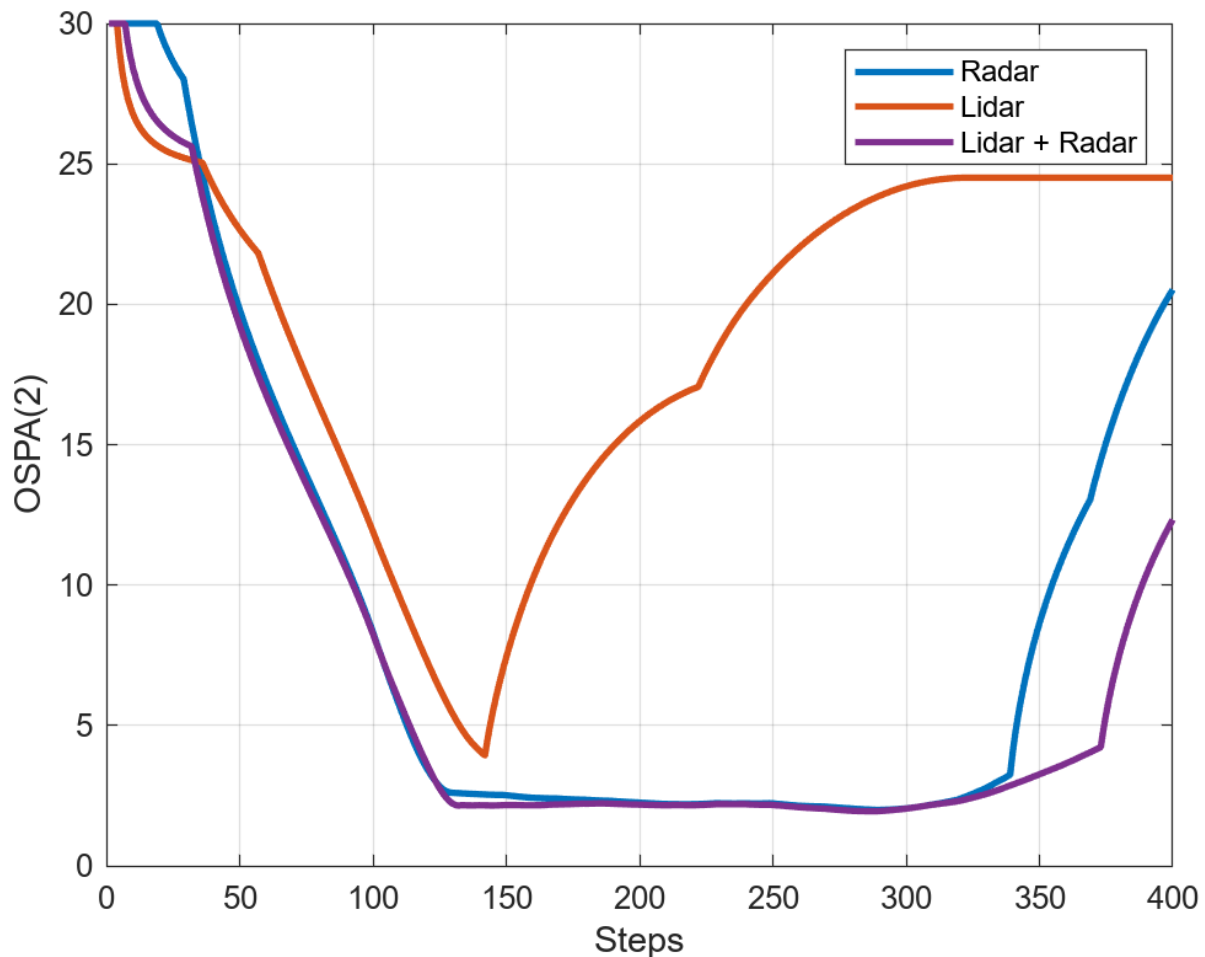

Analyze the overall system performance. Each tracker is penalized for not tracking any of the UAVs even if the target UAV is outside of the sensor coverage. This shows improved performance when fusing lidar and radar due to the added surveillance area. This is particularly noticeable at the end of

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 OSPA is below the minimum of lidar and radar OSPA, showing the fused track has better quality than each individual track.

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

Summary

This example showed you 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 then 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

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
scene.addMesh("buildings", {buildingfile, sceneXLim, sceneYLim, "auto"}, [0 0 0]);

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');
```

```matlab
% 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 trac
    'ConfirmationThreshold',0.99,...% Weight threshold of a filter component to be declared a con
    '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 Labeling
    '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 initialize new birth components in the density and to initialize new components from detection partitions.

```matlab
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,...
            '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;
```

```matlab
    % Measurement Jacobian
    p = detectionPartition{1}.MeasurementParameters;
    H = cvmeasjac(states,p);

    Bk = H(:,1:2:end);
    Bk2 = eye(3)/Bk;
    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 the augmented rectangular state. The Inverse Wishart random matrix eigen values are used to derive rectangular box dimensions. The eigen vectors provide 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 w
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 the 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 here.

```matlab
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;
stateOut(idx,qparts(1,:)<0) = -qparts(:,qparts(1,:)<0);
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
```

`sensorCoverage` constructs sensor coverage configuration structures for visualization.

```matlab
function [radarcov,lidarcov] = sensorCoverage(radarSensor, lidar)
radarcov = coverageConfig(radarSensor);
% Scale down coverage to limit visual clutter
radarcov.Range = 10;
lidarSensor = lidar.SensorModel;
```

```
        lidarcov = radarcov;
        lidarcov.Index = 2;
        lidarcov.FieldOfView = [diff(lidarSensor.AzimuthLimits); diff(lidarSensor.ElevationLimits)];
        lidarcov.Range = 5;
        lidarcov.Orientation = quaternion(lidar.MountingAngles,'eulerd','ZYX','frame');
    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];
    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);
        targetPoses(i).PlatformID = i;
    end
    logEntry = targetPoses;
    end
```

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

```
    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/

Introduction to Radar Scenario Clutter Simulation

This example shows how to generate monostatic surface clutter signals and detections in a radar scenario. Clutter detections will be generated with a monostatic `radarDataGenerator`, and clutter return signals will be generated with a `radarTransceiver,` using both homogenous surfaces and real terrain data from a DTED file. `theaterPlot` is used to visualize the scenario surface and clutter generation.

Configure Scenario for Clutter Generation

Configuration of a radar scenario to simulate surface clutter involves creating a `radarScenario` object, adding platforms with mounted radars, adding surface objects that define the physical properties of the scenario surface, and enabling clutter generation for a specific radar in the scene.

Select a Radar Model

Clutter generation is performed as part of the scenario `detect` and `receive` methods. These methods are used to generate simulated radar detections and IQ signals, respectively. For detections, which consist of measurement-level data along with useful metadata, use the `radarDataGenerator`. For raw IQ signals, use the `radarTransceiver`.

This section begins with a `radarDataGenerator`. Define some typical medium-PRF pulse-Doppler parameters for a side-looking airborne radar. Use a -90 degree mounting yaw angle so the radar faces to the right of the platform and a 10 degree depression angle so the radar is pointed towards the surface. The mounting roll angle can be 0 to indicate no rotation about the antenna boresight. Use a center frequency of 4 GHz, range resolution of 80 m, and a 12 kHz PRF with 128 pulses per coherent processing interval (CPI).

```
mountAng = [-90 10 0];
fc = 4e9;
rngRes = 80;
prf = 12e3;
numPulses = 128;
```

The `radarDataGenerator` is a statistical model that does not directly emulate an antenna pattern. Instead, it has properties that define the field of view and angular resolution. Use 10 degrees for the field of view and angular resolution in each direction. This configuration is comparable to simulating a single mainlobe with no angle estimation.

```
fov = [10 10];
angRes = fov;
```

Construct a `radarDataGenerator` from these parameters. The radar updates once per CPI. The mounting angles point the radar in a broadside direction. Let the field of view and angular resolution equal the beamwidth. Set `DetectionCoordinates` to `'Scenario'` to output detections in scenario coordinates for easier inspection of the results. Calculate the unambiguous range and radial speed and enable range and range-rate ambiguities for the radar. The ambiguities can be calculated with the `time2range` and `dop2speed` functions.

```
c = physconst('lightspeed');
lambda = freq2wavelen(fc);
rangeRateRes = lambda/2*prf/numPulses;
unambRange = time2range(1/prf);
unambRadialSpd = dop2speed(prf/4,lambda);
```

```
cpiTime = numPulses/prf;
rdr = radarDataGenerator(1,'No scanning','UpdateRate',1/cpiTime,'MountingAngles',mountAng,...
    'DetectionCoordinates','Scenario','HasINS',true,'HasElevation',true,'HasFalseAlarms',false,
    'HasRangeRate',true,'HasRangeAmbiguities',true,'HasRangeRateAmbiguities',true, ...
    'MaxUnambiguousRadialSpeed',unambRadialSpd,'MaxUnambiguousRange',unambRange,'CenterFrequency
    'FieldOfView',fov,'AzimuthResolution',angRes(1),'ElevationResolution',angRes(2), ...
    'RangeResolution',rngRes,'RangeRateResolution',rangeRateRes);
```

Create a Scenario

The `radarScenario` object is the top-level manager for the simulation. A radar scenario may be Earth-centered, where the WGS84 Earth model is used, or it may be flat. Use a flat-Earth scenario to enable use of simple kinematic trajectories for the platforms. Set `UpdateRate` to 0 to let the scenario derive an update rate from the objects in the scenario.

```
scenario = radarScenario('UpdateRate',0,'IsEarthCentered',false);
```

Add a radar platform to the scenario. Use a straight-line kinematic trajectory starting 1.5 km up, and moving in the +Y direction at 70 m/s, at a dive angle of 10 degrees. Orient the platform so the platform-frame +X direction is the direction of motion. Use the `Sensors` property to mount the radar.

```
rdrAlt = 1.5e3;
rdrSpd = 70;
rdrDiveAng = 10;
rdrPos = [0 0 rdrAlt];
rdrVel = rdrSpd*[0 cosd(rdrDiveAng) -sind(rdrDiveAng)];
rdrOrient = rotz(90).';
rdrTraj = kinematicTrajectory('Position',rdrPos,'Velocity',rdrVel,'Orientation',rdrOrient);
rdrplat = platform(scenario,'Sensors',rdr,'Trajectory',rdrTraj);
```

Define the Scenario Surface

Physical properties of the scenario surface can be specified by using the scenario `landSurface` and `seaSurface` methods to define regions of land and sea surface types. Each surface added to the scene is a rectangular region with an associated radar reflectivity model and reference height. Land surfaces may additionally have associated static terrain data, and sea surfaces may have an associated spectral motion model. If no terrain data or spectral model is used, a surface may be unbounded, allowing for homogeneous clutter generation.

Create a simple unbounded land surface with a constant-gamma reflectivity model. Use the `surfaceReflectivityLand` function to create a reflectivity model and attach the reflectivity model to the surface with the `RadarReflectivity` parameter. Use a gamma value of -20 dB.

```
refl = surfaceReflectivityLand('Model','ConstantGamma','Gamma',-20);
srf = landSurface(scenario,'RadarReflectivity',refl)

srf =
  LandSurface with properties:

    RadarReflectivity: [1x1 surfaceReflectivityLand]
      ReflectivityMap: 1
      ReferenceHeight: 0
             Boundary: [2x2 double]
               Terrain: []
```

1-771

The `ReferenceHeight` property gives the constant height of the surface when no terrain is specified, or the origin height to which terrain is referenced if terrain is specified. The `ReflectivityMap` property is relevant only when a custom reflectivity model is used, and allows different reflectivity curves to be associated to different parts of the surface. The `Boundary` property gives the rectangular boundary of the surface in two-point form. Elements of `Boundary` can be +/-inf to indicate the surface is unbounded in one or more directions. Check the boundary of the surface created above to see that it is unbounded in all directions.

`srf.Boundary`

```
ans = 2×2

  -Inf   Inf
  -Inf   Inf
```

Access the `SurfaceManager` property of the scenario to see the surface objects that have been added, as well as any additional options related to the scenario surface.

`scenario.SurfaceManager`

```
ans =
  SurfaceManager with properties:

    UseOcclusion: 1
        Surfaces: [1x1 radar.scenario.LandSurface]
```

The `UseOcclusion` property can be set false to disable line-of-sight occlusions by the surface, such as by terrain.

Enable Clutter Generation

Clutter Generator

Monostatic clutter generation is enabled for a specific radar by using the scenario `clutterGenerator` method. This method accepts parameter name-value pairs to configure clutter generation. This configuration is performed on a radar-by-radar basis so that multiple radars can be simulated simultaneously with clutter generation settings appropriate for each radar. The `clutterGenerator` method will return a handle to the `ClutterGenerator` object.

Reflections from large continuous surfaces are approximated by a set of point scatterers. By default, the `ClutterGenerator` operates in "uniform" scatterer distribution mode. In this mode, scatterers are placed randomly with a uniform density on the surface. This is a flexible mode that can be used for any surface and radar configuration. See the section below titled "Simulate Smooth Surface Clutter for a Range-Doppler Radar" for a demonstration of a different scatterer distribution mode that can adapt to range-Doppler resolution cells under some scenario constraints. When operating in uniform scatterer distribution mode, the `Resolution` property specifies the nominal spacing of clutter scatterers used to represent the surface reflection.

The `UseBeam` property is a logical scalar indicating whether or not automatic mainlobe clutter generation should be used (see the next section titled "Clutter Regions" for more details). The `RangeLimit` property is used to place an upper bound on the range of clutter generation, which is important for cases when mainlobe clutter generation is being used and has an unbounded footprint.

Create a `ClutterGenerator` object, enabling clutter generation for the radar created above. Use a `Resolution` of half the radar's range resolution in order to get a couple of clutter scatterers per

range sample. Set `UseBeam` to true to enable automatic clutter generation within the field of view of the radar. Use a `RangeLimit` of 12 km, which is just shorter than the unambiguous range.

```
clutRes = rngRes/2;
clutRngLimit = 12e3;
clut = clutterGenerator(scenario,rdr,'Resolution',clutRes,'UseBeam',true,'RangeLimit',clutRngLim

clut =
  ClutterGenerator with properties:

    ScattererDistribution: "Uniform"
               Resolution: 40
                  Regions: [1x0 radar.scenario.RingClutterRegion]
                  UseBeam: 1
            UseShadowing: 1
               RangeLimit: 12000
                    Radar: [1x1 radarDataGenerator]
               SeedSource: "Auto"
```

The `UseShadowing` property is a logical scalar used to enable/disable shadowing (surface self-occlusion). Shadowing is only relevant for surfaces with terrain data or a spectral model.

The clutter generator has two read-only properties. The `Radar` property stores a handle to the associated radar object, which was passed to the `clutterGenerator` method. The `Regions` property contains the set of user-defined "clutter regions".

Clutter Regions

Surface clutter is distributed over the entire range interval starting from the radar altitude and extending to the horizon range (or ∞ if using a flat-Earth scenario). It is distributed in elevation angle from -90 degrees up to the horizon elevation angle, and over all 360 degrees of azimuth. Finally, surface clutter is distributed in Doppler as a result of platform motion.

There are two options available to designate regions of the surface for clutter generation. The first is automatic mainlobe clutter, which generates clutter inside the footprint of the mainlobe of the radar antenna. This is only possible when the radar being used has a well-defined beam with azimuth and elevation two-sided beamwidths less than 180 degrees. For the `radarDataGenerator`, the "beam" is actually the field of view defined by the `FieldOfView` property, the footprint of which consists of contours of constant azimuth and elevation angle. For the `radarTransceiver`, a conical or fan-shaped beam is assumed based on the array type, and the beam is considered out to 3 dB below the peak gain.

The second option is to use the `ringClutterRegion` method of the clutter generator to specify a ring-shaped region of the scenario surface within which clutter is to be generated. This type of region is defined by a minimum and maximum ground range (relative to the radar nadir point) and an extent and center angle in azimuth. This region type is useful for capturing sidelobe and backlobe clutter return, mainlobe clutter return outside the 3 dB width, or to generate clutter from any other region of interest, such as at the location of a target platform.

The figure below illustrates these two region types. The beam footprint region is displayed as a magenta ellipse where the beam intersects the ground. Two ring regions are shown, one directly underneath the radar for capturing altitude return, and another for capturing some backlobe clutter return.
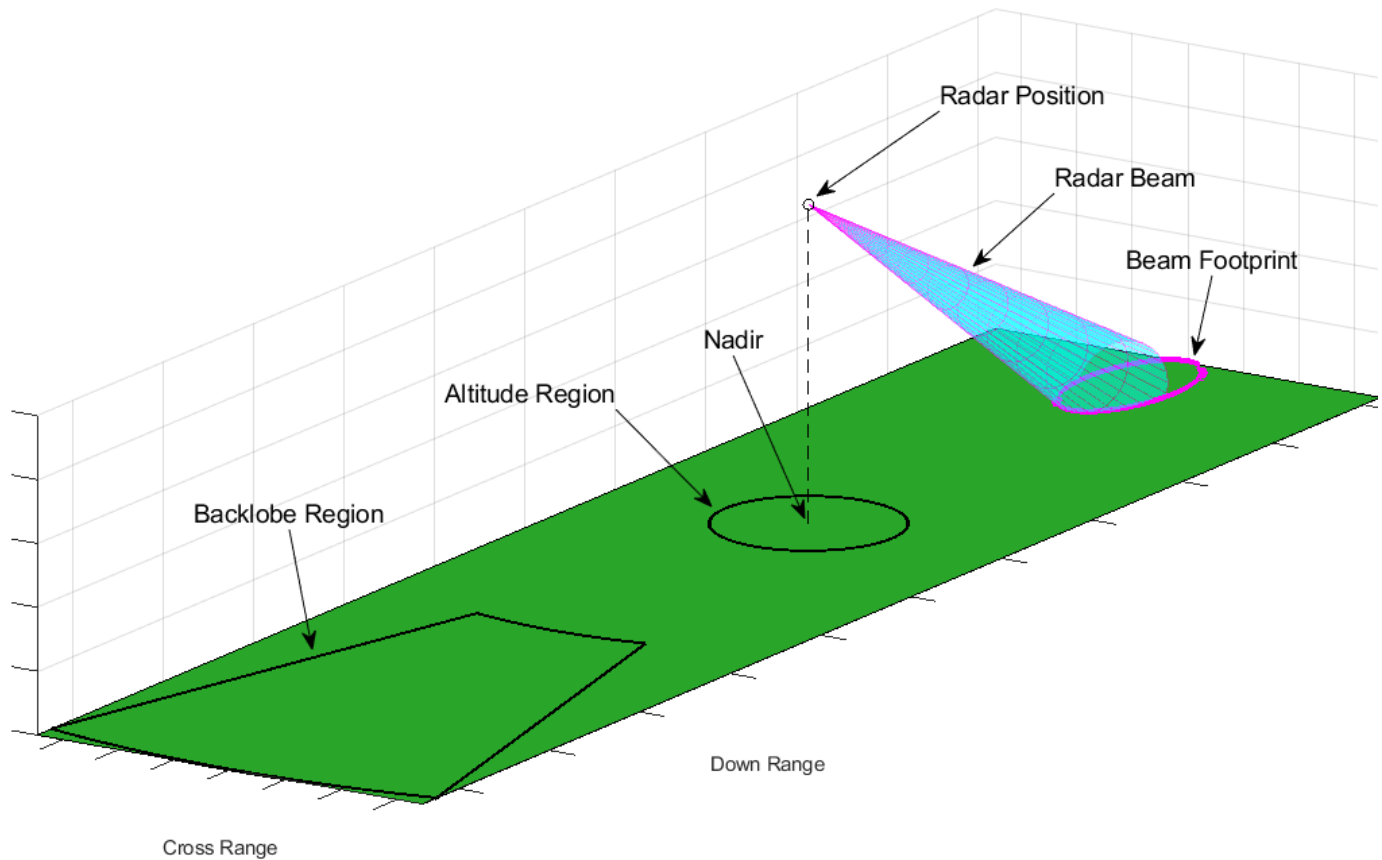
To demonstrate the utility of the customizable ring-shaped regions at capturing clutter return from arbitrary lobes of the antenna pattern, the above geometry is shown again below in a top-down view, with a gain pattern projected onto the ground. Note that a significant backlobe has been encompassed by a ring region. A circular region (which can be achieved by setting the minimum ground range to 0) can be used to capture altitude return. An additional region encompassing the mainlobe is also shown to demonstrate how this can be used to capture more of the mainlobe return, such as from null to null.
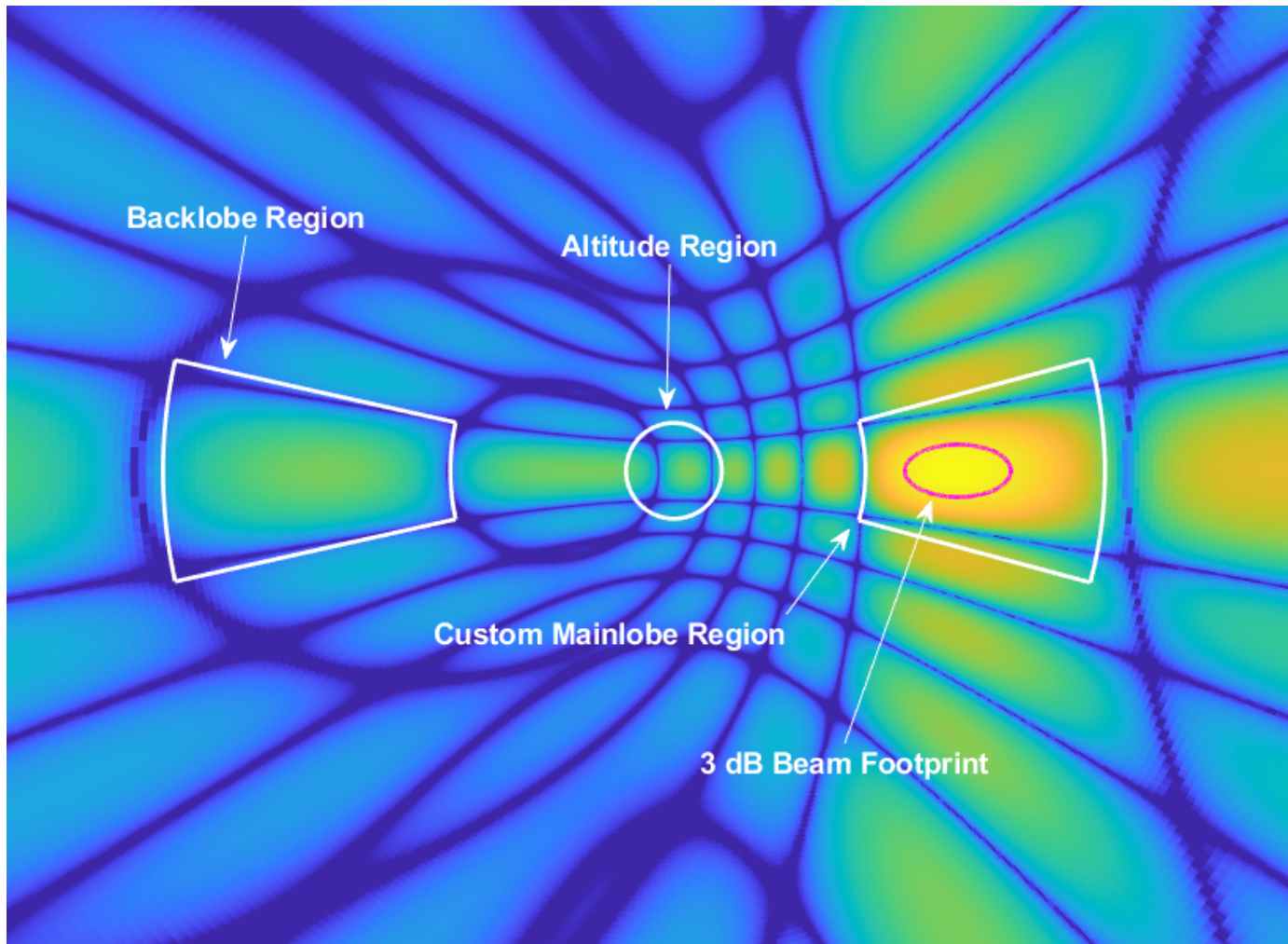
The `radarDataGenerator` is a statistics-based detectability simulator, and only simulates mainlobe detections within the field of view. As such, having `UseBeam` of the `ClutterGenerator` set to true is sufficient to completely capture the effect of clutter interference on the detectability of target platforms when using a `radarDataGenerator`.

Visualize and Run Scenario

Theater Plotter

The `theaterPlot` object can be used along with a variety of theater plotters to create customizable visual representations of the scenario. Start by creating the theater plot.

```
tp = theaterPlot;
```

Now create plotter objects for the scenario surface, clutter regions, and resulting radar detections. The values specified for the `DisplayName` properties are used for the legend entries.

```
surfPlotter = surfacePlotter(tp,'DisplayName','Scenario Surface');
clutPlotter = clutterRegionPlotter(tp,'DisplayName','Clutter Region');
detPlotter = detectionPlotter(tp,'DisplayName','Radar Detections','Marker','.','MarkerEdgeColor'
```

Now that the scenario, clutter generator, and plotters are configured, use the `detect` method on the scenario to simulate a single frame and collect detections.

```
dets = detect(scenario);
```

Plot the clutter region, which in this case is simply the beam footprint, along with the detection positions. Since the land surface used here is unbounded, the `plotSurface` call should come last so that the surface plot extends over the appropriate axis limits. The `clutterRegionData` method on the clutter generator is used to get plot data for the clutter region plotter. Similarly, for the surface plotter, the `surfacePlotterData` method on the scenario surface manager is used.

```
plotClutterRegion(clutPlotter,clutterRegionData(clut))
detpos = cell2mat(cellfun(@(t) t.Measurement(1:3).',dets,'UniformOutput',0));
plotDetection(detPlotter,detpos)
plotSurface(surfPlotter,surfacePlotterData(scenario.SurfaceManager))
```
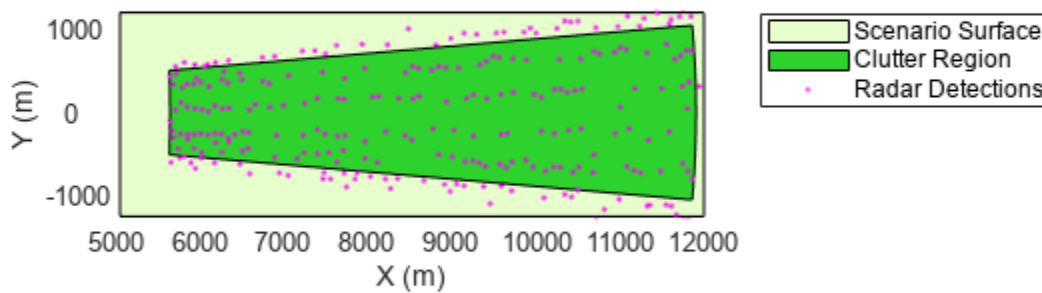

The detection positions can be seen arranged along radial lines corresponding to the radar's azimuth and Doppler resolution bins. The `radarDataGenerator` adds noise to the positions of detections, so may return detections with positions that fall outside the beam footprint.

Simulate Clutter IQ Signals

Now you will create a `radarTransceiver` with similar radar system parameters and simulate clutter at the signal level. The function `helperMakeTransceiver` is provided to quickly create a transceiver with the desired system parameters.

Define the desired beamwidth. For comparison to the above scenario, simply let the beamwidth equal the field of view that was used.

```
beamwidth3dB = fov;
```

The resulting `radarTransceiver` will use a `phased.CustomAntennaElement` to approximate a uniform rectangular array with the specified beamwidth, which is recommended to speed up simulations when only a sum beam is needed.

```
useCustomElem = true;
rdriq = helperMakeTransceiver(beamwidth3dB,fc,rngRes,prf,useCustomElem);
```

Use the same mounting angles and number of pulses as before.

```
rdriq.MountingAngles = mountAng;
rdriq.NumRepetitions = numPulses;
```

Re-create the same scenario, using this new radar model. Start by calling `release` on System Objects that will be re-used.

```
release(rdrTraj)
scenario = radarScenario('UpdateRate',0,'IsEarthCentered',false);
platform(scenario,'Sensors',rdriq,'Trajectory',rdrTraj);
landSurface(scenario,'RadarReflectivity',refl);
```

Enable clutter generation for the radar. This time, disable the beam footprint clutter region in favor of a custom ring-shaped region.

```
clutterGenerator(scenario,rdriq,'Resolution',clutRes,'UseBeam',false,'RangeLimit',clutRngLimit);
```

If the `clutterGenerator` method was called without any output argument, as above, the handle to the constructed `ClutterGenerator` may still be found with the scenario `getClutterGenerator` method by passing in a handle to the associated radar.

```
clut = getClutterGenerator(scenario,rdriq);
```

After creating the `ClutterGenerator`, you can use the `ringClutterRegion` method to create a null-to-null footprint region for clutter generation. Use a simple estimate of the null-to-null beamwidth as about 2.5 times the 3 dB beamwidth, then find the minimum elevation angle to encompass the near edge of the beam, and finally convert that to a minimum ground range for the region.

```
beamwidthNN = 2.5*beamwidth3dB;
minel = -mountAng(2) - beamwidthNN(2)/2;
minrad = -rdrAlt/tand(minel);
```

For the max radius parameter, simply find the ground range corresponding to the clutter range limit specified earlier.

```
maxrad = sqrt(clut.RangeLimit^2 - rdrAlt^2);
```

The azimuth span will equal the null-to-null beamwidth, and the azimuth center will be 0 degrees since the beam is pointing along the +X direction in scenario coordinates.

```
azspan = beamwidthNN(1);
azc = 0;
ringClutterRegion(clut,minrad,maxrad,azspan,azc)
```

```
ans =
  RingClutterRegion with properties:

        MinRadius: 3.6213e+03
        MaxRadius: 1.1906e+04
      AzimuthSpan: 25
    AzimuthCenter: 0
```

Using the provided helper function, plot the ground-projected antenna pattern along with the ring clutter region you just created. The ring region created above nicely encompasses the entire mainlobe.

`helperPlotGroundProjectedPattern(clut)`
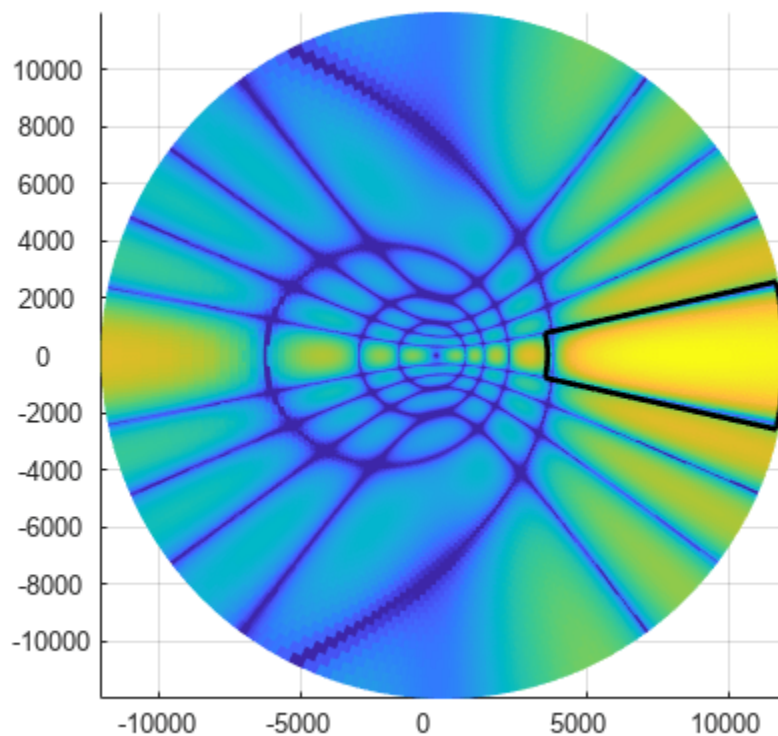
Run the simulation again for one frame, this time using the scenario `receive` method to simulate IQ signals.

```
iqsig = receive(scenario);
PH = iqsig{1};
```

Since the `radarTransceiver` used a single custom element, the resulting signal will be formatted with fast-time samples along the first dimension and pulse index (slow time) along the second dimension. This is the phase history (PH) matrix. Plot a DC-centered range-Doppler map (RDM) using the `helperPlotRDM` function.

```
figure
helperPlotRDM(PH,rngRes,prf,numPulses)
```
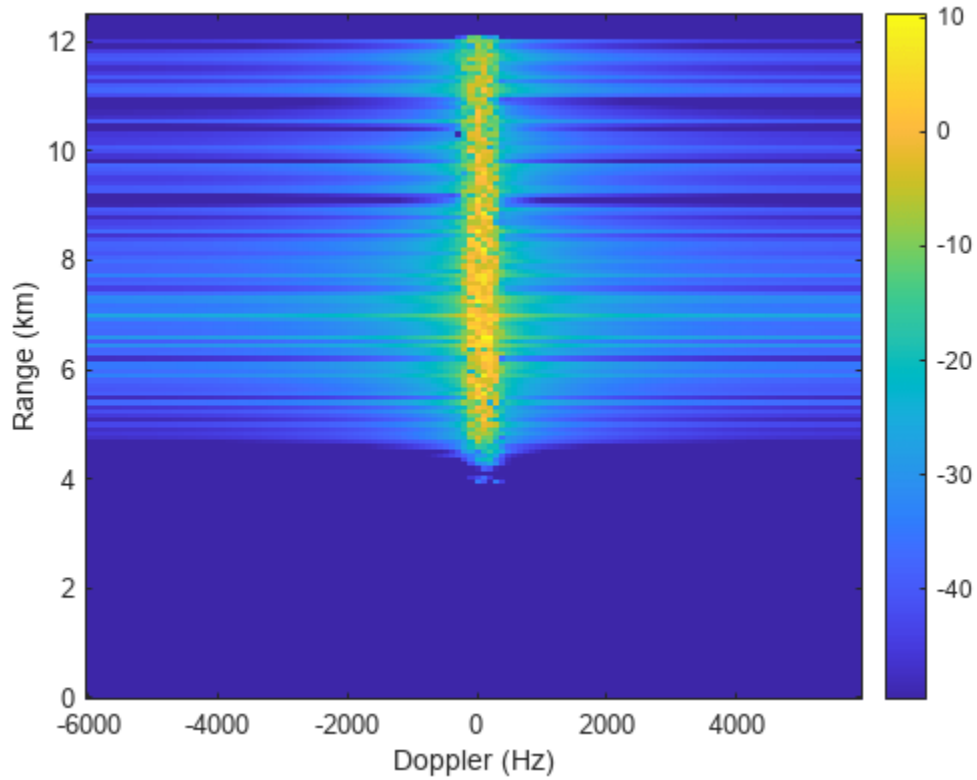

Use the provided helper function to re-create the theater plot visualization and view the ring clutter region.

```
helperTheaterPlot(clut)
```
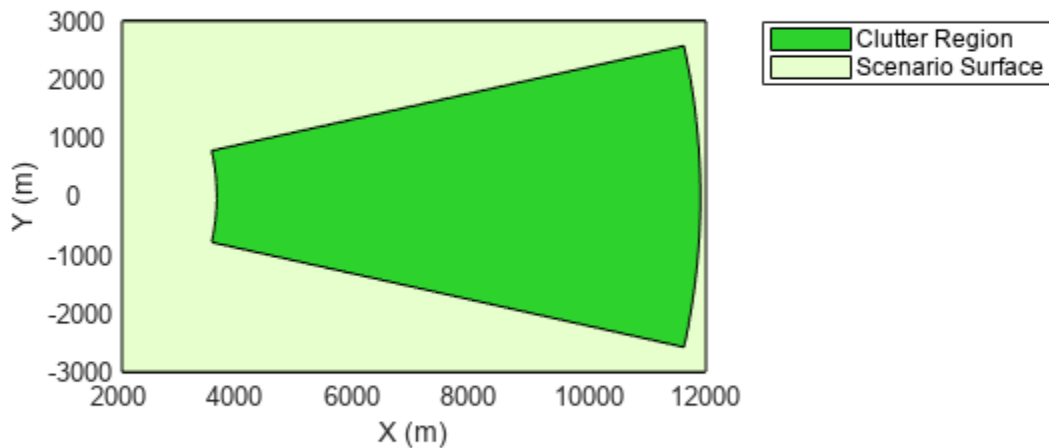
Simulate Surface Range Profiles for a Scanning Radar

The automatic mainlobe clutter option supports scanning radars. In this section you will re-create the scenario to use a stationary scanning linear array that collects a single pulse per scan position. You will add a few stationary surface targets and view the resulting range profiles.

Start by re-creating the radar object. This time, only pass the azimuth beamwidth to the helper function, which indicates a linear array should be used. The custom element cannot be used for a linear array if the automatic mainlobe clutter option is being used, so that the `ClutterGenerator` has knowledge of the array geometry. Reduce the range resolution to 40 meters to reduce the clutter power in gate.

```
useCustomElem = false;
rngRes = 40;
rdriq = helperMakeTransceiver(beamwidth3dB(1),fc,rngRes,prf,useCustomElem);
```

Set the same mounting angles as used earlier, and configure the transceiver for 1 repetition, which indicates a single pulse per scan position.

```
numPulses = 1;
rdriq.MountingAngles = mountAng;
rdriq.NumRepetitions = numPulses;
```

Now configure an electronic sector scan. Set the scanning limits to cover 30 degrees in azimuth, with no elevation scanning. Setting the scan rate to the PRF indicates a single pulse per scan position.

```matlab
rdriq.ElectronicScanMode = 'Sector';
rdriq.ElectronicScanLimits = [-15 15;0 0];
rdriq.ElectronicScanRate = [prf; 0];
```

Re-create the scenario and platform. Set the scenario stop time to run 1 full scan. Use a homogeneous unbounded land surface with the same reflectivity model as used earlier.

```matlab
scenario = radarScenario('UpdateRate',0,'IsEarthCentered',false,'StopTime',30/prf);
platform(scenario,'Sensors',rdriq,'Position',rdrPos,'Orientation',rotz(90).');
landSurface(scenario,'RadarReflectivity',refl);
```

Enable clutter generation, using only the 3 dB beam footprint for clutter generation.

```matlab
clutterGenerator(scenario,rdriq,'Resolution',clutRes,'UseBeam',true,'RangeLimit',clutRngLimit);
```

Add three bright point targets spaced 2 km apart along the cross-range direction at a down-range of 8 km.

```matlab
tgtRCS = 40; % dBsm
platform(scenario,'Position',[8e3 -2e3 0],'Signatures',rcsSignature('Pattern',tgtRCS));
platform(scenario,'Position',[8e3    0 0],'Signatures',rcsSignature('Pattern',tgtRCS));
platform(scenario,'Position',[8e3  2e3 0],'Signatures',rcsSignature('Pattern',tgtRCS));
```

Run the simulation, collecting the range profile at each scan position, and plotting with a scenario overview. Use the `info` output from `receive` to record the look angle used by the radar on each frame.

```matlab
rangeGates = 0:rngRes:(unambRange-rngRes);
frame = 0;
while advance(scenario)
    frame = frame + 1;

    [iqsig,info] = receive(scenario);

    lookAng(:,frame) = info.ElectronicAngle;
    rangeProfiles(:,frame) = 20*log10(abs(sum(iqsig{1},2)));

    if frame == 1
        % Initial plotting
        ax(1) = subplot(1,2,1);
        helperPlotClutterScenario(scenario,[],[],ax(1))
        ax(2) = subplot(1,2,2);
        rpHndl = plot(ax(2),rangeGates/1e3,rangeProfiles(:,frame));
        tHndl=title(sprintf('Frame: %d, Azimuth: %.1f deg',frame,lookAng(1,frame)));
        grid on
        xlabel('Range (km)')
        ylabel('Range Profile (dBW)')
    else
        % Update plots
        helperPlotClutterScenario(scenario,[],[],ax(1))
        rpHndl.YData = rangeProfiles(:,frame);
        tHndl.String = sprintf('Frame: %d, Azimuth: %.1f deg',frame,lookAng(1,frame));
    end

    drawnow limitrate nocallbacks
end
```
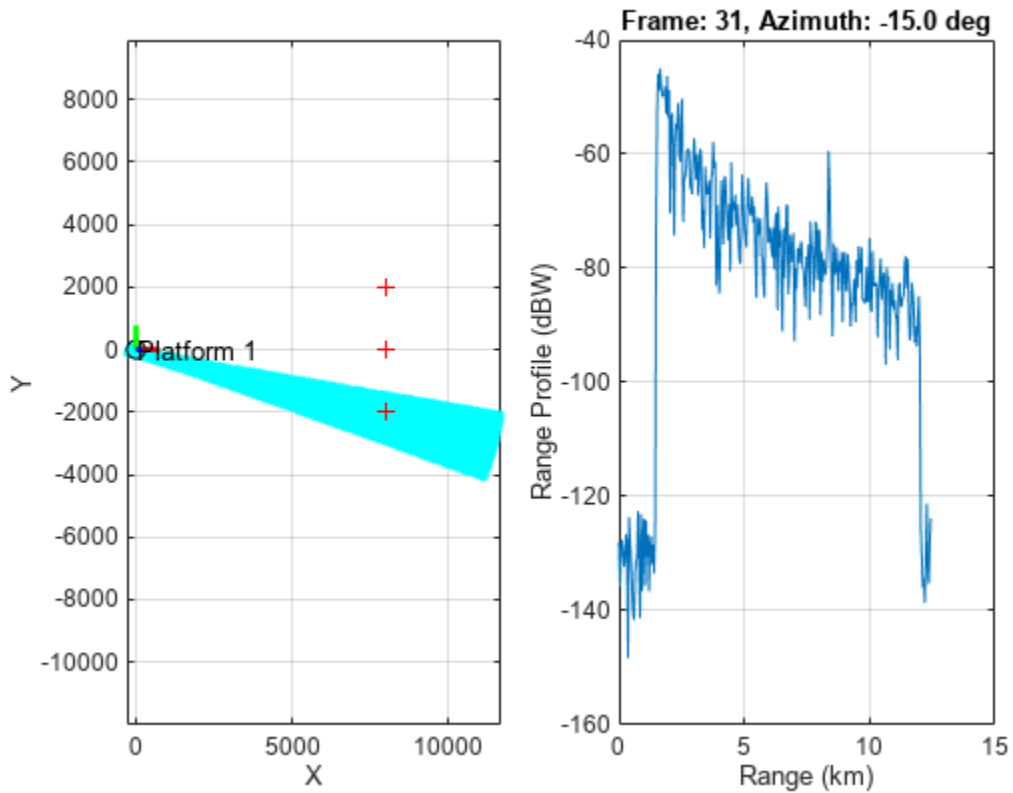
Plot the range profiles against range and azimuth scan angle.

```
figure
imagesc(lookAng(1,:),rangeGates/1e3,rangeProfiles);
set(gca,'ydir','normal')
xlabel('Azimuth Scan Angle (deg)')
ylabel('Range (km)')
title('Clutter Range Profiles (dBW)')
colorbar
```

Clutter Range Profiles (dBW)
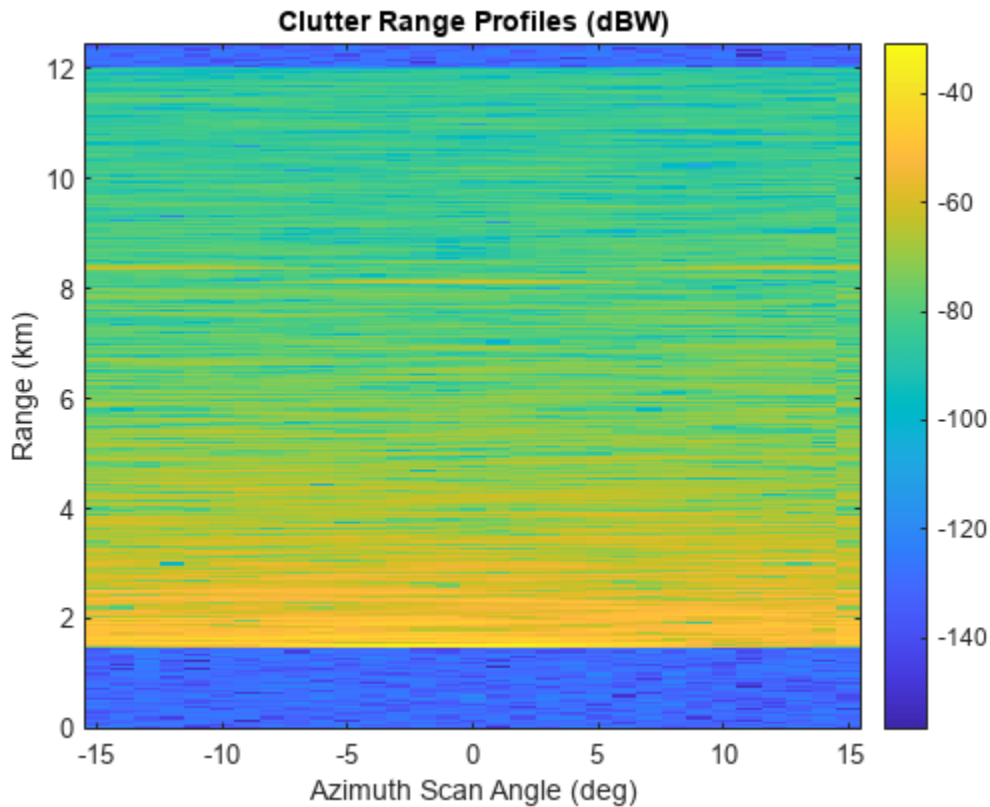
The three target signals are barely visible around 8 km range.

Simulate Smooth Surface Clutter for a Range-Doppler Radar

Up till now you have simulated surface clutter using the "uniform" scatterer distribution mode. For flat-Earth scenarios, the `radarTransceiver` radar model, and smooth surfaces (no terrain or spectral model associated with the surface), a faster range-Doppler-adaptive mode is available which uses a minimal number of clutter scatterers and a more accurate calculation of the clutter power in each range-Doppler resolution cell.

Re-create the radarTransceiver, again with a linear array. The automatic mainlobe region will not be used in this section, so use a custom element to speed things up.

```
useCustomElem = true;
rdriq = helperMakeTransceiver(beamwidth3dB(1),fc,rngRes,prf,useCustomElem);
```

This time, instead of scanning, you will just simulate a single frame with 128 pulses, and perform Doppler processing. The `NumRepetitions` property, along with the specified PRF, determines the Doppler resolution of the adaptive scatterers.

```
numPulses = 128;
rdriq.MountingAngles = mountAng;
rdriq.NumRepetitions = numPulses;
```

Create the scenario as before with the simple homogenous surface.

```
scenario = radarScenario('UpdateRate',0,'IsEarthCentered',false);
landSurface(scenario,'RadarReflectivity',refl);
```

1-783

The radar is again flying in the +Y direction while facing in the +X direction.

```
release(rdrTraj)
platform(scenario,'Sensors',rdriq,'Trajectory',rdrTraj);
```

Enable clutter generation. To use the range-Doppler-adaptive scatterers, specify "RangeDopplerCells" for the ScattererDistribution property. To encompass the three targets, use a ring-shaped region with 60 degrees of azimuth span, and the same min/max radius used earlier.

```
clut = clutterGenerator(scenario,rdriq,'ScattererDistribution','RangeDopplerCells','UseBeam',fals
ringClutterRegion(clut,minrad,maxrad,60,0);
```

Add the same three bright targets.

```
platform(scenario,'Position',[8e3 -2e3 0],'Signatures',rcsSignature('Pattern',tgtRCS));
platform(scenario,'Position',[8e3    0 0],'Signatures',rcsSignature('Pattern',tgtRCS));
platform(scenario,'Position',[8e3  2e3 0],'Signatures',rcsSignature('Pattern',tgtRCS));
```

Run the simulation for a single frame, form the sum beam, and plot the RDM.

```
iqsig = receive(scenario);
PH = iqsig{1};
helperPlotRDM(PH,rngRes,prf,numPulses);
```
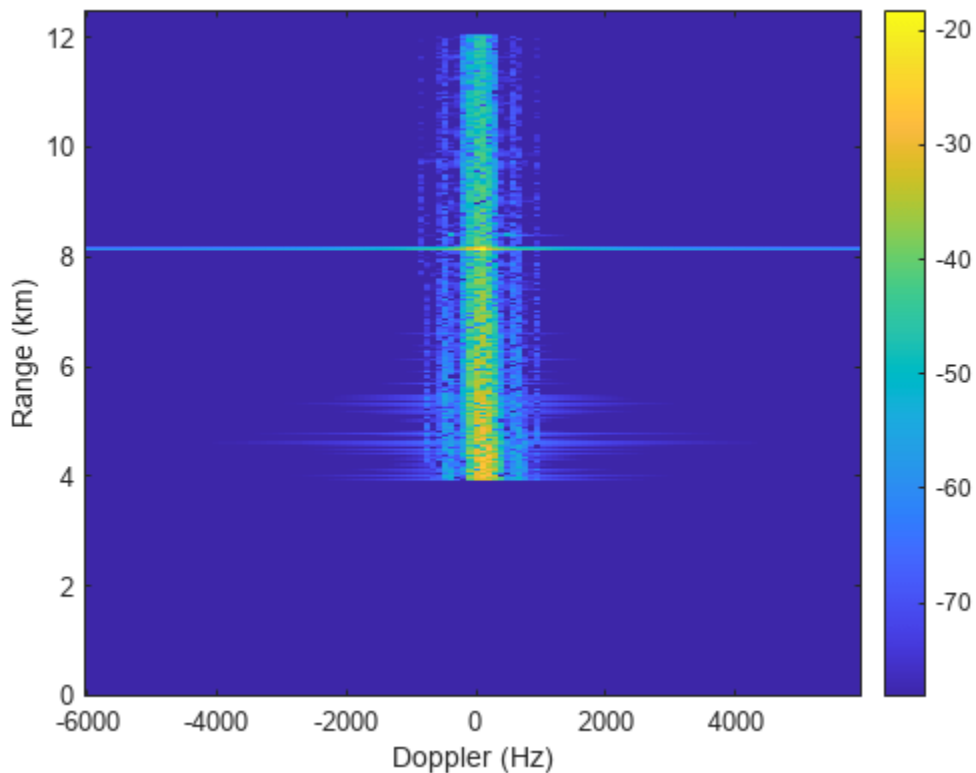

The targets are more visible than in the range-angle case thanks to the Doppler spreading of clutter. With such a large clutter region, the same scenario could be expected to take upwards of 35x longer to simulate with the uniform scatterer distribution.

Clutter from Terrain Data

In the previous sections, you simulated homogeneous clutter from an unbounded flat surface. In this section, you will use a DTED file to simulate clutter return from real terrain data in an Earth-centered scenario. You will collect two frames of clutter return - one with shadowing enabled and one without shadowing, and compare the results.

Start by creating the scenario, this time setting the `IsEarthCentered` flag to true in order to use a DTED file, which consists of surface height samples over a latitude/longitude grid.

```
scenario = radarScenario('UpdateRate',0,'IsEarthCentered',true);
```

Again use the `landSurface` method, passing in the name of the desired DTED file for the value of the `Terrain` parameter. The `Boundary` parameter can be used to constrain the domain of the loaded data. In general, as little of the terrain data should be loaded as needed for the specific application. In this case, you will use a 0.15-by-0.15 degree section of DTED that is referenced to a given latitude-longitude coordinate.

```
refLLA = [39.43; -105.84];
bdry = refLLA + [0 1;-1/2 1/2]*0.15;
```

Continue using the same constant-gamma reflectivity model. Use an output argument to get a handle to the created surface object.

```
srf = landSurface(scenario,'Terrain','n39_w106_3arc_v2.dt1','Boundary',bdry,'RadarReflectivity',
```

Use the surface `height` method to place the platform above the reference point at the altitude specified earlier.

```
srfHeight = height(srf,refLLA);
rdrAlt = srfHeight + rdrAlt;
rdrPos1 = [refLLA; rdrAlt];
```

In this scenario, the radar will travel in a straight line West at the same speed used earlier.

```
rdrVelWest = [-rdrSpd 0 0];
```

Earth-centered scenarios require trajectory information to be specified with waypoints in latitude/longitude/altitude (LLA) format using the `geoTrajectory` object. Set the `ReferenceFrame` to ENU, and use the enu2lla function to find the second waypoint corresponding to the desired velocity vector. If the orientation of the platform is not specified, it will be set automatically such that the platform +X direction corresponds to the direction of motion (West). As such, the -90 degree mounting yaw angle will point the radar North.

```
toa = [0;1]; % Times of arrival at each waypoint
rdrPos2 = enu2lla(rdrVelWest,rdrPos1.','ellipsoid').';
rdrTrajGeo = geoTrajectory('Waypoints',[rdrPos1, rdrPos2].','TimeOfArrival',toa,'ReferenceFrame'
platform(scenario,'Sensors',rdriq,'Trajectory',rdrTrajGeo);
```

Create the clutter generator. Surface shadowing is enabled by default.

```
clut = clutterGenerator(scenario,rdriq,'Resolution',clutRes,'UseBeam',false,'RangeLimit',clutRngl
```

Create a ring-shaped region with the same parameters as earlier. For Earth-centered scenarios, azimuth angles in scenario coordinates are referenced clockwise from North, so an azimuth center of 0 here still coincides with the radar's look direction.

```
ringClutterRegion(clut,minrad,maxrad,azspan,azc);
```

Simulate clutter return for one frame and save the resulting phase history.

```
iqsig = receive(scenario);
PH_withShadowing = iqsig{1};
```

The theater plot does not support visualizations of Earth-centered scenarios, so use the provided helper function to show a scenario overview in a local ENU frame. This time, the terrain is plotted along with the radar frame and clutter patches. Notice the shadowed region can be seen in the plot as a gap in the clutter patches (cyan). Those patches are obstructed by other parts of the terrain that are closer to the radar.

```
helperPlotClutterScenario(scenario)
title('Clutter patches - with terrain shadowing')
```


Clutter patches - with terrain shadowing
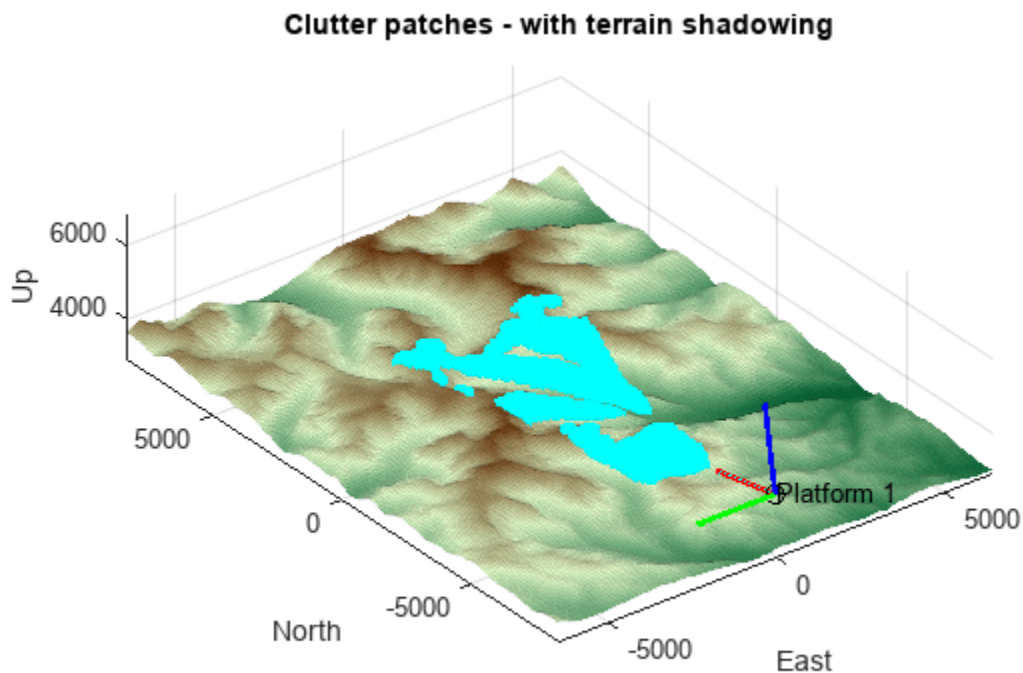
To see the difference when shadowing is not used, turn shadowing off, run the simulation for another frame, and save the result.

```
clut.UseShadowing = false;
iqsig = receive(scenario);
PH_noShadowing = iqsig{1};
```

Plot the scenario overview and notice that some of the clutter region has been filled in with clutter patches. Notice there are still some gaps visible. This is because clutter patches that are facing away from the radar, such as if they are on the other side of a hill, will never be visible, regardless of the value of the UseShadowing property.

```
helperPlotClutterScenario(scenario)
title('Clutter patches - without terrain shadowing')
```

Clutter patches - without terrain shadowing
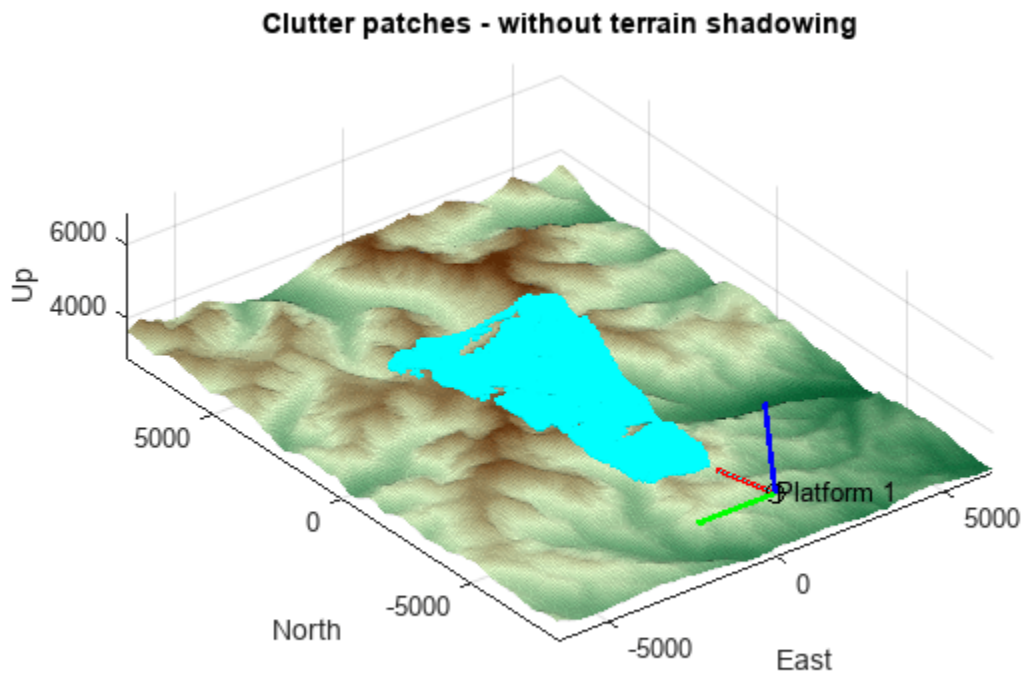
Now to see the effect of shadowing in the RDM, plot the clutter return signals recorded previously side-by-side.

```
figure
subplot(1,2,1)
helperPlotRDM(PH_withShadowing,rngRes,prf,numPulses)
title('RDM - with terrain shadowing')
subplot(1,2,2)
helperPlotRDM(PH_noShadowing,rngRes,prf,numPulses)
title('RDM - without terrain shadowing')
set(gcf,'Position',get(gcf,'Position')+[0 0 560 0])
```
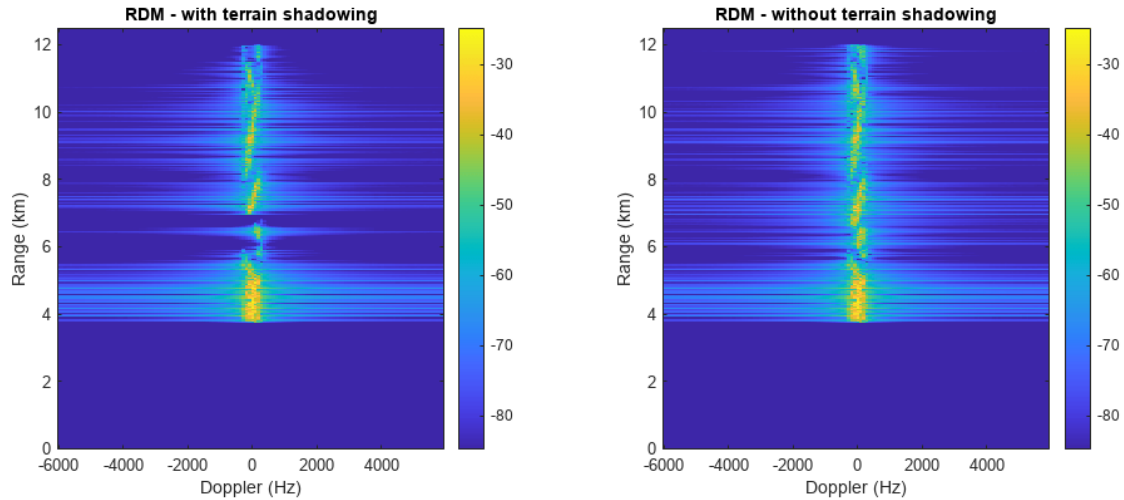
While both cases have "blank" regions due to hills that slope away from the radar, there is also a significant amount of surface return that is visible in the right figure and not visible in the left due to shadowing.

Shadowing is an important phenomenon when simulating clutter return from real surfaces, but it may be disabled for analysis purposes or if not needed.

Conclusion

In this example, you saw how to configure a radar scenario to include clutter return as part of the `detect` and `receive` methods, generating clutter detections and IQ signals with the `radarDataGenerator` and `radarTransceiver`, respectively. You saw how to define a region of the scenario surface with an associated reflectivity model, and how to specify regions of interest for clutter generation. Surface shadowing is simulated when generating clutter returns from surfaces with terrain, and a faster range-Doppler-adaptive mode can be used for flat-Earth scenarios with smooth surfaces.

Supporting Functions

helperTheaterPlot

```
function helperTheaterPlot(clut,params)

arguments
    clut
    params.Parent = []
    params.Detections = []
    params.ShowPatches = false
end

if isempty(params.Parent)
    % Make new figure for theater plot
    figure
    tp = theaterPlot('Parent',gca);
else
    % Use specified axes
    cla(params.Parent)
    tp = theaterPlot('Parent',params.Parent);
```

```
    end

    % Find targets
    tgts = clut.Scenario.Platforms(2:end);
    if ~isempty(tgts)
        tgtPos = cell2mat(cellfun(@(t) t.Position,tgts.','UniformOutput',0));
    end

    % Get detection positions
    if ~isempty(params.Detections)
        detPos = cell2mat(cellfun(@(t) t.Measurement(1:3).',params.Detections,'UniformOutput',0));
    end

    % Make plotters
    if ~isempty(tgts)
        platPlotter = platformPlotter(tp,'DisplayName','Target','Marker','+','MarkerEdgeColor','r');
    end
    if ~isempty(params.Detections)
        detPlotter = detectionPlotter(tp,'DisplayName','Radar Detections','Marker','.','MarkerEdgeCo
    end
    clutPlotter = clutterRegionPlotter(tp,'DisplayName','Clutter Region','ShowPatchCenters',params.Sh
    surfPlotter = surfacePlotter(tp,'DisplayName','Scenario Surface');

    % Do plotting
    if ~isempty(tgts)
        plotPlatform(platPlotter,tgtPos);
    end
    plotHeight = 1;
    plotClutterRegion(clutPlotter,clutterRegionData(clut,plotHeight))
    if ~isempty(params.Detections)
        plotDetection(detPlotter,detPos)
    end
    plotSurface(surfPlotter,surfacePlotterData(clut.Scenario.SurfaceManager))

end
```

helperMakeTransceiver

```
function rdr = helperMakeTransceiver( bw,fc,rangeRes,prf,useCustomElem )
% This helper function creates a radarTransceiver from some basic system
% parameters.

c = physconst('lightspeed');

rdr = radarTransceiver;

rdr.TransmitAntenna.OperatingFrequency = fc;
rdr.ReceiveAntenna.OperatingFrequency = fc;

rdr.Waveform.PRF = prf;

sampleRate = c/(2*rangeRes);
sampleRate = prf*round(sampleRate/prf); % adjust to match constraint with PRF

rdr.Receiver.SampleRate = sampleRate;
rdr.Waveform.SampleRate = sampleRate;

rdr.Waveform.PulseWidth = 2*rangeRes/c;
```

```
if isempty(bw)
    % Use an isotropic element
    rdr.TransmitAntenna.Sensor = phased.IsotropicAntennaElement;
    rdr.ReceiveAntenna.Sensor = phased.IsotropicAntennaElement;
else
    % Get the number of elements required to meet the specified beamwidth
    sinc3db = 0.8859;
    N = round(sinc3db*2./(bw(:).'*pi/180));
    N = flip(N);
    lambda = freq2wavelen(fc,c);
    if numel(N) == 1
        % Use a back-baffled ULA
        array = phased.ULA(N,lambda/2);
        array.Element.BackBaffled = true;
    else
        % Use URA
        array = phased.URA(N,lambda/2);
    end

    if useCustomElem
        % Use a custom element corresponding to the sum beam
        az = -180:.4:180;
        el = -90:.4:90;
        G = pattern(array,fc,az,el,'Type','efield','normalize',false);
        M = 20*log10(abs(G));
        P = angle(G);
        E = phased.CustomAntennaElement('FrequencyVector',[fc-1 fc+1],...
            'AzimuthAngles',az,'ElevationAngles',el,'MagnitudePattern',M,'PhasePattern',P);
        rdr.TransmitAntenna.Sensor = E;
        rdr.ReceiveAntenna.Sensor = E;
    else
        rdr.TransmitAntenna.Sensor = array;
        rdr.ReceiveAntenna.Sensor = array;
    end
end

end
```

helperPlotGroundProjectedPattern

```
function helperPlotGroundProjectedPattern( clut )
% Input a ClutterGenerator that has an associated radarTransceiver. Plots
% the clutter regions and ground-projected gain pattern. Assumes a flat
% infinite ground plane at Z=0.

Naz = 360*4;
Nel = 90*4;

% Force update the patch generator sensor data
clut.PatchGenerator.updateSensorData(clut.Platform,clut.Radar,clut.Scenario.SimulationTime,clut.l

pos = clut.PatchGenerator.SensorData.Position;
fc = clut.PatchGenerator.SensorData.CenterFrequency;
B = clut.PatchGenerator.SensorData.SensorFrame;
maxGndRng = clut.RangeLimit;

% Get azimuth/elevation grid in scenario coordinates
```

```matlab
azScen = linspace(-180,180,Naz);
maxEl = -atand(pos(3)/maxGndRng);
elScen = linspace(-90,maxEl,Nel);

% Convert az/el to sensor frame
[losxScen,losyScen,loszScen] = sph2cart(azScen*pi/180,elScen.'*pi/180,1);
R = B.';
losx = R(1,1)*losxScen + R(1,2)*losyScen + R(1,3)*loszScen;
losy = R(2,1)*losxScen + R(2,2)*losyScen + R(2,3)*loszScen;
losz = R(3,1)*losxScen + R(3,2)*losyScen + R(3,3)*loszScen;
[az,el,~] = cart2sph(losx,losy,losz);
az = az*180/pi;
el = el*180/pi;

% Get gain pattern
sensor = clut.Radar.TransmitAntenna.Sensor;
G = sensor(fc,[az(:) el(:)].');
G = reshape(G,size(az));
G = G/max(G(:));

rtg = -pos(3)./loszScen;
surf(losxScen.*rtg,losyScen.*rtg,zeros(size(az)),20*log10(abs(G)))
axis equal
shading flat
view(0,90)
clim([-60 0])

hold on
for reg = clut.Regions
    helperPlotClutterRegion(reg,pos);
end
hold off

end
```

helperPlotRDM

```matlab
function helperPlotRDM( PH,rangeRes,prf,numPulses )
% This helper function forms and plots an RDM from a phase history matrix

c = physconst('lightspeed');

% Form DC-centered RDM and convert to dBW
RDM = fftshift(fft(PH,[],2),2);
RDM = 20*log10(abs(RDM));

% Range and Doppler bins
rngBins = 0:rangeRes:c/(2*prf);
dopBins = -prf/2:prf/numPulses:prf/2-prf/numPulses;

% Plot
imagesc(dopBins,rngBins/1e3,RDM);
set(gca,'ydir','normal')
xlabel('Doppler (Hz)')
ylabel('Range (km)')
mx = max(RDM(:));
if ~isinf(mx)
    clim([mx-60 mx]);
```

```
        end
    colorbar

    end
```

Simulating Radar Returns from Moving Sea Surfaces

Maritime radar systems operate in a challenging, dynamic environment. To improve detection of targets of interest and assess system performance, it is important to understand the nature of returns from sea surfaces.

In this example, you will simulate an X-band radar system used for oceanographic studies of sea states. The radar system is a fixed, offshore platform. You will first generate a moving sea surface at sea state 4 using the Elfouhaily spectrum model. You will then generate IQ returns from the sea surface and investigate the statistics and time-frequency behavior of the simulated sea surface signals.

Define Radar System Parameters

Define an X-band radar system with a range resolution approximately equal to 30 meters. Verify the range resolution is as expected using the `bw2rangeres` function.

```
rng(2021)          % Initialize random number generator

% Radar parameters
freq = 9.39e9;  % Center frequency (Hz)
prf = 500;      % PRF (Hz)
tpd = 100e-6;   % Pulse width (s)
azBw = 2;       % Azimuth beamwidth (deg)
depang = 30;    % Depression angle (deg)
antGain = 45.7; % Antenna gain (dB)
fs = 10e6;      % Sampling frequency (Hz)
bw = 5e6;       % Waveform bandwidth (Hz)
bw2rangeres(bw)
```

```
ans = 29.9792
```

Next, define the sea surface as a sea state 4 with a wind speed of 19 knots and a resolution of 2 meters. Set the length of the sea surface to 512 meters.

```
% Sea surface
seaState = 4;   % Sea state number
vw = 19;        % Wind speed (knots)
L = 512;        % Sea surface length (m)
resSurface = 2; % Resolution of sea surface (m)

% Calculate wavelength and get speed of light
[lambda,c] = freq2wavelen(freq);

% Setup sensor trajectory and simulation times
rdrht = 100;                                      % Radar platform height (m)
rdrpos = [-L/2 0 rdrht];                          % Radar position [x y z] (m)
numPulses = 1024;                                 % Number of pulses
scenarioUpdateTime = 1/prf;                       % Scenario update time (sec)
scenarioUpdateRate = prf;                         % Scenario update rate (Hz)
simTime = scenarioUpdateTime.*(numPulses - 1);    % Overall simulation time (sec)
```

Model the Sea Surface

Define a scenario with `radarScenario`. Next, add sea motion with `seaSpectrum`. Keep the `SpectrumSource` property default `Auto` to use an Elfouhaily spectrum. Set the resolution to 2

meters. The Elfouhaily model consists of a sea spectrum and an angular spreading function. The sea spectrum model gets physical properties like wind speed and fetch from the `seaSurface`, as you will see later in this section.

You can also import your own custom model by setting the `seaSpectrum SpectrumSource` property to `Custom`. Alternative sea spectrum models in the literature include the JONSWAP, Donelan-Pierson, and Pierson-Moskovitz spectrums.

```
% Create scenario
scene = radarScenario('UpdateRate',scenarioUpdateRate, ...
    'IsEarthCentered',false,'StopTime',simTime);

% Define Elfouhaily sea spectrum
seaSpec = seaSpectrum('Resolution',resSurface) % See Elfouhaily reference

seaSpec =
  seaSpectrum with properties:

    SpectrumSource: 'Auto'
        Resolution: 2
```

Now, select a reflectivity model. Radar Toolbox™ offers 9 different reflectivity models for sea surfaces covering a wide range of frequencies, grazing angles, and sea states. The asterisk denotes the default model. Type `help surfaceReflectivitySea` or `doc surfaceReflectivitySea` in the Command Window for more information on usage and applicable grazing angles for each model. The sea reflectivity models are as follows.

- **APL:** Semi-empirical model supporting low grazing angles over frequencies in the range from 1 to 100 GHz. Applicable for sea states 1 through 6.
- **GIT:** Semi-empirical model supporting low grazing angles over frequencies in the range from 1 to 100 GHz. Applicable for sea states 1 through 6.
- **Hybrid:** Semi-empirical model supporting low to medium grazing angles over frequencies in the range from 500 MHz to 35 GHz. Applicable for sea states 0 through 5.
- **Masuko:** Empirical model supporting low grazing angles over frequencies in the range from 8 to 40 GHz. Applicable for sea states 1 through 6.
- **Nathanson:** Empirical model supporting low grazing angles over frequencies in the range from 300 MHz to 36 GHz. Applicable for sea states 0 through 6.
- **NRL*:** Empirical model supporting low grazing angles over frequencies in the range from 500 MHz to 35 GHz. Applicable for sea states 0 through 6.
- **RRE:** Mathematical model supporting low grazing angles over frequencies in the range from 9 to 10 GHz. Applicable for sea states 1 through 6.
- **Sittrop:** Empirical model supporting low grazing angles over frequencies in the range from 8 to 12 GHz. Applicable for sea states 0 through 7.
- **TSC:** Empirical model supporting low grazing angles over frequencies in the range from 500 MHz to 35 GHz. Applicable for sea states 0 through 5.

For this example, set the reflectivity to the GIT (Georgia Institute of Technology) model. For a general discussion about sea reflectivity models and how they are affected by frequency, grazing angle, sea state, and polarization, see the "Maritime Radar Sea Clutter Modeling" on page 1-443 example.

```
% Define reflectivity model
pol = 'V'; % Polarization
reflectModel = surfaceReflectivity('Sea','Model','GIT','SeaState',seaState,'Polarization',pol)

reflectModel =
  surfaceReflectivitySea with properties:

    EnablePolarization: 0
                 Model: 'GIT'
              SeaState: 4
          Polarization: 'V'
               Speckle: 'None'
```

Add a sea surface to the radar scenario using seaSurface. Assign the previously defined sea spectrum and reflectivity model to the sea surface. seaSurface houses the physical properties of the surface. The WindSpeed, WindDirection, and Fetch properties are used in generating the Elfouhaily sea spectrum. By thoughtful selection of these properties and the resolution of the sea surface, gravity and capillary waves with different wave ages can be produced.

Capillary waves are high-frequency, small-wavelength waves that have a rounded crest with a V-shaped trough that are less than 2 cm. Gravity waves are longer, low-frequency waves. The seaSpectrum object cannot model breaking waves, which are waves that reach a critical amplitude resulting in wave energy being transformed into turbulent, kinetic energy. The greater the wind speed, the greater the energy transferred to the sea surface and the larger the waves.

Fetch is the length of water over which wind blows without obstruction. The Elfouhaily model relates the fetch to the inverse wave age Ω_C as

$$\Omega_C = 0.84\tanh\left(\left(\frac{X}{X_0}\right)^{0.4}\right)^{-0.75}.$$

X is calculated as

$k_0 = \frac{g}{U_{10}}$ and $X = k_0$ Fetch,

where g is the acceleration due to gravity in meters per second squared, U_{10} is the wind speed at 10 meters above the surface in m/s, and X_0 is a constant set to 2.2e4. A fully-developed sea has an inverse wave age of 0.84, a mature sea has a value of approximately 1, and a young sea has a value between 2 and 5.

1-795

The default fetch for `seaSurface` is infinity, which produces a fully-developed sea. More information about the Elfouhaily spectrum can be found in the reference listed at the end of this example, as well as in the documentation for `seaSpectrum`.

The elevation values of the sea surface can be obtained by using the `height` method on the sea surface. Plot the radar and the sea surface using `helperSeaSurfacePlot` on page 1-806.

```
% Configure sea surface
knots2mps = 0.514444; % Knots to meters/sec
vw = vw*knots2mps; % Wind speed (m/s)
seaSurf = seaSurface(scene,'SpectralModel',seaSpec,'RadarReflectivity',reflectModel, ...
    'WindSpeed',vw,'WindDirection',0,'Boundary',[-L/2 L/2; -L/2 L/2])

seaSurf =
  SeaSurface with properties:

            WindSpeed: 9.7744
        WindDirection: 0
                Fetch: Inf
        SpectralModel: [1x1 seaSpectrum]
    RadarReflectivity: [1x1 surfaceReflectivitySea]
      ReflectivityMap: 1
      ReferenceHeight: 0
             Boundary: [2x2 double]


% Plot sea surface and radar
x = -L/2:resSurface:(L/2 - 1);
y = -L/2:resSurface:(L/2 - 1);
[xGrid,yGrid] = meshgrid(x,y);
z = height(seaSurf,[xGrid(:).'; yGrid(:).'],scene.SimulationTime);
helperSeaSurfacePlot(x,y,z,rdrpos)
```
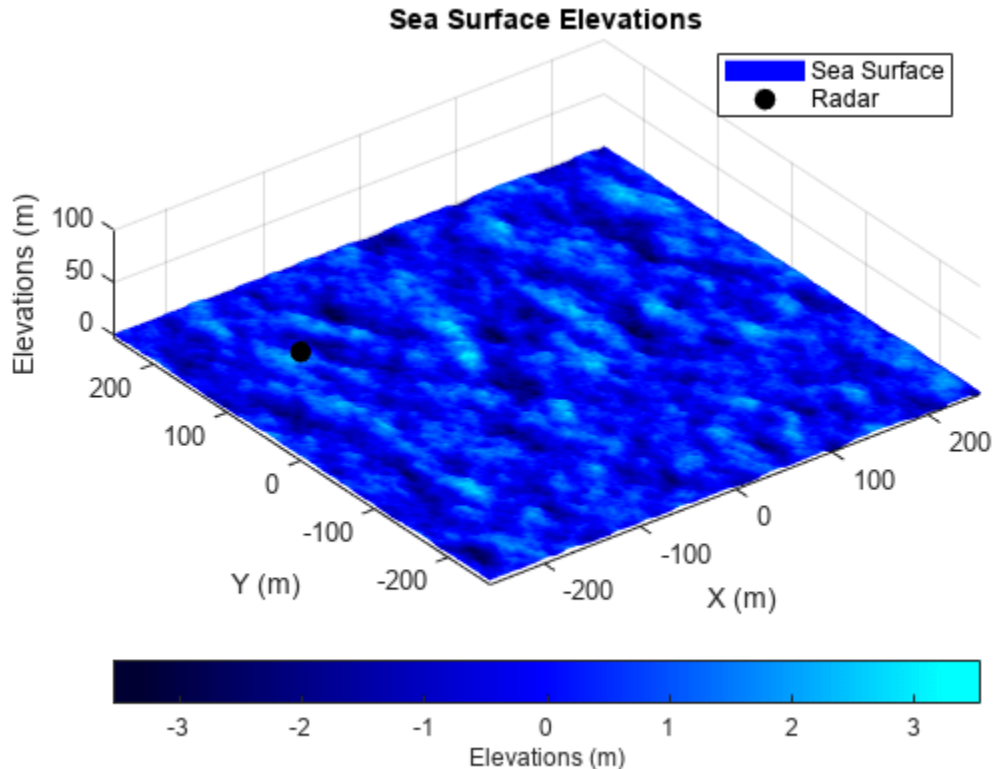
Investigate the statistics of the elevation values of the sea surface. First, calculate and plot the cumulative distribution function using `helperSeaSurfaceCDF` on page 1-806. You can use this plot to determine the probability that an observation will be less than or equal to a certain elevation value. For example, 90% of elevations will be less than or equal to about 1 meter. 95% of elevations will be less than or equal to about 1.5 meters. If you would like to alter the statistics of the generated sea surface to increase the elevations, you can increase the wind speed to deliver more energy to the surface.

```
% CDF
helperSeaSurfaceCDF(z)
```

Wave Elevation CDF
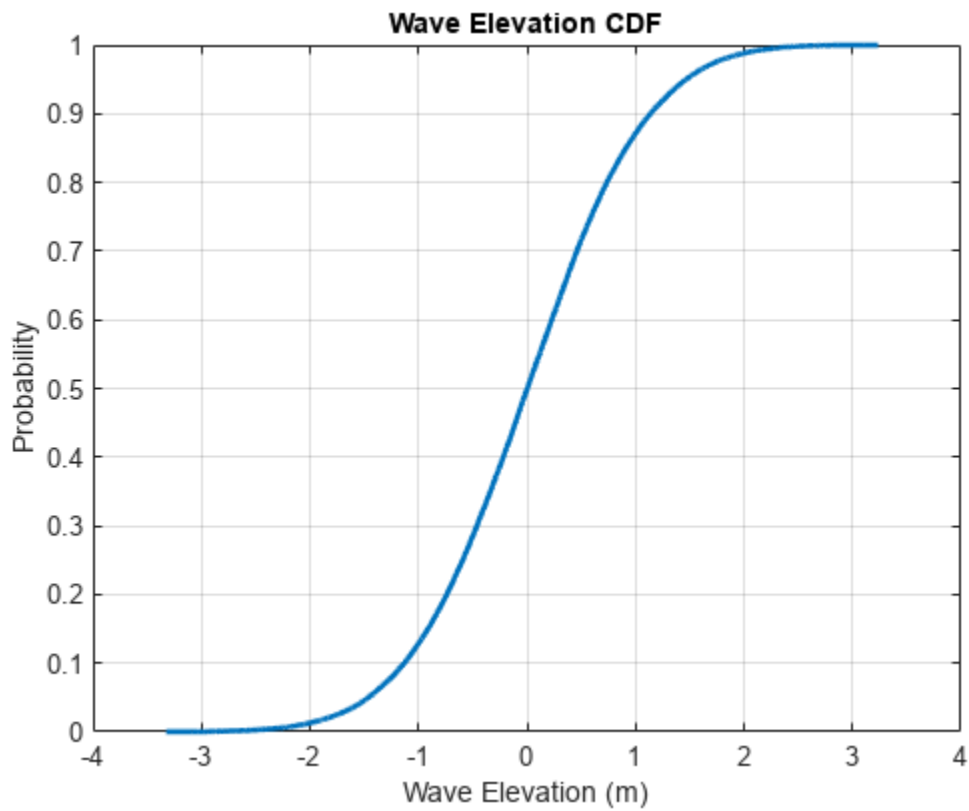
Next, estimate the significant wave height using `helperEstimateSignificantWaveHeight` on page 1-807. The estimate is obtained by taking the mean of the highest 1/3 of the wave heights, where the wave height is defined from trough-to-crest as is visually represented in the figure.

```
% Significant wave height
actSigHgt = helperEstimateSignificantWaveHeight(x,y,z)
```

Wave Height Estimation
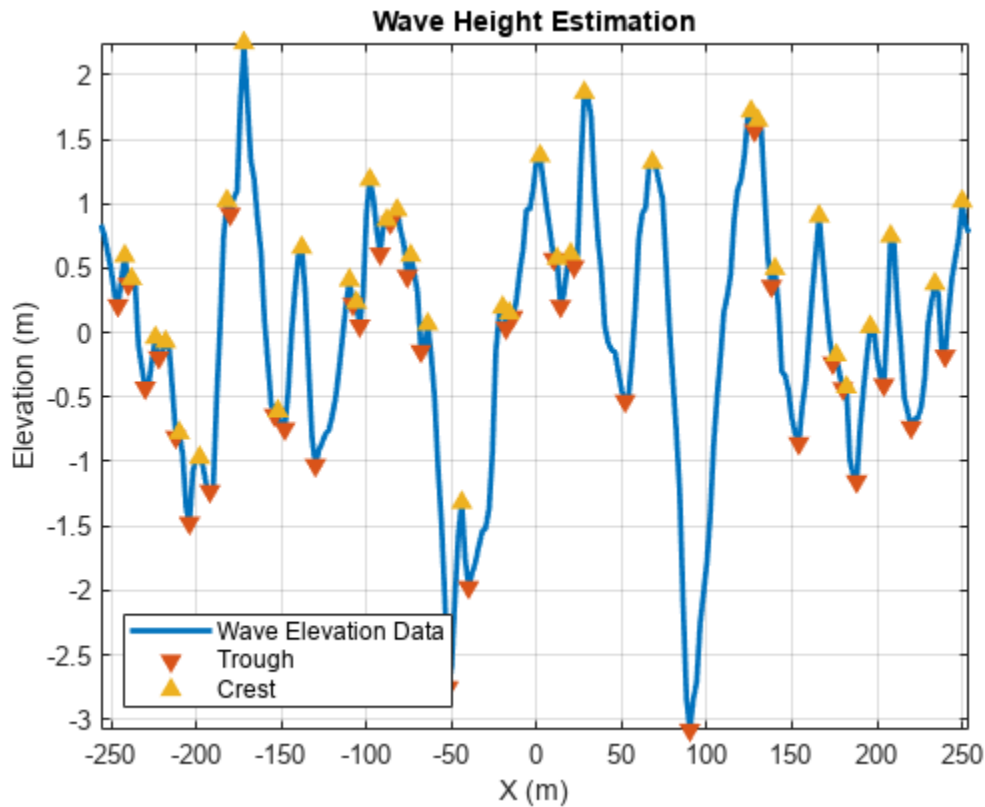

```
actSigHgt = 1.6340
```

The expected wave heights for a sea state 4 is in the range between 1.25 and 2.5 meters. Note that the simulated sea surface is within the range of expected values.

```
expectedSigHgt = [1.25 2.5]; % Sea state 4
actSigHgt >= expectedSigHgt(1) && actSigHgt <= expectedSigHgt(2)
```

```
ans = logical
   1
```

Lastly, plot a subset of the sea surface heights over time to visualize the motion due to the Elfouhaily sea spectrum. Note that the simulation time for the radar data collection only runs for 2 seconds, but plot over a time period of 10 seconds to get a better sense of the motion. Uncomment `helperSeaSurfaceMotionPlot` on page 1-808 to plot the moving sea surface.

```
% Plot sea surface motion
plotTime = 0:0.5:10; % Plot time (sec)
% helperSeaSurfaceMotionPlot(x,y,seaSurf,plotTime); % Uncomment to plot motion
```
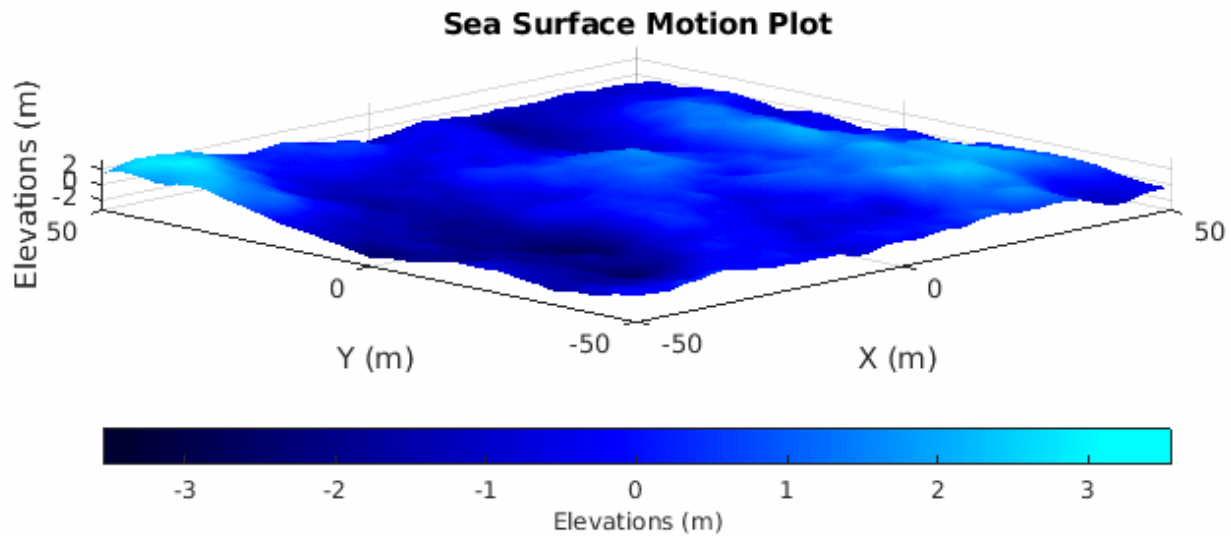
Sea Surface Motion Plot

Configure the Radar Transceiver

In this section, configure the radar system properties. Define the antenna and the transmitted linear frequency modulated (LFM) waveform. Assign the radar sensor to the radar platform.

```
% Create a radar platform using the trajectory information
rdrplat = platform(scene,'Position',rdrpos);

% Create a radar sensor looking to the East
rdr = radarTransceiver('MountingAngles',[0 depang 0],'NumRepetitions',1);

% Configure the LFM signal of the radar
rdr.Waveform = phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',tpd, ...
    'PRF',prf,'SweepBandwidth',bw);

% Set receiver sample rate and noise figure
rdr.Receiver.SampleRate = fs;
rdr.Receiver.NoiseFigure = 10;

% Define transmit and receive antenna and corresponding parameters
ant = phased.SincAntennaElement('Beamwidth',azBw);
rdr.TransmitAntenna.OperatingFrequency = freq;
rdr.ReceiveAntenna.OperatingFrequency = freq;
rdr.Transmitter.Gain = antGain;
rdr.Receiver.Gain = antGain;
rdr.TransmitAntenna.Sensor = ant;
rdr.ReceiveAntenna.Sensor = ant;

% Add radar to radar platform
rdrplat.Sensors = rdr;
```

Generate the Datacube

Now that the scene and the radar system have been defined, generate returns from the sea surface with the clutterGenerator method and collect the IQ data using the receive method, which both reside in the radarScenario object. By default, clutterGenerator simulates clutter returns in

the mainlobe. For more information about clutter modeling, see "Introduction to Radar Scenario Clutter Simulation" on page 1-770.

Use the helpers `helperPlotRawIQ` on page 1-809 and `helperUpdatePlotRawIQ` on page 1-809 to plot the unprocessed, raw IQ data.

```matlab
% Collect clutter returns with the clutterGenerator
clutterGenerator(scene,rdr);

% Run the scenario
numSamples = 1/prf*fs;
maxRange = 20e3;
Trng = (0:1/fs:(numSamples-1)/fs);
rngGrid = [0 time2range(Trng(2:end),c)];
[~,idxTruncate] = min(abs(rngGrid - maxRange));
iqsig = zeros(idxTruncate,numPulses);
ii = 1;
hRaw = helperPlotRawIQ(iqsig);
while advance(scene)
    tmp = receive(scene); % nsamp x 1
    iqsig(:,ii) = tmp{1}(1:idxTruncate);
    if mod(ii,128) == 0
        helperUpdatePlotRawIQ(hRaw,iqsig);
    end
    ii = ii + 1;
end
helperUpdatePlotRawIQ(hRaw,iqsig);
```
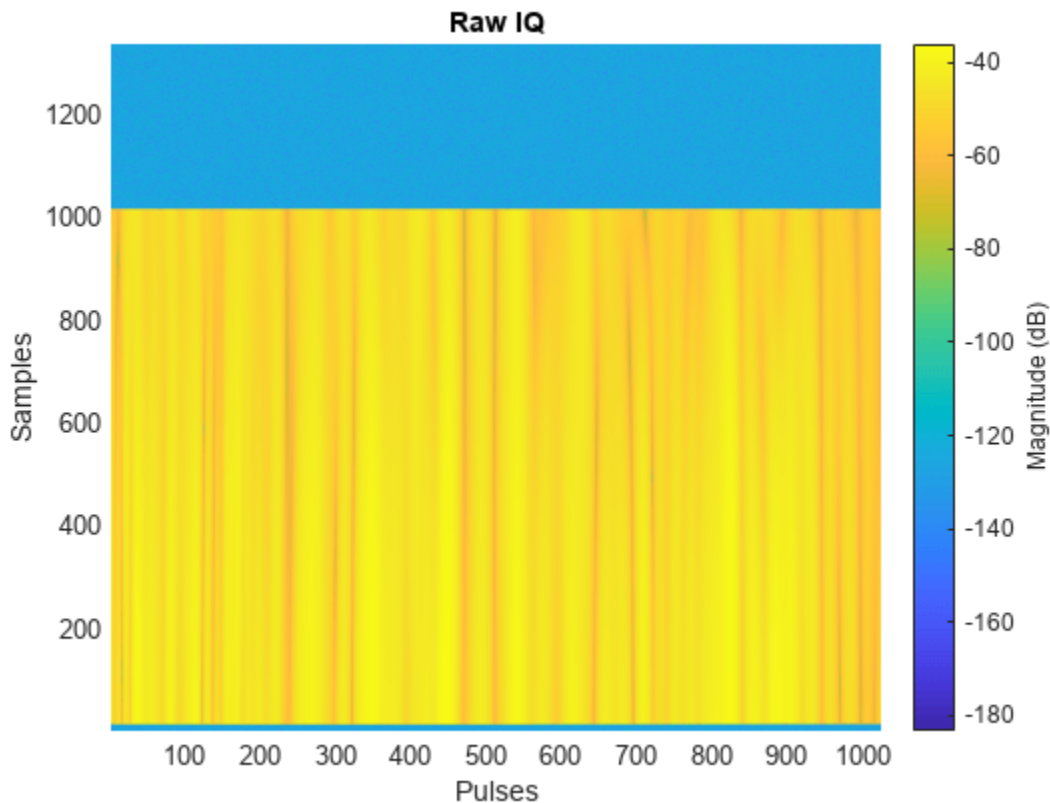
Process Sea Surface Returns

Use `phased.RangeResponse` to pulse compress the returned IQ data. Visualize the range-time behavior using `helperRangeTimePlot` on page 1-809. If the sea surface was static, you would see straight, horizontal lines in the plot, but the sea returns exhibit dynamism. At a given range, the magnitude changes with time. Note that the returns occur in a small subset of range due to the geometry (radar height, beamwidth, and depression angle).

```
% Pulse compress
matchingCoeff = getMatchedFilter(rdr.Waveform);
rngresp = phased.RangeResponse('RangeMethod', 'Matched filter', ...
    'PropagationSpeed',c,'SampleRate',fs);
[pcResp,rngGrid] = rngresp(iqsig,matchingCoeff);

% Plot
pcsigMagdB = mag2db(abs(pcResp));
[maxVal,maxIdx] = max(pcsigMagdB(:));
pcsigMagdB = pcsigMagdB - maxVal;
helperRangeTimePlot(rngGrid,prf,pcsigMagdB);
```
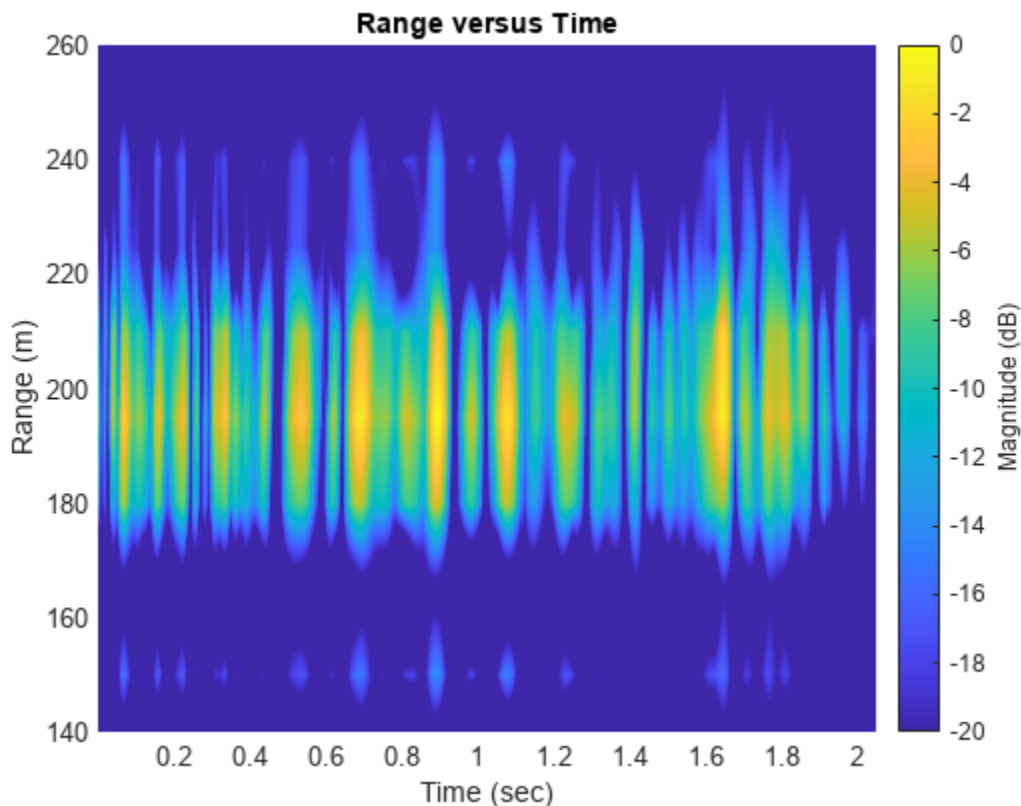

Use the range index of the maximum value and visualize the magnitude versus time at this range bin using `helperMagTimePlot` on page 1-810.

```
% Plot magnitude versus time
[idxRange,~] = ind2sub(size(pcsigMagdB),maxIdx);
helperMagTimePlot(pcsigMagdB(idxRange,:),numPulses,prf,rngGrid(idxRange));
```
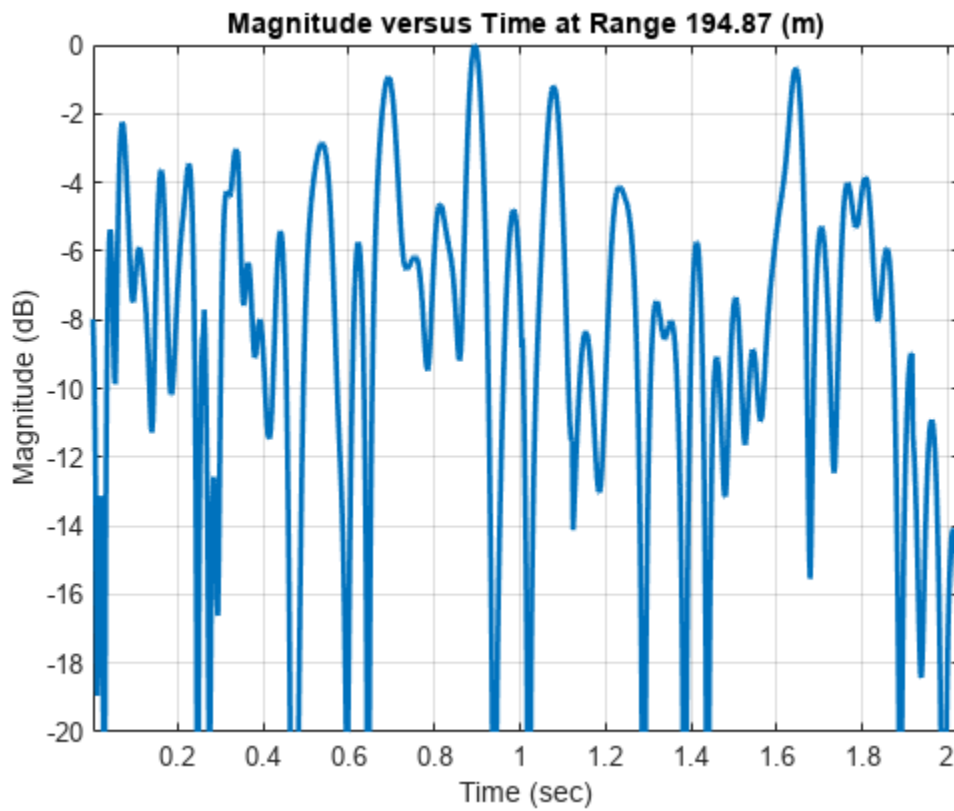
Next, generate a time-frequency plot of that same range bin using `helperSTFTPlot` on page 1-810. Generate the plot using the short-time Fourier transform function `stft`.

```
% STFT
[S,F,T] = stft(pcResp(idxRange,:),scenarioUpdateRate);
helperSTFTPlot(S,F,T,lambda,rngGrid(idxRange));
```

STFT at Range 194.87 (m)
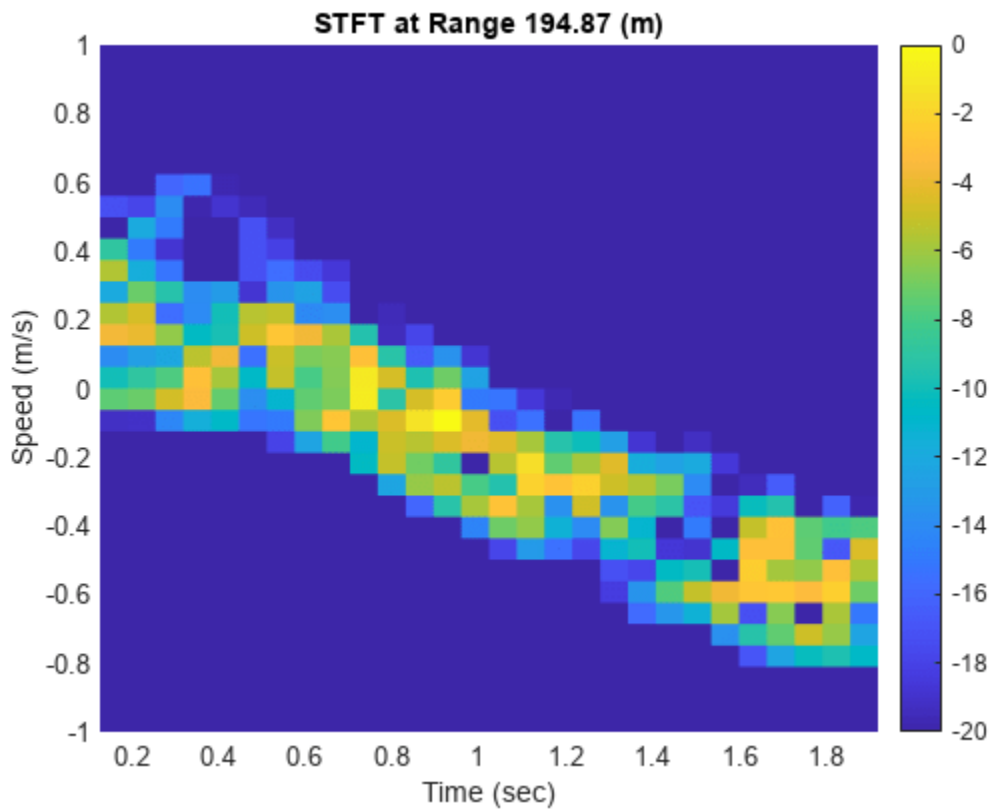
Notice that the detected speed of the range bin changes over time due to the motion of the sea surface. Over longer simulation times, periodicities are detectable.

Lastly, form a histogram from the magnitude values of the data in the ranges between 180 and 210 meters using `helperHistPlot` on page 1-810. Note the shape of the histogram resembles a Weibull distribution. If you have the Statistics and Machine Learning Toolbox™, you can use the `fitdist` function to get the parameters for the Weibull distribution from the magnitude of the data.

```
% Look at a subset of data in range and convert to decibel scale
[~,idxMin] = min(abs(rngGrid - 180));
[~,idxMax] = min(abs(rngGrid - 210));
pcsigMagdB = mag2db(abs(pcResp(idxMin:idxMax,:)));

% Remove any inf values
pcsigMagdB(isinf(pcsigMagdB(:))) = [];

% Shift values to be positive
pcsigMagdB = pcsigMagdB(:) - min(pcsigMagdB(:)) + eps;

% Weibull parameters
% Note: These values can be obtained if you use fitdist(pcsigMagdB,'weibull')
scale = 37.8589;
shape = 7.80291;

% Plot histogram with theoretical PDF overlay
helperHistPlot(pcsigMagdB,scale,shape);
```

Amplitude Histogram
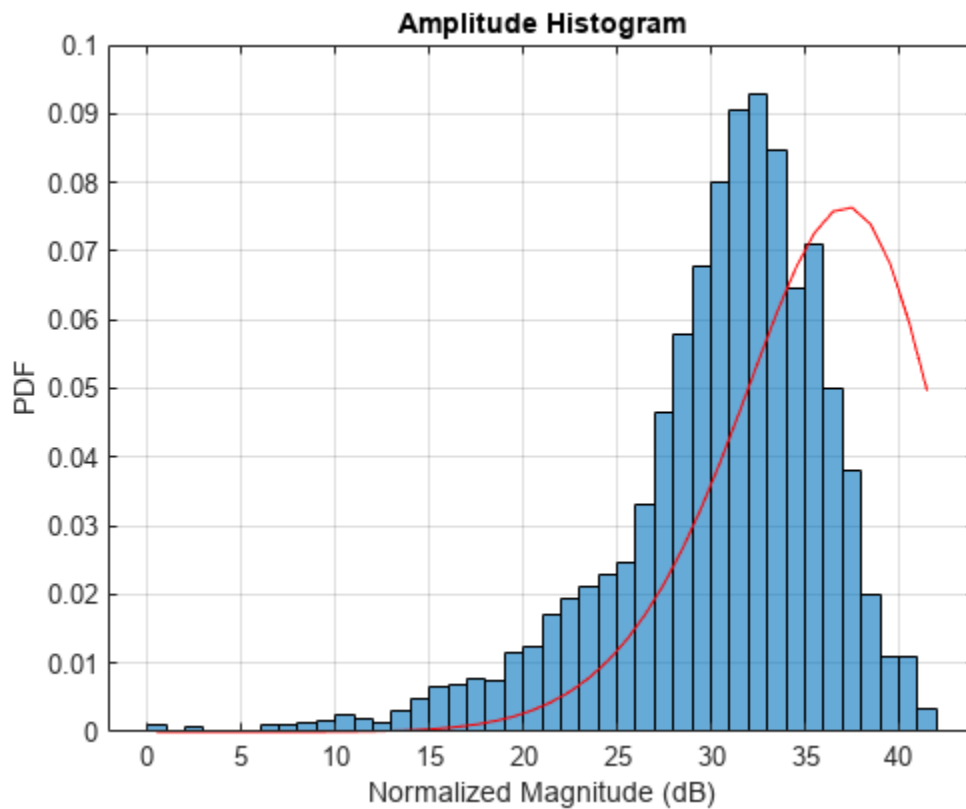
Summary

In this example, you learned how to:

- Build a radar scenario,
- Model a moving sea surface,
- Investigate sea surface statistics and behavior,
- Generate IQ, and
- Perform time-frequency processing on simulated IQ returns.

This example can easily be extended to include targets to support applications ranging such as maritime surveillance and radar imaging.

References

1 Apel, John R. *Principles of Ocean Physics*. International Geophysics Series, v. 38. London; Orlando: Academic Press, 1987.

2 Elfouhaily, T., B. Chapron, K. Katsaros, and D. Vandemark. "A Unified Directional Spectrum for Long and Short Wind-Driven Waves." *Journal of Geophysical Research: Oceans* 102, no. C7 (July 15, 1997): 15781-96. https://doi.org/10.1029/97JC00467.

3 Doerry, Armin W. "Ship Dynamics for Maritime ISAR Imaging ." Sandia Report. Albuquerque, New Mexico: Sandia National Laboratories, February 2008.

4 Ward, Keith D., Simon Watts, and Robert J. A. Tough. *Sea Clutter: Scattering, the K-Distribution and Radar Performance*. IET Radar, Sonar, Navigation and Avionics Series 20. London: Institution of Engineering and Technology, 2006.

Supporting Functions

helperSeaColorMap

```matlab
function cmap = helperSeaColorMap(n)
% Color map for sea elevation plot

c = hsv2rgb([2/3 1 0.2; 2/3 1 1; 0.5 1 1]);
cmap = zeros(n,3);
cmap(:,1) = interp1(1:3,c(:,1),linspace(1,3,n));
cmap(:,2) = interp1(1:3,c(:,2),linspace(1,3,n));
cmap(:,3) = interp1(1:3,c(:,3),linspace(1,3,n));
end
```

helperSeaSurfacePlot

```matlab
function helperSeaSurfacePlot(x,y,z,rdrpos)
% Plot sea elevations

% Color map for sea elevation plot
seaColorMap = helperSeaColorMap(256);

% Plot
figure
z = reshape(z,numel(x),numel(y));
surf(x,y,z)
hold on
plot3(rdrpos(1),rdrpos(2),rdrpos(3),'ok','LineWidth',2,'MarkerFaceColor','k')
legend('Sea Surface','Radar','Location','Best')
shading interp
axis equal
xlabel('X (m)')
ylabel('Y (m)')
zlabel('Elevations (m)')
stdZ = std(z(:));
minC = -4*stdZ;
maxC = 4*stdZ;
minZ = min(minC,rdrpos(3));
maxZ = max(maxC,rdrpos(3));
title('Sea Surface Elevations')
axis tight
zlim([minZ maxZ])
hC = colorbar('southoutside');
colormap(seaColorMap)
hC.Label.String = 'Elevations (m)';
hC.Limits = [minC maxC];
drawnow
pause(0.25)
end
```

helperSeaSurfaceCDF

```matlab
function helperSeaSurfaceCDF(x)
% Calculate and plot CDF

% Form CDF
x = x(~isnan(x));
n = length(x);
x = sort(x(:));
yy = (1:n)' / n;
notdup = ([diff(x(:)); 1] > 0);
xx = x(notdup);
yy = [0; yy(notdup)];

% Create vectors for plotting
k = length(xx);
n = reshape(repmat(1:k, 2, 1),2*k,1);
xCDF = [-Inf; xx(n); Inf];
yCDF = [0; 0; yy(1+n)];

% Plot
figure
plot(xCDF,yCDF,'LineWidth',2)
grid on
title('Wave Elevation CDF')
xlabel('Wave Elevation (m)')
ylabel('Probability')
drawnow
pause(0.25)
end
```

helperEstimateSignificantWaveHeight

```matlab
function sigHgt = helperEstimateSignificantWaveHeight(x,y,z)
% Plot an example showing how the wave heights are estimated and estimate
% the significant wave height

% Plot wave height estimation
figure
numX = numel(x);
z = reshape(z,numX,numel(y));
zEst = z(numX/2 + 1,:);
plot(x,zEst,'LineWidth',2)
grid on
hold on
xlabel('X (m)')
ylabel('Elevation (m)')
title('Wave Height Estimation')
axis tight
idxMin = islocalmin(z(numel(x)/2 + 1,:));
idxMax = islocalmax(z(numel(x)/2 + 1,:));
co = colororder;
plot(x(idxMin),zEst(idxMin),'v', ...
    'MarkerFaceColor',co(2,:),'MarkerEdgeColor',co(2,:))
plot(x(idxMax),zEst(idxMax),'^', ...
    'MarkerFaceColor',co(3,:),'MarkerEdgeColor',co(3,:))
legend('Wave Elevation Data','Trough','Crest','Location','SouthWest')
```

```matlab
% Estimate significant wave height
waveHgts = [];
for ii = 1:numX
    zEst = z(ii,:);
    idxMin = islocalmin(zEst);
    troughs = zEst(idxMin);
    numTroughs = sum(double(idxMin));
    idxMax = islocalmax(zEst);
    crests = zEst(idxMax);
    numCrests = sum(double(idxMax));
    numWaves = min(numTroughs,numCrests);
    waveHgts = [waveHgts ...
        abs(crests(1:numWaves) - troughs(1:numWaves))]; %#ok<AGROW>
end
waveHgts = sort(waveHgts);
idxTopThird = floor(numel(waveHgts)*2/3);
sigHgt = mean(waveHgts(idxTopThird:end));
drawnow
pause(0.25)
end
```

helperSeaSurfaceMotionPlot

```matlab
function helperSeaSurfaceMotionPlot(x,y,seaSurf,plotTime) %#ok<DEFNU>
% Color map for sea elevation plot
seaColorMap = helperSeaColorMap(256);

% Get initial height
[xGrid,yGrid] = meshgrid(x,y);
z = height(seaSurf,[xGrid(:).'; yGrid(:).'],plotTime(1));

% Plot
hFig = figure;
hS = surf(x,y,reshape(z,numel(x),numel(y)));
hold on
shading interp
axis equal
xlabel('X (m)')
ylabel('Y (m)')
zlabel('Elevations (m)')
stdZ = std(z(:));
minZ = -4*stdZ;
maxZ = 4*stdZ;
title('Sea Surface Motion Plot')
xlim([-50 50])
ylim([-50 50])
zlim([minZ maxZ])
hC = colorbar('southoutside');
colormap(seaColorMap)
hC.Label.String = 'Elevations (m)';
hC.Limits = [minZ maxZ];
view([-45 12])

% Change figure size
ppos0 = get(hFig,'Position');
figWidth = 700;
```

```
figHeight = 300;
set(gcf,'position',[ppos0(1),ppos0(2),figWidth,figHeight])

numTimes = numel(plotTime);
for ii = 2:numTimes
    % Update plot
    z = height(seaSurf,[xGrid(:).'; yGrid(:).'],plotTime(ii));
    hS.ZData = reshape(z,numel(x),numel(y));
    drawnow
    pause(0.1)
end
pause(0.25)
end
```

helperPlotRawIQ

```
function hRaw = helperPlotRawIQ(iqsig)
% Plot raw IQ magnitude

figure()
hRaw = pcolor(mag2db(abs(iqsig)));
hRaw.EdgeColor = 'none';
title('Raw IQ')
xlabel('Pulses')
ylabel('Samples')
hC = colorbar;
hC.Label.String = 'Magnitude (dB)';
drawnow
pause(0.25)
end
```

helperUpdatePlotRawIQ

```
function helperUpdatePlotRawIQ(hRaw,iqsig)
% Update the raw IQ magnitude plot

hRaw.CData = mag2db(abs(iqsig));
drawnow
pause(0.25)
end
```

helperRangeTimePlot

```
function helperRangeTimePlot(rngGrid,prf,pcsigMagdB)
% Range-Time Plot

figure()
numPulses = size(pcsigMagdB,2);
hImage = pcolor((1:numPulses)*1/prf,rngGrid,pcsigMagdB);
hImage.EdgeColor = 'none';
shading interp
xlabel('Time (sec)')
ylabel('Range (m)')
```

```matlab
hC = colorbar;
hC.Label.String = 'Magnitude (dB)';
axis tight
title('Range versus Time')
caxis([-20 0])
ylim([140 260])
drawnow
pause(0.25)
end
```

helperMagTimePlot

```matlab
function helperMagTimePlot(magVals,numPulses,prf,rngVal)
% Magnitude vs. Time Plot

figure()
plot((1:numPulses)*1/prf,magVals,'LineWidth',2)
grid on
xlabel('Time (sec)')
ylabel('Magnitude (dB)')
axis tight
title(sprintf('Magnitude versus Time at Range %.2f (m)',rngVal))
ylim([-20 0])
drawnow
pause(0.25)
end
```

helperSTFTPlot

```matlab
function helperSTFTPlot(S,F,T,lambda,rngVal)
% Time-Frequency Plot

figure()
S = mag2db(abs(S)); % Convert to dB
S = S - max(S(:)); % Normalize
Speed = dop2speed(F,lambda)/2; % Speed (m/s)
hSTFT = pcolor(T,Speed,S);
hold on
hSTFT.EdgeColor = 'none';
colorbar
xlabel('Time (sec)')
ylabel('Speed (m/s)')
title(sprintf('STFT at Range %.2f (m)',rngVal))
caxis([-20 0])
ylim([-1 1])
drawnow
pause(0.25)
end
```

helperHistPlot

```matlab
function helperHistPlot(pcsigMagdB,scale,shape)
% Amplitude Distribution
```

```matlab
% Histogram
figure
hHist = histogram(pcsigMagdB,'Normalization','pdf'); % Create histogram
grid on
hold on
xlabel('Normalized Magnitude (dB)')
ylabel('PDF')
title('Amplitude Histogram')

% Lognormal overlay
edges = hHist.BinEdges;
x = (edges(2:end) + edges(1:end-1))./2;
z = x./scale;
w = exp(-(z.^shape));
y = z.^(shape - 1).*w .*shape./scale;
plot(x,y,'-r')
end
```

Simulated Land Scenes for Synthetic Aperture Radar Image Formation

A synthetic aperture radar (SAR) system uses platform motion to mimic a longer aperture to improve cross-range resolution. SAR data is often collected using platforms such as aircraft, spacecraft, or vehicle. Due to the movement of the platform, the data returned from the reflections of targets appears unfocused, looking at times like random noise. In SAR systems, essential information resides in the amplitude and phase of the received data. Phase-dependent processing is critical to the formation of focused images.

In this example, you will simulate an L-band remote-sensing SAR system. The simulation includes generating IQ signals from a scenario containing three targets and a wooded-hills land surface. The returned data will be processed using the range migration focusing algorithm.

For more information on key SAR concepts, see the "Stripmap Synthetic Aperture Radar (SAR) Image Formation" on page 1-322 example.

Generate Simulated Terrain

Generate and plot a random height map for the land surface. The height map will be representative of a hilly scene with peaks up to about 200 m. It is important to ensure that the terrain resolution is less than the resolution of the imaging system. Based on the configuration of the helper function, the resolution of the map will be a little more than 1 meter in the X dimension and about 1.6 in the Y dimension. To increase the resolution of the generated map, increase the number of iterations. The roughness factor is often set to a value of 2. Smaller values result in rougher terrain, and larger values result in smoother terrain.

```
% Initialize random number generator
rng(2021)

% Create terrain
xLimits         = [900 1200]; % x-axis limits of terrain (m)
yLimits         = [-200 200]; % y-axis limits of terrain (m)
roughnessFactor = 1.75;       % Roughness factor
initialHgt      = 0;          % Initial height (m)
initialPerturb  = 200;        % Overall height of map (m)
numIter         = 8;          % Number of iterations
[x,y,A] = helperRandomTerrainGenerator(roughnessFactor,initialHgt, ....
    initialPerturb,xLimits(1),xLimits(2), ...
    yLimits(1),yLimits(2),numIter);
A(A < 0) = 0; % Fill-in areas below 0
xvec = x(1,:);
yvec = y(:,1);
resMapX = mean(diff(xvec))

resMapX = 1.1719

resMapY = mean(diff(yvec))

resMapY = 1.5625

% Plot simulated terrain
helperPlotSimulatedTerrain(xvec,yvec,A)
```

Simulated Terrain
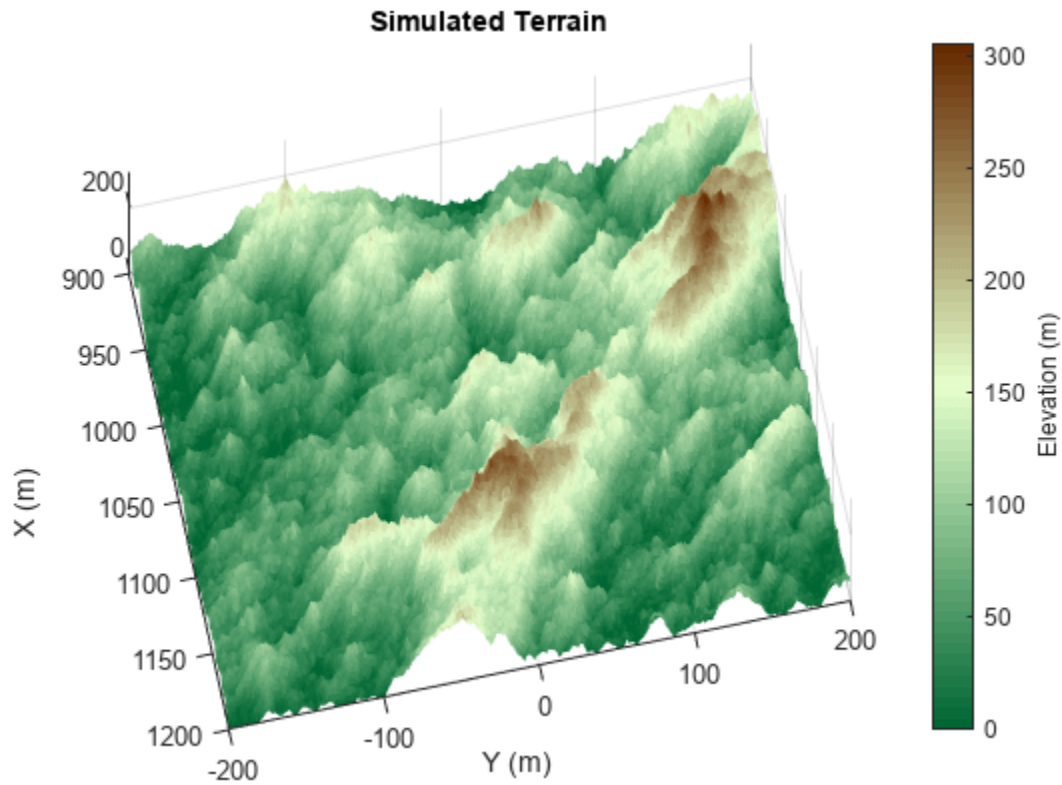
Specify the SAR System and Scenario

Define an L-band SAR imaging system with a range resolution approximately equal to 5 m. This system is mounted on an airborne platform flying at an altitude of 1000 meters. Verify the range resolution is as expected using the `bw2rangeres` function.

```
% Define key radar parameters
freq = 1e9;                          % Carrier frequency (Hz)
[lambda,c] = freq2wavelen(freq);     % Wavelength (m)
bw = 30e6;                           % Signal bandwidth (Hz)
fs = 60e6;                           % Sampling frequency (Hz)
tpd = 3e-6;                          % Pulse width (sec)

% Verify that the range resolution is as expected
bw2rangeres(bw)
```

```
ans = 4.9965
```

Specify the antenna aperture as 6 meters. Set the squint angle to 0 degrees for broadside operation.

```
% Antenna properties
apertureLength = 6;                  % Aperture length (m)
sqa = 0;                             % Squint angle (deg)
```

Set the platform velocity to 100 m/s and configure the platform geometry. The synthetic aperture length is 100 m as calculated by `sarlen`.

```
% Platform properties
v = 100;                             % Speed of the platform (m/s)
```

```
dur = 1;                                % Duration of flight (s)
rdrhgt = 1000;                          % Height of platform (m)
rdrpos1 = [0 0 rdrhgt];                 % Start position of the radar (m)
rdrvel = [0 v 0];                       % Radar plaform velocity
rdrpos2 = rdrvel*dur + rdrpos1;         % End position of the radar (m)
len = sarlen(v,dur)                     % Synthetic aperture length (m)
```

```
len = 100
```

Define the targets. The targets in this example are stationary and are intended to represent calibration targets. Set the target heights to 110 meters, relative to the surface. This height was selected such that targets may be occluded due to the hilly terrain.

```
% Configure the target platforms in x and y
targetpos = [1000,len/2,0;1020,len/2,0;1040,len/2,0]; % Target positions (m)
tgthgts = 110*ones(1,3); % Target height (m)
for it = 1:3
    % Set target height relative to terrain
    [~,idxX] = min(abs(targetpos(it,1) - xvec));
    [~,idxY] = min(abs(targetpos(it,2) - yvec));
    tgthgts(it) = tgthgts(it) + A(idxX,idxY);
    targetpos(it,3) = tgthgts(it);
end
```

Next, set the reference slant range, which is used in subsequent processing steps such as determining appropriate pointing angles for the radar antenna. Calculate the cross-range resolutions using the `sarazres` function. Based on the geometry and radar settings, the cross-range resolution is about 2 meters.

```
% Set the reference slant range for the cross-range processing
rc = sqrt((rdrhgt - mean(tgthgts))^2 + (mean(targetpos(:,1)))^2);
```

```
% Antenna orientation
depang = depressionang(rdrhgt,rc,'Flat','TargetHeight',mean(tgthgts)); % Depression angle (deg)
grazang = depang; % Grazing angle (deg)
```

```
% Azimuth resolution
azResolution = sarazres(rc,lambda,len)  % Cross-range resolution (m)
```

```
azResolution = 1.9572
```

Then, determine an appropriate pulse repetition frequency (PRF) for the SAR system. In a SAR system, the PRF has dual implications. The PRF not only determines the maximum unambiguous range but also serves as the sampling frequency in the cross-range direction. If the PRF is too low, the pulses are long in duration, resulting in fewer pulses illuminating a particular region. If the PRF is too high, the cross-range sampling is achieved but at the cost of reduced maximum range. The `sarprfbounds` function suggests maximum and minimum PRF bounds.

```
% Determine PRF bounds
[swlen,swwidth] = aperture2swath(rc,lambda,apertureLength,grazang);
[prfmin,prfmax] = sarprfbounds(v,azResolution,swlen,grazang)
```

```
prfmin = 51.0925
```

```
prfmax = 6.4742e+05
```

For this example, set the PRF to 500 Hz.

```
% Select a PRF within the PRF bounds
prf = 500; % Pulse repetition frequency (Hz)
```

Now that the parameters for the radar and targets are defined. Set up a radar scene using `radarScenario`. Add the radar platform and targets to the scene with `platform`. Set the target radar cross section (RCS) to 5 dBsm, and plot the scene.

```
% Create a radar scenario
scene = radarScenario('UpdateRate',prf,'IsEarthCentered',false,'StopTime',dur);

% Add platforms to the scene using the configurations previously defined
rdrplat = platform(scene,'Trajectory',kinematicTrajectory('Position',rdrpos1,'Velocity',[0 v 0])

% Add target platforms
rcs = rcsSignature('Pattern',5);
for it = 1:3
    platform(scene,'Position',targetpos(it,:),'Signatures',{rcs});
end

% Plot ground truth
helperPlotGroundTruth(xvec,yvec,A,rdrpos1,rdrpos2,targetpos)
```
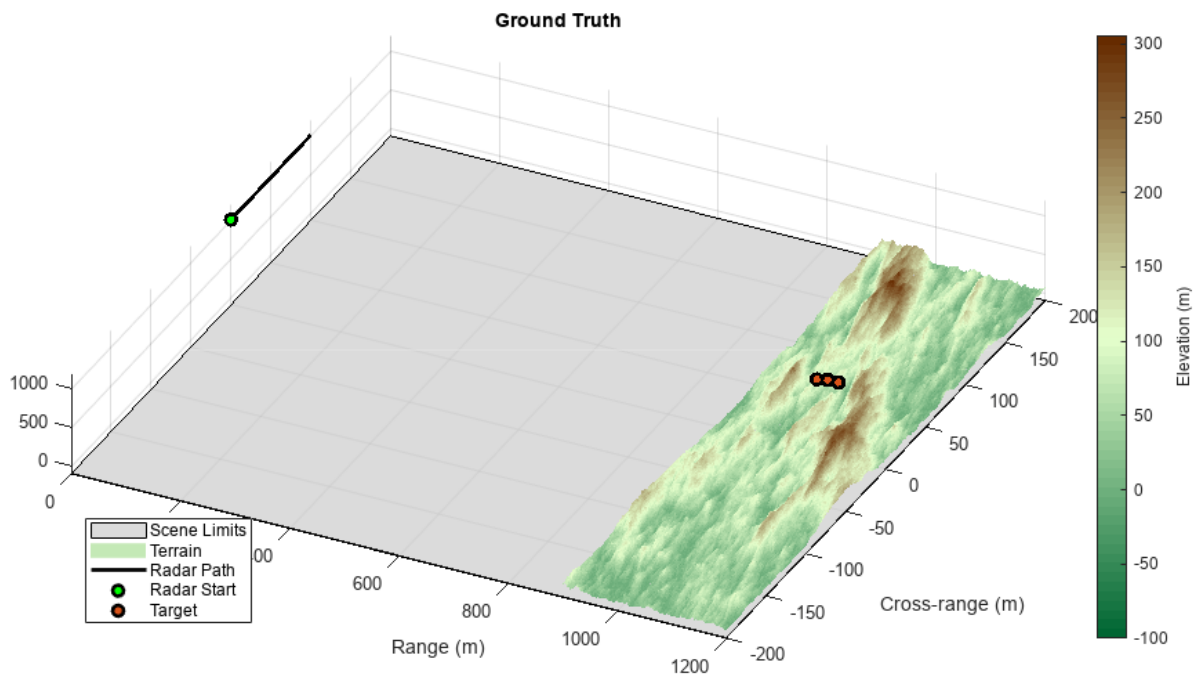

Note that the terrain generated has been limited in range and cross-range to the expected beam location. This is to conserve memory, as well as to speed up the simulation.

Define the Land Surface Reflectivity

Now that the radar scenario and its platforms are set, define the land surface. First, select a reflectivity model. Radar Toolbox™ offers 7 different reflectivity models covering a wide range of

frequencies, grazing angles, and land types. The asterisk denotes the default model. Type `help landreflectivity` or `doc landreflectivity` in the Command Window for more information on usage and applicable grazing angles for each model. The land reflectivity models are as follows.

- **APL:** Mathematical model supporting low to high grazing angles over frequencies in the range from 1 to 100 GHz. Land types supported are urban, high-relief, and low-relief.
- **Barton*:** Mathematical model supporting medium grazing angles over frequencies in the range from 1 to 10 GHz. Land types supported are rugged mountains, mountains, metropolitan, urban, wooded hills, rolling hills, woods, farm, desert, flatland, and smooth. This is the default model.
- **Billingsley:** Empirical model valid for low grazing angles over frequencies from 3 MHz to 12 GHz. Land types supported are low-relief rural, low-relief forest, farm, desert, marsh, grassland, high-relief rural, high-relief forest, mountains, urban, low-relief urban.
- **GIT:** Semi-empirical model for medium grazing angles over frequencies in the range from 3 to 15 GHz. Land types supported are soil, grass, tall grass, trees, and urban.
- **Morchin:** Mathematical model for high grazing angles over frequencies in the range from 300 MHz to 8 GHz. Land types supported are desert, farm, woods, and mountains.
- **Nathanson:** Empirical model for low to medium grazing angles over frequencies in the range from 1 to 36 GHz. Land types supported are desert, farm, woods, jungle, rolling hills, and urban.
- **Ulaby-Dobson:** Semi-empirical model for low to medium grazing angles over frequencies in the range from 1 to 18 GHz. Land types supported are soil, grass, shrubs, and short vegetation.

For this example, use the default Barton model, since it has such a large number of land types. For terrain values above 100 meters, assign reflectivity values for wooded hills. Otherwise, set reflectivity values to woods. Plot the reflectivity map to see the assignments

```matlab
% Specify custom reflectivity map
grazTable = 20:0.1:60;
freqTable = [1e9 10e9];
numSurfaces = 2;
reflectivityLayers = zeros(numel(grazTable),numel(freqTable),numSurfaces);
reflectivityLayers(:,:,1) = landreflectivity('Woods', ...
    grazTable,freqTable);
reflectivityLayers(:,:,2) = landreflectivity('WoodedHills', ...
    grazTable,freqTable);
reflectivityType = ones(size(A));
reflectivityType(A > 100) = 2;

% Plot custom reflectivity map
helperPlotReflectivityMap(xvec,yvec,A,reflectivityType,rdrpos1,rdrpos2,targetpos)
```
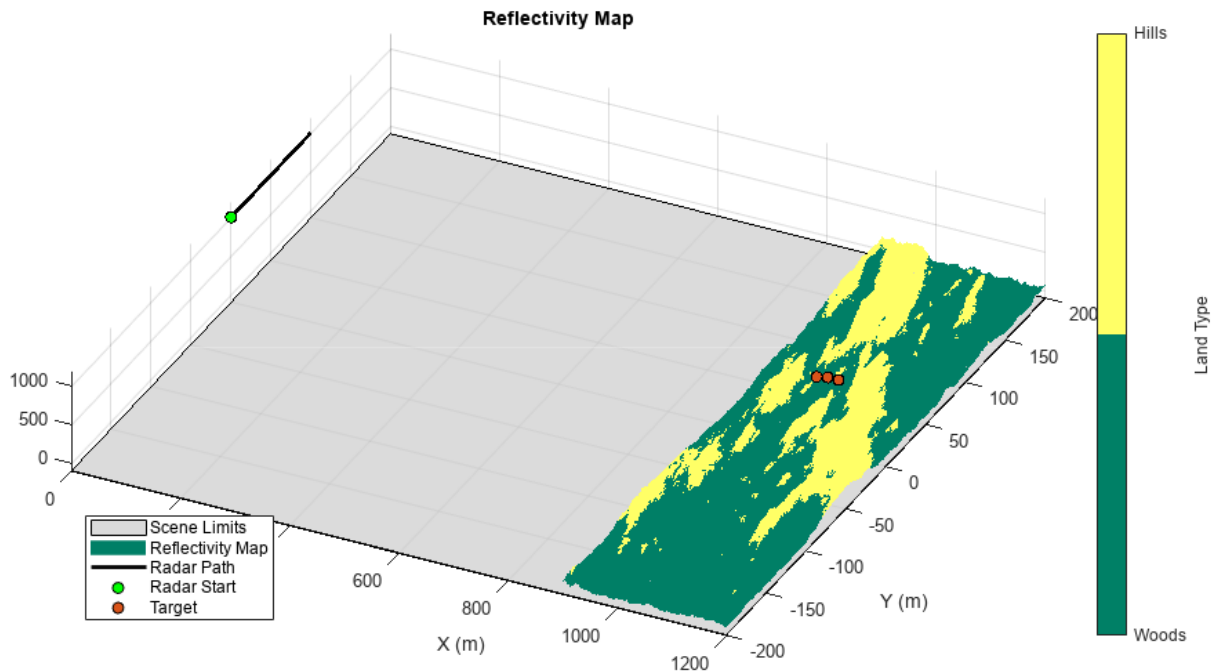
Reflectivity Map

Next, compose the custom reflectivity map for use by radarScenario using `surfaceReflectivity`. Set the `Speckle` property to `Rayleigh`. The `surfaceReflectivity` objects model speckle as an uncorrelated multiplicative factor as

$$I = \sigma \cdot n,$$

where σ is the surface reflectivity and n is an independent and identically distributed (IID) random value that is often defined as having unity mean noise. Common speckle models include Weibull, Rayleigh, and lognormal, which are all options provided by `surfaceReflectivity`. You can tailor the speckle with the distribution's corresponding properties. For a Rayleigh distribution, you can modify the `SpeckleScale` property.

```
reflectivityMap = surfaceReflectivity('Custom','Frequency',freqTable, ...
    'GrazingAngle',grazTable,'Reflectivity',reflectivityLayers, ...
    'Speckle','Rayleigh')

reflectivityMap =
  surfaceReflectivityCustom with properties:

    EnablePolarization: 0
          Reflectivity: [401x2x2 double]
             Frequency: [1.0000e+09 1.0000e+10]
          GrazingAngle: [20 20.1000 20.2000 20.3000 20.4000 20.5000 20.6000 20.7000 20.8000 20.90
                Speckle: 'Rayleigh'
           SpeckleScale: 0.7979
```

Add a land surface to the radar scenario using `landSurface`. Assign the random height map previously generated and the reflectivity map to the land surface.

```
% Add land surface to scene
s = landSurface(scene,'Terrain',A,'Boundary',[xLimits;yLimits], ...
    'RadarReflectivity',reflectivityMap, ...
    'ReflectivityMap',reflectivityType);
```

Configure the Radar Transceiver

In this section, configure the radar system properties. Define the antenna and the transmitted linear frequency modulated (LFM) waveform. Assign the radar sensor to the radar platform.

```
% Create a radar looking to the right
mountAngles = [0 depang 0];
rdr = radarTransceiver('MountingAngles',mountAngles,'NumRepetitions',1);

% Set peak power
rdr.Transmitter.PeakPower = 50e3;

% Set receiver sample rate and noise figure
rdr.Receiver.SampleRate = fs;
rdr.Receiver.NoiseFigure = 30;

% Define transmit and receive antenna and corresponding parameters
antbw = ap2beamwidth(apertureLength,lambda);
ant = phased.SincAntennaElement('FrequencyRange',[1e9 10e9],'Beamwidth',antbw);
rdr.TransmitAntenna.Sensor = ant;
rdr.TransmitAntenna.OperatingFrequency = freq;
rdr.ReceiveAntenna.Sensor = ant;
rdr.ReceiveAntenna.OperatingFrequency = freq;
antennaGain = aperture2gain(apertureLength^2,lambda);
rdr.Transmitter.Gain = antennaGain;
rdr.Receiver.Gain = antennaGain;

% Configure the LFM signal of the radar
rdr.Waveform = phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',tpd, ...
    'PRF',prf,'SweepBandwidth',bw);

% Add radar to radar platform
rdrplat.Sensors = rdr;
```

Generate the Datacube

Now that the scene and the radar system are defined, generate returns from the land surface with the `clutterGenerator` method. By default, `clutterGenerator` will simulate clutter returns in the mainlobe. For more information about clutter modeling, see "Introduction to Radar Scenario Clutter Simulation" on page 1-770.

```
% Collect clutter returns with the clutterGenerator
clutterGenerator(scene,rdr);

% Initialize output IQ datacube
minSample = 500; % Minimum sample range
maxRange = 2500; % Maximum range for IQ collection
truncRngSamp = ceil(range2time(maxRange)*fs); % Limit the number of samples
T = 1/prf; % Pulse repetition interval (sec)
numPulses = dur/T + 1; % Number of pulses
raw = zeros(numel(minSample:truncRngSamp),numPulses); % IQ datacube
```

Data is collected with the `receive` method. Either load in the data or simulate the raw SAR returns. If you choose to simulate the IQ, as the IQ data is received, a plot of the raw signal returns is generated. Otherwise, the unprocessed datacube is plotted at once. The raw signal is the collection of pulses transmitted in the cross-range direction. The plot shows the real part of the signal for the three targets and the land surface.

```
% Collect IQ
ii = 1;
hRaw = helperPlotRawIQ(raw,minSample);
simulateData = false; % Change to true to simulate IQ
if simulateData
    while advance(scene) %#ok<UNRCH>
        tmp = receive(scene); % nsamp x 1
        raw(:,ii) = tmp{1}(minSample:truncRngSamp);
        if mod(ii,100) == 0 % Update plot after 100 pulses
            helperUpdatePlotRawIQ(hRaw,raw);
        end
        ii = ii + 1;
    end
else
    load('rawSAR.mat');
end
helperUpdatePlotRawIQ(hRaw,raw);
```
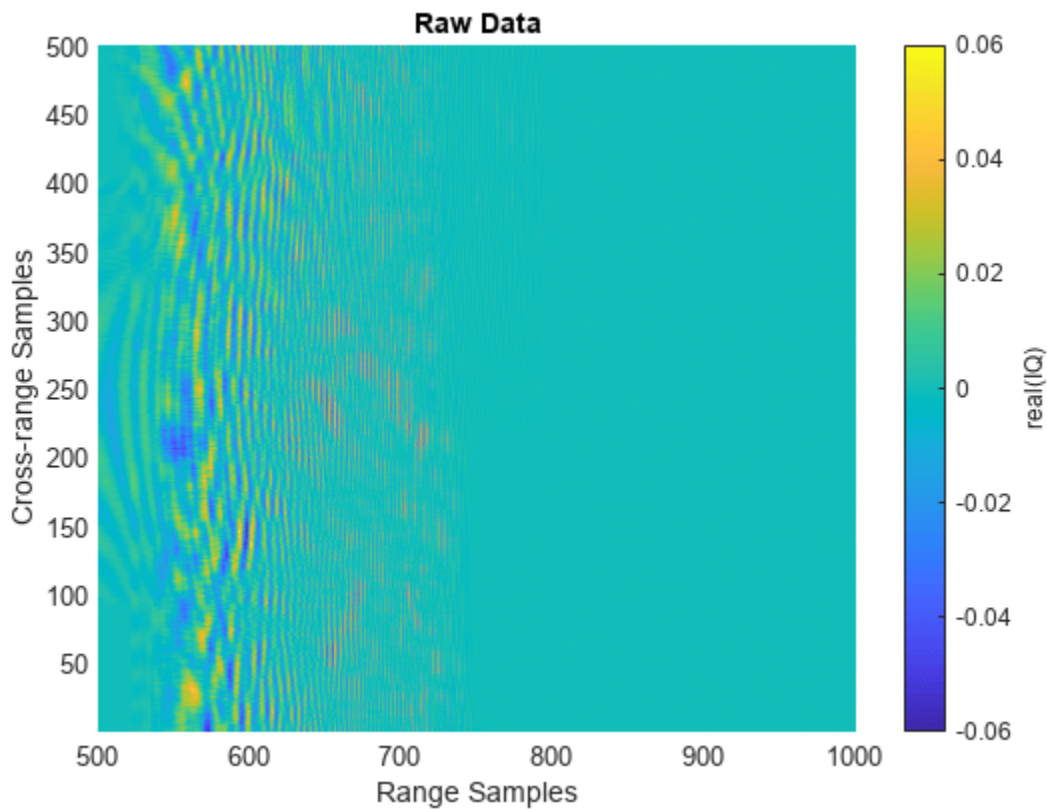
As is evident in the fully formed plot, the returns from the targets and the land surface are widely distributed in range and cross-range. Thus, it is difficult to distinguish the individual targets in the raw two-dimensional SAR data. The area where it is evident that returns are present is the mainlobe of the antenna. If you want to image a larger region, a couple of changes you can implement are:

- Increase the altitude of the SAR imaging platform or
- Increase the beamwidth.

Visualize SAR Data

Focus the image using the range migration algorithm. The range-migration algorithm corrects for the range-azimuth coupling, as well as the azimuth-frequency dependence. The algorithm proceeds as:

1. **FFT:** First, the algorithm performs a two-dimensional FFT. This transforms the SAR signal into wavenumber space.
2. **Matched Filtering:** Second, the algorithm focuses the image using a reference signal. This is a bulk focusing stage. The reference signal is computed for a selected range, often the mid-swath range. Targets at the selected range will be correctly focused, but targets away from the reference are only partially focused.
3. **Stolt Interpolation:** Next is a differential focusing stage that uses Stolt interpolation to focus the remainder of the targets.
4. **IFFT:** Finally, a two-dimensional IFFT is performed to return the data to the time domain.

Based on the radar waveform, use the `rangeMigrationLFM` function to form the single-look complex (SLC) image and plot the results. After range and cross-range processing, two targets can be distinguished from the background.

```
% Generating Single Look Complex image using range migration algorithm
slcimg = rangeMigrationLFM(raw,rdr.Waveform,freq,v,rc);
helperPlotSLC(slcimg,minSample,fs,v,prf,rdrpos1,targetpos, ...
    xvec,yvec,A)
```
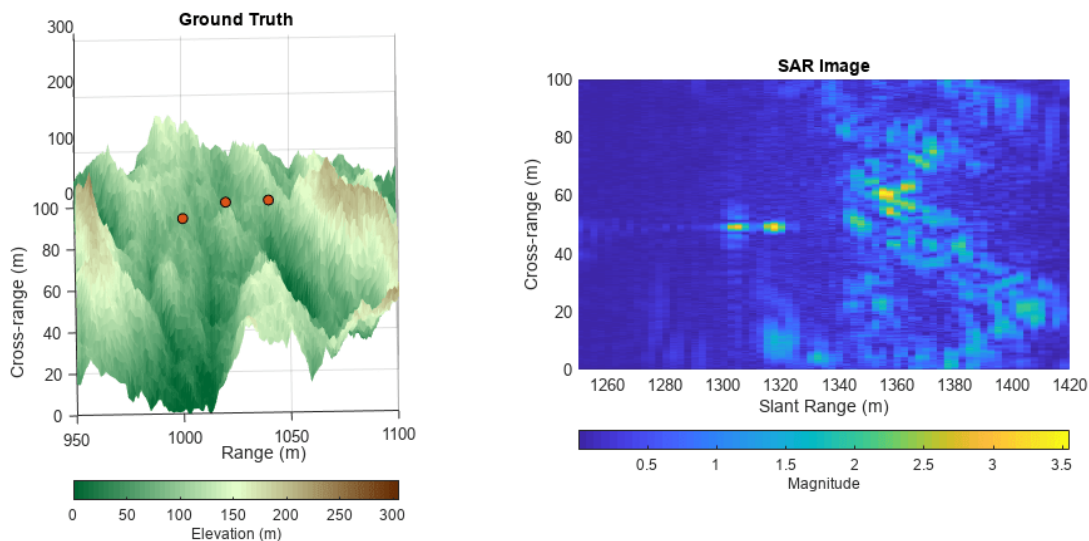

SAR images are similar to optical images, but the physics is quite different. Since SAR uses slant range to form images, higher elevation targets appear closer to the radar than lower elevation

targets, resulting in higher elevation targets appearing at nearer ranges in the SAR image. This distortion is called layover and is noticeable in the image. Additionally, the actual grazing angles change slightly over the imaged swath, with shallower angles existing at farther ranges and steeper angles at closer ranges. These characteristics among others result in warped images in comparison to the Cartesian ground truth.

The occlusion method helps us to further interpret the results. Using the `occlusion` method on the land surface, determine the visibility of the targets throughout the scenario.

```
for it = 1:3
    % Determine whether the target was occluded
    occ = false(1,numPulses);
    for ip = 1:numPulses
        rdrpos = rdrpos1 + rdrvel.*1/prf*(ip - 1);
        occ(ip) = s.occlusion(rdrpos,targetpos(it,:));
    end

    % Translate occlusion values to a visibility status
    helperGetVisibilityStatus(it,occ)
end
```

```
Target 1 is not visible during the scenario (visible 0% of the data collection).
Target 2 is partially visible during the scenario (visible 84% of the data collection).
Target 3 is fully visible during the scenario (visible 100% of the data collection).
```

The first target is not visible at all, because it is occluded by the terrain. The second target is only partially visible throughout the collection. This causes missing data in the cross-range dimension, which results in the increased sidelobes and decreased signal power. The third target is fully visible throughout the target collection. It is bright and focused.

Summary

This example demonstrated how to generate IQ from a stripmap-based SAR scenario with three targets over land terrain. This example showed you how to:

- Configure the SAR radar parameters,
- Define the radar scenario,
- Build a custom reflectivity map,
- Add terrain to the scene with added speckle,
- Generate IQ, and
- Form a focused SAR image.

This example can be easily modified for other platform geometries, radar parameters, and surface types.

Supporting Functions

```
function [x,y,terrain] = helperRandomTerrainGenerator(f,initialHeight, ...
    initialPerturb,minX,maxX,minY,maxY,numIter)
%randTerrainGenerator Generate random terrain
% [x,y,terrain] = helperRandomTerrainGenerator(f,initialHeight, ...
%    initialPerturb,minX,maxX,minY,maxY,seaLevel,numIter)
%
```

```matlab
% Inputs:
%   - f                  = A roughness parameter.  A factor of 2 is a
%                          typical default. Lower values result in a
%                          rougher terrain; higher values result in a
%                          smoother surface.
%   - initialHeight      = Sets the initial height of the lattice before
%                          the perturbations.
%   - initialPerturb     = Initial perturbation amount. This sets the
%                          overall height of the landscape.
%   - minX,maxX,minY,maxY = Initial points. Provides some degree of control
%                          over the macro appearance of the landscape.
%   - numIter            = Number of iterations that affects the density
%                          of the mesh that results from the iteration
%                          process.
%
% Output:
%   - x                  = X-dimension mesh grid
%   - y                  = Y-dimension mesh grid
%   - terrain            = Two-dimensional array in which each value
%                          represents the height of the terrain at that
%                          point/mesh-cell

% Generate random terrain
dX = (maxX-minX)/2;
dY = (maxY-minY)/2;
[x,y] = meshgrid(minX:dX:maxX,minY:dY:maxY);
terrain = ones(3,3)*initialHeight;
perturb = initialPerturb;
for ii = 2:numIter
    perturb = perturb/f;
    oldX = x;
    oldY = y;
    dX = (maxX-minX)/2^ii;
    dY = (maxY-minY)/2^ii;
    [x,y] = meshgrid(minX:dX:maxX,minY:dY:maxY);
    terrain = griddata(oldX,oldY,terrain,x,y);
    terrain = terrain + perturb*random('norm',0,1,1+2^ii,1+2^ii);
    terrain(terrain < 0) = 0;
end
end

function cmap = landColormap(n)
%landColormap Colormap for land surfaces
% cmap = landColormap(n)
%
% Inputs:
%   - n     = Number of samples in colormap
%
% Output:
%   - cmap  = n-by-3 colormap

c = hsv2rgb([5/12 1 0.4; 0.25 0.2 1; 5/72 1 0.4]);
cmap = zeros(n,3);
cmap(:,1) = interp1(1:3,c(:,1),linspace(1,3,n));
cmap(:,2) = interp1(1:3,c(:,2),linspace(1,3,n));
cmap(:,3) = interp1(1:3,c(:,3),linspace(1,3,n));
colormap(cmap);
end
```

```matlab
function helperPlotSimulatedTerrain(xvec,yvec,A)
% Plot simulated terrain

figure()
hS = surf(xvec,yvec,A);
hS.EdgeColor = 'none';
hC = colorbar;
hC.Label.String = 'Elevation (m)';
landmap = landColormap(64);
colormap(landmap);
xlabel('X (m)')
ylabel('Y (m)')
axis equal;
title('Simulated Terrain')
view([78 78])
drawnow
pause(0.25)
end

function helperPlotGroundTruth(xvec,yvec,A,rdrpos1,rdrpos2,targetpos)
% Plot ground truth

f = figure('Position',[505 443 997 535]);
movegui(f,'center');
% Plot boundary. Set plot boundary much lower than 0 for rendering reasons.
hLim = surf([0 1200],[-200 200].',-100*ones(2),'FaceColor',[0.8 0.8 0.8],'FaceAlpha',0.7);
hold on;
hS = surf(xvec,yvec,A);
hS.EdgeColor = 'none';
hC = colorbar;
hC.Label.String = 'Elevation (m)';
landmap = landColormap(64);
colormap(landmap);
hPlatPath = plot3([rdrpos1(1) rdrpos2(1)], ...
    [rdrpos1(2) rdrpos2(2)],[rdrpos1(3) rdrpos2(3)], ...
    '-k','LineWidth',2);
hPlatStart = plot3(rdrpos1(1),rdrpos1(2),rdrpos1(3), ...
    'o','LineWidth',2,'MarkerFaceColor','g','MarkerEdgeColor','k');
hTgt = plot3(targetpos(:,1),targetpos(:,2),targetpos(:,3), ...
    'o','LineWidth',2,'MarkerFaceColor',[0.8500 0.3250 0.0980], ...
    'MarkerEdgeColor','k');
view([26 75])
xlabel('Range (m)')
ylabel('Cross-range (m)')
title('Ground Truth')
axis tight;
zlim([-100 1200])
legend([hLim,hS,hPlatPath,hPlatStart,hTgt], ...
    {'Scene Limits','Terrain','Radar Path','Radar Start','Target'},'Location','SouthWest')
drawnow
pause(0.25)
end

function helperPlotReflectivityMap(xvec,yvec,A,reflectivityType,rdrpos1,rdrpos2,targetpos)
% Plot custom reflectivity map

f = figure('Position',[505 443 997 535]);
```

```
        movegui(f,'center');
        % Plot boundary. Set plot boundary much lower than 0 for rendering reasons.
        hLim = surf([0 1200],[-200 200].',-100*ones(2),'FaceColor',[0.8 0.8 0.8],'FaceAlpha',0.7);
        hold on
        hS = surf(xvec,yvec,A,reflectivityType);
        hS.EdgeColor = 'none';
        hold on;
        colormap(summer(2));
        hC = colorbar;
        clim([1 2]);
        hC.Ticks = [1 2];
        hC.TickLabels = {'Woods','Hills'};
        hC.Label.String = 'Land Type';
        hPlatPath = plot3([rdrpos1(1) rdrpos2(1)],[rdrpos1(2) rdrpos2(2)],[rdrpos1(3) rdrpos2(3)], ...
            '-k','LineWidth',2);
        hPlatStart = plot3(rdrpos1(1),rdrpos1(2),rdrpos1(3), ...
            'o','MarkerFaceColor','g','MarkerEdgeColor','k');
        hTgt = plot3(targetpos(:,1),targetpos(:,2),targetpos(:,3), ...
            'o','MarkerFaceColor',[0.8500 0.3250 0.0980],'MarkerEdgeColor','k');
        view([26 75])
        xlabel('X (m)')
        ylabel('Y (m)')
        title('Reflectivity Map')
        axis tight;
        zlim([-100 1200])
        legend([hLim,hS,hPlatPath,hPlatStart,hTgt], ...
            {'Scene Limits','Reflectivity Map','Radar Path','Radar Start','Target'},'Location','SouthWest
        drawnow
        pause(0.25)
end

function hRaw = helperPlotRawIQ(raw,minSample)
% Plot real of raw SAR IQ

        figure()
        [m,n] = size(raw);
        hRaw = pcolor(minSample:(m + minSample - 1),1:n,real(raw.'));
        hRaw.EdgeColor = 'none';
        title('Raw Data')
        xlabel('Range Samples')
        ylabel('Cross-range Samples')
        hC = colorbar;
        clim([-0.06 0.06])
        hC.Label.String = 'real(IQ)';
        drawnow
        pause(0.25)
end

function helperUpdatePlotRawIQ(hRaw,raw)
% Update the raw SAR IQ plot

        hRaw.CData = real(raw.');
        clim([-0.06 0.06]);
        drawnow
        pause(0.25)
end

function helperPlotSLC(slcimg,minSample,fs,v,prf,rdrpos1,targetpos, ...
```

```
    xvec,yvec,A)
% Plot magnitude of focused SAR image alongside reflectivity map

% Cross-range y-vector (m)
numPulses = size(slcimg,2);
du = v*1/prf; % Cross-range sample spacing (m)
dky = 2*pi/(numPulses*du); % ku domain sample spacing (rad/m)
dy = 2*pi/(numPulses*dky); % y-domain sample spacing (rad/m)
y = dy*(0:(numPulses - 1)) + rdrpos1(2); % Cross-range y-vector (m)

% Range vector (m)
c = physconst('LightSpeed');
numSamples = size(slcimg,1);
samples = minSample:(numSamples + minSample - 1);
sampleTime = samples*1/fs;
rngVec = time2range(sampleTime(1:end),c);

% Initialize figure
f = figure('Position',[264 250 1411 535]);
movegui(f,'center')
tiledlayout(1,2,'TileSpacing','Compact');

% Ground Truth
nexttile;
hS = surf(xvec,yvec,A);
hS.EdgeColor = 'none';
hold on;
plot3(targetpos(:,1),targetpos(:,2),targetpos(:,3), ...
    'o','MarkerFaceColor',[0.8500 0.3250 0.0980],'MarkerEdgeColor','k');
landmap = landColormap(64);
colormap(landmap);
hC = colorbar('southoutside');
hC.Label.String = 'Elevation (m)';
view([-1 75])
xlabel('Range (m)')
ylabel('Cross-range (m)')
title('Ground Truth')
axis equal
xlim([950 1100])
ylim([0 100])

% SAR Image
nexttile;
slcimg = abs(slcimg).';
hProc = pcolor(rngVec,y,slcimg);
hProc.EdgeColor = 'none';
colormap(hProc.Parent,parula)
hC = colorbar('southoutside');
hC.Label.String = 'Magnitude';
xlabel('Slant Range (m)')
ylabel('Cross-range (m)')
title('SAR Image')
axis equal
xlim([1250 1420])
ylim([0 100])

drawnow
pause(0.25)
```

```
end

function helperGetVisibilityStatus(tgtNum,occ)
% Translate occlusion values to a visibility status

visibility = {'not','partially','fully'};
if all(occ)
    idx = 1;
elseif any(occ)
    idx = 2;
else
    idx = 3;
end
visString = visibility{idx};
pctCollect = sum(double(~occ))./numel(occ)*100;
fprintf('Target %d is %s visible during the scenario (visible %.0f%% of the data collection).\n'
    tgtNum,visString,pctCollect)
drawnow
pause(0.25)
end
```

Simulate Radar Detections of Surface Targets in Clutter

This example shows how to simulate the detection of targets in surface clutter. You will first see how to use a statistical radar model to examine the effect of Doppler separation on detectability, then use terrain data to see how line-of-sight occlusions can be simulated in a radar scenario.

```
rng default
```

Detecting Surface Targets With Clutter Doppler Separation

This section demonstrates how Doppler separation allows moving targets to be detected even when surface reflections are significant. You will configure a radar scenario to simulate a typical side-looking airborne radar that is pointing at a surface target. You will first let the target be stationary, then calculate a velocity vector that separates the target from clutter, and examine the resulting detections.

Stationary Target

Start by creating a `radarDataGenerator`. Set `MountingAngles` so that the radar is pointing to the right of the platform (a -90 degree mounting yaw angle) and pointing down at 15 degrees (a 15 degree depression angle). Use a frequency of 1 GHz, 100 meter range resolution, a 5 kHz PRF, and 128 pulses. The beam is symmetric with a 4 degree two-sided beamwidth in azimuth and elevation.

```
mountingYPR = [-90 15 0];
fc = 1e9;
rangeRes = 100;
prf = 5e3;
numPulses = 128;
beamwidth = 4;
```

Use the PRF and number of pulses specification to calculate the nominal Doppler and range-rate resolution.

```
dopRes = prf/numPulses;
lambda = freq2wavelen(fc);
rangeRateRes = dop2speed(dopRes,lambda)/2;
```

Pulse-Doppler radars have both range and range-rate ambiguities. Calculate the unambiguous range and radial speed.

```
unambRange = time2range(1/prf);
unambRadialSpd = dop2speed(prf/4,lambda);
```

Construct the `radarDataGenerator`. Let the sensor index equal 1 and disable scanning functionality. The radar will update once per coherent processing interval (CPI). Configure the operating mode to be monostatic, with detections output in scenario coordinates. Enable elevation angle measurements, range-rate measurements, and the range and range-rate ambiguities. Disable false alarms and detection noise so that the effect of Doppler separation can be seen clearly. Lastly, set the reference target used to compute the loop gain to a 0 dBsm target at 20 km range, with a detection probability of 90%.

```
cpiTime = numPulses/prf;
rdr = radarDataGenerator(1,'No scanning','UpdateRate',1/cpiTime,'MountingAngles',mountingYPR,...
```

```
    'DetectionMode','Monostatic','TargetReportFormat','Detections','DetectionCoordinates','Scena
    'HasINS',true,'HasElevation',true,'HasFalseAlarms',false,'HasNoise',false,'HasRangeRate',true
    'HasRangeAmbiguities',true,'HasRangeRateAmbiguities',true,'MaxUnambiguousRadialSpeed',unambRa
    'MaxUnambiguousRange',unambRange,'CenterFrequency',fc,'FieldOfView',beamwidth,...
    'AzimuthResolution',beamwidth,'ElevationResolution',beamwidth,'RangeResolution',rangeRes,'Ran
    'ReferenceRange',20e3,'ReferenceRCS',0,'DetectionProbability',0.9);
```

Create the scenario, setting the update rate to 0 to let the update intervals be derived from sensors in the scene.

```
scene = radarScenario('UpdateRate',0,'IsEarthCentered',false);
```

The `landSurface` scenario method is used to define properties of regions of the scenario surface. You will use it here to specify an unbounded homogeneous surface with an associated reflectivity model. Use a constant-gamma reflectivity with a gamma value appropriate for flatland, which can be found with the `surfacegamma` function. `surfaceReflectivityLand` is used to create the reflectivity model that will be associated with the scenario surface through the `RadarReflectivity` property.

```
gammaDB = surfacegamma('Flatland');
refl = surfaceReflectivityLand('Model','ConstantGamma','Gamma',gammaDB);
landSurface(scene,'RadarReflectivity',refl);
```

Now define the radar platform. The platform will start 4 km above the origin, and fly in the +Y direction at 200 m/s. Set the platform orientation such that the platform-frame X is aligned with the direction of motion. With the mounting angles specified on the radar, this means the radar will be pointed in the +X direction.

```
altitude = 4e3;
rdrpos = [0 0 altitude];
rdrspd = 200;
rdrtraj = kinematicTrajectory('Position',rdrpos,'Velocity',[0 rdrspd 0],'Orientation',rotz(90).'
rdrplat = platform(scene,'Sensors',rdr,'Trajectory',rdrtraj);
```

The figure below shows a visualization of this scenario geometry, including the homogeneous land surface.
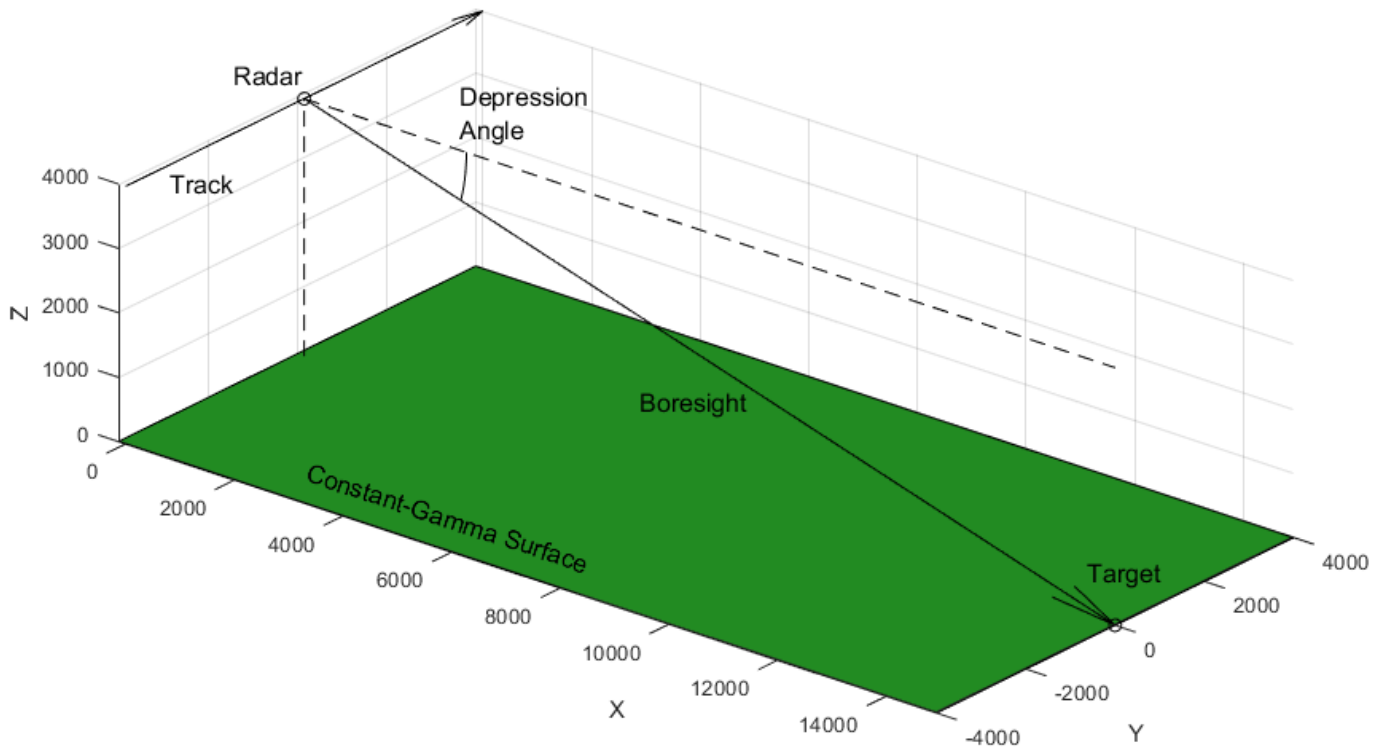
Place the target on the surface along the radar boresight direction. The required ground range can be found from the radar altitude and depression angle.

```
gndrng = altitude/tand(mountingYPR(2));
tgtpos = [gndrng 0 0];
```

Choose the target RCS such that the return signal would be buried in clutter without any Doppler separation. Do this by calculating an approximate RCS of clutter in a broadside range-Doppler resolution cell, and then setting the target RCS to one quarter of that. The reflectivity model defined previously can be used to get the NRCS, and the approximate area of the resolution cell can be found with the provided helper function. Express the RCS in dBsm.

```
clutterCellNRCS = refl(mountingYPR(2),fc);
clutterCellArea = helperBroadsideResCellArea(rangeRes,rangeRateRes,gndrng,mountingYPR(2),rdrspd)
clutterCellRCS = clutterCellNRCS*clutterCellArea;
tgtrcs = clutterCellRCS/4;
tgtrcs = 10*log10(tgtrcs);
```

Calculate the expected SNR of the target for later comparison to the SNR of the resulting detections. The `RadarLoopGain` property of the radar gives the relationship between target RCS, range, and SNR.

```
tgtrng = sqrt(altitude^2 + gndrng^2);
tgtsnr =  tgtrcs - 40*log10(tgtrng) + rdr.RadarLoopGain
```

```
tgtsnr = 32.3399
```

Use the scenario `platform` method to create a stationary target platform and add it to the scene. The `rcsSignature` class can be used to define a constant-RCS signature by using a scalar RCS value for the `Pattern` property.

```
tgtplat = platform(scene,'Position',tgtpos,'Signatures',rcsSignature('Pattern',tgtrcs));
```

Now, use the scenario `clutterGenerator` method to construct a clutter generator and enable clutter generation for your radar. The `Resolution` property defines the nominal spacing of clutter patches. Set this to be 1/5th of the range resolution to get multiple clutter patches per range gate. Set the range limit to 20 km to encompass the entire beam. `UseBeam` indicates if clutter generation should be performed automatically for the mainlobe of the antenna pattern, or for the field of view of a `radarDataGenerator`, as defined by the `FieldOfView` property.

```
clut = clutterGenerator(scene,rdr,'Resolution',rangeRes/5,'RangeLimit',20e3,'UseBeam',true);
```

Now the simulation is fully configured. Run the simulation for a single frame, collecting detections from the radar you constructed. This can be done by calling the scenario `detect` method.

```
dets = detect(scene);
```

Examine one of these detections.

```
dets{1}
```

```
ans =
  objectDetection with properties:

                   Time: 0
            Measurement: [6×1 double]
       MeasurementNoise: [6×6 double]
            SensorIndex: 1
          ObjectClassID: 0
    ObjectClassParameters: []
    MeasurementParameters: [1×1 struct]
        ObjectAttributes: {[1×1 struct]}
```

Each `objectDetection` contains information about a detection made in a particular resolution cell. The `Measurement` property will give the position and velocity of the detection in scenario coordinates, and `SensorIndex` refers to the index assigned to the `radarDataGenerator` upon construction. Useful metadata can be found by inspecting the contents of the `ObjectAttributes` property.

```
dets{1}.ObjectAttributes{1}
```

```
ans = struct with fields:
    TargetIndex: 14
            SNR: 38.8084
```

The `TargetIndex` field gives the index of the target that generated the detection. If multiple targets fall inside the same resolution cell, this will be the index of the target in that cell with the greatest signal power. This index corresponds to the `PlatformID` property of the target platforms. This ID is assigned automatically upon creating a platform, and increments by 1 with each new platform. When clutter is being simulated, the target index may exceed the number of platforms added to the scenario, which indicates that the detection came from a resolution cell where clutter return was dominant.

The low SNR of the first detection shown above is due to the sorted output from the `radarDataGenerator`. Within the field of view of the radar, clutter power in cells on the low side of the range swath are smaller than those at the center, near the intersection of the radar boresight with the ground plane, where the target platform is located. Plot the SNR across all clutter detections to verify the clutter power statistics are as expected.

```
helperPlotDetectionSNR(dets)
title('Clutter Detection SNR')
```
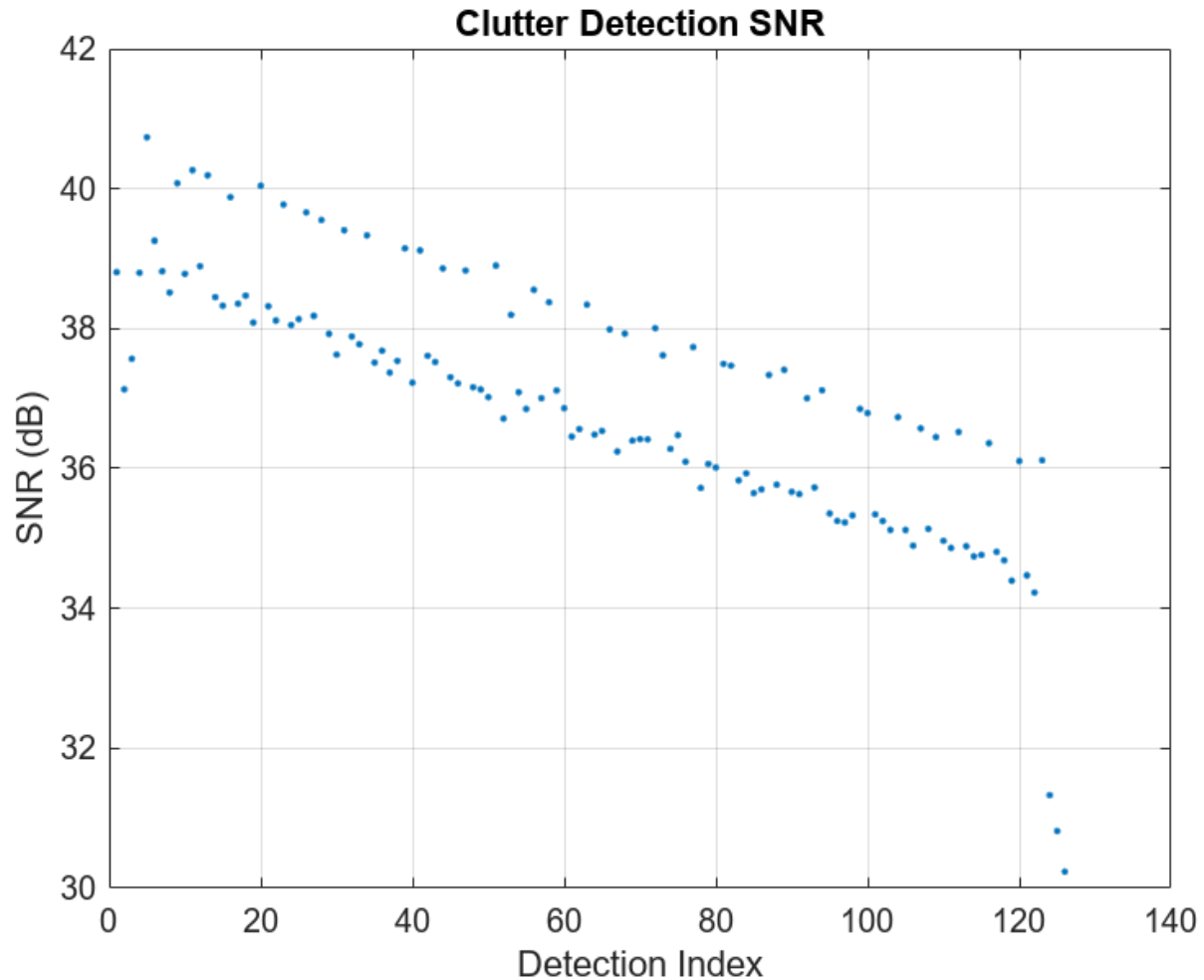

The distinct lines correspond to different Doppler bins. The mean SNR of detections in the 0 Hz Doppler bin (corresponding to the higher trend line) is about 6 dB higher than the target SNR, as expected.

Since the target platform of interest was the second platform added to the scene, it will have an ID of 2.

```
tgtplat.PlatformID
```

ans = 2

This can be used to find the object detection corresponding to the target platform, if the target was detected. Get the `TargetIndex` across all detections and see if any correspond to the target platform ID.

```
tgtidx = cellfun(@(t) t.ObjectAttributes{1}.TargetIndex,dets);
tgtdet = dets(tgtidx == tgtplat.PlatformID);
detectedPlatform = ~isempty(tgtdet)
```

```
detectedPlatform = logical

   0
```

In this scenario, the target platform was not detected because the target return signal fell in the same resolution cell as clutter return. Use the provided helper function to plot a visualization of the target and clutter patches in range-Doppler space with an overlay of the boundaries of the Doppler resolution cells.

```
helperPlotRangeDoppler(clut,rdrplat,tgtplat,lambda,prf,dopRes)
```
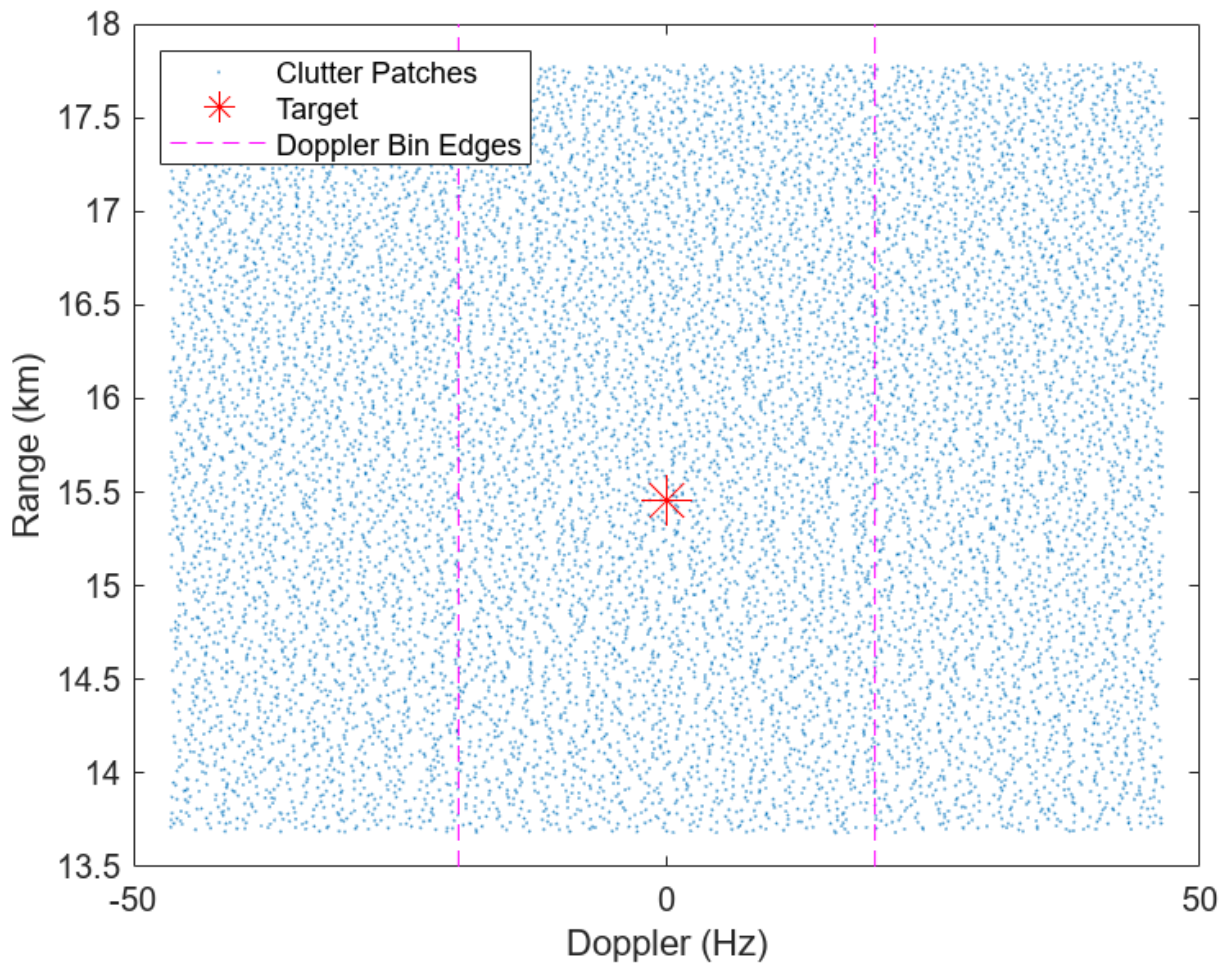

Since the target was placed on the surface along the antenna boresight and is stationary, it appears at the same point in range-Doppler space as the centroid of the mainlobe clutter return, and is undetectable.

Moving Target

Now you will calculate a velocity vector for the target so it moves along the surface towards the radar (in the -X direction) at a speed such that the return signal will fall outside of the clutter return in Doppler.

First calculate the total range-rate spread of the mainlobe clutter return. This equation is valid for broadside clutter, where the antenna boresight vector is perpendicular to the radar velocity.

```
clutSpread = 2*rdrspd*sind(beamwidth/2);
```

Calculate the required closing rate such that the target is one Doppler resolution cell outside of the mainlobe clutter return. Also calculate the total required speed, which can be found by dividing by the magnitude of the X component of the line of sight to the radar position.

```
los = (rdrpos - tgtpos)/tgtrng;
tgtClosingRate = clutSpread/2 + rangeRateRes;
tgtspd = tgtClosingRate/abs(los(1));
```

Set the target velocity by accessing the platform trajectory, which is a `kinematicTrajectory` by default.

```
tgtplat.Trajectory.Velocity = [-tgtspd 0 0];
```

Run the simulation for another frame.

```
dets = detect(scene);
```

Once again, get the target index for each detection and search for one that corresponds to the target platform.

```
tgtidx = cellfun(@(t) t.ObjectAttributes{1}.TargetIndex,dets);
tgtdet = dets(tgtidx == tgtplat.PlatformID);
detectedPlatform = ~isempty(tgtdet)
```

```
detectedPlatform = logical

    0
```

This time, the target was detected. Inspect the measured position and object attributes of the target detection.

```
if detectedPlatform
    tgtdet{1}.Measurement(1:3)
    tgtdet{1}.ObjectAttributes{:}
end
```

The measured target position is accurate to within a few meters, and the reported SNR is as found earlier via the radar loop gain.

View the returns in range-Doppler space.

```
helperPlotRangeDoppler(clut,rdrplat,tgtplat,lambda,prf,dopRes)
```
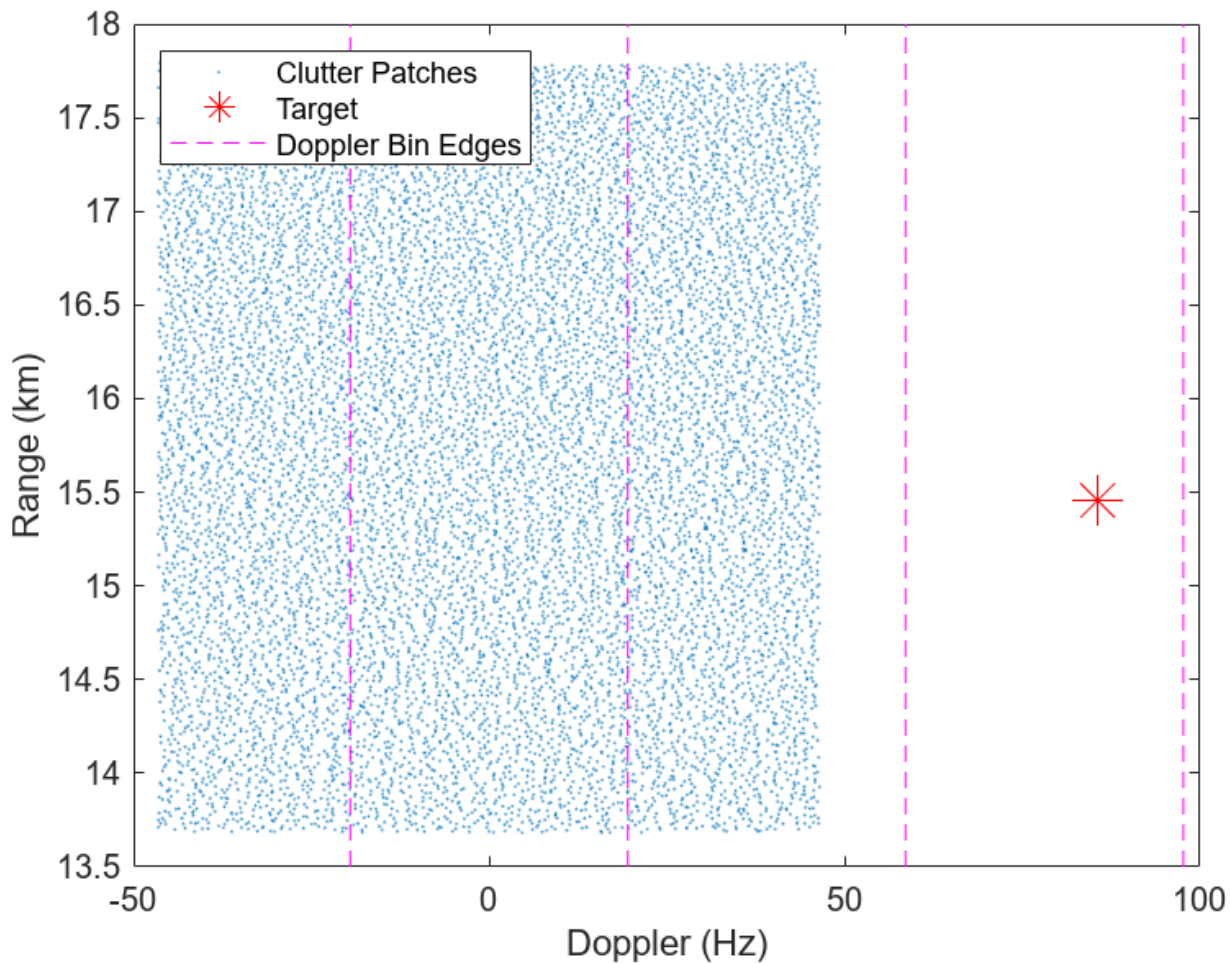
The target return has a Doppler frequency that is one Doppler resolution cell greater than the maximum Doppler of the mainlobe clutter return. Despite the small RCS of the target relative to clutter, it was detectable thanks to Doppler separation.

Detecting Surface Targets in the Presence of Terrain

Terrain Clutter and Occlusions

In this section, you will again construct a radar scenario and examine the detectability of a target in clutter. This time you will use terrain data to place the target on the surface, generate clutter returns, and simulate line-of-sight occlusions between the radar and target.

Start by loading the sample terrain data. This contains the variables `terrain` and `boundary`, which will be passed to the `landSurface` method.

```
load MountainousTerrain.mat
```

Create the scenario as before. Use the `landSurface` method to define a rectangular region of terrain by specifying a matrix of height samples and a boundary for the region in two-point form.

```
scene = radarScenario('UpdateRate',0,'IsEarthCentered',false);
srf = landSurface(scene,'RadarReflectivity',refl,'Terrain',terrain,'Boundary',boundary);
```

The terrain data covers a region of about 23 km in the X direction and 17 km in the Y direction. This can be seen by inspecting the `Boundary` property.

```
srf.Boundary
```

```
ans = 2×2
10⁴ ×

        0    2.2920
  -0.8544    0.8544
```

The radar will again be placed above the origin while viewing a target platform that is moving along the surface towards the radar from the +X direction. Re-create the radar platform, this time using the `height` method of the surface object to place the radar above the surface at the same altitude used previously.

```
rdrpos(3) = height(srf,rdrpos) + altitude;
rdrtraj = kinematicTrajectory('Position',rdrpos,'Velocity',[0 rdrspd 0],'Orientation',rotz(90).'
rdrplat = platform(scene,'Sensors',rdr,'Trajectory',rdrtraj);
```

Re-create the target platform, using the `height` method to place the target platform on the surface.

```
tgtpos(3) = height(srf,tgtpos);
tgttraj = kinematicTrajectory('Position',tgtpos,'Velocity',[-tgtspd 0 0]);
tgtplat = platform(scene,'Trajectory',tgttraj,'Signatures',rcsSignature('Pattern',tgtrcs));
```

Enable clutter generation. This time, since terrain data is being used, the `Resolution` property is not relevant. Clutter patches will correspond to facets of the terrain data.

```
clut = clutterGenerator(scene,rdr,'RangeLimit',20e3,'UseBeam',true);
```

The actual spacing of clutter patches in the X and Y directions can be found by dividing the extent of the surface in each direction by the number of height samples. Note that the X and Y directions are defined for the matrix of height sampled following the meshgrid convention, where the first dimension of the matrix (columns) corresponds to Y and the second dimension (rows) to X.

```
terrainClutterSpacing = diff(boundary,1,2)./flip(size(terrain)).'
```

```
terrainClutterSpacing = 2×1

   29.9608
   25.6188
```

The terrain data is sampled at 25 to 30 meter resolution, which is sufficient to yield a few clutter patches per resolution cell.

Collect one frame of detections.

```
dets = detect(scene);
```

Since the target is moving with the same velocity vector as before, a detection of the target platform might be expected. Plot the clutter patches and target in range-Doppler space again to verify.

```
helperPlotRangeDoppler(clut,rdrplat,tgtplat,lambda,prf,dopRes)
```

1-835
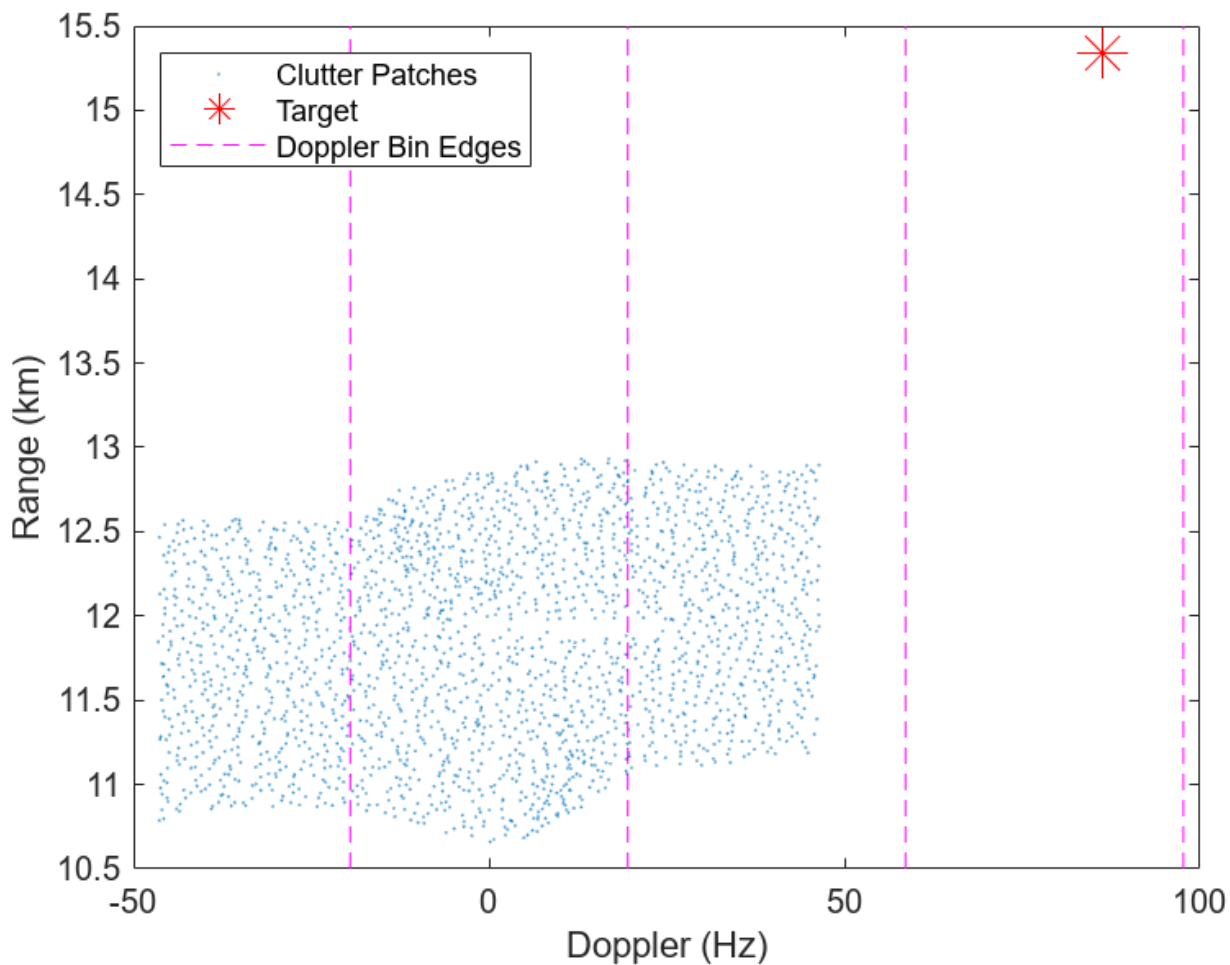
Notice that this time, the target is separated from clutter in both range and Doppler. Notice also that the clutter return is closer in range than in the flat-surface case by about 3 km. The overall extent of clutter in Doppler is the same since that depends only on beamwidth, not the shape of the surface.

Check to see if an object detection corresponding to the target platform was generated.

```
tgtidx = cellfun(@(t) t.ObjectAttributes{1}.TargetIndex,dets);
tgtdet = dets(tgtidx == tgtplat.PlatformID);
detectedPlatform = ~isempty(tgtdet)
```

```
detectedPlatform = logical

   0
```

The target was not detected. Plot the scenario overview to see why, adjusting the view so the effect of terrain can be seen more easily.

```
helperPlotClutterScenario(scene)
set(gca,'view',[-115 22])
```
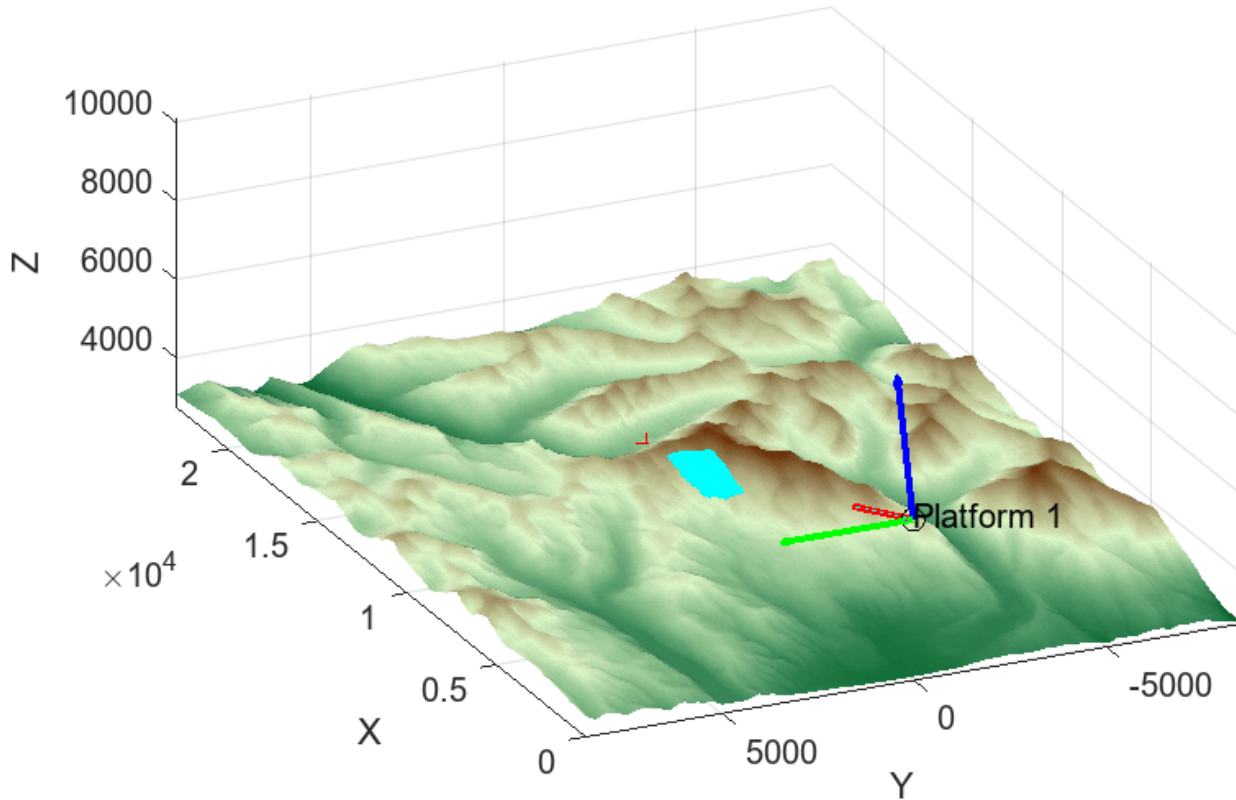
The same geometry parameters were used here as in the homogeneous surface scenario. The target still lies directly along the antenna boresight, but the mainlobe intersects a region of terrain that is at a lower range than the target. The target is thus occluded by the terrain and is not detectable. The occlusion of the LOS to the target can be checked explicitly by using the `occlusion` method on the surface object. This method will take two points as input and indicate whether or not the LOS between them is occluded by the surface. The platform positions can be passed in directly.

```
occluded = occlusion(srf,rdrplat.Position,tgtplat.Position)
```

occluded = *logical*

```
   1
```

This verifies the observation from the scenario overview plot that the target is occluded from the radar by the terrain.

Surface objects in the scenario are managed by a surface manager, which can be accessed through the `SurfaceManager` property of the scenario. Inspect this property.

```
scene.SurfaceManager
```

```
ans =
  SurfaceManager with properties:

    UseOcclusion: 1
        Surfaces: [1×1 radar.scenario.LandSurface]
```

The manager contains a read-only vector of surface objects present in the scenario, and a logical indicator that can be used to enable/disable line-of-sight occlusions by surface features. Try disabling this now, collecting detections for another frame, and checking for a target platform detection.

```
scene.SurfaceManager.UseOcclusion = false;
dets = detect(scene);
tgtidx = cellfun(@(t) t.ObjectAttributes{1}.TargetIndex,dets);
tgtdet = dets(tgtidx == tgtplat.PlatformID);
detectedPlatform = ~isempty(tgtdet)

detectedPlatform = logical

   1
```

With surface occlusion disabled, the platform is detectable.

Simulate Multiple Surface Targets

In this next section, you will place multiple platforms at random positions on the surface and simulate detections while observing terrain occlusions, included automatically as part of the scenario `detect` method. Clutter detections will not be simulated here in order to simplify the simulation and highlight the terrain occlusion effects (that is, assume all targets are detectable given that they are visible to the radar). To include interference from clutter, the `clutterGenerator` method could be used as before, though the targets would need to be in motion or have large RCS signatures to be detectable.

Re-create the scenario, using the same terrain data.

```
scene = radarScenario('UpdateRate',0,'IsEarthCentered',false);
srf = landSurface(scene,'RadarReflectivity',refl,'Terrain',terrain,'Boundary',boundary);
```

Release the radar object so that it may be reconfigured. Using the same field of view, enable electronic scanning over 90 degrees in azimuth and 16 degrees in elevation in order to cover most of the surface. All targets will be illuminated over the course of a full scan.

```
release(rdr)
rdr.ScanMode = 'Electronic';
rdr.ElectronicAzimuthLimits = [-45 45];
rdr.ElectronicElevationLimits = [-8 8];
```

Modify the trajectory created above to place the radar platform at one corner of the loaded terrain data. The same straight-line trajectory will be re-used. Set the scenario stop time as needed to give the radar time to travel from one end of the terrain to the other, in the Y direction. This will allow for the line of sight to some targets to transition between occluded and unoccluded.

```
rdrtraj.Position(2) = boundary(2,1);
rdrplat = platform(scene,'Sensors',rdr,'Trajectory',rdrtraj);
scene.StopTime = diff(srf.Boundary(2,:))/rdrspd;
```

Place down targets at random positions on the surface. The minimum value of the position in the X direction is chosen so that all targets will fall within the field of view of the radar at some point during the scan. The targets are uniformly distributed in the Y direction, and the `height` method is again used to place them on the surface.

```
numtgt = 14;
tgtpos(1,1:numtgt) = 1e4 + rand(1,numtgt)*(srf.Boundary(1,2) - 1e4);
tgtpos(2,:) = srf.Boundary(2,1) + rand(1,numtgt)*diff(srf.Boundary(2,:),1,2);
tgtpos(3,:) = height(srf,tgtpos);
for ind = 1:numtgt
    platform(scene,'Position',tgtpos(:,ind).');
end
```

Each iteration of the simulation loop constitutes one scan position. The second output from the scenario detect method, config, returns some information on the configuration of the radarDataGenerator that produced the detections. The IsScanDone field is a logical scalar that indicates if the radar was on its final scan position on the last call to detect. The detections from each scan position will be saved and when the scan pattern is completed, a visualization of the occlusion states and resulting detections will be plotted.

```
figure;
p = get(gcf,'Position');
set(gcf,'Position',[p(1:2) 2*p(3:4)])
fullScanDets = [];
while advance(scene)
    [dets,config] = detect(scene);
    fullScanDets = [fullScanDets; dets];
    if config.IsScanDone
        helperPlotOcclusions(scene,fullScanDets)
        drawnow
        fullScanDets = [];
    end
end
```
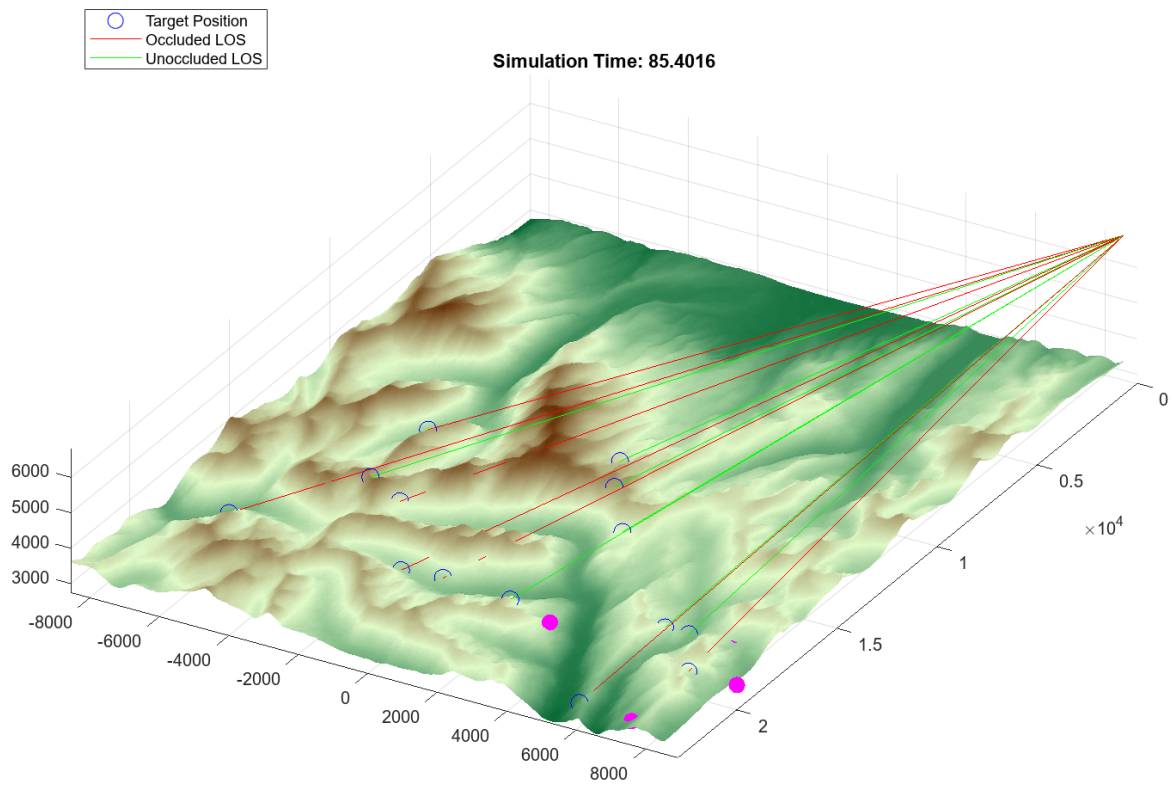
The below gif shows the resulting animation. Since all visible targets are illuminated at some point during a complete scan, detections are made whenever a target is visible, while targets with occluded lines of sight are not detected.
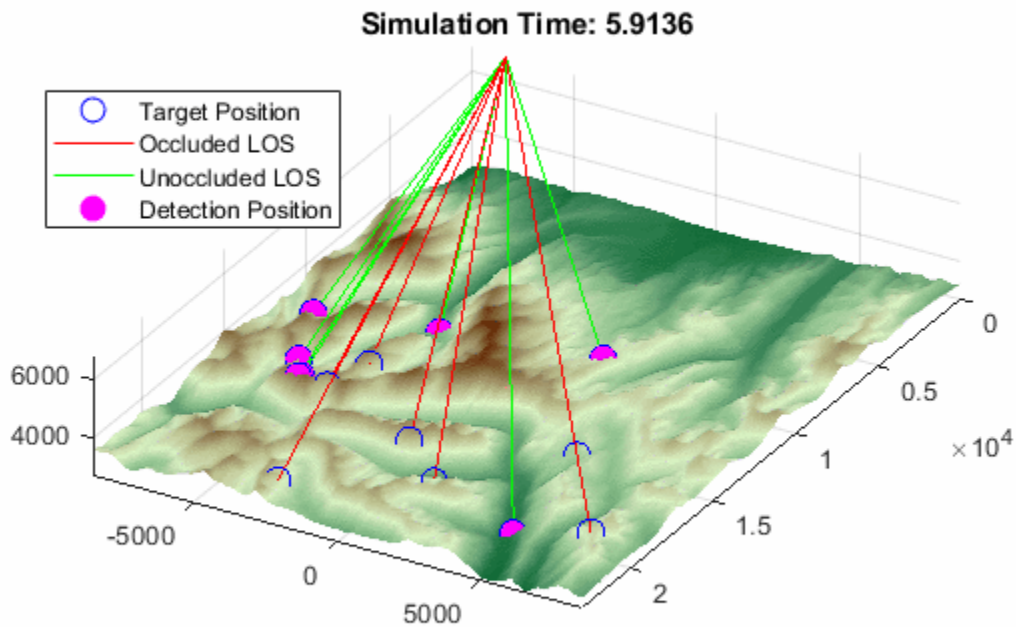
Conclusion

In this example, you saw how to configure a `radarScenario` and `radarDataGenerator` to simulate detections of a surface target in clutter. You used a `ClutterGenerator` with a simple homogeneous surface to examine the effect of Doppler separation, then used terrain data to simulate the effect of line-of-sight occlusions on detectability.

Supporting Functions

helperBroadsideResCellArea

```
function A = helperBroadsideResCellArea( rangeRes,rangeRateRes,groundRange,graze,speed )
% Calculate the approximate area of a broadside range-Doppler resolution
% cell

% Ground range resolution
groundRangeRes = rangeRes/cosd(graze);

% Broadside angular variation corresponding to Doppler resolution
dtheta = 2*asin(rangeRateRes/(2*speed));

% Multiply by ground range to get cross-range extent
crossRangeWidth = groundRange*dtheta;

% Cell area
A = crossRangeWidth*groundRangeRes;
```

```
      end
```

helperPlotDetectionSNR

```
function helperPlotDetectionSNR( dets )
% Plot the SNR of a set of objectDetections

detsnr = cellfun(@(t) t.ObjectAttributes{1}.SNR,dets);
plot(detsnr,'.')
grid on
xlabel('Detection Index')
ylabel('SNR (dB)')

end
```

helperPlotRangeDoppler

```
function helperPlotRangeDoppler( clut,rdrplat,tgtplat,lambda,prf,dopRes )
% Plot the range-Doppler truth of clutter patches and a target, along with
% an indication of the edges of the Doppler resolution cells.

% Get range and Doppler to target
los = rdrplat.Position - tgtplat.Position;
tgtRange = norm(los);
los = los/tgtRange;
tgtDoppler = 2/lambda*sum((tgtplat.Trajectory.Velocity-rdrplat.Trajectory.Velocity).*los);

% Get range and Doppler to clutter patches
patches = clut.LastPatchData.Centers;
los = patches - rdrplat.Position(:);
clutterRange = sqrt(sum(los.^2,1));
los = los ./ clutterRange;
clutterDoppler = 2/lambda*sum(rdrplat.Trajectory.Velocity(:).*los,1);

% Plot clutter and target in range-Doppler space
plot(clutterDoppler,clutterRange/1e3,'.','markersize',.1)
hold on
plot(tgtDoppler,tgtRange/1e3,'*r','markersize',15)

% Plot the Doppler bin edges
xl = xlim;
yl = ylim;
dopBins = -prf/2:dopRes:(prf/2-dopRes);
dopBinEdges = [dopBins(1)-dopRes/2, dopBins+dopRes/2];
dopBinEdges = dopBinEdges(dopBinEdges >= xl(1) & dopBinEdges <= xl(2));
line([1 1].'*dopBinEdges,repmat(yl,numel(dopBinEdges),1).','color','magenta','linestyle','--')
hold off

legend('Clutter Patches','Target','Doppler Bin Edges','Location','northwest')

xlabel('Doppler (Hz)')
ylabel('Range (km)')

end
```

helperPlotOcclusions

```
function helperPlotOcclusions( scene,dets )
% Plot the scenario surface along with target and radar positions and the
% lines of sight between them, color coding to indicate the occlusion state
% of each line of sight. Also plot the position of any detections.

% Plot the surface
helperPlotScenarioSurface(scene.SurfaceManager.Surfaces(1))
hold on

% Find radar platform and get current position
plats = [scene.Platforms{:}];
hasSensor = ~cellfun(@isempty,{plats.Sensors});
rdrplat = plats(find(hasSensor,1));
rdrpos = rdrplat.sensorPosition(1);

% Target platforms
tgtplats = plats(plats ~= rdrplat);

for ind = 1:numel(tgtplats) % for each target

    % Position of current target
    tgtpos = tgtplats(ind).Position;

    hndls(1) = plot3(tgtpos(1),tgtpos(2),tgtpos(3),'oblue','MarkerSize',10);

    % Get occlusion state by querying the scenario surface manager
    occ = occlusion(scene.SurfaceManager,rdrpos,tgtpos,scene.SimulationTime);

    % Plot color-coded line of sight
    if occ
        hndls(2) = line([rdrpos(1) tgtpos(1)],[rdrpos(2) tgtpos(2)],[rdrpos(3) tgtpos(3)],'color
    else
        hndls(3) = line([rdrpos(1) tgtpos(1)],[rdrpos(2) tgtpos(2)],[rdrpos(3) tgtpos(3)],'color
    end

end

% Keep only detections originating from target platforms
tgtidx = cellfun(@(t) t.ObjectAttributes{1}.TargetIndex,dets);
targetDetection = any(tgtidx == [plats.PlatformID],2);
dets = dets(targetDetection);

% Plot detections
if ~isempty(dets)
    detpos = cell2mat(cellfun(@(t) t.Measurement(1:3),dets.','UniformOutput',0));
    plot3(detpos(1,:),detpos(2,:),detpos(3,:),'omagenta','MarkerSize',9,'MarkerFaceColor','magen
end

hold off
legendstr = ["Target Position" "Occluded LOS" "Unoccluded LOS" "Detection Positions"];
I = ishghandle(hndls);
hndls = hndls(I);
legendstr = legendstr(I);
legend(hndls,legendstr,'Location','Northwest');
title(sprintf('Simulation Time: %.4f',scene.SimulationTime))
view([120 28])
```

```
end
```

Generate Clutter and Target Returns for MTI Radar

This example shows how to generate surface clutter and target returns for the simulation of a moving target indicator (MTI) radar in a `radarScenario`. MTI filtering is performed and the resulting range profiles and Doppler spectra are examined. The effects of platform geometry are demonstrated by looking at both broadside and squinted trajectories, and at target placement within the radar antenna mainlobe.

```
rng default
```

MTI Radar Basics

MTI is a clutter mitigation method for pulsed radar that involves a slow-time filter and produces a time series of range profiles where returns from stationary objects have been filtered out. This is a simpler alternative to pulse-Doppler processing which does not yield a velocity estimate for targets, though direction-of-arrival estimation can still be performed on target detections.

MTI Filter

An MTI filter is a notch filter that is configured to remove the signal component at the Doppler frequency of surface clutter. Derivative filters are commonly used for their simplicity, with first- and second-order differences being the most common. This example will use the three-pulse canceller, which is a derivative filter of order 2 and length 3, and uses central differences. Such a filter is simple to implement and, since the order is 1 less than the length, it has only one null per Doppler ambiguity. In general, a balance can be found for a particular application between the width of the passband, attenuation, and the number of null frequencies.

The figure below shows the filter response for pulse cancellers of length 2 through 5 and a PRF of 5 kHz. Note that they all attenuate signals close to 0 Hz, with varying degrees of passband width and rolloff. This nominal filter type is appropriate for broadside trajectories, where clutter return is centered about 0 Hz. If using a squinted trajectory, the filter can be offset in frequency to notch out the expected frequency of the clutter centroid.
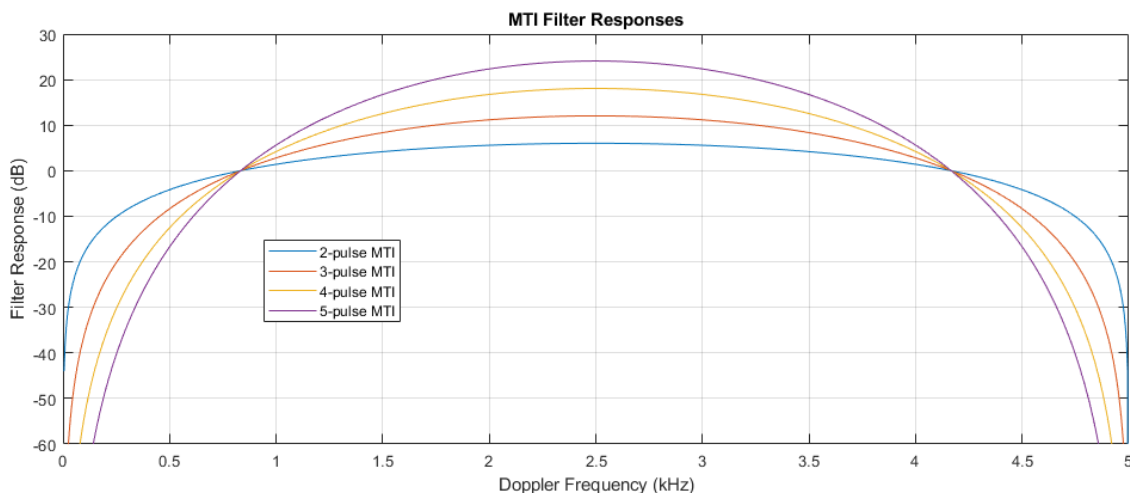
Note that spectral components of the received signals wrap at the PRF, so signals at any integer multiple of the PRF fall in the nulls of the filter. Radials speeds that impart a Doppler frequency close to these integer multiples are referred to as blind speeds because, even though the target may be in

motion, its Doppler frequency is congruent to that of the clutter return. Note also that the nominal filter response is symmetric about 0 Hz, positive and negative range rates of the same magnitude are filtered similarly. The figure below shows the length-3 filter response for all Doppler frequencies between -3 and +3 multiples of the 5 kHz PRF, with the Dopplers corresponding to blind speeds indicated by vertical lines.
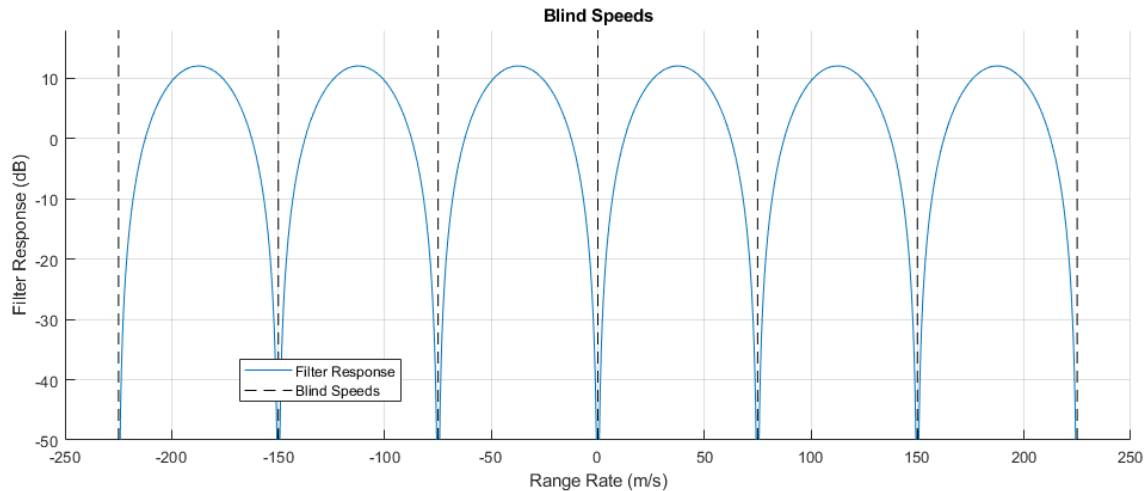
For a target to be detectable with range-only measurements, the target signal power must exceed that of the total clutter signal power in the target's range gate. The goal of MTI filtering is to reduce as much of the clutter signal power as possible, while preserving most of the target signal power.

Beam and Platform Geometry

Surface clutter is distributed over 360 degrees of azimuth, almost 90 degrees of elevation angle, and is bounded in range only by the horizon. When illuminated by a conical beam, the presentation of surface clutter depends on the geometry of the beam (shape and pointing direction), and on the motion of the radar platform. The detectability of a surface target with MTI processing is dependent on this clutter presentation as well as the radial speed of the target, which is defined by the absolute speed and direction of the target motion. The required positive or negative range rate for detection can depend on the location of the target within the mainlobe. A target that is located towards the high-Doppler edge of clutter might be detectable if it is moving towards the radar, but not be detectable if it is moving away from the radar at the same speed. There is thus an asymmetry in the detectable velocities for targets in all cases except for that of a broadside trajectory with the target located along the boresight vector.

The diagram below illustrates how the endoclutter region in range/cross-range changes with target speed when the target velocity is fixed in the uprange direction. In the first two subplots, the example target falls in the endoclutter region and is not detectable with MTI processing. In the 3rd subplot, the target is moving quickly enough to fall in the exoclutter region and should be detectable with MTI processing.
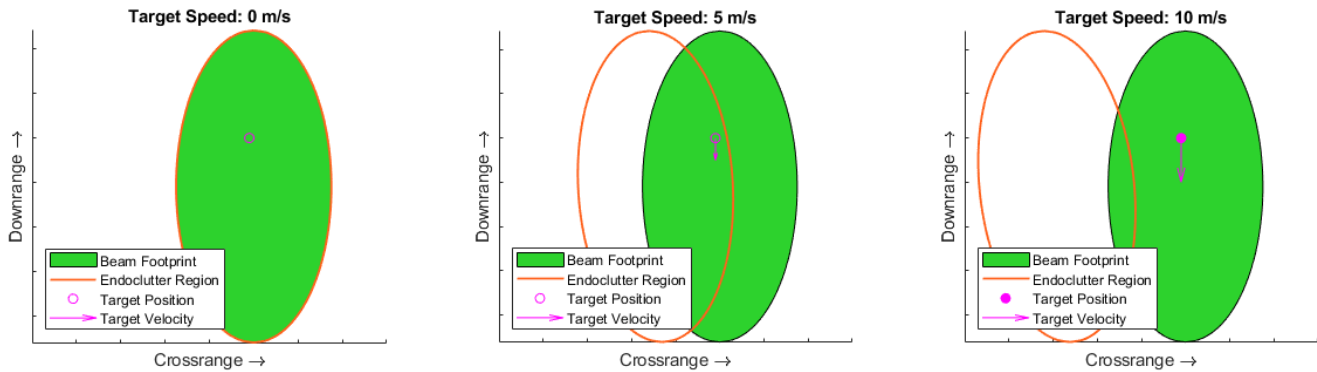
This next diagram illustrates how the direction of target motion affects the radial speed and thus detectability of a target. In this scenario, the target is located close to the high-Doppler edge of the beam footprint and has a constant speed of 10 m/s. Since the target is already close to the high-Doppler edge of the beam, there is a large span of directions of motion that place the target in the exoclutter region at a higher Doppler than clutter, while there is only a small span of directions of motion that place the target at a lower Doppler than clutter.
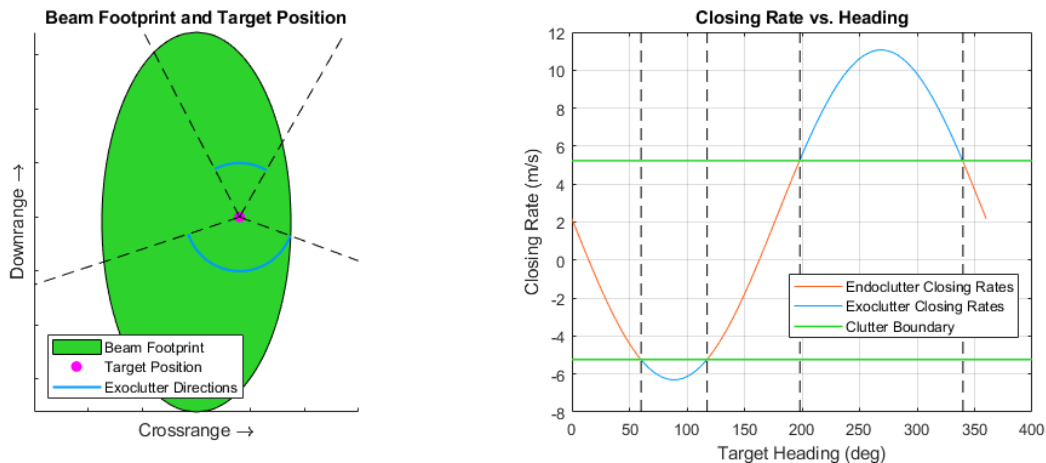
An analytical treatment is further complicated by the asymmetric beamwidth that results from any steering off broadside with a phased array, and from any rolling of the sensor frame about boresight. A physical simulation that supports such arbitrary platform geometries can be important for estimating the real-world performance of an MTI system.

Performance Metrics

While an exact solution for the detectability of a target under MTI processing is complex in general, a couple useful metrics may be computed with relative simplicity. The most commonly referenced is the minimum detectable velocity (MDV), the radial speed at which a target becomes detectable. This metric typically uses the simplifying assumption that the target lies exactly along the radar boresight. For broadside trajectories, which are the most common, this also means that the detectable velocity is symmetric about positive and negative range rates. The actual required velocity for detection is also affected by any broadening of the clutter spectrum for Doppler sidelobe control. In general, a balance must be found between broadening of the clutter spectrum and sidelobe levels required to detect the target.

Another useful metric is the MTI improvement factor, which relates the signal-to-clutter ratio (SCR) before and after MTI processing. An example demonstrating the effect of system parameters on the improvement factor is available here: "MTI Improvement Factor for Land-Based Radar" on page 1-406.

Broadside MTI Simulation

In this section, you will see how to configure a radar scenario to simulate clutter and target returns using a basic broadside trajectory. You will inspect the resulting range profiles for the case of a target that lies along the radar boresight, both above and below the MDV, then see what happens when the target is offset in the cross-range direction.

Scenario Configuration

A simulation involving radar clutter may be configured in four steps:

- Create a radar model and define system parameters and physical mounting angles
- Create a scenario container and define the Earth type, surface information, and simulation stop time
- Create scenario platforms, defining their trajectories and mounting sensors
- Create a clutter generator, defining clutter generation parameters for each radar

Configure Radar

The `radarTransceiver` object is used to simulate signal-level radar returns. This radar model contains sub-objects to handle various components of a monostatic radar system. Start by defining the desired system parameters, then use the provided helper function to construct the radar. Use parameters appropriate for an X-band side-looking airborne MTI radar that is configured to detect surface targets moving between about 20 to 40 mph. The antenna will be a symmetric uniform rectangular array (URA).

```
fc = 10e9;
prf = 7e3;
rangeRes = 20;
beamwidth = 5;
rdr = helperMakeTransceiver(beamwidth,fc,rangeRes,prf);
```

Set the radar mounting parameters. The radar will face out the left side of the platform (90 degree yaw), and down at the specified depression angle, with no roll about boresight.

```
depressionAngle = 18;
rdr.MountingAngles = [90 depressionAngle 0];
```

The total number of pulses collected on each activation of the radar model is defined by the `NumRepetitions` property. With range-only MTI processing, the pulses will be integrated noncoherently. Set the number of pulses to 64.

```
numPulses = 64;
rdr.NumRepetitions = numPulses;
```

Calculate the wavelength, Doppler and range-rate resolutions, and range gate centers for later use. The range swath extends from 0 out to the unambiguous range.

```
[lambda,c] = freq2wavelen(fc);
dopRes = prf/numPulses;
```

```
rangeRateRes = dop2speed(dopRes,lambda)/2;
unambRange = time2range(1/prf);
rangeGates = 0:rangeRes:(unambRange-rangeRes/2);
```

Configure Scenario

Create a `radarScenario` object. Use a flat Earth and set the update rate to 0 to indicate that the update rate should be derived from sensors in the scene.

```
scene = radarScenario('IsEarthCentered',false,'UpdateRate',0);
```

The scenario `landSurface` method can be used to define a homogeneous, static surface with an associated reflectivity model. Create a constant-gamma reflectivity model with the `surfaceReflectivityLand` function, and use a gamma value of -10 dBsm.

```
gamma = -10;
refl = surfaceReflectivityLand('Model','ConstantGamma','Gamma',gamma);
landSurface(scene,'RadarReflectivity',refl);
```

Configure Platforms

Create the radar platform. Let the platform fly a straight-line trajectory at the specified altitude and speed. The platform will move in the +X direction at 180 m/s.

```
altitude = 2e3;
rdrspd = 180;
rdrtraj = kinematicTrajectory('Position',[0 0 altitude],'Velocity',[rdrspd 0 0]);
rdrplat = platform(scene,'Sensors',rdr,'Trajectory',rdrtraj);
```

Create a target platform that lies on the surface along the antenna boresight, and give it a velocity vector such that the return signal will be separated from clutter in Doppler. Start by calculating the expected spread of clutter in Doppler. For broadside geometries and a symmetric beam, the two-sided extent of clutter in Doppler, W_D, can be calculated with a simple equation:

$$W_D = \frac{4}{\lambda}|v|\sin\left(\frac{\theta}{2}\right)$$

where λ is the wavelength, $|v|$ is the radar speed, and θ is the two-sided beamwidth. In reality, Doppler sidelobes will extend beyond this width, and this width may be increased by slow-time windowing. The MDV may be seen as a function of this width and of the MTI filter response. Calculate the Doppler width of clutter.

```
Wd = 4/lambda*rdrspd*sind(beamwidth/2);
```

To place the target along the antenna boresight, calculate the required ground range from the radar altitude and boresight depression angle. Recall that the radar is pointed in the +Y direction.

```
gndrng = altitude/tand(depressionAngle);
tgtrng = sqrt(gndrng^2 + altitude^2);
bspos = [0 gndrng 0];
```

Calculate a closing rate that places the target at 500 Hz greater than the high-Doppler clutter edge. Use the helper function to calculate the required speed for a target moving in the -Y direction, and create a kinematic trajectory.

```
tgtClosingRate = (Wd/2 + 500)*lambda/2;
tgtvel = helperTargetVelocityFromClosing(rdrplat.Position,rdrplat.Trajectory.Velocity,bspos,tgtC
tgttraj = kinematicTrajectory('Position',bspos,'Velocity',tgtvel);
```

In order to understand the effect of the MTI filter on detectability, express the target RCS in terms of the expected clutter RCS. The NRCS can be found using the reflectivity model created earlier, and `clutterSurfaceRCS` can be used to approximate the total RCS of clutter in the range gate containing the target and within the span of the mainlobe. Express the RCS in dBsm.

```
clutterNRCS = refl(depressionAngle,fc);
clutterRCS = clutterSurfaceRCS(clutterNRCS,tgtrng,beamwidth,beamwidth,depressionAngle,rdr.Wavefor
clutterRCS = 10*log10(clutterRCS);
```

Set the target RCS to be 8 dBsm less than the clutter RCS. Without any processing, the target will not be detectable.

```
tgtrcs = clutterRCS - 8; % dBsm
tgtplat = platform(scene,'Trajectory',tgttraj,'Signatures',rcsSignature('Pattern',tgtrcs));
```

The figure below shows the beam footprint with this scenario, and the location and velocity vector of the target.
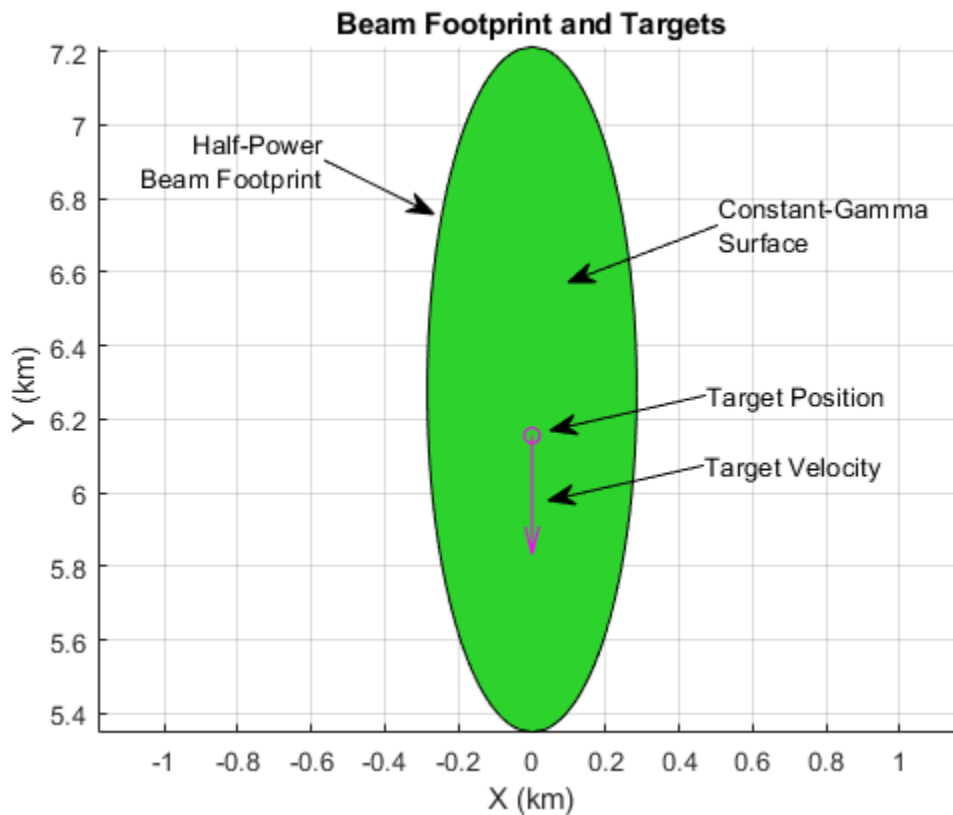
Configure Clutter Generation

Finally, create a `ClutterGenerator` and specify clutter generation settings. Set the clutter resolution to one half of the radar range resolution to ensure multiple clutter patches per resolution cell (the range resolution is smaller than the cross-range resolution with this radar configuration). Set `UseBeam` to true to generate mainlobe clutter.

```
clutterGenerator(scene,rdr,'Resolution',rangeRes/2,'UseBeam',true);
```

Simulate Clutter and Target Return Signals

Use the scenario `receive` method to collect one interval of pulse data. The result is formatted as fast time-by-slow time, and is referred to as a phase history (PH) matrix. This sum beam is an accumulation of the typical radar data cube across channels. The output will be a cell array, with one cell for each `radarTransceiver` in the scene. This scenario has only one radar, so unpack PH into a matrix by accessing the first element.

```
PH = receive(scene);
PH = PH{1};
```

Process the PH data with the MTI filter using the provided helper function. The resulting matrix will be formatted similarly to the original PH matrix, except it has been filtered along the slow-time dimension and has 2 fewer pulses of data to omit the invalid initial filter outputs. The filter has been normalized by the expected noise gain for easier comparison of the results.

```
PH_mti = helperMTIFilter(PH);
```

Plot the original phase history matrix alongside the filtered matrix.

```
helperPlotPhaseHistory(PH,PH_mti,rangeGates)
```
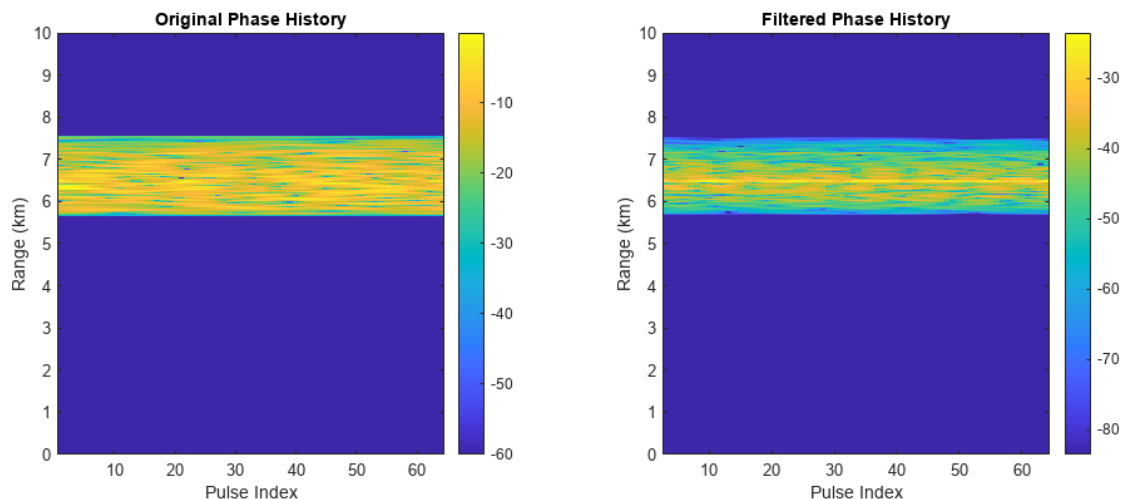

In the filtered phase history, the target return is visible above clutter on most pulses.

To average out the small fluctuations in clutter and target power, MTI systems will often perform a noncoherent integration of the filtered pulses to form a final range profile. Use the `pulsint` function to form range profiles from the original PH matrix and the filtered PH matrix. Use the provided helper function to plot the range profiles along with the filter response and annotations indicating the target range and radial speed. The Doppler spectra before and after filtering are also shown. These spectra are computed by accumulating the power in each Doppler bin over all range gates.

```
rangeProfile = pulsint(PH);
rangeProfile_mti = pulsint(PH_mti);
helperPlotMTI(rdrplat,tgtplat,rangeProfile,rangeProfile_mti,rangeGates,prf,lambda,PH,PH_mti)
```
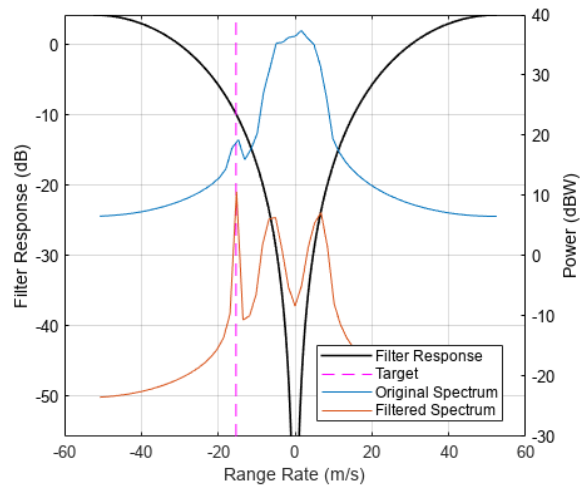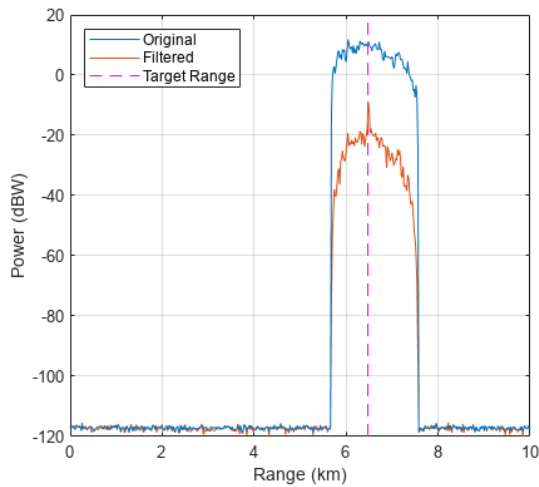
In the original range profile, the target return is not detectable over clutter. In the filtered range profile, a peak at the target range is clearly visible.

The target was separated from the edge of clutter by 500 Hz. The signal component at the Doppler frequency of the target saw an attenuation by about 10 dB while clutter was attenuated by about 21 dB at the high-Doppler edge, which is the *minimum* attenuation seen by the mainlobe clutter signal (not including Doppler sidelobes). After filtering, the total clutter power in a range bin is dominated by the least-attenuated component of the signal, so the total gain in SCR is at least 21-10=11 dB. Since the clutter RCS was only 8 dBsm greater than the target RCS, this gain in SCR is sufficient to detect the target.

Now adjust the target velocity to place it 50 Hz lower than the high-Doppler edge of the clutter return.

```
tgtClosingRate = (Wd/2 - 50)*lambda/2;
tgtvel = helperTargetVelocityFromClosing(rdrplat.Position,rdrplat.Trajectory.Velocity,bspos,tgtC
tgttraj.Velocity = tgtvel;
```

The figure below shows the beam footprint with this slow-moving target.

Slow-Moving Target
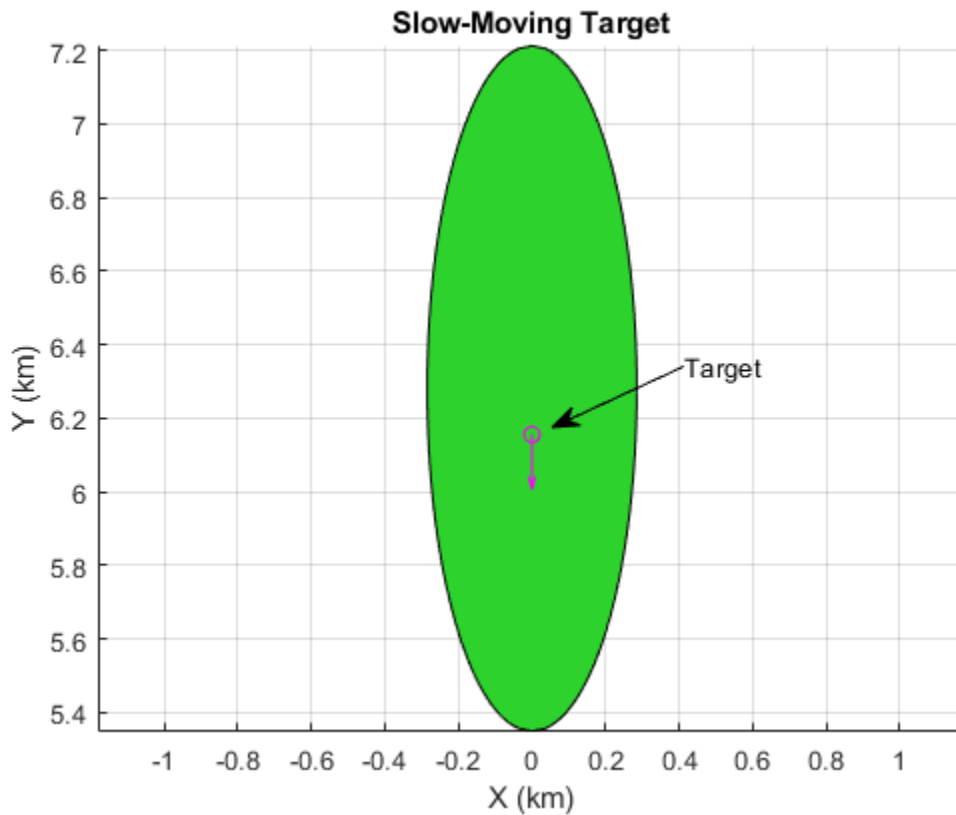
Use the provided helper function to re-run the scenario and collect results.

```
[rangeProfile,rangeProfile_mti,PH,PH_mti] = helperRunMTIScenario(scene);
```

Plot the resulting range profiles, filter response, and spectra.

```
helperPlotMTI(rdrplat,tgtplat,rangeProfile,rangeProfile_mti,rangeGates,prf,lambda,PH,PH_mti)
```
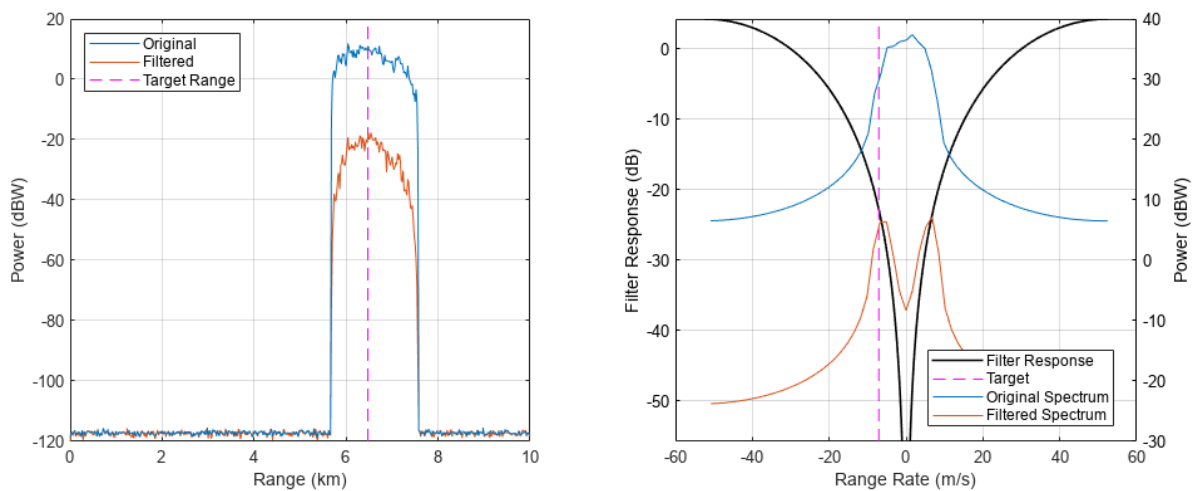
In this case, the target signal is not benefitting from any gain over the clutter signal, and is not detectable.

Offset Target Position

Now, without changing the velocity vector, offset the target from the boresight-ground intersection point by 260 m in the +X direction. Because this is the direction of radar motion, the target is now close to the high-Doppler edge of the clutter signal.

```
tgttraj.Position = bspos + [260 0 0];
```

The figure below shows this new geometry.
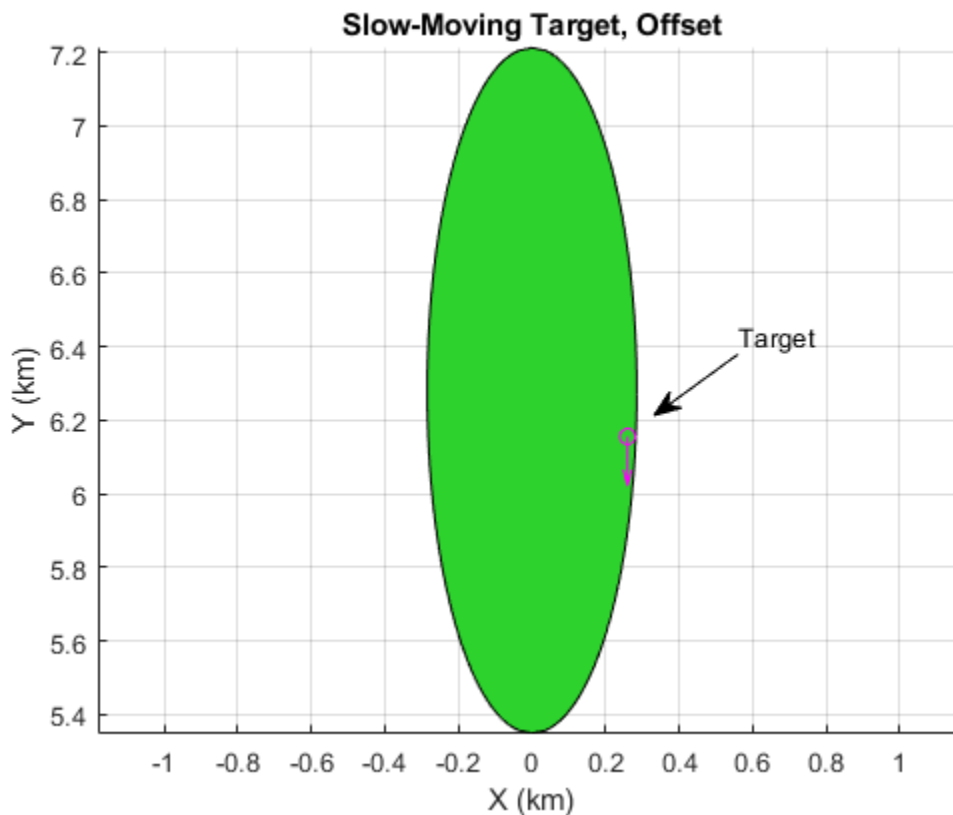
Repeat the simulation and plot the results.

```
[rangeProfile,rangeProfile_mti,PH,PH_mti] = helperRunMTIScenario(scene);
helperPlotMTI(rdrplat,tgtplat,rangeProfile,rangeProfile_mti,rangeGates,prf,lambda,PH,PH_mti)
```
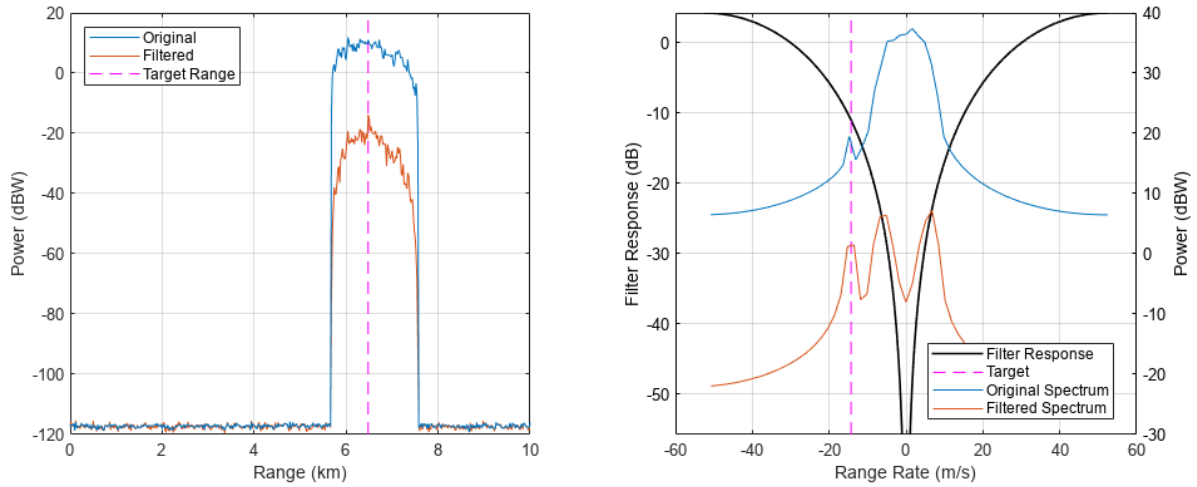
While not as prominent as in the case of the fast-moving target, a peak corresponding to the target range can be seen thanks to the off-broadside location of the target within the mainlobe.

Now, with the target still offset towards the high-Doppler edge of the clutter signal, reverse the direction of the target velocity vector.

```
tgttraj.Velocity = -tgttraj.Velocity;
```
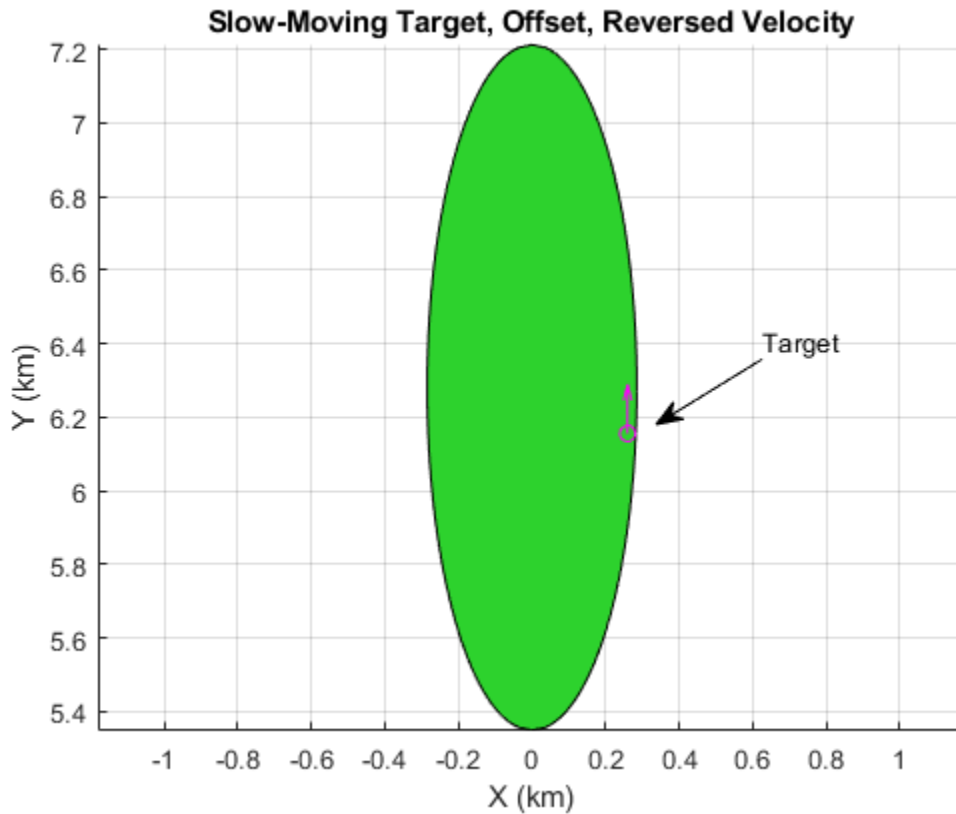

1-855

```
[rangeProfile,rangeProfile_mti,PH,PH_mti] = helperRunMTIScenario(scene);
helperPlotMTI(rdrplat,tgtplat,rangeProfile,rangeProfile_mti,rangeGates,prf,lambda,PH,PH_mti)
```
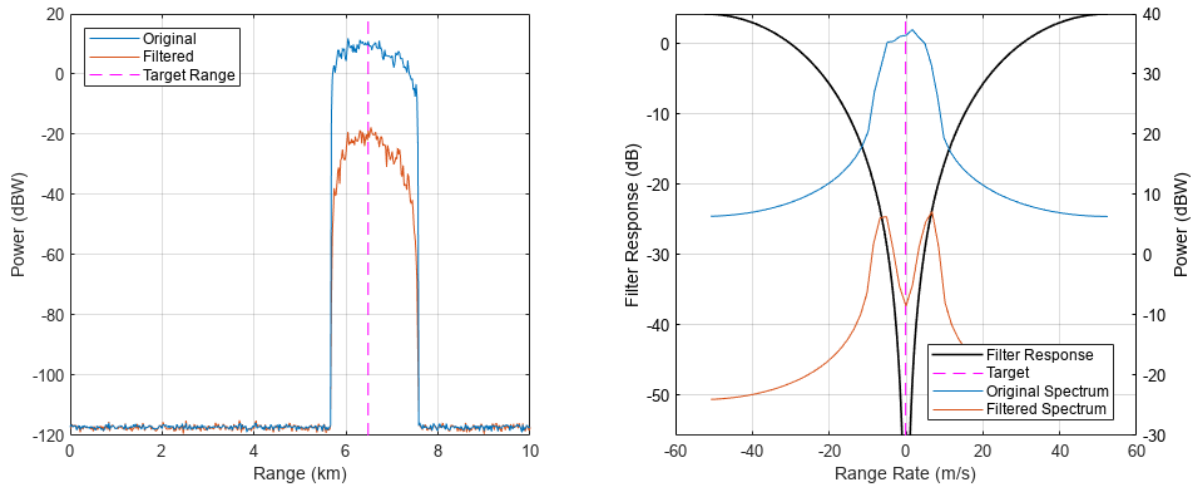

The peak has again disappeared thanks to the opposing influence of the target position within the mainlobe and its velocity vector.

Squinted MTI Simulation

In this section you will simulate MTI filtering of squinted beam data in order to inspect the asymmetry in the detectable velocities across range. Mainlobe clutter under a squinted beam will exhibit *slant*: a varying Doppler centroid over range. If a single MTI filter is used across all range gates, this means that the effectiveness of the filter could vary over range.

The radar `MountingAngles` property can be used to rotate the radar frame so that the boresight is pointed as desired. The depression angle will be kept constant. Use a forward-squinted trajectory. Release the radar and set the mounting angles.

```
azPointing = 10;
release(rdr)
rdr.MountingAngles = [azPointing depressionAngle 0];
```

Get the antenna boresight direction, and a ground-projected boresight vector for placing targets on the ground along the same direction.

```
[bs(1),bs(2),bs(3)] = sph2cart(azPointing*pi/180,-depressionAngle*pi/180,1);
bsxy = [bs(1:2)/norm(bs(1:2)) 0];
```

Calculate approximate range and Doppler clutter centroids from the boresight vector. This will be used to place targets at ranges and Dopplers relative to clutter.

```
centerRange = -altitude/bs(3);
centerDop = 2/lambda*rdrspd*cosd(azPointing)*cosd(depressionAngle);
```

Select a closing rate that puts the targets 200 Hz above the calculated clutter centroid.

```
tgtClosingRate = (centerDop + 200)*lambda/2
```

```
tgtClosingRate = 171.5873
```

Place the first target platform along the boresight vector (as in the last section), calculate the required velocity vector, and update the trajectory properties.

```
tgtrng1 = centerRange;
tgtpos1 = sqrt(tgtrng1^2-altitude^2)*bsxy;
tgtvel1 = helperTargetVelocityFromClosing(rdrplat.Position,rdrplat.Trajectory.Velocity,tgtpos1,tg
tgttraj.Position = tgtpos1;
tgttraj.Velocity = tgtvel1;
```

Create a second platform and place it 800 m further along the boresight direction, and give it the same closing rate.

```
tgtrng2 = centerRange + 800;
tgtpos2 = sqrt(tgtrng2^2-altitude^2)*bsxy;
tgtvel2 = helperTargetVelocityFromClosing(rdrplat.Position,rdrplat.Trajectory.Velocity,tgtpos2,tg
tgttraj2 = kinematicTrajectory('Position',tgtpos2,'Velocity',tgtvel2);
tgtplat(2) = platform(scene,'Trajectory',tgttraj2,'Signatures',rcsSignature('Pattern',tgtrcs));
```

The figure below shows the location of the beam footprint, targets, and their velocity vectors.
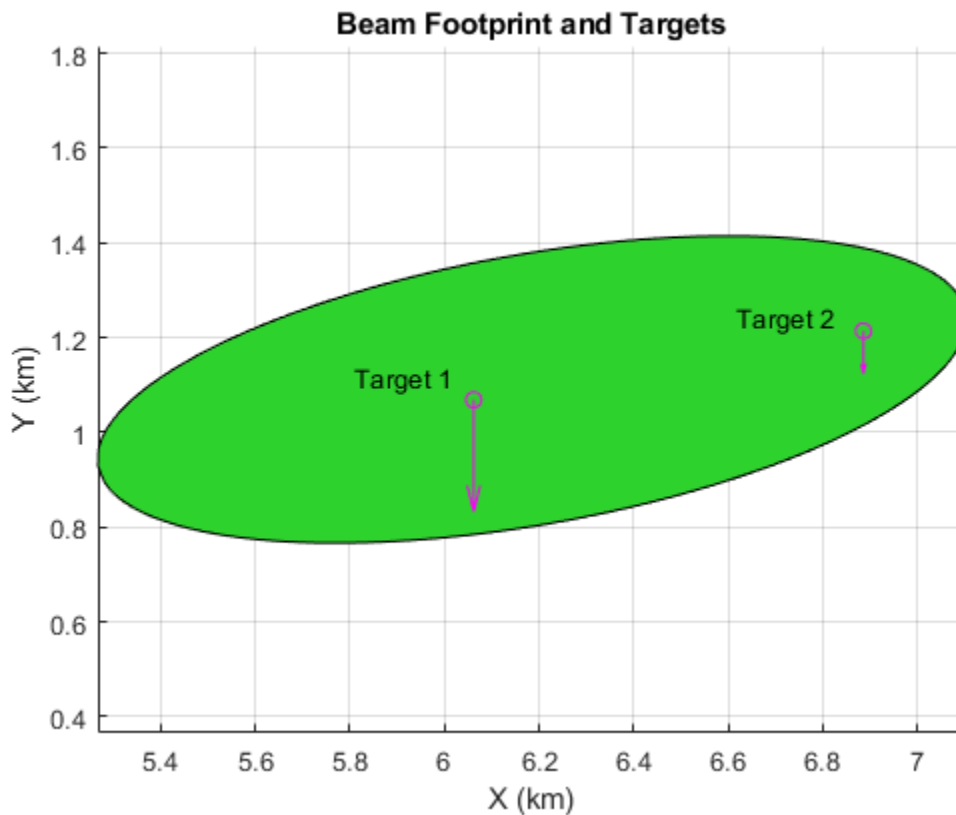
Run the simulation and plot the resulting range profiles, filter response, and spectra.

```
[rangeProfile,rangeProfile_mti,PH,PH_mti] = helperRunMTIScenario(scene,true,centerDop/prf);
helperPlotMTI(rdrplat,tgtplat,rangeProfile,rangeProfile_mti,rangeGates,prf,lambda,PH,PH_mti,cente
```
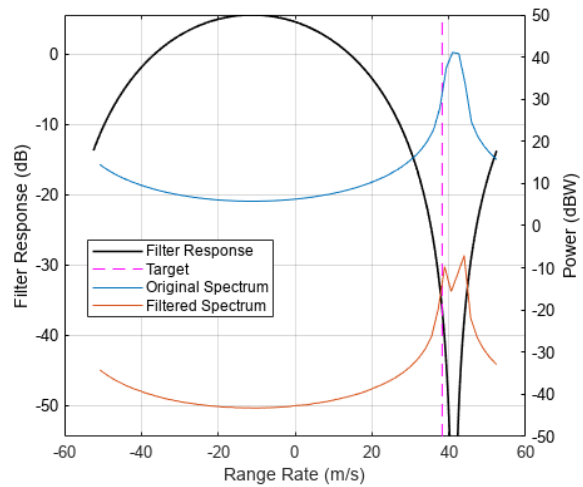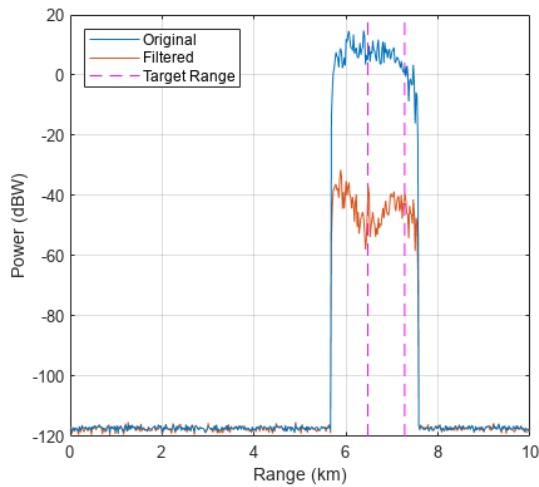
The first important observation of the range profiles is the inflection point around the center range. The low- and high-range sides of the clutter return were offset from the Doppler centroid by a larger amount than in the broadside case, and experienced lower attenuation than the attenuation at the centroid. As such, the target signal at that range is visible in the filtered data. The second target at a greater range is not visible because the clutter return at that range was not attenuated as well.

Note that in this scenario, while the platforms are moving at speeds of about 18 and 7 m/s for targets 1 and 2 respectively, their closing rates are both large enough (about 172 m/s) to be ambiguous and appear as if they are moving away from the radar at about 40 m/s. This range-rate ambiguity would typically be resolved by using multiple PRFs or by feeding measurements into a tracker.

To better see the range dependence on the clutter Doppler spectra, plot the full range-Doppler map (RDM) before and after filtering, using the provided helper function. The displayed region will be between 4 and 9 km range and -3.5 to -2 kHz to see the data more clearly.

```
helperPlotRDM(PH,PH_mti,rangeGates,prf,lambda,rdrplat,tgtplat)
```
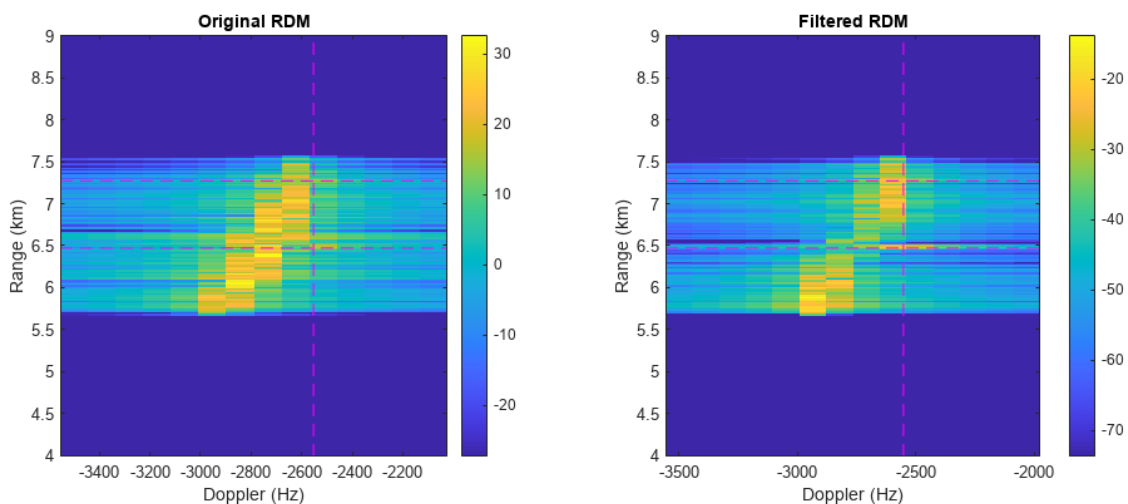
From the RDM it can be seen that much of the clutter signal at the range of the first target (about 6.5 km) has been attenuated below the target signal power, while the clutter signal at the range of the second target has not been as well attenuated.

Conclusion

In this example you saw how to configure a radar scenario to generate target and clutter returns appropriate for simulation of MTI performance. The effect of target placement and direction of motion with the mainlobe during both broadside and squinted collections was demonstrated.

Supporting Functions

helperMakeTransceiver

```
function [rdr,bw] = helperMakeTransceiver( bw,fc,rangeRes,prf )
% This helper function creates a radarTransceiver from some basic system
% parameters.

c = physconst('lightspeed');

rdr = radarTransceiver;

rdr.TransmitAntenna.OperatingFrequency = fc;
rdr.ReceiveAntenna.OperatingFrequency = fc;

sampleRate = c/(2*rangeRes);
sampleRate = prf*round(sampleRate/prf); % adjust to match constraint with PRF

rdr.Receiver.SampleRate = sampleRate;
rdr.Waveform.SampleRate = sampleRate;
rdr.Waveform.PulseWidth = 2*rangeRes/c;
rdr.Waveform.PRF = prf;

% Create a rectangular array with a beamwidth close to the specification
N = round(0.8859*2./(bw(:).'*pi/180));
bw = 0.8859*2./N*180/pi; % Achieved beamwidth
N = flip(N);
lambda = freq2wavelen(fc,c);
array = phased.URA(N,lambda/2);
az = -180:.4:180;
el = -90:.4:90;
G = pattern(array,fc,az,el,'Type','efield','normalize',false);
M = 20*log10(abs(G));
P = angle(G);
E = phased.CustomAntennaElement('FrequencyVector',[fc-1 fc+1],...
    'AzimuthAngles',az,'ElevationAngles',el,'MagnitudePattern',M,'PhasePattern',P);
rdr.TransmitAntenna.Sensor = E;
rdr.ReceiveAntenna.Sensor = E;

end
```

helperTargetVelocityFromClosing

```
function vel = helperTargetVelocityFromClosing( rdrpos,rdrvel,tgtpos,tgtClosingRate )
% Calculates the required speed in the -Y direction for the target platform
% to have the desired closing rate.

los = rdrpos(:) - tgtpos(:);
```

```matlab
los = los/norm(los);
spd = (tgtClosingRate + dot(rdrvel(:),los))/los(2);
vel = [0 spd 0];

end
```

helperMTIFilter

```matlab
function y = helperMTIFilter( x,cdn )
% Filter the input x across the second dimension using a second-order
% central difference.

% Filter coefficients
h = [1 -2 1];

% If non-zero center Doppler, apply phase ramp to filter coefficients
if nargin > 1
    phi = 2*pi*cdn*(0:numel(h)-1);
    h = h.*exp(1i*phi);
end

% Run filter and omit invalid output samples
y = filter(h,1,x,[],2);
y = y(:,numel(h):end);

% Normalize by the noise gain
y = y/sqrt(sum(abs(h).^2));

end
```

helperPlotPhaseHistory

```matlab
function helperPlotPhaseHistory( PH,PH_mti,rg )
% This function plots the phase history matrices from before and after MTI
% filtering

figure;
set(gcf,'Position',get(gcf,'Position')+[0 0 560 0])

% Limit displayed ranges to 10 km
maxRange = 10e3;
maxGate = find(rg <= maxRange,1,'last');
rg = rg(1:maxGate);
PH = PH(1:maxGate,:);
PH_mti = PH_mti(1:maxGate,:);

numPulses = size(PH,2);

subplot(1,2,1)
imagesc(1:numPulses,rg/1e3,20*log10(abs(PH)))
set(gca,'ydir','normal')
cx = caxis;
caxis([cx(2)-60 cx(2)])
colorbar
xlabel('Pulse Index')
ylabel('Range (km)')
title('Original Phase History')
```

```matlab
subplot(1,2,2)
imagesc(3:numPulses,rg/1e3,20*log10(abs(PH_mti)))
set(gca,'ydir','normal')
cx = caxis;
caxis([cx(2)-60 cx(2)])
colorbar
xlabel('Pulse Index')
ylabel('Range (km)')
title('Filtered Phase History')

end
```

helperPlotMTI

```matlab
function helperPlotMTI( rdrplat,tgtplat,rp,rp_mti,rg,prf,lambda,PH,PH_mti,centerDop )
% This function plots the range profiles from before and after MTI
% processing, and plots the filter response along with the clutter spectrum
% and annotations.

fh = figure;

% Calculate range/Doppler to tgt
for ind = 1:numel(tgtplat)
    tgtrng(ind) = norm(rdrplat.Position - tgtplat(ind).Position);
    los = (rdrplat.Position-tgtplat(ind).Position)/tgtrng(ind);
    tgtClosingRate(ind) = sum((tgtplat(ind).Trajectory.Velocity-rdrplat.Trajectory.Velocity).*los
    tgtdop(ind) = 2/lambda*tgtClosingRate(ind);
end

% Limit displayed ranges to 10 km
maxRange = 10e3;
maxGate = find(rg <= maxRange,1,'last');
rg = rg(1:maxGate);
rp = rp(1:maxGate);
rp_mti = rp_mti(1:maxGate);

% Range-power plots
subplot(1,2,1)

% Plot range profiles
hndls1(1) = plot(rg/1e3,20*log10(rp)); % original
hold on
hndls1(2) = plot(rg/1e3,20*log10(rp_mti)); % filtered

% Plot truth target ranges
for ind = 1:numel(tgtrng)
   hndls1(3) = line([1 1]*tgtrng(ind)/1e3,ylim,'color','magenta','linestyle','--');
end
hold off
grid on
xlabel('Range (km)')
ylabel('Power (dBW)')
set(gca,'Children',flip(get(gca,'Children')))
legend(hndls1,{'Original','Filtered','Target Range'},'Location','NorthWest')

% Spectrum plots
subplot(1,2,2)
```

1-861

```matlab
% Plot filter response
h = [1 -2 1];
if nargin > 9
    phi = 2*pi*centerDop/prf*(0:numel(h)-1);
    h = h.*exp(1i*phi);
end
f = linspace(-prf/2,prf/2,1e3);
rr = -lambda/2*f;
hresp = freqz(h,1,f,prf);
hresp = hresp/sqrt(sum(h.^2)); % normalize to keep noise floor constant
hresp = 20*log10(abs(hresp));
hndls2(1) = plot(rr,hresp,'black','linewidth',1.2);
mn = max(max(hresp)-60,min(hresp)); % limit to 60 dB down
ylim([mn max(hresp)])
hold on

% Plot annotation at tgt RR
tgtdop = mod(tgtdop+prf/2,prf)-prf/2;
tgtrr = -tgtdop*lambda/2;
for ind = 1:numel(tgtrr)
    hndls2(2) = line([1 1]*tgtrr(ind),ylim,'color','magenta','linestyle','--');
end

xlabel('Range Rate (m/s)')
ylabel('Filter Response (dB)')

% Get before/after Doppler spectra and plot
numPulses = size(PH,2);
rrBins = -(-prf/2:prf/numPulses:(prf/2-prf/numPulses))*lambda/2;
RDM = fftshift(fft(PH,[],2),2);
dopSpec = sum(abs(RDM).^2,1);
numPulses_mti = size(PH_mti,2);
rrBins_mti = -(-prf/2:prf/numPulses_mti:(prf/2-prf/numPulses_mti))*lambda/2;
RDM_mti = fftshift(fft(PH_mti,[],2),2);
dopSpec_mti = sum(abs(RDM_mti).^2,1);

yyaxis right
hndls2(3) = plot(rrBins,10*log10(abs(dopSpec)),'Color',hndls1(1).Color,'linestyle','-');
hndls2(4) = plot(rrBins_mti,10*log10(abs(dopSpec_mti)),'Color',hndls1(2).Color,'linestyle','-');
hold off
grid on
ylabel('Power (dBW)')
set(gca,'YColor','black')

legend(hndls2,{'Filter Response','Target','Original Spectrum','Filtered Spectrum'},'Location','Be

set(fh,'Position',get(fh,'Position')+[0 0 560 0])

end
```

helperRunMTIScenario

```matlab
function [ rp,rp_mti,PH,PH_mti,RDM ] = helperRunMTIScenario( scene,doRestart,centerDopNorm )

if nargin < 2
    doRestart = true;
end
```

```matlab
if nargin < 3
    centerDopNorm = 0;
end

if doRestart
    restart(scene)
end

% Call scenario receive method
PH = receive(scene);
PH = PH{1};

% Perform MTI filtering
PH_mti = helperMTIFilter(PH,centerDopNorm);

% Generate before/after range profiles
rp = pulsint(PH);
rp_mti = pulsint(PH_mti);

% Perform Doppler processing
RDM = fftshift(fft(PH,[],2),2);

end
```

helperPlotRDM

```matlab
function helperPlotRDM( PH,PH_mti,rg,prf,lambda,rdrplat,tgtplat )

figure;
set(gcf,'Position',get(gcf,'Position')+[0 0 560 0])

% Calculate range/Doppler to tgt
for ind = 1:numel(tgtplat)
    tgtrng(ind) = norm(rdrplat.Position - tgtplat(ind).Position);
    los = (rdrplat.Position-tgtplat(ind).Position)/tgtrng(ind);
    tgtClosingRate(ind) = sum((tgtplat(ind).Trajectory.Velocity-rdrplat.Trajectory.Velocity).*los
    tgtdop(ind) = 2/lambda*tgtClosingRate(ind);
end
tgtdop = mod(tgtdop+prf/2,prf)-prf/2;

% Limit displayed ranges to 3-10 km
maxRange = 9e3;
minRange = 4e3;
maxGate = find(rg <= maxRange,1,'last');
minGate = find(rg >= minRange,1,'first');
rg = rg(minGate:maxGate);
PH = PH(minGate:maxGate,:);
PH_mti = PH_mti(minGate:maxGate,:);

maxDop = -2e3;

subplot(1,2,1)
RDM = fftshift(fft(PH,[],2),2);
numPulses = size(PH,2);
dopBins = -prf/2:prf/numPulses:prf/2-prf/numPulses;
maxDopBin = find(dopBins <= maxDop,1,'last');
imagesc(dopBins(1:maxDopBin),rg/1e3,20*log10(abs(RDM(:,1:maxDopBin))))
for ind = 1:numel(tgtplat)
```

```matlab
        line(xlim,[1 1]*tgtrng(ind)/1e3,'color','magenta','linestyle','--')
        line([1 1]*tgtdop(ind),ylim,'color','magenta','linestyle','--')
    end
    set(gca,'ydir','normal')
    cx = caxis;
    caxis([cx(2)-60 cx(2)])
    colorbar
    xlabel('Doppler (Hz)')
    ylabel('Range (km)')
    title('Original RDM')

    subplot(1,2,2)
    RDM_mti = fftshift(fft(PH_mti,[],2),2);
    numPulses_mti = size(PH_mti,2);
    dopBins_mti = -prf/2:prf/numPulses_mti:prf/2-prf/numPulses_mti;
    maxDopBin = find(dopBins_mti <= maxDop,1,'last');
    imagesc(dopBins_mti(1:maxDopBin),rg/1e3,20*log10(abs(RDM_mti(:,1:maxDopBin))))
    for ind = 1:numel(tgtplat)
        line(xlim,[1 1]*tgtrng(ind)/1e3,'color','magenta','linestyle','--')
        line([1 1]*tgtdop(ind),ylim,'color','magenta','linestyle','--')
    end
    set(gca,'ydir','normal')
    cx = caxis;
    caxis([cx(2)-60 cx(2)])
    colorbar
    xlabel('Doppler (Hz)')
    ylabel('Range (km)')
    title('Filtered RDM')

end
```

Quality-of-Service Optimization for Radar Resource Management

This example shows how to set up a resource management scheme for a multifunction phased array radar (MPAR) surveillance based on a quality-of-service (QoS) optimization. It starts by defining parameters of multiple search sectors that must be surveyed simultaneously. It then introduces the cumulative detection range as a measure of search quality and shows how to define suitable utility functions for QoS optimization. Finally, the example shows how to use numeric optimization to allocate power-aperture product (PAP) to each search sector such that QoS is maximized.

MPAR Resource Management

Multifunction Phased Array Radar (MPAR) uses automated resource management to control individual elements of the antenna array [1]. This allows MPAR to partition the antenna array into a varying number of subarrays to create multiple transmit and receive beams. The number of beams, their positions, as well as the parameters of the transmitted waveforms can be controlled on the dwell-to-dwell basis. This allows MPAR to perform several functions such as surveillance, tracking, and communication simultaneously. Each such function can comprise one or multiple tasks that are managed by the radar resource manager (RRM). Based on the mission objectives and a current operational situation, the RRM allocates some amount of radar resources to each task.

This example focuses on the resource management of the surveillance function. Since MPAR is performing multiple functions at the same time, only a fraction of the total radar resources (typically 50% under normal conditions [2]) are available for surveillance. This amount can be further reduced if the RRM has to reallocate resources from the surveillance to other functions.

Search Sectors

The surveillance volume of an MPAR is typically divided into several sectors. Each such sector has different range and angle limits and a different search frame time - the amount of time it takes to scan the sector once [2]. The RRM can treat surveillance of each such sector as a separate independent search task. Thus, the resources available to the surveillance function must be balanced between several search sectors. This example develops an RRM scheme for the MPAR with three search sectors: horizon, long-range, and high-elevation. The long-range sector has the largest volume and covers targets that approach the radar from a long distance, while the horizon and the high-elevation sectors are dedicated to the targets that can emerge close to the radar site.
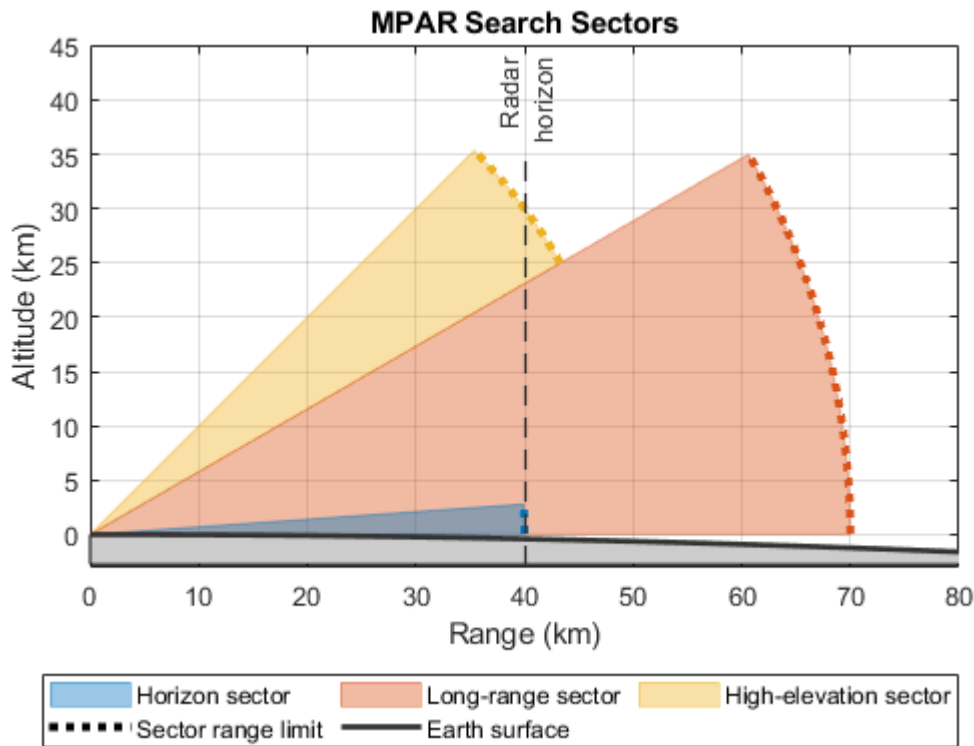
Set the horizon sector limits to -45 to 45 degrees in azimuth and 0 to 4 degrees in elevation, the long-range sector limits to -30 to 30 degrees in azimuth and 0 to 30 degrees in elevation, and the high-elevation sector limits to -45 to 45 degrees in azimuth and to 30 to 45 degrees in elevation.

```
% Number of search sectors
M = 3;

% Sector names
sectorNames = {'Horizon', 'Long-range', 'High-elevation'};

% Azimuth limits of the search sectors (deg)
azimuthLimits = [-45 -30 -45; 45 30 45];

% Elevation limits of the search sectors (deg)
elevationLimits = [0 0 30; 4 30 45];
```

The range limit of the horizon sector is determined by the distance beyond which a target becomes hidden by the horizon. Given the height of the radar platform this distance can be computed using the `horizonrange` function. The range limit of the long-range sector is determined by the instrumented range of the radar - the range beyond which the detection probability approaches zero. Finally, the range limit of the high-elevation sector is determined by the maximum target altitude. Set the range limits of the horizon, the long-range, and the high-elevation sectors to 40 km, 70 km, and 50 km respectively.

```
% Range limits (m)
rangeLimits = [40e3 70e3 50e3];
```

This example assumes that the MPAR performs surveillance of all three search sectors simultaneously. Since each sector has a different search volume, the search frame time, t_f, is different

for each sector. Additionally, each sector can have a different number of beam positions, N_b. Assuming the dwell time, T_d, (the time the beam spends in each beam position) is constant within a search sector, it is related to the search frame time as $T_d = t_f/N_b$. Set the search frame time for the horizon sector to 0.5 seconds, for the long-range sector to 6 seconds, and for the high-elevation sector to 2 seconds.

```
% Search frame time
frameTime = [0.5 6 2];
```

The search frame times are selected such that the closure range covered by an undetected target in one scan is significantly smaller than the target range. Consider a target with the radial velocity of 250 m/s that emerges at a sector's range limit. Compute the closure range for this target in each search sector.

```
frameTime*250
```

ans = *1×3*

      125   1500   500

In one scan this target will get closer to the radar by 125 m if it is in the horizon sector. Since the horizon sector covers targets that can emerge very close to the radar site, it is important that this sector is searched frequently and that an undetected target does not get too close to the radar before the next attempted detection. On the other hand, the range limit of the long-range sector is 70 km and the corresponding target closure distance is 1.5 km. Hence, an undetected target in the long-range sector will be able to cover only a small fraction of the sector's range limit in a single scan.

For convenience, organize the search sector parameters into an array of structs.

```
searchSectorParams(M) = helperSearchSector();

for i = 1:M
    searchSectorParams(i).SectorName = sectorNames{i};
    searchSectorParams(i).AzimuthLimits = azimuthLimits(:, i);
    searchSectorParams(i).ElevationLimits = elevationLimits(:, i);
    searchSectorParams(i).RangeLimit = rangeLimits(i);
    searchSectorParams(i).FrameTime = frameTime(i);
end
```

Quality-of-Service Resource Management

Under normal operational conditions the surveillance function is typically allocated enough resources such that the required detection performance is achieved in each search sector. But in circumstances when the surveillance function must compete for the radar resources with other radar functions, the RRM might allocate only a fraction of the desired resources to each search sector. These resources can be allocated based on hard rules such as sector priorities. However, coming up with rules that provide optimal resource allocation under all operational conditions is very difficult. A more flexible approach is to assign a metric to each task to describe the quality of the performed function [3]. The performance requirements can be specified in terms of the task quality, and the RRM can adjust the task control parameters to achieve these requirements.

This example assumes that resources are allocated to the search sectors in a form of PAP while the sectors' search frame times remain constant. It is convenient to treat PAP as a resource because it captures both the power and the aperture budget of the system. The presented resource allocation

approach can be further extended to also include the search frame time to optimize the time budget of the system.

Let $q_i(PAP_i; \theta_i)$ be a quality metric that describes performance of a search task created for the ith sector. Here PAP_i is the power aperture allocated by the RRM to the ith sector, and θ_i is the set of control and environmental parameters for the ith sector. To allocate the optimal amount of PAP to each search sector, the radar resource manager must solve the following optimization problem

$$\max_{PAP} u(PAP) = \sum_{i=1}^{M} w_i \cdot u_i(q_i(PAP_i; \theta_i))$$

such that $\sum_{i=1}^{M} PAP_i \leq PAP_{search}$

where

M is the total number of search sectors;

$PAP = [PAP_1, PAP_2, ..., PAP_M]$ is a vector that specifies the PAP allocation to each search sector;

PAP_{search} is the total amount of PAP available to the MPAR's surveillance function;

u_i is the utility function for ith search sector;

and w_i is the weight describing the priority of the ith search sector.

Prior to weighting each sector with a priority weight, the QoS optimization problem first maps the quality of the ith search tasks to the utility space. Such mapping is necessary because different tasks can have very different desired quality levels or even use entirely different metrics to express the task quality. A utility function associates a degree of satisfaction with each quality level according to the mission objectives. Optimizing the sum of weighted utilities across all tasks aims to allocate resources such that the overall satisfaction with the surveillance function is maximized.

Search Task Quality

The goal of a search task is to detect a target. As a target enters a search sector, it can be detected by the radar over multiple scans. With each scan the cumulative probability of detection grows as the target approaches the radar site. Therefore, a convenient metric that describes quality of a search task is the cumulative detection range R_c - the range at which the cumulative probability of detection reaches a desired value of P_c [4]. A commonly used P_c is 0.9 and the corresponding cumulative detection range is denoted as R_{90}.

Cumulative Probability of Detection

Let R_m be the range limit of a search sector beyond which the probability of detecting a target can be assumed to be equal to zero. The cumulative detection probability for a target that entered the search sector at R_m is then a function of the target range R, the target radial speed v_r, and the sector frame time t_f

$$P_c(R) = E_\Delta\{1 - \prod_{n=0}^{N} [1 - P_d(R_m - \Delta \cdot dR - n \cdot dR)]\}$$

where

$P_d(R)$ is the single-look probability of detection computed at range R;

$N = \lfloor (R_m - R)/t_f v_r \rfloor$ is the number of completed scans until the target reaches the range R from R_m;

$dR = t_f v_r$ is the closure range (distance traveled by the target in a single scan);

E_Δ is the expectation taken with respect to the uniformly distributed random variable $\Delta \sim U(0, 1)$ that represents the possibility of the target entering the search volume at any time within a scan.

Assuming a Swerling 1 case target and that all pulses within a dwell are coherently integrated, the single-look detection probability at range R is

$$P_d(R) = P_{fa}^{\dfrac{1}{1 + SNR_m \left(\dfrac{R_m}{R} \right)^4}}$$

where P_{fa} is the probability of false alarm;

and SNR_m is the reference SNR measured at the sector's range limit R_m.

It follows from the radar search equation that SNR_m depends on the PAP allocated to the search sector.

$$SNR_m = PAP \cdot \frac{\sigma}{4\pi k T_s R_m^4 L} \cdot \frac{t_f}{\Omega}$$

where

Ω is the volume of the sector;

T_s is the system temperature;

L is the combined operational loss;

σ is the target radar cross section (RCS);

and k is the Boltzmann's constant.

Use the `solidangle` function to compute the volume of a search sector and the `radareqsearchsnr` function to solve the radar search equation for SNR_m. Then use the `helperCumulativeDetectionProbability` function to compute the cumulative detection probability as a function of the target range given the found SNR_m.

First, assume that the system noise temperature is the same for each search sector and is equal to 913 K.

```
% System noise temperature (K)
Ts = [913 913 913];
```

Then, set the operational loss term for each sector.

```
% Operational loss (dB)
Loss = [22 19 24];
```

Accurate estimation of the operational loss term is critical to the use of the radar equation. It corrects for the assumption of an ideal system and an ideal environment. The operational loss includes contributions that account for environmental conditions, target positional uncertainty, scan loss, hardware degradation, and many other factors typically totaling ≈ 20 dB or more [1, 2]. For more information on losses that must be included in the radar equation refer to Introduction to "Introduction to Pulse Integration and Fluctuation Loss in Radar" on page 1-644, "Introduction to Scanning and Processing Losses in Pulse Radar" on page 1-656, and "Modeling the Propagation of Radar Signals" on page 1-684 examples.

Set the radial speed of the target to 250 m/s and the target RCS to 1 m². This example assumes that the radar is searching for the same kind of target in all three search sectors. In practice, the radial velocity and the RCS of a typical target of interest can be different for each search sector.

```
% Target radial speed (m/s)
[searchSectorParams.TargetRadialSpeed] = deal(250);

% Target RCS (m²)
[searchSectorParams.TargetRCS] = deal(1);
```

Finally, set the false alarm probability to 1e-6.

```
% Probability of false alarm
[searchSectorParams.FalseAlarmProbability] = deal(1e-6);
```

Consider different values of the PAP allocated to each sector.

```
% Allocated PAP (W*m^2)
pap = [20 40 80];
```

Compute the cumulative probability of detection.

```
% Target range
R = 20e3:0.5e3:80e3;

% Search sector volume (sr)
sectorVolume = zeros(M, 1);

figure
tiledlayout(3, 1, 'TileSpacing', 'tight')

for i = 1:M
    % Volume of the search sector
    sectorVolume(i) = solidangle(searchSectorParams(i).AzimuthLimits, searchSectorParams(i).Eleva

    % Store loss and system noise temperature in the sector parameters
    % struct
    searchSectorParams(i).OperationalLoss = Loss(i);
    searchSectorParams(i).SystemNoiseTemperature = Ts(i);

    % SNR at the sector's range limit
    SNRm = radareqsearchsnr(searchSectorParams(i).RangeLimit, pap, sectorVolume(i), searchSector
        'RCS', searchSectorParams(i).TargetRCS, 'Ts', searchSectorParams(i).SystemNoiseTemperatu

    % Cumulative probability of detection
```

```
    Pc = helperCumulativeDetectionProbability(R, SNRm, searchSectorParams(i));

    % Plot
    ax = nexttile;
    colororder(flipud(ax.ColorOrder));

    hold on
    plot(R*1e-3, Pc, 'LineWidth', 2)
    xline(searchSectorParams(i).RangeLimit*1e-3, ':', 'LineWidth', 1.2, 'Color', 'k', 'Alpha', 1
    yline(0.9, 'LineWidth', 1.2, 'Alpha', 1.0)

    ylabel('P_c')
    title(sprintf('%s Sector', searchSectorParams(i).SectorName))

    grid on

    xlim([20 80])
    ylim([-0.05 1.05])
end

xlabel('Target range (km)')
legendStr = {sprintf('PAP=%.0f (W·m²)', pap(1)), sprintf('PAP=%.0f (W·m²)', pap(2)), sprintf('PAI
legend([legendStr 'Range limit', 'P_c = 0.9'], 'Location', 'southoutside', 'Orientation', 'horizo
```
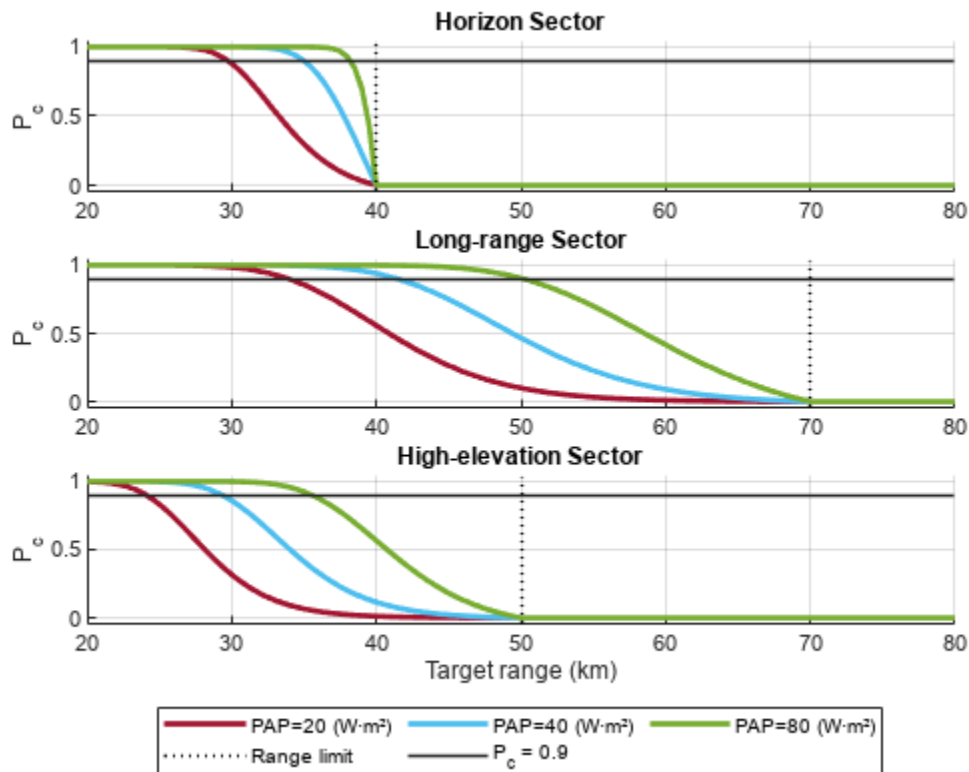

Beyond the sector's range limit a target is undetectable and the cumulative detection probability is zero. Starting at the sector's range limit, as the target approaches the radar, the number of detection

attempts increases with each scan. Thus, the cumulative detection probability increases as the target range decreases. By varying the allocated PAP for each sector, the radar can vary the R_{90} range.

Cumulative Detection Range

The range R_{90} at which the cumulative detection probability equals 0.9, can be found by numerically solving equation $P_c(R) - 0.9 = 0$ with respect to the range variable R.

Compute R_{90} in each sector as the function of the allocated PAP.

```
% The desired cumulative detection probability in the sector
[searchSectorParams.CumulativeDetectionProbability] = deal(0.9);

% Power-aperture product (W·m²)
PAP = 1:2:600;

% Range at which the cumulative detection probability is equal to the
% desired value
R90 = zeros(M, numel(PAP));

str1 = cell(1, M);
str2 = cell(1, M);
for i = 1:M
    for j = 1:numel(PAP)
        % SNR at the sector's range limit
        SNRm = radareqsearchsnr(searchSectorParams(i).RangeLimit, PAP(j), sectorVolume(i), search
            'RCS', searchSectorParams(i).TargetRCS, 'Ts', searchSectorParams(i).SystemNoiseTemper

        % Cumulative detection probability at the range r
        % minus the required value of 0.9
        func = @(r)(helperCumulativeDetectionProbability(r, SNRm, searchSectorParams(i))- searchS

        % Solve for range
        options = optimset('TolX', 0.0001);
        R90(i, j) = fzero(func, [1e-3 1] * searchSectorParams(i).RangeLimit, options);
    end

    str1{i} = sprintf('R_{90} (%s)', searchSectorParams(i).SectorName);
    str2{i} = sprintf('Range limit (%s)', searchSectorParams(i).SectorName);
end

% Plot
figure
hold on
hl = plot(PAP, R90*1e-3, 'LineWidth', 2);
hy = yline([searchSectorParams.RangeLimit]*1e-3, ':', 'LineWidth', 1.5, 'Alpha', 1.0);
[hy.Color] = deal(hl.Color);

xlabel('Power-aperture product (W·m²)')
ylabel('R_{90} (km)')
ylim([0 80])

legend([str1 str2], 'Location', 'southeast')
title('Search Task Quality vs. Resource')
grid on
```

Search Task Quality vs. Resource
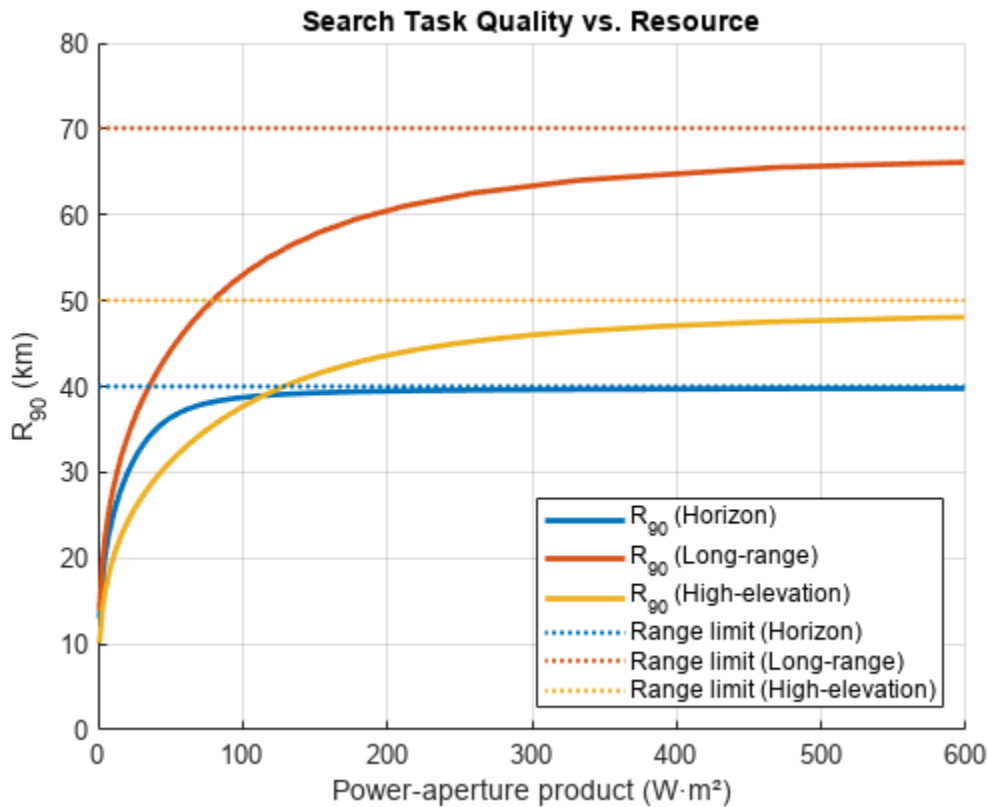
This result shows for each search sector the dependence of the search task quality, R_{90}, on the allocated resource, the PAP. The RRM can allocate less or more PAP to a search sector to decrease or increase the effective value of R_{90}. However, as the allocated PAP increases, R_{90} approaches the asymptotic value bounded by the sector's range limit. If R_{90} is already close to the sector's range limit, allocating more PAP to the sector will not provide any significant increase in the search task quality.

Search Task Utility

The QoS optimization problem seeks to find a resource allocation that jointly optimizes search quality across all sectors. Since the different values of R_{90} can be desirable by different search sectors, simply maximizing the sum of the cumulative detection ranges will not result in a fair resource allocation. In that case search sectors with large R_{90} will contribute significantly more to the objective function dominating search sectors with smaller R_{90}. To account for differences in the desired values of R_{90} across the sectors, the QoS optimization problem first maps the quality metric to a utility. The utility describes the degree of satisfaction with how the search task is performing. The joint utility can be maximized by first weighting and then summing the utilities of the individual tasks.

A simple utility function for a search task can be defined as follows [4]

$$
u_i(R_{90}) = \begin{cases} 0, & R_{90} < R_{t_i} \\[2mm] \dfrac{R_{90} - R_{t_i}}{R_{o_i} - R_{t_i}}, & R_{t_i} \leq R_{90} \leq R_{o_i} \\[2mm] 1, & R_{90} > R_{o_i} \end{cases}
$$

where R_{t_i} and R_{o_i} are respectively the threshold and the objective value for R_{90} in the sector i. The objective determines the desired detection performance for a search sector, while the threshold is a value of the cumulative detection range below which the search task performance is considered unsatisfactory. The intuition behind this utility function is that if the task's quality is above the objective, allocating more resources to this task is wasteful since it will not increase the overall system's satisfaction with the search performance. Similarly, if the quality is below the threshold the utility is zero, and the RRM should allocate resources to the task only when it has enough resources to make the utility positive. The sector's utility is always between 0 and 1 regardless of the R_{90} which makes it convenient to weight and add utilities across the sectors to obtain the joint utility for the surveillance function.

Define sets of R_{90} objective and threshold values for each search sector.

```matlab
% Objective range for each sector
Ro = [38e3 65e3 45e3];

% Threshold range for each sector
Rt = [25e3 45e3 30e3];
```

Create a utility function for each search sector using `helperUtilityFunction` and then plot it against the R_{90} range.

```matlab
% Ranges at which evaluate the utility functions
r90 = 0:1e3:80e3;

figure
tiledlayout(3, 1, 'TileSpacing', 'tight')
for i = 1:M
    % Store threshold and objective values for cumulative detection range
    % in the sector parameters struct
    searchSectorParams(i).ThresholdDetectionRange = Rt(i);
    searchSectorParams(i).ObjectiveDetectionRange = Ro(i);

    % Evaluate and plot the utility at different ranges
    ax = nexttile;
    colors = ax.ColorOrder;

    hold on
    plot(r90*1e-3, helperUtilityFunction(r90, Rt(i), Ro(i)), 'LineWidth', 2, 'Color', colors(i,
    ht = xline(Rt(i)*1e-3, '-.', 'Color', 'k', 'LineWidth', 1.2, 'Alpha', 1.0);
    ho = xline(Ro(i)*1e-3, '--', 'Color', 'k', 'LineWidth', 1.2, 'Alpha', 1.0);

    ylabel('Utility')
    ylim([-0.05 1.05])
    title(sprintf('%s Sector', searchSectorParams(i).SectorName))
    grid on
end
xlabel('R_{90} (km)')
```

```
legend([ht ho], {'Threshold R_{90}', 'Objective R_{90}'}, 'Location', 'southoutside', 'Orientati
```
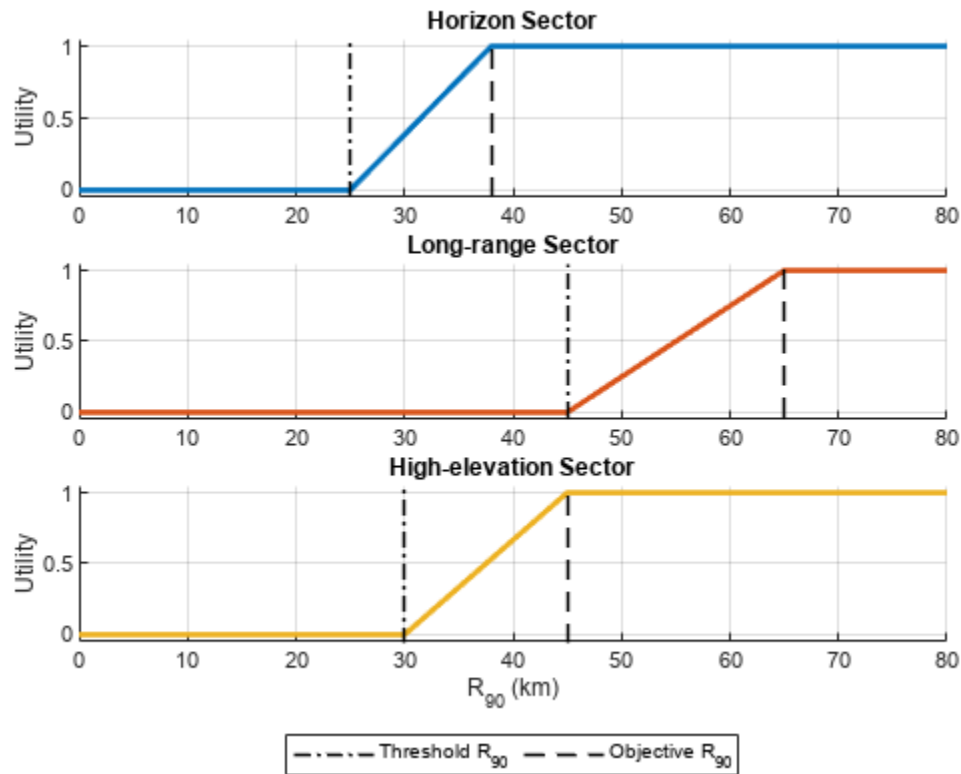

Using these utility functions, the PAP (the resource) can be mapped to the utility.

```
figure
hold on
for i = 1:M
    plot(PAP, helperUtilityFunction(R90(i, :), Rt(i), Ro(i)), 'LineWidth', 2)
end

xlabel('Power-aperture product (W·m²)')
ylabel('Utility')
ylim([-0.05 1.05])
title('Utility vs. Resource')
grid on
legend({searchSectorParams.SectorName}, 'Location', 'best')
```
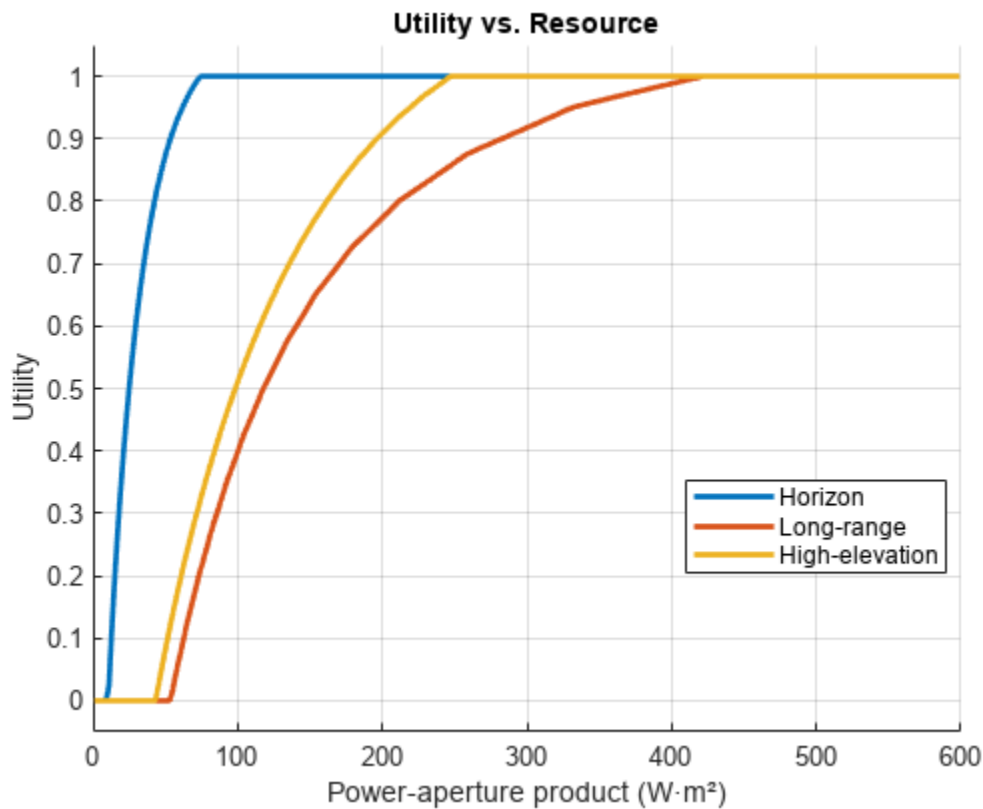
The long-range and the high-elevation sectors require about 50 W·m² of PAP to have a non-zero utility. At the same time, the utility of the horizon sector is maximized at 75 W·m² and allocating more PAP will not improve the objective of the QoS optimization problem. Overall, the horizon and the high-elevation sectors require less PAP than the long-range sector to achieve the same utility.

Utility Under Normal Operational Conditions

Assume that under the normal operational conditions the maximum utility is achieved in each search sector. In this case the RRM does not need to optimize the resource allocation. Each sector uses a nominal amount of resources enough to satisfy the corresponding objective requirement for the cumulative detection range.

Compute the PAP needed to achieve the maximum utility in each sector. First, find the SNR needed to achieve the cumulative probability of detection of 0.9 at the objective value of R_{90}. Then use the `radareqsearchpap` function to solve the power-aperture form of the radar search equation to find the corresponding values of the PAP for the sectors.

```
% Power-aperture product
maxUtilityPAP = zeros(1, M);

for i = 1:M
    % Cumulative detection probability at the range Ro minus the required
    % value
    func = @(snr)(helperCumulativeDetectionProbability(Ro(i), snr, searchSectorParams(i)) ...
        - searchSectorParams(i).CumulativeDetectionProbability);
```

```
    % Solve for SNR
    options = optimset('TolX', 0.0001);
    snr = fzero(func, [-20 30], options);

    % PAP needed to make R90 equal to the objective range Ro
    maxUtilityPAP(i) = radareqsearchpap(searchSectorParams(i).RangeLimit, snr, sectorVolume(i), s
        'RCS', searchSectorParams(i).TargetRCS, 'Ts', searchSectorParams(i).SystemNoiseTemperatu
end
```

```
maxUtilityPAP
```

```
maxUtilityPAP = 1×3

   74.0962   421.6260   247.2633
```

The total amount of the PAP used by the surveillance function under the normal operation conditions is a sum of the maximum utility PAP values used by each sector.

```
% Total PAP needed for search under the normal operational conditions
normalSearchPAP = ceil(sum(maxUtilityPAP))
```

```
normalSearchPAP = 743
```

Use the provided `helperDrawSearchSectors` helper function to visualize the radar search sectors. This function also plots the objective value of R_{90} for each sector - the cumulative detection range corresponding to the maximum utility. Use the bar chart to plot the PAP allocated to the sectors.

```
figure
tiledlayout(1, 6, 'TileSpacing', 'loose')

ax1 = nexttile([1 5]);
helperDrawSearchSectors(ax1, searchSectorParams, true);
title('MPAR Search Sectors Under Normal Operational Conditions')

nexttile
bar(categorical("Normal conditions"), maxUtilityPAP, 'stacked')
set(gca(), 'YAxisLocation', 'right')
ylabel('Power-aperture product (W·m²)')
title('Resource Allocation')
grid on
```

Solution to QoS Optimization Problem

Under the normal operational conditions all search tasks operate at the maximum utility, resulting in the cumulative detection range being equal or exceeding the objective value in each search sector. Since the total amount of resources available to MPAR is finite, activities of other radar functions such as tracking, maintenance, or communication, as well as system errors and failures can impose new constraints on the amount of resources accessible to the surveillance function. These deviations from the normal operational conditions prompt the RRM to recompute the resource allocations to the search tasks such that the new constraints are met, and the weighted utility is again maximized across all sectors.

The QoS optimization problem can be solved numerically using the `fmincon` function from the Optimization Toolbox™.

Set the sector priority weights such that the horizon sector has the highest priority, and the long-range sector has the lowest.

```
% Sector priority weights
w = [0.55 0.15 0.3];
```

Normalize the sector weights such that they add up to 1.

```
w = w/sum(w);
```

Assume that the surveillance function has access to only 50% of the PAP used for search under the normal operational conditions.

```
searchPAP = 0.5*normalSearchPAP

searchPAP = 371.5000
```

Set up the objective and the constraint functions.

```
% Objective function
fobj = @(x)helperQoSObjective(x, searchSectorParams, w);

% Constraint
fcon = @(x)helperQoSConstraint(x, searchPAP);
```

Set the lower and the upper bound on the QoS solution.

```
% Lower bound on the QoS solution
LB = 1e-6*ones(M, 1);

% Upper bound on the QoS solution
UB = maxUtilityPAP;
```

As the initial point for the optimization distribute the available PAP according to the sector priority weights.

```
% Starting points
startPAP = w*searchPAP;
```

Show the summary of the search sector parameters used to set up the QoS optimization problem.

```
helperSectorParams2Table(searchSectorParams)
```

```
ans=12×4 table
        Parameter                              Horizon         Long-range       High-elevation
    _____        _____    _____      _____

    {'AzimuthLimits'                  }       {2x1 double  }  {2x1 double  }    {2x1 double  }
    {'ElevationLimits'                }       {2x1 double  }  {2x1 double  }    {2x1 double  }
    {'RangeLimit'                     }       {[      40000]} {[      70000]}   {[      50000]}
    {'FrameTime'                      }       {[     0.5000]} {[          6]}   {[          2]}
    {'OperationalLoss'                }       {[         22]} {[         19]}   {[         24]}
    {'SystemNoiseTemperature'         }       {[        913]} {[        913]}   {[        913]}
    {'CumulativeDetectionProbability' }       {[     0.9000]} {[     0.9000]}   {[     0.9000]}
    {'FalseAlarmProbability'          }       {[1.0000e-06]}  {[1.0000e-06]}    {[1.0000e-06]}
    {'TargetRCS'                      }       {[          1]} {[          1]}   {[          1]}
    {'TargetRadialSpeed'              }       {[        250]} {[        250]}   {[        250]}
    {'ThresholdDetectionRange'        }       {[      25000]} {[      45000]}   {[      30000]}
    {'ObjectiveDetectionRange'        }       {[      38000]} {[      65000]}   {[      45000]}
```

Use `fmincon` to find an optimal resource allocation.

```
options = optimoptions('fmincon', 'Display', 'off');
papAllocation = fmincon(fobj, startPAP, [], [], [], [], LB, UB, fcon, options)

papAllocation = 1×3

   74.0855  114.0675  183.3469
```

Verify that this allocation satisfies the constraint and does not exceed the available PAP amount.

```
sum(papAllocation)

ans = 371.5000
```

Given the found PAP allocations for each sector, compute R_{90} and the corresponding value of the utility.

```
[~, optimalR90, utility] = helperQoSObjective(papAllocation, searchSectorParams, w)

optimalR90 = 1×3
10⁴ ×

    3.7999    5.4588    4.3005


utility = 1×3

    0.9999    0.4794    0.8670
```

For the horizon sector, plot the found optimal resource allocation on the PAP vs. R_{90} curve and the corresponding utility value on the utility vs. R_{90} curve.

```
helperPlotR90AndUtility(PAP, R90(1, :), searchSectorParams(1), papAllocation(1), optimalR90(1), r
sgtitle({sprintf('%s Sector', searchSectorParams(1).SectorName), '(High Priority)'})
```
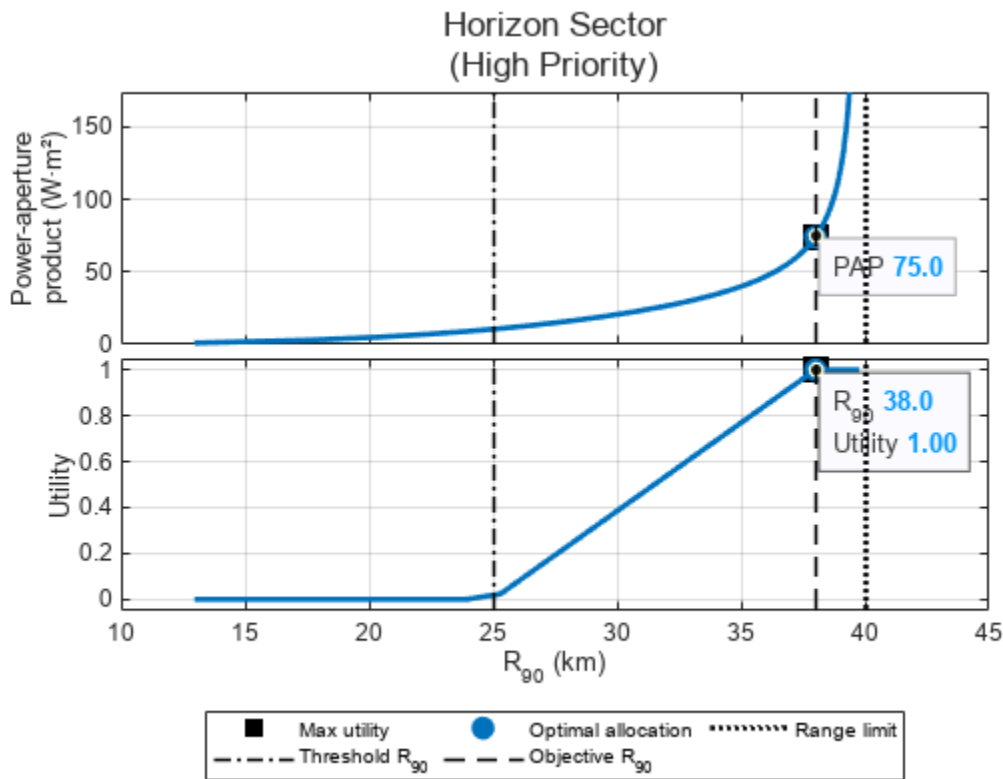

These plots visualize the two-step transformation performed inside the QoS optimization. The first step computes R_{90} from the PAP, while the second step computes the utility from R_{90}. To follow this

transformation, start on the y-axis of the top subplot and choose a PAP value. Then find the corresponding point on the PAP vs. R_{90} curve. Mark the R_{90} value of this point on the x-axis. Using this R_{90} value find a point on the utility curve in the bottom subplot. Finally, find the corresponding utility on the y-axis of the bottom subplot. The same set of steps can be traced in reverse to go from the utility to the PAP.

The surveillance function in this example is set up such that the horizon sector has the highest priority. It also requires much smaller PAP to achieve the objective R_{90} compared to the other two sectors. The QoS optimization allocates 74.1 W·m² of PAP to the horizon sector. This is equal to the amount allocated to this sector under the normal operational conditions, resulting in the horizon sector utility approaching 1.

Plot the optimization results for the long-range and the high-elevation sectors.

```
helperPlotR90AndUtility(PAP, R90(2, :), searchSectorParams(2), papAllocation(2), optimalR90(2), r
sgtitle({sprintf('%s Sector', searchSectorParams(2).SectorName), '(Low Priority)'})
```
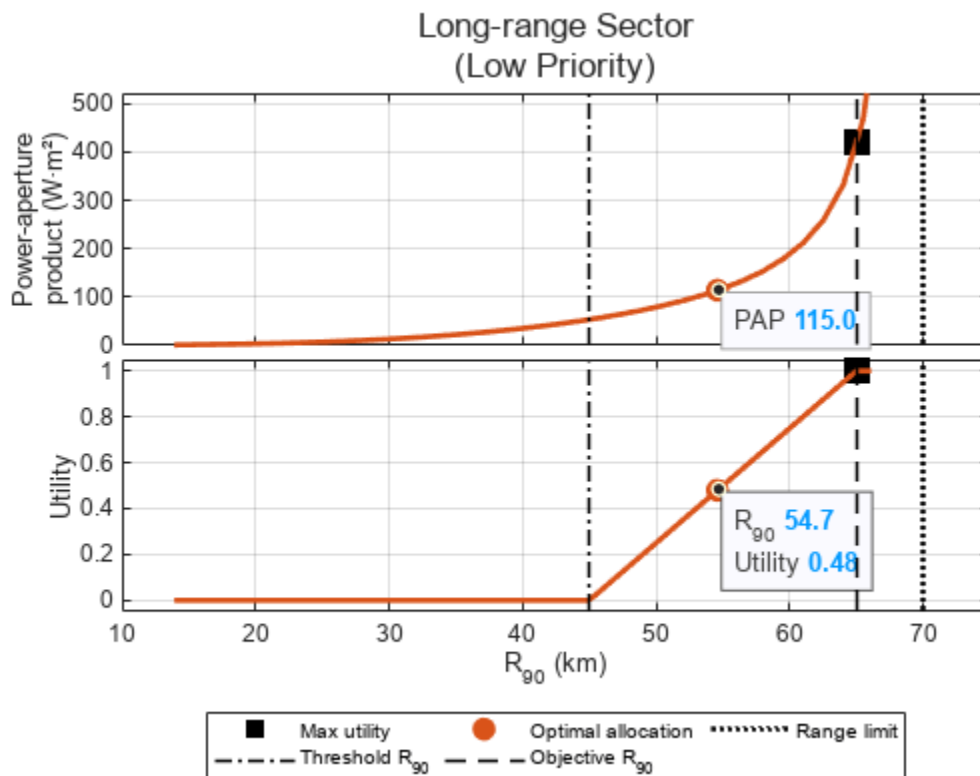


```
helperPlotR90AndUtility(PAP, R90(3, :), searchSectorParams(3), papAllocation(3), optimalR90(3), r

sgtitle({sprintf('%s Sector', searchSectorParams(3).SectorName), '(Medium Priority)'})
```

High-elevation Sector
(Medium Priority)
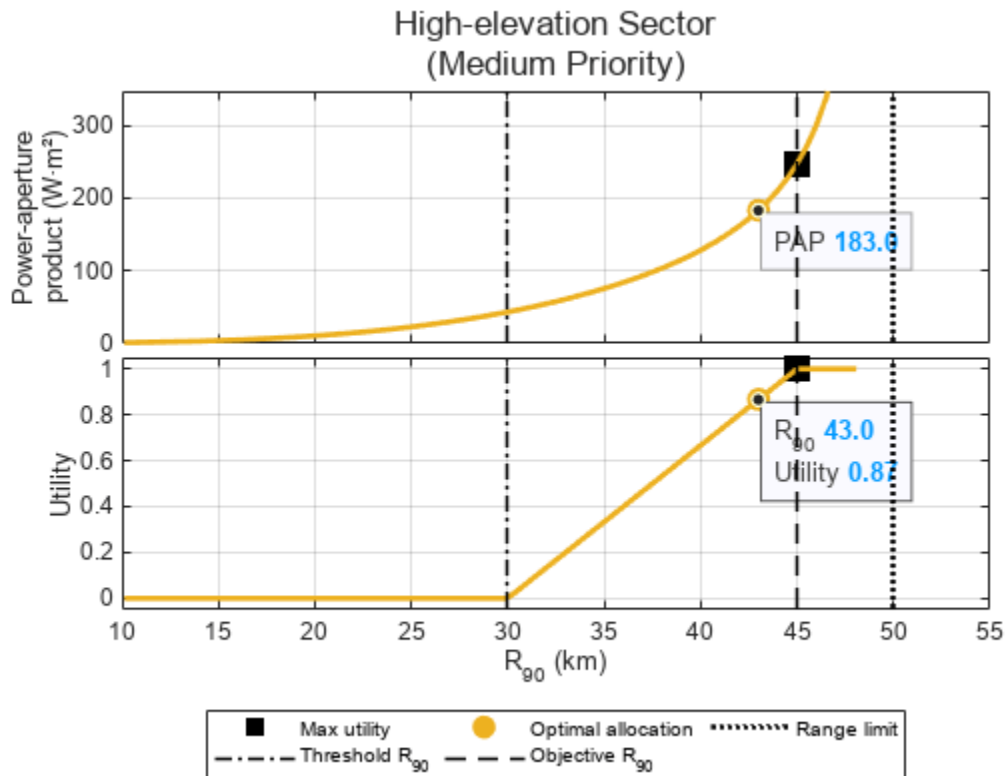
The QoS optimization allocates 114.2 W·m² and 183.25 W·m² to the long-range and the high-elevation sectors respectively. This allocation results in R_{90} of about 55 km in the long-range sector, which corresponds to the utility of 0.5. The R_{90} range in the high-elevation sector is equal to 43 km and the utility is about 0.87. The high-elevation sector is favored by the QoS optimization over the long-range sector because the corresponding sector priority weight is higher.

Use the provided `helperDrawSearchSectors` helper function to visualize the radar search sectors with the optimal value of R_{90} obtained from the QoS optimization. Then use the bar chart to plot the computed optimal PAP allocation and compare it to the PAP allocation under the normal operational conditions.

```
figure
tiledlayout(1, 6, 'TileSpacing', 'compact')

ax1 = nexttile([1 5]);
helperDrawSearchSectors(ax1, searchSectorParams, false, optimalR90);
title('MPAR Search Sectors When 50% of PAP Is Available')

nexttile
ax2 = bar(categorical(["Normal conditions", "50% available"]), [maxUtilityPAP; papAllocation], 's
set(gca(), 'YAxisLocation', 'right')
ylabel('Power-aperture product (W·m²)')
title('Resource Allocation')
grid on
```
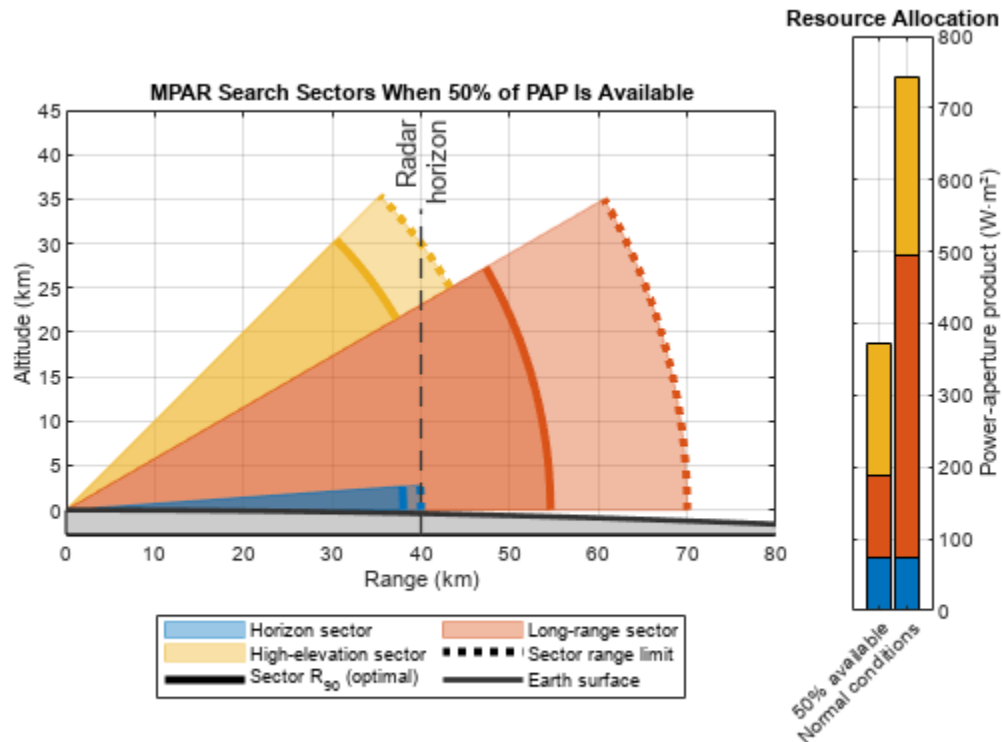
Conclusions

This example develops a resource management scheme for the surveillance function of an MPAR with multiple search sectors. The example starts by defining parameters of three search sectors: horizon, long-range, and high-elevation. It then introduces the QoS optimization problem as a resource allocation scheme. The cumulative detection range is used as a performance measure describing the quality of a search task and the utility functions based on the threshold and objective values are used to map the quality into the utility space. Finally, the example shows how to solve the QoS optimization problem numerically to obtain the optimal allocation of PAP to the search sectors.

References

1 James A. Scheer, William L. Melvin, *Principles of Modern Radar: Radar Applications, Volume 3*. United Kingdom: Institution of Engineering and Technology, 2013.

2 Barton, David Knox. *Radar equations for modern radar*. Artech House, 2013.

3 Charlish, Alexander, Folker Hoffmann, Christoph Degen, and Isabel Schlangen. "The Development From Adaptive to Cognitive Radar Resource Management." *IEEE Aerospace and Electronic Systems Magazine* 35, no. 6 (June 1, 2020): 8–19.

4 Hoffmann, Folker, and Alexander Charlish. "A resource allocation model for the radar search function." In *2014 International Radar Conference*, pp. 1-6. IEEE, 2014.

Supporting Functions

helperSearchSector

Creates a struct with placeholders for search sector parameters.

```
type('helperSearchSector.m')
```

```
function searchSector = helperSearchSector()
searchSector = struct('SectorName', [], 'AzimuthLimits', [], 'ElevationLimits', [], 'RangeLimit'
    'FrameTime', [], 'OperationalLoss', [], 'SystemNoiseTemperature', [], ...
    'CumulativeDetectionProbability', [], 'FalseAlarmProbability', [], 'TargetRCS', [], ...
    'TargetRadialSpeed', [], 'ThresholdDetectionRange', [], 'ObjectiveDetectionRange', []);
end
```

helperCumulativeDetectionProbability

Compute cumulative detection probability given range, SNR at the sector's range limit, and sector parameters.

```
type('helperCumulativeDetectionProbability.m')
```

```
function Pc = helperCumulativeDetectionProbability(R, SNRm, searchSector)
% The expectation and the dependency of the number of scans N on the target
% range make the exact computation of the cumulative detection probability
% difficult. Instead, it could be conveniently approximated by linearly
% interpolating the lower bound expression
%
%   P_c(R) >= 0, when R <= Rm && R > Rm - dR
%   P_c(R) >= 1 - prod(1 - P_d(Rm - n*dR)), when R <= Rm - n*dR and n = 1:N
%
% computed at ranges Rm-k*dR, k=0,1,...

% Normalize range to the target popup range
R = R/searchSector.RangeLimit;

% Normalized target closure range
dR = searchSector.FrameTime*searchSector.TargetRadialSpeed/searchSector.RangeLimit;

Pfa = searchSector.FalseAlarmProbability;

SNRm = db2pow(SNRm(:).');
M = numel(SNRm);
Pc = zeros(numel(R), M);

idxs = R < 1;
r = R(idxs);
N = numel(r);
p = zeros(N, M);

for i = 1:N
    range = (1:-dR:max(r(i)-dR, 0)).';

    if dR > r(i) && range(end) ~= 0
        range = [range; 0];
    end

    Pd = zeros(numel(range), M);

    SNR = (range(2:end).^(-4))*SNRm;
    Pd(2:end,:) = 1 - cumprod(1 - Pfa.^(1./(1+SNR)), 1);

    p(i, :) = interp1(range, Pd, r(i));
end
```

```
Pc(idxs, :) = p;

end
```

helperUtilityFunction

Utility function.

```
type('helperUtilityFunction.m')
```

```
function u = helperUtilityFunction(r, Rt, Ro)
u = zeros(size(r));

idxo = r > Ro;
u(idxo) = 1;

idxt = r < Rt;
u(idxt) = 0;

idxs = ~idxo & ~idxt;
u(idxs) = (r(idxs)-Rt)/(Ro-Rt);
end
```

helperWeightedUtility

Function that returns weighted utility given PAP allocations, sectors weights, and sector parameters.

```
type('helperQoSObjective.m')
```

```
function [val, R90, utility] = helperQoSObjective(papAllocation, searchSectorParams, weights)
val = 0;
M = numel(searchSectorParams);

R90 = zeros(1, M);
utility = zeros(1, M);

for i = 1:M
    if papAllocation(i) > 0
        sectorVolume = solidangle(searchSectorParams(i).AzimuthLimits, searchSectorParams(i).Elev
        snr = radareqsearchsnr(searchSectorParams(i).RangeLimit, papAllocation(i), sectorVolume,
            'RCS', searchSectorParams(i).TargetRCS, 'Ts', searchSectorParams(i).SystemNoiseTemper

        func = @(r)(helperCumulativeDetectionProbability(r, snr, searchSectorParams(i)) - searchS

        options = optimset('TolX', 0.00001);
        R90(i) = fzero(func, [1e-6 1] * searchSectorParams(i).RangeLimit, options);
        utility(i) = helperUtilityFunction(R90(i), searchSectorParams(i).ThresholdDetectionRange,

        val = val - weights(i)*utility(i);
    end
end
end
```

helperConstraintPAP

Function that defines the constraint on the PAP.

```
type('helperQoSConstraint.m')
```

```
function [c, ceq] = helperQoSConstraint(pap, total)
c = [];
ceq = sum(pap) - total;
end
```

Design and Simulate an FMCW Long-Range Radar (LRR)

This example shows how to generate measurement-level radar detections using a `radarDataGenerator` created from a radar design exported from the Radar Designer app. The radar will be able to detect vehicles at ranges greater than 300 meters with sufficient resolution to resolve targets in adjacent lanes at this range. You will also learn how to generate signal-level radar detections using an equivalent `radarTransceiver` created from the `radarDataGenerator`. The measurement- and signal-level detections are compared to the system-level design analysis provided by the Radar Designer app.

Design System Parameters and Predict Performance Metrics

Design a long-range automotive radar based on the transmit beamforming (TXBF) design in [1 on page 1-902] using the Radar Designer app. To open this design in the app, run the command:

```
radarDesigner('LRR_RadarDesignerSession.mat')
```

The system parameters under the **Main** section in the left-hand panel of the radar designer app come from Table 3 in [1 on page 1-902] and Section 8.7 in [2 on page 1-902]. In particular, the 26 us sweep repetition interval (or PRI in the app) is calculated as the sum of the FMCW waveform sweep time (17 us), the FMCW transceiver's idle time (4 us) and ADC start time (5 us). The 0.65 duty cycle is the ratio of sweep time over the sweep repetition interval.

The **Antenna and scanning** parameters are found in Table 1 in [3 on page 1-902]. Since arrays are used on both transmit (Tx) and receive (Rx), total transmit and receive gains are calculated as the sum of the antenna element directivity (12 dBi) and the array factor (11 dB for the 12 element Tx array, 12 dB for the 16 element Rx array).

The **Detection and Tracking** parameters are found in Table 3 in [1 on page 1-902] and a typical false alarm rate of 1e-6 is assumed.

A 10 dBsm, non-fluctuating radar cross-section (RCS) target positioned 20 cm above the ground is used for Radar Designer app system-level performance analysis and metrics. This target is representative of a mid-sized sedan for an automotive radar. The target must be detected with a probability of 0.9. The environment losses considered are limited to free space propagation loss.
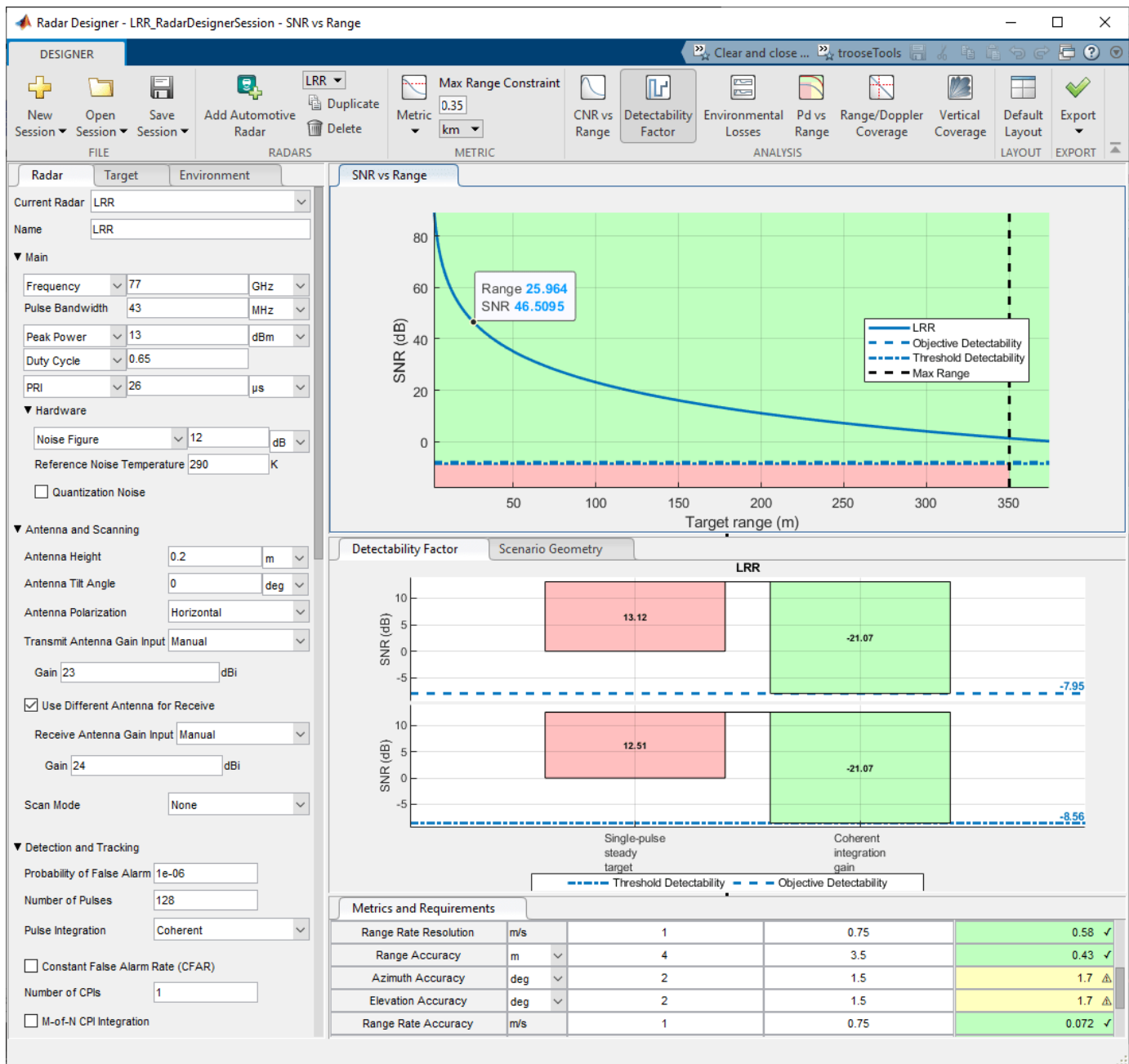
The SNR vs. range plot shows the SNR of a single sweep that's available before any signal processing in the receiver. For a target at approximately 25.4 meters, the single sweep SNR is predicted to be around 47 dB. In addition, the detectability factor charts indicate that the 128 sweeps (or "pulses" on the app) yield a coherent integration gain of 21 dB. Run the range analysis script exported from the app (the plot has been commented out). The script provides an API for the system parameters.

Model the FMCW waveform transceiver's idle and ADC start time using a sweep repetition interval and duty cycle.

```
LRRRangeAnalysis;
lrrSysParam = radar
```

```
lrrSysParam = struct with fields:
                    Name: 'LRR'
        AzimuthBeamwidth: [2x1 double]
              CustomLoss: 0
      ElevationBeamwidth: [2x1 double]
               Frequency: 7.7000e+10
                    Gain: [23 24]
                  Height: 0.2000
        NoiseTemperature: 4.5962e+03
                 NumCPIs: 1
       NumCoherentPulses: 128
    NumNonCoherentPulses: 1
                     PRF: 38462
               PeakPower: 0.0200
                     Pfa: 1.0000e-06
            Polarization: 'H'
              Pulsewidth: 1.6900e-05
               TiltAngle: 0
              Wavelength: 0.0039
       SystemTemperature: 4.5962e+03
```

```
tgtRg = 26; % Target range (m)
singleSweepSNR = interp1(target.ranges,availableSNR,tgtRg)
```

```
singleSweepSNR = 46.4858
```

```
coherentIntGain = pow2db(lrrSysParam.NumCoherentPulses)
```

```
coherentIntGain = 21.0721
```

```
predictedSNR = singleSweepSNR + coherentIntGain
```

```
predictedSNR = 67.5579
```

The table at the bottom of the Radar Designer app shows that the predicted performance metrics of the design (right column) meet all threshold requirements (left column) and all objective requirements (middle column) except for the azimuth and elevation accuracy. Run the metrics report script that was also exported from the app to enable comparisons in the following sections.

```
LRRExportedMetricsReport;
predictedLRRMetrics = metricsTable;
predictedLRRMetrics([2:3 12:15],:) = [] % Remove less relevant metrics
```

```
predictedLRRMetrics=9×4 table
```

	Units	Threshold	Objective	LRR
Probability of Detection	""	0.8	0.9	1
Unambiguous Range	"m"	250	350	3897.3
Range Resolution	"m"	4	3.75	3.486
First Blind Speed	"m/s"	64	70	74.873
Range Rate Resolution	"m/s"	1	0.75	0.58495
Range Accuracy	"m"	4	3.5	0.4279
Azimuth Accuracy	"deg"	2	1.5	1.7128
Elevation Accuracy	"deg"	2	1.5	1.7128
Range Rate Accuracy	"m/s"	1	0.75	0.071803

Configure a Measurement-Level Model of the Radar

Use the design parameters and metrics exported from the Radar Designer app to configure a `radarDataGenerator` which is a measurement-level model that you can use to simulate detections in different scenarios. The model abstracts the detailed signal processing chain and produces detections who statistics are governed by the desired probabilities of detection and false alarm evaluated for a reference RCS at a reference range corresponding to the detectability threshold.

```
Fc = lrrSysParam.Frequency;
detThreshold = Dx(2); % Dx is a variable created by the LRRRangeAnalysis script
maxSpd = predictedLRRMetrics{'First Blind Speed',end}/2; % The range-rate limits are [-vb/2, +vb/
measLvlLRR = radarDataGenerator(1, 'No scanning', ...
    'DetectionProbability', predictedLRRMetrics{'Probability of Detection',3}, ... % Objective Po
    'FalseAlarmRate', lrrSysParam.Pfa, ...
    'ReferenceRCS', target.RCS, ...
    'ReferenceRange', interp1(availableSNR,target.ranges,detThreshold), ... % Range at which the
    'MountingLocation', [3.7 0 lrrSysParam.Height], ...
    'CenterFrequency', Fc, ...
    'Bandwidth', rangeres2bw(predictedLRRMetrics{'Range Resolution',end}), ...
    'RangeResolution', predictedLRRMetrics{'Range Resolution',end}, ...
    'HasRangeRate', true, ...
    'RangeRateResolution', predictedLRRMetrics{'Range Rate Resolution',end}, ...
    'RangeLimits', [0 predictedLRRMetrics{'Unambiguous Range',end}], ...
    'RangeRateLimits', [-1 1]*maxSpd, ...
    'HasRangeAmbiguities', true, ...
    'MaxUnambiguousRange', predictedLRRMetrics{'Unambiguous Range',end}, ...
    'HasRangeRateAmbiguities', true, ...
    'MaxUnambiguousRadialSpeed', maxSpd, ...
    'TargetReportFormat', 'Detections', ...
    'DetectionCoordinates', 'Sensor spherical');

rng default % Set random seed for reproducible results
```

A few parameters required by the `radarDataGenerator` are not exported by the app. These values are taken from Table 1 and Table 3 in [1 on page 1-902] and Table 1 in [3 on page 1-902] .

- 4-element series-fed patch azimuth field of view: 120 deg
- 4-element series-fed patch elevation field of view: 60 deg
- Required azimuth resolution: 1.4 deg
- Bandwidth: 43 MHz

```
measLvlLRR.FieldOfView = [120 60];
measLvlLRR.AzimuthResolution = 1.4;
```

The measurement-level radar is now configured to match the design parameters analyzed in the Radar Designer app.

```
disp(measLvlLRR)

  radarDataGenerator with properties:

              SensorIndex: 1
               UpdateRate: 1
            DetectionMode: 'Monostatic'
                 ScanMode: 'No scanning'
       InterferenceInputPort: 0
```

```
        EmissionsInputPort: 0

          MountingLocation: [3.7000 0 0.2000]
            MountingAngles: [0 0 0]

               FieldOfView: [120 60]
               RangeLimits: [0 3.8973e+03]
           RangeRateLimits: [-37.4366 37.4366]
        MaxUnambiguousRange: 3.8973e+03
  MaxUnambiguousRadialSpeed: 37.4366

        DetectionProbability: 0.9000
              FalseAlarmRate: 1.0000e-06
              ReferenceRange: 596.8580

          TargetReportFormat: 'Detections'

  Use get to show all properties
```

Generate Measurements and Verify SNR Levels

Use `targetPoses` (Automated Driving Toolbox) to generate detections from the straight road scenario exported using the Driving Scenario Designer (Automated Driving Toolbox) (DSD) app.

```
% Load the scenario and ego vehicle using the script exported from the DSD app
[scenario, egoVehicle] = helperStraightRoadDSDExport();
measLvlLRR.Profiles = actorProfiles(scenario); % Register the target profiles with the radar

% Get the target poses in the ego's body frame
tposes = targetPoses(egoVehicle);
time = scenario.SimulationTime;
dets = measLvlLRR(tposes,time);
```

Show the scenario ground truth and detections in a bird's-eye plot.

```
[bep,pltrs] = helperSetupDisplay(egoVehicle,measLvlLRR);
axis(bep.Parent,[-3 50 30*[-1 1]]);
pltrs.DetectionPlotters(1).Plotter(dets);
```
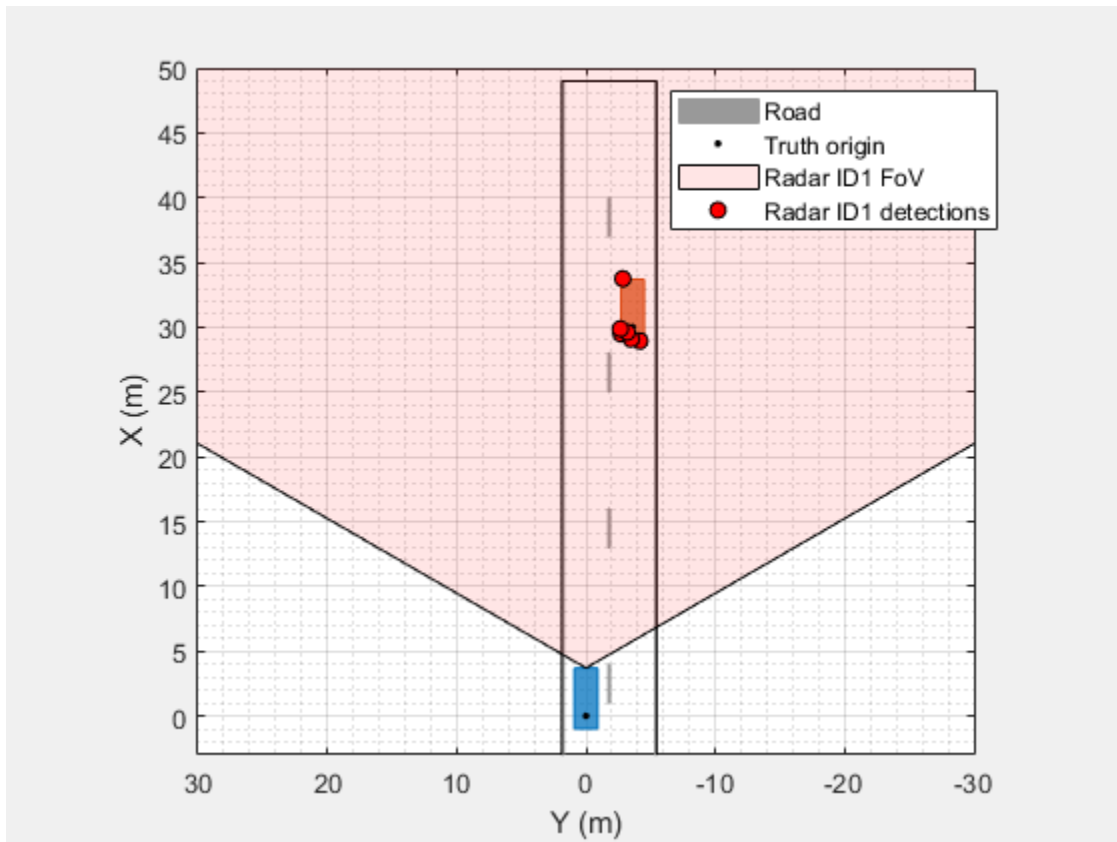
The generated detections lie along the outline of the target vehicle. Find the maximum signal-to-noise ratio (SNR) of the generated detections.

```
[measLvlSNR,iMax] = max(cellfun(@(d)d.ObjectAttributes{1}.SNR,dets(:)'));
disp(measLvlSNR)
```

```
67.2708
```

Compare the SNR of the detections generated by the measurement-level radar to the system-level radar in the Radar Designer app. The SNR exported by the Radar Designer app is the SNR for a single pulse (or sweep) of the radar. To compare the SNR predicted by the app to the measurement-level radar, the coherent processing gain from all pulses in the processing interval must be included.

```
% Use the exact range of the measurement. The measurement vector for this
% radar configuration is [az;rg;rr], so select the second element.
tgtRg = dets{iMax}.Measurement(2);
singleSweepSNR = interp1(target.ranges,availableSNR,tgtRg);
predictedSNR = singleSweepSNR + coherentIntGain
```

```
predictedSNR = 67.8409
```

```
ax = axes(figure);
X = categorical({'Predicted (Analytical Model)','Simulated (Meas-Level Model)'});
bar(ax,X,[predictedSNR measLvlSNR],'FaceColor', [0.75 1 0.75]);
xtickangle(ax,15)
grid(ax,'on')
ylabel(ax,'SNR (dB)')
title(ax,['SNR Verification: ' num2str(target.RCS) 'dBsm Target @ ' num2str(tgtRg,'%.1f') ' mete
```

SNR Verification: 10dBsm Target @ 25.6 meters
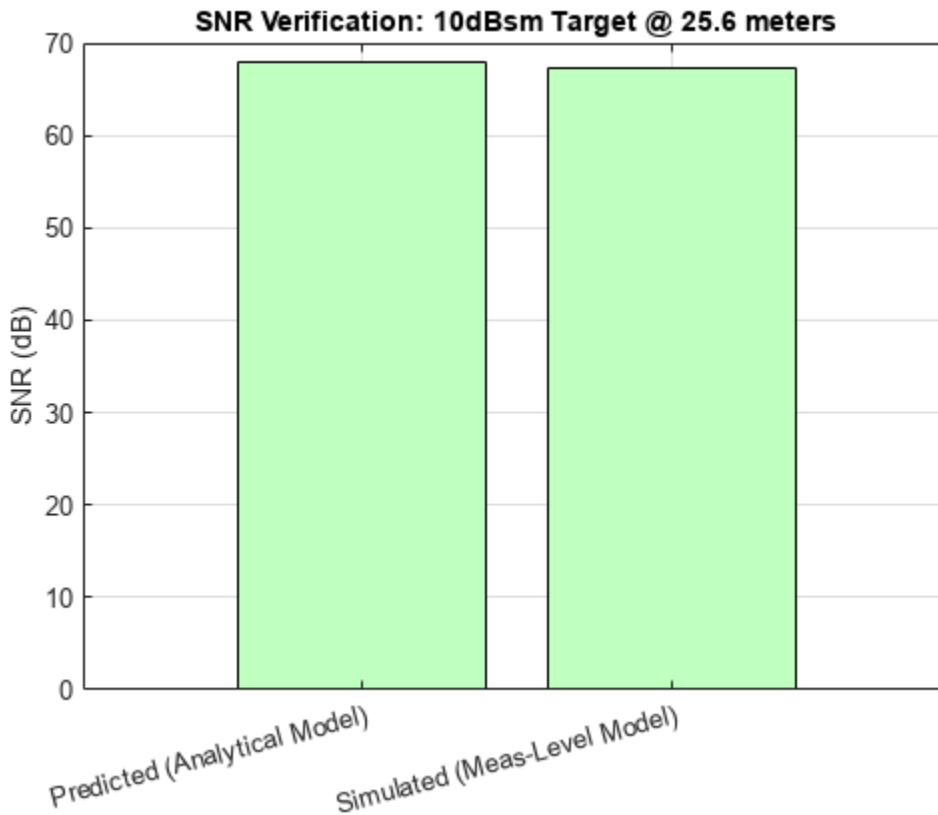
```
drawnow
```

Create an Equivalent Radar Transceiver Model

Reuse the `radarDataGenerator` model to create an equivalent `radarTransceiver` that can simulate I/Q signals.

```
% Select the PRF based on the max radial speed constraint by setting the
% max unambiguous range accordingly
release(measLvlLRR);
lambda = lrrSysParam.Wavelength;
prfMaxSpeed = 4*speed2dop(predictedLRRMetrics{'First Blind Speed',end},lambda);
measLvlLRR.MaxUnambiguousRange = time2range(1/prfMaxSpeed,lambda*Fc);
xcvrLRR = radarTransceiver(measLvlLRR);
measLvlLRR.MaxUnambiguousRange = predictedLRRMetrics{'Unambiguous Range',end}; % Restore max rang
disp(xcvrLRR)

  radarTransceiver with properties:

                Waveform: [1x1 phased.RectangularWaveform]
             Transmitter: [1x1 phased.Transmitter]
         TransmitAntenna: [1x1 phased.Radiator]
          ReceiveAntenna: [1x1 phased.Collector]
                Receiver: [1x1 phased.ReceiverPreamp]
       MechanicalScanMode: 'None'
       ElectronicScanMode: 'None'
         MountingLocation: [3.7000 0 0.2000]
           MountingAngles: [0 0 0]
```

```
       NumRepetitionsSource: 'Property'
             NumRepetitions: 256
          RangeLimitsSource: 'Property'
                RangeLimits: [0 3.8973e+03]
            RangeOutputPort: false
             TimeOutputPort: false
```

Model elaboration: antenna array design

The equivalent `radarTransceiver` has only a single element for transmit and for receive. A linear array is needed to measure azimuth. The number of array elements required to achieve the desired azimuth resolution is given by:

```
numElmnts = beamwidth2ap(measLvlLRR.AzimuthResolution,lambda,0.8859)*2/lambda % For an untapered
```

```
numElmnts = 72.5119
```

Use `helperDesignArray` on page 1-903 to create the array configuration documented in [1 on page 1-902] to realize the large array size required by the azimuth resolution.

```
[txArray, rxArray, arrayMetrics, txPos, rxPos, ant] = helperDesignArray(lambda,Fc);
```

Attach the designed transmit and receive antenna arrays to the `radarTransceiver`.

```
xcvrLRR.ReceiveAntenna.Sensor = rxArray;
xcvrLRR.TransmitAntenna.Sensor = txArray;
```

Set the electronic scan mode to 'Custom' and enable the weights input port on the transmit antenna to enable beamforming on transmit.

```
xcvrLRR.ElectronicScanMode = 'Custom';
xcvrLRR.TransmitAntenna.WeightsInputPort = true;
```

Model elaboration: waveform design

The `radarTransceiver` uses a rectangular waveform by default, but FMCW waveforms are more common in automotive radars. Attach an FMCW waveform to the `radarTransceiver` with its waveform parameters set to achieve the same range resolution as was used by the default rectangular waveform designed for the equivalent radar.

```
Fs = xcvrLRR.Waveform.SampleRate;
numSmps = ceil(Fs*lrrSysParam.Pulsewidth); % Must be a positive integer
wfm = phased.FMCWWaveform('SampleRate',Fs,'SweepTime',numSmps/Fs,'SweepBandwidth',Fs,'SweepInterv
```

```
wfm =
  phased.FMCWWaveform with properties:

         SampleRate: 4.3000e+07
          SweepTime: 1.6907e-05
     SweepBandwidth: 4.3000e+07
     SweepDirection: 'Up'
      SweepInterval: 'Symmetric'
       OutputFormat: 'Sweeps'
          NumSweeps: 1
```

```
xcvrLRR.Waveform = wfm;
```

The sweep time computed for the `radarTransceiver` matches the sweep time of 17 us in Table 3 of [1 on page 1-902].

Model elaboration: receiver noise figure and transmitted peak power

Adjust the transmitter's peak power level to account for the beamforming gains from the transmit and receive arrays and set the noise figure to match the value used by the design in the Radar Designer app. The power level computed for the transmitter agrees with the value of 12 dBm used in the Radar Designer app.

```
Lnf = pow2db(radar.SystemTemperature/xcvrLRR.Receiver.ReferenceTemperature) % Noise figure
```

```
Lnf = 12.0000
```

```
xcvrLRR.Receiver.NoiseFigure = Lnf;
```

The power level computed for the transmitter agrees with the value of 13 dBm used in the Radar Designer app. A difference of 1 dB is explained by the rounded values taken from [1]-[3] that were entered in the app.

```
numReps = xcvrLRR.NumRepetitions;
xcvrLRR.NumRepetitions = lrrSysParam.NumCoherentPulses; % Use the number of repetitions specified
pwr = (numReps/xcvrLRR.NumRepetitions)*xcvrLRR.Transmitter.PeakPower; % Keep the total power the
pwrdBm = pow2db(pwr*1e3)-(arrayMetrics.TxGain+arrayMetrics.RxGain)-Lnf % dBm
```

```
pwrdBm = 14.1514
```

```
xcvrLRR.Transmitter.PeakPower = db2pow(pwrdBm)*1e-3; % Watts
```

Simulate Radar Transceiver I/Q Signals

Create radar propagation paths from the target poses in the scenario. The extended target is modeled using a set of point targets located along the target's extent. The spacing of the point targets is determined by the azimuth and range resolution of the radar.

```
paths = poses2paths(tposes,xcvrLRR,scenario,measLvlLRR,bep);
```
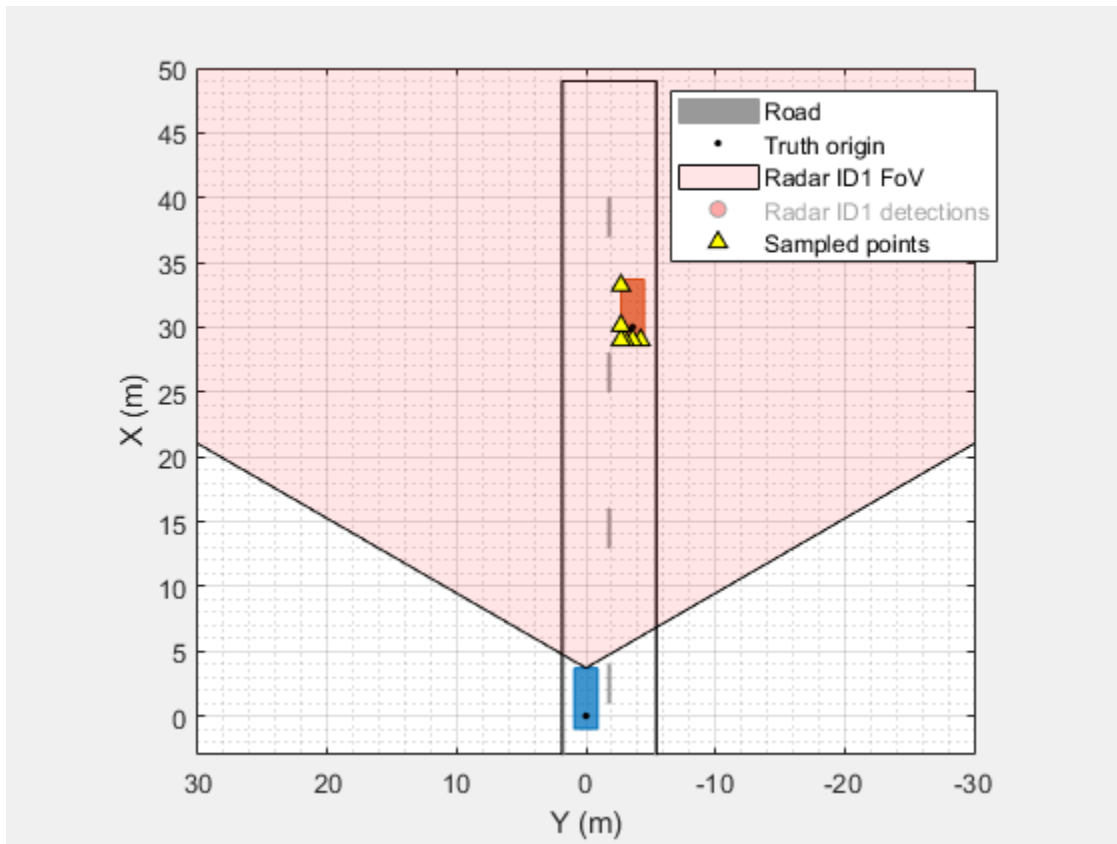
Use `radarTransceiver` to generate baseband I/Q samples at the output of the radar receiver from the propagation paths for the target. The radar's field of view is electronically scanned by beamforming on transmit and stepping the transmit angle by the receive array's beamwidth.

```
azfov = measLvlLRR.FieldOfView(1);
numBeams = ceil(azfov/arrayMetrics.RxBeamwidth);
numBeams = floor(numBeams/2)*2+1; % Force the number of beams to be odd to keep a beam centered o
azgrid = azfov*linspace(-0.5,0.5,numBeams);
```

Use `phased.SteeringVector` to compute steering vectors for the transmit array for each scan angle.

```
txSV = phased.SteeringVector( ...
    'SensorArray',xcvrLRR.TransmitAntenna.Sensor, ...
    'PropagationSpeed',xcvrLRR.TransmitAntenna.PropagationSpeed);
txSteer = txSV(xcvrLRR.TransmitAntenna.OperatingFrequency,azgrid);
```

Use `radarTransceiver` to collect baseband I/Q data cubes for each scan angle.

```
numFT = xcvrLRR.Waveform.SampleRate*xcvrLRR.Waveform.SweepTime; % Number of fast-time samples (ra
numST = xcvrLRR.NumRepetitions; % Number of slow-time samples (Doppler bins)
numRxElmnts = getNumElements(xcvrLRR.ReceiveAntenna.Sensor); % Number of receive array elements
numFrames = numel(azgrid); % Number of frames (scan angles)

% Allocate array to collect data cubes
Xframes = NaN(numFT,numRxElmnts,numST,numFrames);
```

```
% Advance the simulation time by the duration of each frame
tFrame = xcvrLRR.Waveform.SweepTime*xcvrLRR.NumRepetitions;
for iFrm = 1:numFrames
    % Current scan angle
    txAng = [azgrid(iFrm);0];

    % Collect data cube at current scan angle
    Xcube = xcvrLRR(paths,time+(iFrm-1)*tFrame,conj(txSteer(:,iFrm)));
    Xframes(:,:,:,iFrm) = Xcube;
end
```

Show the spectrogram of the collected data cube for a scan angle in the direction of the target.

```
% Find the scan angle closest to the target's direction
tgtAng = [paths.AngleOfArrival];
tgtAz = mean(tgtAng(1,:));
[~,iFrm] = min(abs(azgrid-tgtAz));

% Show the spectrogram for the data taken from the first sweep and first
% receive element
ax = axes(figure);
spectrogram(Xframes(:,1,1,iFrm),[],[],[],Fs);
title(ax,'FMCW Return from Target'); caxis(ax,'auto');
```



```
drawnow
```

Process I/Q Signals into Object Detections

Apply range, Doppler, and beamform processing

Use `helperRangeDopplerProcessing` on page 1-905 to apply range and Doppler processing for each frame of the I/Q data.

```
[Xrngdop,rggrid,rrgrid] = helperRangeDopplerProcessing(Xframes,xcvrLRR);
```

Show the range-Doppler image for the scan angle in the direction of the target.

```
Xdb = pow2db(abs(Xrngdop).^2);
```

```
% Select the frame with the largest return, that is probably the target
[~,iMax] = max(Xdb(:));
[~,~,~,iFrm] = ind2sub(size(Xdb),iMax);
ax = axes(figure);
imagesc(ax,rrgrid,rggrid,squeeze(max(Xdb(:,:,:,iFrm),[],2))); % Select the max across the receive
xlabel(ax,'Range-Rate (m/s)'); ylabel(ax,'Range (m)');
set(ax,'YDir','reverse'); ylim(ax,[0 50]); grid(ax,'on');
title(ax,'Range-Doppler Image from Target');
set(get(colorbar(ax),'YLabel'),'String','Power (dB)');
```
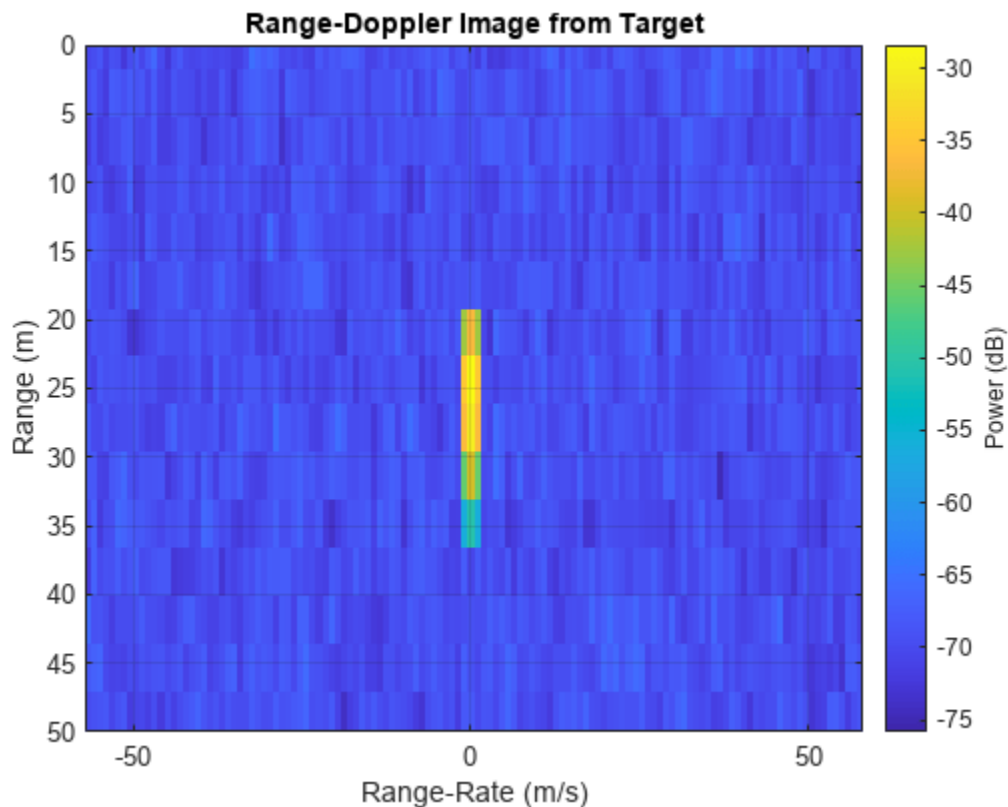

Use `helperBeamProcessing` on page 1-906 to apply beamforming for the virtual array at each scan direction.

```
Xbmfrngdop = helperBeamProcessing(Xrngdop,xcvrLRR,azgrid);
```

The returned data cube has dimensions: range x azimuth x Doppler.

```
disp(size(Xbmfrngdop))
```

```
   364    91   128
```

Generate and cluster detections

Use `helperFindDetections` on page 1-906 to compare the SNR of the range, Doppler, and beamformed data cube to a detection threshold. Select the detection threshold to match the desired detection probability of 0.9 and false alarm rate of 1e-6 of our initial radar design.

```
Pd = predictedLRRMetrics{'Probability of Detection',3}
```

```
Pd = 0.9000
```

```
Pf = lrrSysParam.Pfa
```

```
Pf = 1.0000e-06
```

```
[iDet,estnoiselvldB,threshdB] = helperFindDetections(Xbmfrngdop,Xrngdop,xcvrLRR,Pd,Pf);
```

Show the number of detections found in the processed radar data.

```
numel(iDet)
```

```
ans = 14
```

There are more detected threshold crossings than there are targets. Multiple threshold crossings are expected for each detected target. This occurs not only for extended targets, but even in the case of point targets. This is because the ambiguity functions associated with the waveform and array are not an impulse response but have a sinc-type shape. As a result, multiple range, Doppler, and beam bins may cross the threshold level for a single point target. This is especially true for targets that have a large signal-to-noise ratio (SNR), such as those with a large RCS or that are very close to the radar. Detection crossings need to be clustered to address this. DBSCAN is a common algorithm used for this. Use the `clusterDBSCAN` to identify threshold crossings that are likely coming from a single target. Use `helperIdentifyClusters` on page 1-908 to identify clusters of detections.

```
[clusterIDs,detidx,numClusters] = helperIdentifyClusters(Xbmfrngdop,iDet,azgrid,rggrid,rrgrid);
disp(numClusters)
```

```
     4
```

The number of clustered detections now agrees with the number of point targets used to model the extended target.

Estimate range, range-rate, and azimuth for each cluster

Use `helperMeasurementEstimation` to estimate the range, range-rate, and azimuth angle associated with each cluster of detection threshold crossings.

```
[meas,noise,snrdB] = helperMeasurementEstimation(Xbmfrngdop,estnoiselvldB,xcvrLRR,azgrid,rggrid,
```

Use `helperAssembleDetections` to assemble the detections into a cell array of `objectDetection` objects. This is the interface used by the trackers that ship with Radar toolbox and Sensor Fusion and Tracking toolbox. This also matches the format returned by `radarDataGenerator`.

```
dets = helperAssembleDetections(xcvrLRR,time,meas,noise,snrdB)
```

```
dets=4×1 cell array
    {1x1 objectDetection}
    {1x1 objectDetection}
    {1x1 objectDetection}
    {1x1 objectDetection}
```

Plot the range-angle image of the SNR values in the range, Doppler, and beamformed data cube and overlay the image with the signal-level detections extracted from the data cube.

```
axRgAz = helperPlotRangeAzimuthDetections(bep,Xbmfrngdop,estnoiselvldB,threshdB,xcvrLRR,azgrid,rg
```
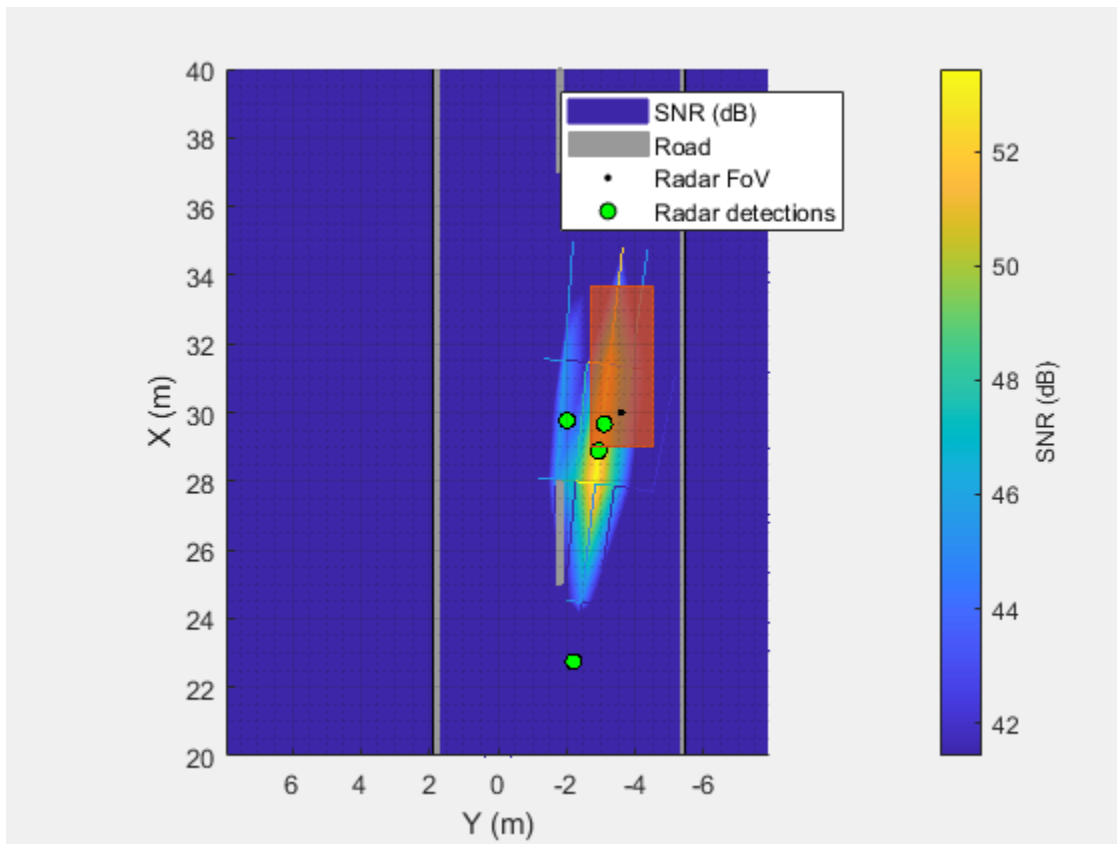

Compare Simulated Measurements with Predicted Metrics

Find the maximum SNR from the signal-level detections.

```
sigLvlSNR = max(cellfun(@(d)d.ObjectAttributes{1}.SNR,dets(:)'));
```

Include processing loss from the target's range relative to the duration of the FMCW sweep. This loss can be reduced by increasing the sweep time of the FMCW waveform, but with the cost of processing more data and longer waveform transmission times.

```
Lsweep = pow2db(numel(rggrid)/sum(rggrid<tgtRg)) % Loss due to range of target and FMCW sweep du
```

```
Lsweep = 16.5801
```

```
sigLvlSNR = sigLvlSNR+Lsweep;
```

Compare the SNR of the detections generated by the signal-level radar to the measurement-level and system-level radars. after accounting for the sweep loss.

```
ax = axes(figure);
X = categorical({'Predicted (Analytical Model)','Simulated (Meas-Level Model)','Simulated (Signa
bar(ax,X,[predictedSNR measLvlSNR sigLvlSNR],'FaceColor', [0.75 1 0.75]);
xtickangle(ax,15)
grid(ax,'on')
ylabel(ax,'SNR (dB)')
title(ax,['SNR Verification: ' num2str(target.RCS) 'dBsm Target @ ' num2str(tgtRg,'%.1f') ' mete
```
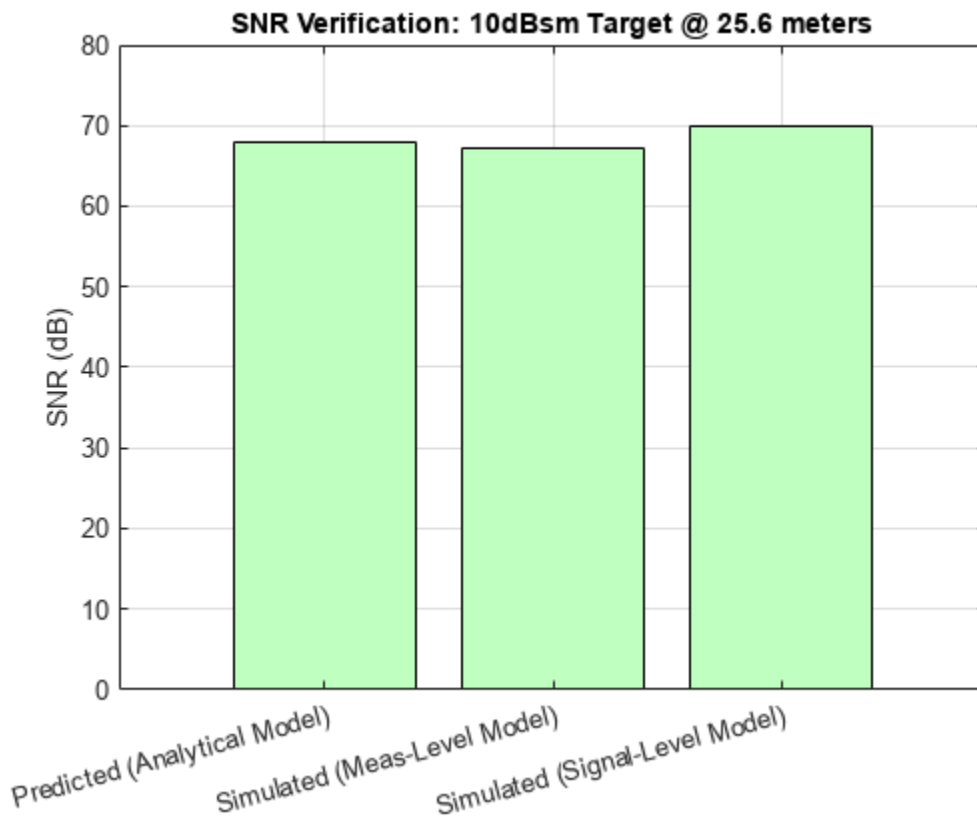

The detections generated from `radarTransceiver` agree with the system- and measurement-level models. A small variation, less-than 3 dB, in SNR is expected due to losses present in the signal-level model which are not included or are assumed constant in the measurement- and system-level models.

Compare the detection locations between the signal-level and measurement-level radars.

```
plotSideBySide(axRgAz,bep);
```
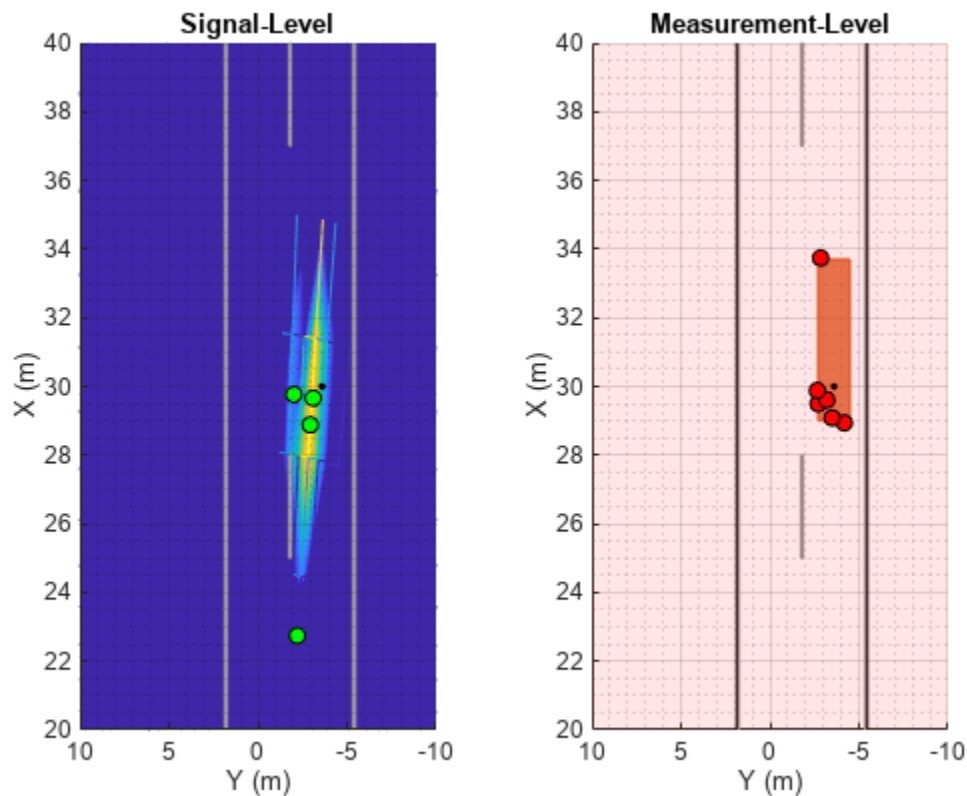
Both radar models generate detections that lie along the exterior of the extended target.

Summary

In this example, you learned how to design and predict performance for a system-level radar using the Radar Designer app. You then learned how to create an equivalent measurement-level radar from the system-level design by using `radarDataGenerator` to generate consistent SNR for the generated detections. You then learned how to generate an equivalent signal-level radar using `radarTransceiver`. You simulated I/Q data using `radarTransceiver` and applied range, Doppler, and beamforming to the I/Q data to extract detections for the baseband I/Q radar data. The signal-level detections were consistent with the measurement-level and system-level radar models and satisfied the design requirements of the original system-level radar.

References

[1] Texas Instruments Incorporated, "Design Guide: TIDEP-01012 - Imaging Radar Using Cascaded mmWave Sensor Reference Design," Texas Instruments, Post Office Box 655303, Dallas, Mar 2020.

[2] Texas Instruments Incorporated, "AWR2243 Single-Chip 76- to 81-GHz FMCW Transceiver," Texas Instruments, Post Office Box 655303, Dallas, Aug 2020.

[3] Texas Instruments Incorporated, "User's Guide: AWRx Cascaded Radar RF Evaluation Module - (MMWCAS-RF-EVM)," Texas Instruments, Post Office Box 655303, Dallas, Feb 2020.

Helper Functions

helperDesignArray

Use `helperDesignArray` to create the array configuration documented in [1 on page 1-902] to realize the large array size required by the azimuth resolution.

```
[txArray, rxArray, arrayMetrics, txPos, rxPos, ant] = helperDesignArray(lambda,Fc);
```

Confirm that the directivity of the `phased.CosineAntennaElement` matches the directivity of 12 dBi reported in Table 1 of [3 on page 1-902]. Directivity is strongly related to an antenna's field of view.

```
disp(pattern(ant,Fc,0,0))

    12.6731
```

Confirm that the receive array gain matches the gain of 24 dBi modeled in the Radar Designer app.

```
Grx = pattern(rxArray,Fc,0,0)
```

```
Grx = 23.5432
```

Compute the azimuth 3-dB beamwidth for the receive array. The 3-dB beamwidth should match the desired azimuth resolution of 1.4 degrees.

```
rxAzBw = beamwidth(rxArray,Fc,'Cut','Azimuth')
```

```
rxAzBw = 1.3400
```

Confirm that the transmit array gain matches the gain of 23 dBi modeled in the Radar Designer app.

```
Gtx = pattern(txArray,Fc,0,0)
```

```
Gtx = 23.3902
```

The transmit array beamwidth is larger than the receive array beamwidth. This is useful since the look directions will be spaced according to the radar's azimuth resolution. The larger transmit beamwidth ensures that all targets within a receive beamwidth will be illuminated by the transmit beam.

```
txAzBw = beamwidth(txArray,Fc,'Cut','Azimuth')
```

```
txAzBw = 3.1800
```

Compute the virtual array element positions by convolving the transmit and receive array element positions. Plot the receive, transmit, and virtual array element positions. The resulting array is a 2D array with the x-axis corresponding to the horizontal elements used to measure azimuth and the y-axis corresponds to vertical elements.

```
vxPos = conv2(txPos,rxPos);
viewArray(txArray,rxArray,lambda,vxPos)
```
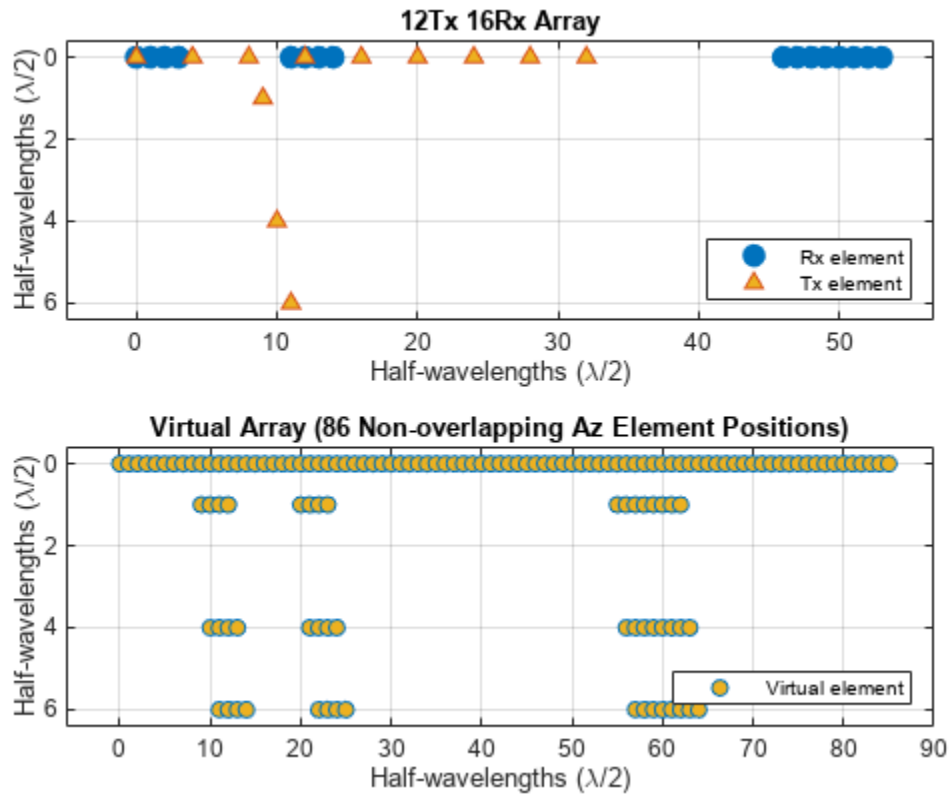
Plot the directivity of the transmit, receive, and virtual arrays.

```
plotDirectivities(txArray,rxArray,Fc)
```
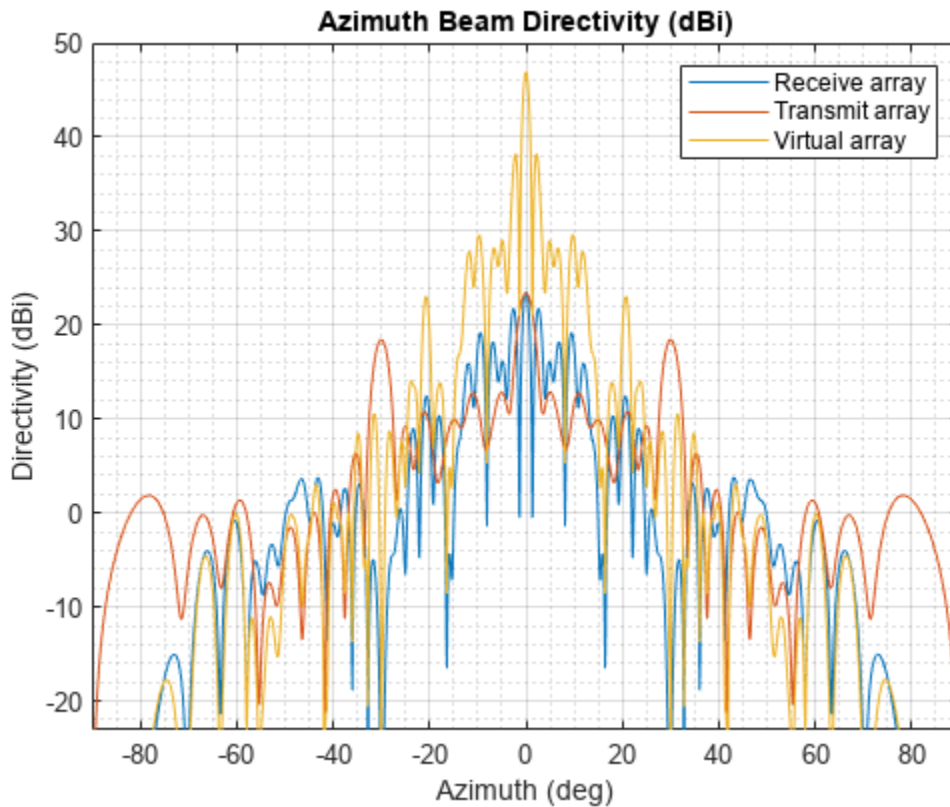
helperRangeDopplerProcessing

`helperRangeDopplerProcessing` creates a `phased.RangeDopplerResponse` and uses `helperRangeDopplerProcess` to apply range and Doppler processing to the collected data cubes.

```
[Xrngdop,rggrid,rrgrid,rngdopproc] = helperRangeDopplerProcessing(Xframes,xcvrLLR)
```

By default, Hanning windows are used in range and Doppler to suppress sidelobes in these dimensions.

```
rngWinFcn = @hanning;
dopWinFcn = @hanning;

rngdopproc = phased.RangeDopplerResponse('RangeMethod','FFT', ...
    'DopplerOutput','Speed','SweepSlope',xcvrLRR.Waveform.SweepBandwidth/xcvrLRR.Waveform.SweepT:

% Configure Doppler properties
rngdopproc.PropagationSpeed = xcvrLRR.ReceiveAntenna.PropagationSpeed;
rngdopproc.OperatingFrequency = xcvrLRR.ReceiveAntenna.OperatingFrequency;
rngdopproc.SampleRate = xcvrLRR.Receiver.SampleRate;
rngdopproc.DopplerWindow = 'Custom';
rngdopproc.CustomDopplerWindow = dopWinFcn;
rngdopproc.DopplerFFTLengthSource = 'Property';
Nd = 2^nextpow2(xcvrLRR.NumRepetitions);
rngdopproc.DopplerFFTLength = Nd;
```

```
% Release memory
clear Xrngdop rggrid rrgrid

[Xrngdop,rggrid,rrgrid] = helperRangeDopplerProcess(rngdopproc,Xframes,xcvrLRR,rngWinFcn);
```

helperBeamProcessing

`helperBeamProcessing` processes the receive beams for each of the scan angles.

```
Xbmfrngdop = helperBeamProcessing(Xrngdop,xcvrLLR,azgrid);
```

`helperBeamProcessing` uses `phased.SteeringVector` to compute the steering vectors for the receive array and `helperVirtualBeamform` to compute the steering vectors for the virtual array and beamform at each scan angle. Use a Hanning window across the non-overlapping elements of the virtual array to suppress azimuth sidelobes.

```
bmfWinFcn = @hanning;

rxSV = phased.SteeringVector( ...
    'SensorArray',xcvrLRR.ReceiveAntenna.Sensor, ...
    'PropagationSpeed',xcvrLRR.ReceiveAntenna.PropagationSpeed);
rxSteer = rxSV(xcvrLRR.ReceiveAntenna.OperatingFrequency,azgrid);

Xbmfrngdop = helperVirtualBeamform(txSteer,rxSteer,Xrngdop,bmfWinFcn);
```

helperFindDetections

`helperFindDetections` extracts detection locations in the range-Doppler and beamform processed data cube.

Compare the SNR of the range, Doppler, and beamformed data cube to a detection threshold. Select the detection threshold to match the desired detection probability of 0.9 and false alarm rate of 1e-6 of our initial radar design.

```
Pd = predictedLRRMetrics{'Probability of Detection',3};
Pf = lrrSysParam.Pfa;

threshdB = detectability(Pd,Pf)

threshdB = 13.1217
```

Compute the SNR of the processed radar data cube. First, estimate the noise floor of the processed data by collecting noise only data. This is noise only data is often collected between transmissions during the idle time of the transceiver or between repetition intervals. A CFAR estimator such as `phased.CFARDetector2D` could also be used to estimate the noise level in cases where the noise floor is not constant.

```
noSignal = zeros(numFT,numRxElmnts,numST);
Ncube = xcvrLRR.Receiver(noSignal(:));
Ncube = reshape(Ncube,numFT,numRxElmnts,numST);
```

Apply the same range, Doppler, and beamform processing to the noise data as was applied to the detection data.

```
Nrngdop = helperRangeDopplerProcess(rngdopproc,Ncube,xcvrLRR,rngWinFcn);
Nbmfrngdop = helperBeamProcessing(Nrngdop,xcvrLRR,0,bmfWinFcn);

% Estimate noise level in dB
estnoiselvldB = pow2db(mean(abs(Nbmfrngdop(:)).^2))

estnoiselvldB = -51.3473
```

Normalize the radar data cube using the noise floor estimate.

```
Xpow = abs(Xbmfrngdop).^2; % Power is amplitude squared
XsnrdB = pow2db(Xpow)-estnoiselvldB; % Normalize using the estimated noise floor power level to g
clear Xpow % release memory
```

Compare the SNR values in the radar data cube to the detection threshold to identify range, Doppler, and azimuth resolution cells that have detections.

```
isDet = XsnrdB(:)>threshdB;

% Number of detection threshold crossings
sum(isDet)

ans = 524
```

Many of the threshold crossings correspond to the sidelobes of the virtual array. Suppression of the sidelobes using a spatial taper alone is often not enough to avoid these spurious detections. Identify sidelobe detections by comparing the SNR of the detection locations for a single element to the SNR in the beams.

```
% Estimate sidelobe level from sidelobe detections
estnoiseslldB = pow2db(mean(abs(Nrngdop(:)).^2));

XslldB = squeeze(max(pow2db(abs(Xrngdop).^2),[],2)); % Max across receive elements
XslldB = XslldB-estnoiseslldB; % Convert to SNR
XslldB = permute(XslldB,[1 3 2]); % range x azimuth x Doppler
isLob = isDet & XsnrdB(:)<XslldB(:);
sum(isLob)

ans = 404
```

Adjust the detection threshold according to the max sidelobe level to further suppress sidelobe detections.

```
maxSLLdB = max(XslldB(isLob)-XsnrdB(isLob))

maxSLLdB = 25.9154

clear XslldB % release memory
threshdB = threshdB+maxSLLdB

threshdB = 39.0371

iDet = find(XsnrdB(:)>threshdB & ~isLob);
```

Remaining number of detections after removal of sidelobe threshold crossings.

```
numel(iDet)

ans = 14
```

helperIdentifyClusters

`helperIdentifyClusters` identifies groups of detection locations coming from a common target. The returned list of cluster IDs are used for measurement estimation from each cluster of detections.

There are more detected threshold crossings than there are targets. Multiple threshold crossings are expected for each detected target. This occurs not only for extended targets, but even in the case of point targets. This is because the ambiguity functions associated with the waveform and array are not an impulse response but have a sinc-type shape. As a result, multiple range, Doppler, and beam bins may cross the threshold level for a single point target. This is especially true for targets that have a large signal-to-noise ratio (SNR), such as those with a large RCS or that are very close to the radar. Detection crossings need to be clustered to address this. DBSCAN is a common algorithm used for this. Use the `clusterDBSCAN` to identify threshold crossings that are likely coming from a single target.

```matlab
% Compute the range, angle, and Doppler indices of the detection crossings
[iRg,iAng,iRr] = ind2sub(size(XsnrdB),iDet);
detidx = [iRg iAng iRr];

% Select the range, angle, and Doppler bin centers associated with each
% detection crossing
meas = [rggrid(iRg) azgrid(iAng)' rrgrid(iRr)];
```

Estimate the epsilon to use for DBSCAN clustering.

```matlab
minNumPoints = 2;
maxNumPoints = 10;
epsilon = clusterDBSCAN.estimateEpsilon(meas,minNumPoints,maxNumPoints)
```

```
epsilon = 2.8052
```

```matlab
% Range and Doppler ambiguity limits
ambLims = [ ...
    [min(rggrid) max(rggrid)]; ...
    [min(rrgrid) max(rrgrid)]];
clusterDets = clusterDBSCAN( ...
    'MinNumPoints',1, ...
    'EnableDisambiguation',true, ...
    'AmbiguousDimension',[1 3], ...
    'Epsilon',epsilon);
```

Identify the detections that should be clustered into a single detection. The threshold crossings are assigned IDs. Crossings that appear to come from the same target will be assigned the same ID.

```matlab
[indClstr,clusterIDs] = clusterDets(meas,ambLims);
clusterIDs = clusterIDs(:)';

% Number of unique targets estimated after clustering the threshold crossings.
if isempty(clusterIDs)
    numDets = 0;
else
    numDets = max(clusterIDs);
end

% Plot clusters
```

```
ax = axes(figure);
plot(clusterDets,meas,indClstr,'Parent',ax);
xlabel(ax,'Range (m)'); ylabel(ax,'Azimuth (deg)'); zlabel(ax,'Range-rate (m/s)');
view(ax,[30 60]);
```
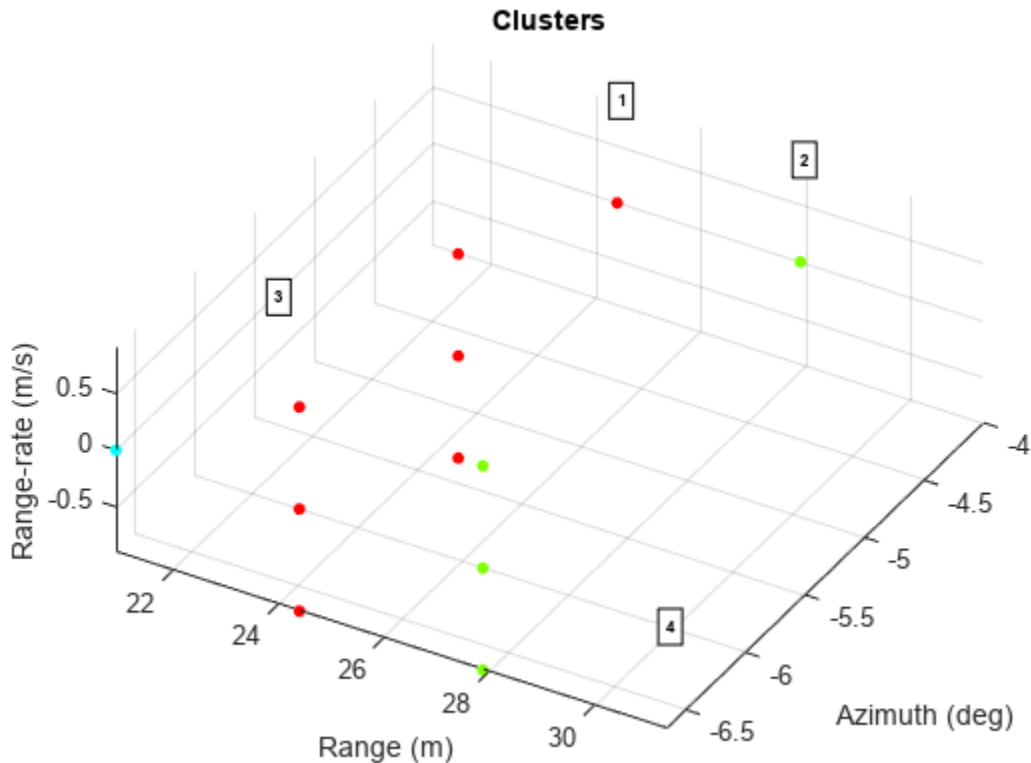

viewArray

viewArray displays the element positions of the transmit, receive, and virtual arrays generated by helperDesignArray.

```
function viewArray(txArray,rxArray,lambda,vxPos)
tl = tiledlayout(figure,2,1,'TileSpacing','loose');

% Plot Tx an Rx arrays
ax = nexttile(tl);
clrs = colororder(ax);
spacing = lambda/2;
p1 = plot(ax,rxArray.ElementPosition(2,:)/spacing,rxArray.ElementPosition(3,:)/spacing,'o','Marke
hold(ax,'on'); drawnow;
p2 = plot(ax,txArray.ElementPosition(2,:)/spacing,txArray.ElementPosition(3,:)/spacing,'^','Marke
xlabel(ax,'Half-wavelengths (\lambda/2)'); ylabel(ax,'Half-wavelengths (\lambda/2)');
title(ax,'12Tx 16Rx Array'); grid(ax,'on');
set(ax,'YDir','reverse');
legend([p1 p2],'Rx element','Tx element','Location','SouthEast');
xlim(ax,'padded'); ylim(ax,'padded');

% Plot virtual array
ax = nexttile(tl);
```

```
[row,col] = ind2sub(size(vxPos),find(vxPos(:)));
p3 = plot(ax,row-1,col-1,'o','Color',clrs(1,:),'MarkerFaceColor',clrs(3,:));
xlabel(ax,'Half-wavelengths (\lambda/2)'); ylabel(ax,'Half-wavelengths (\lambda/2)');
title(ax,'Virtual Array (86 Non-overlapping Az Element Positions)'); grid(ax,'on');
set(ax,'YDir','reverse');
legend(p3,'Virtual element','Location','SouthEast');
xlim(ax,'padded'); ylim(ax,'padded');
end
```

plotDirectivities

plotDirectivities plots the azimuth antenna patterns for the transmit, receive, and virtual arrays.

```
function plotDirectivities(txArray,rxArray,Fc)
angs = -180:0.1:180;
patRx = patternAzimuth(rxArray,Fc,'Azimuth',angs);
patTx = patternAzimuth(txArray,Fc,'Azimuth',angs);
patVx = patTx+patRx;

ax = axes(figure);
p1 = plot(ax,angs,patRx);
hold(ax,'on');
p2 = plot(ax,angs,patTx);
p3 = plot(ax,angs,patVx);
xlabel(ax,'Azimuth (deg)'); ylabel(ax,'Directivity (dBi)');
Gmax = max([patRx(:);patTx(:);patVx(:)]);
grid(ax,'on'); grid(ax,'minor');
ylim(ax,[Gmax-70 max(ylim)]); xlim(ax,[-90 90]);
title(ax,'Azimuth Beam Directivity (dBi)');
legend([p1 p2 p3],'Receive array','Transmit array','Virtual array');
end
```

poses2paths

poses2paths converts the target poses to a path struct array defining the two-way freespace propagation paths between the radarTransceiver and the targets in the scenario.

```
function paths = poses2paths(tposes,txrx,scenario,rdr,bep)
% Transport the target poses from the ego's body frame to the sensor's
% mounted frame
tposesMnt = helperPosesToSensorFrame(tposes,txrx);

% Sample target extent and compute RCS and viewed angles
rf = txrx.ReceiveAntenna.OperatingFrequency;
lambda = freq2wavelen(rf,txrx.ReceiveAntenna.PropagationSpeed);

% Profiles define the dimensions and RCS of the targets
profiles = actorProfiles(scenario);

% Compute the angle at which each target is viewed by the sensor
[rcsdBsm,tids] = helperTargetRCS(tposesMnt,profiles,rf);

map = containers.Map(tids,rcsdBsm);
rcsTbl = @(ids)arrayfun(@(id)map(id),ids); % Create a lookup table

% Map the targets to sets of points according to the sensor's resolution
res = NaN(1,3);
```

```
        res(1) = rdr.AzimuthResolution;
        if rdr.HasElevation
            res(2) = rdr.ElevationResolution;
        end
        res(3) = rdr.RangeResolution;
        rgLim = [0 rdr.RangeLimits(2)];
        fov = rdr.FieldOfView;
        [posMnt,velMnt,ids] = helperSphericallySampleTargets(tposesMnt,profiles,res,fov,rgLim);
        rcsdBsm = rcsTbl(ids);

        [rot,off] = helperSensorFrame(txrx);
        posBdy = rot*posMnt + off;

        if nargin>4
            % Hide the detections
            h = findobj(bep.Parent,'Tag','bepTrackPositions_Radar ID1 detections');
            h.Visible = 'off';
            %clearData(bep.Plotters(end)); % Clear detections

            hold(bep.Parent,'on');
            hndl = plot(bep.Parent,posBdy(1,:),posBdy(2,:),'^k','MarkerFaceColor','y','DisplayName','Samp
            hold(bep.Parent,'off');
            leg = legend(bep.Parent);
            leg.PlotChildren = [leg.PlotChildren;hndl];
        end

        paths = helperFreespacePaths(posMnt,velMnt,lambda,rcsdBsm);
        end
```

plotSideBySide

`plotSideBySide` creates a side-by-side comparison of the detections generated using `radarTransceiver` on the left and the detections generated using `radarDataGeneartor` on the right.

```
function plotSideBySide(axRgAz,bep)
tl = tiledlayout(figure,1,2,'TileSpacing','loose');

ax = nexttile(tl);
copyobj(axRgAz.Children,ax);
view(ax,-90,90)
axis(ax,axis(axRgAz));
caxis(ax,caxis(axRgAz));
xlabel(ax,'X (m)'); ylabel(ax,'Y (m)');
title(ax,'Signal-Level');
grid(ax,'on'); grid(ax,'minor');
set(ax,'Layer','top');
ylim(ax,10*[-1 1])

ax = nexttile(tl);
axBEP = bep.Parent;
shh = get(groot,'ShowHiddenHandles');
set(groot,'ShowHiddenHandles','on');
copyobj(axBEP.Children,ax);
set(groot,'ShowHiddenHandles',shh);
view(ax,-90,90)

axis(ax,axis(axRgAz));
```

```matlab
xlabel(ax,'X (m)'); ylabel(ax,'Y (m)');
title(ax,'Measurement-Level');
grid(ax,'on'); grid(ax,'minor');
ylim(ax,10*[-1 1])

h = findobj(ax,'Tag','bepTrackPositions_Radar ID1 detections');
if ~isempty(h) && ishghandle(h)
    h.Visible = 'on';
end

h = findobj(ax,'DisplayName','Sampled points');
if ~isempty(h) && ishghandle(h)
    delete(h);
end
end
```

ERS SAR Raw Data Extraction And Image Formation

This example shows the steps to extract the European Remote Sensing (ERS) Synthetic Aperture Radar (SAR) system parameters, unfocused raw data and form a focused image from raw data using range migration image formation algorithm.

ERS dataset is NASA's provision of the European Space Agency's ERS satellites (ERS-1 and ERS-2) SAR data. As part of the Earth observation Heritage Data Program (LTDP+), the ERS missions provide scientists with historically accurate and easily accessible information to help further understand the dynamics of our planet. ERS dataset contains raw unfocused data along with system parameters file which can be used to generate focused image of the scene.

To illustrate this workflow, ERS dataset published by the NASA's Alaska Satellite Facility (ASF) [1] is used. The dataset is available for download here. Our goal is to develop a model to generate a focused image from the raw data.

Download Dataset

This example uses ERS dataset containing Committee on Earth Observation Satellites (CEOS) standard files namely: leader file (.ldr), level zero frame data file (.raw), volume descriptor file (.vol), processing information file (.pi), metadata file (.meta) and null file (.nul). The CEOS leader file contains relevant metadata information to be able to process the SAR data (.raw) it accompanies. Level zero frame data file (.raw) contains binary SAR signal data collected after processing the analog SAR signal. The volume descriptor is part of the CEOS frame distribution, containing a short summary about the processing. The processing information file provides information on the CEOS conversion describing the data set itself as well as its location on the system. The metadata file is an ASCII file that contains most of the metadata needed for using the ASF software tools. The null volume directory file is part of the CEOS frame distribution. It is used to terminate a logical data volume.

Download the dataset from the given URL using the `helperDownloadERSData` helper function. The size of dataset is 134 MB. `ERSData.zip` contains three files namely: leader file, raw file, and license file.

```
outputFolder = pwd;
dataURL = ['https://ssd.mathworks.com/supportfiles/radar/data/' ...
    'ERSData.zip'];
helperDownloadERSData(outputFolder,dataURL);
```

Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser and extract the file. If you do so, change the `outputFolder` variable in the code to the location of the downloaded file.

Parameter Extraction

SAR image formation involves extracting the parameters from the parameter file (E2_84699_STD_L0_F299.000.ldr) given with raw data file. The parameter file contains satellite and scene specific data which is required to form the image. Several parameters like pulse repetition frequency, wavelength, sampling rate, pulsewidth of the chirp signal, range gate delay and velocity of

the sensor are extracted from the parameter file according to their specific addresses given in the description of the parameter file. Other parameters like effective velocity, minimum distance, chirp bandwidth and chirp frequency are estimated using the data extracted from the parameter file.

In this example, system parameters are extracted using the helper function ERSParameterExtractor which takes E2_84699_STD_L0_F299.000.ldr as an input file.

```
c = physconst('LightSpeed');                    % Speed of light

% Extract ERS system parameters
[fs,fc,prf,tau,bw,v,ro,fd] = ERSParameterExtractor('E2_84699_STD_L0_F299.000.ldr');
```

Raw Data Extraction

Raw data is extracted from the data file (E2_84699_STD_L0_F299.000.raw) having a total of 28652 lines and 11644 bytes per line. In each line, header consists of 412 bytes and remaining 11232 bytes (5616 complex) of data for each echo. For the ERS radar, the synthetic aperture is 1296 points long. Therefore, the helper function ERSDataExtractor reads 2048 rows since it is a power of 2. Using 80% beamwidth the helper function calculates valid azimuthal points and takes overlapping patches.

In this example, unfocused raw data is extracted using the helper function which takes E2_84699_STD_L0_F299.000.raw as an input file along with system parameters.

```
% Extract raw data
rawData = ERSDataExtractor('E2_84699_STD_L0_F299.000.raw',fs,fc,prf,tau,v,ro,fd).';
```

SAR Image Formation

The next step is to generate single-look complex (SLC) image using the extracted system parameters and unfocused raw data. For the ERS-radar, a linear frequency modulated chirp is emitted by the radar. The phased.LinearFMWaveform System object creates a linear FM pulse waveform using the extracted system parameters. Squint angle is calculated using Doppler frequency which is close to zero for the ERS system. Range migration image formation algorithm is a frequency domain algorithm also known as wavenumber domain processing method. Use rangeMigrationLFM function to focus the raw data to SLC image.

SAR SLC image is characterized by speckle which can be modeled as a multiplicative noise, resulting in salt and pepper appearance of SAR image.

```
% Create LFM waveform
waveform = phased.LinearFMWaveform('SampleRate',fs,'PRF',prf,'PulseWidth',tau,'SweepBandwidth',bw
sqang = asind((c*fd)/(2*fc*v));          % Squint angle

% Range migration algorithm
slcimg = rangeMigrationLFM(rawData,waveform,fc,v,ro,'SquintAngle',sqang);

% Display image
figure(1)
imagesc(log(abs(slcimg)))
axis image
colormap('gray')
title('SLC Image')
ylabel('Range bin')
xlabel('Azimuth bin')
```
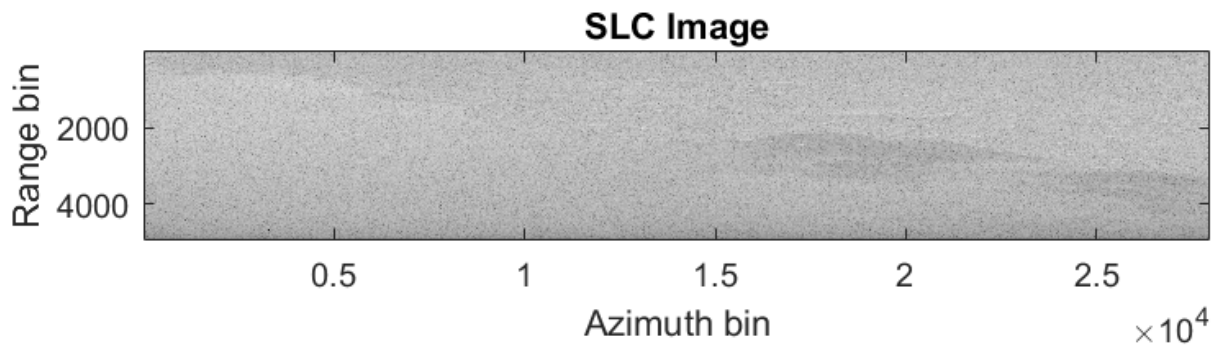
Multi-look Processing

The effect of speckle is reduced by performing multi-look processing by making a trade-off for image resolution. Either in range or azimuth direction or in both the directions, subsequent lines are averaged to get a better image with reduced speckle. The multi-looking can be performed either in range or azimuth or both the direction.

The helper function `multilookProcessing` performs averaging in range and azimuth direction with number of looks 4 and 20 respectively.

```
mlimg = multilookProcessing(abs(slcimg),4,20);
```

```
% Display Image
figure(2)
imagesc(log(abs(mlimg(1:end-500,:))))
axis image
colormap('gray')
title('Multi-look Image')
ylabel('Range bin')
xlabel('Azimuth bin')
```

Multi-look Image
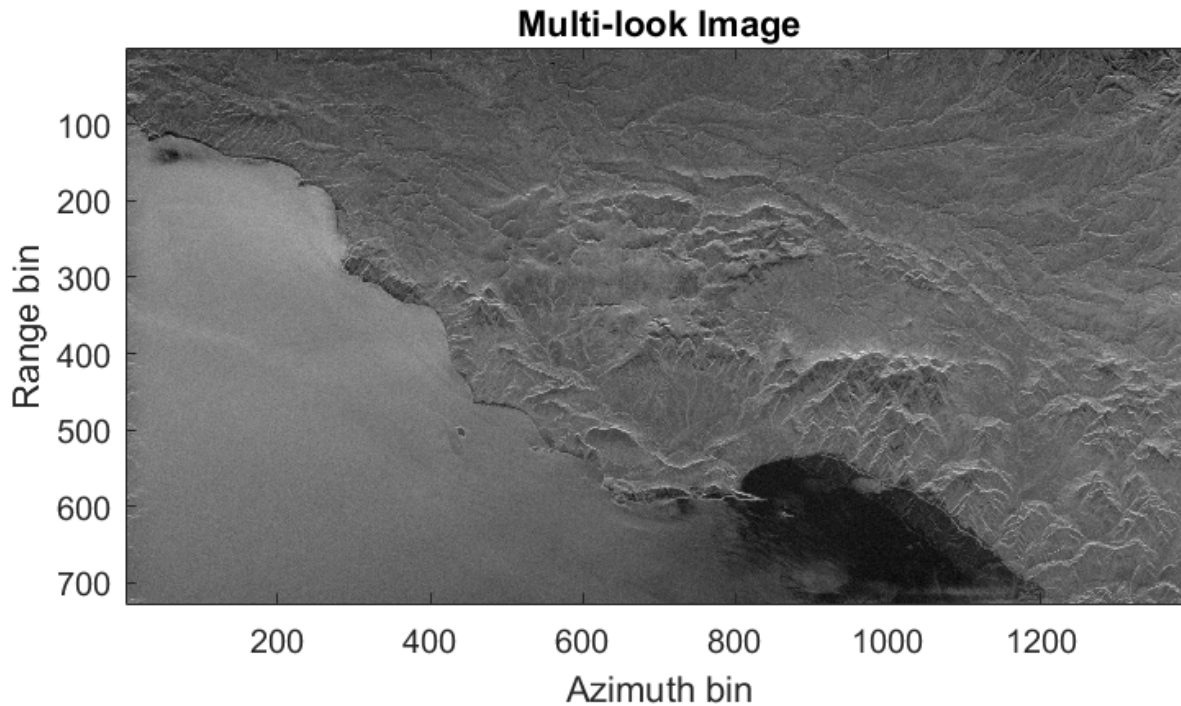
Summary

This example shows how to extract the system parameters such as pulse repetition frequency, wavelength, sampling rate, pulsewidth of the chirp signal, range gate delay, velocity of the sensor and raw data. Then the range migration image formation algorithm is used to focus the raw data. Finally, the multi-look processing is applied to remove the multiplicative noise.

Helper Functions

ERSParameterExtractor

```
function [fs,fc,prf,tau,bw,veff,ro,fdop] = ERSParameterExtractor(file)
% Open the parameter file to extract required parameters
fid = fopen(file,'r');

% Radar wavelength (satellite specific)
status = fseek(fid,720+500,'bof');
lambda = str2double(fread(fid,[1 16],'*char'));        % Wavelength (m)

% Pulse Repetition Frequency (satellite specific)
status = fseek(fid,720+934,'bof')|status;
```

```matlab
prf = str2double(fread(fid,[1 16],'*char'));              % PRF (Hz)

% Range sampling rate (satellite specific)
status = fseek(fid,720+710,'bof')|status;
fs =str2double(fread(fid,[1 16],'*char'))*1e+06;          % Sampling Rate (Hz)

% Range Pulse length (satellite specific)
status = fseek(fid,720+742,'bof')|status;
tau = str2double(fread(fid,[1 16],'*char'))*1e-06;        % Pulse Width (sec)

% Range Gate Delay to first range cell
status = fseek(fid,720+1766,'bof')|status;
rangeGateDelay = str2double(fread(fid,[1 16],'*char'))*1e-03;   % Range Gate Delay (sec)

% Velocity X
status = fseek(fid,720+1886+452,'bof')|status;
xVelocity = str2double(fread(fid,[1 22],'*char'));        % xVelocity (m/sec)

% Velocity Y
status = fseek(fid,720+1886+474,'bof')|status;
yVelocity = str2double(fread(fid,[1 22],'*char'));        % yVelocity (m/sec)

% Velocity Z
status = fseek(fid,720+1886+496,'bof')|status;
zVelocity = str2double(fread(fid,[1 22],'*char'));        % zVelocity (m/sec)
fclose(fid);

% Checking for any file error
if(status==1)
    fs = NaN;
    fc = NaN;
    prf = NaN;
    tau = NaN;
    bw = NaN;
    veff = NaN;
    ro = NaN;
    fdop = NaN;
    return;
end

% Values specific to ERS satellites
slope = 4.19e+11;            % Slope of the transmitted chirp (Hz/s)
h = 790000;                  % Platform altitude above ground (m)
fdop = -1.349748e+02;        % Doppler frequency (Hz)

% Additional Parameters
Re = 6378144 ;               % Earth radius (m)

% Min distance
ro = time2range(rangeGateDelay);  % Min distance (m)

% Effective velocity
v = sqrt(xVelocity^2+yVelocity^2+zVelocity^2);
veff = v*sqrt(Re/(Re+h));    % Effective velocity (m/sec)

% Chirp frequency
fc = wavelen2freq(lambda);   % Chirp frequency (Hz)
```

1-917

```matlab
% Chirp bandwidth
bw = slope*tau;                 % Chirp bandwidth (Hz)
end
```

ERSDataExtractor

```matlab
function rawData = ERSDataExtractor(datafile,fs,fc,prf,tau,v,ro,doppler)
c = physconst('LightSpeed');                    % Speed of light

% Values specific to data file
totlines = 28652;                               % Total number of lines
numLines = 2048;                                % Number of lines
numBytes = 11644;                               % Number of bytes of data
numHdr = 412;                                   % Header size
nValid = (numBytes-numHdr)/2 - round(tau*fs);   % Valid range samples

% Antenna length specific to ERS
L = 10;

% Calculate valid azimuth points
range = ro + (0:nValid-1) * (c/(2*fs));         % Computes range perpendicular to azimuth di
rdc = range/sqrt(1-(c*doppler/(fc*(2*v))^2));   % Squinted range
azBeamwidth = rdc * (c/(fc*L)) * 0.8;           % Use only 80%
azTau = azBeamwidth / v;                        % Azimuth pulse length
nPtsAz = ceil(azTau(end) * prf);                % Use the far range value
validAzPts = numLines - nPtsAz ;                % Valid azimuth points

% Start extracting
fid = fopen(datafile,'r');
status = fseek(fid,numBytes,'bof');       % Skipping first line
numPatch = floor(totlines/validAzPts);    % Number of patches


if(status==-1)
    rawData = NaN;
    return;
end
rawData=zeros(numPatch*validAzPts,nValid);
% Patch data extraction starts
for patchi = 1:numPatch
    fseek(fid,11644,'cof');
    data = fread(fid,[numBytes,numLines],'uint8')'; % Reading raw data file

    % Interpret as complex values and remove mean
    data = complex(data(:,numHdr+1:2:end),data(:,numHdr+2:2:end));
    data = data - mean(data(:));

    rawData((1:validAzPts)+((patchi-1)*validAzPts),:) = data(1:validAzPts,1:nValid);
    fseek(fid,numBytes + numBytes*validAzPts*patchi,'bof');
end
fclose(fid);
end
```

multilookProcessing

```matlab
function image = multilookProcessing(slcimg,sx,sy)
[nx,ny] = size(slcimg);
nfx = floor(nx/sx);
```

```matlab
nfy = floor(ny/sy);
image = (zeros(nfx,nfy));
for i=1:nfx
    for j = 1:nfy
        fimg=0;
        for ix = 1:sx
            for jy = 1:sy
                fimg = fimg+slcimg(((i-1)*sx)+ix,((j-1)*sy)+jy);
            end
        end
        image(i,j) = fimg/(sx*sy);
    end
end
end
```

helperDownloadERSData

```matlab
function helperDownloadERSData(outputFolder,DataURL)
% Download the data set from the given URL to the output folder.

    radarDataZipFile = fullfile(outputFolder,'ERSData.zip');

    if ~exist(radarDataZipFile,'file')

        disp('Downloading ERS data (134 MiB)...');
        websave(radarDataZipFile,DataURL);
        unzip(radarDataZipFile,outputFolder);
    end
end
```

References

[1] Dataset: ERS-2, ESA 2011. Retrieved from ASF DAAC 10 November 2021.

Processing Radar Reflections Acquired with the Demorad Radar Sensor Platform

Introduction

This example demonstrates how to process and visualize FMCW radar echoes acquired via the Demorad Radar Sensor Platform with the Phased Array System Toolbox™ and Simulink®. The I/Q samples and operating parameters are read from a binary file that is provided with this example. The system setup is similar to the MATLAB® "Processing Radar Reflections Acquired with the Demorad Radar Sensor Platform" on page 1-347 example.

System Setup
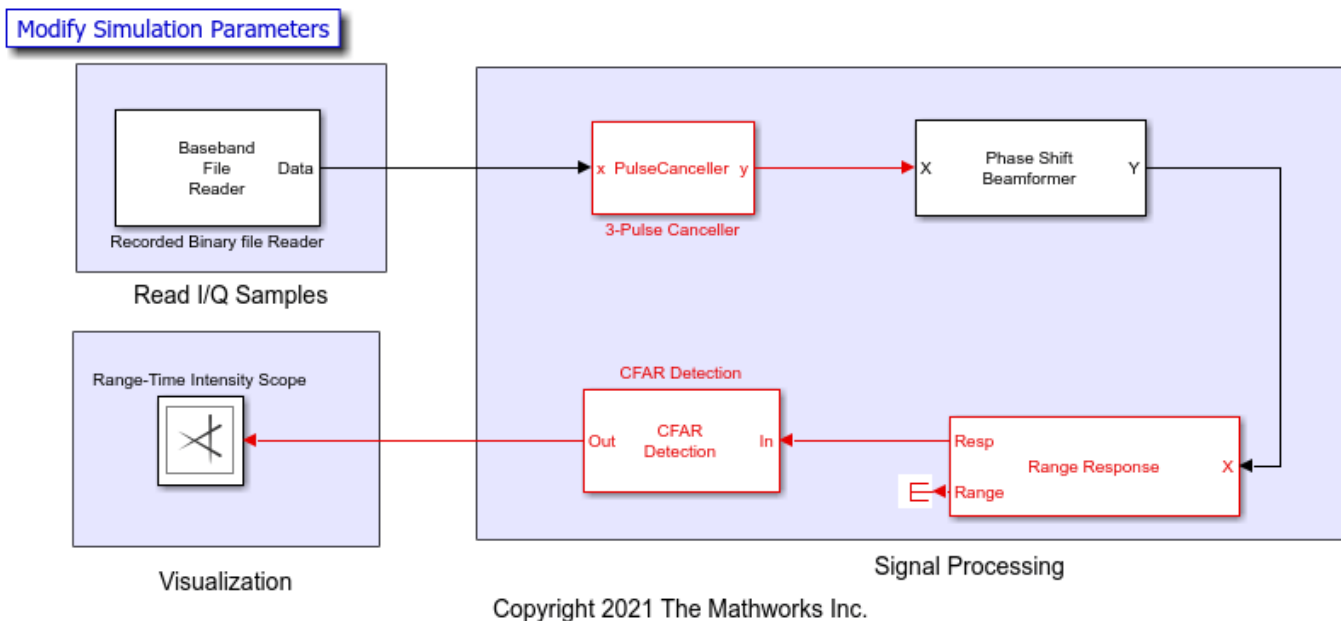
Radar Data Acquisition

In this section, the I/Q samples are read from the binary file using the baseband file reader. These I/Q samples recorded in the binary file are returns from the sensor without any digital processing. The file also contains the parameters that define the transmitted FMCW chirp, which were written when the file was created. These parameters are used to initialize the signal processing blocks in the model. The radar capabilities are same as what is defined in "Processing Radar Reflections Acquired with the Demorad Radar Sensor Platform" on page 1-347 example.

Signal Processing

The signal processing involves pulse cancellation of received I/Q samples to remove stationary clutters in the acquired data. This is followed by beamforming, range processing and finally moving targets are detected using a constant false alarm rate (CFAR) detector.

Pulse canceller A three-pulse canceller is then used to eliminate the stationary clutters from the data read using the binary file reader.

Range processing Followed by the pulse cancelling, we perform beamforming and leverage the sensor model and operating frequency of the Demorad. Demorad transmits FMCW waveform, hence the beamformed signal is range processed via FFT.

CFAR Detection A constant false alarm rate (CFAR) detector is then used to detect any moving targets within range 0 - 15 meters. Followed by this we preprocess the data before sending it on to the scope for visualization.
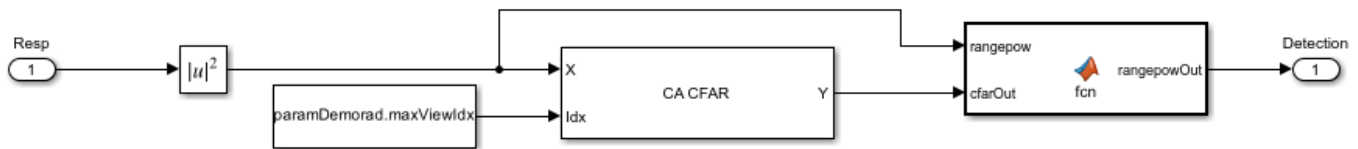
Visualization

Initialize the scope to visualize the range processed FMCW reflections. We set up the range-time intensity scope to visualize the data for 15 seconds.
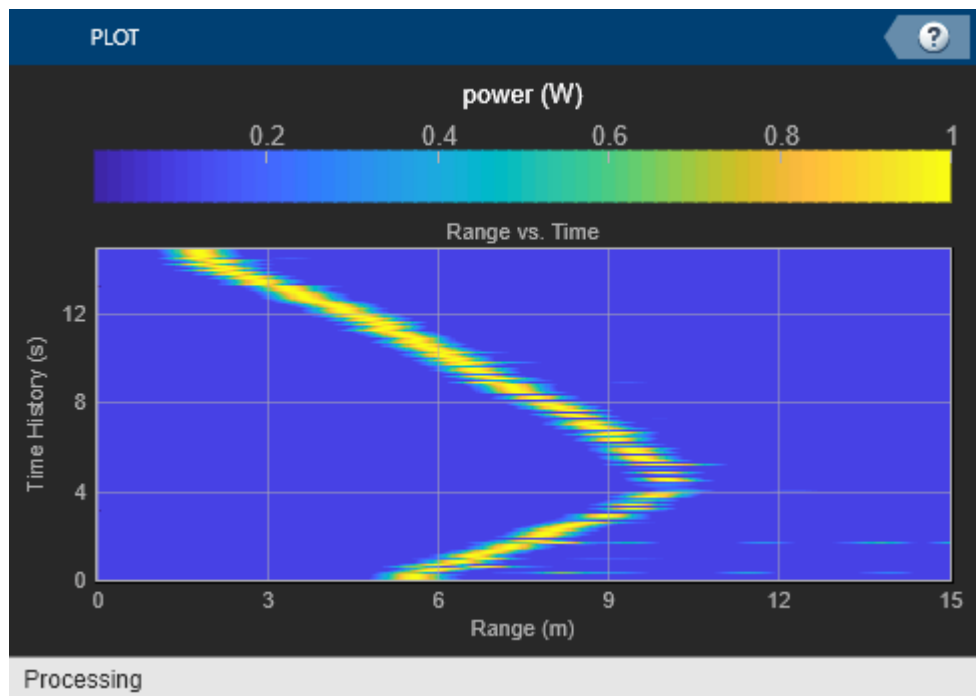
Exploring the model

Several dialog parameters of the model are calculated by the helper function `helperDemoradParams`. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced by the dialogs.

Results and Display
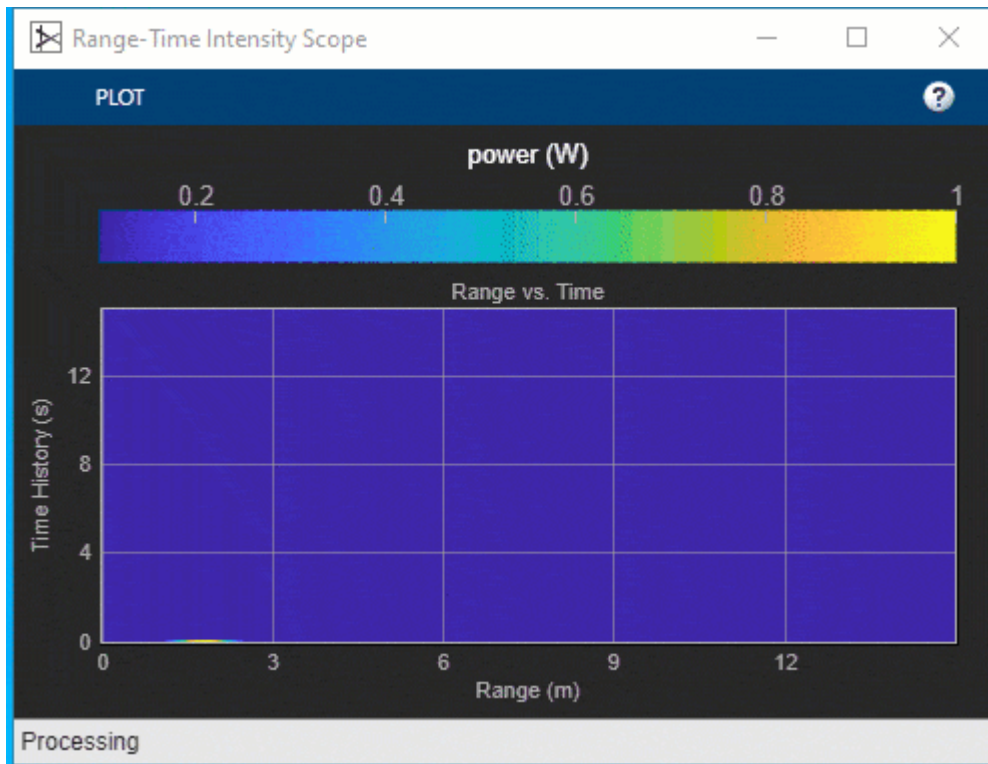
The range-processed data is shown in the range-time intensity scope. It can be seen from the scope that a target is moving away from radar until 10 m and then returning towards the radar.

Summary

This example demonstrates how to process I/Q data acquired and visualize radar reflections.

Predict Surface Clutter Power in Range-Doppler Space

This example shows how to calculate the radar cross section (RCS) of a flat featureless surface in range-Doppler space. You will see how to work with the surface RCS to examine the presentation of clutter, analyze the detectability of surface targets, and estimate the total clutter power in a resolution cell by using approximate resolution cell centroids.

Doppler-Processed Clutter Returns

This section introduces the concept of Doppler-processed surface clutter returns and the `clutterSurfaceRangeDopplerRCS` function, which computes the RCS of clutter in a set of range-Doppler cells. This function expands upon the `clutterSurfaceRCS` function by computing how clutter returns are distributed over Doppler in addition to range. The function is limited to flat-Earth models with a smooth surface where the normalized RCS (NRCS) can be specified as a function of range.

Range-Doppler Cells

A pulse-Doppler radar system uses a series of coherent pulses to measure the Doppler frequency of a return signal. A set of range gates are used to bin received energy according to range, and a set of Doppler filters are used to bin received energy according to the Doppler frequencies of the signal. This forms a discrete two-dimensional range-Doppler map (RDM) of the received energy. Each pixel of this map corresponds to a range-Doppler cell - a region in space demarcated by two iso-range spheres and two iso-Doppler cones. The intersection of this 3-dimensional region with the surface defines a surface clutter range-Doppler cell whose size corresponds to the amount of received energy in the corresponding pixel of the RDM.
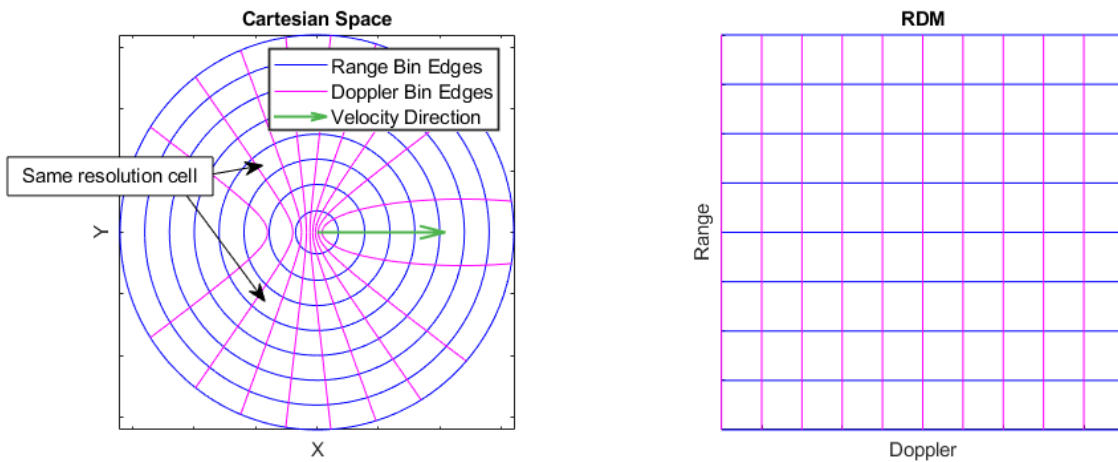
The above figure illustrates the mapping of range-Doppler cells from Cartesian space to the RDM. Note that the distribution of range-Doppler cells is symmetric across the radar's velocity. As a result, most of the range-Doppler cells consist of two disjoint regions in Cartesian space.

Clutter Presentation

This section demonstrates workflows for using the output of `clutterSurfaceRangeDopplerRCS` to examine how surface clutter is presented in an RDM. You will see how to find the overall range and

Doppler extent of the surface, the range and Doppler extent of a conical beam, and how to quickly estimate the Doppler bandwidth of clutter as a function of range.

The Surface in Range-Doppler Space

Surface clutter is distributed over range starting at the range to the nadir point (the radar altitude) out to the horizon. It is also distributed over 360 degrees of azimuth angle and up to 90 degrees of elevation angle. The large angular spread of surface clutter causes it to be distributed over a range of Doppler frequencies as well. Understanding where surface clutter will appear in a set of range-Doppler cells, and how much of the surface contributes to each cell, is important for many applications including control system design and detectability analysis. Typical radar antennas use a tight beam to isolate small regions of the surface, and in this example you will see how to incorporate radar beam information into the analysis of clutter return. In this analysis, a flat-Earth assumption is used and as such, surface clutter is unbounded in range. The interval of ranges covered is then [Altitude, ∞), but a maximum range of interest can be specified.

Unlike range, the Doppler frequency of surface clutter will always be bounded and may be limited either by the maximum range sampled by the radar or by the speed of the radar. Besides the total speed of the radar, the z-component of the radar's velocity must be known, which is captured by the *dive angle* - the angle of the velocity vector with the horizontal, with positive dive indicating the velocity is pointing down.

The function `helperSurfaceDopplerLimits` is provided to calculate the min and max Doppler of surface clutter, and the figure below shows the maximum surface Doppler as a function of radar speed and max range for a 10 GHz radar at an altitude of 3 km. When the dive angle is nonzero, there will be a critical range (indicated as a red dashed line in the right subplot) at which the velocity vector intersects the surface, above which the maximum Doppler depends only on the radar speed.

```
freq = 10e9; % Hz
alt = 3e3;   % m
helperSurfaceDopplerLimitsChart(freq,alt)
```
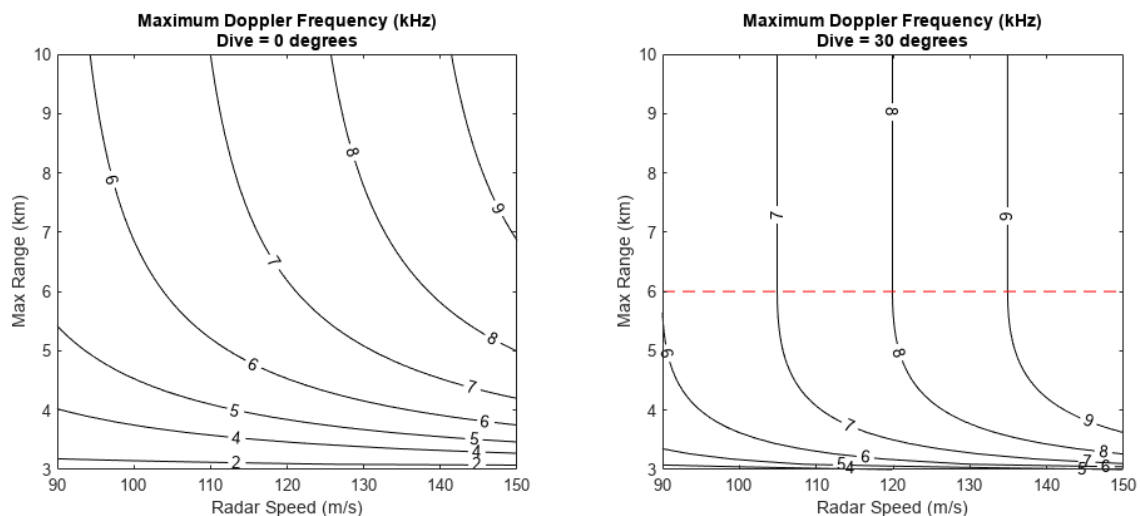

Run the `clutterSurfaceRangeDopplerRCS` function to see the surface RCS. Start by defining the required system and platform parameters. Use a 78 Hz Doppler resolution, and let the platform travel at 120 m/s with a dive angle of 10 degrees.

```
dopres = 78;  % Hz
speed = 120;  % m/s
dive = 10;    % deg
```

Define a range swath that starts at 5 km, covering 5 km of range at a 40 m resolution. This defines the centers of each slant range bin.

```
R = 5e3:40:10e3;
```

Create a `surfaceReflectivityLand` object for getting the required NRCS values. The Barton model for flat land surfaces is used by default. Use the `grazingang` function to find the grazing angles corresponding to each range, then get the NRCS at each range.

```
graze = grazingang(alt,R,"Flat");
refl = surfaceReflectivityLand;
nrcs = refl(graze,freq);
```

Compute the RCS of clutter over the specified range swath and use the provided helper function to plot the results.

```
[rcs,dop] = clutterSurfaceRangeDopplerRCS(nrcs,R,freq,dopres,alt,speed,dive);
helperPlotSurfaceRCS(R,dop,rcs)
```
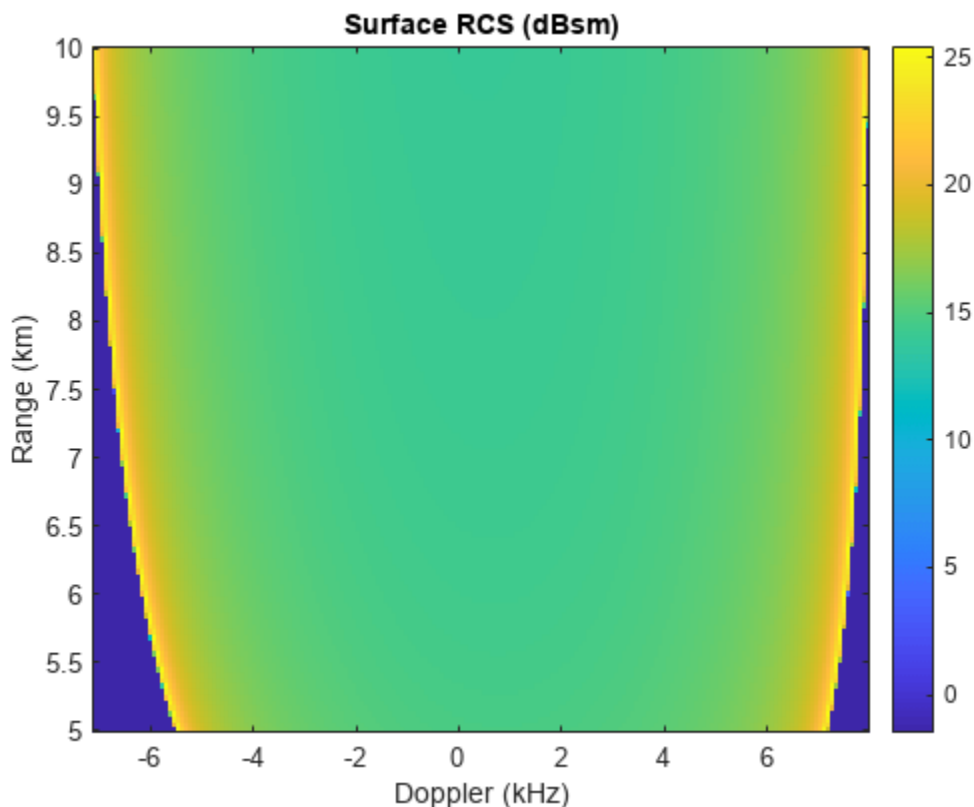

Some interesting things can be observed in this result:

- The nonzero dive angle results in the clutter return being skewed slightly towards higher Doppler at lower ranges and lower Doppler at higher ranges.

- The Doppler extent of surface clutter increases with range
- The clutter around 0 Hz, corresponding to the broadside direction, shows an RCS that changes only slightly with range
- The clutter return at the min and max Doppler in each range gate is spikier and less predictable

The Beam Footprint in Range-Doppler Space

The previous section looked at the overall extent of the surface in range-Doppler space, but to understand the whole story, the radar beam should be considered as well.

The `clutterSurfaceRangeDopplerRCS` function will compute the RCS of clutter over all 360 degrees of azimuth and does not consider limiting by a radar beam. For a simple conical beam defined by an azimuth and elevation beamwidth, the min and max range of the beam footprint and the min and max Doppler cone inside the beam can be calculated in closed form. This information may be used in tandem with the output from `clutterSurfaceRangeDopplerRCS` to determine the RCS of clutter that falls within the beam.

The function `helperConicalBeamLimits` is provided to compute the extent of the beam footprint in range-Doppler space for a simple symmetric conical beam. The figure below shows how the min and max Doppler limits of the beam footprint change with azimuth angle relative to broadside (sometimes referred to as ground-plane squint angle) for two different dive angles. Use a beam with a 10 degree beamwidth pointed down at 30 degrees.

```
beamw = 10; % deg
dep = 30;   % deg
helperBeamDopplerLimitsChart(freq,alt,speed,beamw,dep)
```
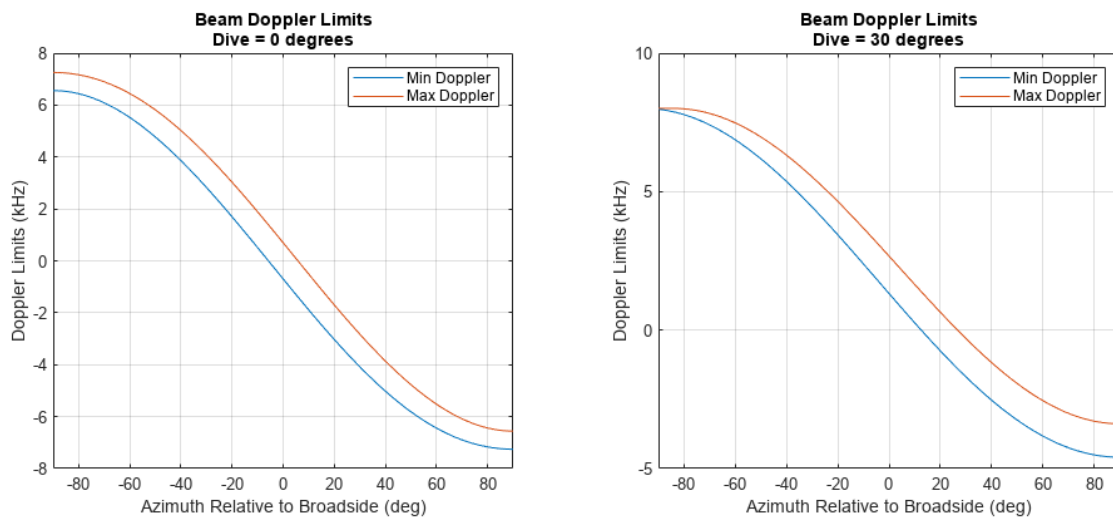

The above figure illustrates that, with a level flight path, the beam always covers the same range of Doppler frequencies, and the broadside direction centers the beam on 0 Hz. With a 30 degree dive angle, the extent of the beam in Doppler varies depending on azimuth pointing.

The range and Doppler limits of the beam footprint can be used to extract the surface RCS that falls within the beam. Let the radar beam point in the broadside direction at the 30 degree depression angle specified earlier.

```
az = 0; % deg
[R1,R2,d1,d2] = helperConicalBeamLimits(freq,alt,speed,dive,beamw,az,dep)
```

```
R1 = 5.2303e+03

R2 = 7.0986e+03

d1 = -2.6652

d2 = 1.3875e+03
```

R1 and R2 are the min and max slant ranges to the beam footprint (in meters), and d1 and d2 are the min and max Doppler frequencies of the footprint (in Hz).

Plot the surface RCS again, showing the region within these range and Doppler limits.

```
helperPlotSurfaceRCS(R,dop,rcs,R1,R2,d1,d2)
```
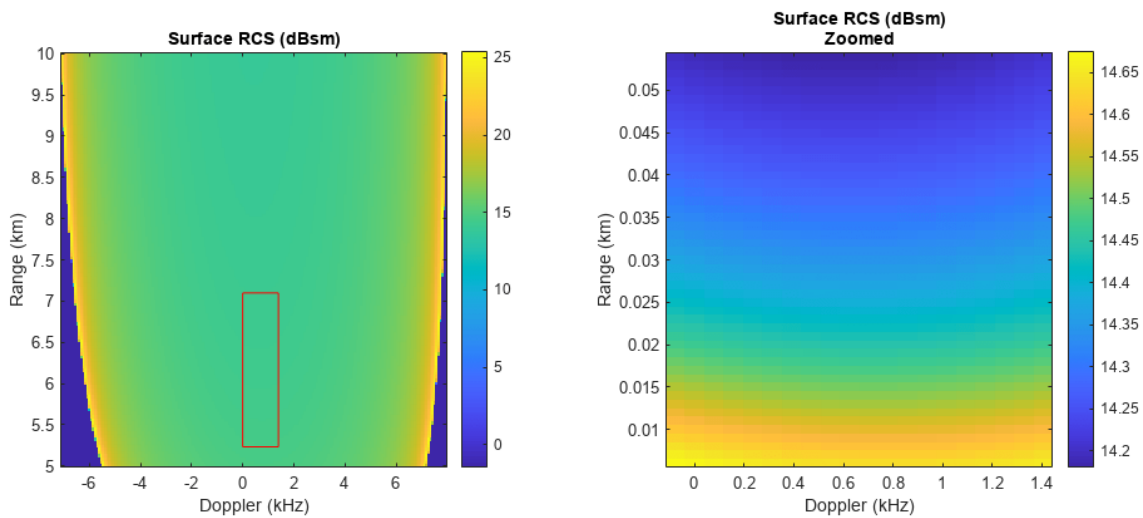

In this nice broadside case, the surface RCS within the beam shows only a slight variation with Doppler and varies only by about 0.4 dBsm across the beam.

Now try an off-broadside beam geometry. Set the azimuth relative to broadside to -60 degrees (pointing towards the direction of motion) and plot the results again.

```
az = -60; % deg
[R1,R2,d1,d2] = helperConicalBeamLimits(freq,alt,speed,dive,beamw,az,dep);
helperPlotSurfaceRCS(R,dop,rcs,R1,R2,d1,d2)
```
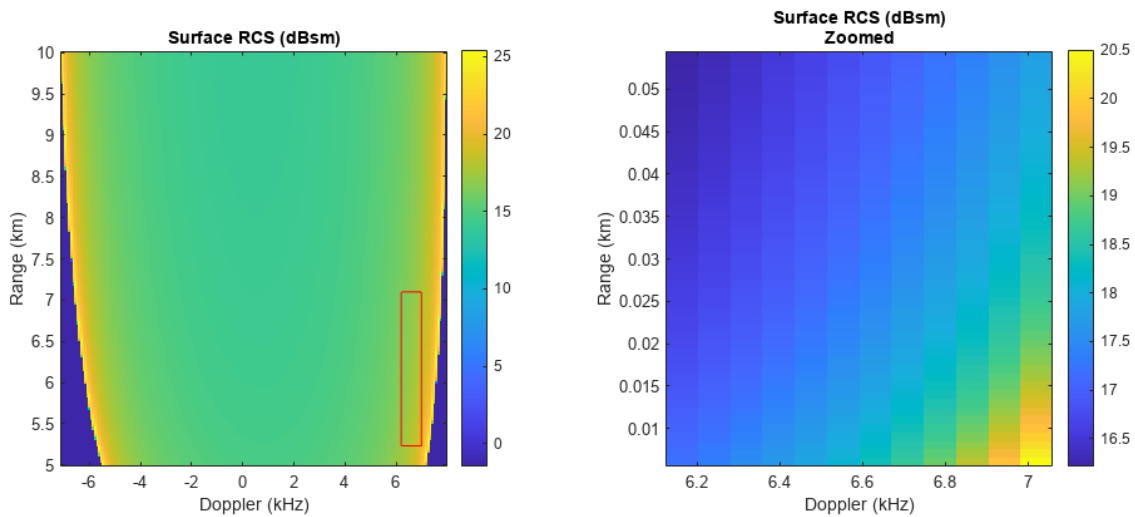
In this squinted case, the surface RCS varies significantly with both range and Doppler, and shows about a 4 dBsm variation across the beam. Such variation can cause trouble for detection algorithms and gain control.

Doppler Bandwidth

The total Doppler bandwidth of clutter at a given range is an important metric in applications such as receiver chain design, PRF selection, and MTI filter design. The range-Doppler map of surface RCS can be used to find the Doppler extent of clutter in each range bin.

Loop over range bins and, for each range, find the first and last Doppler bins with nonzero RCS. Estimate the bandwidth as the number of bins covered times the Doppler resolution. This approximation is a baseline that can be expanded upon to incorporate information about things like Doppler sidelobes and intrinsic clutter motion.

```
bandw = zeros(1,numel(R));
for gIdx = 1:numel(R)
    I1 = find(rcs(gIdx,:),1,'first');
    I2 = find(rcs(gIdx,:),1,'last');
    bandw(gIdx) = (I2 - I1 + 1)*dopres;
end
```

Find a quadratic fit and plot the results.

```
pf = polyfit(R,bandw,2);
figure
plot(bandw/1e3,R/1e3)
hold on
plot(polyval(pf,R)/1e3,R/1e3);
hold off
xlabel('Bandwidth (kHz)')
ylabel('Range (km)')
legend('Bandwidth Estimate','Quadratic Fit','Location','northwest')
set(gca,'XLimitMethod','tight')
```
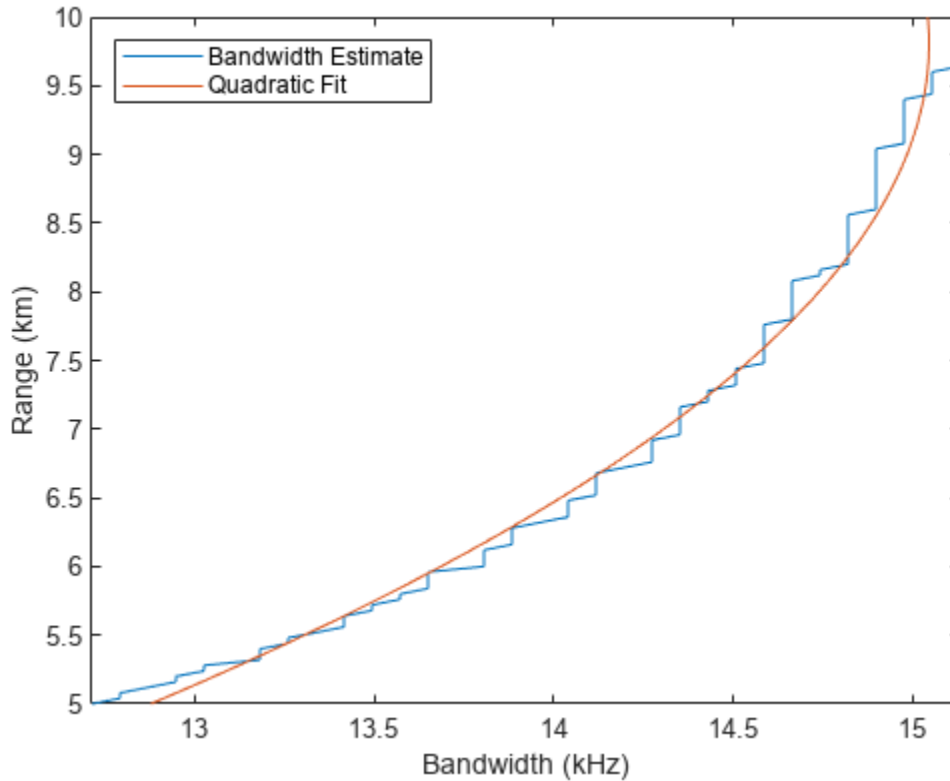
Doppler Wrapping

Coherent radar pulses are collected at a constant rate, known as the pulse-repetition frequency (PRF), over the course of a coherent processing interval (CPI). The pulses are stored in a matrix sometimes referred to as a data cube or phase history (PH), which gives a range-time map of received energy. Doppler processing is implemented in modern digital systems with a slow-time (across pulses) fast Fourier transform (FFT). As such, the Doppler spectrum of the return signal is subject to wrapping around a frequency interval of length PRF. This means that the apparent distribution of clutter returns in Doppler may be quite complex, and it's possible for a pixel of the RDM to contain return from multiple range-Doppler cells at congruent Doppler frequencies.

```
helperDopplerWrappingExample
```
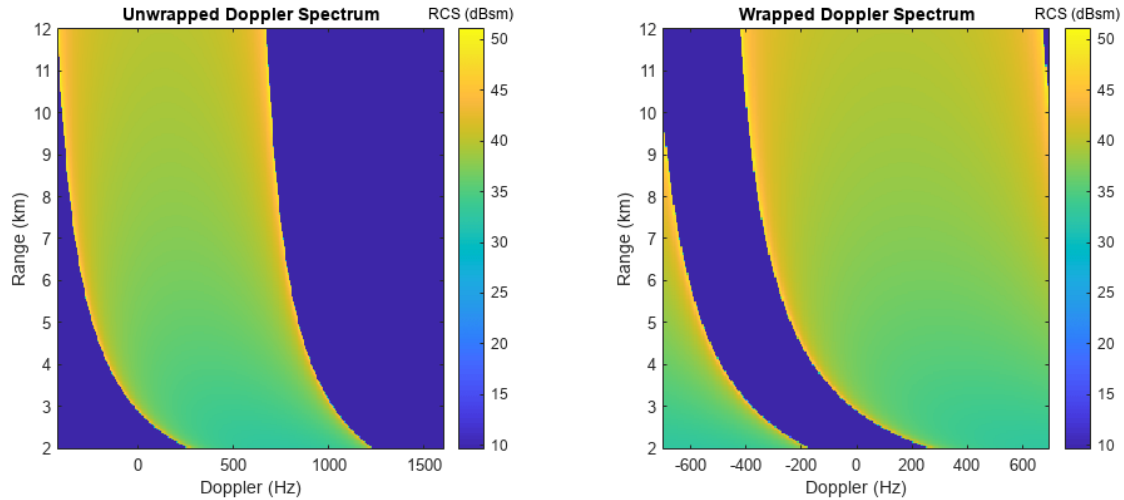
It can be useful to examine the unwrapped Doppler spectrum of clutter. For example, to see how various fast-time filters in the receiver chain might affect the clutter return signal. It is also useful to examine the wrapped Doppler spectrum to see how much total clutter return appears in a given pixel of the RDM. The `clutterSurfaceRangeDopplerRCS` function can do both. Doppler wrapping is enabled by specifying the number of Doppler bins to use via the optional `NumDopplerBins` parameter. No wrapping due to ambiguous ranges is performed.

Start by specifying the number of Doppler bins to use.

```
numdop = 128;
```

Find the PRF that is implied by this specification using the Doppler resolution that was specified earlier.

```
prf = dopres*numdop
```

```
prf = 9984
```

Call the function as before, but this time specify the number of Doppler bins to use. The resulting clutter RCS will be centered about 0 Hz and extend over the interval $\left[-\frac{\text{PRF}}{2}, \frac{\text{PRF}}{2}\right]$.

```
[rcs,dop] = clutterSurfaceRangeDopplerRCS(nrcs,R,freq,dopres,alt,speed,dive,'NumDopplerBins',numc
```

Plot the result.

```
helperPlotSurfaceRCS(R,dop,rcs)
```
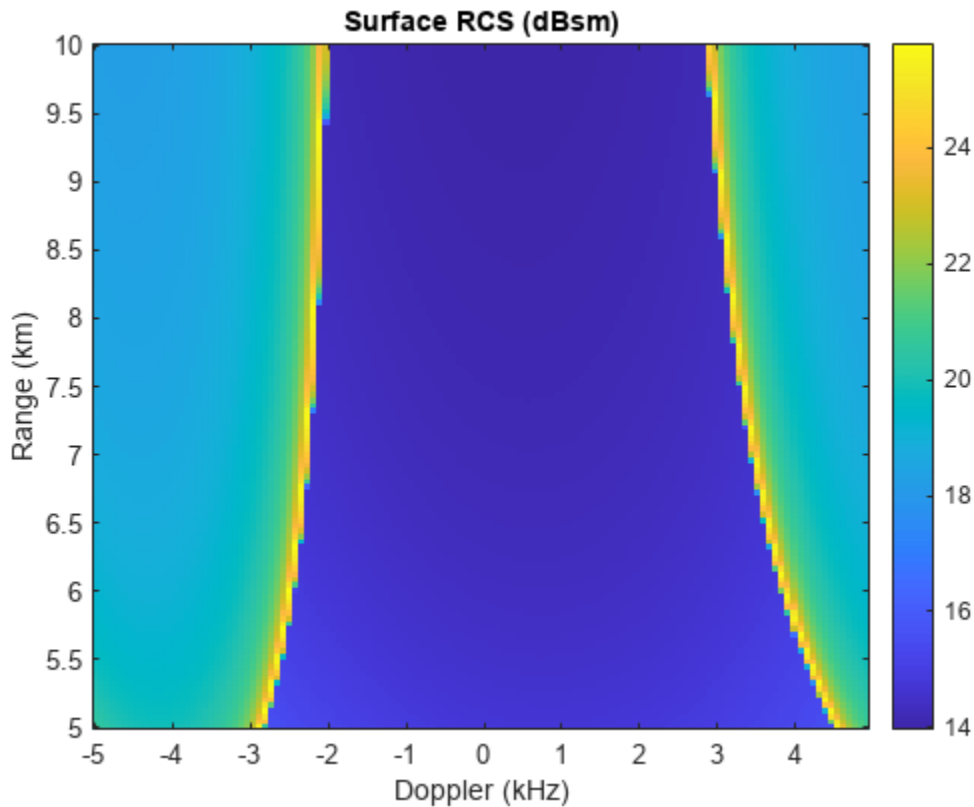
Since the PRF is lower than the minimum Doppler bandwidth over the range swath and because no limiting by the beam is considered here, the clutter return will fill the entire RDM. Ridges corresponding to the min and max unwrapped Doppler are still visible.

Analyze the Detectability of Surface Targets

In this section, you will see how to use `clutterSurfaceRangeDopperRCS` to make a rough determination of the detectability of a surface target based on its RCS. Assume that the beam is pointing at the target and that the gain does not vary significantly over a resolution cell. With this assumption, the signal-to-clutter ratio (SCR) can be found from the ratio of target RCS to clutter RCS in the resolution cell that contains the target.

Start by defining collection parameters for a typical airborne pulse-Doppler radar system. Use a 10 GHz frequency, a range resolution of 80 m, and a 31 Hz Doppler resolution. The range swath will start at 10 km and cover 400 range gates.

```
freq = 10e9;        % Hz
rngres = 80;        % m
dopres = 31;        % Hz
startrange = 10e3;  % m
numrange = 400;
R = startrange + (0:numrange-1)*rngres;
```

Position the radar at an altitude of 4 km above the origin, flying horizontally in the +X direction at 240 m/s.

```
alt = 4e3;   % m
speed = 240; % m/s
dive = 0;    % deg
```

Now define the target. Place it on the surface at a range of 14 km in the broadside direction. Let the target have an RCS of 6 dBsm.

```
tgtrng = 14e3; % m
tgtgndrng = sqrt(tgtrng^2-alt^2);
tgtpos = [0 tgtgndrng 0];
tgtrcs = 6; % dBsm
```

Calculate the Doppler frequency of the target.

```
rdrpos = [0 0 alt];
rdrvel = speed*[cosd(dive) 0 -sind(dive)];
lambda = freq2wavelen(freq);
tgtdop = 2/lambda*dot(rdrvel,tgtpos-rdrpos)/tgtrng
```

```
tgtdop = 0
```

Since the target lies in the broadside direction and the radar is not diving, the Doppler frequency of the target is 0 Hz.

Compute the surface RCS. Use the same reflectivity model as earlier.

```
graze = grazingang(alt,R,"Flat");
nrcs = refl(graze,freq);
[rcs,dop] = clutterSurfaceRangeDopplerRCS(nrcs,R,freq,dopres,alt,speed,dive);
```

Recall that the surface RCS in each range-Doppler cell returned by `clutterSurfaceRangeDopplerRCS` includes clutter return from the full 360 degrees of azimuth. As such, with the assumption that the surface returns are limited by a beam that only intersects the resolution cell on one side of the radar velocity vector, the surface RCS within the beam is half of the value returned by `clutterSurfaceRangeDopplerRCS`. Divide the RCS by two now.

```
rcs = rcs/2;
```

Extract the surface RCS in the 0 Hz Doppler bin.

```
rcs0Hz = rcs(:,dop==0);
```

Find the range bin containing the target and the surface RCS at the target range. Estimate the SCR (in dB) by finding the difference between the target and surface RCS.

```
[~,tgtrngbin] = min(abs(tgtrng-R));
surfrcs = pow2db(rcs0Hz(tgtrngbin));
tgtscr = tgtrcs - surfrcs
```

```
tgtscr = -2.1058
```

With a negative SCR, the target is unlikely to be detected. Now try decreasing the range resolution to 30 m and repeat the experiment.

```
rngres = 30; % m
R = startrange + (0:numrange-1)*rngres;
```

```
graze = grazingang(alt,R,"Flat");
nrcs = refl(graze,freq);
[rcs,dop] = clutterSurfaceRangeDopplerRCS(nrcs,R,freq,dopres,alt,speed,dive);
rcs = rcs/2;
rcs0Hz_highRes = rcs(:,dop==0);
[~,tgtrngbin] = min(abs(tgtrng-R));
surfrcs = pow2db(rcs0Hz_highRes(tgtrngbin));
tgtscr = tgtrcs - surfrcs
```

tgtscr = 2.1536

With the improved range resolution, the size of the range-Doppler cell on the ground is smaller, resulting in less reflected power from the surface, and the target may be detected. This assumes the entire target falls within one range resolution cell and that range straddling is not significant. Use the provided helper function to plot a visualization of the surface RCS at both range resolutions along with the target.

```
helperPlotSurfTgtRCS(R,rcs0Hz,rcs0Hz_highRes,tgtrng,tgtrcs)
```
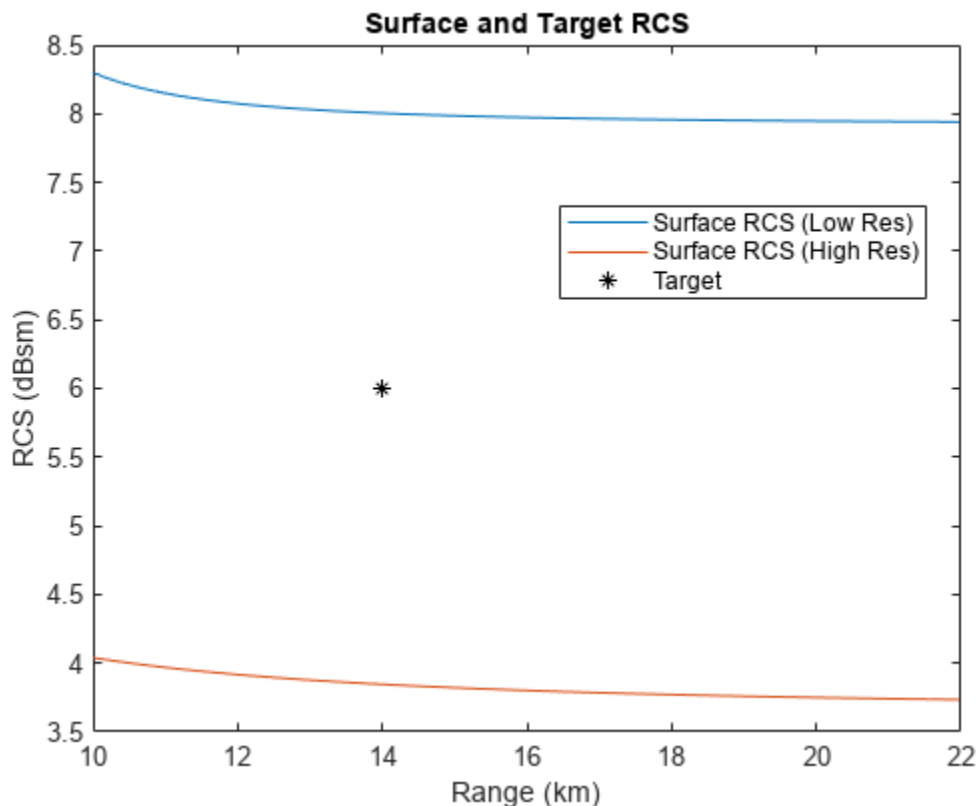

Estimate Clutter Power and Returns

In this section you will see how to quickly generate clutter returns using the clutter RCS information along with an estimate of the centroids of the range-Doppler cells on the surface. You will treat each centroid as a point target with an RCS equal to that of the corresponding cell, use the radar equation to find the received clutter power in each cell, then construct the slow-time signal with phase noise and thermal noise.

Compute Clutter Power

Start by defining the radar system parameters. Use a 2.4 GHz frequency, 80 m range resolution, a PRF of 6 kHz, and 128 Doppler bins (which will correspond to 128 pulses). The `NumDopplerBins` parameter will be used again to get a DC-centered Doppler spectrum.

```
freq = 2.4e9; % Hz
rngres = 80;  % m
prf = 6e3;    % Hz
numdop = 128;
```

Find the Doppler resolution of this system.

```
dopres = prf/numdop;
```

Place the radar at an altitude of 4 km, traveling at 100 m/s with a slight 10 degree dive.

```
alt = 4e3;   % m
speed = 100; % m/s
dive = 10;   % deg
```

Define a range swath that starts at the nadir point and covers 120 gates.

```
startrange = alt;
numrange = 120;
R = startrange + (0:numrange-1)*rngres;
```

This particular workflow requires that the clutter bandwidth not exceed the PRF, which can be checked with the provided helper function.

```
[d1,d2] = helperSurfaceDopplerLimits(freq,alt,speed,dive,R(end));
d2 - d1 < prf

ans = logical
   1
```

If the Doppler bandwidth does exceed the PRF, Doppler wrapping could be performed after calculating the clutter power in each cell of the unwrapped spectrum.

Calculate the NRCS of the surface in each range bin using the reflectivity model created earlier.

```
graze = grazingang(alt,R,"Flat");
nrcs = refl(graze,freq);
```

In order to generate a more realistic clutter signal, enable Weibull speckle on the reflectivity model using a shape parameter of 0.8, and generate a new instance of speckle for each Doppler bin.

```
release(refl)
refl.Speckle = "Weibull";
refl.SpeckleShape = 0.8;
spck = zeros(numrange,numdop);
for ind = 1:numdop
    [~,spck(:,ind)] = refl(graze,freq);
end
```

Compute the surface RCS for the given parameters. Multiply-in the speckle values, and once again divide the returned RCS by two to get the RCS on one side of the radar velocity vector.

```
[rcs,dop] = clutterSurfaceRangeDopplerRCS(nrcs,R,freq,dopres,alt,speed,dive,'NumDopplerBins',num
rcs = rcs.*spck;
rcs = rcs/2;
```

Define the radar look direction. Point the radar in the broadside direction (90 degrees, relative to the velocity vector) and down by 30 degrees.

```
azLook = 90;  % deg, relative to velocity vector
elLook = -30; % deg
```

The `helperRangeDopplerCellCenters` function is provided to find the coordinates of the range-Doppler cell centers on the surface, which approximates the power centroid for each cell. This function assumes the radar is positioned above the origin and is traveling in the +X direction. The returned coordinate values will be NaN for cells whose centers do not intersect the surface.

```
lambda = freq2wavelen(freq);
[cx,cy] = helperRangeDopplerCellCenters(R,dop,lambda,alt,speed,dive);
```

Plot the computed centroid positions using the provided helper function.

```
helperPlotCellCentroids(cx,cy)
```
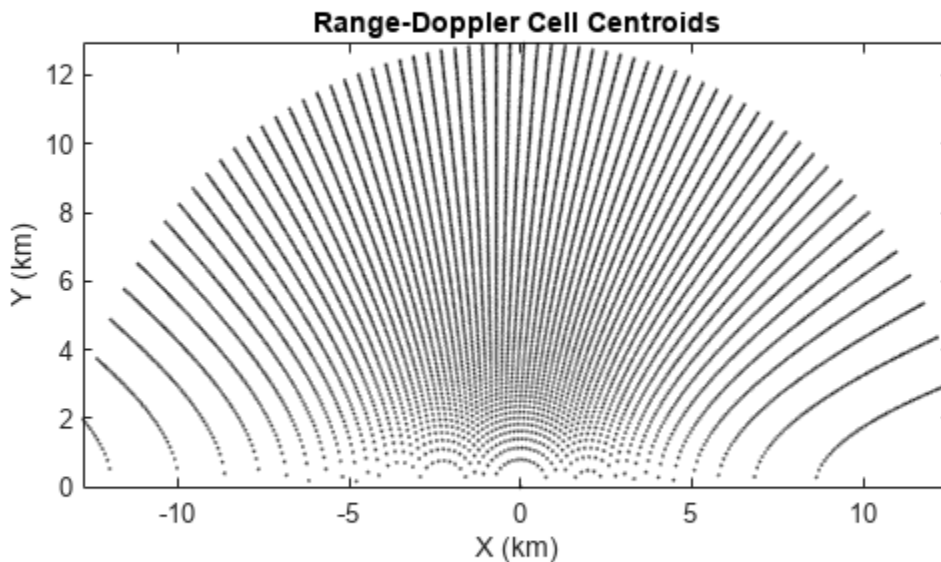

Only the cells in the +Y half of the plane are considered, since that is the direction the radar is pointed.

Find the lines of sight (LOS) from the radar to the centroids, convert them to the sensor frame, and get the angles to each cell in the sensor frame.

```
sensorFrame = rotz(azLook)*roty(-elLook);

% Lines of sight to centroids in scenario frame
losx = cx;
losy = cy;
losz = -alt;

% Convert to sensor frame
[losx,losy,losz] = deal(sensorFrame(1,1)*losx+sensorFrame(2,1)*losy+sensorFrame(3,1)*losz,...
                        sensorFrame(1,2)*losx+sensorFrame(2,2)*losy+sensorFrame(3,2)*losz,...
                        sensorFrame(1,3)*losx+sensorFrame(2,3)*losy+sensorFrame(3,3)*losz);

% Get az/el angles to centroids
[az,el,~] = cart2sph(losx,losy,losz);
az = rad2deg(az);
el = rad2deg(el);
```

Create a 14-by-14 uniform rectangular array (URA) for computing the antenna gain in the direction of each range-Doppler cell.

```
array = phased.URA([14 14],lambda/2);
```

Create a `phased.ArrayResponse` System object for getting the magnitude response of the array in the direction of each cell centroid. Only input the valid (non-NaN) centroids to the array response and express the results in dB.

```
antresp = phased.ArrayResponse('SensorArray',array);
nonnan = ~isnan(az) & ~isnan(el);
antGain = zeros(numrange,numdop);
antGain(nonnan) = antresp(freq,[az(nonnan) el(nonnan)].');
antGain = mag2db(abs(antGain));
```

The radar equation returns signal-to-noise (SNR) ratios, so calculate the noise power now in order to back out the received clutter power. Calculate the pulse width assuming a simple rectangular pulse. Use 290 K for the noise temperature, then find the noise power in dBW.

```
tau = range2time(rngres);
Ts = 290;
noisePwr = 10*log10(physconst('Boltzmann')*Ts/tau);
```

Use a 10 kW transmit power.

```
txPwr = 10e3;
```

Loop over the range-Doppler cells, and use `radareqsnr` and the noise power computed earlier to get the expected clutter power in each cell.

```
clutPwr = -inf(numrange,numdop);
for rIdx = 1:numrange
    for dIdx = 1:numdop
        if rcs(rIdx,dIdx) > 0
            clutPwr(rIdx,dIdx) = radareqsnr(lambda,R(rIdx),txPwr,tau,'RCS',rcs(rIdx,dIdx),'Gain'
        end
    end
end
```

Plot the results.

```
imagesc(dop/1e3,R/1e3,clutPwr)
set(gca,'ydir','normal')
xlabel('Doppler (kHz)')
ylabel('Range (km)')
title('Power of Surface Reflection (dBW)')
colorbar
```
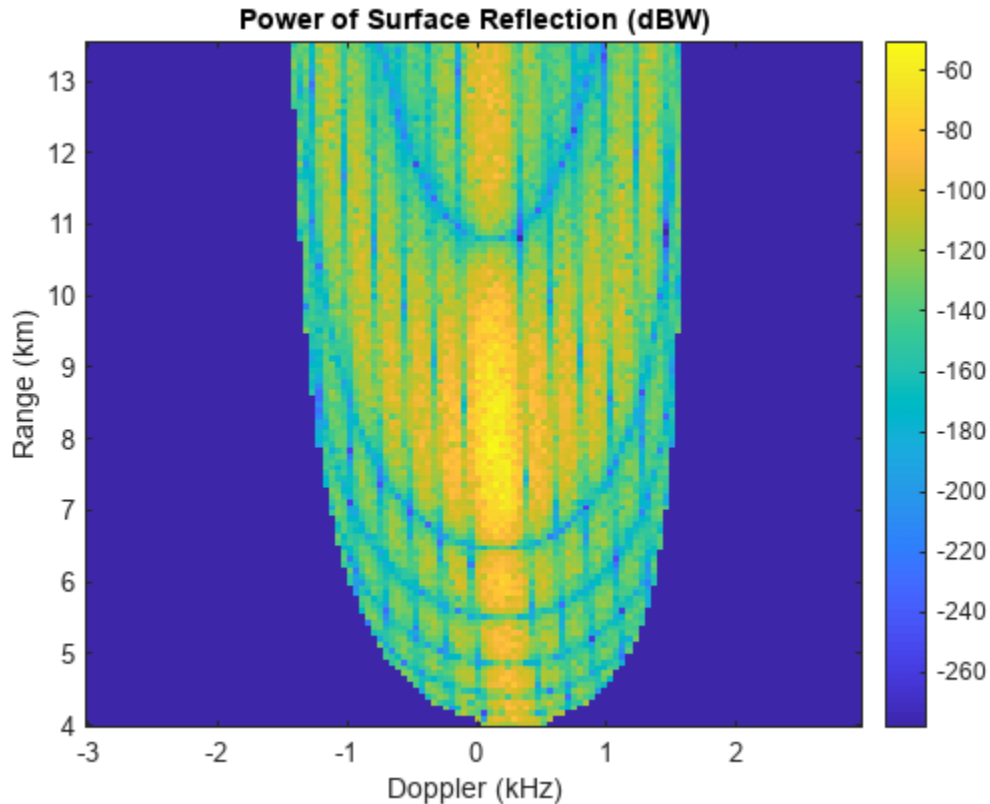

The antenna lobe structure and the effect of the added speckle can be seen clearly in the map of clutter power. Note that this result shows how the power received by reflections off the surface is distributed in range and Doppler but does not include signal processing gains. Speckle could be omitted to get an average clutter power instead.

Plot the cell centroids again, but this time color code them to see how the returned power is distributed in Cartesian space.

```
helperPlotCellCentroids(cx,cy,clutPwr)
```

Range-Doppler Cell Centroids
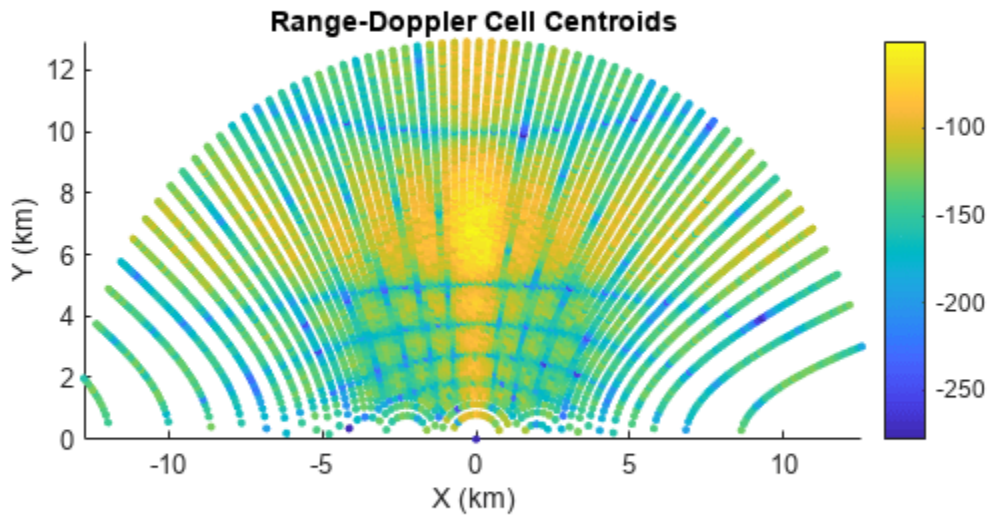
Simulate Clutter Returns

A simple clutter return signal can be generated by constructing a slow-time signal with the appropriate power and phase, and adding Doppler sidelobes and thermal noise. Start by converting the clutter power to magnitude values.

```
clutMag = db2mag(clutPwr);
```

Use the provided helper function to generate a slow-time signal from the computed magnitude spectrum. This function adds Doppler sidelobes by applying frequency and phase offsets to the signal at each Doppler frequency.

```
PH = helperFormSlowTimeSignal(clutMag,prf,R);
```

Add thermal noise. Adjust the noise power to account for the number of independent channels of the array that are summed together.

```
numchan = getNumElements(array);
noisePwrSum = noisePwr + 10*log10(numchan);
ni = randn(size(PH));
nq = randn(size(PH));
PH = PH + db2mag(noisePwrSum)*(ni+1i*nq)/sqrt(2);
```

Form the DC-centered RDM

```
RDM = fftshift(fft(PH,[],2),2);
```

Plot the resulting RDM. The color scale extends from 20 dBW below the expected noise power in the RDM up to -10 dBW.

```
imagesc(dop/1e3,R/1e3,20*log10(abs(RDM)))
set(gca,'ydir','normal')
xlabel('Doppler (kHz)')
ylabel('Range (km)')
title('Predicted RDM (dBW)')
colorbar
clim([noisePwrSum-20+20*log10(numdop) -10])
```
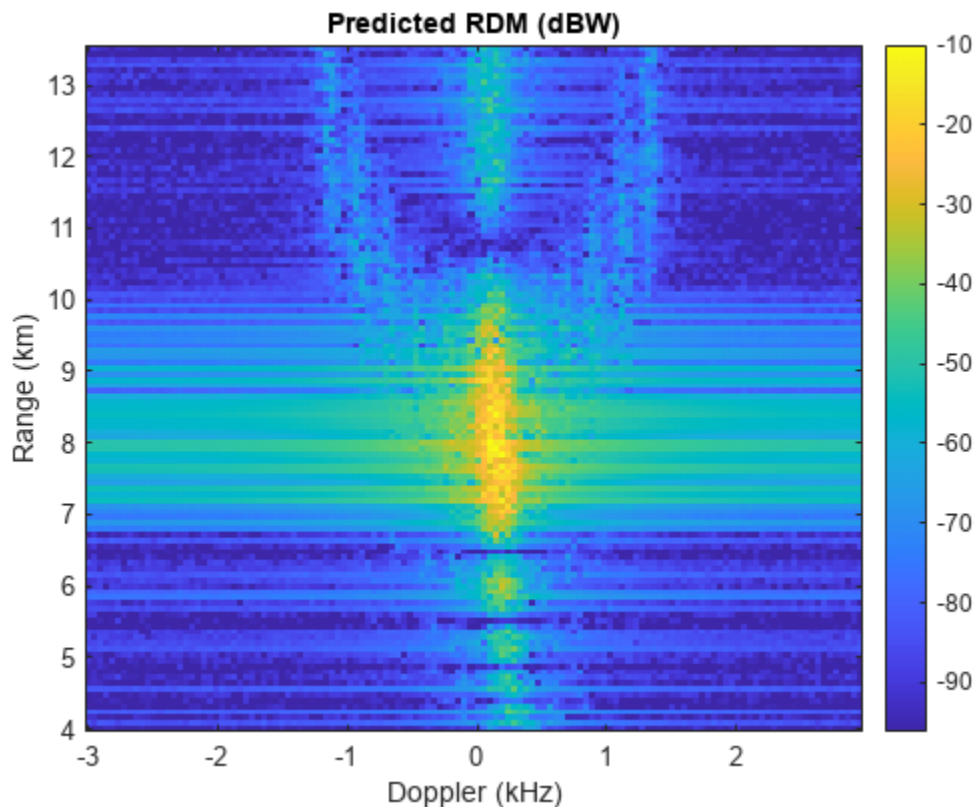

In addition to the Doppler sidelobes, fast-time effects could be accounted for by applying waveform information to each pulse and simulating range sidelobes.

Compare to Radar Scenario

The provided helper function `helperCompareToScenario` demonstrates how to collect clutter returns for the same scenario parameters using `radarScenario`. The radar scenario allows flexible clutter simulation with support for any of the available waveform types, realistic platform trajectories, and heterogeneous surfaces (terrain, reflectivity maps, curved-Earth, etc.).

The figure below shows the output that can be expected from the scenario simulation.

Simulated RDM (dBW)
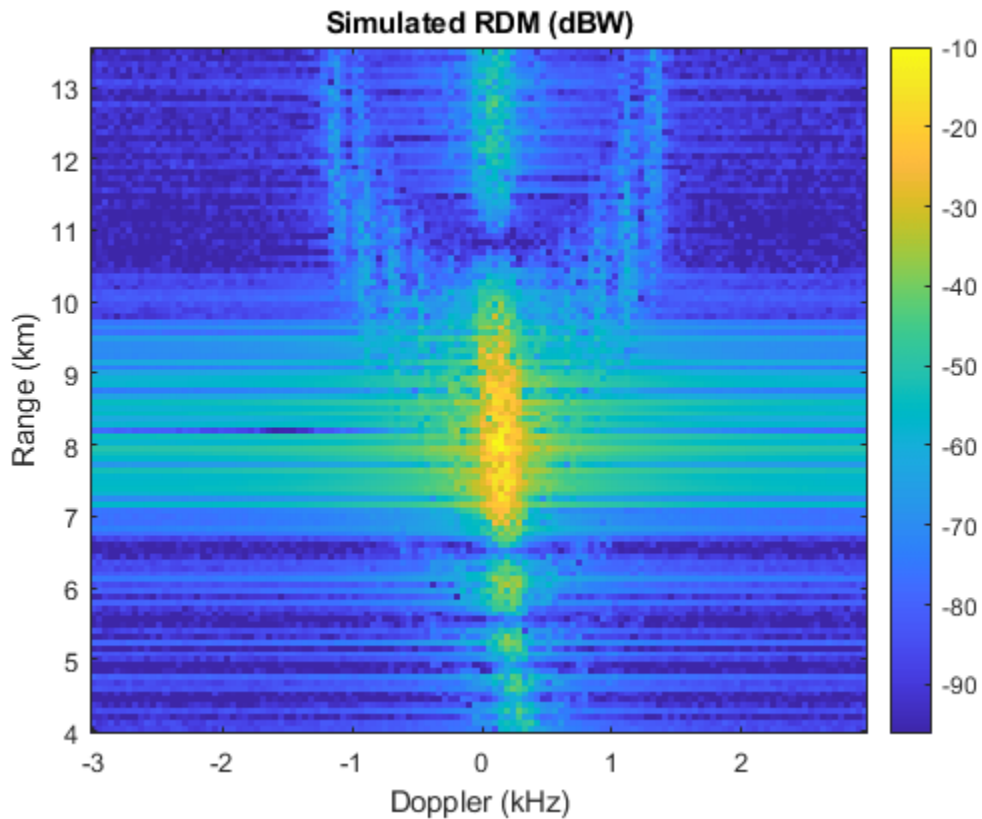
There is a close agreement between these two very different methods of clutter generation.

You can run the scenario simulation as part of this example by uncommenting the line below. The simulation takes 10 to 15 minutes on a machine with 64 GB of RAM and a 3 GHz processor.

```
%helperCompareToScenario(array,numdop,prf,refl,rngres,freq,azLook,elLook,txPwr,noisePwr,alt,speec
```

Conclusion

In this example you saw how to compute the RCS of a flat surface in range-Doppler space. You saw how to investigate the presentation of surface clutter when interrogated by a conical beam, and how to use the surface RCS in tandem with simple geometric calculations to perform a quick analysis of the detectability of surface targets based on their RCS. Finally, you saw how to quickly estimate clutter power and simulate clutter returns using approximate cell centroids and the radar equation.

Supporting Functions

helperSurfaceDopplerLimitsChart

```
function helperSurfaceDopplerLimitsChart(freq,alt)
% Create a chart to visualize the Doppler limits of a flat surface.

spd = linspace(90,150,80);
maxRange = linspace(3e3,10e3,80);

dive = 0;
d1_0 = zeros(numel(maxRange),numel(spd));
```

```matlab
    d2_0 = d1_0;
    for spdIdx = 1:numel(spd)
        for rIdx = 1:numel(maxRange)
            [d1_0(rIdx,spdIdx),d2_0(rIdx,spdIdx)] = helperSurfaceDopplerLimits(freq,alt,spd(spdIdx),
        end
    end

    dive = 30;
    d1_30 = zeros(numel(maxRange),numel(spd));
    d2_30 = d1_30;
    for spdIdx = 1:numel(spd)
        for rIdx = 1:numel(maxRange)
            [d1_30(rIdx,spdIdx),d2_30(rIdx,spdIdx)] = helperSurfaceDopplerLimits(freq,alt,spd(spdIdx
        end
    end

    critR = alt/sind(dive);

    h = figure;
    subplot(1,2,1)
    contour(spd,maxRange/1e3,d2_0/1e3,[2 4 5 6 7 8 9],'black','showtext','on')
    title({'Maximum Doppler Frequency (kHz)';'Dive = 0 degrees'})
    xlabel('Radar Speed (m/s)')
    ylabel('Max Range (km)')
    subplot(1,2,2)
    contour(spd,maxRange/1e3,d2_30/1e3,[2 4 5 6 7 8 9],'black','showtext','on')
    title({'Maximum Doppler Frequency (kHz)';'Dive = 30 degrees'})
    xlabel('Radar Speed (m/s)')
    ylabel('Max Range (km)')
    line(xlim,[critR critR]/1e3,'color','red','linestyle','--')
    h.Position = h.Position + [0 0 560 0];
    set(gca,'XLimitMethod','tight')

end
```

helperPlotSurfaceRCS

```matlab
function helperPlotSurfaceRCS(R,dop,rcs,R1,R2,d1,d2)
% Plot the RCS of a surface in range-Doppler space. Optionally, zoom in on
% a specified interval.

h = figure;

if nargin > 3
    subplot(1,2,1)
end

imagesc(dop/1e3,R/1e3,10*log10(rcs))
set(gca,'ydir','normal')
xlabel('Doppler (kHz)')
ylabel('Range (km)')
title('Surface RCS (dBsm)')
colorbar

if nargin > 3
    patch([d1 d2 d2 d1]/1e3,[R1 R1 R2 R2]/1e3,'','FaceColor','none','EdgeColor','red')

    subplot(1,2,2)
```

```
        RIdx1 = find(R <= R1,1,'last');
        RIdx2 = find(R >= R2,1,'first');
        DIdx1 = find(dop <= d1,1,'last');
        DIdx2 = find(dop >= d2,1,'first');
        imagesc(dop(DIdx1:DIdx2)/1e3,(RIdx1:RIdx2)/1e3,10*log10(rcs(RIdx1:RIdx2,DIdx1:DIdx2)))
        set(gca,'ydir','normal')
        xlabel('Doppler (kHz)')
        ylabel('Range (km)')
        title({'Surface RCS (dBsm)';'Zoomed'})
        colorbar

        h.Position = h.Position + [0 0 560 0];
    end

end
```

helperBeamDopplerLimitsChart

```
function helperBeamDopplerLimitsChart(freq,alt,spd,bw,dep)
% Create a chart to visualize the Doppler limits of a beam.

az = linspace(-90,90,1e3);

dive = 0;
d1_0 = zeros(1,numel(az));
d2_0 = d1_0;
for ind = 1:numel(az)
    [~,~,d1_0(ind),d2_0(ind)] = helperConicalBeamLimits(freq,alt,spd,dive,bw,az(ind),dep);
end

dive = 30;
d1_30 = zeros(1,numel(az));
d2_30 = d1_30;
for ind = 1:numel(az)
    [~,~,d1_30(ind),d2_30(ind)] = helperConicalBeamLimits(freq,alt,spd,dive,bw,az(ind),dep);
end

h = figure;
subplot(1,2,1)
plot(az,d1_0/1e3,az,d2_0/1e3)
title({'Beam Doppler Limits';'Dive = 0 degrees'})
xlabel('Azimuth Relative to Broadside (deg)')
ylabel('Doppler Limits (kHz)')
legend('Min Doppler','Max Doppler')
grid on
set(gca,'XLimitMethod','tight')
subplot(1,2,2)
plot(az,d1_30/1e3,az,d2_30/1e3)
title({'Beam Doppler Limits';'Dive = 30 degrees'})
xlabel('Azimuth Relative to Broadside (deg)')
ylabel('Doppler Limits (kHz)')
grid on
legend('Min Doppler','Max Doppler')
set(gca,'XLimitMethod','tight')
h.Position = h.Position + [0 0 560 0];

end
```

helperConicalBeamLimits

```matlab
function [R1,R2,d1,d2] = helperConicalBeamLimits(freq,alt,spd,dive,bw,az,graze)
% Return the range and Doppler limits of a conical beam footprint.

if graze <= -bw/2
    R1 = inf;
    R2 = inf;
elseif graze <= bw/2
    R1 = alt/sind(graze+bw/2);
    R2 = inf;
elseif graze <= 90-bw/2
    R1 = alt/sind(graze+bw/2);
    R2 = alt/sind(graze-bw/2);
else
    R1 = alt;
    R2 = alt/sind(graze-bw/2);
end

lambda = freq2wavelen(freq);

azVel = az + 90; % use azimuth relative to velocity vector
squint = acosd(cosd(dive)*cosd(azVel)*cosd(graze) + sind(dive)*sind(graze));

if squint <= bw/2
    t1 = squint + bw/2;
    t2 = 0;
else
    t1 = squint + bw/2;
    t2 = squint - bw/2;
end

d1 = 2/lambda*spd*cosd(t1);
d2 = 2/lambda*spd*cosd(t2);

end
```

helperDopplerWrappingExample

```matlab
function helperDopplerWrappingExample
% Plot a demonstration of Doppler wrapping

alt = 1e3;
freq = 2e9;
prf = 1.4e3;
ndop = 256;
dopres = prf/ndop;
dive = 70;
speed = 120;

R = 2e3:40:12e3;
nrcs = ones(size(R));

[RCS1,dopbins1] = clutterSurfaceRangeDopplerRCS(nrcs,R,freq,dopres,alt,speed,dive);
[RCS2,dopbins2] = clutterSurfaceRangeDopplerRCS(nrcs,R,freq,dopres,alt,speed,dive,'NumDopplerBins
h = figure;
subplot(1,2,1)
```

```
imagesc(dopbins1,R/1e3,10*log10(RCS1))
set(gca,'ydir','normal')
title('Unwrapped Doppler Spectrum')
xlabel('Doppler (Hz)')
ylabel('Range (km)')
ch=colorbar;
ch.Title.String='RCS (dBsm)';
subplot(1,2,2)
imagesc(dopbins2,R/1e3,10*log10(RCS2))
set(gca,'ydir','normal')
title('Wrapped Doppler Spectrum')
xlabel('Doppler (Hz)')
ylabel('Range (km)')
ch=colorbar;
ch.Title.String='RCS (dBsm)';
h.Position = h.Position + [0 0 560 0];

end
```

helperPlotSurfTgtRCS

```
function helperPlotSurfTgtRCS(R,rcsLow,rcsHigh,tgtrng,tgtrcs)
% Plot RCS against range and annotate the target range and rcs.

plot(R/1e3,10*log10(rcsLow))
hold on
plot(R/1e3,10*log10(rcsHigh))
plot(tgtrng/1e3,tgtrcs,'*black')
hold off
xlabel('Range (km)')
ylabel('RCS (dBsm)')
legend('Surface RCS (Low Res)','Surface RCS (High Res)','Target','Location','best')
title('Surface and Target RCS')

end
```

helperSurfaceDopplerLimits

```
function [d1,d2] = helperSurfaceDopplerLimits(freq,alt,spd,dive,maxRange)
% Return the Doppler limits of a flat surface.

% Depression angle to max range
if maxRange < alt
    maxRangeDep = 90;
else
    maxRangeDep = 90 - acosd(alt/maxRange);
end

% Normalized maximum closing rate
if dive < maxRangeDep
    maxClosing = cosd(maxRangeDep-dive);
else
    maxClosing = 1;
end

% Normalized minimum closing rate
if -dive < maxRangeDep
    minClosing = -cosd(maxRangeDep+dive);
```

```matlab
else
    minClosing = -1;
end

% Factor in speed and convert to Doppler
lambda = freq2wavelen(freq);
d1 = 2/lambda*spd*minClosing;
d2 = 2/lambda*spd*maxClosing;

end
```

helperRangeDopplerCellCenters

```matlab
function [x,y] = helperRangeDopplerCellCenters(R,dop,lambda,alt,speed,dive)
% Return the x/y coordinates of the centers of a set of range-Doppler cells
% on the surface

R = R(:);
dop = dop(:).';

r = sqrt(R.^2-alt^2);
ctheta = (lambda*R.*dop/(2*speed) - alt*sind(dive))./(r*cosd(dive));
ctheta(r == 0,:) = 0;
ctheta(ctheta>1) = nan;
ctheta(ctheta<-1) = nan;
theta = acosd(ctheta);

x = r.*cosd(theta);
y = r.*sind(theta);

end
```

helperPlotCellCentroids

```matlab
function helperPlotCellCentroids(cx,cy,cdata)
% Plot cell centroids in Cartesian space. Optionally specify color data.

if nargin < 3
    plot(cx(:)/1e3,cy(:)/1e3,'.black','MarkerSize',1)
else
    scatter(cx(:)/1e3,cy(:)/1e3,10,cdata(:),'filled')
    colorbar
end
title('Range-Doppler Cell Centroids')
xlabel('X (km)')
ylabel('Y (km)')
axis equal
set(gca,'XLimitMethod','tight')
set(gca,'YLimitMethod','tight')

end
```

helperFormSlowTimeSignal

```matlab
function PH = helperFormSlowTimeSignal(specmag,prf,R)
% Form a time-domain signal from the magnitude spectrum

[numr,numd] = size(specmag);
```

```matlab
% Unshift magnitude spectrum
specmag = ifftshift(specmag,2);

% Initialize time domain signal
PH = zeros(size(specmag));

% Sample times
tslow = 0:1/prf:(numd-1)/prf;
tfast = 2*(R(:)-R(1))/299792458;
t = tslow + tfast;

% Randomized phase offsets for each bin
phi = rand(numr,numd)-1/2;

% Doppler axis
df = prf/numd;
dop = 0:df:(prf-df);

% Random Doppler offsets in each bin
fc = dop + (rand(numr,numd)-1/2)*df;

for ind = 1:numd
    % Entire signal at the current Doppler frequency
    y = specmag(:,ind).*exp(1i*2*pi*(fc(:,ind).*t+phi(:,ind)));

    % Accumulate
    PH = PH + y;
end

end
```

helperCompareToScenario

```matlab
function helperCompareToScenario(array,numdop,prf,refl,rngres,fc,azLook,elLook,txPwr,noisePwrDb,a
% Simulate clutter returns with radarScenario and plot the result

% Make scenario
scenario = radarScenario('UpdateRate',0,'StopTime',(numdop-1/2)/prf);
landSurface(scenario,'RadarReflectivity',refl);

% Make radar
c = physconst('lightspeed');
rdr = radarTransceiver('MountingAngles',[azLook,-elLook,0],'NumRepetitions',numdop);
rdr.TransmitAntenna.Sensor = array;
rdr.ReceiveAntenna.Sensor = array;
rdr.TransmitAntenna.OperatingFrequency = fc;
rdr.ReceiveAntenna.OperatingFrequency = fc;
rdr.Waveform.PRF = prf;
sampleRate = c/(2*rngres);
sampleRate = prf*round(sampleRate/prf); % adjust to match constraint with PRF
rdr.Receiver.SampleRate = sampleRate;
rdr.Waveform.SampleRate = sampleRate;
rdr.Waveform.PulseWidth = 2*rngres/c;
rdr.Transmitter.Gain = 0;
rdr.Transmitter.LossFactor = 0;
rdr.Transmitter.PeakPower = txPwr*sqrt(2);
rdr.Receiver.Gain = 0;
```

```matlab
    rdr.Receiver.LossFactor = 0;
    rdr.Receiver.NoiseMethod = "Noise power";
    rdr.Receiver.NoisePower = 10^(noisePwrDb/10);

    % Add radar platform
    platform(scenario,'Sensors',rdr,'Trajectory',kinematicTrajectory('Position',[0 0 alt],'Velocity'

    % Configure clutter generation
    clut = clutterGenerator(scenario,rdr,'UseBeam',false,'Resolution',rngres/2,'RangeLimit',R(end));
    ringClutterRegion(clut,0,sqrt(R(end)^2-alt^2),180,azLook);

    % Collect return signals and form sum beam
    iqsig = receive(scenario);
    PH = permute(sum(iqsig{1},2),[1 3 2]);

    % Form RDM
    RDM = fftshift(fft(PH,[],2),2);

    % Truncate range
    uar = c/(2*prf);
    Rall = 0:rngres:(uar-rngres);
    RDM = RDM(Rall>=R(1) & Rall<=R(end),:);

    % Plot results
    figure
    imagesc(dop/1e3,R/1e3,20*log10(abs(RDM)))
    set(gca,'ydir','normal')
    xlabel('Doppler (kHz)')
    ylabel('Range (km)')
    title('Simulated RDM (dBW)')
    colorbar
    numchan = getNumElements(array);
    noisePwrSum = noisePwrDb + 10*log10(numchan);
    clim([noisePwrSum-20+20*log10(numdop) -10])

end
```

Simulate a Maritime Radar PPI

This example shows how to simulate a plan position indicator (PPI) radar image for a rotating antenna array in a maritime environment. You will configure a radar scenario and spectral sea surface model, emulate a cargo ship with a cuboid extended target, generate return signals, and plot a PPI image of the result.

Configure the Radar Scenario

Set the RNG seed for repeatable results.

```
rng default
```

The scenario will consist of a large rotating uniform linear array (ULA) mounted in a fixed position above a sea surface. A large cuboid target will be placed on the sea surface.

Define the radar system parameters. Use an X-band radar at 10 GHz with a 1 degree azimuth beamwidth and 14 meter range resolution.

```
freq = 10e9; % Hz
azbw = 1;    % deg
rngres = 14; % m
```

Calculate the number of uniformly-spaced elements required to achieve the desired beamwidth, and create a `phased.ULA`. Set the `BackBaffled` property of the array element to true so that only returns from the pointing direction are included.

```
sinc3db = 0.8859;
numelems = round(2*sinc3db/deg2rad(azbw));
lambda = freq2wavelen(freq);
array = phased.ULA(numelems,lambda/2);
array.Element.BackBaffled = true;
```

Let the radar rotate at 50 rpm and express this in deg/sec. Calculate the rotation period from this rate.

```
rotrate = 50/60*360;     % deg/sec
rotperiod = 360/rotrate; % sec
```

Calculate the whole number of pulses required to minimally cover 360 degrees with the specified azimuth beamwidth. Use the rotation period to find the required pulse repetition frequency (PRF).

```
npulses = ceil(360/azbw);
prf = npulses/rotperiod; % Hz
```

The fast-time sampling rate must be a whole multiple of the PRF. Use the desired range resolution to find the required sampling rate then adjust to match this constraint with the PRF. The radar will use a short pulse with one sample per resolution cell.

```
sampleRate = 1/range2time(rngres); % Hz
sampleRate = prf*round(sampleRate/prf);
rngres = time2range(1/sampleRate); % m
```

Now create the `radarTransceiver` and set the required parameters.

```
rdr = radarTransceiver;
rdr.TransmitAntenna.OperatingFrequency = freq;
```

```
rdr.ReceiveAntenna.OperatingFrequency = freq;
rdr.TransmitAntenna.Sensor = array;
rdr.ReceiveAntenna.Sensor = array;
rdr.Waveform.PRF = prf;
rdr.Receiver.SampleRate = sampleRate;
rdr.Waveform.SampleRate = sampleRate;
rdr.Waveform.PulseWidth = 1/sampleRate;
```

Create a `radarScenario` and set the `UpdateRate` to 0 to let the update rate be derived from the `radarTransceiver` configuration.

```
scenario = radarScenario('UpdateRate',0);
```

Now configure the sea surface. The surface will be defined on a square region of size 2 km-by-2 km.

```
seaLength = 2e3; % m
```

The `seaSpectrum` object is used to define the spectral model for the surface. The `Resolution` property defines the spacing of the underlying grid in the spatial domain, and the `WaveVectorSpacing` defines the spacing of the grid in the frequency domain. The resolution must be a factor of the length of the surface, so specify a desired surface resolution of 1/4th of the radar's range resolution (which is sufficient to capture the shape of the waves), then adjust to match this constraint. Set the `WaveVectorSpacing` to 2π divided by the length of the surface.

```
seaRes = rngres/4; % m
seaRes = seaLength/round(seaLength/seaRes);
spec = seaSpectrum('Resolution',seaRes,'WaveVectorSpacing',2*pi/seaLength);
```

The `surfaceReflectivitySea` object is used to specify the reflectivity model that will be associated with the surface. Use the NRL reflectivity model with a sea state of 3 for horizontal polarization.

```
refl = surfaceReflectivitySea('Model','NRL','SeaState',3,'Polarization','H');
```

Create the 2 km-by-2 km sea surface using the reflectivity and spectral models. Specify the wind speed as 10 m/s, and the wind direction as 0 degrees so the wind is blowing in the +X direction.

```
windSpeed = 10; % m/s
windDir = 0;    % deg
bdry = [-1 1;-1 1]*seaLength/2;
seaSurface(scenario,'Boundary',bdry,'RadarReflectivity',refl,'SpectralModel',spec,'WindSpeed',win
```

Create the radar platform. Use the `kinematicTrajectory` object to mount the radar 24 meters above sea level, and use the `AngularVelocity` parameter to specify the rotation rate about the Z axis in radians per second.

```
rdrHeight = 24; % m
traj = kinematicTrajectory('Position',[0 0 rdrHeight],'AngularVelocitySource','Property','Angula
platform(scenario,'Sensors',rdr,'Trajectory',traj);
```

Now define the cuboid target. Specify the target dimensions, total RCS, position, heading, and speed.

```
tgtdims = struct('Length',120,'Width',18,'Height',22,'OriginOffset',[0 0 0]); % m
tgtrcs = 40;  % dBsm
tgtpos = [seaLength/3 seaLength/16 5];
tgthdg = -30; % deg
tgtspd = 8;   % m/s
```

Form the target velocity vector, create a kinematic trajectory, and add the target platform to the scene.

```
tgtvel = tgtspd*[cosd(tgthdg) sind(tgthdg) 0];
tgttraj = kinematicTrajectory('Position',tgtpos,'Velocity',tgtvel,'Orientation',rotz(tgthdg).');
platform(scenario,'Trajectory',tgttraj,'Signatures',rcsSignature('Pattern',tgtrcs),'Dimensions',
```

Finally, enable clutter generation for the radar by calling the `clutterGenerator` method on the scenario. The 3 dB beamwidth of the beam is used for clutter generation by default. In order to capture more of the surface return, disable this by setting `UseBeam` to false, and instead use a `RingClutterRegion` with an azimuth coverage 16 times that of the beam to encompass some sidelobes. Set the clutter resolution to half the range resolution. Returns will only be considered from a minimum ground range of 200 meters out to the maximum ground range of the surface. The azimuth center can be 0 to indicate that the region is centered about the +X direction. The `ringClutterRegion` method returns a handle to this region so that the parameters can be modified during the simulation.

```
clut = clutterGenerator(scenario,rdr,'UseBeam',false,'Resolution',rngres/2);
minGndRng = 200;
maxGndRng = seaLength/2;
azCov = 16*azbw;
azCen = 0;
reg = ringClutterRegion(clut,minGndRng,maxGndRng,azCov,azCen);
```

Use the provided helper function to create a visualization of the scenario.

```
helperPlotScenarioGeometry(rdrHeight,seaLength,array,freq,tgtpos)
```

Scenario Geometry
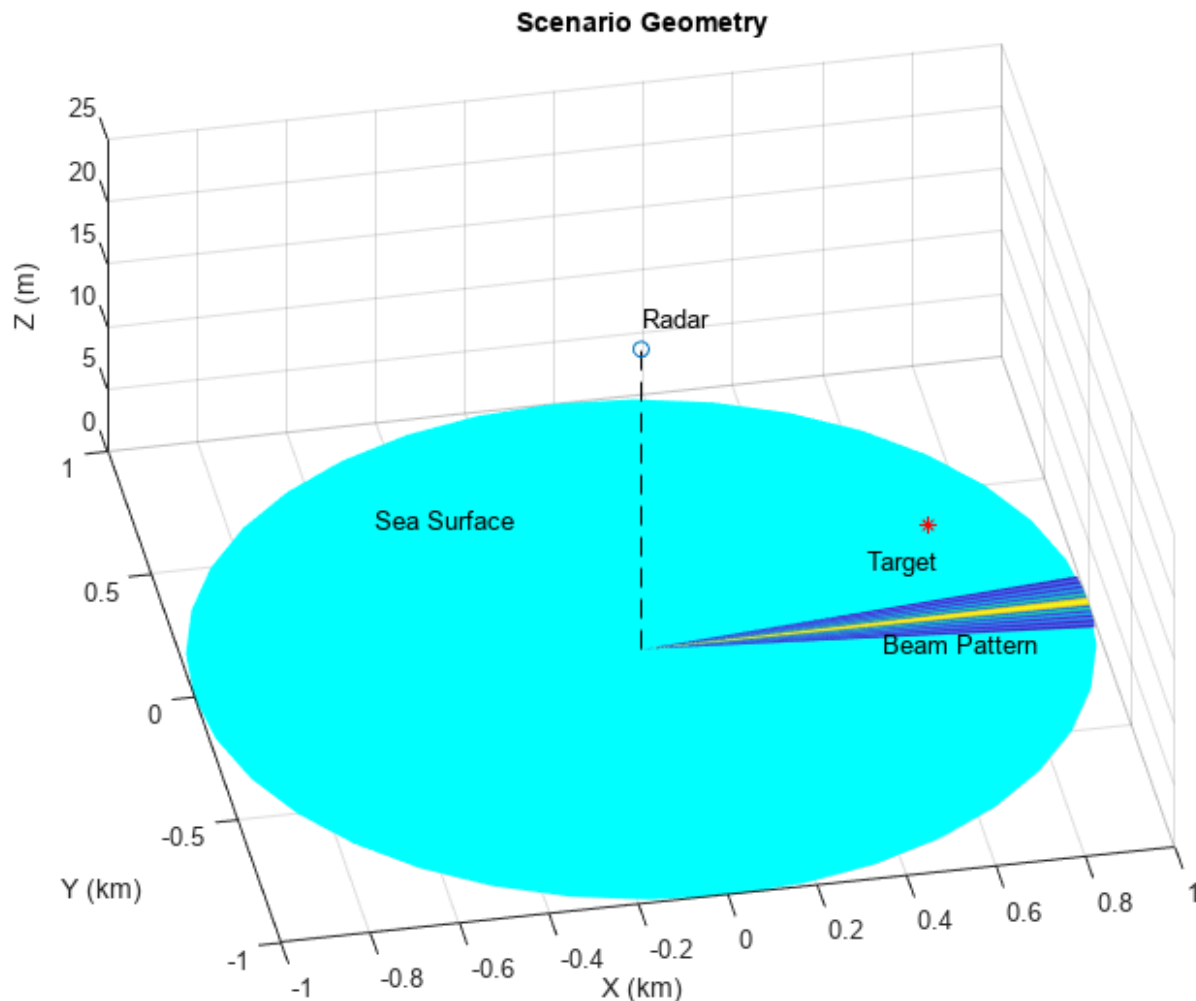
Run the Simulation and Collect Returns

Each frame of the simulation will generate a range profile for one azimuth pointing direction. After receiving the raw IQ data on each frame, a `phased.RangeResponse` object will be used to perform matched filtering. Create the response object now, specifying the sample rate used, and get the required matched filter coefficients from the radar's waveform object.

```
resp = phased.RangeResponse('RangeMethod','Matched filter','SampleRate',sampleRate);
mfc = getMatchedFilter(rdr.Waveform);
```

Specify how much of the full 360 degrees you would like to simulate. The scan starts at 0 degrees azimuth, and 45 degrees of coverage is enough to see the target. The full 360 degrees takes about 10 minutes on a machine with 64 GB of RAM and a 3.6 GHz CPU.

azcov = 45 ⬦━━━━━━━━ ;

Set the scenario stop time based on the desired azimuth coverage.

```
scenario.StopTime = azcov/360*npulses/prf;
```

Run the simulation. Keep track of the frame number of each loop, and get the vector of range bins from the range response object. The matrix `ppi` will contain the signal data formatted as range-by-azimuth.

```
frame = 0;
nrng = floor(time2range(1/prf)/rngres);
ppi = zeros(nrng,npulses);
while advance(scenario)
    frame = frame + 1;
    reg.AzimuthCenter = (frame-1)*rotrate/prf;
    iqsig = receive(scenario);
    [ppi(:,frame),rngbins] = resp(sum(iqsig{1},2),mfc);
end
```

Create a PPI Image

A PPI image consists of a set of range profiles arranged in radial lines to form a circular image of the scene in Cartesian space.

The IQ data covers the full range ambiguity, so start by trimming the data to the ranges of interest. Use the min and max ground range defined earlier, then find the range gate indices that correspond to those ranges.

```
minRange = sqrt(minGndRng^2 + rdrHeight^2);
maxRange = sqrt(maxGndRng^2 + rdrHeight^2);
minGate = find(rngbins >= minRange,1,'first');
maxGate = find(rngbins <= maxRange,1,'last');
```

Convert the azimuth pointing angles and range bins to rectangular coordinates, then use the `surf` function to plot the image. When specifying the azimuth domain for the image, use one more point than the number of pulses so that the image fully encompasses all azimuth angles.

```
az = rotrate*(0:frame)/prf;
gndrng = sqrt(rngbins(minGate:maxGate).^2-rdrHeight^2);
x = gndrng.*cosd(az);
y = gndrng.*sind(az);
surf(x,y,zeros(size(x)),20*log10(abs(ppi(minGate:maxGate,1:frame))))
shading flat
view(0,90)
colorbar
axis equal
axis tight
colormap winter
title('PPI Image')
```
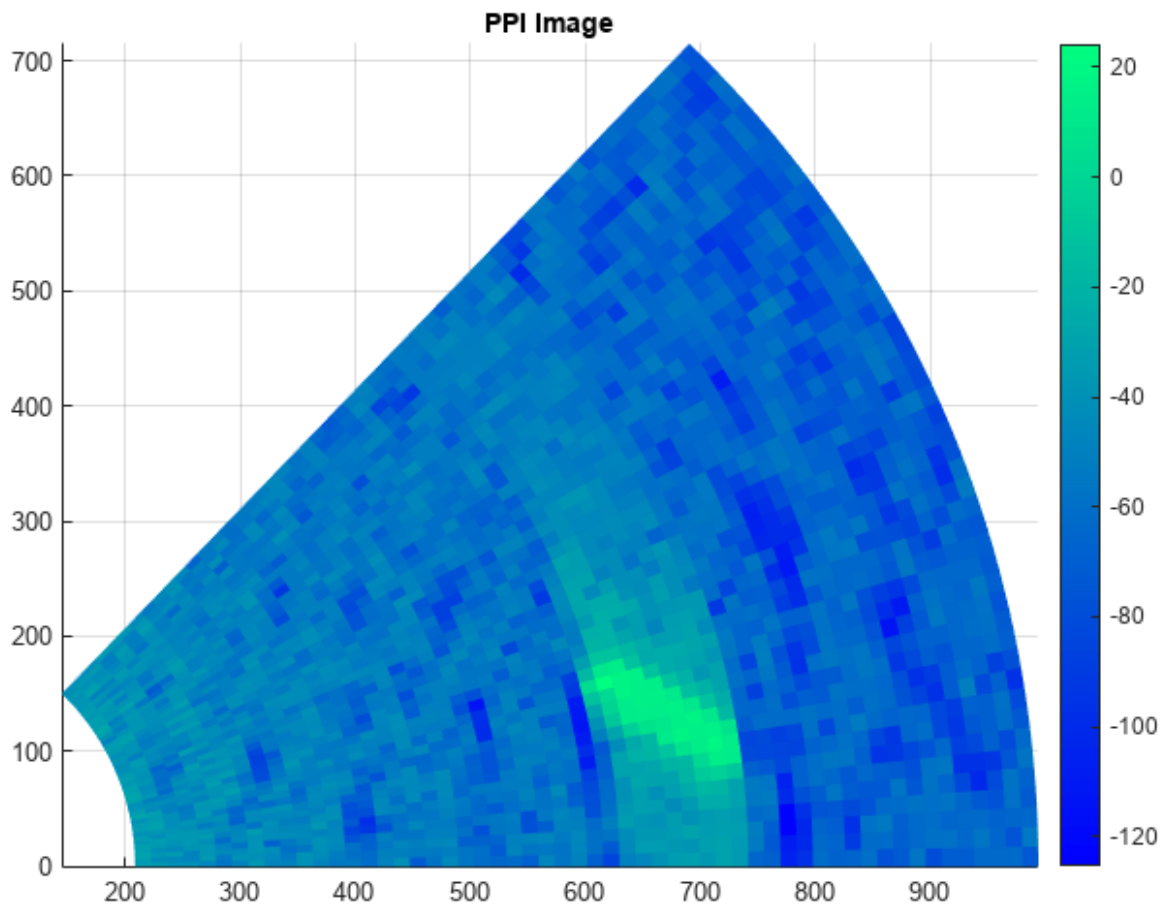
The gif below shows a recording of this scenario for 30 full rotations of the antenna over about 34 seconds.
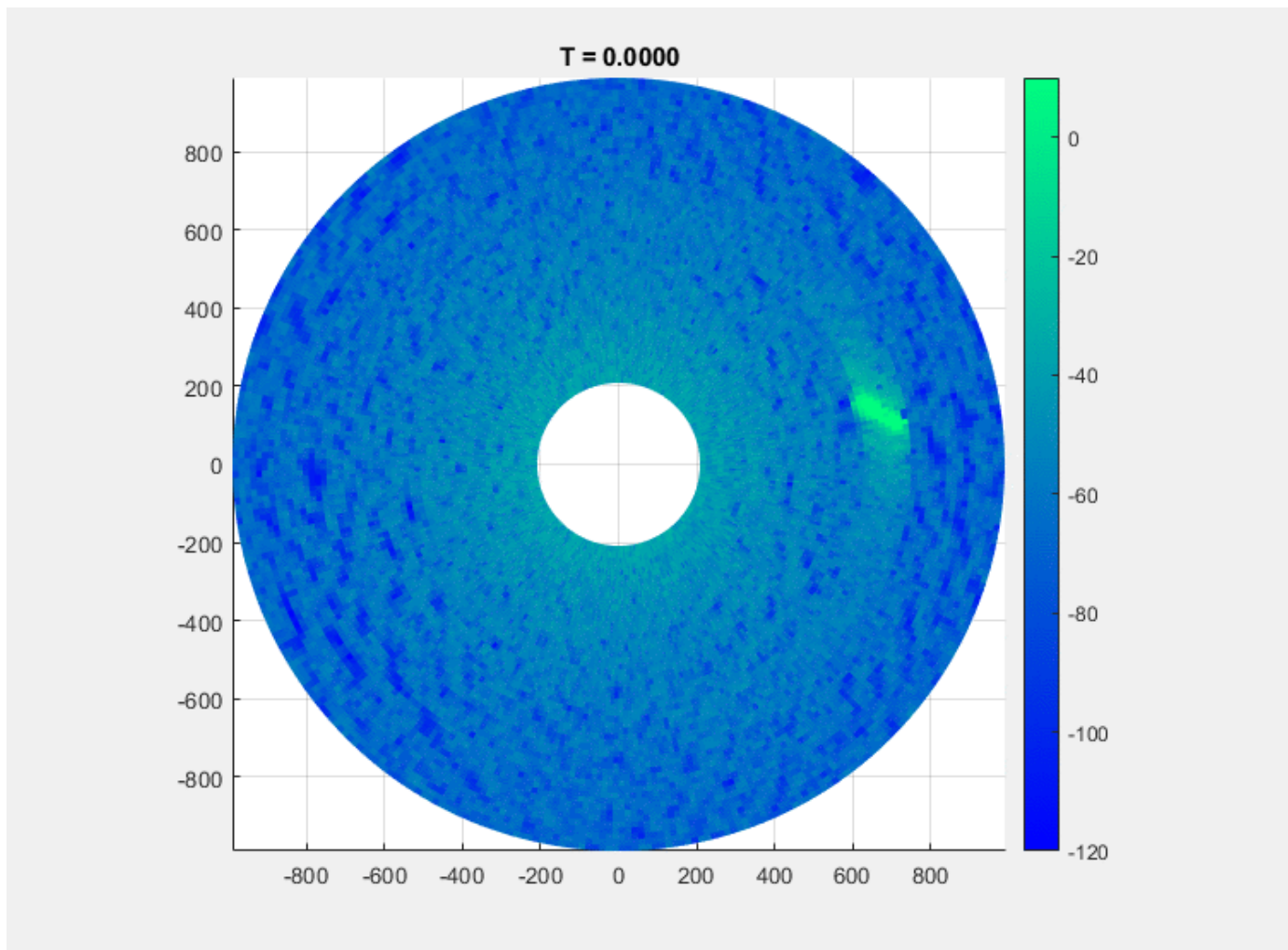
Conclusion

In this example, you saw how to generate clutter and target returns with a rotating radar in a maritime environment. You saw how to use a spectral model to get realistic sea heights and surface shadowing, and how to emulate an extended target with a set of point targets, which allows for partial occlusion of the target by the surface. The IQ data was converted from polar format to Cartesian and plotted with the color channel of the `surf` function to create a simple PPI image.

Supporting Functions

helperPlotScenarioGeometry

```
function helperPlotScenarioGeometry(rdrHeight,seaLength,array,freq,tgtpos)
% Plot a visualization of the scenario

fh = figure;

% Surface
t = 0:10:360;
r = linspace(0,seaLength/2,10).';
x = r.*cosd(t);
```

```matlab
y = r.*sind(t);
surf(x/1e3,y/1e3,zeros(size(x)),repmat(permute([0 1 1],[1 3 2]),[size(x) 1]),'EdgeColor','none')

hold on

% Antenna pattern
az = linspace(-6,6,80);
maxEl = -atand(2*rdrHeight/seaLength);
el = linspace(-90,maxEl,80);
G = pattern(array,freq,az,el);
[az,el] = meshgrid(az,el);
[x,y,~] = sph2cart(az*pi/180,el*pi/180,1);
rtg = rdrHeight./sind(-el);

surf(x.*rtg/1e3,y.*rtg/1e3,1e-2*ones(size(x)),G,'EdgeColor','none')

% Radar
plot3(0,0,rdrHeight,'o')
line([0 0],[0 0],[0 rdrHeight],'color','black','linestyle','--')

% Target
plot3(tgtpos(1)/1e3,tgtpos(2)/1e3,tgtpos(3),'*r')

hold off

text(0,0,rdrHeight*1.1,'Radar')
text(tgtpos(1)*.8/1e3,tgtpos(2)*1.1/1e3,rdrHeight/10,'Target')
text(seaLength/4/1e3,-200/1e3,rdrHeight/10,'Beam Pattern')
text(-seaLength/4/1e3,seaLength/4/1e3,rdrHeight/10,'Sea Surface')

xlabel('X (km)')
ylabel('Y (km)')
zlabel('Z (m)')
title('Scenario Geometry')

cl = clim;
clim([cl(2)-30 cl(2)])

fh.Position = fh.Position + [0 0 150 150];
view([-11 58])


end
```

Maritime Clutter Suppression with Neural Networks

This example shows how to train and evaluate neural networks to suppress maritime clutter returns from radar images using the Deep Learning Toolbox™. The Deep Learning Toolbox provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and apps.

Two example scenarios are shown here. In the first scenario, a denoising convolutional neural network is used to suppress clutter in a plan position indicator (PPI) image. The "Simulate a Maritime Radar PPI" on page 1-948 example demonstrates how to use Radar Toolbox™ to create PPI images for a rotating radar at sea. In the second scenario, a convolutional autoencoder is used to suppress clutter in a range-time image. The "Simulate a Coastal Surveillance Radar" on page 1-1051 example shows how to use Radar Toolbox to create range-time images for a stationary coastal surveillance radar.

Scenario 1: Clutter Suppression for a Maritime Radar PPI

In this first scenario, a rotating radar on a tall platform in open water has been simulated to create PPI radar images with clutter and target returns. A denoising convolutional network is used to suppress the clutter returns.

Set the random seed for repeatability.

```
rng default
```

The Maritime Radar PPI Dataset

The dataset contains 84 pairs of synthetic radar images. Each pair consists of an input image, which has both sea clutter and extended target returns, and a desired response image, which includes only the target returns. The images were created using a `radarScenario` simulation with a `radarTransceiver` and a rotating uniform linear array (ULA). Each image contains two nonoverlapping cuboid targets with one representing a small container ship and the other representing a larger container ship.

The following parameters are fixed from image to image:

- Frequency (10 GHz)
- Pulse length (80 ns)
- Range resolution (7.5 m)
- PRF (1 kHz)
- Azimuth beamwidth (1 deg)
- Radar height (55 m)
- Rotation rate (50 RPM)
- Small target dimensions (120-by-18-by-22 m)
- Large target dimensions (200-by-32-by-58 m)
- Small target fixed RCS (30 dBsm)
- Large target fixed RCS (40 dBsm)

The following parameters are randomized from image to image:

- Wind speed (7 to 17 m/s)
- Wind direction (0 to 180 deg)
- Target position (anywhere on the surface)
- Target heading (0 to 360 deg)
- Target speed (4 to 19 m/s)

This variation ensures that a network trained on this data will be applicable to a fairly wide range of target profiles and sea states for this radar configuration. For more information on sea states, see the "Maritime Radar Sea Clutter Modeling" on page 1-443 example.

Download the Maritime Radar PPI Images dataset and unzip the data and license file into the current working directory.

```
dataURL = 'https://ssd.mathworks.com/supportfiles/radar/data/MaritimeRadarPPI.zip';
unzip(dataURL)
```

Load the image data and pretrained network into a struct called `imdata`. This struct will have fields `img1` through `img84` and `resp1` through `resp84`.

```
imdata = load('MaritimeRadarPPI.mat');
```

Prepare the Data

You can use the pretrained network to run the example without having to wait for training. To perform the training steps, set the `doTrain` variable to `true` by selecting the check box below. The training for this network takes about 5 minutes on a GPU.

```
doTrain = ;
if ~doTrain
    load PPIDeclutterNetwork.mat
end
```

Image sets 1-70 are used for training and 71-80 for validation. The last 4 images will be used for evaluation of the network.

Format the data as a 4D array for use with the network trainer and training options. The first two dimensions are considered spatial dimensions. The third dimension is for channels (such as color channels). The separate images are arranged along the 4th dimension. The cluttered inputs are simply referred to as *images,* and the desired output is known as the *response*. Single precision is used since that is native to the neural network trainer.

```
imgs  = zeros(626,626,1,84,'single');
resps = zeros(626,626,1,84,'single');
for ind = 1:84
   imgs(:,:,1,ind) = imdata.(sprintf('img%d',ind));
   resps(:,:,1,ind) = imdata.(sprintf('resp%d',ind));
end
```

After formatting, clear the loaded data struct to save RAM.

```
clearvars imdata
```

Network Architecture

A network is defined by a sequence of layer objects, including an input and output layer. An `imageInputLayer` is used as the input layer so that the images may be used without any

reformatting. A `regressionLayer` is used for the output to evaluate a simple mean-squared-error (MSE) loss function. A cascade of 2D convolution layers with normalizations and nonlinear activations is used for the hidden layers.

Start by creating the input layer. Specify the spatial size of the input images along with the number of channels (1).

```
layers = imageInputLayer([626 626 1]);
```

Add 3 sets of convolution+normalization+activation. Each convolution layer consists of a set of spatial filters. The `batchNormalizationLayer` biases and scales each mini batch to improve numerical robustness and speed up training. The `leakyReluLayer` is a nonlinear activation layer that scales values below 0 while leaving values greater than 0 unmodified.

Care must be taken to ensure the spatial and channel dimensions are consistent from layer to layer and that the size and number of channels of the output from the last layer matches the size and number of channels of the desired response images. Set the `Padding` property of the convolution layers to `'same'` so that the filtering process does not change the spatial size of the images.

```
layers = [layers;
          convolution2dLayer([5 5],1,Padding='same');
          batchNormalizationLayer;
          leakyReluLayer(0.2);
          convolution2dLayer([6 6],4,Padding='same');
          batchNormalizationLayer;
          leakyReluLayer(0.2);
          convolution2dLayer([5 5],1,Padding='same');
          batchNormalizationLayer;
          leakyReluLayer(0.2)];
```

Finally, add the output layer, which is a simple regression layer.

```
layers = [layers; regressionLayer];
```

Train the Network

Use the `trainingOptions` function to configure exactly how the network is trained. In addition to specifying the training method to use, this provides control over things like learn-rate scheduling and the size of mini batches. The `trainingOptions` can also be used to specify a validation data set, which is used to determine the running performance. Since the performance of a network may not improve monotonically with iterations, this also provides a way to return the network at whichever iteration yielded the lowest validation error.

Define the IDs of the sets to use for training and for validation.

```
trainSet = 1:70;
valSet = 71:80;
```

Now create the `trainingOptions`. Use the adaptive moment estimation (Adam) solver. Train for a maximum of 80 epochs with a mini batch size of 20. Set the initial learn rate to 0.1. The validation set is specified with a 1-by-2 cell array containing the validation image and response arrays. Set the `ValidationFrequency` to 25 to evaluate the loss for the validation set every 25 iterations. Specify `OutputNetwork` as `'best-validation-loss'` to return the network at the iteration which had the least validation loss. Set `Verbose` to true to print the training progress.

```
opts = trainingOptions("adam", ...
    MaxEpochs=80, ...
```

```
    MiniBatchSize=20, ...
    Shuffle="every-epoch", ...
    InitialLearnRate=0.1, ...
    ValidationData={imgs(:,:,:,valSet),resps(:,:,:,valSet)}, ...
    ValidationFrequency=25, ...
    OutputNetwork='best-validation-loss', ...
    Verbose=true);
```

Training is initiated with the `trainNetwork` method. Input the 4D training image and response arrays, the vector of network layers, and the training options. This will only run if the `doTrain` flag is set to `true`.

A compatible GPU is used by default if available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Computing Requirements" (Parallel Computing Toolbox). Otherwise, `trainNetwork` uses the CPU.

```
if doTrain
    [net,info] = trainNetwork(imgs(:,:,:,trainSet),resps(:,:,:,trainSet),layers,opts);
end
```

Use the provided helper function to plot the training and validation loss on a log scale.

```
helperPlotTrainingProgress(info)
```
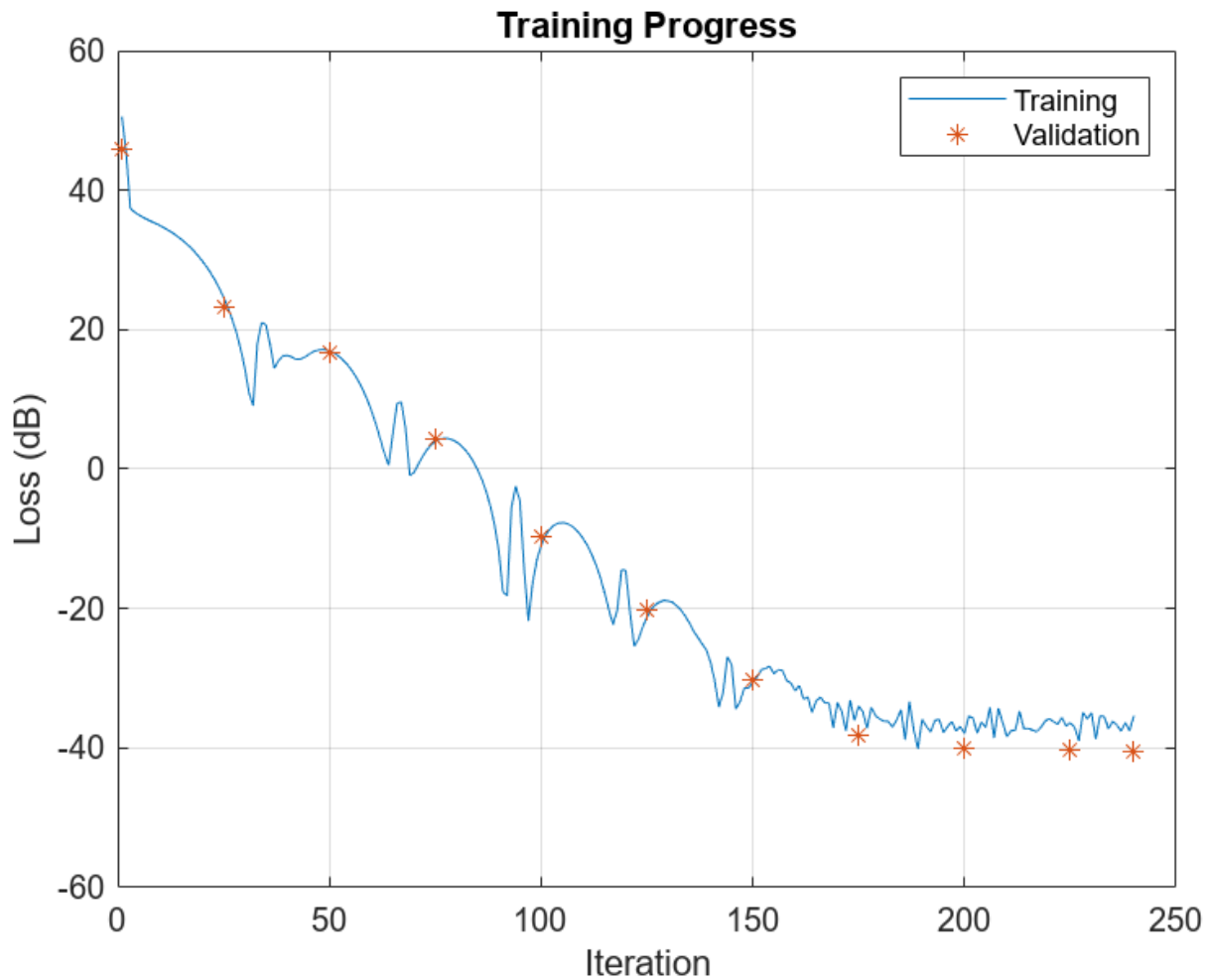
The training and validation loss decreased steadily until an error floor was hit at around iteration 200. More training samples are needed to improve the performance of this network.

Evaluate the Network

Now that the network has been trained, use the last 4 images to evaluate the network.

```
evalSet = 81:84;
```

Use the provided helper function to plot the input images alongside the responses output by the network. The results are normalized and pixels below -60 dB are clipped for easier comparison.

```
helperPlotEvalResults(imgs(:,:,:,evalSet),net);
```
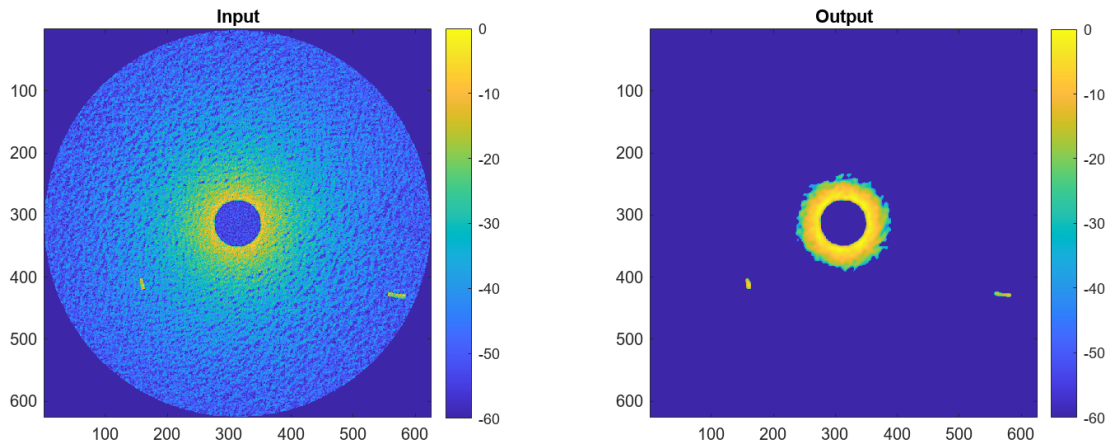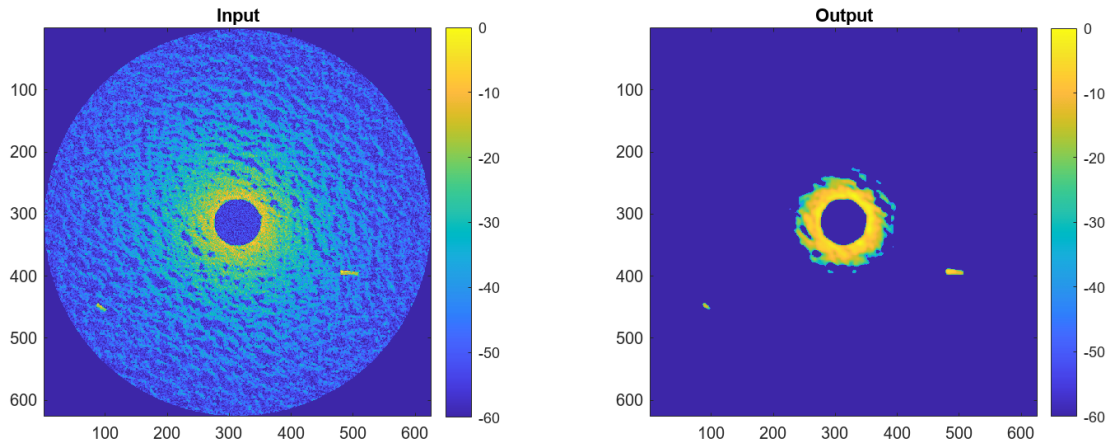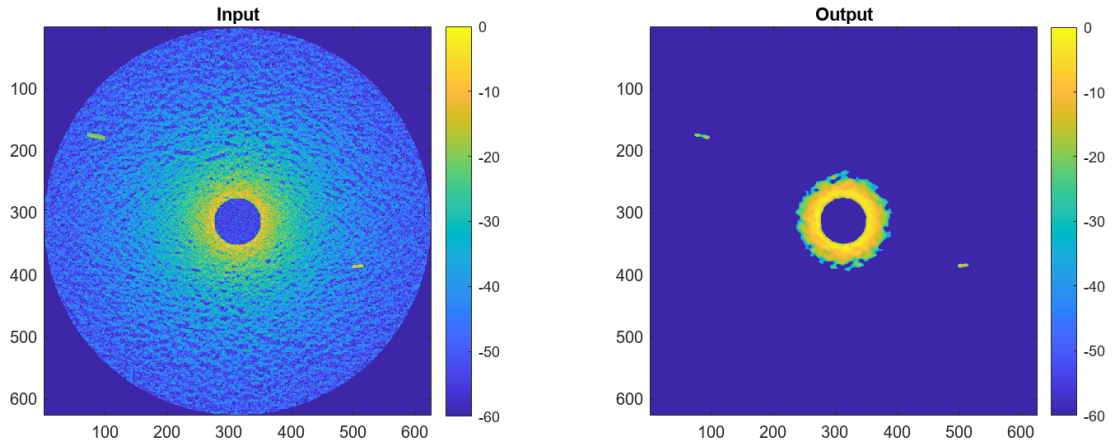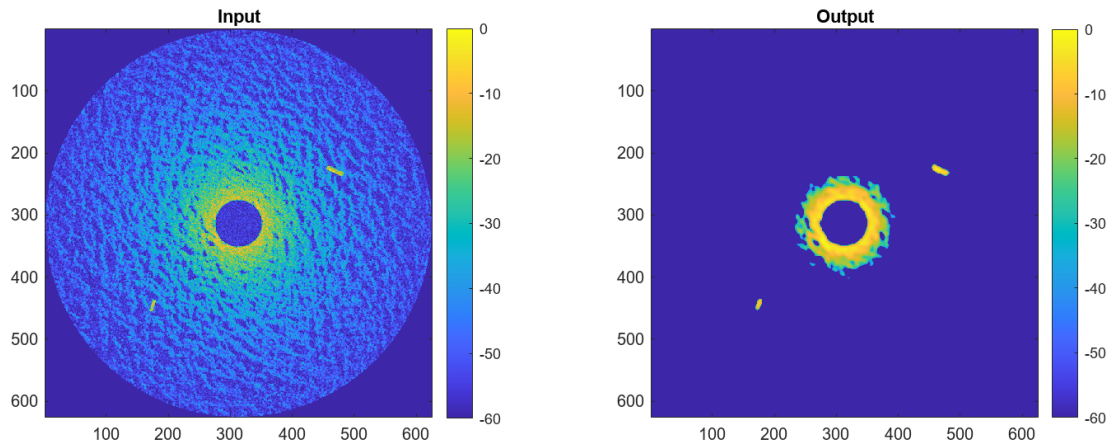
The network completely removes the sea clutter below a certain threshold of returned power while retaining the target signals with only a small dilation effect due to the size of the convolution filters used. The remaining high-power clutter near the center of the images could be removed by a spatially-aware layer, such as a fully-connected layer, or by preprocessing the original images to remove the range-dependent losses.

Scenario 2: Clutter Suppression for a Coastal Surveillance Radar

In this second scenario demonstrating maritime clutter suppression, a stationary radar viewing moving targets at sea has been simulated to create range-time radar images with clutter and target returns. A convolutional autoencoder is used to suppress the clutter returns.

Set the random seed for repeatability.

```
rng default
```

The Coastal Surveillance Radar Dataset

The dataset contains 10,000 pairs of synthetic range-time-intensity radar images. Each pair consists of an input image, which has both sea clutter and target returns, and a desired response image, which includes only the target returns. The images were created using a `radarScenario` simulation with a `radarTransceiver` and uniform linear array (ULA). Each image contains a single small extended target. A fairly high range resolution and a low PRF were used so that target range migration is visible. A short pulse is used for simplicity.

The following parameters are fixed from image to image:

- Frequency (10 GHz)
- Pulse length (21 ns)
- Range resolution (3.2 m)
- PRF (100 Hz)
- Azimuth beamwidth (4 deg)
- Number of pulses (80)
- Number of range gates (80)

- Radar height above sea surface (11 m)

The following parameters are randomized from image to image:

- Sea state (1 to 5, proportional wind speeds from 3 to 13 m/s)
- Target position (anywhere on the surface)
- Target heading (0 to 360 deg)
- Target speed (5 to 25 m/s)
- Target RCS (-14 to 6 dBsm)
- Target dimensions (length from 10 to 30 m, with proportional beam and height)

This variation ensures that a network trained on this data will be applicable to a fairly wide range of target profiles and sea states for this radar configuration.

Download the Coastal Surveillance Radar Images dataset and unzip the data and license file into the current working directory.

```
dataURL = 'https://ssd.mathworks.com/supportfiles/radar/data/CoastalSurveillanceRadarImages.zip'
unzip(dataURL)
```

Load the image data into a struct called `imdata`. This struct will have fields X and Y where X contains cluttered images and Y contains the ideal response images. Both are single-valued matrices of size 80-by-80-by-10,000 with each page representing an image. The images are formatted with range along the first dimension and slow time along the second dimension.

```
imdata = load('CoastalSurveillanceRadarImages.mat');
```

Prepare the Data

The data will be randomly segmented into a training set (80%), a validation set (10%), and an evaluation set (10%). Start by using `helperSegmentData` to get an index for each set.

```
numSets = size(imdata.X,3);
[trainIdx,valIdx,evalIdx] = helperSegmentData(numSets);
```

Format the sets into 1-by-2 cell arrays as expected by the network object.

```
trainData = {imdata.X(:,:,trainIdx), imdata.Y(:,:,trainIdx)};
valData   = {imdata.X(:,:,valIdx),   imdata.Y(:,:,valIdx)};
evalData  = {imdata.X(:,:,evalIdx),  imdata.Y(:,:,evalIdx)};
```

The network uses the 3rd dimension for channels and the 4th for different images, so swap the 3rd and 4th array dimensions now.

```
trainData = cellfun(@(t) permute(t,[1 2 4 3]),trainData,'UniformOutput',false);
valData   = cellfun(@(t) permute(t,[1 2 4 3]),valData,'UniformOutput',false);
evalData  = cellfun(@(t) permute(t,[1 2 4 3]),evalData,'UniformOutput',false);
```

Lastly, the data should be normalized for good training performance. Perform normalization on all of the training and validation input and output images using a column norm. Only the input images from the evaluation set need to be normalized. Save the scaling factor used for each column in each cluttered evaluation image so the normalization can be reversed on the output image for better comparison to the original.

```
trainData{1} = normalize(trainData{1},'norm');
trainData{2} = normalize(trainData{2},'norm');
```

```
valData{1}   = normalize(valData{1},'norm');
valData{2}   = normalize(valData{2},'norm');
[evalData{1},~,evalNormScale]  = normalize(evalData{1},'norm');
```

You can use the pretrained network to run the example without having to wait for training. To perform the training steps, set the `doTrain` variable to `true` by selecting the check box below. The training in this section can be expected to take about 10-15 minutes on a GPU.

```
doTrain = ▆▄;
if ~doTrain
    load CoastSurvDeclutterNetwork.mat
end
```

After formatting, clear the loaded data struct to save RAM.

```
clearvars imdata
```

Network Architecture

An autoencoder uses input and output layers with the same data size but uses a smaller data size for the hidden layers to find compressed versions of the input. If the desired output response is set equal to the input then the compressed (hidden) data can be thought of as a latent space representation of the features that make up the input. By using clutter-free data for the desired response, the network can learn a latent space representation that omits the clutter return, thus filtering out clutter from the images.

Start by creating the input layer. The input images have size 80-by-80.

```
inputLayer = imageInputLayer([80 80 1]);
```

The autoencoder consists of a cascade of uniform "encoding layers" that perform strided convolutions and max-pooling followed by a cascade of "decoding layers" that perform the inverse operation. After the decoding layers is a simple cascade of 3 noise-reducing convolutional layers, along with batch normalization and activations. All of the encoding and decoding convolution layers use filters of size 3-by-3, and the number of filters per encode/decode layer is stepped up closer to the bottleneck to implement the logic needed to identify clutter and target signal features. Each encoding layer uses a stride of 2, reducing the number of pixels by 1/2 in each direction. The final compressed image then has size 10-by-10.

```
encodingLayers = [convolution2dLayer(3,8, Padding="same"); leakyReluLayer; maxPooling2dLayer(2,Pa
                  convolution2dLayer(3,16,Padding="same"); leakyReluLayer; maxPooling2dLayer(2,Pa
                  convolution2dLayer(3,32,Padding="same"); leakyReluLayer; maxPooling2dLayer(2,Pa
```

Transposed convolutional layers are used for decoding.

```
decodingLayers = [transposedConv2dLayer(3,32,Stride=2,Cropping="same"); leakyReluLayer;
                  transposedConv2dLayer(3,16,Stride=2,Cropping="same"); leakyReluLayer;
                  transposedConv2dLayer(3,8, Stride=2,Cropping="same"); leakyReluLayer];
```

A simple noise-reducing cascade can be formed with more convolution layers. Use a single filter for the last layer so there is only one output channel.

```
postProcessingLayers = [convolution2dLayer(3,4,Padding="same"); leakyReluLayer;
                        convolution2dLayer(3,4,Padding="same"); leakyReluLayer;
                        convolution2dLayer(3,1,Padding="same")];
```

Use a regression layer as the output layer.

```
outputLayer = regressionLayer;
```

Put the layers together in a single vector of layer objects.

```
layers = [inputLayer; encodingLayers; decodingLayers; postProcessingLayers; outputLayer];
```

Train the Network

Use the `trainingOptions` function to configure exactly how the network is trained. In addition to specifying the training method to use, this provides control over things like learn-rate scheduling and the size of mini batches. The `trainingOptions` can also be used to specify a validation data set, which is used to determine the running performance.

Now create the `trainingOptions`. Use the adaptive moment estimation (Adam) solver. Train for a maximum of 20 epochs with a mini batch size of 128. Start with an initial learn rate of 0.01 and drop this by a factor of 50% every 5 epochs. The validation set is specified with a 1-by-2 cell array containing the validation arrays. Set the `ValidationFrequency` to 50 to evaluate the loss for the validation set every 50 iterations. Set `Verbose` to true to print the training progress.

```
opts = trainingOptions('adam', ...
    'InitialLearnRate',0.01, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropFactor',0.5, ...
    'LearnRateDropPeriod',5, ...
    'Verbose',true, ...
    'MiniBatchSize',128, ...
    'Shuffle','every-epoch', ...
    'MaxEpochs',20, ...
    'OutputNetwork','last-iteration', ...
    'ValidationData',valData, ...
    'ValidationFrequency',50);
```

Training is initiated with the `trainNetwork` method. Input the 4D training image and response arrays, the vector of network layers, and the training options. This will only run if the `doTrain` flag is set to `true`.

A compatible GPU is used by default if available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see "GPU Computing Requirements" (Parallel Computing Toolbox). Otherwise, `trainNetwork` uses the CPU.

```
if doTrain
    [net,info] = trainNetwork(trainData{1},trainData{2},layers,opts);
end
```

Use the provided helper function to plot the training and validation loss on a log scale.

```
helperPlotTrainingProgress(info)
```
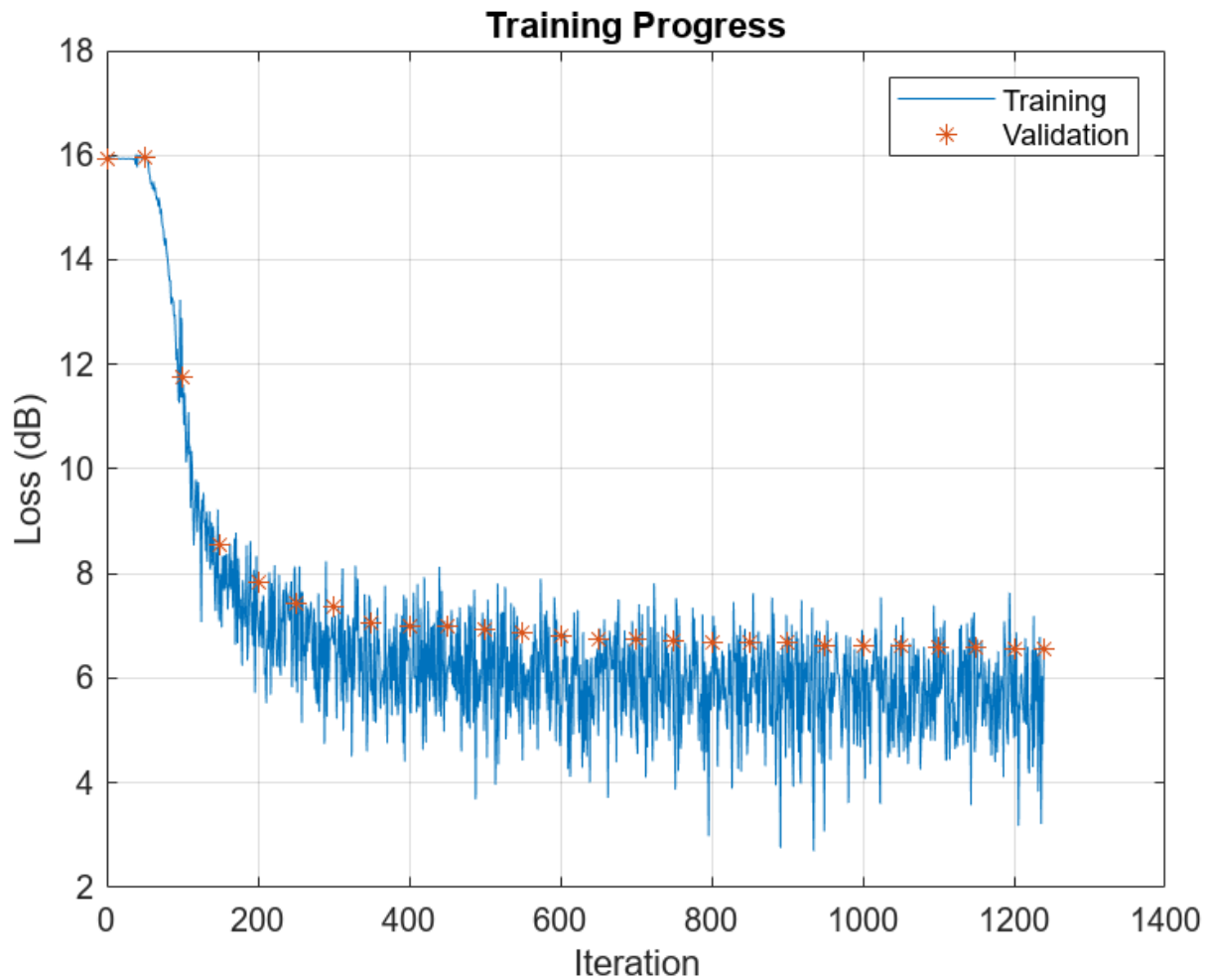
The training and validation loss decreases quickly at first but slows around iteration 400.

Evaluate the Network

Now that the network is trained, view the results for 6 pre-selected evaluation images.

```
showEvalResultsIdx = [1 3 7 22 25 40];
```

For each of these evaluation images, plot the original cluttered image, the desired response image, and the network output. The images are all plotted on a log scale with 60 dB of dynamic range.

```
for idx = showEvalResultsIdx
    helperPlotResult(net,evalData,evalNormScale,idx)
end
```
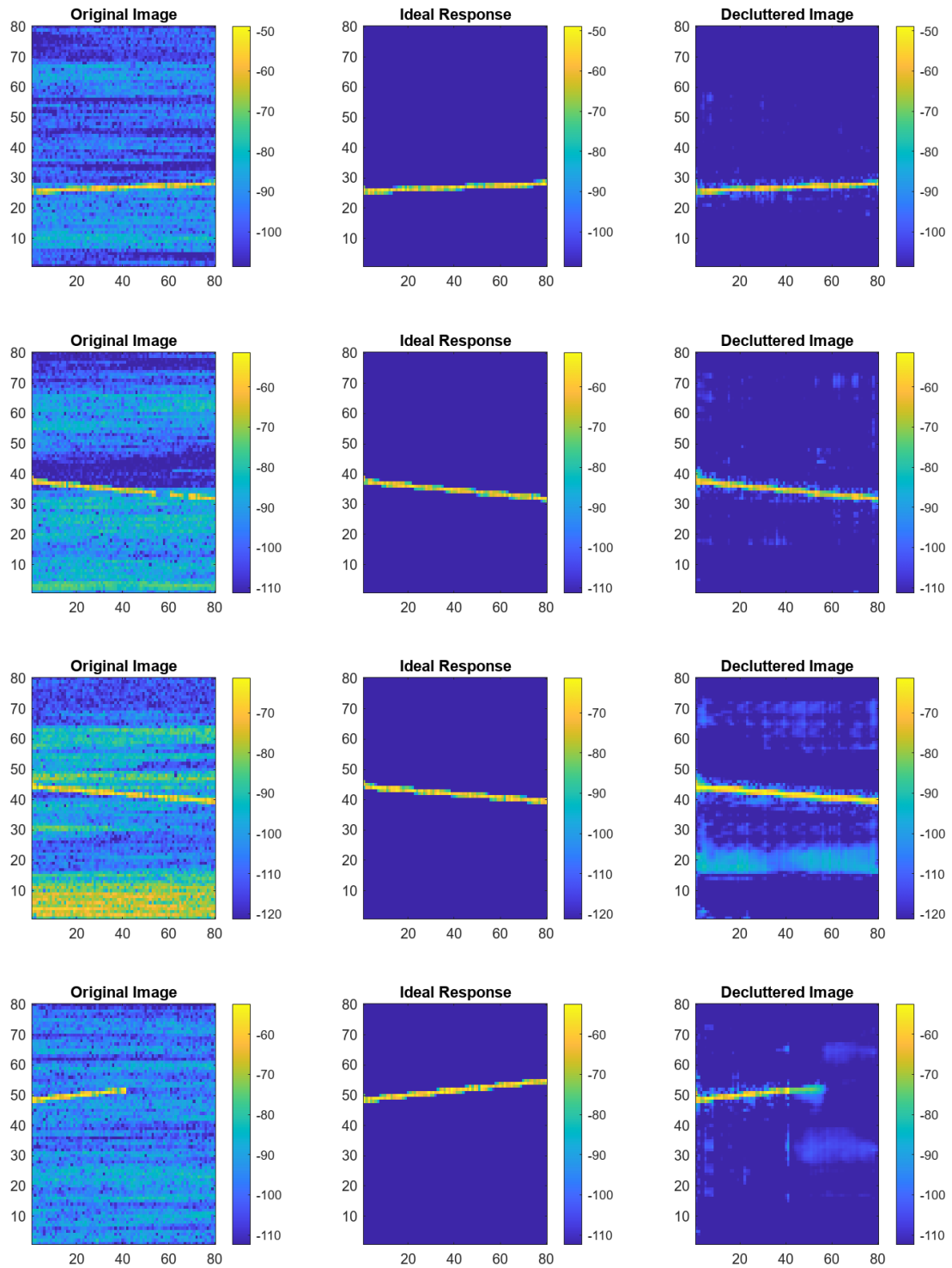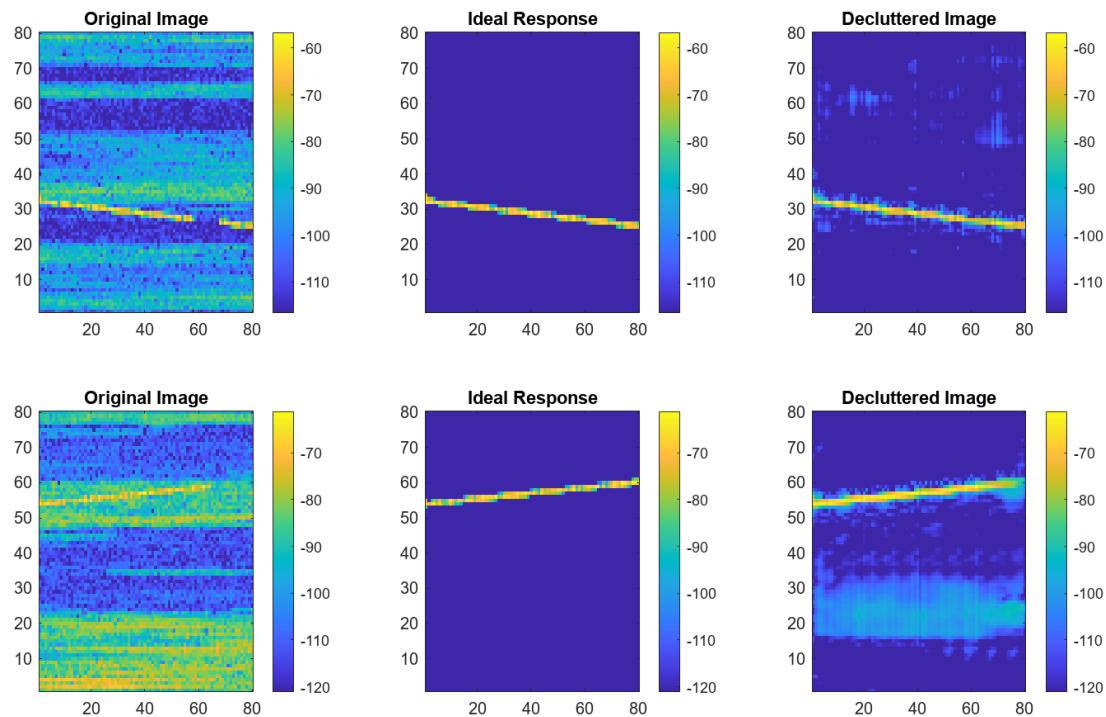
The network does a good job of removing unwanted clutter signals while retaining the target signal at its true power level. In cases where the target signal drops out entirely, the network is able to extrapolate out to fill in some of the missing data. Some background patches of noise are still present, which might be mitigated by a longer post processing network and further training.

Conclusion

In this example, you saw how to train and evaluate cascaded convolutional neural networks on PPI and range-time images to remove sea clutter while retaining target returns. You saw how to configure the input and output layers, the hidden convolution, normalization, and activation layers, and the training options.

Reference

[1] Vicen-Bueno, Raúl, Rubén Carrasco-Álvarez, Manuel Rosa-Zurera, and José Carlos Nieto-Borge. "Sea Clutter Reduction and Target Enhancement by Neural Networks in a Marine Radar System." *Sensors (Basel, Switzerland)* 9, no. 3 (March 16, 2009): 1913–36.

[2] Zhang, Qi, Yuming Shao, Sai Guo, Lin Sun and Weidong Chen. "A Novel Method for Sea Clutter Suppression and Target Detection via Deep Convolutional Autoencoder." (2017).

Supporting Functions

helperPlotTrainingProgress

```
function helperPlotTrainingProgress(info)
% Plot training progress

figure
plot(10*log10(info.TrainingLoss))
hold on
```

```matlab
plot(10*log10(info.ValidationLoss),'*')
hold off
grid on
legend('Training','Validation')
title('Training Progress')
xlabel('Iteration')
ylabel('Loss (dB)')

end
```

helperPlotEvalResults

```matlab
function helperPlotEvalResults(imgs,net)
% Plot original images alongside desired and actual responses

for ind = 1:size(imgs,4)

    resp_act = predict(net,imgs(:,:,1,ind));
    resp_act(resp_act<0) = 0;
    resp_act = resp_act/max(resp_act(:));

    fh = figure;

    subplot(1,2,1)
    im = imgs(:,:,1,ind);
    im = im/max(im(:));
    imagesc(20*log10(im))
    clim([-60 0])
    colorbar
    axis equal
    axis tight
    title('Input')

    subplot(1,2,2)
    imagesc(20*log10(resp_act))
    clim([-60 0])
    colorbar
    axis equal
    axis tight
    title('Output')

    fh.Position = fh.Position + [0 0 560 0];
end

end
```

helperSegmentData

```matlab
function [trainIdx,valIdx,testIdx] = helperSegmentData(numSets)

% 80% train, 10% validation, 10% test
props = [0.8 0.1 0.1];

% Training samples
trainIdx = randsample(numSets,floor(props(1)*numSets));

% Get remaining samples
valAndTestIdx = setdiff(1:numSets,trainIdx);
```

```matlab
% Validation samples
valIdx = randsample(valAndTestIdx,floor(props(2)*numSets));

% Remaining samples for test
testIdx = setdiff(valAndTestIdx,valIdx);

end
```

helperPlotResult

```matlab
function helperPlotResult(net,data,scale,idx)

% Get predicted output using scaled test data input
in = data{1}(:,:,1,idx);
out = predict(net,in);

% Denormalize the input and output for plotting
in = in.*scale(1,:,1,idx);
out = out.*scale(1,:,1,idx);

% It's possible for the network to output negative values for some pixels,
% so we need to take an abs
out = abs(out);

% The ideal response containing only the target image
ideal = data{2}(:,:,1,idx);

% Get color axis limits
mxTgtPwr = 20*log10(max(ideal(:)));
cl = [mxTgtPwr-60, mxTgtPwr];

fh = figure;
fh.Position = [fh.Position(1:2) 3*fh.Position(3) fh.Position(4)];

subplot(1,3,1)
imagesc(20*log10(in))
set(gca,'ydir','normal')
colorbar
clim(cl)
title('Original Image')

subplot(1,3,2)
imagesc(20*log10(ideal))
set(gca,'ydir','normal')
colorbar
clim(cl)
title('Ideal Response')

subplot(1,3,3)
imagesc(20*log10(out))
set(gca,'ydir','normal')
colorbar
clim(cl)
title('Decluttered Image')

end
```

FMCW Radar Altimeter Simulation

This example shows how to model a radar altimeter and measure the performance by simulating two scenarios using a land surface and moving platform. You will see how to build a waveform level altimeter model using a `radarTransceiver` and custom signal processor class. You will also see how to build a `radarScenario` by generating a `LandSurface` from an imported elevation data file and a `Platform` with an imported flight trajectory. Finally, you will see how to simulate the radar scenario to measure the performance of the radar altimeter.

Radar Parameter Definition

Aircraft have radar altimeters to determine the altitude above the local terrain as depicted in the figure below. Radar altimeters typically operate at 4.3 GHz and employ either FMCW or pulsed waveforms. These radars typically have very wide beamwidths (40 - 60 degrees) to provide accurate altitude estimates as the aircraft pitch and roll angles fluctuate throughout the flight.
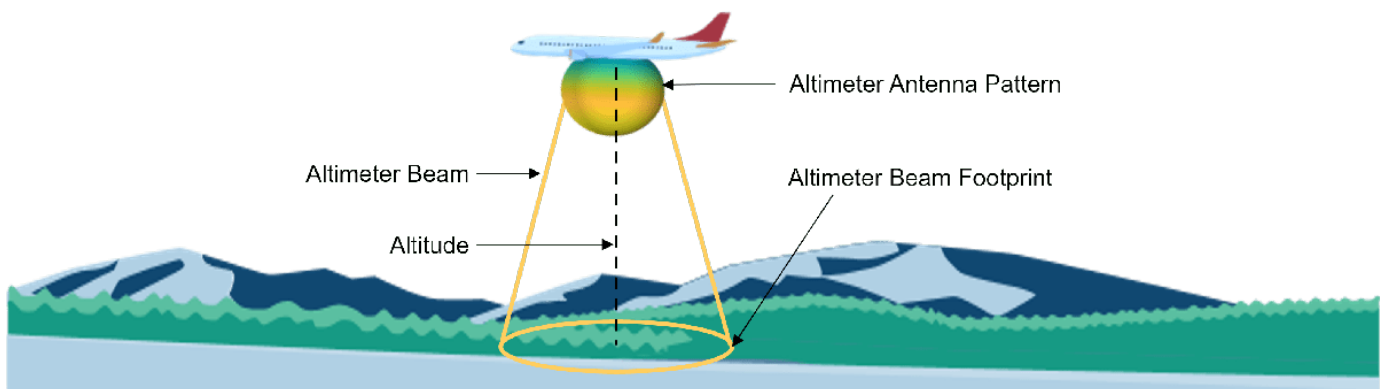
The altimeter model uses parameters from the first altimeter described in Table 2 of the Radar Altimeter Recommendation ITU Document [1].

```
fCenter = 4.3e9;            % Center frequency (Hz)
bandwidth = 150e6;         % Bandwdith (Hz)
prf = 143;                 % Pulse repetition frequency (Hz)
antBeamwidth = 40;         % Beamwidth (degrees)
pTransmitter = 0.4;        % Transmitter power (W)
rMax = 1676;               % Highest altitude required (m)
sweepDir = 'Triangle';     % Sweep direction of the FMCW waveform
noiseFigure = 8;           % Receiver noise figure (dB)
```

Creating the Radar Altimeter Model

The radar altimeter model consists of two components. The first component is a `radarTransceiver`. This component models the transmission of the waveform and reception of the ground echo at the altimeter. The second component is a custom component created for this example. This component takes the IQ samples generated by the `radarTransceiver` and performs the signal processing necessary to calculate the altitude.

Antenna Design

A `phased.GaussianAntennaElement` is used to model the altimeter antenna element. The idealized antenna pattern of this antenna element has no sidelobes but will suffice for the purposes of this simulation.

```
% Generate the Gaussian antenna element
frequencyRange = [fCenter-bandwidth/2 fCenter+bandwidth/2];
altimeterAntennaElement = phased.GaussianAntennaElement("Beamwidth",antBeamwidth,"FrequencyRange"

% Plot the beamwidth to verify that it matches the expected antenna beamwidth
beamwidth(altimeterAntennaElement,fCenter);
```
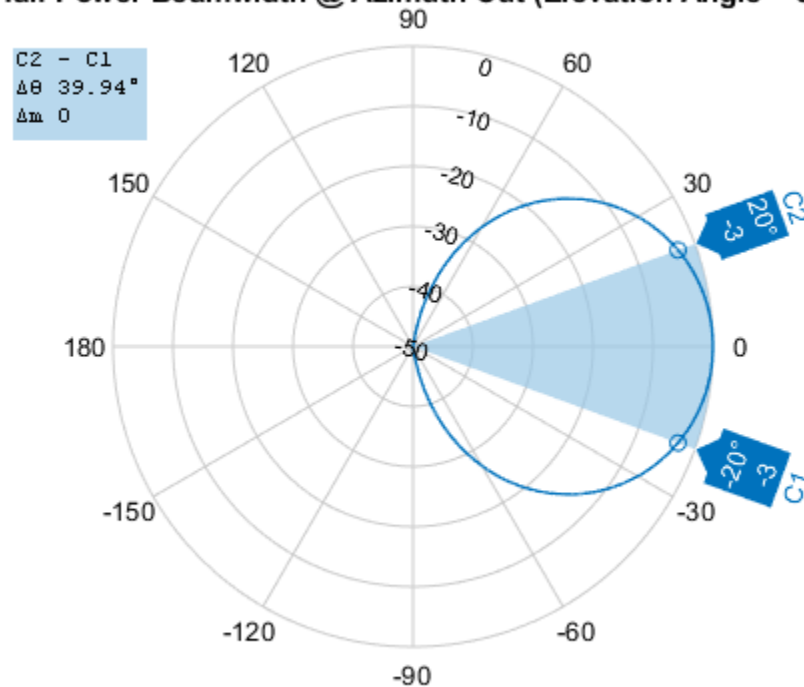

Power Pattern (dB), Broadside at 0.00° @ 4GHz

The transmit and receive antennas are modeled as a `phased.Radiator` and `phased.Collector`. Each antenna contains the `phased.GaussianAntennaElement`.

```
altimeterTxAntenna = phased.Radiator('OperatingFrequency',fCenter,'Sensor',altimeterAntennaElemen
altimeterRxAntenna = phased.Collector('OperatingFrequency',fCenter,'Sensor',altimeterAntennaEleme
```

Waveform Design

Altimeters commonly use FMCW technology for many reasons, including the ability to measure short distances, high range accuracy, and low power requirements as discussed in [1]. Therefore, this example uses a triangle FMCW waveform.

A key processing step for FMCW radar is dechirping. Dechirping is the process of mixing the transmitted waveform with the received signal. For dechirping to work properly in simulation, the

sampling frequency of the waveform must be at least the signal bandwidth. The simulation differs from the real world in this regard, because in a real receiver dechirping occurs in the analog domain. The A/D sampling rate in an FMCW receiver only needs to be high enough to capture the bandwidth of the dechirped signal, which is much lower than the signal bandwidth.

Therefore, two sampling rates are used for the purpose of this simulation. The first sampling rate called the waveformSamplingRate is based on the waveform bandwidth and is used to generate the IQ and dechirp the signal. The second sampling rate called the receiverSamplingRate is based on the maximum expected beat frequency in the dechirped signal. After dechirping, the signal is down-sampled to the receiverSamplingRate to perform the remainder of the signal processing tasks.

To generate the waveform, first calculate the sweep slope and set the waveformSamplingRate, which is based on the bandwidth. The sampling rate of the waveform is not exactly the bandwidth because the waveform sampling rate times the sweep time must be an integer for the simulation to work properly.

```
tSweep = 1/(2*prf);                                    % Sweep time (s) is half of the sweep
nSweep = 2;                                             % There are two sweeps in one processi
sweepSlope = bandwidth/tSweep;                         % Sweep slope (Hz / s)
waveformSamplingRate = round(bandwidth*tSweep)/tSweep; % Sampling rate of the waveform (Hz)
```

Create the FMCW waveform once the sweep time and waveform sampling rate have been calculated.

```
altimeterWaveform = phased.FMCWWaveform('SweepBandwidth',bandwidth,'SampleRate',waveformSamplingR
```

Create the Radar Transceiver

The radarTransceiver can be fully modeled now that the antenna and waveform have been created.

Create the transmitter and receiver using previously defined system parameters and the selected sampling rate calculated in the previous section.

```
altimeterTransmitter = phased.Transmitter('PeakPower',pTransmitter);
altimeterReceiver = phased.ReceiverPreamp('SampleRate', waveformSamplingRate,'NoiseFigure', noise
```

Create the radarTransceiver. The mounting angle is set so that the altimeter is facing towards the surface in the North, East, Down (NED) reference frame.

```
altimeterTransceiver = radarTransceiver('Waveform',altimeterWaveform,...
    'Transmitter',altimeterTransmitter, ...
    'TransmitAntenna',altimeterTxAntenna, ...
    'ReceiveAntenna',altimeterRxAntenna, ...
    'Receiver',altimeterReceiver, ...
    'MountingAngles', [0 -90 0], ...
    'NumRepetitions', nSweep)

altimeterTransceiver =
  radarTransceiver with properties:

              Waveform: [1x1 phased.FMCWWaveform]
           Transmitter: [1x1 phased.Transmitter]
        TransmitAntenna: [1x1 phased.Radiator]
         ReceiveAntenna: [1x1 phased.Collector]
              Receiver: [1x1 phased.ReceiverPreamp]
      MechanicalScanMode: 'None'
       ElectronicScanMode: 'None'
```

```
     MountingLocation: [0 0 0]
       MountingAngles: [0 -90 0]
 NumRepetitionsSource: 'Property'
       NumRepetitions: 2
    RangeLimitsSource: 'Property'
          RangeLimits: [0 Inf]
      RangeOutputPort: false
       TimeOutputPort: false
```

Create the Signal Processor

Collect the upsweep and downsweep waveform samples from the `altimeterTransceiver`. The signal processor uses these samples to dechirp the received signals.

```
upsweepWaveformSamples = altimeterTransceiver.Waveform();        % Upsloping signal IQ
downsweepWaveformSamples = altimeterTransceiver.Waveform();      % Downsloping signal IQ
```

Plot the spectrogram of the waveform to ensure it matches expectations for a triangular FMCW waveform.

```
figure;
stft([upsweepWaveformSamples;downsweepWaveformSamples],waveformSamplingRate,'FrequencyRange','two
title('Spectrogram of Altimeter Waveform (BW = 150 MHz, PRF = 143 Hz)')
```
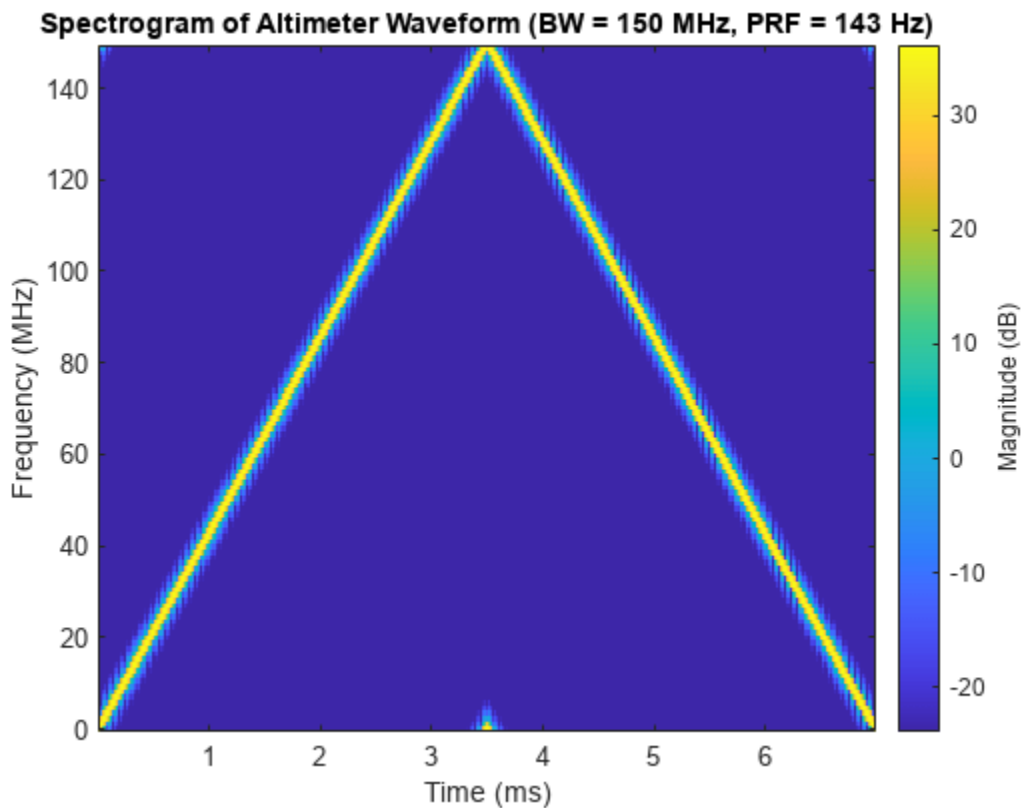

The IQ data is down-sampled to the `receiverSamplingRate` by the signal processor after dechirping. The `receiverSamplingRate` must be at least twice the maximum beat frequency. The

maximum beat frequency is the sum of the maximum range delay frequency and the maximum Doppler shift frequency. Calculate the maximum beat frequency and receiver sampling rate.

```matlab
frmax = range2beat(rMax,sweepSlope);   % Maximum frequency (Hz) due to range
vmax = 8;                              % Max rate of descent (m/s)
h = freq2wavelen(fCenter);             % Wavelength (m)
fdmax = 2*speed2dop(vmax,h);           % Maximum frequency shift (Hz) due to radial velocity
fbmax = frmax+fdmax;                   % Maximum beat frequency (Hz)
receiverSamplingRate = 2*fbmax;        % Sampling rate of the receiver (Hz)
```

A custom class represents the altimeter signal processor for this example. Now that all required inputs have been generated, create the `altimeterSignalProcessor`. The concept of this signal processor is discussed more in a later section.

```matlab
altimeterSignalProcessor = helperAltimeterSignalProcessor(upsweepWaveformSamples,downsweepWavefo

altimeterSignalProcessor =
  helperAltimeterSignalProcessor with properties:

      UpsweepReference: [524476x1 double]
    DownsweepReference: [524476x1 double]
              Detector: [1x1 phased.CFARDetector]
         RangeResponse: [1x1 phased.RangeResponse]
```

Create Radar Scenario with a Land Surface

The simulation of the altimeter model is composed of an altimeter mounted on a platform with a flight trajectory over a land surface. The first part of the scenario setup is creating a `LandSurface` from imported elevation data. This elevation data was generated from a Digital Terrain Elevation Data (DTED) file which contains real world terrain data from Chicago O'Hare Airport. For more information on generating surface clutter signals using real terrain data, see "Introduction to Radar Scenario Clutter Simulation" on page 1-770.

Initialize the Radar Scenario

Initialize the `radarScenario` before adding a `LandSurface` and `ClutterGenerator`. `tStart` and `tStop` refer to the start and stop time for the trajectory. `tUpdate` controls the size of the time step used by the simulation. This value is set to 5 seconds instead of the default update rate to speed up the simulation. The default update rate if unset would be the PRF of the radar.

```matlab
tStart = 300;   % Simulated trajectory start (s)
tStop = 395;    % Simulated trajectory stop (s)
tUpdate = 5;    % Simulated trajectory update interval (s)

scene = radarScenario('StopTime',tStop-tStart,'UpdateRate',1/tUpdate,'IsEarthCentered',true);
```

Generate the Surface and Clutter

Elevation data is loaded to generate the surface in the boundary of interest. The reflectivity of the land is created using an APL model which supports a 90 degree grazing angle in an urban environment. The coordinates of the flight trajectory are used to choose the boundaries of the surface. A `LandSurface` and `ClutterGenerator` are added to the radarScenario to generate IQ data of the ground reflection as the simulation progresses.

```
load('surfaceHeightData.mat','surfaceHeight')                    % Load surface elevation
refl = surfaceReflectivityLand(Model = "APL", LandType = 'Urban');   % Land reflectivity model
boundaryLatLim = [41.9799   41.9879];                            % Set the boundary latitu
boundaryLonLim = [-87.9000  -87.7750];                           % Set the boundary longitu

% Create the land surface and clutter generator
surface = landSurface(scene,'Terrain',surfaceHeight.','Boundary',[boundaryLatLim; boundaryLonLim]

clutter = clutterGenerator(scene,altimeterTransceiver);
```

Landing Scenario Simulation

Altimeter performance requirements vary as a function of altitude and altimeters must continue to provide accurate altitude information as the aircraft roll angle changes. To verify that the altimeter meets these requirements, a landing scenario and a banking scenario are simulated. The landing scenario is discussed first followed by the banking scenario.

Generate Flight Trajectory

The landing scenario uses a real flight trajectory of an aircraft approaching Chicago O'Hare Airport. The last step in setting up the simulation is adding a `platform` to the `radarScenario` with the altimeter `radarTransceiver` as a sensor and a trajectory using the landing trajectory.

```
flightTrajectoryFile = 'ORD_FlightTraj.mat';
landingFlightTrajectory = helperGetFlightTrajectory(surface,flightTrajectoryFile,tStart,tStop,tUp
altimeterPlatform = platform(scene,'Sensors',altimeterTransceiver,'Trajectory',landingFlightTraje
```

The animations below visualize the landing scenario. Execute the following line of code to generate the animations.

```
helperCreateAltimeterScenarioAnimation(altimeterTransceiver,{'3d','side'},tStop,tStart,tUpdate,bo
```

Landing 3D View

Flight Path · Aircraft · Beam Footprint · Ground

Landing Side View

Flight Path · Aircraft · Beam Footprint · Ground
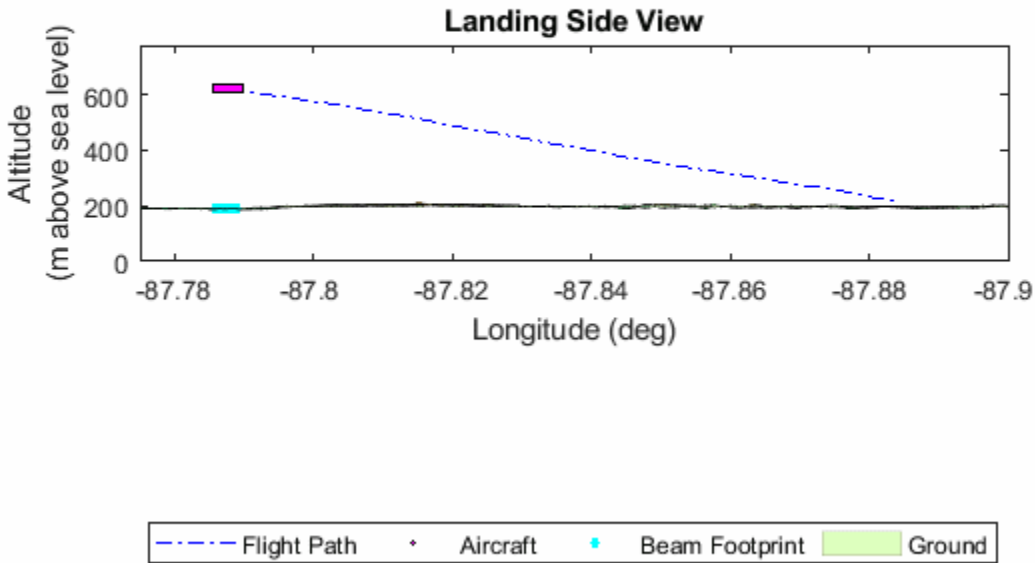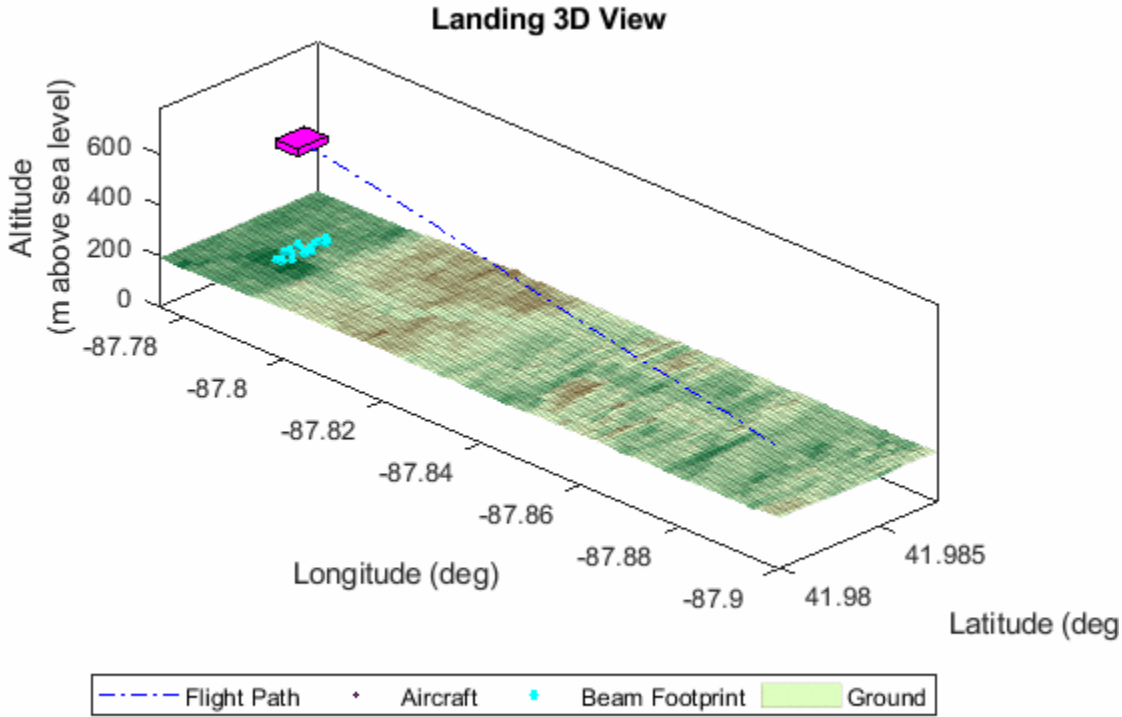
Run Simulation

First set the random number generator seed for reproducible results and setup containers for storing

data as the simulation proceeds.

```matlab
rng("default");                                        % Set random number generator seed
nSimulationPoints = floor((tStop-tStart)/tUpdate)+1;   % Calculate the number of simulation po
landingRadarSignal = cell(1,nSimulationPoints);        % Store the raw radar signal
landingMeasuredAltitude = zeros(1,nSimulationPoints);  % Store measured altitude
landingTruthAltitude = zeros(1,nSimulationPoints);     % Store truth altitde
landingRollAngle = zeros(1,nSimulationPoints);         % Store the aircraft roll angle in deg
cnt = 1;                                               % Initialize count variable
```

With the variables set up, run the simulation calculating the measured altitude, true altitude, and roll angle at each step. The clutter resolution is set at each simulation step so that the density of clutter scatterers is sufficient. A higher scatterer density is required as the aircraft descends.

```matlab
while advance(scene)

    % Calculate truth altitude
    [landingTruthAltitude(cnt), landingRollAngle(cnt)] = helperCalculateGroundTruth(surface,altim

    % Calculate required clutter resolution and set accordingly
    clutter.Resolution = helperDetermineClutterResolution(landingTruthAltitude(cnt));

    % Receive the IQ and track aircraft position and simulation time
    iqsig = receive(scene);
    radarSignal = iqsig{:};

    % save the first step of iq data for later analysis
    if cnt == 1
        firstLandingSignal = radarSignal;
    end

    % Calculate measured and truth altitude
    landingMeasuredAltitude(cnt) = altimeterSignalProcessor.estimateAircraftAltitude(radarSignal

    cnt = cnt + 1;
end

% Store the last signal for later analysis
lastLandingSignal = radarSignal;
```

Analyze Altimeter Signal Processor Ground Detection Algorithm

FMCW radars determine the range and velocity of a target by dechirping the received signal with the transmitted signal and measuring the beat frequency of the dechirped signal. For more information on using FMCW radars for target range estimation, see "Automotive Adaptive Cruise Control Using FMCW Technology" on page 1-148. This section discusses the altimeter signal processor functionality before looking at the overall performance of the altimeter during the landing scenario. The `helperAltimeterSignalProcessor` class contains the logic for the signal processor.

Use the data from the first step of the landing scenario simulation to analyze the ground detection algorithm.

```matlab
trueAltitude = landingTruthAltitude(1);      % Get the true altitude based on aircraft position
```

The first step in the signal processing algorithm is dechirping the received signal by mixing it with the known transmitted signal. The frequency of the dechirped signal is directly proportional to the

range of the target, in this case the ground. After the signal is dechirped, the signal processor converts frequency values to range values using the known sweep slope. This processing step is handled by the `phased.RangeResponse` object in the `altimeterSignalProcesor`. The following plot illustrates the range response for the upsweep and downsweep signals which is the magnitude of the received signal as a function of range.

```
plotStruct = struct('TrueAltitude',trueAltitude,'XLims',[trueAltitude-50 trueAltitude+50]);
altimeterSignalProcessor.plotRangeResponse(firstLandingSignal,plotStruct);
```
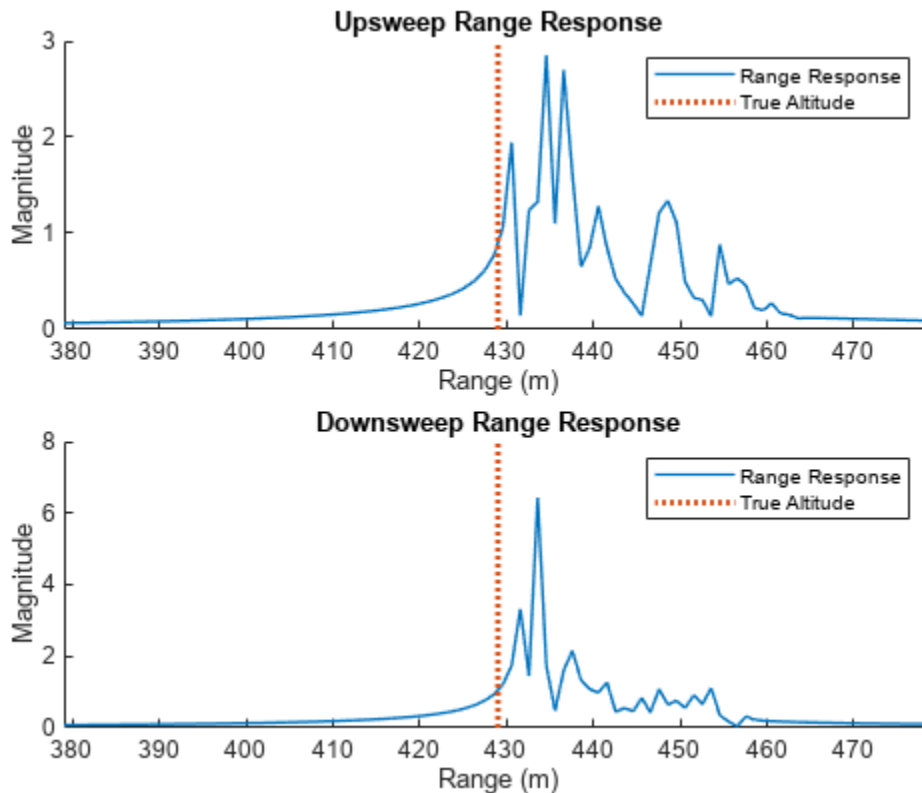

The plots above show that the returned energy from an altimeter at this altitude are spread over ~30 meters. This is expected because the target is the ground which has a large RCS and the altimeter uses a large beamwidth.

A CFAR algorithm is used to generate detections using the smallest of cell averaging technique to select for detections on the edge of the clutter return as this is where the ground lies. The number of detections is limited by using a false alarm rate of 1e-7. The following plots show the range response of the signals in addition to the detections that the CFAR algorithm in the signal processor found.

```
plotStruct = struct('TrueAltitude',trueAltitude,'Detections',[],'XLims',[trueAltitude-50 trueAlt:
altimeterSignalProcessor.plotRangeResponse(firstLandingSignal,plotStruct);
```

Once the detection algorithm has been applied, the detection with the highest power is used to estimate the aircraft altitude. The following plot shows the ranges that are selected to estimate the altitude from the upsweep and downsweep signals.

```
plotStruct = struct('TrueAltitude',trueAltitude,'Detections',[],'SelectedRange',[],'XLims',[trueA
altimeterSignalProcessor.plotRangeResponse(firstLandingSignal,plotStruct);
```
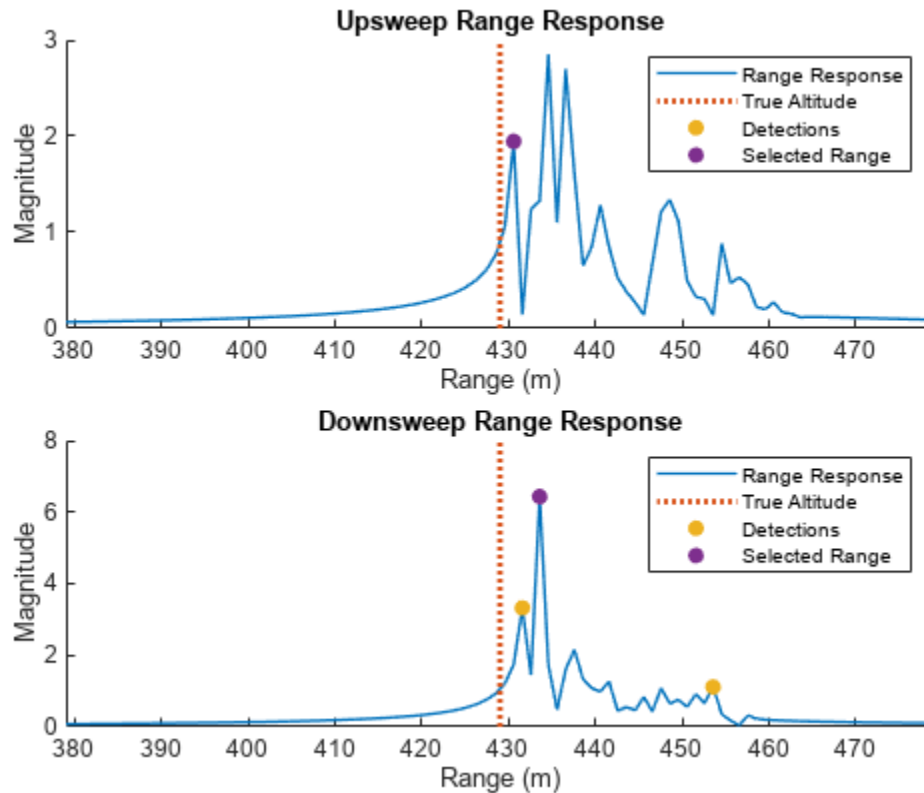
The effect of Doppler shift on the altitude estimate is removed by averaging the upsweep and downsweep altitude. The effectiveness of the altimeter algorithm can be checked by comparing the measured altitude to the ground truth altitude. For the first simulation step, the estimate is ~3 m from the true distance to the ground.

```
altitudeMeasured = altimeterSignalProcessor.estimateAircraftAltitude(firstLandingSignal);
fprintf('Measured Altitude = %0.1f m, True Altitude = %0.1f m, Altitude Error = %0.1f m',altitude
```

Measured Altitude = 432.1 m, True Altitude = 429.0 m, Altitude Error = -3.1 m

Landing Scenario Performance

The landing scenario measures the ability of the altimeter model to continue to produce good results as the range spread of the clutter return and the power level of the return vary. At higher altitudes, the return occurs over a much wider range and lower power level than at lower altitudes.

The plot below shows the range response of the highest and lowest altitude data point. Notice the difference in power as well as the width of the range response. The landing scenario tests the ability of the altimeter to meet performance requirements as the returned signal varies dramatically in shape and power.

```
plotStruct = struct('XLims',[0 600],'LegendName', ['Altitude = ',num2str(landingTruthAltitude(1)
[upaxis, downaxis] = altimeterSignalProcessor.plotRangeResponse(firstLandingSignal,plotStruct);
plotStruct = struct('XLims',[0 600],'LegendName', ['Altitude = ',num2str(landingTruthAltitude(en
altimeterSignalProcessor.plotRangeResponse(lastLandingSignal,plotStruct);
```
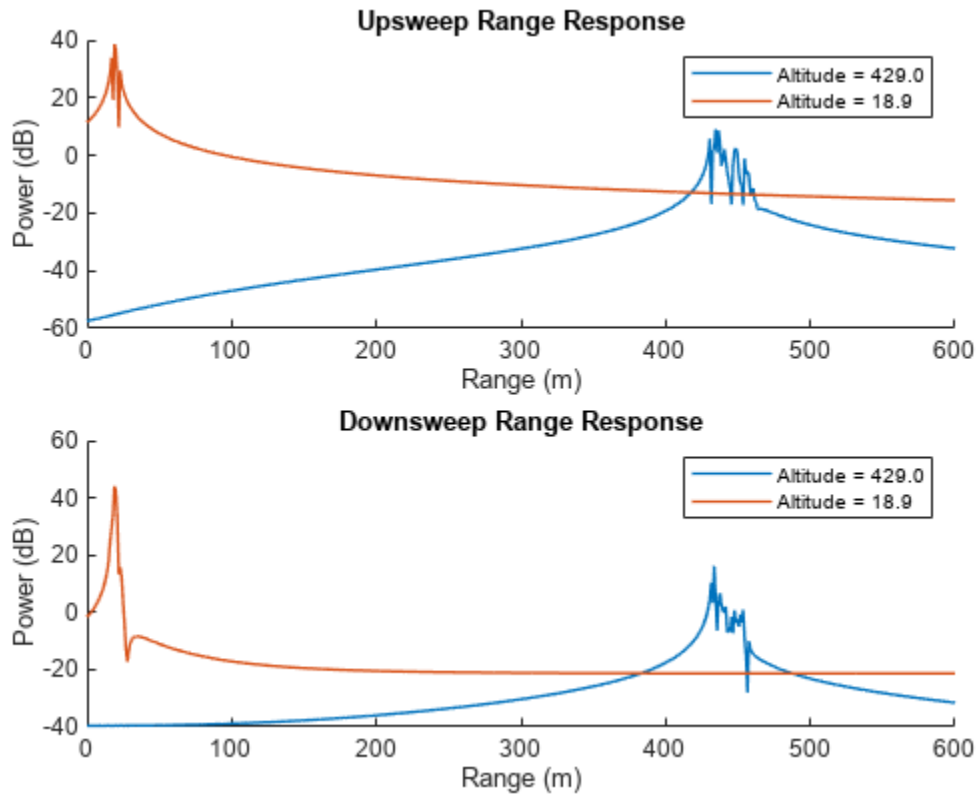
The following two plots visualize altimeter performance as the altitude of the aircraft changes. The first plot shows the altimeter altitude and true altitude throughout the duration of the landing scenario superimposed on top of the accuracy requirements derived from [1]. The second plot shows the altitude error throughout the landing scenario compared to the accuracy requirements derived from [1].

```
simTime = 0:tUpdate:(tStop-tStart);
helperPlotAltimeterAccuracy(landingTruthAltitude, landingMeasuredAltitude, simTime, 'Time (s)',
```
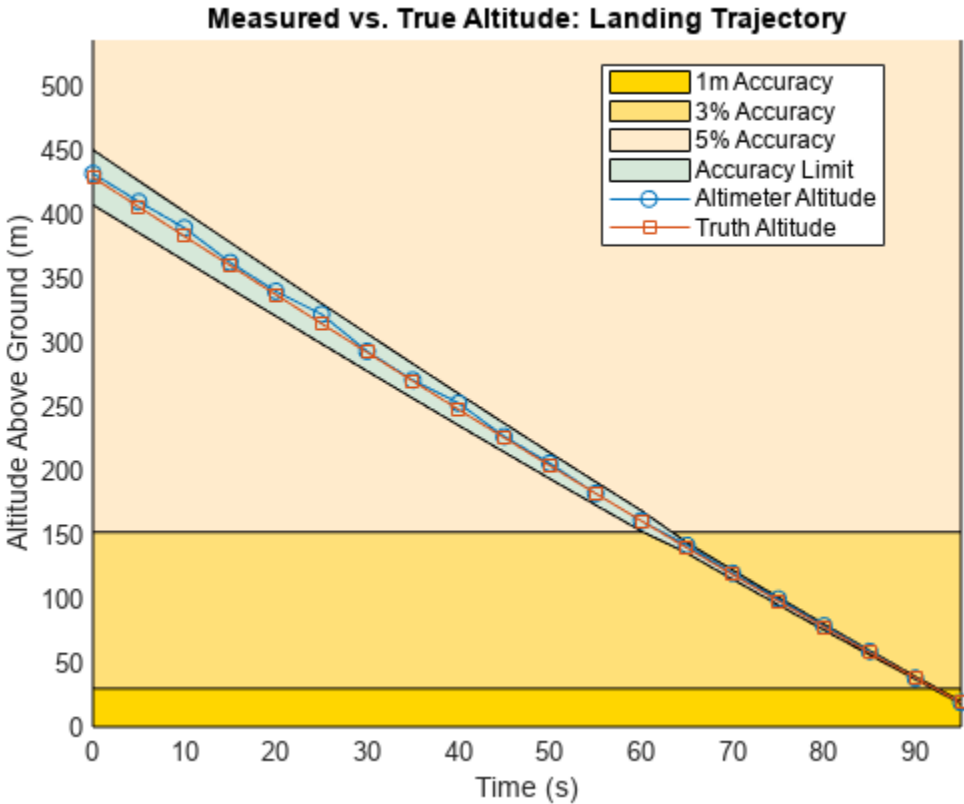
helperPlotAltimeterError(landingTruthAltitude,landingMeasuredAltitude,landingTruthAltitude,'True
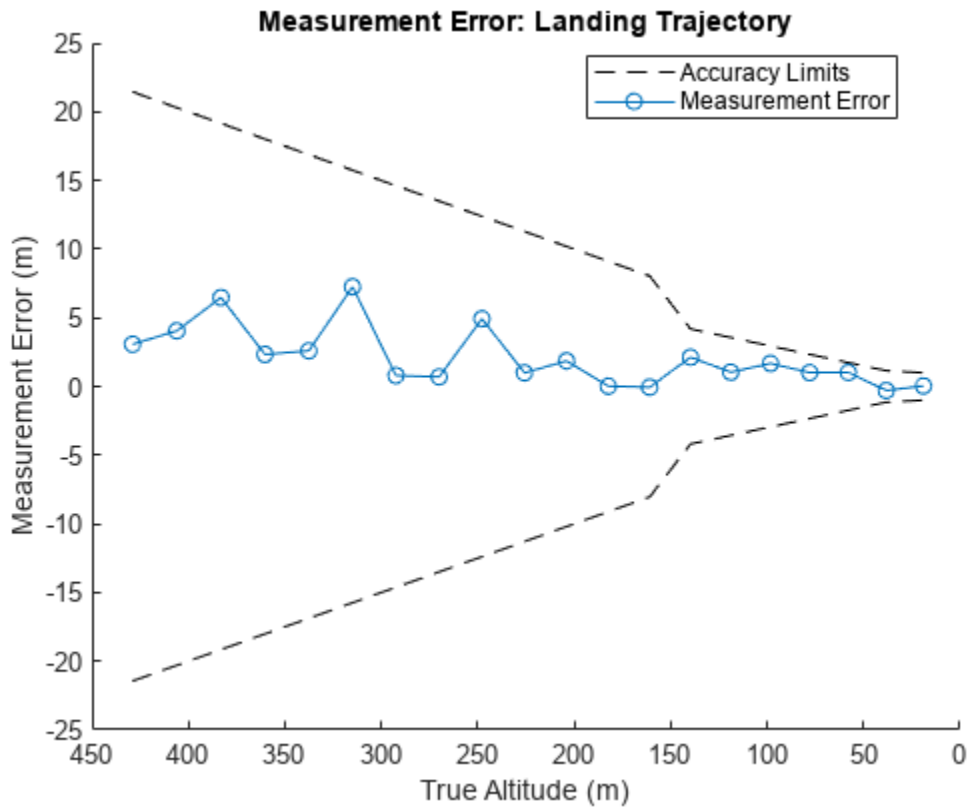
Overall, the altimeter as modeled meets the performance requirements laid out in [1] for the landing scenario. The error appears to reduce as the aircraft altitude drops. This makes sense because the beam footprint gets smaller as the aircraft altitude drops so the clutter returns happens over a smaller area.

Banking Scenario Simulation

The radar altimeter must continue to meet performance requirements as the aircraft roll and pitch angles change. A scenario in which the aircraft roll angle varies is simulated to evaluate this capability.

Generate Flight Trajectory and Update Radar Scenario

A helper function generates the flight trajectory for the aircraft banking scenario by modifying the landing trajectory to have a constant altitude and a roll angle that varies between positive and negative 1/2 beamwidth of the antenna. In this scenario, the flight direction remains constant despite the variation in aircraft roll for the sake of simplicity.

```
bankingFlightTrajectory = helperGetFlightTrajectory(surface,flightTrajectoryFile,tStart,tStop,tU
```

With the banking trajectory created, the scenario is restarted, and the flight trajectory of the platform is updated to use the banking trajectory instead of the landing trajectory.

```
altimeterPlatform.Trajectory = bankingFlightTrajectory;
```

The following animations visualize the banking scenario. Execute the following line of code to generate the animations.

```
helperCreateAltimeterScenarioAnimation(altimeterTransceiver,{'3d','front'},tStop,tStart,10,bound
```

Banking 3D View

Banking Front View
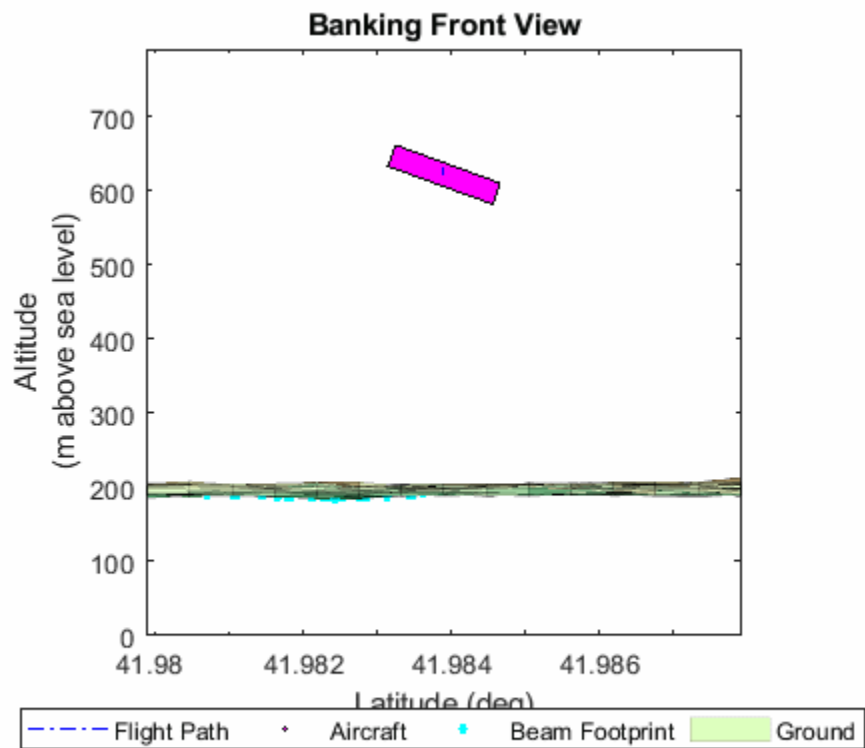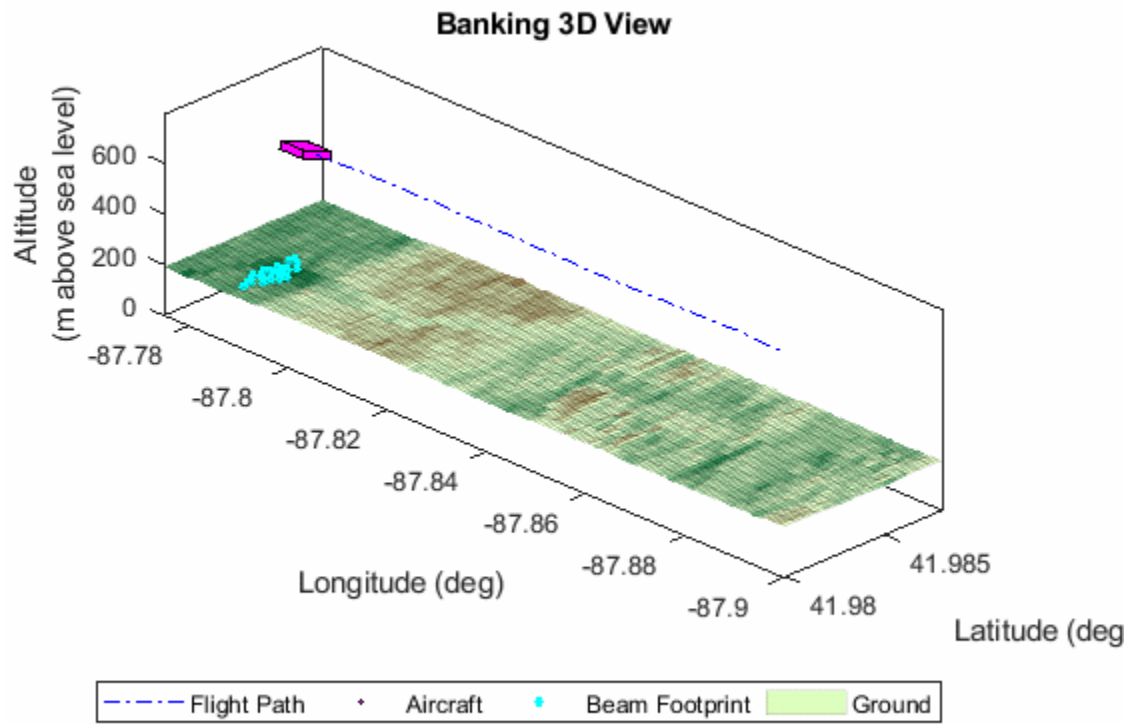
Run Simulation

Run the simulation now that the platform in the scenario follows a banking flight trajectory. In this

case a helper function runs the simulation because the exact same block of code was used to simulate the landing scenario.

```
signalToSave = [1 ceil(nSimulationPoints/2)];      % Save the first radar signal and the middle ra
[bankingMeasuredAltitude,bankingTruthAltitude,bankingRollAngle,bankingRadarSignal,maxErrorSignal
```

Banking Scenario Performance

The banking scenario measures the ability of the altimeter model to continue to produce good results as the shape of the clutter return changes due to aircraft roll angle. When the aircraft has a high roll angle, the nadir is at the edge of the beam footprint and the beam footprint covers a greater area. When the aircraft has a low roll angle, the nadir is in the center of the beam footprint and the beam footprint covers a smaller area.

The following plots show the range response when the roll angle of the aircraft is at it's highest (20 degrees) vs. lowest (1 degree). Notice that when the roll angle is lower, the maximum range response power is greater, and the spread of ranges is more narrow compared to the highest roll angle case.

```
plotStruct = struct('XLims',[400 600],'LegendName', ['Roll Angle = ',num2str(bankingRollAngle(1)
[upaxis, downaxis] = altimeterSignalProcessor.plotRangeResponse(bankingRadarSignal{1},plotStruct
plotStruct = struct('XLims',[400 600],'LegendName', ['Roll Angle = ',num2str(bankingRollAngle(ce
altimeterSignalProcessor.plotRangeResponse(bankingRadarSignal{end},plotStruct);
```
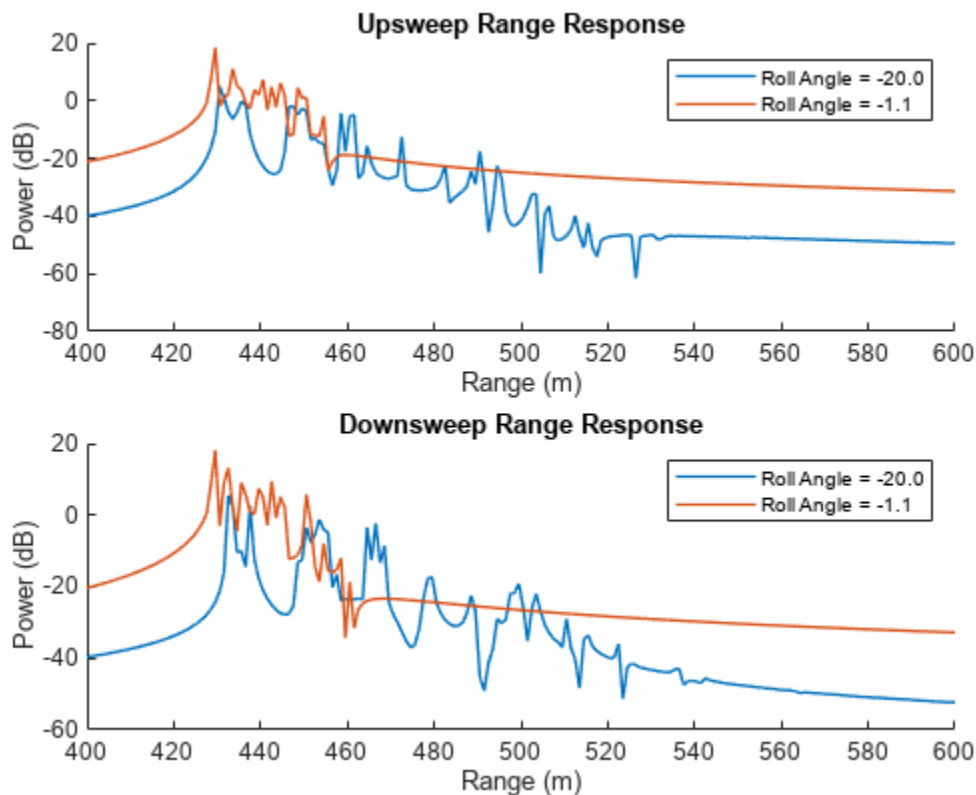

The following two plots visualize altimeter performance as the roll angle of the aircraft changes. Once again plot the measured and true altitude on the same graph along with the acceptable error limits for altimeters derived from [1]. Additionally, plot the measurement error along with the acceptable error limits. Roll angle is the independent variable in this case.

```
helperPlotAltimeterAccuracy(bankingTruthAltitude, bankingMeasuredAltitude, bankingRollAngle, 'Ro
```
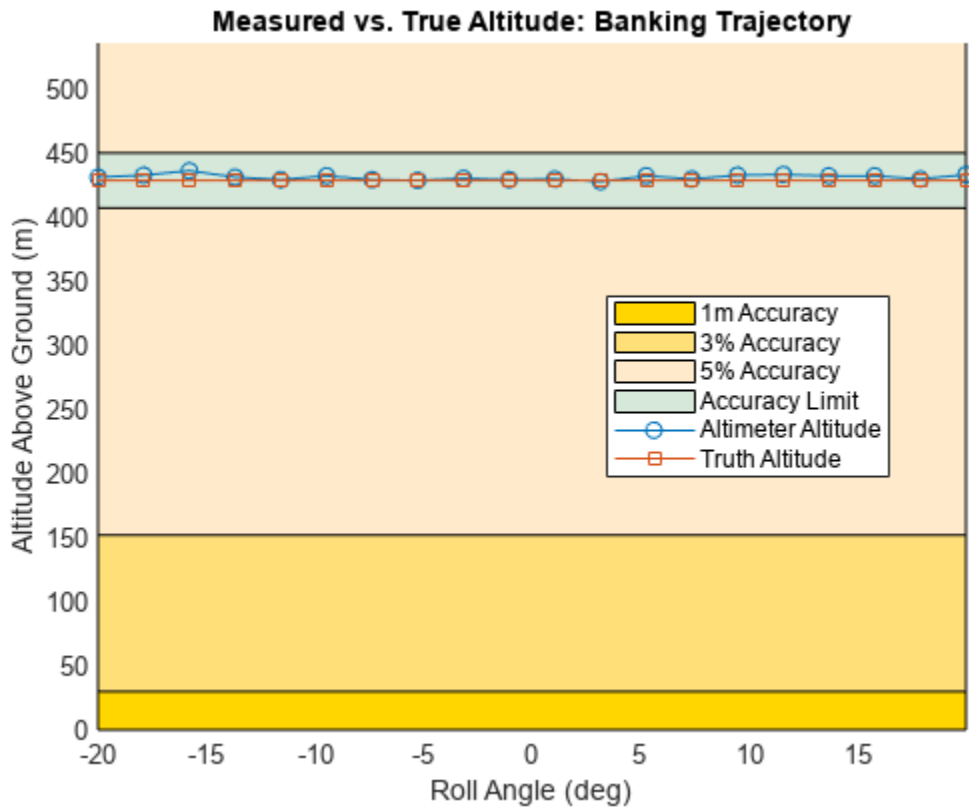


```
helperPlotAltimeterError(bankingTruthAltitude,bankingMeasuredAltitude,bankingRollAngle,'Roll Ang
```
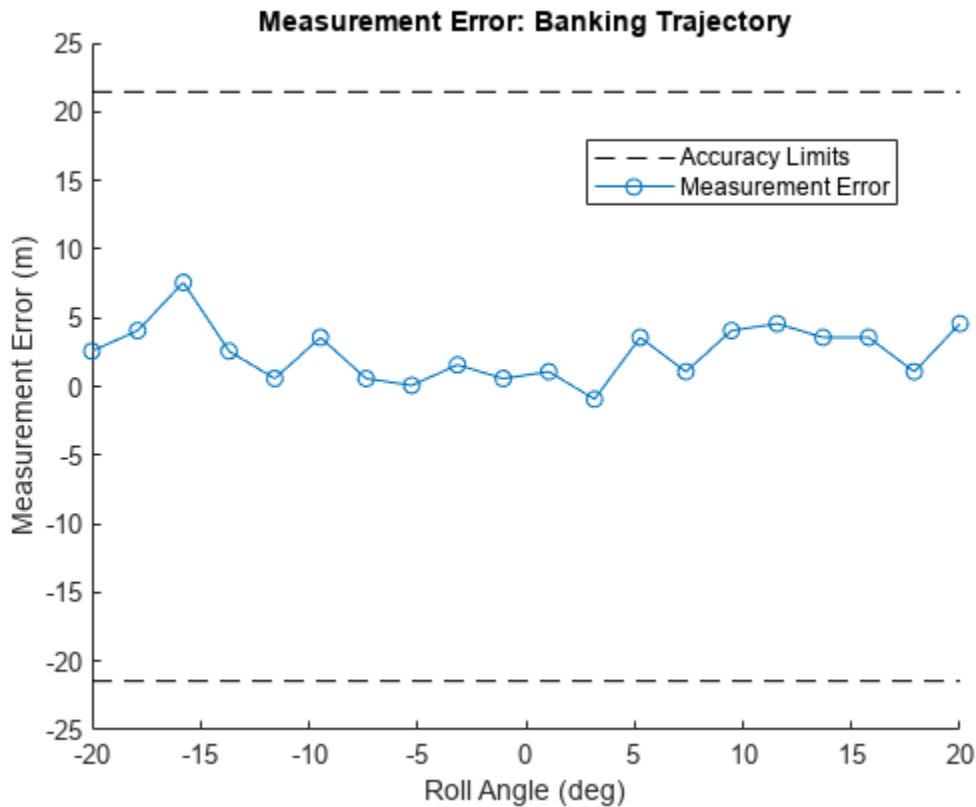
The performance of the altimeter in the banking scenario meets the requirements described in [1] as aircraft rolls and the shape of the signal return changes. There are a couple of points around 0 degrees that appear to be outliers in terms of error magnitude which are discussed in the next section.

Results Discussion

The altimeter meets the performance requirements described in [1] in the simulations of the landing and banking scenarios. In both cases there are points with particularly high measurement errors suggesting that improvements could still be made to this altimeter design.

As discussed, the ground is detected using a smallest of cell averaging CFAR algorithm to select for points on the edge of the clutter return. The highest power detection is used to estimate the aircraft altitude. The larger errors occur when the selected detection is not made on the leading edge of the clutter return. The following plot shows the clutter return with detections in addition to the CFAR threshold from the point with the largest error in the banking scenario.

```
maxErrorAltitude = bankingTruthAltitude(maxErrorIdx);                          % Return signal fo

% Plot the detections for the maximum measurement error point in the
% landing scenario
plotStruct = struct('TrueAltitude',maxErrorAltitude,'SelectedRange',[],'Detections',[],'Threshol
altimeterSignalProcessor.plotRangeResponse(maxErrorSignal,plotStruct);
```

1-989
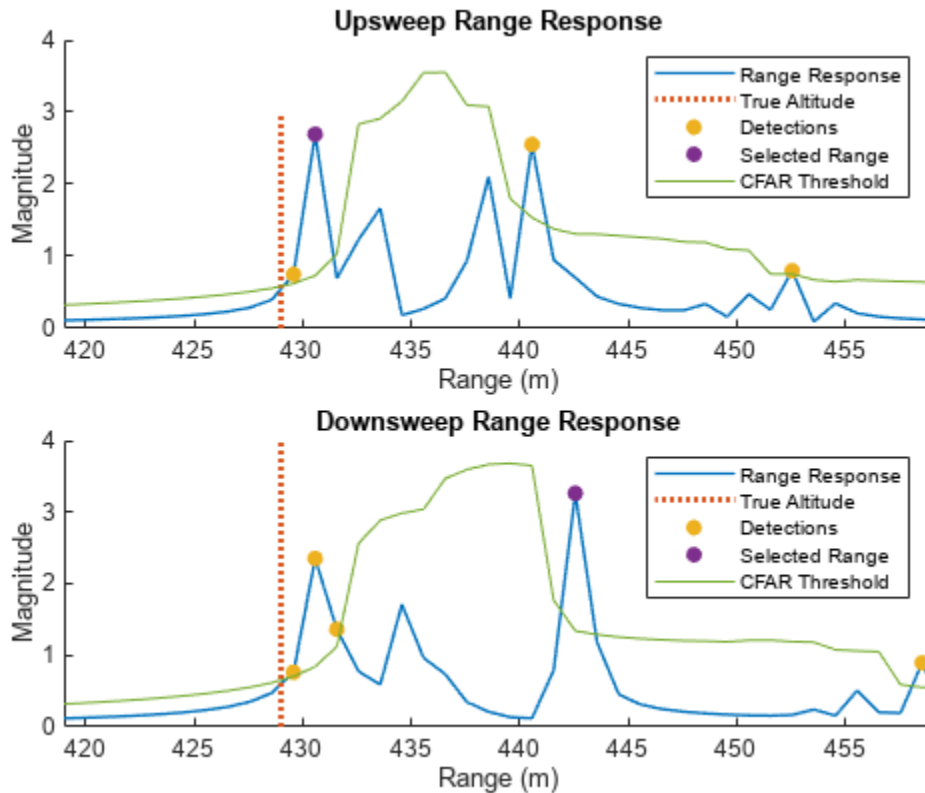
In the plot above, there is a higher power detection that occurs on the trailing edge of the clutter return. This leads to a relatively high measurement error. A similar pattern is seen in other points with large errors This suggests that some further improvements could be made to the altimeter design by fine-tuning the ground detection algorithm.

Summary

This example models a simple radar altimeter and uses surface clutter as a target to determine aircraft altitude. The example starts by building the radar altimeter mode from a transceiver that generates IQ data and a signal processor that generates the ground range estimate from the IQ data. Then, two radar scenarios are simulated and performance of the altimeter is measured in both scenarios.

References

[1] *Operational and Technical Characteristics and Protection Criteria of Radio Altimeters Utilizing the Band 4200-4400 MHz*. Recommendation, M.2059-0, ITU-R, Feb. 2014, https://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.2059-0-201402-I!!PDF-E.pdf.

Supporting Functions

The helperAltimeterSignalProcessor class is shown below.

```
type('helperAltimeterSignalProcessor.m')
```

```
classdef helperAltimeterSignalProcessor
    % This class is used in the Altimeter Simulation example to perform the
```

```
% signal processing for the radar altimeter.

properties
    UpsweepReference
    DownsweepReference
    Detector
    RangeResponse
end

methods
    function obj = helperAltimeterSignalProcessor(refSigUp,refSigDown,sweepSlope,fsWaveform,
        obj.UpsweepReference = refSigUp;
        obj.DownsweepReference = refSigDown;
        decimationFactor = floor(fsWaveform / fsReceiver);

        obj.Detector =  phased.CFARDetector('Method','SOCA','NumTrainingCells',50,'NumGuardCe
        obj.RangeResponse = phased.RangeResponse("RangeMethod","FFT",...
            "SampleRate",fsWaveform,"SweepSlope",sweepSlope,"DechirpInput",true,"DecimationFa
    end

    function [altitude,rngup,rngdn,respup,respdn] = estimateAircraftAltitude(obj,radarSignal

        % Apply CFAR detector to return signal
        [grndrangesup,grndrangesdn,grndmagrespup,grndmagrespdn] = detectGround(obj,radarSigna

        % Find the maximum power detection for upsweep and downsweep
        [rngup,respup] = findMaxMagResp(grndmagrespup,grndrangesup);
        [rngdn,respdn] = findMaxMagResp(grndmagrespdn,grndrangesdn);


        % Estimate altitude as average of up and down sweeps
        if isempty(rngup)
            rngup = -rngdn;
        end
        if isempty(rngdn)
            rngdn = -rngup;
        end

        altitude = (-rngdn + rngup)/2;

        function [rng,rngrspmax] = findMaxMagResp(rngrsp,fullrng)
            [rngrspmax,rngidx] = max(abs(rngrsp));
            rng = fullrng(rngidx);
        end
    end

    function [grndrangesup,grndrangesdn,grndmagrespup,grndmagrespdn,range,upthreshold,downth

        % Get the range response of the radar signal
        [rngrespup,rngrespdn,range] = obj.getRangeResponse(radarSignal);

        % Detect ground ranges above threshold from upsweep
        [upidx,upthreshold] = obj.Detector(abs(rngrespup).^2,1:length(rngrespup));
        grndrangesup = range(upidx);
        grndmagrespup = rngrespup(upidx);

        % Detect ground ranges above threshold from downsweep
        [dnidx,downthreshold] = obj.Detector(abs(rngrespdn).^2,1:length(rngrespdn));
```

```
            grndrangesdn = range(dnidx);
            grndmagrespdn = rngrespdn(dnidx);

            % Take square root to get thresholds because thresholding done
            % on squares
            upthreshold = sqrt(upthreshold);
            downthreshold = sqrt(downthreshold);
        end

        function [rngrespup,rngrespdn,range] = getRangeResponse(obj,radarSignal)

            % Get the range response of the radar signal
            [rngrespup,~] = obj.RangeResponse(radarSignal(:,1),obj.UpsweepReference);
            [rngrespdn,range] = obj.RangeResponse(radarSignal(:,2),obj.DownsweepReference);

        end

        %-------------------------------------------------------------------
        % Visualization Functions
        %-------------------------------------------------------------------

        function [upsweepSubplotAxis,downsweepSubplotAxis] = plotRangeResponse(obj,radarSignal,pl
            % Plot the dechirped signals in the frequency domain.

            [rangeResponseUpsweep,rangeResponseDownsweep,range] = obj.getRangeResponse(radarSigna

            if isfield(plotStruct,'UpDownAxis')
                upsweepSubplotAxis = plotStruct.UpDownAxis(1);
                downsweepSubplotAxis = plotStruct.UpDownAxis(2);
            else
                figure();
                upsweepSubplotAxis = subplot(2,1,1);
                downsweepSubplotAxis = subplot(2,1,2);
            end

            if isfield(plotStruct,'LegendName')
                rangeResponseName = plotStruct.LegendName;
                showLegend = true;
            else
                rangeResponseName = 'Range Response';
                showLegend = false;
            end

            if isfield(plotStruct,'PlotDb')
                rangeResponseUpsweepDb = 20*log10(abs(rangeResponseUpsweep));
                rangeResponseDownsweepDb = 20*log10(abs(rangeResponseDownsweep));
                plotRangeResponse(upsweepSubplotAxis,range,rangeResponseUpsweepDb,'Upsweep Range
                plotRangeResponse(downsweepSubplotAxis,-range,rangeResponseDownsweepDb,'Downsweep
            else
                plotRangeResponse(upsweepSubplotAxis,range,abs(rangeResponseUpsweep),'Upsweep Ra
                plotRangeResponse(downsweepSubplotAxis,-range,abs(rangeResponseDownsweep),'Downsw
            end

            if isfield(plotStruct,'TrueAltitude')
                truthAltitude = plotStruct.TrueAltitude;
                obj.plotTruthAltitude(truthAltitude,upsweepSubplotAxis,downsweepSubplotAxis);
            end
```

```
        if isfield(plotStruct,'Detections')
            obj.plotDetections(radarSignal,upsweepSubplotAxis,downsweepSubplotAxis);
        end

        if isfield(plotStruct,'SelectedRange')
            obj.plotSelectedRange(radarSignal,upsweepSubplotAxis,downsweepSubplotAxis);
        end

        if isfield(plotStruct,'Threshold')
            obj.plotThreshold(radarSignal,upsweepSubplotAxis,downsweepSubplotAxis);
        end

        if isfield(plotStruct,'XLims')
            xlim(upsweepSubplotAxis,plotStruct.XLims)
            xlim(downsweepSubplotAxis,plotStruct.XLims)
        end

        function plotRangeResponse(axis,range,mResponse,plotTitle,yLabel,rangeResponseName,sh
            hold(axis,"on")
            plot(axis,range,mResponse,'DisplayName',rangeResponseName,'LineWidth',1);
            title(axis,plotTitle);
            ylabel(axis,yLabel);
            xlabel(axis,'Range (m)');
            xlim(axis,plotLims);
            if showLegend
                legend(axis,'Location','northeast');
            end
            hold(axis,"off")
        end

    end

    function [upsweepSubplotAxis,downsweepSubplotAxis] = plotTruthAltitude(~,truthAltitude,up
    % This function plots the truth altitude on top of the
    % range response

    plotTruthAltitude(upsweepSubplotAxis,truthAltitude);
    plotTruthAltitude(downsweepSubplotAxis,truthAltitude);

        function plotTruthAltitude(axis,truthFrequency)
            hold(axis,"on")
            ylimit = axis.YLim;
            x = [truthFrequency truthFrequency];
            y = ylimit;
            plot(axis,x,y,':','DisplayName','True Altitude','LineWidth',2);
            legend(axis,'Location','northeast');
            hold(axis,"off")
        end
    end

    function [upsweepSubplotAxis,downsweepSubplotAxis] = plotDetections(obj,radarSignal,upswe
        % Plot the CFAR detections on the axis provided

        [rDetectionsUpsweep,rDetectionsDownsweep,mDetectionsUpsweep,mDetectionsDownsweep] = 

        plotDetections(obj,upsweepSubplotAxis,rDetectionsUpsweep,mDetectionsUpsweep);
        plotDetections(obj,downsweepSubplotAxis,-rDetectionsDownsweep,mDetectionsDownsweep);
```

```
            function plotDetections(obj,axis,rDetection,mDetection)
                obj.plotScatterOnAxis(axis,rDetection,abs(mDetection),'Detections');
            end

        end

        function [upsweepSubplotAxis,downsweepSubplotAxis] = plotThreshold(obj,radarSignal,upswee
            % Plot the CFAR threshold on the axis provided

            [~,~,~,~,range,upthreshold,downthreshold] = obj.detectGround(radarSignal);
            plotThresholdOnAxis(upsweepSubplotAxis,range,upthreshold);
            plotThresholdOnAxis(downsweepSubplotAxis,-range,downthreshold);

            function plotThresholdOnAxis(axis,range,threshold)
                hold(axis,"on");
                plot(axis,range,threshold,'DisplayName','CFAR Threshold');
                legend(axis,'Location','northeast');
                hold(axis,"off");
            end
        end

        function [upsweepSubplotAxis,downsweepSubplotAxis] = plotSelectedRange(obj,radarSignal,up
            % Plot the selected range on the axis provided

            [~,rSelectedUpsweep,rSelectedDownsweep,mSelectedUpsweep,mSelectedDownsweep] = obj.es

            plotSelectedRange(obj,upsweepSubplotAxis,rSelectedUpsweep,mSelectedUpsweep);
            plotSelectedRange(obj,downsweepSubplotAxis,-rSelectedDownsweep,mSelectedDownsweep);

            function plotSelectedRange(obj,axis,fSelected,mSelected)
                obj.plotScatterOnAxis(axis,fSelected,abs(mSelected),'Selected Range');
            end
        end

        function plotScatterOnAxis(~,axis,xdata,ydata,name)
            % Plot a scatter plot on an existing axis without modifying the
            % axis.

            hold(axis,"on");
            scatter(axis,xdata,ydata,'filled','DisplayName',name);
            legend(axis,'Location','northeast');
            hold(axis,'off')
        end

    end

end
```

Various other functions defined below are used throughout this example.

```
function [TruthAltitude, RollAngle] = helperCalculateGroundTruth(surface,platform)
% Calculate the actual altitude and aircraft roll angle based on current aircraft position.

% Terrain elevation height at the current aircraft position
ElevationHeight = surface.height([platform.Position(1);platform.Position(2)]);

% Ground truth altitude
TruthAltitude = platform.Position(3) - ElevationHeight;
```

```matlab
    % Roll angle
    RollAngle = platform.Orientation(2);
end

function flightTrajectory =  helperGetFlightTrajectory(surface,FlightTrajectoryFile,n1,n2,updateI
% Return a flight trajectory with elevation from trajectory data file.

% Load flight trajectory
load(FlightTrajectoryFile,'flightTrajectory');

% Init arrival time
ArrivalTime = 0:size(flightTrajectory,1)-1;

% Add the ground elevation to flight elevation
flightTrajectory = flightTrajectory(n1:updateInterval:n2,:);
groundElevation = surface.height(flightTrajectory(:,1:2)')';
flightTrajectory(:,3) = flightTrajectory(:,3) + groundElevation;

% Generate the flight trajectory
ArrivalTime = ArrivalTime(n1:updateInterval:n2)-ArrivalTime(n1);
stepNumber = numel(ArrivalTime);
orientationEuler = zeros(stepNumber,3);

% If provided, generate a fixed altitude trajectory with bank angle
if nargin > 5
    bankAngle = varargin{1};
    allBankAngles = -bankAngle:2*bankAngle/(stepNumber-1):bankAngle;
    orientationEuler(:,2) = allBankAngles;
    orientationQuat = quaternion(orientationEuler,'rotvecd');
    flightTrajectory(:,3) = flightTrajectory(:,3) - groundElevation;
    flightTrajectory(:,3) = flightTrajectory(1,3);
    flightTrajectory(:,3) = flightTrajectory(:,3) + groundElevation;
    flightTrajectory = geoTrajectory(flightTrajectory,ArrivalTime,'Orientation',orientationQuat)
else
    orientationQuat = quaternion(orientationEuler,'rotvecd');
    flightTrajectory = geoTrajectory(flightTrajectory,ArrivalTime,'Orientation',orientationQuat)
end

end

function [MeasuredAltitude,TruthAltitude,RollAngle,RadarSignal,MaxErrorSignal,MaxErrorIdx] = hel
% Simulate the scenario and return key metrics

% Setup scene
rng(1);
restart(scene);

% Initialize variables
RadarSignal = cell(1,length(radarSignalPoints));
MeasuredAltitude = zeros(1,nSimulationPoints);
TruthAltitude = zeros(1,nSimulationPoints);
RollAngle = zeros(1,nSimulationPoints);
maxMeasurementError = -Inf;
cnt = 1;

% Run the simulation
while advance(scene)
```

```matlab
    % Calculate truth altitude
    [TruthAltitude(cnt), RollAngle(cnt)] = helperCalculateGroundTruth(surface,scene.Platforms{1}

    % Calculate required clutter resolution and set accordingly
    clutter.Resolution = helperDetermineClutterResolution(TruthAltitude(cnt));

    % Receive the iq and track aircraft position and simulation time
    iqsig = receive(scene);
    radarSignal = iqsig{:};

    % Calculate measured altitude
    MeasuredAltitude(cnt) = altimeterSignalProcessor.estimateAircraftAltitude(radarSignal);
    measurementError = abs(MeasuredAltitude(cnt)-TruthAltitude(cnt));

    if (measurementError > maxMeasurementError)
        maxMeasurementError = measurementError;
        MaxErrorSignal = radarSignal;
        MaxErrorIdx = cnt;
    end

    if any(radarSignalPoints == cnt)
        RadarSignal{radarSignalPoints == cnt} = radarSignal;
    end

    cnt = cnt + 1;
end

end

function helperPlotAltimeterAccuracy(TruthAltitude,MeasuredAltitude,xValues,xAxisName,chartTitle
% Plot the actual altimeter accuracy compared to the required accuracy

% Get the accuracy limits for each truth altitude
accuracyLimits = helperGetAccuracyLimits(TruthAltitude);

% Determine whether any of the measured altitudes were too high or too low
measureErrorLow = MeasuredAltitude < accuracyLimits.lowerLimit;
measureErrorHigh = MeasuredAltitude > accuracyLimits.upperLimit;
measureErrorIdx = measureErrorLow | measureErrorHigh;

figure; hold on;

% Plot the accuracy limits
plotAllAccuracyLimits(xValues,accuracyLimits)

plot(xValues,MeasuredAltitude,'-o','DisplayName','Altimeter Altitude', ...
    'color',[0 0.45 0.74])
plot(xValues,TruthAltitude,'-s','DisplayName','Truth Altitude', ...
    'color',[0.85 0.33 0.1])
plot(xValues(measureErrorIdx),MeasuredAltitude(measureErrorIdx),'x','MarkerSize',10, ...
    'linewidth',2,'Color','red','DisplayName','Out of Accuracy Lim.');
ylabel('Altitude Above Ground (m)'); xlabel(xAxisName)
title(chartTitle)
legend('Location','best'); grid on; box on;
xlim([min(xValues) max(xValues)])
ylim([0 max(TruthAltitude)*1.25])
```

```matlab
function plotAllAccuracyLimits(xValues,accuracyLimits)

    % From 0 to 30 m, 1 m of accuracy
    curve1 = zeros(size(xValues));
    curve2 = zeros(size(xValues)) + 30;
    helperPlotAccuracyLimitsSurface(xValues, curve1, curve2, [1 0.84 0],'1m Accuracy')

    % From 30 to 152 m, 3% accuracy
    curve1 = zeros(size(xValues)) + 30;
    curve2 = zeros(size(xValues)) + 152;
    helperPlotAccuracyLimitsSurface(xValues, curve1, curve2, [1 0.88 .47],'3% Accuracy')

    % From 152 to 762 m, 5% accuracy
    curve1 = zeros(size(xValues)) + 152;
    curve2 = zeros(size(xValues)) + 762;
    helperPlotAccuracyLimitsSurface(xValues, curve1, curve2, [1 0.92 0.8],'5% Accuracy')

    % Accuracy limit for the flight
    curve1 = accuracyLimits.lowerLimit(1:end);
    curve2 = accuracyLimits.upperLimit(1:end);
    helperPlotAccuracyLimitsSurface(xValues,curve1, curve2,[0.84 0.91 0.85],'Accuracy Limit')
end

end

function helperPlotAltimeterError(TruthAltitude,MeasuredAltitude,xValues,xAxisName,chartTitle,rev
% Plot the altimeter error

% Get the accuracy limits for each truth altitude
accuracyLimits = helperGetAccuracyLimits(TruthAltitude);
normalizedLimits.lowerLimit = accuracyLimits.lowerLimit - TruthAltitude;
normalizedLimits.upperLimit = accuracyLimits.upperLimit - TruthAltitude;

% Get the measurement error
measurementError = MeasuredAltitude - TruthAltitude;

figure; hold on;

% Accuracy limit for the flight
curve1 = normalizedLimits.lowerLimit;
curve2 = normalizedLimits.upperLimit;

lLower = plot(xValues,curve1,'--','DisplayName','Accuracy Limits','Color','k');
plot(xValues,curve2,'--','DisplayName','Accuracy Limits','Color','k');
lPlot = plot(xValues,measurementError,'-o','DisplayName','Measurement Error', 'color',[0 0.45 0.7

ylabel('Measurement Error (m)'); xlabel(xAxisName)
title(chartTitle)
legend([lLower,lPlot],'Location','best');

if reverseAxis
    set(gca, 'XDir','reverse')
end

end

function accuracy = helperGetAccuracyLimits(altitude)
%% Get the accuracy requirements for each altitude in the altitude
```

```matlab
    % vector based on the ITU document.

    accuracy.lowerLimit = zeros(size(altitude));
    accuracy.upperLimit = zeros(size(altitude));

    % 0 to 100 ft. <= +- 1m
    idx = altitude < 30;
    accuracy.lowerLimit(idx) = altitude(idx) - 1;
    accuracy.upperLimit(idx) = altitude(idx) + 1;

    % 100 to 500 ft. <= 3%
    idx = and((altitude >= 30) , (altitude < 152));
    accuracy.lowerLimit(idx) = altitude(idx) - 0.03*altitude(idx);
    accuracy.upperLimit(idx) = altitude(idx) + 0.03*altitude(idx);

    % 500 to 2500 ft. <= 5%
    idx = and((altitude >= 152), (altitude < 762));
    accuracy.lowerLimit(idx>0) = altitude(idx) - 0.05*altitude(idx);
    accuracy.upperLimit(idx>0) = altitude(idx) + 0.05*altitude(idx);

    % above 2500 --> No limit (let's assume 10%)
    idx = ((altitude >= 762));
    accuracy.lowerLimit(idx>0) = altitude(idx) - 0.1*altitude(idx);
    accuracy.upperLimit(idx>0) = altitude(idx) + 0.1*altitude(idx);

end

function error = helperGetAllowedError(altitude)
    limits = helperGetAccuracyLimits(altitude);
    error = limits.upperLimit - mean([limits.upperLimit,limits.lowerLimit]);
end

function helperPlotAccuracyLimitsSurface(x,curve1, curve2, color,name)
% Plot an accuracy limit surface

x2 = [x, fliplr(x)];
inBetween = [curve1, fliplr(curve2)];
fill(x2, inBetween,color,'DisplayName',name);

end

function clutterResolution = helperDetermineClutterResolution(altitude)
% Set the resolution so that the density of scatters is high enough to
% obtain sufficient results. This is set set that ~4 scatters fall within
% the required system resolution.

requiredResolution = helperGetAllowedError(altitude);
longerRange = altitude + requiredResolution;
clutterCircleRadiusSq = (longerRange)^2-altitude^2;
clutterCircleArea = clutterCircleRadiusSq*pi;
numScatterers = 4;
clutterResolution = sqrt(clutterCircleArea/numScatterers);

end
```

Simulating Radar Signals with Atmospheric Refraction Effects

The performance of a radar system is closely tied to the environment in which it operates. This example will discuss one of the environmental factors that is outside of the system designer's control, namely the atmosphere. In particular, this example will focus on the refraction effects due to the atmosphere and its impact on target parameter estimation.

This example demonstrates errors due to refraction for an L-band surveillance radar operating at a high latitude during winter. It first configures a radar scenario and reviews built-in atmospheric models available in Radar Toolbox™. Next it presents a simple exponential approximation for the refractivity versus height. It then uses these models to generate IQ using `radarScenario` and processes the IQ to generate detections. Lastly, the detections are then used in an analysis showing the refraction errors for the specified geometry.

Define Radar Parameters

Define an L-band monostatic surveillance radar that has coherent processing intervals (CPIs) that are 24 pulses long. The sensor is a stationary system mounted to a tower 10 meters above the surface.

```
% Set random number generator to ensure results are repeatable
rng('default')

% Sensor parameters
freq       = 2e9;  % L-band center frequency (Hz)
numPulses  = 24;   % Number of pulses in a CPI
prf        = 1e3;  % PRF (Hz)
tpd        = 1e-5; % Pulse width (s)
fs         = 3e7;  % Sampling frequency (Hz)
bw         = fs/2; % Bandwidth (Hz)
sensorHgt  = 10;   % Sensor height above the surface (m)
```

Given the bandwidth, determine the range resolution using the bw2rangeres function.

```
bw2rangeres(bw)     % Range resolution (m)
```

```
ans = 9.9931
```

Configure the Radar Scenario

First configure the target positions for the analysis. Within this scenario, 21 different target positions will be considered. Set the target elevation angles to a constant 2 degrees with true slant ranges (also called the geometric range) varying from 1 to 30 km. Set the target radar cross section (RCS) to 20 dBsm, which is representative of a large aircraft.

```
% Target parameters
numTgtPos   = 21; % Target positions
tgtRCSdBsm  = 20; % Target RCS (dBsm)
tgtEl       = 2;  % Target elevation (deg)
tgtSR       = linspace(1,30,numTgtPos)*1e3; % Target slant range (m)
```

Calculate the true target heights over a curved Earth using the `range2height` function with an effective Earth radius set to the true Earth radius. Use the target trajectory to define an appropriate update time for the radar scenario.

```
% Calculate target heights
Rearth   = physconst('EarthRadius'); % True Earth radius (m)
```

```
tgtHgt    = range2height(tgtSR,sensorHgt,tgtEl, ...
    'EffectiveEarthRadius',Rearth);  % Target heights (m)
tgtSpeed = 90;                       % Target speed (m/s)
updateTime = (tgtSR(2) - tgtSR(1))/tgtSpeed;
```

Calculate the flight duration and create a radar scenario.

```
flightDuration = (numTgtPos - 1).*updateTime;
updateRate = 1/updateTime;
scene = radarScenario('UpdateRate',updateRate, ...
    'IsEarthCentered',true,'StopTime',flightDuration);
```

Add the Atmosphere

Simulating the refraction phenomenon requires a model of the atmosphere. For this example, the built-in atmospheres available in the `refractiveidx` function will be used. The `refractiveidx` function offers six ITU reference atmospheric models, namely:

- Standard atmospheric model, also known as the Mean Annual Global Reference Atmosphere (MAGRA)
- Summer high latitude model (higher than 45 degrees)
- Winter high latitude model (higher than 45 degrees)
- Summer mid latitude model (between 22 and 45 degrees)
- Winter mid latitude model (between 22 and 45 degrees)
- Low latitude model (smaller than 22 degrees)

Select the high latitude model in winter. More information about these atmospheric models can be found in the "Modeling Target Position Errors Due to Refraction" on page 1-671 example.

The Central Radio Propagation Laboratory developed a widely used reference atmosphere model that approximates the refractivity profile as an exponential decay versus height. In general, the CRPL model is a good approximation for refractivity profiles under normal propagation conditions. This exponential decay model underlies the CRPL method in the following functions:

- `height2range`
- `height2grndrange`
- `range2height`
- `slant2range`
- `refractionexp`

It also exists as an atmosphere option within a radar scenario.

The CRPL exponential model is given by

$$N = N_S \exp(-c_e h),$$

where N_S is the surface refractivity (N-units), h is the height (km), and c_e is a decay constant (1/km) calculated as

$$c_e = \ln \frac{N_S}{N_S - \Delta N}.$$

In the above equation, ΔN is the difference between the surface refractivity N_S and the refractivity at a height of 1 km. ΔN can be approximated by an exponential function expressed as

$$\Delta N = 7.32\exp(0.005577N_S).$$

The CRPL exponential model is easily modified to match local conditions by setting appropriate surface refractivity and refractive exponent values. Use the surface refractivity output from the `refractiveidx` function as an input to the CRPL model. Calculate the decay constant (refraction exponent) using the `refractionexp` function.

```
% Define CRPL model parameters
[~,N0] = refractiveidx(0,'LatitudeModel','High','Season','Winter');  % Surface refractivity (N-un
rexp   = refractionexp(N0);                                          % Corresponding refraction e
```

Define the atmosphere model of the radar scenario as CRPL. Set the surface refractivity and refraction exponent to the previously obtained values.

```
atmosphere(scene,'CRPL','SurfaceRefractivity',N0,'RefractionExponent',rexp)

ans =
  AtmosphereCRPL with properties:

    SurfaceRefractivity: 313.4066
     RefractionExponent: 0.1440
        MaxNumIterations: 10
               Tolerance: 0.0100
```

Create a Radar Transceiver

First define a sensor trajectory using `geoTrajectory` and add the trajectory to a platform.

```
% Surface parameters
surfHgt    = 52;  % Surface height at sensor (m)

% Setup sensor trajectory
sensoralt  = sensorHgt + surfHgt; % Surface altitude (m)
sensorpos1 = [42.30141195164364,-71.376098592854,sensoralt]; % Natick, MA USA [Lat (deg) Lon (deg
sensorpos2 = sensorpos1;
sensorTraj = geoTrajectory([sensorpos1;sensorpos2],[0;flightDuration]);

% Add sensor platform to scenario
sensorplat = platform(scene,'Trajectory',sensorTraj);
```

Next configure the monostatic radar sensor using `radarTransceiver`. Define the antenna as a rectangular phased array of size 4-by-4 with half-wavelength spacing. Set the waveform modulation to be an LFM.

```
% Configure radarTransceiver
[lambda,c] = freq2wavelen(freq);
elementSpacing = lambda/2;
uraSize = [4 4];
ant = phased.URA('Size',[4 4],'ElementSpacing',elementSpacing);
sensor = radarTransceiver('MountingAngles',[-90 0 0],'NumRepetitions',numPulses);
sensor.Waveform = phased.LinearFMWaveform('SampleRate',fs,'PulseWidth',tpd, ...
    'PRF',prf,'SweepBandwidth',bw);
sensor.Transmitter.Gain = 30;
sensor.Receiver.Gain = 30;
```

```
sensor.Receiver.SampleRate = fs;
sensor.Receiver.NoiseFigure = 0;
sensor.TransmitAntenna.Sensor = ant;
sensor.ReceiveAntenna.Sensor = ant;
sensor.TransmitAntenna.OperatingFrequency = freq;
sensor.ReceiveAntenna.OperatingFrequency = freq;
```

Attach the sensor to the platform.

```
sensorplat.Sensors = sensor;
```

Define Target Trajectory

Convert target range and angle to geodetic coordinates using `aer2geodetic`.

```
% Define a structure for the target truth
tgtTruth.Range = tgtSR;
tgtTruth.Elevation = tgtEl*ones(1,numTgtPos);
tgtTruth.Height = tgtHgt;

% Convert azimuth, elevation, and range to geodetic positions
tgtpos     = zeros(numTgtPos,3);      % Target [lat lon alt]
tgtposTime = zeros(numTgtPos,1);      % Position times
refS       = referenceSphere('earth'); % Earth model
for m = 1:numTgtPos
    [tgtLat,tgtLon,tgtalts] = aer2geodetic(90,tgtEl,tgtSR(m), ...
        sensorpos1(1),sensorpos1(2),sensoralt,refS);
    tgtpos(m,:) = [tgtLat tgtLon tgtalts];
    tgtposTime(m) = (m - 1).*updateTime;
end
```

Define the target trajectory using `geoTrajectory` and attach to a target platform.

```
% Add target platform to scenario
tgtTraj = geoTrajectory(tgtpos,tgtposTime);
platform(scene,'Trajectory',tgtTraj);
```

Visualize the radar and target geometry by using the helper `helperRadarGlobeViewer`. The radar platform is shown as the blue dot and the target platform is shown as the red dot. The white line indicates the constant elevation target trajectory.

```
% Visualize the scenario
gv = helperRadarGlobeViewer;
showScenario(gv,scene)
positionCamera(gv,[42.1417 -71.3920 1.4668e+03],[359.9981 5.3302 31.7642])
```
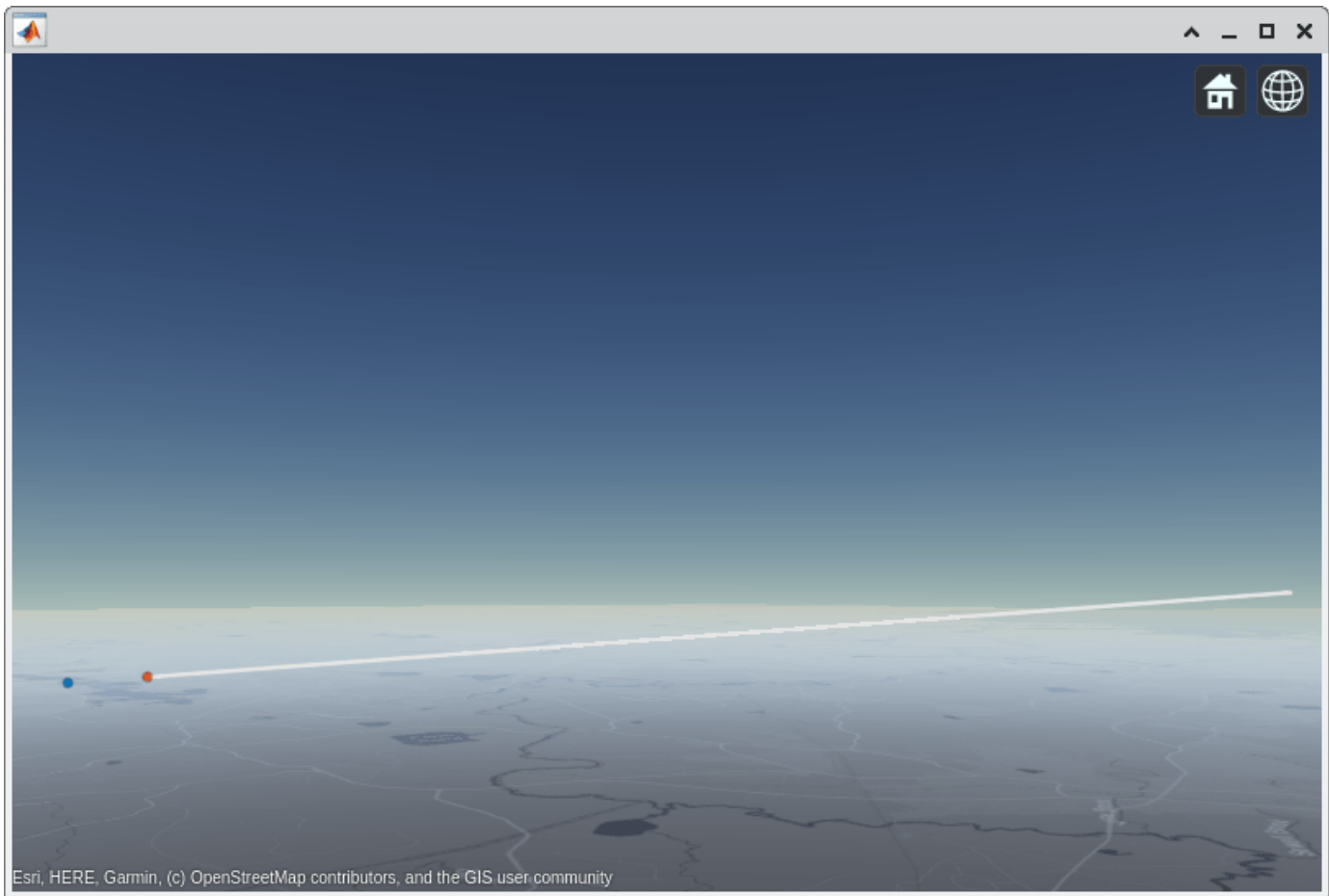
Setup a Signal Processing Chain

In this section, the objects for a signal processing chain will be configured. At a high-level, the signal processing chain proceeds as depicted.
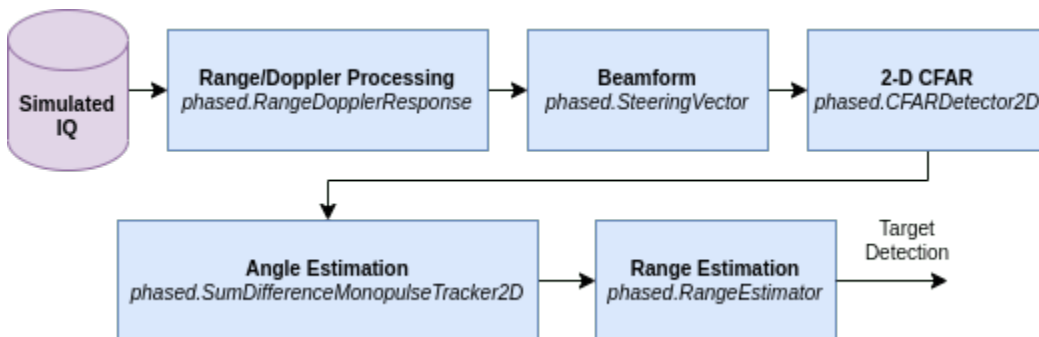
1 First, the simulated IQ is range and Doppler processed.

2 Next, the data is beamformed to broadside.

3 The beamformed data is then processed by a 2-dimensional constant false alarm rate detector (CFAR) to generate detections. Only the highest SNR detection is retained.

4   The azimuth and elevation angles of the target are then estimated using a sum/difference beamformer.

5   Lastly, the range of the target is further refined by quadratic interpolation. The output of this chain is the target detection.

First, configure `phased.RangeDopplerResponse` to create an object to perform range and Doppler processing on the simulated datacube.

```matlab
% Initialize range/Doppler processing objects
matchingCoeff = getMatchedFilter(sensor.Waveform);
rngdopresp = phased.RangeDopplerResponse('SampleRate',fs, ...
    'DopplerFFTLengthSource','Property','DopplerFFTLength',numPulses, ...
    'DopplerOutput','Speed','PRFSource','Property','PRF',prf);
```

Define a steering vector object that will be used to create a range-angle plot, as well as to beamform prior to performing CFAR detection.

```matlab
% Beamforming
steeringvec = phased.SteeringVector('SensorArray',ant);

% Create steering vector for initial range-angle visualization
elVec = -90:90;
sv = steeringvec(freq,[zeros(1,numel(elVec)); elVec]);
```

Next, define a 2-D CFAR with 3 guard cells and 15 training cells on each side of the cell under test in range and 1 guard cell and 3 training cells in Doppler. Set the false alarm rate to 1e-6.

```matlab
% 2-D CFAR
cfar = phased.CFARDetector2D('GuardBandSize',[3 1], ...
    'TrainingBandSize',[15 3], ...
    'ProbabilityFalseAlarm',1e-6,'OutputFormat','Detection index','NoisePowerOutputPort',true);
```

Define a sum/difference beamformer for angle estimation and create a range estimator object for improved range estimation.

```matlab
% Angle estimation
sumdiff = phased.SumDifferenceMonopulseTracker2D('SensorArray',ant,'OperatingFrequency',freq);

% Range estimation object
rangeestimator = phased.RangeEstimator;
```

Simulate IQ and Create Detections

Simulating the IQ is as simple as calling the `receive` method on the radar scenario scene. Advance the scenario and collect IQ from `radarTransceiver`.

```matlab
% Initialize simulation
numSamples = 1/prf*fs;
dets = struct('Range',zeros(1,numTgtPos), ...
    'Angles',zeros(2,numTgtPos), ...
    'SNRdB',zeros(1,numTgtPos));
for m = 1:numTgtPos
    % Advance scene
    advance(scene);

    % Update geometry visualization
    updateDisplay(gv,scene.SimulationTime,[scene.Platforms{:}]);
```

```matlab
    positionCamera(gv,[42.1417 -71.3920 1.4668e+03],[359.9981 5.3302 31.7642]);
    drawnow;
    pause(0.1);

    % Collect IQ
    tmp = receive(scene);
    iqsig = tmp{1};
```

Range and Doppler process the simulated IQ.

```matlab
    % Range-Doppler process
    [resp,rngVec,dopVec] = rngdopresp(iqsig,matchingCoeff);
```

Next, beamform the data to get a sense of the range-angle performance.

```matlab
    % Beamform across all angles
    respAng = (sv'*squeeze(max(resp,[],3)).');
```

Plot the range-angle and range-Doppler data.

```matlab
    % Convert to dB
    angRespMagdB = mag2db(abs(respAng));
    respMagdB = mag2db(abs(resp));

    if m == 1
        % Initialize range-angle plot
        figure
        tiledlayout(1,2)
        nexttile
        hRAMaxes = gca;
        hRAMplot = pcolor(rngVec*1e-3,elVec,angRespMagdB);
        hRAMplot.EdgeColor = 'none';
        hRAMplot.FaceColor = 'interp';
        ylabel('Elevation Angles (deg)')
        xlabel('Range (km)')
        title('Range-Angle')
        hC = colorbar;
        hC.Label.String = 'Magnitude (dB)';

        % Initialize range-Doppler plot
        nexttile
        hRDMaxes = gca;
        hRDMplot = pcolor(rngVec*1e-3,dopVec,squeeze(max(respMagdB,[],2)).');
        hRDMplot.EdgeColor = 'none';
        hRDMplot.FaceColor = 'interp';
        ylabel('Speed (m/s)')
        xlabel('Range (km)')
        title('Range-Doppler')
        hC = colorbar;
        hC.Label.String = 'Magnitude (dB)';
    else
        % Update range-angle plot
        hRAMplot.CData = angRespMagdB;

        % Update range-Doppler plot
        hRDMplot.CData = squeeze(max(respMagdB,[],2)).';
    end
    xlim(hRAMaxes,[max((tgtSR(m) - 2e3),rngVec(1)) min((tgtSR(m) + 2e3),rngVec(end))]*1e-3)
```

```
xlim(hRDMaxes,[max((tgtSR(m) - 2e3),rngVec(1)) min((tgtSR(m) + 2e3),rngVec(end))]*1e-3)

drawnow limitrate
pause(0.25)
```

Finally, detect and estimate the target range and angle using the helper function `helperDetectAndEstimate`. This helper function contains the broadside beamforming, CFAR detection, and angle and range estimation processing steps.

```
% Detect targets and estimate range and angle
dets = helperDetectAndEstimate(cfar,sumdiff,steeringvec,rangeestimator, ...
    m,rngVec,dopVec,resp,dets);
end
```
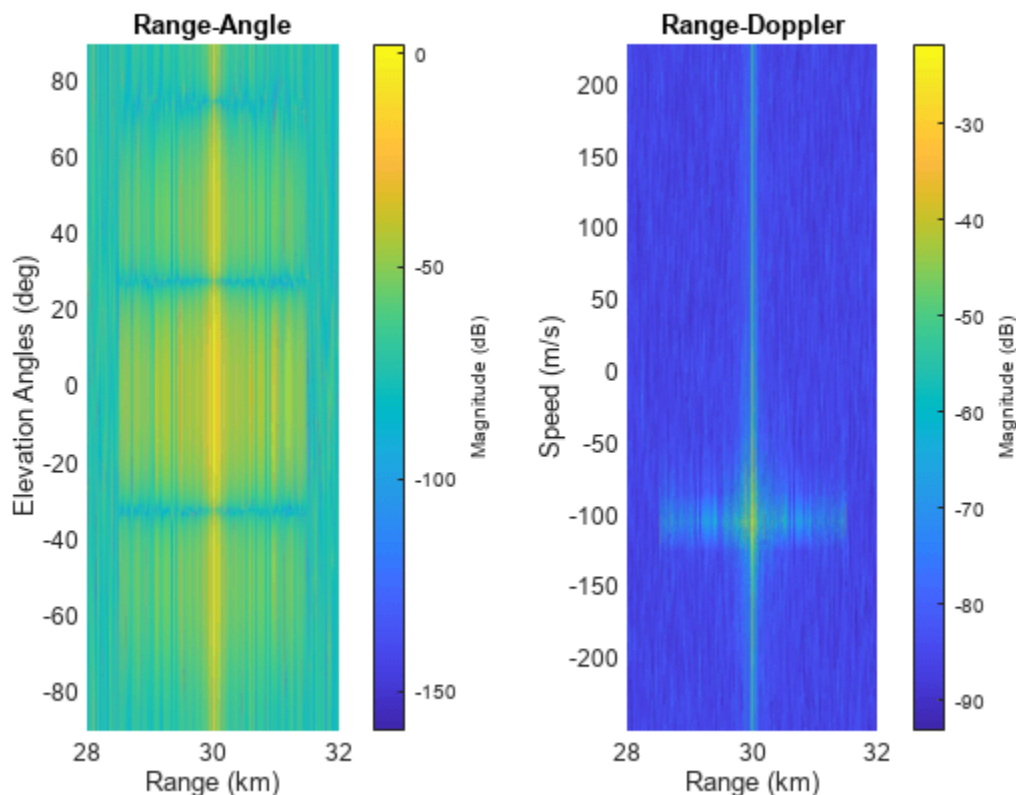

Assess the Refraction Errors

Predict the range and elevation that will be output from the detector including refraction effects. For the calculation of the predicted range and elevation angle, use the `slant2range` function with the CRPL method selected.

```
% Estimate propagated range using CRPL
[predicted.Range,predicted.Elevation] = slant2range( ...
    tgtTruth.Range,sensorHgt,tgtTruth.Height, ...
    'Method','CRPL');
```

Now calculate the range error. The range error is the difference between the propagated range and the true slant range. The detected errors in blue follow the predicted range errors, shown as the

dashed line in the figure. Note that the IQ processing introduced an additional source of bias. This bias can be attributed to factors like:

- The range resolution of the system, which is fairly large at about 10 m and
- Range bin straddling.

```
% Calculate the range error (m)
co = colororder;
figure('Name','Range Error (m)')
plot(tgtSR*1e-3,dets.Range - tgtTruth.Range,'o','MarkerFaceColor',co(1,:),'LineWidth',1.5)
hold on
grid on
plot(tgtSR*1e-3,predicted.Range - tgtTruth.Range,'--k','LineWidth',1.5)
xlabel('Slant Range (km)')
ylabel('Range Error (m)')
title('Range Error (m)')
legend('Detected Range Error','Predicted Range Error','Location','Best')
```
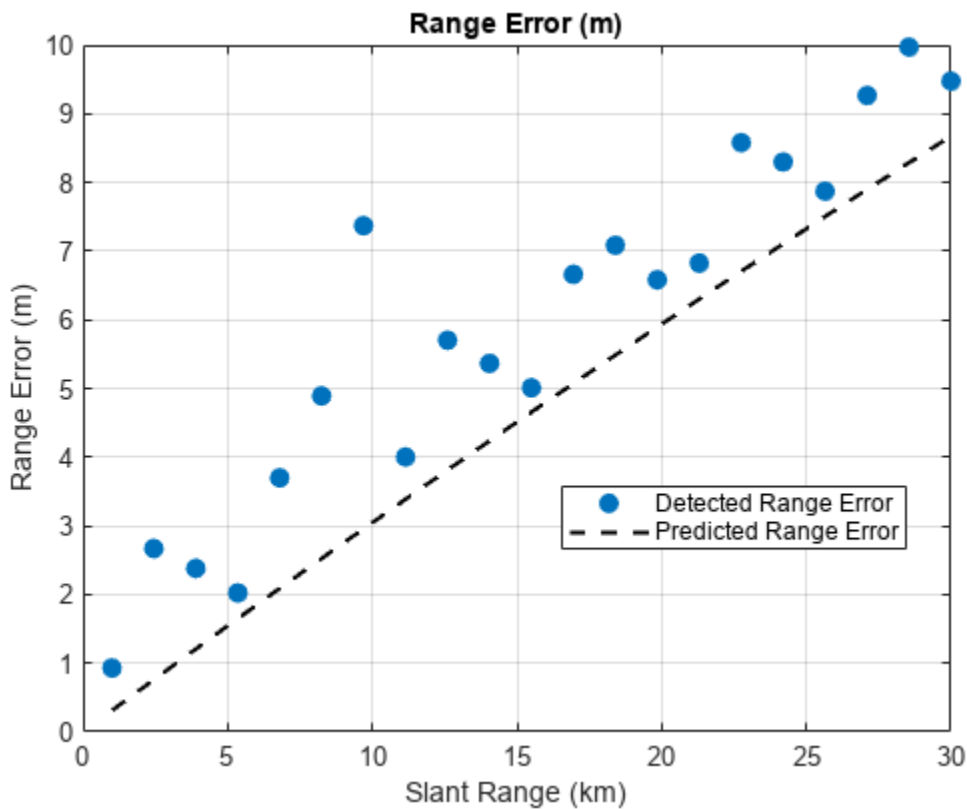
Next calculate the elevation error. The elevation error is the difference between the detected elevation angle and the true elevation angle. Note that the detected errors match the predicted elevation errors well. As the target moves farther out in range, the elevation errors increase.

```
% Calculate the elevation angle error (deg)
figure('Name','Elevation Error (deg)')
plot(tgtSR*1e-3,dets.Angles(2,:) - tgtTruth.Elevation,'o','MarkerFaceColor',co(1,:),'LineWidth',1
hold on
grid on
```

```
plot(tgtSR*1e-3,predicted.Elevation - tgtTruth.Elevation,'--k','LineWidth',1.5)
xlabel('Slant Range (km)')
ylabel('Elevation Error (deg)')
title('Elevation Error (deg)')
legend('Detected Elevation Error','Predicted Elevation Error','Location','Best')
```
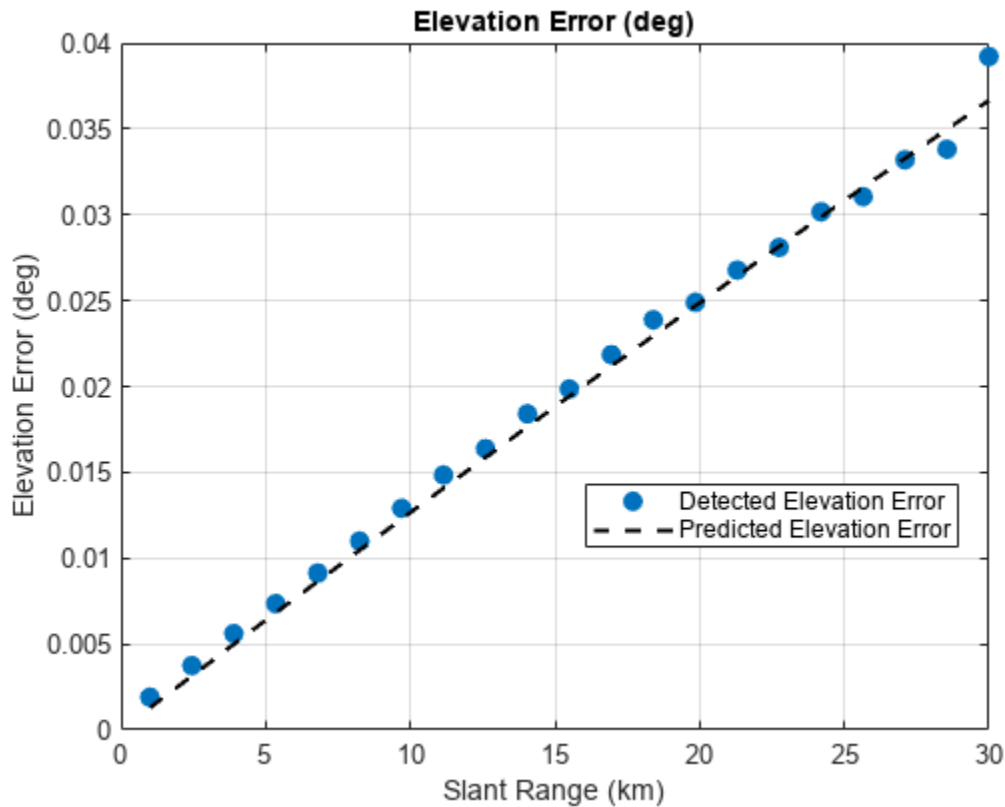

Next, calculate the height error. The height error is the difference between the apparent height, which is the height calculated assuming no refraction, and the true target height. To calculate the apparent height, use the `range2height` function with the true Earth radius. Note that the detected height errors match the predicted height errors well. As the target moves farther out in range, the height errors slowly increase. At the farthest range, height errors of about 20 m can be seen.

```
% Calculate apparent height from detections
dets.Height = range2height(dets.Range,sensorHgt,dets.Angles(2,:), ...
    'EffectiveEarthRadius',Rearth);

% Calculate apparent height from predicted path
predicted.Height = range2height(predicted.Range,sensorHgt,predicted.Elevation, ...
    'EffectiveEarthRadius',Rearth);

% Plot height errors (m)
figure('Name','Height Error (m)')
plot(tgtSR*1e-3,dets.Height - tgtTruth.Height,'o','MarkerFaceColor',co(1,:),'LineWidth',1.5)
hold on
grid on
plot(tgtSR*1e-3,predicted.Height - tgtTruth.Height,'--k','LineWidth',1.5)
xlabel('Slant Range (km)')
```

```matlab
ylabel('Height Error (m)')
legend('Detected Height Error','Predicted Height Error','Location','Best')
```
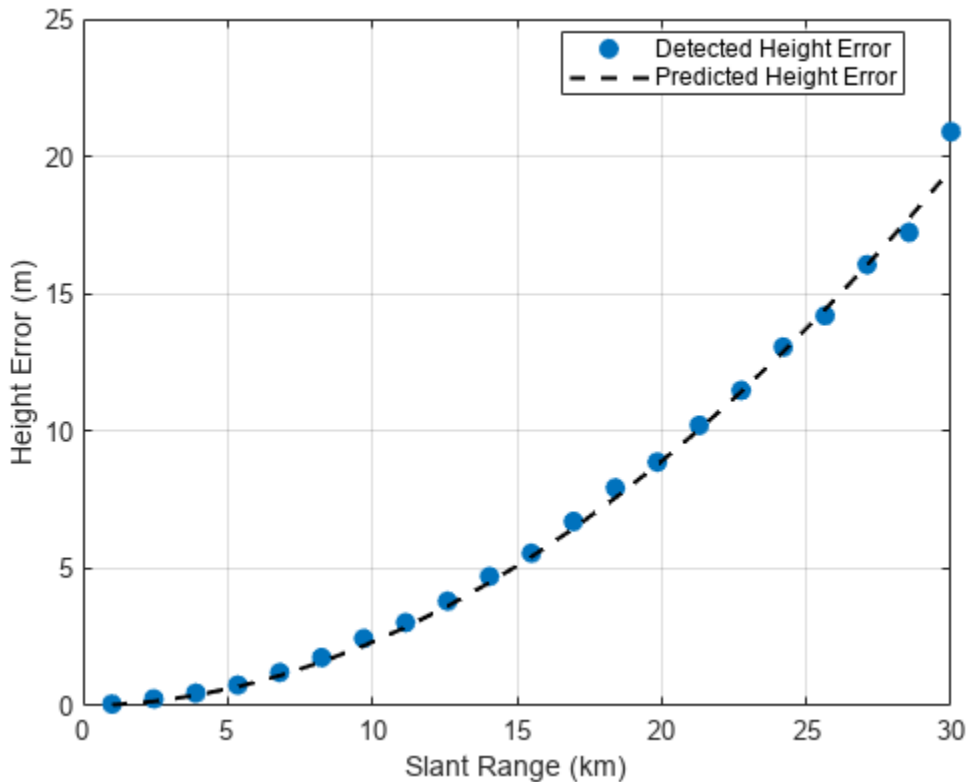

Summary

This example demonstrated how to model refraction for an L-band surveillance radar operating at a high latitude during winter. It also demonstrated how to configure and implement a signal processing chain to detect the target. Lastly, it discussed the target detection errors due to refraction.

Helpers

helperDetectAndEstimate

```matlab
function dets = helperDetectAndEstimate(cfar,sumdiff,steeringvec,rangeestimator, ...
    tgtIdx,rngVec,dopVec,iqPCDP,dets)
% Detection processing. Performs the following:
% - Broadside beamforming
% - 2D CFAR
% - Angle estimation
% - Range estimation

% Number of bins
numRngBins = numel(rngVec);
numDopBins = numel(dopVec);

% Convert datacube to power
freq = sumdiff.OperatingFrequency; % Operating frequency (Hz)
sv = steeringvec(freq,[0;0]); % Broadside
```

```
        iqPCDPtmp = permute(iqPCDP,[2 1 3]); % <elements x range x Doppler>
        iqPCDPBF = (sv'*reshape(iqPCDPtmp,[],numRngBins*numDopBins)); % Beamform <beams x range x Doppler
        iqPow = abs(squeeze(iqPCDPBF)).^2; % Power (Watts)
        iqPow = reshape(iqPow,numel(rngVec),numel(dopVec)); % Power <range x Doppler>

        % Run 2D CFAR. The indices of the cells under test for the CFAR must
        % permit the stencil size.
        numRngStencil = cfar.GuardBandSize(1) + cfar.TrainingBandSize(1);
        numDopStencil = cfar.GuardBandSize(2) + cfar.TrainingBandSize(2);
        idxRngTest = (numRngStencil + 1):(numel(rngVec) - numRngStencil);
        idxDopTest = (numDopStencil + 1):(numel(dopVec) - numDopStencil);
        [idxRngGrid,idxDopGrid] = meshgrid(idxRngTest,idxDopTest);
        [idxDet,noisePwr] = cfar(iqPow,[idxRngGrid(:).'; idxDopGrid(:).']);

        % Assume maximum detection SNR is target
        idxDetLinear = sub2ind(size(iqPow),idxDet(1,:),idxDet(2,:));
        snrDet = iqPow(idxDetLinear)./noisePwr;
        [~,idxMax] = max(snrDet);
        idxRng = idxDet(1,idxMax);
        idxDop = idxDet(2,idxMax);

        % Estimate true angle. Assume sensor is aligned with target.
        dets.Angles(:,tgtIdx) = sumdiff(squeeze(iqPCDP(idxRng,:,idxDop)), ...
            [0; 0]);
        dets.Angles(2,tgtIdx) = -dets.Angles(2,tgtIdx);

        % Beamform to detected angle
        sv = steeringvec(freq,dets.Angles(:,tgtIdx));
        iqPCDPBF = (sv'*squeeze(iqPCDP(:,:,idxDop)).');

        % Estimate SNR and range
        dets.SNRdB(tgtIdx) = pow2db(abs(squeeze(iqPCDPBF(idxRng))).^2./noisePwr(idxMax));
        dets.Range(tgtIdx) = rangeestimator(iqPCDPBF(:),rngVec,idxRng);
        end
```

Multibeam Radar for Adaptive Search and Track

This example shows how to use `radarDataGenerator` for a closed-loop simulation of a multifunction phased array radar (MPAR). The example starts by defining MPAR system parameters and parameters of three radar tasks: volume search, cued search, and tracking. The example then sets up a scenario with multiple maneuvering targets. At each update interval the tasks request resources from the MPAR to search for targets and revisit existing tracks. The example shows how different radar tasks that share the radar bandwidth and aperture can be executed in parallel. The simulation loop is closed by passing target detections back to the radar tasks to update their internal state and generate new look requests.

MPAR Simulation Set Up

The MPAR in this example can carry out three tasks: *volume search*, *cued search*, and *tracking*. The goal of the volume search is to repeatedly scan a large volume of space searching for targets. The volume search task uses a wide beam and a narrow band waveform to reduce the total search time. The detections obtained by the volume search provide only approximate target locations. These detections are passed to the cued search task. The cued search allows the MPAR to perform surveillance in an adaptive fashion by scanning a small volume around a detection produced by the volume search. It uses a narrower beam and a wider bandwidth to provide better accuracy. The detections produced by the cued search are passed to the tracking task. If the tracking task confirms these detections, it initiates new target tracks. The goal of the tracking task is to accurately track all targets inside the search volume. The tracking task performs adaptive tracking by deciding when to update the tracks based on the estimated target dynamics such that the maneuvering targets are updated more frequently than non-maneuvering.
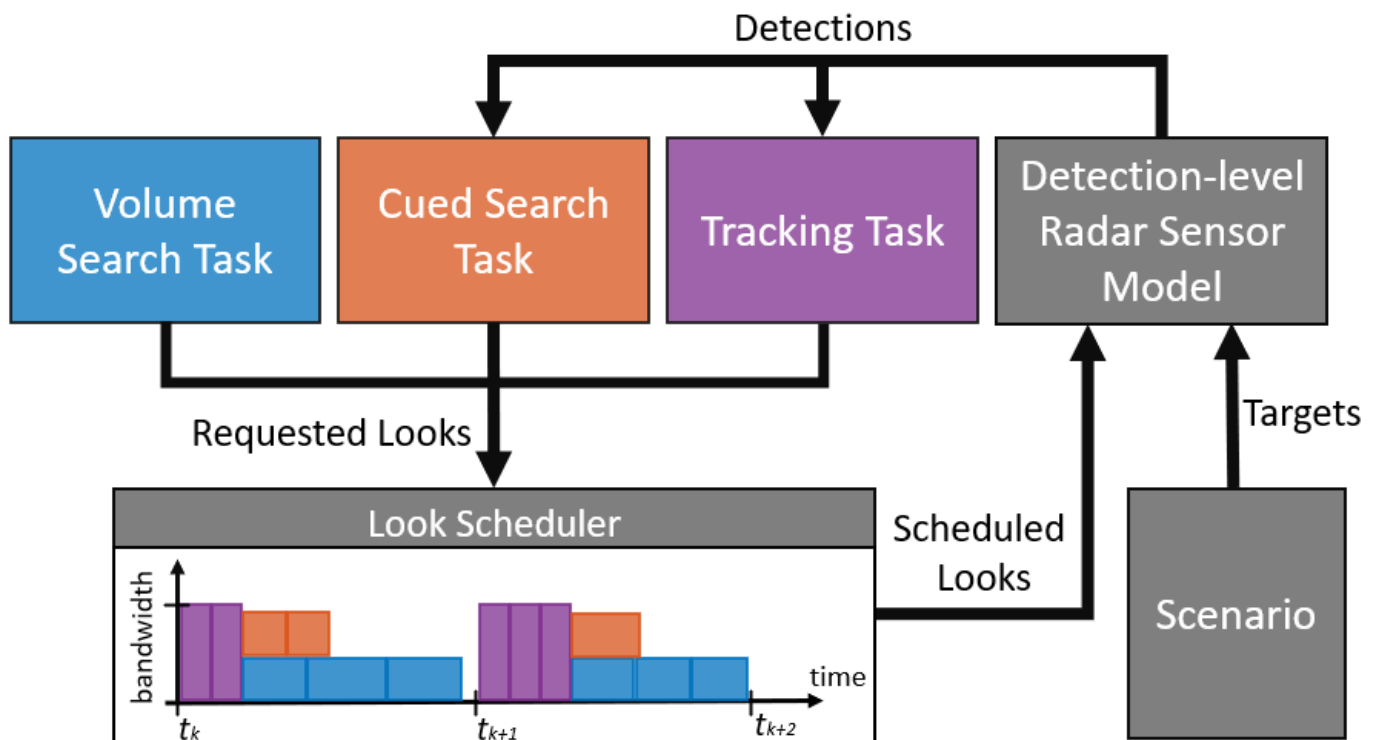
The MPAR can partition its phased array into several subarrays to create multiple simultaneous beams. This allows MPAR to perform multiple tasks in parallel and transmit different waveforms that are the most appropriate for each task. Each radar task requires access to some amount of bandwidth, power-aperture product, and time resources of the MPAR system. Since these resources are limited, the radar tasks must compete with each other for the access to them. At each update interval a radar task produces look requests asking the MPAR for resources required to achieve the task goals given the current operational situation (that is, the scenario, the number of targets, their positions, and dynamics). A radar look contains information about when, where, and for how long the MPAR should steer a beam, the parameters of the steered beam, and the parameters of the transmitted waveform. The *look scheduler* arranges a subset of the generated looks into a transmittable timeline such that the system resource constraints are satisfied [1]. The *radar sensor model* then executes the selected looks and produces target detections. The generated detections are passed back to either the cued search task or the tracking task based on the look that produced the detections. Each radar task has its own idea about the current operational situation and uses the generated detections to update its parameters and to request new looks at the next update interval. The looks that were not selected at the current update interval are considered again at the next update interval.

MPAR Parameters

Set the peak transmit power of the MPAR to 100 kW, the wavelength to 0.09 m, and the noise figure to 5 dB. Let the azimuth and the elevation beamwidths of the phased array, when pointed to boresight, be 2 and 3 degrees, respectively. Set the bandwidth of the system to 10 MHz.

```
mparPrms.PeakPower = 100e3;                 % Peak power (W)
mparPrms.Wavelength = 0.09;                 % Wavelength (m)
mparPrms.NoiseFigure = 5;                   % Noise figure (dB)
mparPrms.Beamwidth = [2; 3];                % Min possible beamwidth [Az; El] (deg)
mparPrms.Bandwidth = 10e6;                  % Max available bandwidth (Hz)
```

System Resources

MPAR can control the individual elements of the phased array antenna on a dwell-to-dwell basis. This allows it to change the transmitted waveform parameters and to partition the phased array into multiple subarrays, possibly creating several simultaneous beams transmitting different waveforms [2].

In this example the tracking task uses the entire available bandwidth and power-aperture product of the system to produce accurate detections needed for tracking. At the same time, the volume search and the cued search tasks use only a portion of the available bandwidth and the array aperture such that the looks from these two tasks can be executed simultaneously. Each radar task requires a different amount of bandwidth, power-aperture, and time resources.

Bandwidth

Set the bandwidth of the volume search waveform to 0.5 MHz and the cued search waveform to 3 MHz.

```
volumeSearchPrms.Bandwidth = 0.5e6;         % Volume search bandwidth (Hz)
cuedSearchPrms.Bandwidth = 3e6;             % Cued search bandwidth (Hz)
```

Verify that the total bandwidth needed to transmit the volume and the cued search waveforms at the same time is less than the total available bandwidth allowing the looks from these tasks to be executed in parallel.

```
volumeSearchPrms.Bandwidth + cuedSearchPrms.Bandwidth <= mparPrms.Bandwidth
```

```
ans = logical
   1
```

Set the tracking task to use the entire system bandwidth of 10 MHz.

```
trackingPrms.Bandwidth = mparPrms.Bandwidth;     % Tracking bandwidth (Hz)
```

This means that other tasks cannot be executed in parallel with the tracking task because all the available bandwidth is occupied by the tracking waveform.

Power-Aperture Product

The MPAR performs the volume and the cued search simultaneously by partitioning its array into two subarrays. The first subarray transmits the volume search waveform. Its beamwidth is equal to the beamwidth required by the volume search task. The second subarray transmits the cued search waveform and has a beamwidth equal to the cued search task beamwidth. Thus, in addition to the frequency bandwidth, the MPAR must also have a sufficient array aperture and transmit power. The power-aperture product used by the volume and the cued search tasks together must be less or equal to the total power-aperture product of the MPAR.

The power-aperture product PAP is a product of the average transmitted power P_{av} and the subarray aperture size A [3]. It also can be expressed in terms of the peak power P_t the pulse width τ and the pulse repetition frequency PRF

$$PAP = P_{av}A(bw_{az}, bw_{el}) = P_t\tau \cdot PRF \cdot A(bw_{az}, bw_{el})$$

where

and $A(bw_{az}, bw_{el})$ is the aperture size of a subarray with the azimuth and the elevation beamwidths of bw_{az} and bw_{el} respectively.

Since the tracking task uses the entire aperture of the radar, compute its required power-aperture product first. Assume that this is also the total power-aperture product of the entire MPAR system. For that, set the tracking task beamwidth to the minimum beamwidth achieved by the MPAR.

```
trackingPrms.Beamwidth = mparPrms.Beamwidth;     % Tracking beamwidth
```

Let the tracking waveform have pulse width of 2.5 μs and PRF of 1500 Hz.

```
trackingPrms.Pulsewidth = 2.5e-6;                % Tracking pulse width (sec)
trackingPrms.PRF = 1500;                         % Tracking PRF (Hz)
```

Use `helperPowerApertureProduct` function to find the power-aperture product (in W·m²) needed for the tracking task.

```
trackingAperture = beamwidth2ap(trackingPrms.Beamwidth, mparPrms.Wavelength);
trackingPAP = helperPowerApertureProduct(mparPrms.PeakPower, trackingPrms.Pulsewidth, ...
    trackingPrms.PRF, trackingAperture)
```

```
trackingPAP = 1.6619e+03
```

Now set the corresponding volume search and cued search parameters.

```
volumeSearchPrms.Beamwidth = [8; 10];            % Volume search beamwidth [Az; El] (deg)
volumeSearchPrms.Pulsewidth = 10e-6;             % Volume search pulse width (sec)
```

```
volumeSearchPrms.PRF = 1500;                      % Volume search PRF (Hz)

cuedSearchPrms.Beamwidth = [4; 5];                % Cued search beamwidth [Az; El] (deg)
cuedSearchPrms.Pulsewidth = 5e-6;                 % Cued search pulse width (sec)
cuedSearchPrms.PRF = 1500;                        % Cued search PRF (Hz)
```

The volume search uses a much wider beam and a much narrower bandwidth compared to the tracking task. This allows it to quickly traverse the search volume. The resultant detections will have poor accuracies and should not be used for tracking. The MPAR cues a local search around the produced detections with a narrower beam and a wider bandwidth waveform to improve accuracy of the initial detection.

Compute the power-aperture products needed for the volume and the cued search tasks.

```
volumeSearchAperture = beamwidth2ap(volumeSearchPrms.Beamwidth, mparPrms.Wavelength);
volumeSearchPAP = helperPowerApertureProduct(mparPrms.PeakPower, volumeSearchPrms.Pulsewidth, ..
     volumeSearchPrms.PRF, volumeSearchAperture)
```

```
volumeSearchPAP = 498.5762
```

```
cuedSearchAperture = beamwidth2ap(cuedSearchPrms.Beamwidth, mparPrms.Wavelength);
cuedSearchPAP = helperPowerApertureProduct(mparPrms.PeakPower, cuedSearchPrms.Pulsewidth, ...
     cuedSearchPrms.PRF, cuedSearchAperture)
```

```
cuedSearchPAP = 997.1524
```

Check that the sum of the volume search and the cued search power-aperture products is no more than the total available power-aperture product.

```
volumeSearchPAP+cuedSearchPAP <= trackingPAP
```

```
ans = logical
   1
```

Thus, the MPAR has enough power-aperture product and bandwidth for the volume and cued search radar looks to be executed in parallel.

Time

If two or more looks are not sharing the total aperture and bandwidth, they must be executed successively in time. Prior to the execution, the MPAR arranges the looks into a time window of a fixed duration. Hence, the radar looks must compete for time slots within this window [1]. The execution time required by each look is determined by the dwell time, which in turn is determined by the detection requirements, the PRF, and the number of transmitted pulses. The number of transmitted pulses for a radar look that points a beam at a target located at a range R, azimuth az, and elevation el, can be computed using the radar equation [3]

$$n = \frac{(4\pi)^3 k T_s D_x(P_d, P_{fa}) R^4 L(az, el)}{P_t \tau G^2(bw_{az}, bw_{el}) \lambda^2 \sigma}$$

where

λ is the wavelength;

σ is the target radar cross section (RCS);

$G(bw_{az}, bw_{el})$ is the gain of a subarray with the azimuth and the elevation beamwidths of bw_{az} and bw_{el} respectively;

$D_x(P_d, P_{fa})$ is the radar detectability factor that determines the received signal energy needed to achieve the desired probability of detection of P_d given that the required probability of false alarm is P_{fa};

$L(az, el) = \cos^{-3}(az)\cos^{-6}(el)$ is the array scan loss due to scanning off boresight to the angles $[az, el]$;

T_s is the system noise temperature;

and k is the Boltzmann constant.

It is evident from the above equation that to maintain the same detection performance, the radar looks that point the beam at off-boresight angles would have to transmit more pulses because of the array scan loss. Similarly, the volume and the cued search tasks, which use wider beams compared to the tracking task, must generate looks with more pulses to achieve the same probability of detection because of the smaller antenna gain.

Set the desired probability of detection for all tasks to 0.9 and the required probability of false alarm to 1e-6. Assume that the radar tasks do not have any knowledge of the target RCS and the number of pulses is computed for a reference non-fluctuating target with a 1 m² RCS.

```
% The desired probability of detection is the same for all tasks
volumeSearchPrms.DetectionProbability = 0.9;
cuedSearchPrms.DetectionProbability = 0.9;
trackingPrms.DetectionProbability = 0.9;

% The required probability of false alarm is the same for all tasks
volumeSearchPrms.FalseAlarmProbability = 1e-6;
cuedSearchPrms.FalseAlarmProbability = 1e-6;
trackingPrms.FalseAlarmProbability = 1e-6;

% The reference target RCS is the same for all tasks
volumeSearchPrms.ReferenceRCS = 1;
cuedSearchPrms.ReferenceRCS = 1;
trackingPrms.ReferenceRCS = 1;
```

Radar Tasks

The radar tasks are iterative processes that at each update interval can request a varying amount of looks depending on the current operational situation. A radar look is an object containing information about when, where, and for how long a radar task wants the MPAR to point a beam.

```
% A radar look generated by the volume search task. The starting time is 0
% seconds, the look angle is -20 deg in azimuth and -12 deg in elevation,
% and the number of transmitted pulses is set to 10.
helperRadarLook("VolumeSearch", 0, [-20; -12], 10, volumeSearchPrms)

ans =
  helperRadarLook with properties:

              Type: VolumeSearch
         StartTime: 0
         LookAngle: [2x1 double]
         NumPulses: 10
```

```
        DetectableTrackID: [0x1 uint32]
                 Priority: 1
                DwellTime: 0.0067
                Bandwidth: 500000
                Beamwidth: [2x1 double]
               Pulsewidth: 1.0000e-05
                      PRF: 1500
     DetectionProbability: 0.9000
    FalseAlarmProbability: 1.0000e-06
             ReferenceRCS: 1
```

Volume Search

The volume search task places beams at the predetermined beam positions that tile the entire search volume. Set the search volume range limit to 75 km and set the angular limits to -60 and 60 degrees in azimuth and to 0 and -30 degrees in elevation.

```
volumeSearchPrms.RangeLimit = 75e3;             % Search volume range limit (km)
volumeSearchPrms.ScanLimits = [-60 60; 0 -30];  % Search volume angular limits [Az; El] (deg)
```

To improve the chances of detecting targets positioned away from the beam centers let the adjacent beams overlap slightly by making the spacing between them less than the beamwidth.

```
volumeSearchPrms.BeamSpace = [0.85; 0.85];      % Spacing between the adjacent beams specified in
```

Create the volume search task using `helperVolumeSearchTask` class.

```
volumeSearchTask = helperVolumeSearchTask(volumeSearchPrms, mparPrms)

volumeSearchTask =
  helperVolumeSearchTask with properties:

            BeamPositions: [2x56 double]
                BeamSpace: [2x1 double]
               RangeLimit: 75000
               ScanLimits: [2x2 double]
                Bandwidth: 500000
                Beamwidth: [2x1 double]
                      PRF: 1500
               Pulsewidth: 1.0000e-05
     DetectionProbability: 0.9000
    FalseAlarmProbability: 1.0000e-06
             ReferenceRCS: 1
                 Priority: 1
```

Use the `helperPlotSearchGrid` function to visualize the beam positions of the volume search task together with the corresponding 3 dB contours of the volume search beams.

```
figure;
helperPlotSearchGrid(volumeSearchTask.BeamPositions, volumeSearchPrms.Beamwidth, ...
    'ScanLimits', volumeSearchPrms.ScanLimits, 'SearchTaskName', 'Volume search', 'Color', [0 0.4
legend('location', 'best')
title("Search Grid")
```
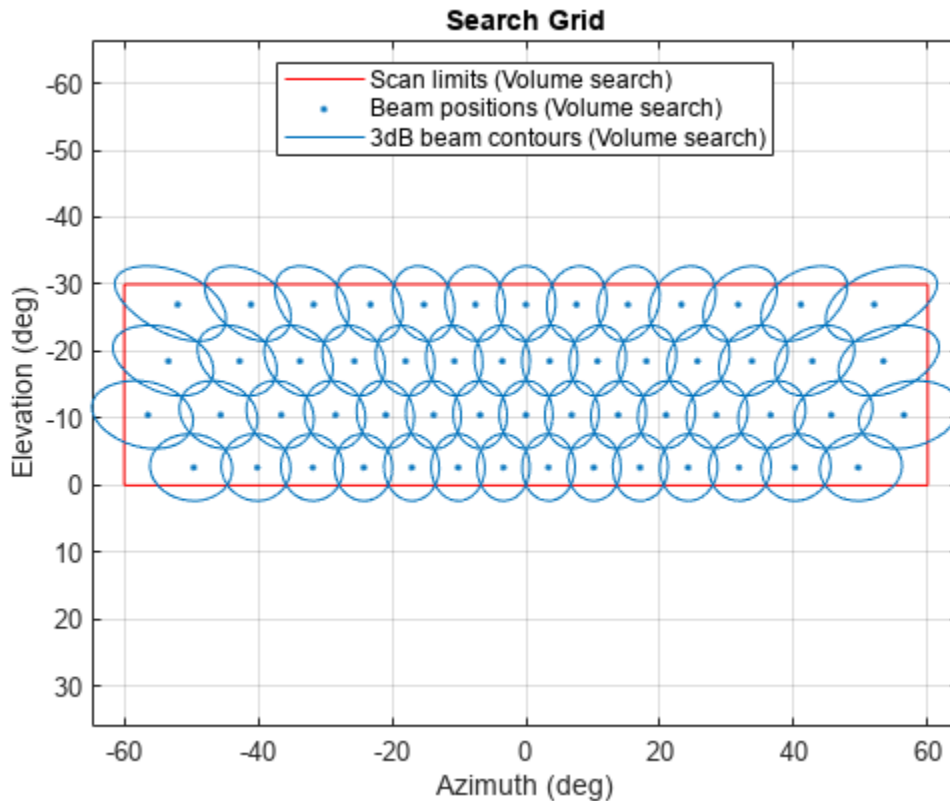
Notice the beam broadening effect when the beam points at off-boresight angles.

Cued Search

A separate cued search task allows the MPAR to adapt to the scenario and the operational conditions when searching for targets. The cued search scans a small volume around a detection produced during a volume search look. This means that the cued search task creates look requests at the current update interval only if the volume search task produced detections at the previous update interval. Otherwise, the cued search task is idle and does not create any look requests. Create the cued search task using `helperCuedSearchTask` class.

```
cuedSearchTask = helperCuedSearchTask(cuedSearchPrms, mparPrms, volumeSearchPrms)
```

```
cuedSearchTask =
  helperCuedSearchTask with properties:

            SearchGridUV: [2x7 double]
               Bandwidth: 3000000
               Beamwidth: [2x1 double]
                     PRF: 1500
              Pulsewidth: 5.0000e-06
     DetectionProbability: 0.9000
    FalseAlarmProbability: 1.0000e-06
             ReferenceRCS: 1
                 Priority: 1
```

Assuming the volume search produced a detection at -20 degrees in azimuth and -12 degrees in elevation, show the beam positions of the cued search task using the `helperPlotSearchGrid` function.

```
az = -20;
el = -12;
plot(az, el, 'o', 'Color', 'k', 'MarkerSize', 5, 'DisplayName', 'Detection');
helperPlotSearchGrid(cuedSearchTask.beamPositions([-20; -12]), cuedSearchPrms.Beamwidth,...
    'SearchTaskName', 'Cued search', 'Color', [0.8500 0.3250 0.0980]);
```
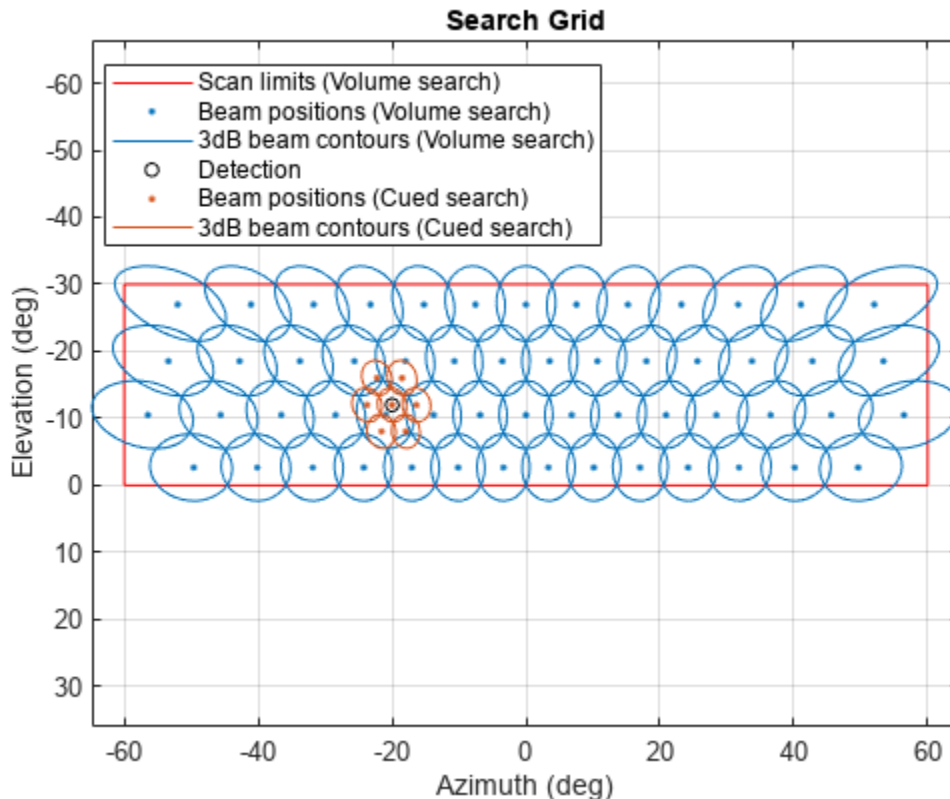

Tracking

The tracking task confirms detections produced by the cued search, and initiates and updates target tracks. The tracking task can request two types of looks: track confirmation and track update. A confirmation look is produced when the tracking task processes a detection that has not been associated with any existing tracks. In this case the tracking task initiates an unconfirmed track and requests a confirmation look. The confirmation looks revisit the track at the highest possible rate and have the highest priority among all look requests. Set the track confirmation rate to 20 Hz.

```
trackingPrms.ConfirmRate = 20;
```

Once the track has been confirmed by passing an M-of-N threshold, the tracking task performs adaptive tracking [1, 2] (for more information on adaptive tracking please see the "Adaptive Tracking of Maneuvering Targets with Managed Radar" (Sensor Fusion and Tracking Toolbox) example). Adaptive tracking updates non-maneuvering targets less frequently than maneuvering, thus optimizing the number of requested looks to efficiently utilize the available radar system resources. When the tracking task requests a track update look it finds the lowest track revisit rate at which the

predicted azimuthal track accuracy does not exceed a specified threshold. This threshold is defined in terms of a fraction of the azimuth beamwidth known as the *track sharpness*. A maneuvering target will exceed this threshold sooner than a non-maneuvering thus requiring more frequent updates. Set the track sharpness to 0.05.

```
trackingPrms.TrackSharpness = 0.05;
```

Since the azimuth beamwidth of the tracking task is 2 degrees, a track will be revisited at the lowest rate at which the predicted azimuth standard deviation does not exceed 2*0.05 = 0.1 degrees. Set the highest track update rate to 5 Hz and the lowest track update rate to 0.5 Hz.

```
trackingPrms.HighRevisitRate = 5;
trackingPrms.LowRevisitRate = 0.5;
```

Create the tracking task using `helperTrackingTask` class.

```
trackingTask = helperTrackingTask(trackingPrms, mparPrms, volumeSearchPrms.ScanLimits)

trackingTask =
  helperTrackingTask with properties:

                  Tracker: [1x1 trackerGNN]
           TrackSharpness: 0.0500
              ConfirmRate: 20
           HighRevisitRate: 5
            LowRevisitRate: 0.5000
                Bandwidth: 10000000
                Beamwidth: [2x1 double]
                      PRF: 1500
               Pulsewidth: 2.5000e-06
       DetectionProbability: 0.9000
      FalseAlarmProbability: 1.0000e-06
             ReferenceRCS: 1
                 Priority: [1 1]
```

Look Scheduler

The look objects generated by the radar tasks have a `StartTime` property and a `Priority` property. `StartTime` indicates when, from a task's point of view, it would be best to execute that radar look. Priority indicates the priority level of the look, such that the looks with the same `StartTime` but higher priority are scheduled first. At the start of each update interval the radar tasks produce look requests that have `StartTime` within this update interval. But not all of these looks can be executed within this update interval because of the resource constraints. A simple frame-based scheduler selects a subset of looks that can be executed within this update interval without violating any resource constraints and arranges them in time based on their priorities [2].

Set the system update rate to 20 Hz.

```
updateRate = 20;
```

This update rate defines a 1/20=50 ms update interval and a 50 ms window used for look scheduling.

This example assumes that the track confirmation looks have the highest priority level of 4, the track update looks have the priority level of 3, the priority level of the cued search looks is 2, and the volume search looks have the lowest priority of 1.

```
volumeSearchTask.Priority = 1;
cuedSearchTask.Priority = 2;
trackingTask.Priority = [3, 4];        % [Track update, Track confirm];
```

Since the volume and the cued search can be executed at the same time, the scheduler schedules the volume and the cued search looks in parallel. The looks that were scheduled for the current update interval but do not fit into the 50 ms window are considered for scheduling at the next update interval.

Scenario 1: Maneuvering Targets

Use the radarScenario object to simulate radar targets and trajectories. Set the update rate of the scenario to updateRate.

```
% Simulate the first 20 seconds of the scenario. To simulate the entire
% scenario set stopTime1 to 200
stopTime1 = 20;
scenario1 = radarScenario('UpdateRate', updateRate, 'StopTime', stopTime1);
```

Load a set of waypoint trajectories.

```
trajectories = struct2cell(load('modifiedBenchmarkTrajectories.mat', '-mat'));
numTrajectories = numel(trajectories);
```

These are the benchmark trajectories used in the "Benchmark Trajectories for Multi-Object Tracking" (Sensor Fusion and Tracking Toolbox) example modified such that they are more distributed in altitude. Add targets that follow these trajectories to the scenario assuming all targets are Swerling 1 with 3 m² RCS.

```
targetRCSSignature = rcsSignature('Pattern', 3, 'FluctuationModel', 'Swerling1');
for trajIdx = 1:numTrajectories
    platform(scenario1, 'Trajectory', trajectories{trajIdx}, 'Signature', targetRCSSignature);
end
```

Use radarDataGenerator class to create a detection-level radar sensor model.

```
targetProfiles.Signatures = {targetRCSSignature};
radar = radarDataGenerator(1, ...
    'ScanMode', 'Custom', ...
    'UpdateRate', updateRate, ...
    'HasElevation', true, ...
    'HasFalseAlarms', true, ...
    'HasScanLoss', true, ...
    'DetectionCoordinates', 'Sensor spherical', ...
    'Profiles', targetProfiles);
```

The radarDataGenerator is set up to operate in the Custom scan mode. Notice that none of the properties that specify the detection performance or the measurement accuracy are set at this point. The Custom mode allows for changing these properties dynamically during a simulation based on the properties of the executed radar looks.

```
% Object to store simulation data for later visualization of the results
scenario1Data = helperScenarioData(updateRate, stopTime1);

% Radar looks that were deferred by the scheduler
deferredLooks = helperRadarLook.empty;

% For repeatable results, set the random seed and revert it when done
```

```matlab
s = rng(2022, 'twister');
oc = onCleanup(@() rng(s));

% Reset the objects to be able to rerun this section
reset(volumeSearchTask);
reset(cuedSearchTask);
reset(trackingTask);
reset(radar);
restart(scenario1);

while advance(scenario1)
    time = scenario1.SimulationTime;

    % Get look requests that are due during this update interval
    volumeSearchLooks = lookRequests(volumeSearchTask, time, updateRate);
    cuedSearchLooks   = lookRequests(cuedSearchTask, time, updateRate);
    trackingLooks     = lookRequests(trackingTask, time, updateRate);

    % Combine all requested looks adding any looks remaining from the previous
    % update interval
    allLookRequests = [deferredLooks volumeSearchLooks cuedSearchLooks trackingLooks];

    % Generate a timeline of looks that can be executed within one
    % update interval based on the task priorities
    [scheduledLooks, deferredLooks] = helperScheduler(allLookRequests, time, updateRate);

    % Get target platforms from the scenario
    targets = platformPoses(scenario1);

    allDetections = [];
    trackPositions = [];
    trackIDs = [];

    % Execute each scheduled look one at a time
    for lookIdx = 1:numel(scheduledLooks)
        look = scheduledLooks(lookIdx);

        % Set radarDataGenerator properties
        radar.LookAngle            = look.LookAngle;
        radar.DetectionProbability = look.DetectionProbability;
        radar.FalseAlarmRate       = look.FalseAlarmProbability;
        radar.ReferenceRCS         = look.ReferenceRCS;

        % Set the reference range - the range at which a Swerling 0 target
        % with the Reference RCS is detected with the set
        % DetectionProbability and FalseAlarmRate - using the
        % helperReferenceRange function
        radar.ReferenceRange       = helperReferenceRange(look, mparPrms);

        radar.AzimuthResolution    = look.Beamwidth(1);
        radar.ElevationResolution  = look.Beamwidth(2);
        radar.RangeResolution      = bw2rangeres(look.Bandwidth);

        % Linearly interpolate the target positions from the last scenario
        % update time to the StartTime of the look
        dt = look.StartTime - time;
        for targetIdx = 1:numel(targets)
            targets(targetIdx).Position = targets(targetIdx).Position + dt * targets(targetIdx).\
```

```
        end

        % Generate detections
        detections = radar(targets, time);

        % Set the detection time property to the look StartTime
        for detIdx = 1:numel(detections)
            detections{detIdx}.Time = look.StartTime;
        end

        % If this is a volume search look, the generated detections are
        % processed by the cued search
        if ~isempty(detections) && look.Type == helperRadarLookType.VolumeSearch
            cuedSearchTask.processDetections(detections);
        end

        % If this is a track confirmation or a track update look, or if
        % this a cued search look that resulted in detections, process the
        % detections with the trackingTask
        if (~isempty(detections) && look.Type == helperRadarLookType.CuedSearch) ...
                || look.Type == helperRadarLookType.ConfirmTrack ...
                || look.Type == helperRadarLookType.UpdateTrack
            [trackPositions, trackIDs] = trackingTask.processDetections(look.StartTime, detection
        end

        allDetections = [allDetections; detections];
    end

    % Cued search task maintenance
    cuedSearchTask.maintenance();

    % Tracking task maintenance
    trackingTask.maintenance(time);

    % Log simulation data
    truePositions = reshape([targets.Position], 3, [])';
    measuredPositions = helperGetCartesianMeasurement(allDetections);
    scenario1Data.log(truePositions, scheduledLooks, measuredPositions, trackPositions, trackIDs)
end
```

After running the simulation, the true target positions, the scheduled looks, the detections, and the tracks are stored in scenario1Data. You can use the recorded data to visualize the positions of the radar beams together with the true target locations, detections, and tracks.

```
% Produce scenario visualizations

generateVisualizations = ◩;
if generateVisualizations
    % Use helperScenarioPlotter class to visualize:
    %   - targets, trajectories, detections, and tracks
    %   - resulting beams placed by the radar
    % for the last 2 seconds of the scenario
    scenarioPlotter1 = helperScenarioPlotter(scenario1, volumeSearchTask.BeamPositions,...
        volumeSearchPrms.Beamwidth, volumeSearchPrms.ScanLimits);
    scenarioPlotter1.plot(scenario1Data, stopTime1-2, stopTime1);
end
```
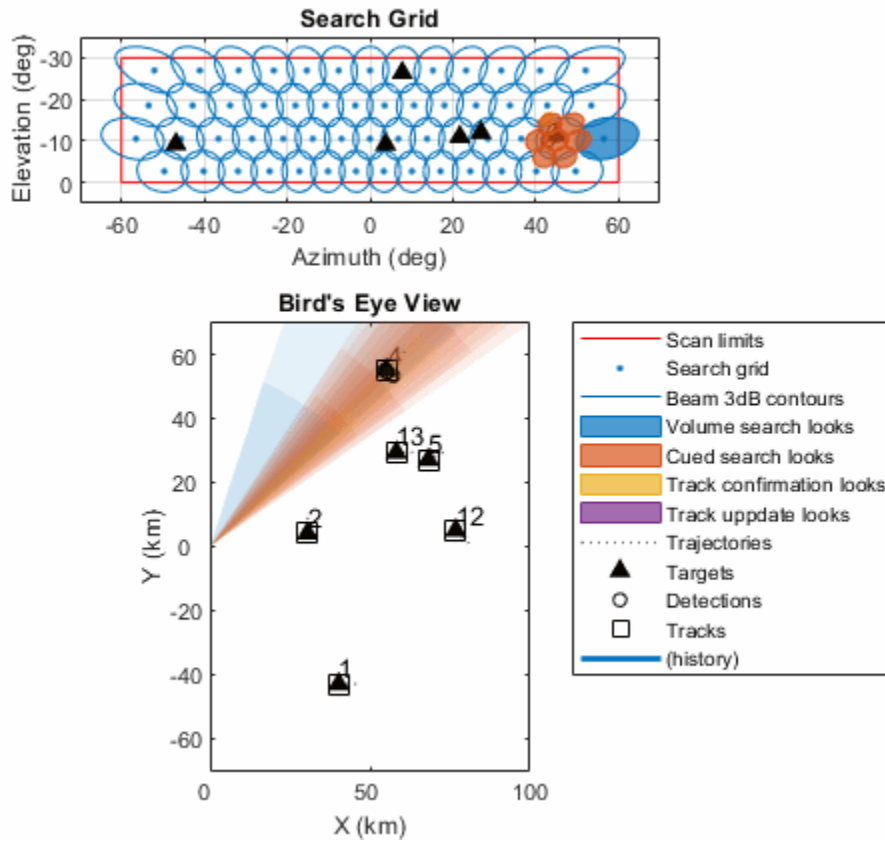
Use the `helperResourcesPlotter` function to visualize the use of the MPAR resources over the time of the scenario.

```
helperResourcesPlotter(scenario1Data);
```
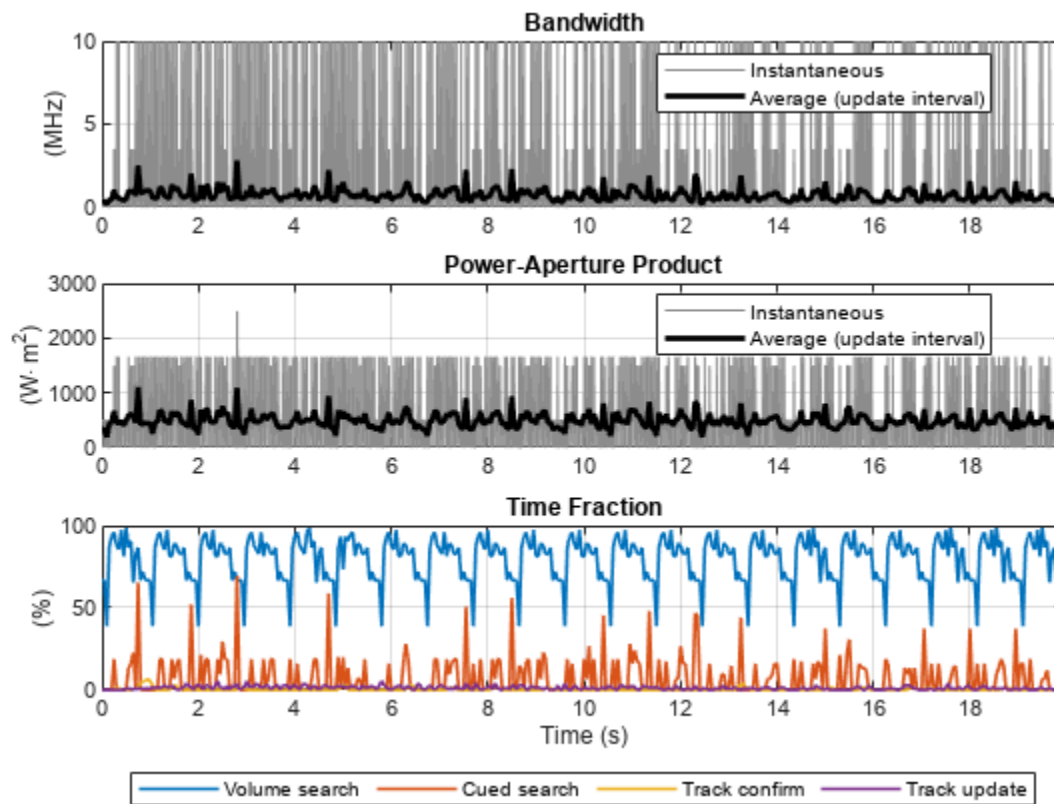
Although the instantaneous bandwidth used by the MPAR varies significantly over time, the average bandwidth in each update interval remains close to 0.5 Hz, which is the bandwidth required by the volume search task. Similarly, the power-aperture product averaged over the update intervals stays close to 500 W·m². This indicates that, despite spending some resources on the cued search and target tracking, the MPAR is mostly performing the volume search. This is also confirmed by the radar time fraction plotted against the simulation time. It shows that the scheduler almost always fills 90% or more of its time window with the volume search looks. The fluctuations in the volume search time fraction are due to 1) triggering a cued search task every time the volume search produces a detection even if it is a detection from a target that is already being tracked, and 2) using a simplistic scheduling logic based on the task priorities that does not optimize the use of the MPAR resources.

Overall, this result shows that tracking six targets in addition to performing volume and cued search does not significantly strain the radar system resources. However, if the number of targets in the search volume would grow, more and more resources would be diverted from the search task to the tracking task.

Scenario 2: High Probability of False Alarm

Higher values of the probability of false alarm can result in additional stress on the resource management system since false detections require scheduling of additional cued search and track confirmation beams. Create a new `radarScenario` object without any targets, set the probability of false alarm to 1.3e-5 for all tasks, and run the simulation.

```
stopTime2 = 8;
scenario2 = radarScenario('UpdateRate', updateRate, 'StopTime', stopTime2);
volumeSearchPrms.FalseAlarmProbability = 1.3e-5;
```

```
cuedSearchPrms.FalseAlarmProbability = 1.3e-5;
trackingPrms.FalseAlarmProbability = 1.3e-5;

updateTaskParameters(volumeSearchTask, volumeSearchPrms);
updateTaskParameters(cuedSearchTask, cuedSearchPrms);
updateTaskParameters(trackingTask, trackingPrms);

% Run the simulation. For convenience, the scenario simulation loop is moved
% to the helperSimulateMPARScenario function
scenario2Data = helperSimulateMPARScenario(scenario2, radar, volumeSearchTask, cuedSearchTask, tr
```

Visualize the scheduled beams, the detections, and the tracks for the update intervals between the 5th and the 7th second of the scenario.

```
% Produce scenario visualizations

generateVisualizations = ⬛;
if generateVisualizations
    % Use helperScenarioPlotter class to visualize:
    %    - targets, trajectories, detections, and tracks
    %    - resulting beams placed by the radar
    scenarioPlotter2 = helperScenarioPlotter(scenario2, volumeSearchTask.BeamPositions,...
        volumeSearchPrms.Beamwidth, volumeSearchPrms.ScanLimits);
    scenarioPlotter2.plot(scenario2Data, 5, 7);
end
```
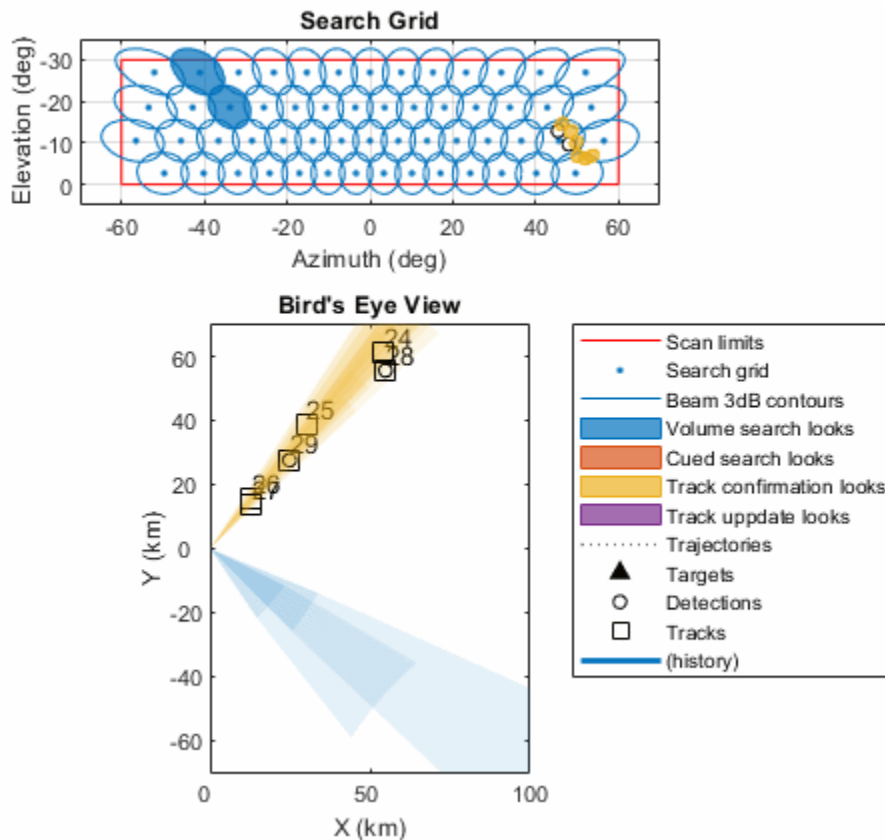

Plot the bandwidth, the power-aperture product, and the radar time fraction over time.

```
helperResourcesPlotter(scenario2Data);
```
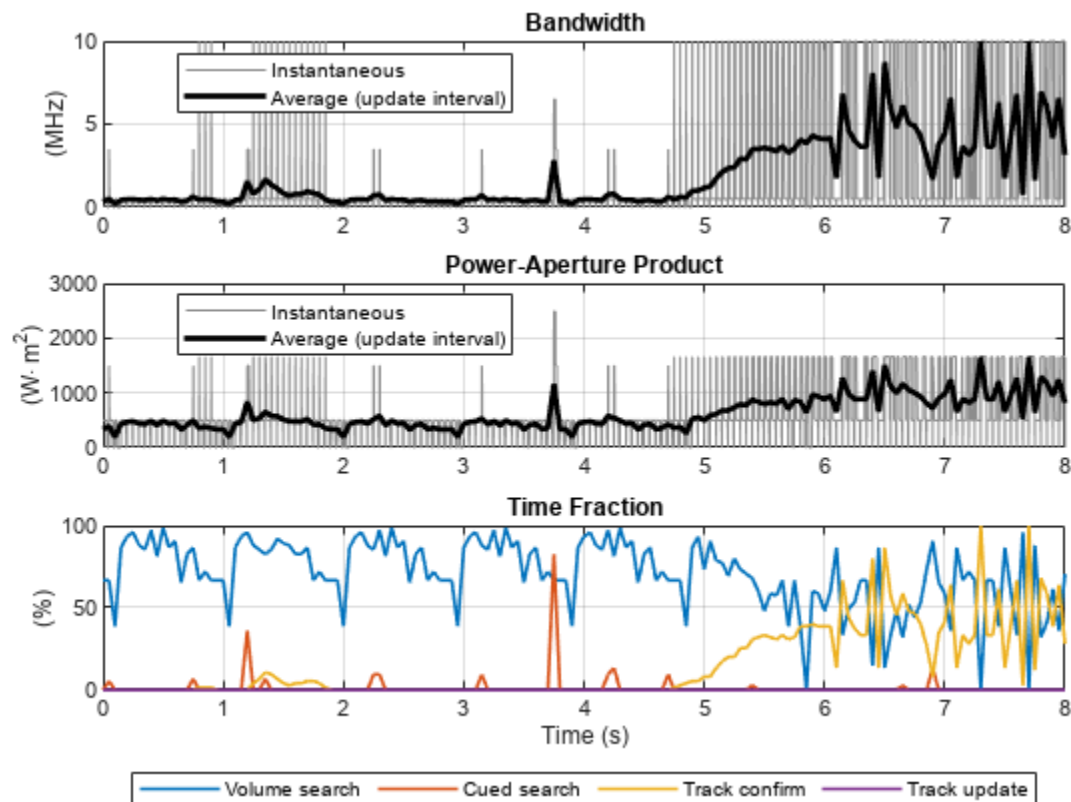

Starting at around 5 seconds into the simulation the bandwidth and the power-aperture used by the MPAR start to grow. The time fraction plot shows that the track confirmation looks occupy about 50% of the radar time at around 6.5 seconds. For some update intervals after the 7th second the confirmation looks fill 100% of the scheduler time window. This is the consequence of the high probability of false alarm, which leads to a high volume of false detections and a high number of confirmation looks. The volume search task starts to slow down starting at the 5 second mark because the track task and the confirmation looks have higher priority than the volume search looks. Thus, the system resources are being diverted from the volume search to the tracking task. This result shows how sensitive the resource management performance could be to the probability of false alarm.

Conclusions

This example shows how to use `radarDataGenerator` to simulate an MPAR that can perform volume search, cued search, and target tracking. The volume and the cued search tasks can be executed at the same time by sharing the frequency bandwidth and the phased array aperture. On the other hand, the target tracking task uses the entire available bandwidth and aperture and thus cannot be multiplexed with other tasks. The example simulates two scenarios. It uses a frame-based radar look scheduler to arrange looks generated by the radar tasks into a timeline that must be executed within one scenario update interval. In the first scenario the MPAR searches and tracks six maneuvering targets. The computed radar time fraction indicates that tracking a small number of targets does not take away a lot of resource from the search functions. In the second scenario no targets are present in the search volume, but the probability of false alarm is set to a high value resulting in a spontaneous increase in the track confirmation looks that end up consuming more than

50% of the radar time. This scenario shows how such MPAR simulations can be used to study sensitivity of a resource management system to various parameters.

References

1 Moo, Peter, and Zhen Ding. *Adaptive radar resource management*. Academic Press, 2015.

2 Charlish, Alexander, Fotios Katsilieris, R. Klemm, U. Nickel, and C. Gierull. "Array radar resource management." *Novel radar techniques and applications volume 1: real aperture array radar, imaging radar, and passive and multistatic radar* (2017): 135-171.

3 Richards, M.A., Scheer, J. A. & Holm, W. A. Principle of Modern Radar: Basic Principles (SciTech Publishing, 2010).

Supporting Functions

```matlab
type('helperRadarLook.m')
```

```matlab
classdef helperRadarLook

    properties
        Type
        StartTime
        LookAngle
        NumPulses

        DetectableTrackID
        Priority
    end

    properties (Hidden)
        PowerApertureProduct
        StartFrequency
    end

    properties (Dependent)
        DwellTime
        Bandwidth
        Beamwidth
        Pulsewidth
        PRF
        DetectionProbability
        FalseAlarmProbability
        ReferenceRCS
    end

    properties (Access = private)
        TaskPrms
    end

    methods
        function obj = helperRadarLook(type, startTime, lookAngle, numPulses, taskPrms)
            obj.Type = helperRadarLookType(type);
            obj.StartTime = startTime;
            obj.LookAngle = lookAngle;
            obj.NumPulses = numPulses;
            obj.TaskPrms = taskPrms;

            obj.DetectableTrackID = zeros(0, 1, 'uint32');
```

```
            obj.Priority = uint32(obj.Type);

            obj.PowerApertureProduct = 0;
            obj.StartFrequency = 0;
        end

        function value = get.DwellTime(obj)
            value = obj.NumPulses / obj.PRF;
        end

        function value = get.Bandwidth(obj)
            value = obj.TaskPrms.Bandwidth;
        end

        function value = get.Beamwidth(obj)
            value = obj.TaskPrms.Beamwidth;
        end

        function value = get.Pulsewidth(obj)
            value = obj.TaskPrms.Pulsewidth;
        end

        function value = get.PRF(obj)
            value = obj.TaskPrms.PRF;
        end

        function value = get.DetectionProbability(obj)
            value = obj.TaskPrms.DetectionProbability;
        end

        function value = get.FalseAlarmProbability(obj)
            value = obj.TaskPrms.FalseAlarmProbability;
        end

        function value = get.ReferenceRCS(obj)
            value = obj.TaskPrms.ReferenceRCS;
        end
    end
end

type('helperRadarLookType.m')

classdef helperRadarLookType < uint32
    enumeration
        VolumeSearch (1)
        CuedSearch   (2)
        ConfirmTrack (3)
        UpdateTrack  (4)
    end
end

type('helperPlotSearchGrid.m')

function helperPlotSearchGrid(varargin)
    [ax, beamPositions, beamwidth, scanLimits, taskName, color] = parseInputs(varargin{:});

    hold(ax, "on");
    scanLimitsDisplayName = "Scan limits";
    beamPositionsDisplayName = "Beam positions";
```

```matlab
        beamContoursDisplayName = "3dB beam contours";


        if ~isempty(taskName)
            scanLimitsDisplayName = sprintf("%s (%s)", scanLimitsDisplayName, taskName);
            beamPositionsDisplayName = sprintf("%s (%s)", beamPositionsDisplayName, taskName);
            beamContoursDisplayName = sprintf("%s (%s)", beamContoursDisplayName, taskName);
        else
        end

        if ~isempty(scanLimits)
            plot(ax, [scanLimits(1, 1) scanLimits(1, 1)], scanLimits(2, :), 'r', 'DisplayName', scanL
            plot(ax, scanLimits(1, :), [scanLimits(2, 2) scanLimits(2, 2)], 'r', 'HandleVisibility',
            plot(ax, [scanLimits(1, 2) scanLimits(1, 2)], scanLimits(2, :), 'r', 'HandleVisibility',
            plot(ax, scanLimits(1, :), [scanLimits(2, 1) scanLimits(2, 1)], 'r', 'HandleVisibility',
        end

        plot(ax, beamPositions(1, :), beamPositions(2, :), '.', 'Color', color, 'DisplayName', beamP

        p = helperBeamwidthContour(beamPositions(:, 1), beamwidth);
        plot(ax, p(1,:), p(2,:), 'Color', color, 'DisplayName', beamContoursDisplayName);

        for i = 2:size(beamPositions, 2)
            p = helperBeamwidthContour(beamPositions(:, i), beamwidth);
            plot(ax, p(1,:), p(2,:), 'Color', color, 'HandleVisibility', 'off');
        end

        grid(ax, "on");
        box(ax, "on");

        xlabel(ax, 'Azimuth (deg)');
        ylabel(ax, 'Elevation (deg)');
        axis(ax, "equal");
        ax.YDir = 'reverse';
    end

    function [ax, beamPositions, beamwidth, scanLimits, taskName, color] = parseInputs(varargin)
        args = varargin;

        k = 1;
        if isgraphics(args{1}, 'axes')
            ax = args{1};
            k = 2;
        else
            ax = gca();
        end

        defaultScanLimits = [];
        defaultTaskName = [];
        defaultColor = [0.45 0.45 0.45];

        p = inputParser;
        p.addRequired('beamPositions');
        p.addRequired('beamwidth');
        p.addParameter('ScanLimits', defaultScanLimits);
        p.addParameter('SearchTaskName', defaultTaskName)
        p.addParameter('Color', defaultColor)
```

```
        p.parse(args{k:end});
        beamPositions = p.Results.beamPositions;
        beamwidth = p.Results.beamwidth;
        scanLimits = p.Results.ScanLimits;
        taskName = p.Results.SearchTaskName;
        color = p.Results.Color;
end
```

type('helperPowerApertureProduct')

```
function pap = helperPowerApertureProduct(peakPower, pulsewidth, PRF, A)
    pap = peakPower .* pulsewidth .* PRF * prod(A);
end
```

type('helperNumPulses')

```
function n = helperNumPulses(lookAngle, range, taskPrms, mparPrms)
    % Detectability factor
    Dx = detectability(taskPrms.DetectionProbability, taskPrms.FalseAlarmProbability);

    % Reference range
    referenceRange = radareqrng(mparPrms.Wavelength, Dx, mparPrms.PeakPower, 1/mparPrms.Bandwidth
        'Gain', beamwidth2gain(mparPrms.Beamwidth), 'RCS', taskPrms.ReferenceRCS, 'Ts', systemp(

    % Array scan loss
    Ls = abs(cosd(lookAngle(1, :))).^(-3) .* abs(cosd(lookAngle(2, :))).^(-6);

    % Number of pulses
    deltaAntennaGain = beamwidth2gain(mparPrms.Beamwidth) - beamwidth2gain(taskPrms.Beamwidth);
    n = (range/referenceRange)^4 * db2pow(2*deltaAntennaGain) * Ls/taskPrms.Pulsewidth/mparPrms.

    n = ceil(n);
end
```

type('helperReferenceRange')

```
function R = helperReferenceRange(radarLook, mparPrms)
    % Detectability factor
    Dx = detectability(radarLook.DetectionProbability, radarLook.FalseAlarmProbability);

    % Reference range
    R = radareqrng(mparPrms.Wavelength, Dx, mparPrms.PeakPower, radarLook.NumPulses * radarLook.
        'Gain', beamwidth2gain(radarLook.Beamwidth), 'RCS', radarLook.ReferenceRCS, 'Ts', systemp
end
```

Simulate FMCW Interference Between Automotive Radars

This example shows how to simulate frequency-modulated continuous-wave (FMCW) interference between two multiple input multiple output (MIMO) automotive radars in a highway driving scenario. Missed and ghost targets caused by interferences can be simulated and visualized. Multipath propagation effects between vehicles are taken into account.

Introduction

With the increasing number of vehicles being equipped with multiple radars in automotive scenarios, interference detection and mitigation have become essential for automotive radars. Interference can affect the functionality of radar systems in many different ways. Interference can increase the noise level of the range-angle-Doppler map and decrease the probability of detection. The signal to interference and noise ratio (SINR) depends on the distance and angles separating two radars, as well as radar parameters such as antenna gain, antenna transmit power, array size, bandwidth, center frequency, and chirp duration. Low SINR results in missed detections for low radar cross section (RCS) targets, creating a blind spot in some regions. Another challenge with radar interference is ghost targets, in which the radar detects an object that doesn't exist. The victim radar can mitigate interference by fast-time/slow-time domain interference removal/signal reconstruction, spatial-domain beamforming/detector design, etc.

Waveform and Transceiver Definitions

Define Victim Baseband FMCW Radar Waveform

The victim radar is designed to use an FMCW waveform which is a common waveform in automotive applications because it enables range and Doppler estimation through computationally efficient fast Fourier transform (FFT) operations. For illustration purposes, in this example, configure the radar to a maximum range of 150 m. The victim radar operates at a 77 GHz frequency. The radar is required to resolve objects that are at least 1 meter apart. Because this is a forward-facing radar application, the maximum relative speed of vehicles on highway is specified as 230 km/h.

```
rng(2022);              % Set random number generator for repeatable results
fc = 77e9;              % Center frequency (Hz)
c = physconst('LightSpeed');         % Speed of light (m/s)
lambda = freq2wavelen(fc,c);         % Wavelength (m)
rangeMax = 150;         % Maximum range (m)
rangeRes = 1;           % Desired range resolution (m)
vMax = 230*1000/3600;   % Maximum relative velocity of cars (m/s)
fmcwwav1 = helperFMCWWaveform(fc,rangeMax,rangeRes,vMax);
sig = fmcwwav1();
Ns = numel(sig);
```

Define Interfering Baseband FMCW Radar Waveform

Consider an interfering radar using FMCW modulation. The frequency spectrum characteristics of the interference in the victim radar depend on the chirp's relative timing and slopes.
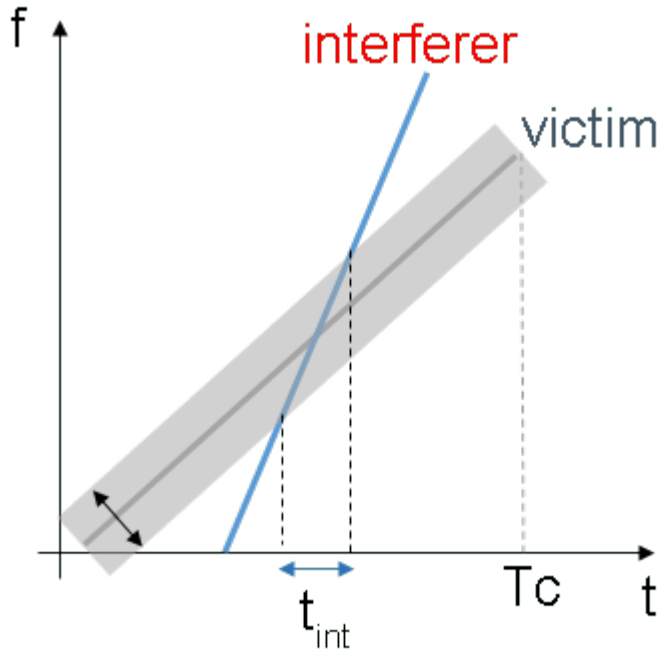
```
fcRdr2 = 77e9;              % Center frequency (Hz)
lambdaRdr2 = freq2wavelen(fcRdr2,c);              % Wavelength (m)
rangeMaxRdr2 = 100;     % Maximum range (m)
rangeResRdr2 = 0.8;      % Desired range resolution (m)
vMaxRdr2 = vMax;        % Maximum Velocity of cars (m/s)
fmcwwav2 = helperFMCWWaveform(fcRdr2,rangeMaxRdr2,rangeResRdr2,vMaxRdr2);
sigRdr2 = fmcwwav2();
```

Plot the victim radar baseband waveform in the first 3 pulses.

```
figure
pspectrum(repmat(sig,3,1),fmcwwav1.SampleRate,'spectrogram', ...
    'Reassign',true,'FrequencyResolution',10e6)
axis([0 15 -80 80]); title('Victim Radar'); colorbar off
```

Plot the interfering radar baseband waveform with steeper chirp slopes in the first 3 pulses.
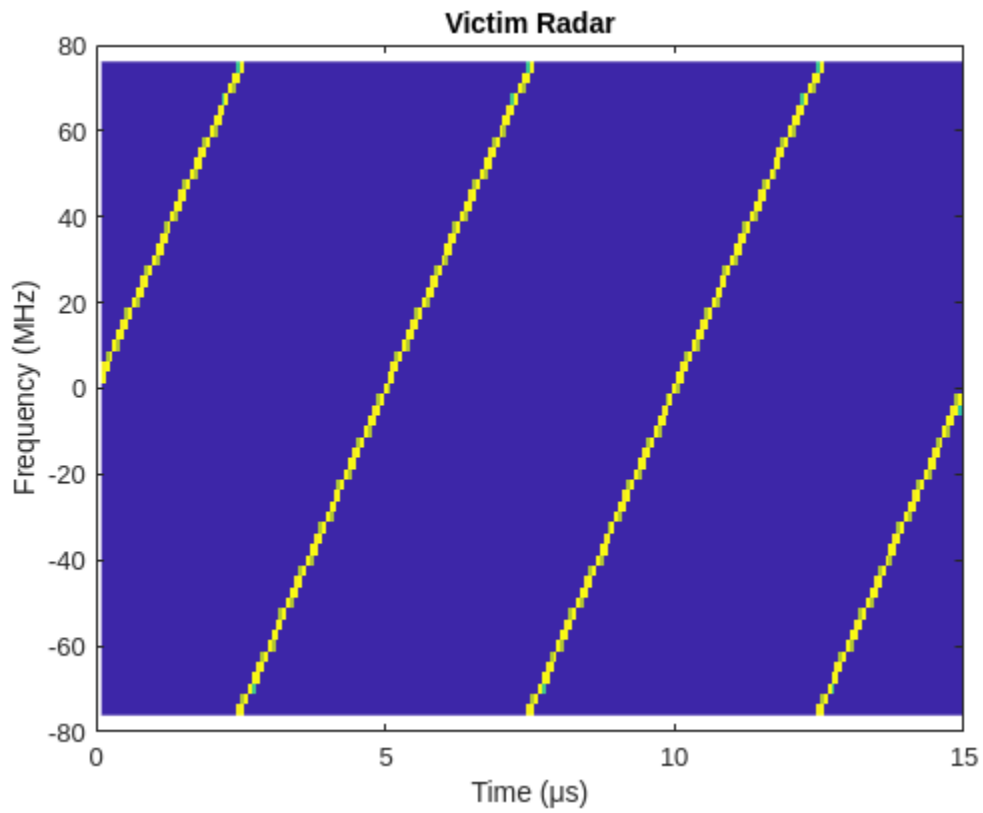
```
figure
pspectrum(repmat(sigRdr2,3,1),fmcwwav1.SampleRate,'spectrogram',...
    'Reassign',true,'FrequencyResolution',10e6,'MinThreshold',-13)
axis([0 15 -80 80]); title('Interfering Radar'); colorbar off;
```
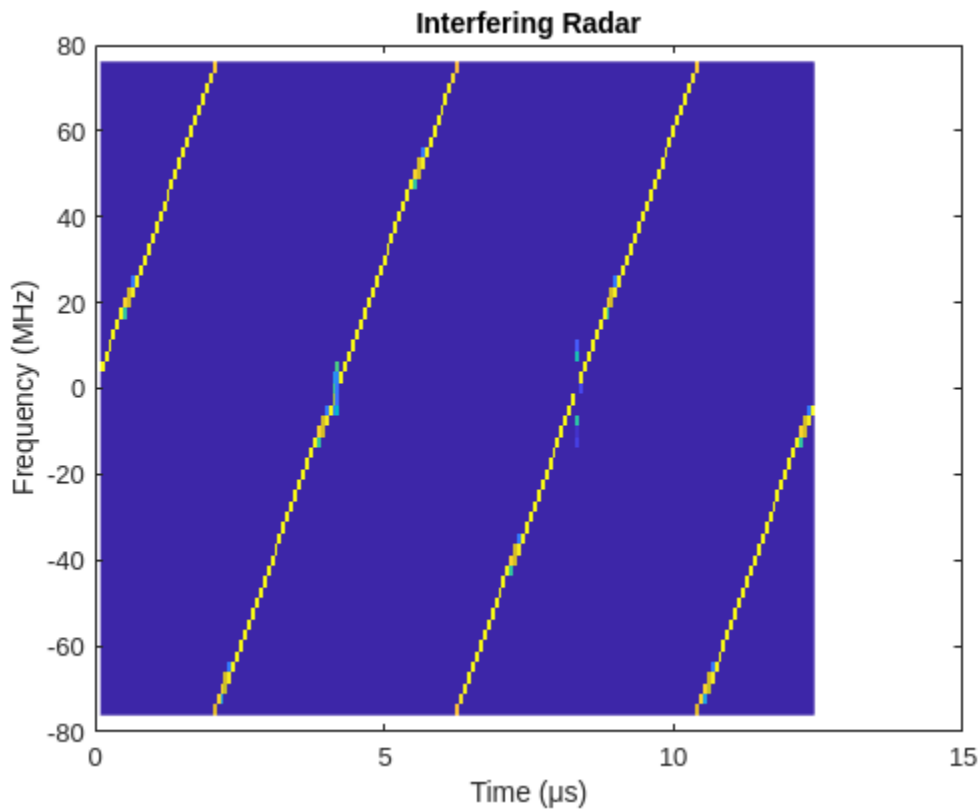
Model Victim and Interfering MIMO Radar Transceivers

Consider both radars use a uniform linear array (ULA) to transmit and use a ULA to receive the radar waveforms. Using a receive array enables the radar to estimate the azimuthal direction of the reflected energy received from potential targets. Using a transmit array enables the radar to form a large virtual array at the receiver to improve the azimuthal angle resolution. To receive target and noise signals at the victim radar, model the victim radar's monostatic transceiver using `radarTransceiver` and specifying the transmit and receive ULAs of the victim radar in its properties.

```
% Set radar basic transceiver parameters
antAperture = 6.06e-4;                          % Antenna aperture (m^2)
antGain = aperture2gain(antAperture,lambda);    % Antenna gain (dB)
txPkPower = db2pow(13)*1e-3;                     % Tx peak power (W)
rxNF = 4.5;                                      % Receiver noise figure (dB)

% Construct the receive array for victim radar
Nvr = 16;     % Number of receive elements for victim radar
vrxEleSpacing = lambda/2;         % Receive element spacing for victim radar
antElmnt = phased.IsotropicAntennaElement('BackBaffled',false);
vrxArray = phased.ULA('Element',antElmnt,'NumElements',Nvr,...
    'ElementSpacing',vrxEleSpacing);

% Construct the transmit array for victim radar
Nvt = 2;     % Number of transmit elements for victim radar
vtxEleSpacing = Nvr*vrxEleSpacing;     % Transmit element spacing for victim radar
vtxArray = phased.ULA('Element',antElmnt,'NumElements',Nvt,...
```

```
    'ElementSpacing',vtxEleSpacing);

% Model victim radar monostatic transceiver for receiving target and noise
% signals
vradar = radarTransceiver("Waveform",fmcwwav1,'ElectronicScanMode','Custom');
vradar.Transmitter = phased.Transmitter('PeakPower',txPkPower,'Gain',antGain);
vradar.TransmitAntenna = phased.Radiator('Sensor',vtxArray,'OperatingFrequency',fc,'WeightsInput
vradar.ReceiveAntenna = phased.Collector('Sensor',vrxArray,'OperatingFrequency',fc);
vradar.Receiver = phased.ReceiverPreamp('Gain',antGain,'NoiseFigure',rxNF,'SampleRate',fmcwwav1.S
```

Different from the target signals, the interference is transmitted from the transmit ULA of the interfering radar and received by the receive ULA of the victim radar. Model the interference transceiver using the `helperInterferenceTransceiver` function, which includes the transmit ULA of the interfering radar and the receive ULA of the victim radar in its inputs.

```
% Construct the transmit array for interfering radar
Nit = 3;      % Number of transmit elements for interfering radar
Nir = 4;      % Number of receive elements for interfering radar
irxEleSpacing = lambda/2;        % Receive element spacing for interfering radar
itxEleSpacing = Nir*irxEleSpacing;      % Transmit element spacing for interfering radar
itxArray = phased.ULA('Element',antElmnt,'NumElements',Nit,...
    'ElementSpacing',itxEleSpacing);

% Model interference transceiver for receiving interference signal at the
% victim radar
iradar = helperInterferenceTransceiver(fmcwwav2,txPkPower,antGain,fc,itxArray,vrxArray);
```

Simulate Driving Scenario

Create a highway driving scenario with two vehicles traveling in the vicinity of the ego vehicle with a forward-facing victim radar. The vehicles are modeled as cuboids and have different velocities and positions defined in the driving scenario. The ego vehicle is moving with a velocity of 25 m/s, the target vehicle is moving at 34 m/s in the same direction, and the interfering vehicle with a forward-facing interfering radar is moving at -50 m/s in the opposite direction. For details on modeling a driving scenario, see the example "Create Actor and Vehicle Trajectories Programmatically" (Automated Driving Toolbox). The radar sensor is mounted on the front of the ego and interfering vehicles.

To create the driving scenario, use the `helperAutoDrivingScenario` function. To examine the contents of this function, use the `edit('helperAutoDrivingScenario')` command.

```
% Create driving scenario
[scenario, egoCar] = helperAutoDrivingScenario;
```
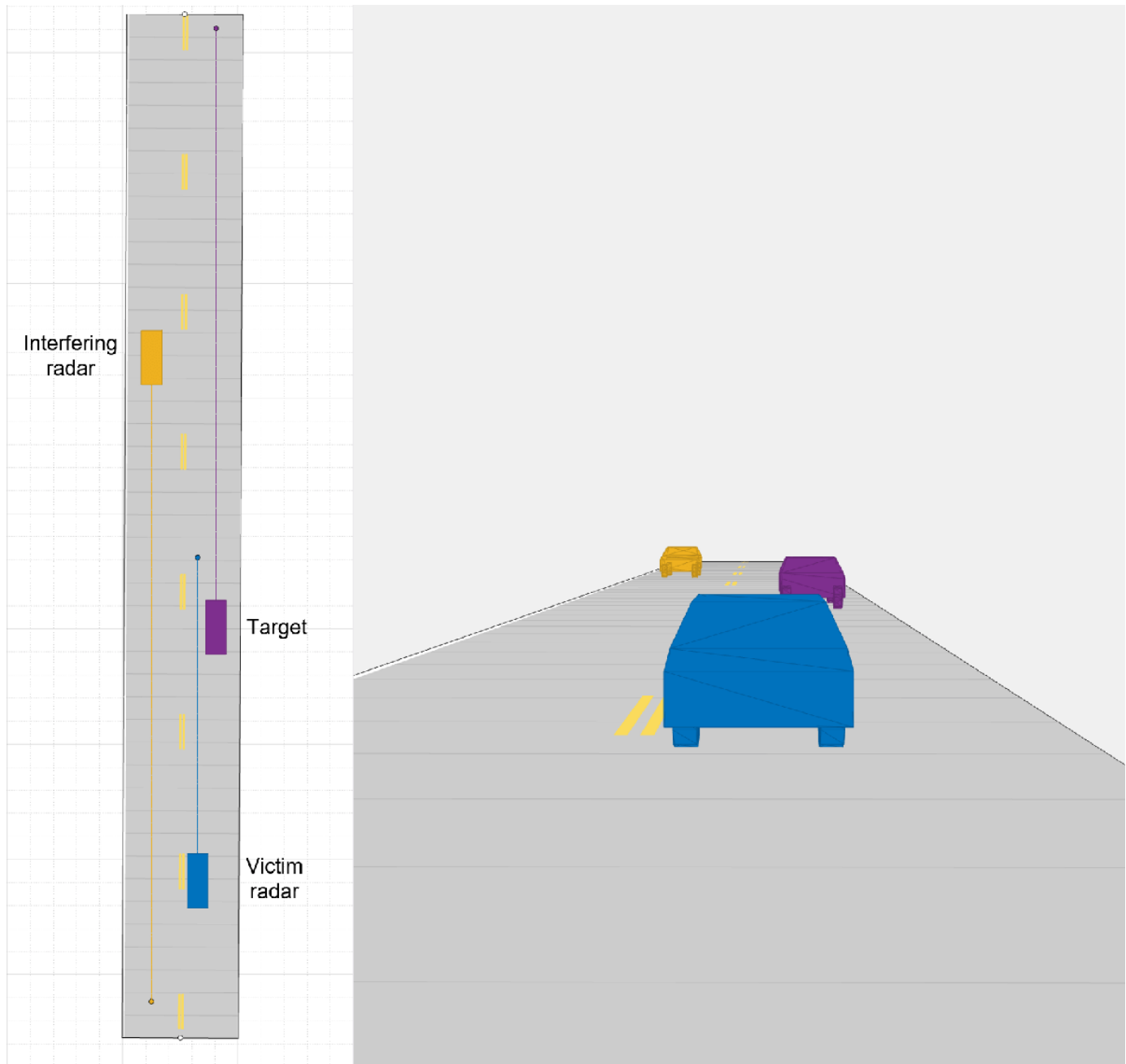
To visualize and modify the driving scenario, use the `drivingScenarioDesigner` with the current scenario as the input:

`drivingScenarioDesigner(scenario)`

Use the following code to see the initial ranges, angles and velocities of the target vehicle and the interfering vehicle relative to the ego vehicle. The leaving target vehicle is located at 22.0 m and -4.2 degrees at a velocity of 9.0 m/s relative to the victim radar, and the coming interfering vehicle on the opposite direction is located at 48.3 m and 4.8 degrees at a velocity of -74.7 m/s relative to the victim radar.

```
% Initialize target poses in ego vehicle's reference frame
tgtPoses = targetPoses(egoCar);

% Target vehicle's range and angle relative to the victim radar
[tarRange,tarAngle] =  rangeangle(tgtPoses(1).Position');

% Target vehicle's radial velocity relative to the victim radar (the negative sign is due to dif
tarVelocity = -radialspeed(tgtPoses(1).Position',tgtPoses(1).Velocity');

% Interfering vehicle's range and angle relative to the victim radar
[intRange,intAngle] =  rangeangle(tgtPoses(2).Position');

% Interfering vehicle's radial velocity relative to the victim radar
intVelocity = -radialspeed(tgtPoses(2).Position',tgtPoses(2).Velocity');
```

Transceiver Signal Processing

To form a large virtual array at the MIMO radar receiver, the transmit waveforms on different transmit antennas need to be orthogonal such that the receiver can separate signals from different transmit antennas. This example adopts time division multiplexing (TDM) to achieve the waveform orthogonality. In each pulse, a vector of TDM-MIMO code is weighted on the transmit array. For the principle on virtual array and TDM-MIMO radar, see the example "Increasing Angular Resolution with Virtual Arrays".

The radar collects a train of the pulses of the FMCW waveform at each of the receive ULA antenna elements. These collected pulses on different receive antennas form a data cube, which is defined in "Radar Data Cube". The data cube is coherently processed along the fast-time, slow-time, and array dimensions to estimate the range, Doppler and angle of potential targets.

The overall transceiver signal processing workflow for a MIMO radar is summarized below. The gray blocks are implemented using `radarTransceiver` with the channel paths and the TDM-MIMO code in its inputs. The path generation block generates the two-way target path, the one-way interference path, and the interference multipath. The green blocks show the data cube processing flow for MIMO radar.
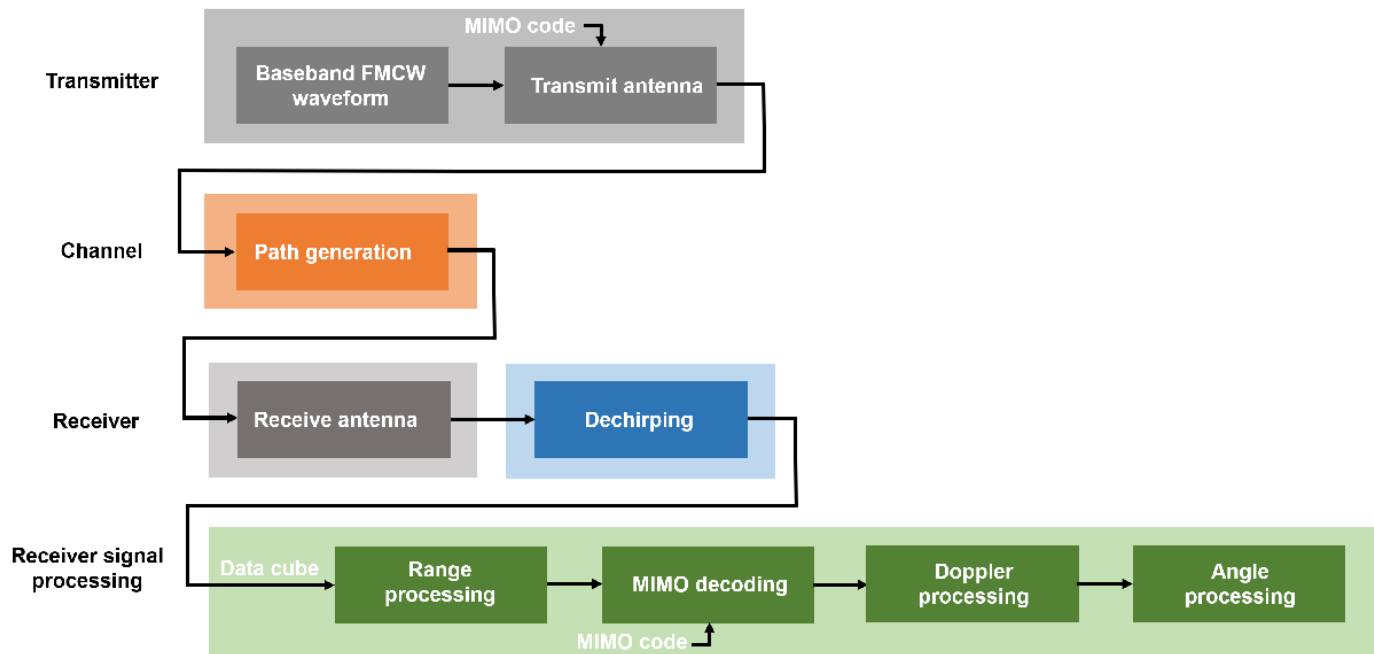
Obtain Interference-free and Interfered Data Cubes

The following code generates the received interference pulse train at the victim radar. For victim radar to dechirp interference in each of its pulse, the received interference pulse train is parsed according to the victim radar's frame structure, such that the number of fast-time samples in each parsed interference pulse is equal to the number of fast-time samples of the victim radar `Nft`.

```
% Define fast-time and slow-time samples
Nft = round(fmcwwav1.SweepTime*fmcwwav1.SampleRate);  % Number of fast-time samples of victim rad
Nsweep = 192;                                         % Number of slow-time samples of victim rad
Nift = round(fmcwwav2.SweepTime*fmcwwav2.SampleRate); % Number of fast-time samples of interferin
Nisweep = ceil(Nft*Nsweep/Nift);                      % Number of slow-time samples of interferin

% Generate TDM-MIMO code for interfering antenna elements
wi = helperTDMMIMOEncoder(Nit,Nisweep);

% Assemble interference pulse train at interfering radar's scenario time
rxIntTrain = zeros(Nift*Nisweep,Nvr);

% Initialize scenario time
time = scenario.SimulationTime;

% Initialize actor profiles
actProf = actorProfiles(scenario);

% Get interference pulse train at interfering radar's scenario time
for l = 1:Nisweep
    % Generate all paths to victim radar
    ipaths = helperGenerateIntPaths(tgtPoses,actProf,lambdaRdr2);

    % Get the received signal with interference only
    rxInt = iradar(ipaths,time,wi(:,l));
    rxIntTrain((l-1)*Nift+1:l*Nift,:) = rxInt;
```

```
    % Get the current scenario time
    time = time + fmcwwav2.SweepTime;

    % Move targets forward in time for next sweep
    tgtPoses(1).Position=[tgtPoses(1).Position]+[tgtPoses(1).Velocity]*fmcwwav2.SweepTime;
    tgtPoses(2).Position=[tgtPoses(2).Position]+[tgtPoses(2).Velocity]*fmcwwav2.SweepTime;
end

% Truncate interference pulse train to the length of target pulse train
rxIntTrain = rxIntTrain(1:Nft*Nsweep,:);

% Assemble interference pulse train at victim radar's scenario time
rxIntvTrain = permute(reshape(rxIntTrain,Nft,Nsweep,Nvr),[1,3,2]);
```

Dechirp the target signal at the victim radar to obtain the interference-free data cube, which is used to plot the target response as an ideal performance baseline. Also, dechirp the combined target and interference signal to obtain the interfered data cube, which is used to plot interference effects on target detection.

```
% Generate TDM-MIMO code for victim antenna elements
wv = helperTDMMIMOEncoder(Nvt,Nsweep);

% Assemble interference-free data cube
XcubeTgt = zeros(Nft,Nvr,Nsweep);

% Assemble interfered data cube
Xcube = zeros(Nft,Nvr,Nsweep);

% Initialize scenario time
time = scenario.SimulationTime;

% Initialize target poses in ego vehicle's reference frame
tgtPoses = targetPoses(egoCar);

% Initialize actor profiles
actProf = actorProfiles(scenario);

% Get data cube at victim radar's scenario time
for l = 1:Nsweep
    % Generate target paths to victim radar
    vpaths = helperGenerateTgtPaths(tgtPoses,actProf,lambda);

    % Get the received signal without interference
    rxVictim = vradar(vpaths,time,wv(:,l));

    % Dechirp the interference-free target signal
    rxVsig = dechirp(rxVictim,sig);

    % Save sweep to data cube
    XcubeTgt(:,:,l) = rxVsig;

    % Get the received interference signal
    rxInt = rxIntvTrain(:,:,l);

    % Dechirp the interfered signal
    rx = dechirp(rxInt+rxVictim,sig);
    Xcube(:,:,l) = rx;
```

```
        % Get the current scenario time
        time = time + fmcwwav1.SweepTime;

        % Move targets forward in time for next sweep
        tgtPoses(1).Position=[tgtPoses(1).Position]+[tgtPoses(1).Velocity]*fmcwwav1.SweepTime;
        tgtPoses(2).Position=[tgtPoses(2).Position]+[tgtPoses(2).Velocity]*fmcwwav1.SweepTime;
end
```

Receiver Signal Processing on Interference-free Data Cube

Use the `phased.RangeResponse` object to perform the range processing on the interference-free radar data cubes. Use a Hann window to suppress the large sidelobes produced by the vehicles when they are close to the radar.

```
% Calculate number of range samples
Nrange = 2^nextpow2(Nft);

% Define range response
rngresp = phased.RangeResponse('RangeMethod','FFT', ...
    'SweepSlope',fmcwwav1.SweepBandwidth/fmcwwav1.SweepTime, ...
    'RangeFFTLengthSource','Property','RangeFFTLength',Nrange, ...
    'RangeWindow','Hann','SampleRate',fmcwwav1.SampleRate);

% Calculate the range response of interference-free data cube
XrngTgt = rngresp(XcubeTgt);
```

The victim radar decodes TDM-MIMO radar waveform by separating the `Nvt` TDM-MIMO transmit waveforms at each of its `Nrt` receive chains to form a virtual array of size `Nv`. After the waveform separation, the victim radar applies Doppler FFT on pulses for each range bin and virtual array element.

```
% Decode TDM-MIMO waveform
[XdecTgt,Nsweep] = helperTDMMIMODecoder(XrngTgt,wv);

% Number of Doppler samples
Ndoppler = 2^nextpow2(Nsweep);

% Size of virtual array
Nv = Nvr*Nvt;

% Doppler FFT with Hann window
XrngdopTgt = helperVelocityResp(XdecTgt,Nrange,Nv,Ndoppler);
```

Next, plot the range-Doppler map for the interference-free case. The range-Doppler map only shows the velocity from negative maximum unambiguous velocity to positive maximum unambiguous velocity. The maximum unambiguous velocity is inversely proportional to the pulse repetition interval on each transmit antenna elements. The range-Doppler map below shows that the target vehicle is located at near 22.0 m at a velocity of near 9.1 m/s and the interfering vehicle is located at near 47.6 m at a velocity of near -74.5 m/s.

```
% Pulse repetition interval for TDM-MIMO radar
tpri = fmcwwav1.SweepTime*Nvt;

% Maximum unambiguous velocity
vmaxunambg = lambda/4/tpri;
```

```
% Plot range-Doppler map
helperRangeVelocityMapPlot(XrngdopTgt,rngresp.SweepSlope,...
    fmcwwav1.SampleRate,tpri,rangeMax,vmaxunambg,Nrange,Ndoppler,lambda);
```
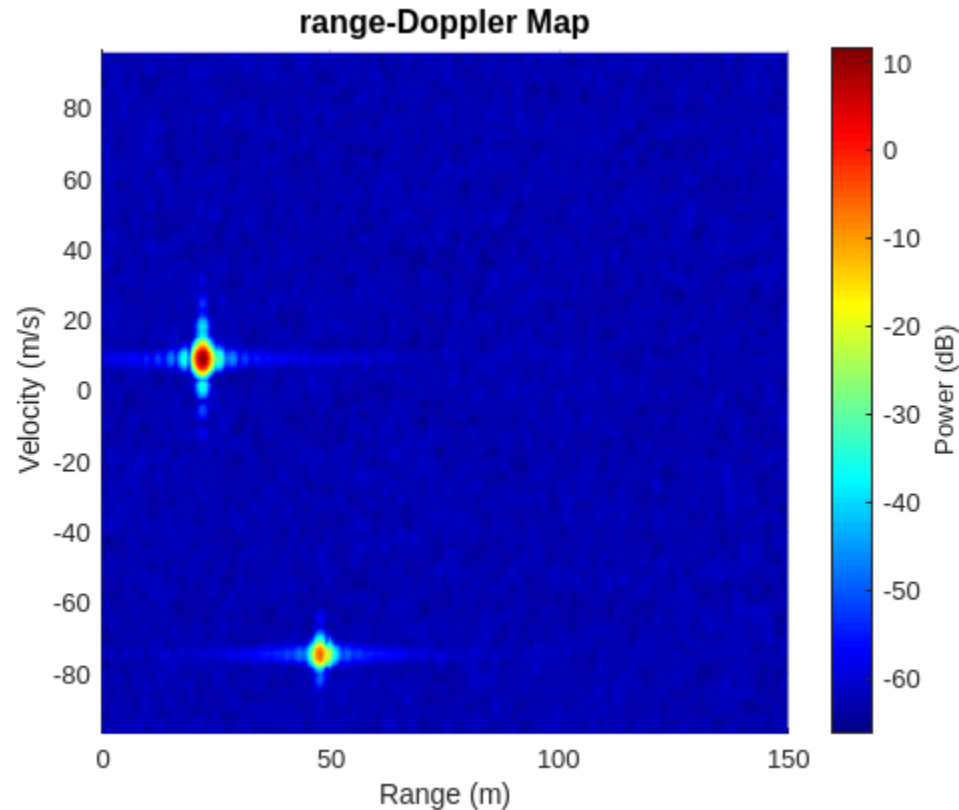

Apply angle FFT on virtual array at each range-Doppler bin for angle estimation. At the target's Doppler bin, the signal of the target vehicle is coherently integrated in the Doppler domain while the signal of the interfering vehicle with another velocity is not coherently integrated. Thus, in the range-angle map at the target's Doppler bin, the coherently integrated target's peak is strong. The range-angle map below shows that the target is located near 22.0 m and -3.6 degrees.

```
% Number of angle samples
Nangle = 2^nextpow2(Nv);

% Angle FFT with Hann window
XrngangdopTgt = helperAngleResp(XrngdopTgt,Nrange,Nangle,Ndoppler);

% Target Doppler bin
tarDopplerBin = ceil(tarVelocity/lambda*2*tpri*Ndoppler+Ndoppler/2);

% Plot range-angle map
helperRangeAngleMapPlot(XrngangdopTgt,rngresp.SweepSlope,...
    fmcwwav1.SampleRate,rangeMax,vrxEleSpacing,Nrange,Nangle,tarDopplerBin,lambda);
```
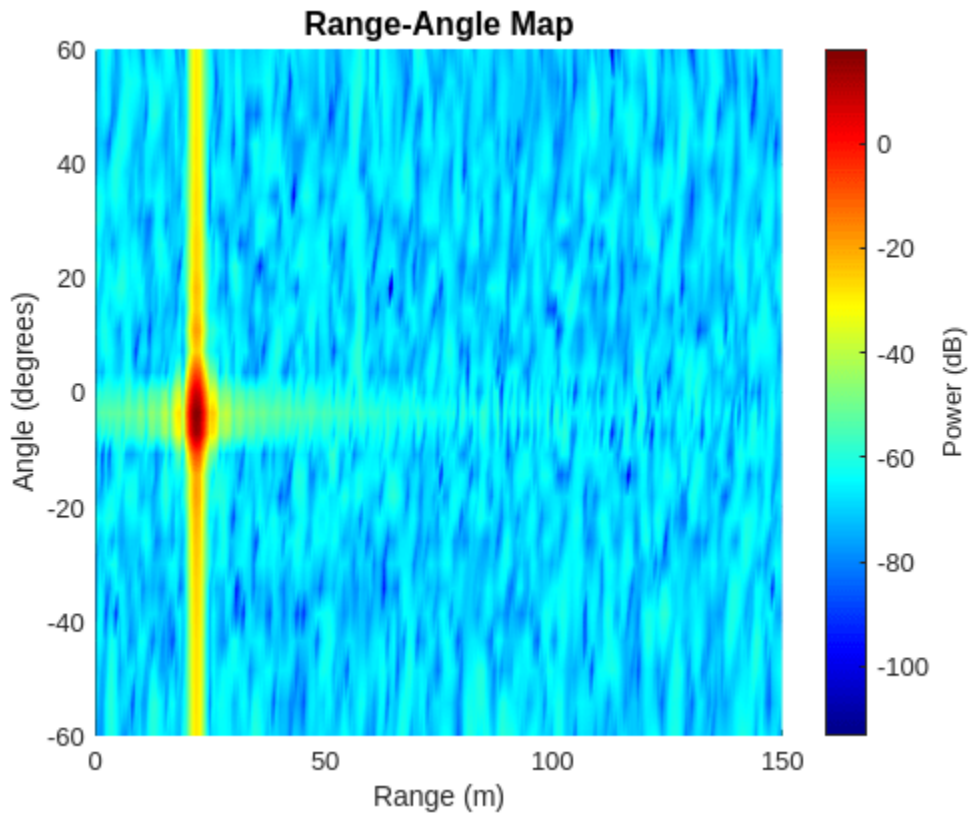
Receiver Signal Processing on Interfered Data Cube

Apply the same receiver signal processing workflow to the interfered data cube.

```
% Calculate the range response of interfered data cube
Xrng = rngresp(Xcube);

% Decode TDM-MIMO waveform
[Xdec,Nsweep] = helperTDMMIMODecoder(Xrng,wv);

% Velocity FFT with Hann window
Xrngdop = helperVelocityResp(Xdec,Nrange,Nv,Ndoppler);
```

Plot range-Doppler map of the interfered data cube. We can see the strong interference is a wideband signal that occupies a large number of range bins and hides the peaks of the two vehicles.

```
% Plot range-Doppler map
helperRangeVelocityMapPlot(Xrngdop,rngresp.SweepSlope,...
    fmcwwav1.SampleRate,tpri,rangeMax,vmaxunambg,Nrange,Ndoppler,lambda);
```

range-Doppler Map
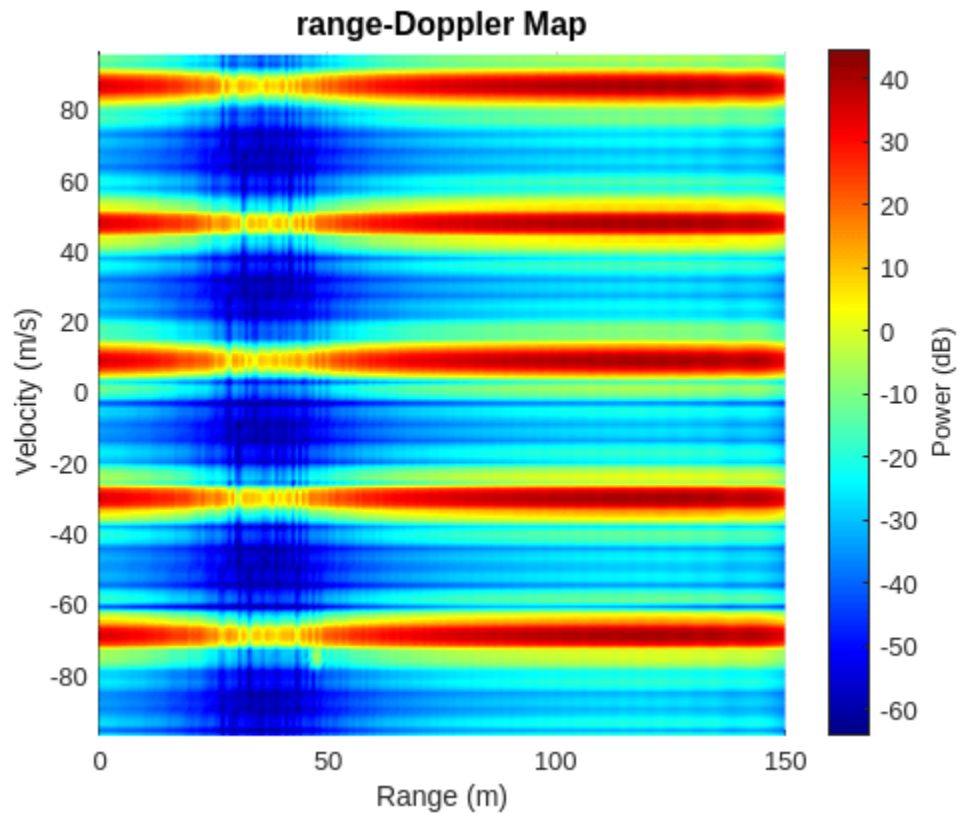
Plot range-angle map of the interfered data cube at the target's Doppler bin. We can see the strong interference raises the noise floor and hides the peak of the target. We can also observe that the interference angle of arrival is near 3.6 degrees.

```
% Angle FFT with Hann window
Xrngangdop = helperAngleResp(Xrngdop,Nrange,Nangle,Ndoppler);

% Plot range-angle map
helperRangeAngleMapPlot(Xrngangdop,rngresp.SweepSlope,...
    fmcwwav1.SampleRate,rangeMax,vrxEleSpacing,Nrange,Nangle,tarDopplerBin,lambda);
```
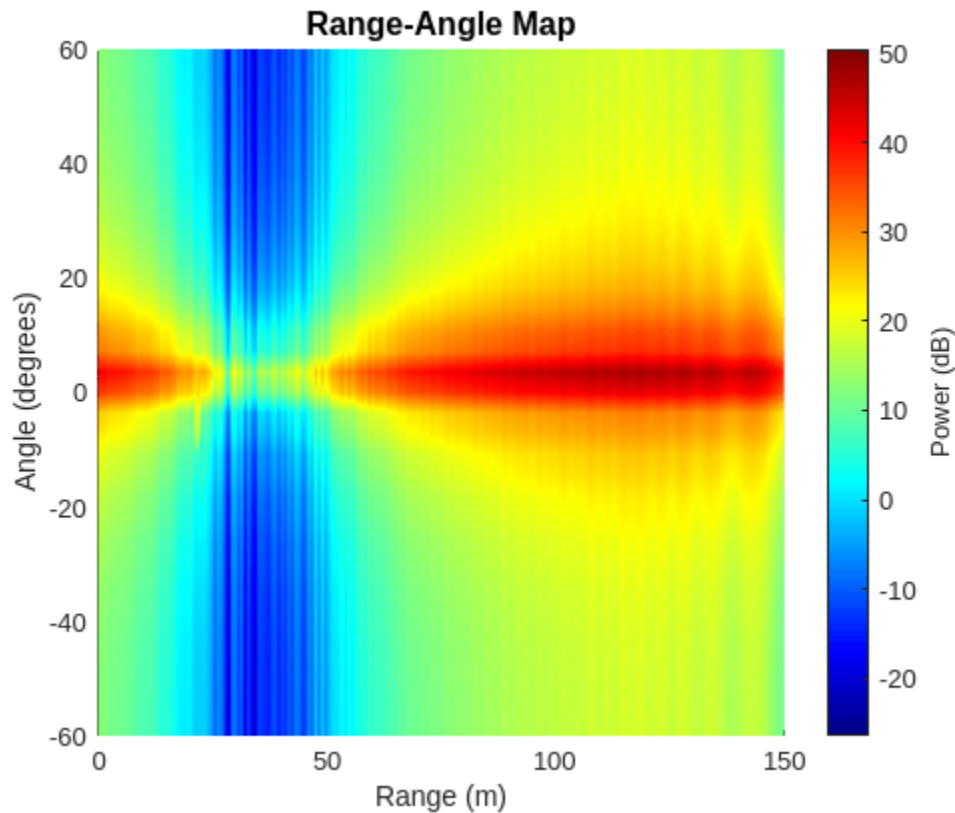
Interference Effect Animation

The following animation shows the interference effects for FMCW radars with different waveform parameters for the interfering radar. In this example, the bandwidth for both radars is set to be 150.0 MHz, the chirp duration for the victim radar is considered to be 5.0 μs, and the chirp duration for the interfering radar is changed from 2.0 μs to 5.0 μs with a step size of 0.33 μs. We can observe that when the chirp duration of the interfering radar changes, the range-Doppler map also changes. If the interference chirp slope is the same as the victim radar's chirp slope, point-like ghost targets may appear in the range-Doppler map. If the strong interference or its sidelobe is overlapped with the target on the target's range-Doppler bin, in the range-angle map, we can observe the interference angle of arrival and may observe the missing of the target due to the high noise level elevation caused by the direct-path interference.
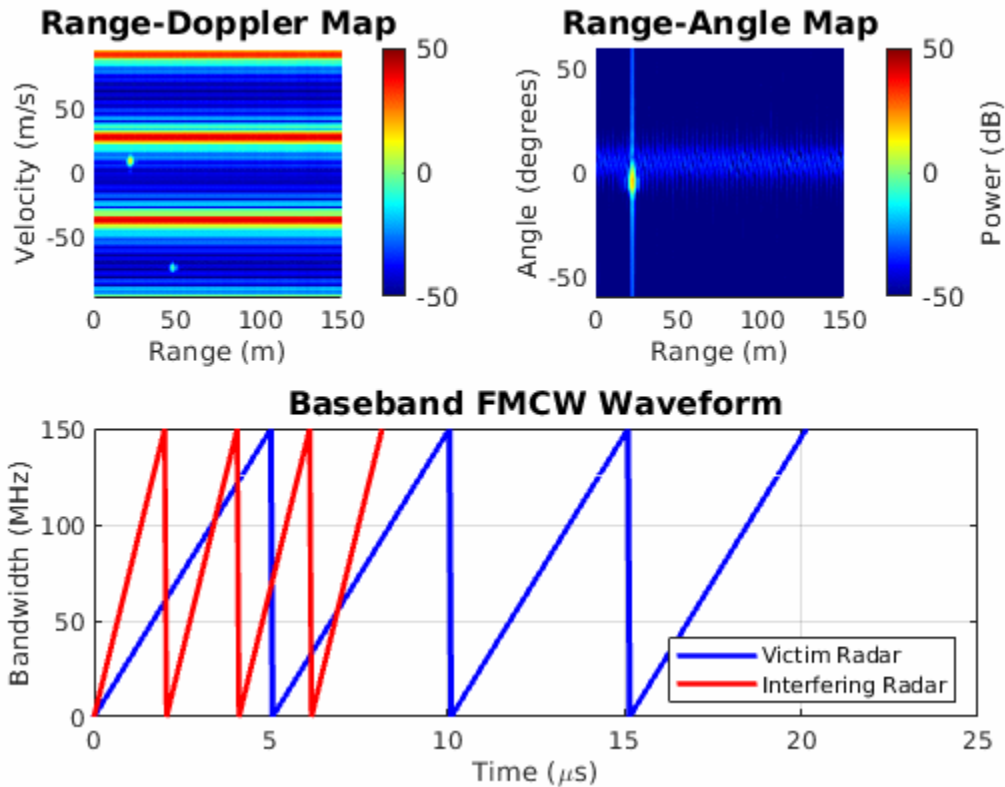
Summary

This example shows how to generate a highway driving scenario with interfering radars using Automated Driving Toolbox. This example also shows how to model the FMCW interference for MIMO automotive radars using Radar Toolbox. This example demonstrates the phenomena of mutual automotive radar interference and the need for interference mitigation algorithms.

Reference

[1] M. Richards. *Fundamentals of Radar Signal Processing*. New York: McGraw Hill, 2005.

[2] S. Jin, P. Wang, P. Boufounos, P. V. Orlik, R. Takahashi and S. Roy, "Spatial-Domain Interference Mitigation for Slow-Time MIMO-FMCW Automotive Radar," 2022 IEEE 12th Sensor Array and Multichannel Signal Processing Workshop (SAM), 2022, pp. 311-315.

Helper Functions

helperFMCWWaveform Function

```
function fmcwwav = helperFMCWWaveform(fc,maxrange,rangeres,maxvel)
c = physconst('LightSpeed');                    % Speed of light in air (m/s)
lambda = freq2wavelen(fc,c);                    % Wavelength (m)

% Set the chirp duration to be 5 times the max range requirement
tm = 5*range2time(maxrange,c);                  % Chirp duration (s)

% Determine the waveform bandwidth from the required range resolution
```

```
bw = rangeres2bw(rangeres,c);                     % Corresponding bandwidth (Hz)

% Set the sampling rate to satisfy both range and velocity requirements
sweepSlope = bw/tm;                               % FMCW sweep slope (Hz/s)
fbeatMax = range2beat(maxrange,sweepSlope,c);     % Maximum beat frequency (Hz)
fdopMax = speed2dop(2*maxvel,lambda);             % Maximum Doppler shift (Hz)
fifMax = fbeatMax+fdopMax;                        % Maximum received IF (Hz)
fs = max(2*fifMax,bw);                            % Sampling rate (Hz)

% Configure the FMCW waveform using the waveform parameters derived from the requirements
fmcwwav = phased.FMCWWaveform('SweepTime',tm,'SweepBandwidth',bw,'SampleRate',fs,'SweepDirection
end
```

helperInterferenceTransceiver Function

```
function iradar = helperInterferenceTransceiver(waveform,txPkPower,antGain,fc,itxArray,vrxArray)
% Reuse radarTransceiver for transmitting interference signal at the
% interfering radar and receiving interference signal at the victim radar
iradar = radarTransceiver("Waveform",waveform,'ElectronicScanMode','Custom');

% Sepcify interfering radar transmitter
iradar.Transmitter = phased.Transmitter('PeakPower',txPkPower,'Gain',antGain);

% Sepcify transmit array as the interfering radar transmit array
iradar.TransmitAntenna = phased.Radiator('Sensor',itxArray,'OperatingFrequency',fc,'WeightsInput

% Sepcify receive array as the victim radar receive array
iradar.ReceiveAntenna = phased.Collector('Sensor',vrxArray,'OperatingFrequency',fc);

% Specify victim radar receiver without noise
iradar.Receiver = phased.ReceiverPreamp('Gain',antGain,'NoiseMethod','Noise power','NoisePower',(
end
```

helperTDMMIMOEncoder Function

```
function w = helperTDMMIMOEncoder(Nt,Nsweep)
% Time division multiplexing (TDM) code applied on each antenna element
% at each sweep
w = zeros(Nt,Nsweep);
for l = 1:Nsweep
    wl = int8((1:Nt)' == mod(l-1,Nt)+1);
    w(:,l) = wl;
end
end
```

helperTDMMIMODecoder Function

```
function [Xdec,Nsweep] = helperTDMMIMODecoder(X,w)
[Nrange,Nr,Nsweep] = size(X);
Nt = size(w,1);

% Reduce number of sweeps after TDM decoding
Nsweep = floor(Nsweep/Nt);

% TDM decoding
X = X(:,:,1:Nsweep*Nt);
Xdec = reshape(X,Nrange,Nt*Nr,Nsweep);
end
```

helperGenerateTgtPaths Function

```matlab
function vpaths = helperGenerateTgtPaths(tgtPoses,actProf,lambda)
tgtPos = reshape([zeros(1,3) tgtPoses.Position],3,[]);
tgtVel = reshape([zeros(1,3) tgtPoses.Velocity],3,[]);

% Position point targets at half of each target's height
tgtPos(3,:) = tgtPos(3,:)+0.5*[actProf.Height];
% Number of Targets
Ntargets = length(tgtPos(:,1))-1;

vpaths = repmat(struct(...
    'PathLength', zeros(1, 1), ...
    'PathLoss', zeros(1, 1), ...
    'ReflectionCoefficient', zeros(1,1), ...
    'AngleOfDeparture', zeros(2, 1), ...
    'AngleOfArrival', zeros(2, 1), ...
    'DopplerShift', zeros(1, 1)),...
    1,Ntargets);

for m = 1:Ntargets
    % Each target is already in the path length, angle
    [plength,tang] = rangeangle(tgtPos(:,m+1),tgtPos(:,1),eye(3));

    % path loss
    ploss = fspl(plength,lambda);

    % reflection gain
    rgain = aperture2gain(actProf(m+1).RCSPattern(1),lambda);

    % Doppler
    dop = speed2dop(2*radialspeed(tgtPos(:,m+1),tgtVel(:,m+1),tgtPos(:,1),tgtVel(:,1)),...
        lambda);

    % Target paths to victim radar
    vpaths(m).PathLength = 2*plength;
    vpaths(m).PathLoss = 2*ploss;
    vpaths(m).ReflectionCoefficient = db2mag(rgain);
    vpaths(m).AngleOfDeparture = tang;
    vpaths(m).AngleOfArrival = tang;
    vpaths(m).DopplerShift = dop;
end
end
```

helperGenerateIntPaths Function

```matlab
function ipaths = helperGenerateIntPaths(tgtPoses,actProf,lambda)
tgtPos = reshape([zeros(1,3) tgtPoses.Position],3,[]);
tgtVel = reshape([zeros(1,3) tgtPoses.Velocity],3,[]);

% Position point targets at half of each target's height
tgtPos(3,:) = tgtPos(3,:)+0.5*[actProf.Height];
% Number of Targets
Ntargets = length(tgtPos(:,1))-1;

% Interference paths to victim radar
ipaths = repmat(struct(...
    'PathLength', zeros(1, 1), ...
```

```matlab
        'PathLoss', zeros(1, 1), ...
        'ReflectionCoefficient', zeros(1,1), ...
        'AngleOfDeparture', zeros(2, 1), ...
        'AngleOfArrival', zeros(2, 1), ...
        'DopplerShift', zeros(1, 1)),...
        1,Ntargets);

    % Each target is already in the path length, angle
    [plengthicar2vcar,tangicar2vcar] = rangeangle(tgtPos(:,3),tgtPos(:,1),eye(3));
    [plengthicar2tgt,tangicar2tgt] = rangeangle(tgtPos(:,3),tgtPos(:,2),eye(3));
    [plengthtgt2vcar,tangtgt2vcar] = rangeangle(tgtPos(:,2),tgtPos(:,1),eye(3));

    % path loss
    plossicar2vcar = fspl(plengthicar2vcar,lambda);  % from interfering radar to victim radar
    plossicar2tgt = fspl(plengthicar2tgt,lambda);    % from interfering radar to target
    plosstgt2vcar = fspl(plengthtgt2vcar,lambda);    % from target to victim radar

    % reflection gain
    rgainicar2vcar = aperture2gain(0.01,lambda);
    rgainicar2tgt = aperture2gain(actProf(2).RCSPattern(1),lambda);

    % Doppler
    dopicar2vcar = speed2dop(2*radialspeed(tgtPos(:,3),tgtVel(:,3),tgtPos(:,1),tgtVel(:,1)),...
        lambda);

    % Generate interference path from interfering radar to victim radar
    ipaths(1).PathLength = plengthicar2vcar;
    ipaths(1).PathLoss = plossicar2vcar;
    ipaths(1).ReflectionCoefficient = db2mag(rgainicar2vcar);
    ipaths(1).AngleOfDeparture = tangicar2vcar;
    ipaths(1).AngleOfArrival = tangicar2vcar;
    ipaths(1).DopplerShift = dopicar2vcar;

    % Generate interference path from interfering radar to target and from target to victim radar
    ipaths(2).PathLength = plengthicar2tgt+plengthtgt2vcar;
    ipaths(2).PathLoss = plossicar2tgt+plosstgt2vcar;
    ipaths(2).ReflectionCoefficient = db2mag(rgainicar2tgt);
    ipaths(2).AngleOfDeparture = tangicar2tgt;
    ipaths(2).AngleOfArrival = tangtgt2vcar;
    ipaths(2).DopplerShift = dopicar2vcar;
end
```

helperVelocityResp Function

```matlab
function Xrngdop = helperVelocityResp(Xdec,Nrange,Nv,Ndoppler)
Xrngdop = complex(zeros(Nrange,Nv,Ndoppler));
for n = 1: Nrange
    for m = 1:Nv
        % Data in different pulses
        XdecPulse = squeeze(Xdec(n,m,:));

        % Add Hann window on data
        XdecHannPulse = hanning(length(XdecPulse)).*XdecPulse;

        % Doppler FFT
        Xrngdop(n,m,:) = fftshift(fft(XdecHannPulse,Ndoppler));
    end
```

```
    end
end
```

helperAngleResp Function

```
function Xrngangdop = helperAngleResp(Xrngdop,Nrange,Nangle,Ndoppler)
Xrngangdop = complex(zeros(Nrange,Nangle,Ndoppler));
for n = 1: Nrange
    for l = 1: Ndoppler
        % Data in different virtual array elements
        XrngdopArray = squeeze(Xrngdop(n,:,l)');

        % Add Hann window on data
        XrngdopHannArray = hanning(length(XrngdopArray)).*XrngdopArray;

        % Angle FFT
        Xrngangdop(n,:,l) = fftshift(fft(XrngdopHannArray,Nangle));
    end
end
end
```

helperRangeVelocityMapPlot Function

```
function helperRangeVelocityMapPlot(Xrngdop,sweepSlope,sampleRate,tpri,rangeMax,vmaxunambg,Nrange
% Noncoherent integration over all virtual array elements
XrngdopPowIncoInt = 2*pow2db(squeeze(pulsint(Xrngdop)));

% Beat frequency grid
fbeat = sampleRate*(-Nrange/2:Nrange/2-1)'/Nrange;

% Convert beat frequency to range
rangeGrid = beat2range(fbeat,sweepSlope);

% Doppler frequency grid
fDoppler = 1/tpri*(-Ndoppler/2:Ndoppler/2-1)'/Ndoppler;

% Convert Doppler frequency to velocity for two-way propagation
velocityGrid = dop2speed(fDoppler,lambda)/2;

% Obtain range-Doppler meshgrid
[Range, Velocity] = meshgrid(rangeGrid,velocityGrid);

% Plot range-Doppler map
figure
surf(Range,Velocity,XrngdopPowIncoInt.')
view(0,90);
xlim([0, rangeMax]); ylim([-vmaxunambg,vmaxunambg])
xlabel('Range (m)'); ylabel('Velocity (m/s)'); zlabel('Power (dB)')
title('range-Doppler Map','FontSize',12); colormap(jet);
cb = colorbar; cb.Label.String = 'Power (dB)'; cb.FontSize = 10;
axis square; shading interp
end
```

helperRangeAngleMapPlot Function

```
function helperRangeAngleMapPlot(Xrngangdop,sweepSlope,sampleRate,rangeMax,vrxEleSpacing,Nrange,N
% Power of range-angle-Doppler response
XrngangdopPow = abs(Xrngangdop).^2;
```

```matlab
% Beat frequency grid
fbeat = sampleRate*(-Nrange/2:Nrange/2-1)'/Nrange;

% Convert beat frequency to range
rangeGrid = beat2range(fbeat,sweepSlope);

% Angle grid
angleGrid = asind(lambda/vrxEleSpacing*(-(Nangle/2):(Nangle/2)-1)'/Nangle);

% Obtain range-angle meshgrid
[Range, Angle] = meshgrid(rangeGrid,angleGrid);

% Plot range-angle map
figure
surf(Range, Angle, pow2db(squeeze(XrngangdopPow(:,:,tarDopplerBin)).'))
view(0,90);
xlim([0, rangeMax]); ylim([-60, 60]);
xlabel('Range (m)'); ylabel('Angle (degrees)'); zlabel('Power (dB)')
title('Range-Angle Map','FontSize',12); colormap(jet);
cb = colorbar; cb.Label.String = 'Power (dB)'; cb.FontSize = 10;
axis square; shading interp
end
```

Simulate a Coastal Surveillance Radar

This example shows how to simulate a stationary land-based radar that collects returns from extended targets at sea. You will see how to configure sea surface reflectivity models, modify a kinematic trajectory to match wave heights, and emulate a low-PRF radar system in a radar scenario.

Configure the Radar Scenario

Set the RNG for repeatable results.

```
rng default
```

The scenario will consist of a stationary radar looking out over a sea surface and observing a small ship target in clutter.

Radar

Start by defining the radar system parameters. Use an X-band radar at 10 GHz with a 5 degree nominal beamwidth. A linear array will be used.

```
freq = 10e9;
beamwidthAz = 5;
```

The nominal range resolution is 5 meters. A short pulse willl be used for simplicity.

```
rngRes = 5;
```

Data will be collected in range-time only. 80 pulses will be collected at a slow 100 Hz update rate so that target range migration is visible.

```
numPulses = 80;
updateRate = 100;
```

The `radarTransceiver` by default samples the entire unambiguous range swath. Since Doppler processing is not needed for this example, you can use a PRF specification on the radar that is much greater than the desired update rate. This decreases the number of range samples to speed up the simulation and avoid memory overflows. A PRF of 50 kHz has an unambiguous range of about 3 km, which is enough to contain the entire scenario within the first range ambiguity.

```
prfAct = 50e3;
```

The fast-time sampling rate must be a whole multiple of the actual PRF used. Use the desired range resolution to find the required sampling rate then adjust the PRF to match this constraint.

```
Fs = 1/range2time(rngRes);
prfAct = Fs/round(Fs/prfAct);
```

Calculate the number of uniformly-spaced elements required to achieve the desired beamwidth, and create a linear array with `phased.ULA`.

```
sinc3db = 0.8859;
numElems = ceil(2*sinc3db/(beamwidthAz*pi/180));
lambda = freq2wavelen(freq);
array = phased.ULA(numElems,lambda/2);
```

Now create the `radarTransceiver` object with these parameters.

```
rdr = radarTransceiver;
rdr.TransmitAntenna.OperatingFrequency = freq;
rdr.ReceiveAntenna.OperatingFrequency = freq;
rdr.TransmitAntenna.Sensor = array;
rdr.ReceiveAntenna.Sensor = array;
rdr.Waveform.PRF = prfAct;
rdr.Waveform.SampleRate = Fs;
rdr.Receiver.SampleRate = Fs;
rdr.Waveform.PulseWidth = 1/Fs; % short pulse with one sample per pulse
```

Scenario

Create a `radarScenario`. Use the update rate specified earlier and set the stop time to the end of the last update interval.

```
scenario = radarScenario(UpdateRate=updateRate,StopTime=numPulses/updateRate);
```

Place the radar 11 meters above the origin. The radar will face horizontally in the +X direction.

```
rdrHeight = 11;
platform(scenario,Sensors=rdr,Position=[0 0 rdrHeight]);
```

Use `surfaceReflectivitySea` to create the reflectivity model. You will use the Hybrid reflectivity model for vertical polarization and a sea state of 4. The reflectivity model has built-in multiplicative noise known as "speckle". Use a Weibull distribution for the speckle with a shape parameter of 0.7, and calculate the scale parameter required to keep the mean speckle at 1. When the mean is 1, speckle noise can increase the variability of the reflectivity without modifying the mean reflectivity.

```
seaState = 4;
spckShape = 0.7;
spckScale = 1/gamma(1+1/spckShape);
refl = surfaceReflectivitySea(Model='Hybrid',Polarization='V',SeaState=seaState,Speckle='Weibull
```

Now configure the surface region. Use a 650-by-650 meter patch of sea surface that begins 1 km out from the radar (in the +X direction). The `Boundary` parameter is specified in two-point form where the columns are opposite corners of the surface.

```
seaStartX = 1e3;
seaLen = 650;
bdry = [seaStartX; 0] + [0 1;-1/2 1/2]*seaLen;
```

Use the default sea surface spectral model with a surface resolution of no more than 1/4th of the range resolution of the radar. The actual resolution used needs to evenly divide the length of the surface.

```
surfRes = rngRes/4;
surfRes = seaLen/ceil(seaLen/surfRes);
spec = seaSpectrum(Resolution=surfRes);
```

Finally, use the `searoughness` function to get a wind speed that corresponds to the specified sea state, and create the surface object.

```
[~,~,windSpd] = searoughness(seaState);
seaSurface(scenario,Boundary=bdry,SpectralModel=spec,RadarReflectivity=refl,WindSpeed=windSpd);
```

Target

Now that the surface is defined, add the target to the scene. A cuboid target will be used to emulate a small boat. Use a target size of 6-by-1.5-by-1.5 meters. Target dimensions are specified with a struct

with fields `Length`, `Width`, `Height`, and `OriginOffset`. Let the target have a fixed RCS of 4 dBsm. Set the origin offset so that the position of the target will correspond to the center of the bottom face of the cuboid.

```
tgtRcs = 4; % dBsm
tgtDims = struct(Length=6,Width=1.5,Height=1.5,OriginOffset=[0 0 -1.5/2]);
```

Place the target in the middle of the sea surface patch at sea level. For a boat of this size, the bottom will be about 0.4 meters below the surface, known as the draft.

```
tgtPos = [seaStartX+seaLen/2, 0, 0];
tgtDraft = 0.4;
```

Update the Z coordinate of the target position so the bottom of the target is below the surface by the specified draft value. The height of the surface can be found with the `height` method on the `SurfaceManager` (note that this method takes column-oriented position vectors). This operation will be repeated inside the simulation loop to keep the target at a fixed height relative to the surface. With some of the target below the level of the surface, the observed RCS will be reduced.

```
tgtPos(3) = height(scenario.SurfaceManager,tgtPos.') - tgtDraft;
```

Let the target travel at 10 m/s on a heading of 150 degrees, which is 30 degrees off of a course directly towards the radar, and form the velocity vector.

```
tgtSpd = 10;
tgtHdg = 150;
tgtVel = tgtSpd*[cosd(tgtHdg) sind(tgtHdg) 0];
```

With the target position and speed defined, create a kinematic trajectory and add the target to the scene. The `Orientation` property of the trajectory is a rotation matrix that converts scenario-frame vectors to platform-frame vectors. Use `rotz` and transpose the result to get the rotation matrix that corresponds to the specified heading.

```
tgtOrient = rotz(tgtHdg).';
tgtTraj = kinematicTrajectory(Position=tgtPos,Velocity=tgtVel,Orientation=tgtOrient);
platform(scenario,Trajectory=tgtTraj,Dimensions=tgtDims,Signatures=rcsSignature(Pattern=tgtRcs))
```

Clutter Generation

Use the `clutterGenerator` method on the scenario to create a `ClutterGenerator` object and enable clutter generation for the radar. Set the clutter resolution to 1/4th the range resolution, and disable the automatic mainbeam clutter.

```
clut = clutterGenerator(scenario,rdr,Resolution=rngRes/4,UseBeam=false);
```

Calculate parameters for a `RingClutterRegion` that will encompass as much of the surface as possible with a constant azimuth coverage so that there's no bias in the magnitude of clutter return with range due to the finite size of the surface. The azimuth center is 0 since the radar is pointed in the +X direction.

```
azcov = 2*asind(seaLen/2/(seaStartX+seaLen));
minr = sqrt(tand(azcov/2)^2+1)*seaStartX;
maxr = seaStartX + seaLen;
azcen = 0;
```

Check that the computed azimuth coverage is at least 4.5x the 3 dB azimuth beamwidth in order to fully capture surface returns through the first sidelobes, then use the `ringClutterRegion` method to designate the region for clutter generation.

```
azcov/beamwidthAz
```

```
ans = 4.5439
```

```
ringClutterRegion(clut,minr,maxr,azcov,azcen);
```

Run the Simulation and Collect Returns

Use `theaterPlot` to visualize the scenario configuration before running. The `surfacePlotter` and `clutterRegionPlotter` are used to visualize the surface heights along with the regions designated for clutter generation. The `RingClutterRegion` created earlier will be shown with a red outline. The clutter region is plotted in a horizontal plane at a specified height. Use 5 m for the clutter region plot height so the region is visible above the surface in the top-down view.

```
tp = theaterPlot(XLimits=bdry(1,:),YLimits=bdry(2,:));
pp = platformPlotter(tp,Marker='o',MarkerFaceColor='magenta',MarkerEdgeColor='magenta',DisplayNam
cp = clutterRegionPlotter(tp,RegionFaceColor='none',RegionEdgeColor='red',DisplayName='Clutter Re
sp = surfacePlotter(tp,DisplayName='Sea Surface');
regPlotHeight = 5;
plotPlatform(pp,tgtPos,tgtVel,tgtDims,tgtOrient)
plotClutterRegion(cp,clutterRegionData(clut,regPlotHeight))
plotSurface(sp,surfacePlotterData(scenario.SurfaceManager))
title('Scenario Overview')
```
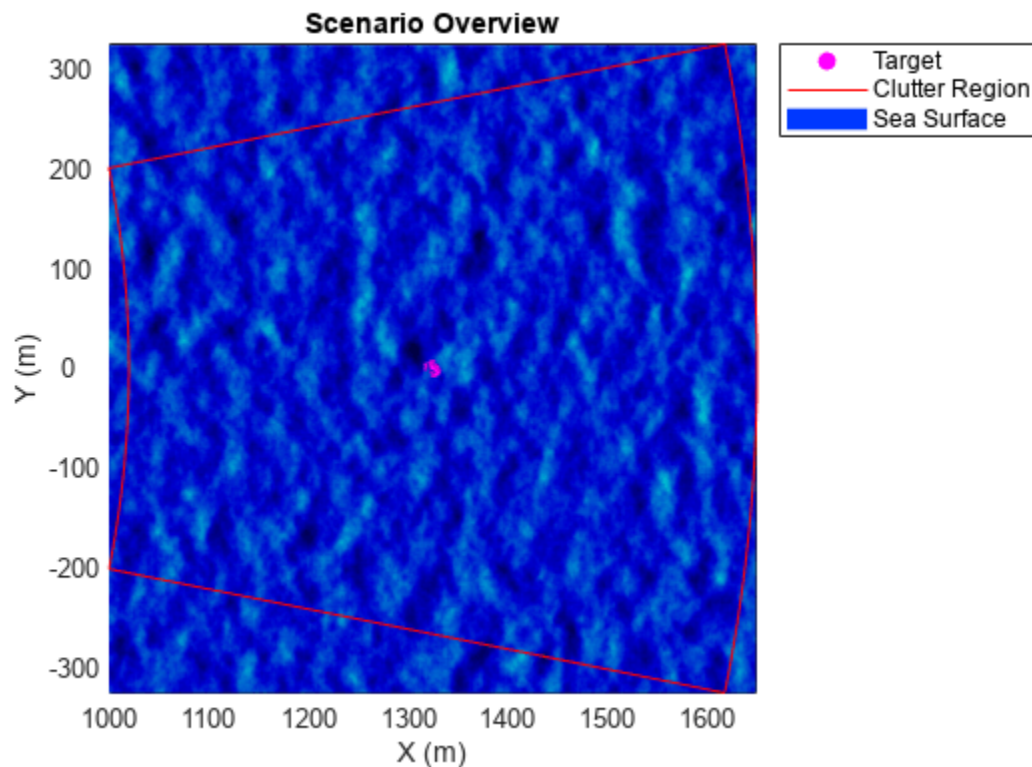
Notice that the ring-shaped clutter region covers as much of the surface as possible without going outside the surface boundary.

The `advance` method on the scenario is used to advance the simulation by one frame and determine if the stop time has been reached. Inside the simulation loop, start by updating the Z coordinate of the target position to match the current surface height. The `receive` method of the scenario is used to collect IQ signals received by radars in the scene. The return is summed over array elements and stored in the matrix PH.

```
frame = 0;
while advance(scenario)
    frame = frame + 1;

    % Update target position
    tgtTraj.Position(3) = height(scenario.SurfaceManager,tgtTraj.Position.') - tgtDraft;

    % Collect sum-beam range profiles
    iqsig = receive(scenario);
    PH(:,frame) = sum(iqsig{1},2);
end
```

Use the `phased.RangeResponse` object to perform matched filtering of the range profiles. The required matched filter coefficients are retrieved from the `Waveform` property on the radar.

```
resp = phased.RangeResponse(RangeMethod="Matched filter",SampleRate=Fs);
[PH,rngGates] = resp(PH,getMatchedFilter(rdr.Waveform));
```

Clip the data to only include range bins that contain some clutter return, as defined by the min and max radius used for the `RingClutterRegion` earlier.

```
minRng = sqrt(rdrHeight^2 + minr^2);
maxRng = sqrt(rdrHeight^2 + maxr^2);
minGate = find(rngGates >= minRng,1,'first');
maxGate = find(rngGates <= maxRng,1,'last');
rngGates = rngGates(minGate:maxGate);
PH = PH(minGate:maxGate,:);
```

Now plot the range-time data.

```
imagesc(1:numPulses,rngGates/1e3,20*log10(abs(PH)))
ch = colorbar;
ch.Title.String = 'dBW';
cl = clim;
clim([cl(2)-60, cl(2)])
set(gca,'ydir','normal')
xlabel('Pulse Index')
ylabel('Range (km)')
title('Pulse Data')
```
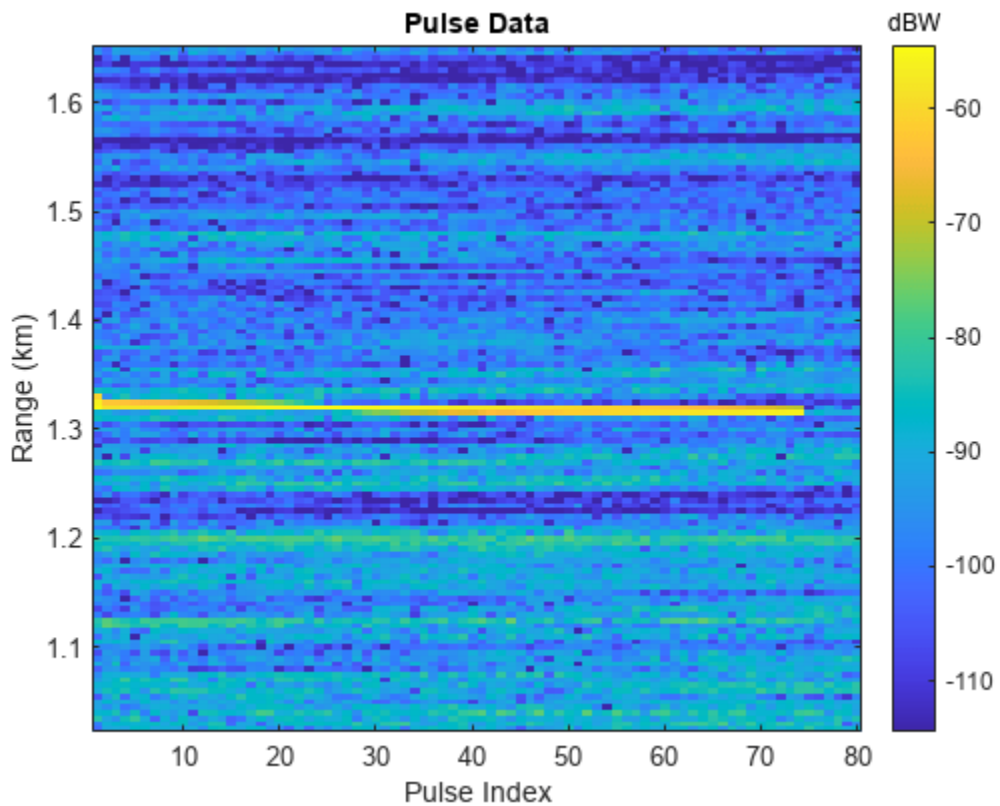
The target signal is visible around the center range, with a couple bins of range migration present.

Conclusion

In this example you simulated a stationary coastal surveillance radar with sea surface clutter. An extended cuboid target was used to emulate a ship with partial occlusion by the sea surface. Range migration of the target was visible due to the high resolution and slow update rate of the radar.

Simulate an Automotive 4D Imaging MIMO Radar

This example shows how to simulate a 4D imaging MIMO radar for automotive applications using `radarTransceiver`. You will learn how to generate I/Q data with the radar in a simulated environment. The I/Q data will be processed to generate a point cloud of radar detections.

Model a MIMO Radar Transceiver

Use `radarTransceiver` to create a signal-level radar transceiver model for an FMCW automotive MIMO radar. You will use this model to simulate baseband I/Q signals at the output of the radar receiver.

```matlab
rng default; % Set random seed for reproducible results
xcvrMIMO = radarTransceiver();

% Create FMCW waveform
rgRes = 0.5; % meters
rgMax = 150; % meters
sweepBW = rangeres2bw(rgRes); % Hz
Fs = sweepBW;

% Set the sweep time to at least twice the max range time to avoid power
% loss at the max range extent. Longer sweep times give higher
% signal-to-noise ratios (SNRs) at greater ranges, but at the cost of
% maximum ambiguous velocity
sweepTime = 4*range2time(rgMax); % s
Ncp = 512; % Number of chirps

wfm = phased.FMCWWaveform( ...
    'SweepBandwidth',sweepBW, ...
    'SweepTime',sweepTime, ...
    'SampleRate',Fs);
xcvrMIMO.Waveform = wfm;
xcvrMIMO.Receiver.SampleRate = Fs;
xcvrMIMO.NumRepetitions = Ncp;

% Set receiver gain and noise level
xcvrMIMO.Receiver.NoiseFigure = 12;

disp(xcvrMIMO);

  radarTransceiver with properties:

             Waveform: [1×1 phased.FMCWWaveform]
          Transmitter: [1×1 phased.Transmitter]
      TransmitAntenna: [1×1 phased.Radiator]
       ReceiveAntenna: [1×1 phased.Collector]
             Receiver: [1×1 phased.ReceiverPreamp]
    MechanicalScanMode: 'None'
    ElectronicScanMode: 'None'
      MountingLocation: [0 0 0]
        MountingAngles: [0 0 0]
   NumRepetitionsSource: 'Property'
        NumRepetitions: 512
     RangeLimitsSource: 'Property'
           RangeLimits: [0 Inf]
```

```
        RangeOutputPort: false
         TimeOutputPort: false
```

Define MIMO Transmit and Receive Arrays

MIMO radar is widely used for automotive radar applications due to its ability to realize small angular resolutions in azimuth and elevation within a small package. For high-resolution 4D radar, beamwidths in azimuth and elevation less than 2 degrees are desired. Compute the number of array elements that would be needed to realize a 2 degree beamwidth for a linear array.

```
fc = 77e9; % Hz
lambda = freq2wavelen(fc); % m

% Compute array aperture
d = beamwidth2ap(2,lambda,0.8859);

% Compute number of array elements required for half-wavelength spacing
disp(2*d/lambda)
```

```
    50.7583
```

If conventional arrays with elements at half-wavelength spacing were used, an array with 50 elements would be required to realize a resolution of 2 degree in azimuth, and another 50 elements would be needed if a similar resolution was desired in elevation. If a uniform rectangular array (URA) was to be used, a total of 50 x 50 = 2500 elements would be required.

Instead, let's define a virtual array which is composed of separate transmit and receive antenna arrays. Create a uniform linear array (ULA) for the transmit antenna with half-wavelength spacing and a rectangular receive array which does not maintain a half-wavelength spacing in the azimuth dimension.

```
% Define transmit ULA element positions
Ntx =   10 ━━━━━▽━━━━━━━━━━━  ;
txPos = [ ...
    0:Ntx-1; ...        % y positions
    zeros(1,Ntx)];      % z positions
txPos = sortrows(txPos',[2 1])';

% Define receive URA element positions
Naz =   5 ━━━━▽━━━━━━━━  ; % Azimuth array is number of transmit elements times half-wavel
Nel =   50 ━━━━━━▽━━━━━  ; % Elevation array at half-wavelength spacing

[yPos,zPos] = meshgrid((0:Naz-1)*Ntx,0:Nel-1);
rxPos = [yPos(:) zPos(:)]';

% Sort the rows to allow the virtual array elements to be assembled later
rxPos = sortrows(rxPos',[2 1])';
```

Use `helperVirtualArray` to generate the transmit and receive arrays and the corresponding virtual array element positions.

```
[txArray,rxArray,vxPos] = helperVirtualArray(txPos,rxPos,lambda);
helperViewArrays(txPos,rxPos,vxPos);
```
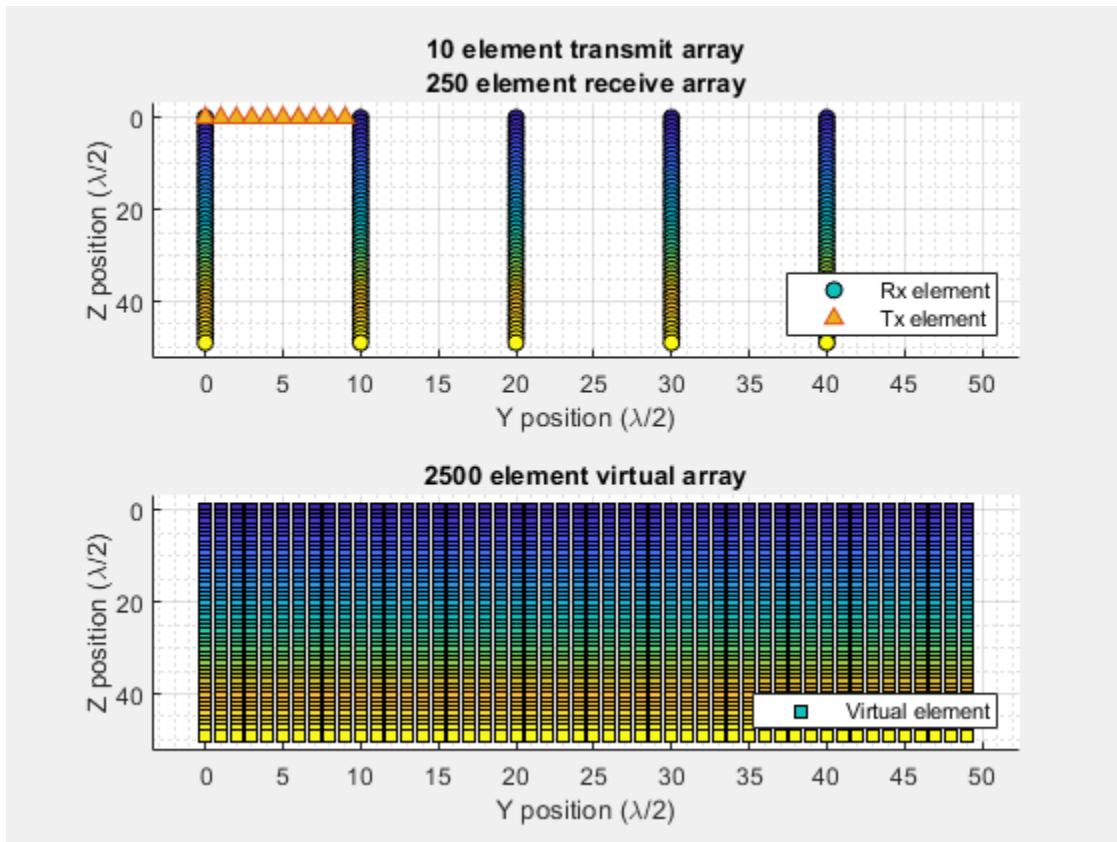
A virtual array with 2500 elements is created by using a 10 element transmit array and a 5 x 50 = 250 element receive array. This results in a significant reduction in the number of required elements and hardware complexity. For more information on how a virtual array is defined, see the "Increasing Angular Resolution with Virtual Arrays" example.

Use `helperPlotDirectivities` to plot the directivities for the transmit, receive, and virtual arrays in azimuth and elevation.

```
bwAz = helperPlotDirectivities(txArray,rxArray,fc);
```
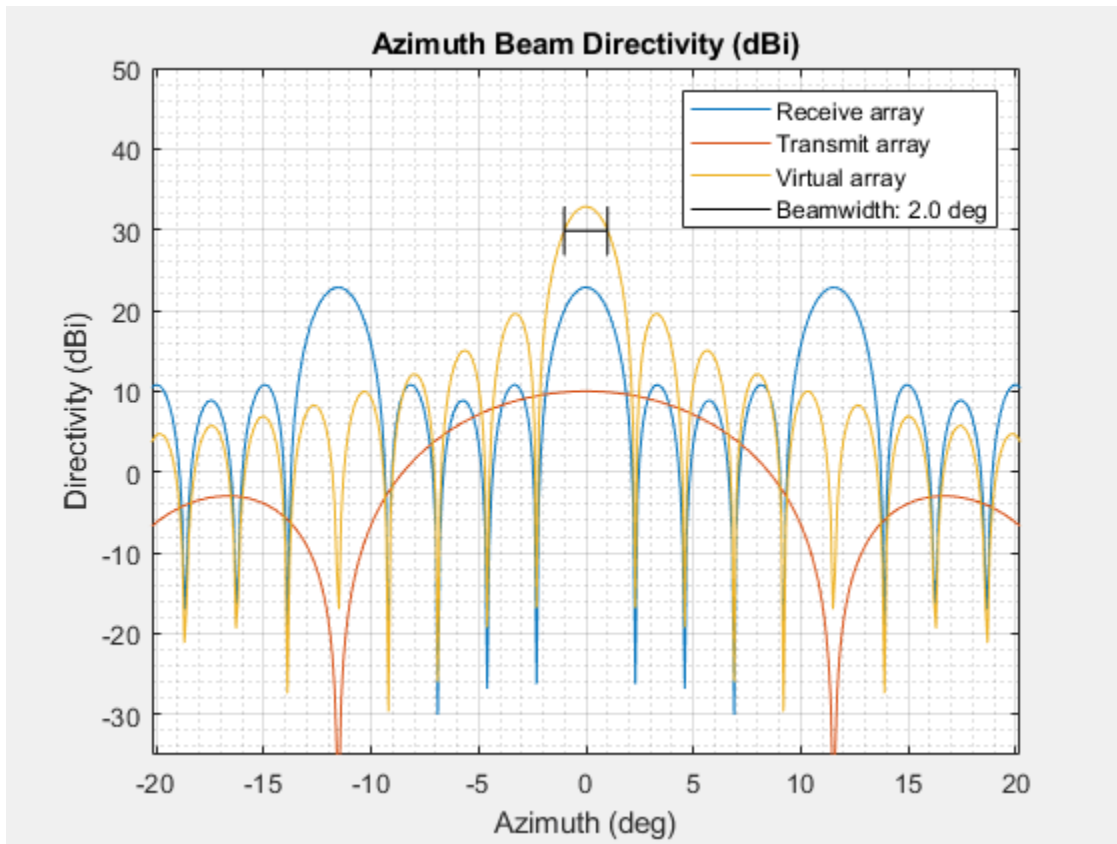
```
bwEl = helperPlotDirectivities(txArray,rxArray,fc,'Elevation');
```
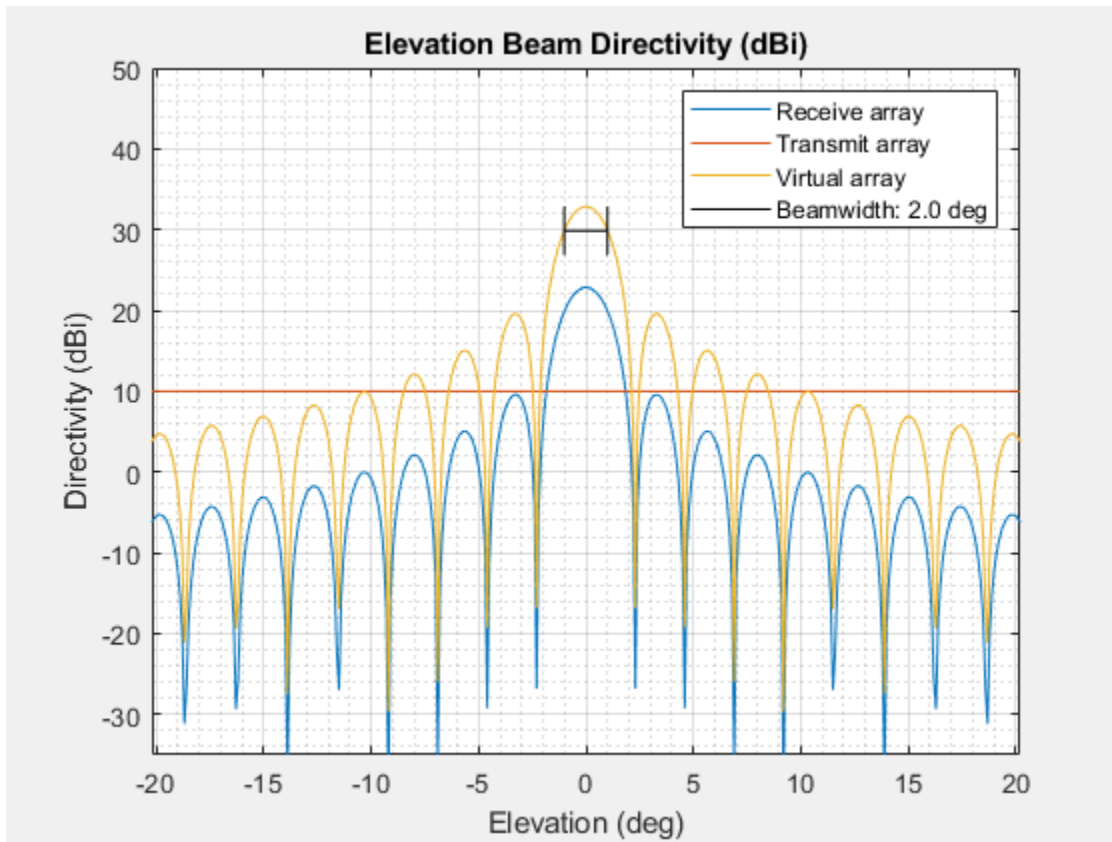
```
% Compute the total number of elements used
disp(getNumElements(txArray) + getNumElements(rxArray))
```

```
260
```

We realized a 2 degree beamwidth in azimuth and elevation by using 260 elements instead of the 2500 elements that would have been needed for a full rectangular array.

Attach the transmit and receive arrays used to define the virtual array to the MIMO radar model.

```
% Attach transmit and receive arrays
xcvrMIMO.TransmitAntenna.Sensor = txArray;
xcvrMIMO.TransmitAntenna.OperatingFrequency = fc;
xcvrMIMO.ReceiveAntenna.Sensor = rxArray;
xcvrMIMO.ReceiveAntenna.OperatingFrequency = fc;

% Configure transmitter for MIMO mode
xcvrMIMO.ElectronicScanMode = 'Custom';
xcvrMIMO.TransmitAntenna.WeightsInputPort = true;

% Set transmitter gain and power level
xcvrMIMO.Transmitter.PeakPower = db2pow(10);
```

Define MIMO DDMA Waveform

An essential characteristic of any MIMO radar implementation is a waveform that maintains orthogonality along all the transmit antennas. This orthogonality is often realized by encoding the transmitted waveforms in time, frequency, or code domains. Maintaining this orthogonality across the

transmit antennas is essential in order to assemble the virtual array used by the MIMO radar as will be demonstrated later in the Signal Processing on page 1-1064 section of this example. Time Division Multiple Access (TDMA) is a common technique used to maintain orthogonality in the time domain. TDMA only transmits from one antenna element at any given time. Only using one transmit element at a time maintains the needed orthogonality but at the cost of reducing the total transmit power and decreasing the maximum unambiguous velocity of the radar. For more information on TDMA MIMO radar, see the "Increasing Angular Resolution with Virtual Arrays" example.

Instead of maintaining orthogonality in the time domain, new techniques using the frequency and code domains have been proposed. One such technique, Doppler Division Multiple Access (DDMA) MIMO maintains orthogonality in the frequency domain by shifting the waveforms transmitted by each antenna in Doppler. The motivation of this conventional DDMA scheme is to separate transmit signals from the `Ntx` transmit antennas into `Ntx` Doppler frequency subbands at the receiver. This is realized by DDMA MIMO weights which modulate each transmit antenna's signal into its own Doppler frequency subband. A drawback of this technique is that the maximum ambiguous velocity is still reduced by the number of transmit antennas due to the `Ntx` peaks from the transmitted signals spread across the Doppler space. An adaptation of the conventional DDMA MIMO which introduces an *unused* region of Doppler without regularly spaced peaks by adding *virtual* transmit elements to the array is used here. This enables the maximum ambiguous velocity of the radar's waveform to be maintained. [1 on page 1-1074], [2 on page 1-1074]

Create a DDMA waveform for the MIMO radar that assigns a unique Doppler offset to each of the transmit elements so that each element will transmit the same FMCW waveform, but with a unique offset in Doppler. Generate the Doppler offsets uniformly across the Doppler space for the 10 transmit antennas and 2 virtual antennas. This will assign two Doppler offsets to *unused* virtual transmit elements. These unused Doppler offsets will enable Doppler ambiguities to be unwrapped, preventing a reduction in the maximum ambiguous velocity of the MIMO radar.

```
% Define an (Ntx + Nvtx)-by-Ncp DFT matrix where Nvtx > 0 so that Nvtx
% Doppler bins are unused
Nvtx =    2  ⬦————————————   ;

[dopOffsets,Nv] = DDMAOffsets(Ntx,Nvtx);
% Recompute Ncp so that it is a multiple of Nv. This is important for
% recovering transmitted signal.
Ncp = ceil(Ncp/Nv)*Nv;
xcvrMIMO.NumRepetitions = Ncp;

% Select Ntx consecutive Doppler offsets for the DDMA weights
wDDMA = dopMatrix(dopOffsets(1:Ntx),Ncp);
% Show the DDMA Doppler offsets
plotDDMAOffsets(dopOffsets,Ntx);
```
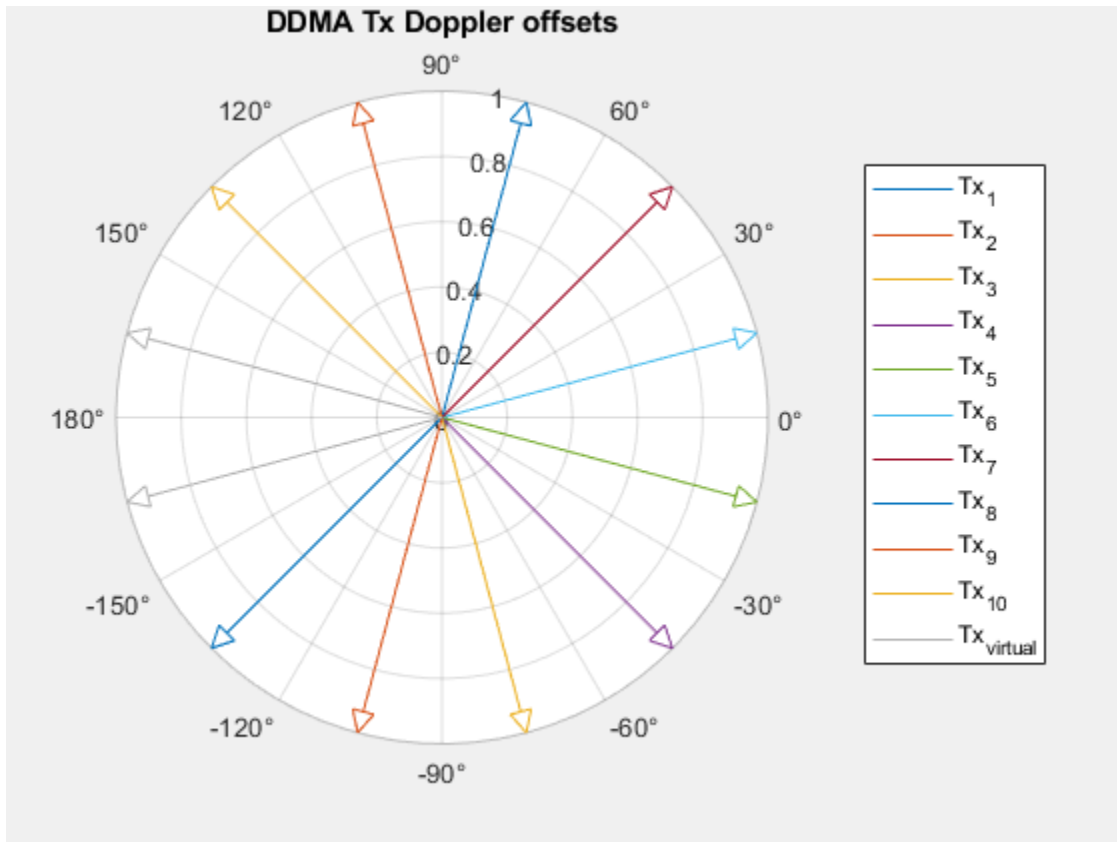
The FMCW waveform transmitted on each of the 10 elements will have a Doppler offset corresponding to each of the vectors in the previous figure. The DDMA weights have been chosen so that they are centered on zero Doppler. In the preceding figure, Doppler is represented as degrees on the unit circle by normalizing the Doppler shift frequencies from [-PRF/2,PRF/2] by PRF/2π. The Doppler shift applied to transmit element is -135 degrees.

Simulate Radar Data Cube

Use `helperCreateOverpassScenario` to create a `drivingScenario` that includes a highway overpass along with a vehicle merging into the lane in front of the ego vehicle. Mount the MIMO radar to the front of the ego vehicle as a forward-looking radar.

```
% Create drivingScenario
[scenario, egoVehicle] = helperCreateOverpassScenario();

% Mount MIMO radar as forward-looking radar on the ego vehicle
xcvrMIMO.MountingLocation(1) = egoVehicle.Length-egoVehicle.RearOverhang;
xcvrMIMO.MountingLocation(3) = 0.5;
xcvrMIMO.MountingAngles(2) = -5; % Pitch radar upward, away from the road's surface

% Define a coverage configuration structure for use with helperSetupDisplay
fov = [85 25]; % [azimuth elevation] degrees
% rgRes = bw2rangeres(wfm.SweepBandwidth); % meters
covCfg = struct( ...
    'SensorLocation',xcvrMIMO.MountingLocation(1:2), ...
    'MaxRange',rgMax, ...
    'Yaw',xcvrMIMO.MountingAngles(1), ...
```
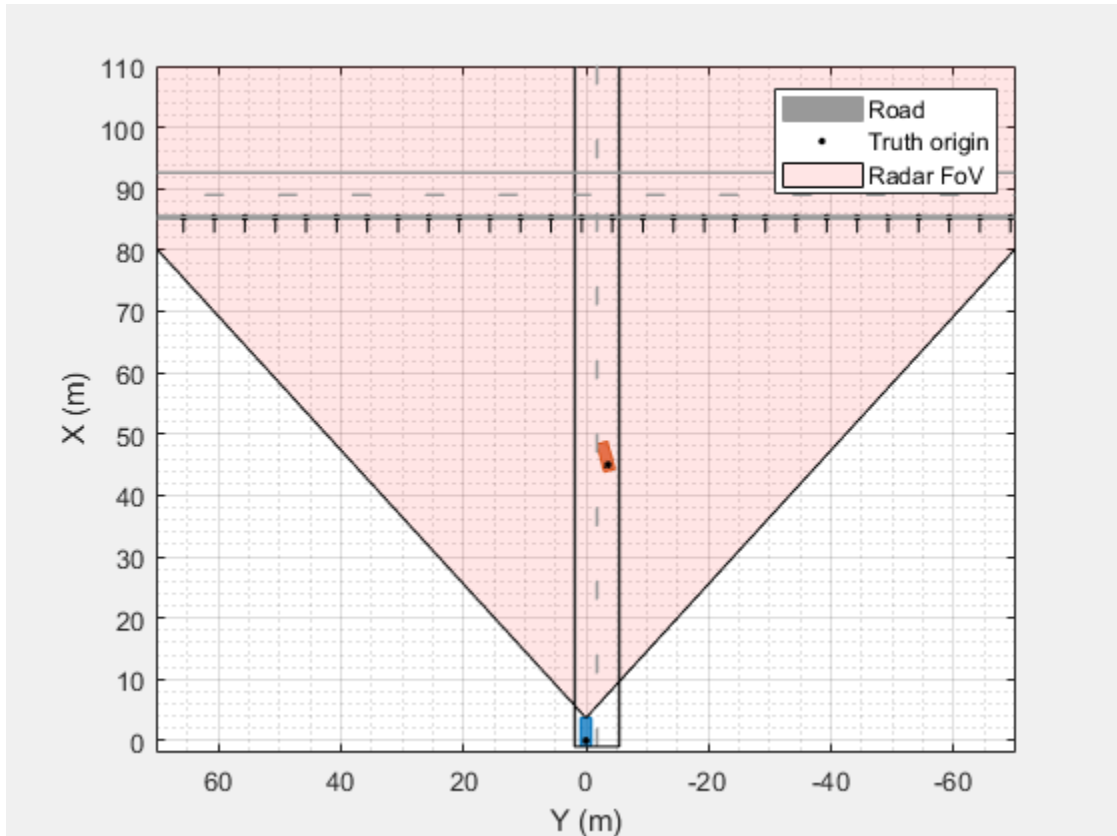
```
    'FieldOfView',fov, ...
    'AzimuthResolution',bwAz, ...
    'ElevationResolution',bwEl, ...
    'RangeResolution',rgRes);

% Plot scenario
bep = helperSetupDisplay(egoVehicle,covCfg);
delete(bep.Plotters(end));
axis(bep.Parent,[-2 110 70*[-1 1]]);
```


Use `helperPosesToPaths` to generate propagation path structs from the scenario.

```
% Convert target poses to path structs
tposes = targetPoses(egoVehicle);
paths = helperPosesToPaths(tposes,xcvrMIMO,scenario,covCfg);
```

Use the MIMO radar model to generate a "Radar Data Cube" from the propagation path structs. The radar data cube contains the I/Q signals corresponding to the complex-valued baseband samples collected at the output of the radar's receiver.

```
time = scenario.SimulationTime;
datacube = xcvrMIMO(paths,time,conj(wDDMA));
```

Process MIMO Radar Data Cube

Range, Doppler, beamforming, and detection processing need to be applied to the data cube to extract the detection point cloud. For DDMA MIMO, the range and Doppler processing and CFAR detection are applied first so that the virtual array elements can be assembled prior to beamforming.

Range and Doppler Processing

Use `phased.RangeDopplerResponse` to take the 2D FFT along the first and third dimensions of the data cube to transform the fast- and slow-time dimensions of the data cube into range and Doppler bins respectively. Use Hann windows along the fast- and slow-time dimensions to suppress sidelobes in range and Doppler.

```
wfm = xcvrMIMO.Waveform;
rgdpProc = phased.RangeDopplerResponse( ...
    'RangeMethod','FFT', ...
    'SampleRate',wfm.SampleRate, ...
    'SweepSlope',wfm.SweepBandwidth/wfm.SweepTime, ...
    'DechirpInput',true, ...
    'RangeWindow','Hann', ...
    'DopplerWindow','Hann', ...
    'DopplerOutput','Frequency');
refSig = wfm();
[rgdpCube,rgBins,dpBins] = rgdpProc(datacube,refSig);

% Limit data to ranges of interest
inRg = rgBins>=0 & rgBins<=covCfg.MaxRange;
rgBins = rgBins(inRg);
rgdpCube = rgdpCube(inRg,:,:);
dpBins = -dpBins;
```

CFAR Detection

Use a constant false-alarm rate (CFAR) detector to identify the locations of the target returns in the range and Doppler processed data cube.

```
% Compute the power of each cell in the data cube
pwrCube = abs(rgdpCube).^2;
```

Configure the CFAR detector to exclude the multiple peaks generated in Doppler for each target due to the Doppler offsets applied at each transmit element by the DDMA MIMO waveform.

```
% CFAR configuration example
%    [++++---o---++++]
%    o: cell under test
%    -: 6 guard cells
%    +: 8 training cells
fpks = Ncp/Nv; % Distance between DDMA peaks
cfar = phased.CFARDetector( ...
    'NumGuardCells',even(ceil(fpks*0.5)), ... % Use 1/2 of space between DDMA Doppler peaks as gu
    'ProbabilityFalseAlarm',1e-6, ...
    'NoisePowerOutputPort',true);
cfar.NumTrainingCells = 2*(fpks-cfar.NumGuardCells); % Use remaining space for training
```

Compute the locations of the target returns in the range and Doppler processed data cube for each receive element.

```
% Reshape data cube to perform CFAR averaging along the Doppler dimension.
[~,Nrx,Ndp] = size(rgdpCube);
data = reshape(pwrCube,[],Ndp);
[detMtx,nPow] = cfar(data,1:size(data,1));

% Reshape the detection matrix to match the dimensions of the original
% range and Doppler processed data cube.
```

```
detMtx = reshape(detMtx,size(pwrCube));

% Compute the signal-to-noise ratio (SNR) for the range-Doppler map.
nPow = reshape(nPow,size(pwrCube));
snrCube = pwrCube./nPow;

% Identify the range and Doppler bins where the same target appears across
% the receive elements in the data cube.
detMtx = squeeze(sum(detMtx,2)); % Count the number of common target returns.

% Plot the range and Doppler map.
prf = 1/xcvrMIMO.Waveform.SweepTime;
helperPlotRangeDopplerDDMA(snrCube,rgBins,dpBins,prf,Ntx,Nv);
```
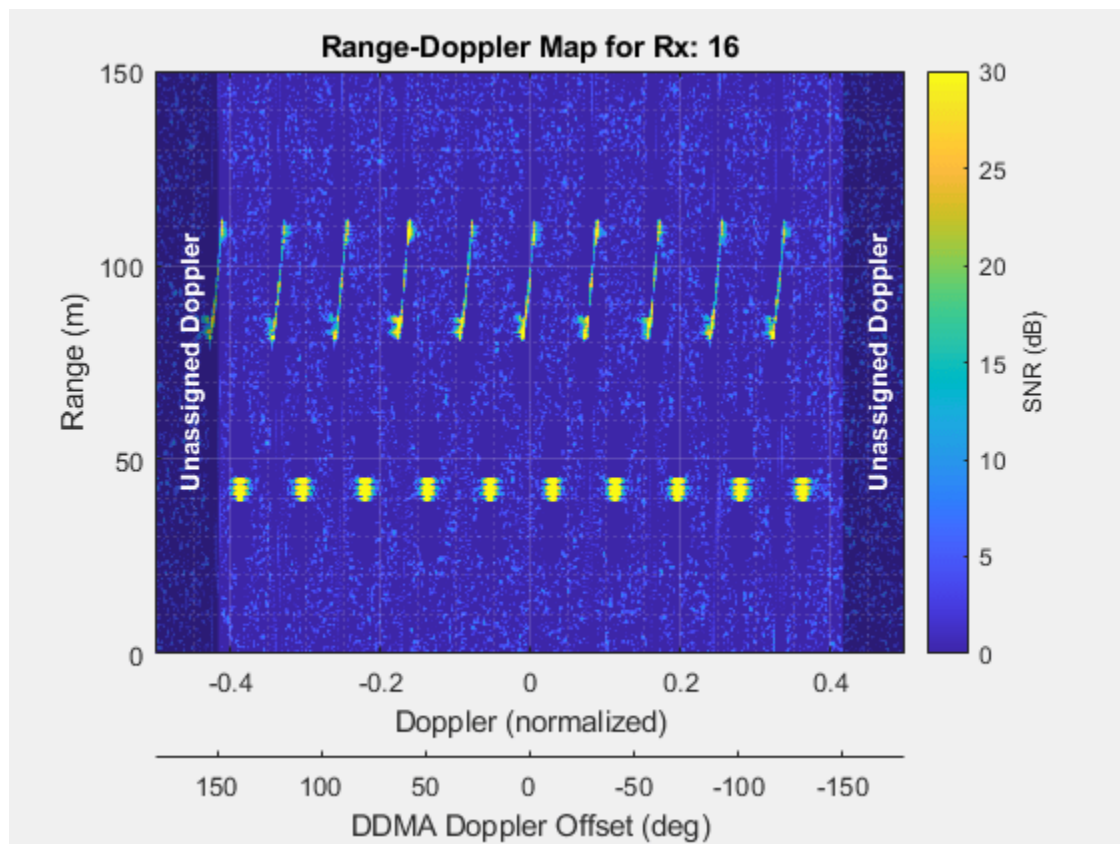

A range and Doppler map for one of the receive elements is shown in the previous figure. The returns from the target vehicle merging into the lane in front of the ego vehicle appear at a range of 40 meters from the radar and returns corresponding to the highway overpass are located at approximately 80 meters. There are ten returns modulated in Doppler at the same range for each target corresponding to the ten Doppler offsets applied across each of the ten transmit elements. A gap in Doppler between the first and last transmit element's target returns exists because of the virtual transmit antennas used to compute the DDMA Doppler offsets. If no virtual transmit elements were used, it would be impossible to identify the first and last elements of the transmit array due to the cyclic nature of the Doppler domain.

Remove Doppler Ambiguities

Create a matched filter to identify the location of the first transmit element for each of the target returns locations identified by the CFAR detector.

```
% Generate Doppler transmit test sequence
dPks = Ncp/Nv; % Number of Doppler bins between modulated DDMA target peaks
mfDDMA = zeros(dPks,Ntx);
mfDDMA(1,:) = 1;
mfDDMA = mfDDMA(:)';

fig = findfig('DDMA Matched Filter'); ax = axes(fig);
stem(ax,mfDDMA);
xlabel(ax,'Doppler bin (k)'); ylabel(ax,'h_k');
title(ax,fig.Name);
grid(ax,'on'); grid(ax,'minor');
```
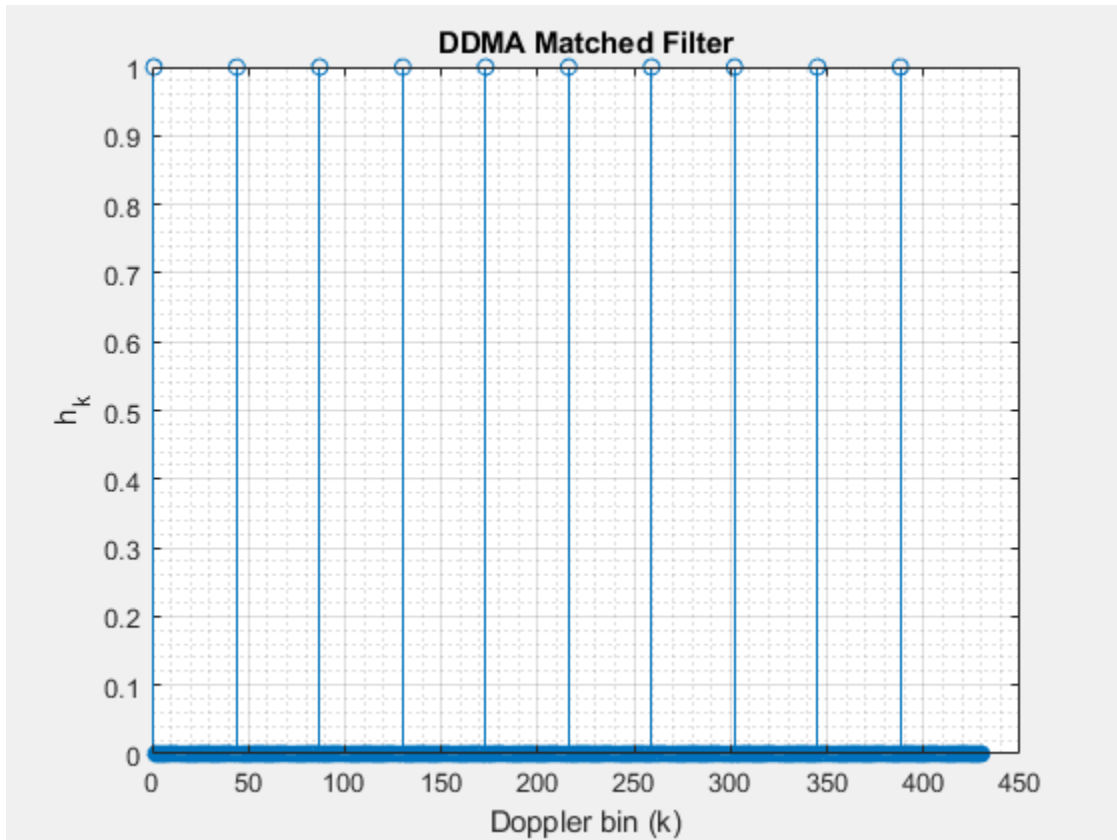

Compute the cyclical cross-correlation between the DDMA matched filter and the detection matrix returned by the CFAR detector to find the Doppler bins corresponding to the target returns generated by the waveform from the first transmit element.

```
% Compute cyclical cross-correlation.
[Nrg,Ndp] = size(detMtx);

% Require the same target to appear across the receive channels.
corrMtx = detMtx>=Nrx*0.8;
corrMtx = ifft( fft(corrMtx,Ndp,2) .* fft(mfDDMA,Ndp,2), Ndp,2);
```

```
rdDetMtx = corrMtx>=Ntx*0.95;

% Multiple detection crossings will occur due to the waveform's ambiguity
% function in range and Doppler. Isolate the local maxima for each group of
% detections in the range-Doppler image.
snrDetMtx = squeeze(sum(snrCube,2));
snrDetMtx(~rdDetMtx) = NaN;
rdDetMtx = islocalmax(snrDetMtx,1);

% Identify the range and Doppler bins associated with localized detection
% maxima.
iRD = find(rdDetMtx(:));
[iRgDet,iDpDet] = ind2sub(size(rdDetMtx),iRD);
rgDet = rgBins(iRgDet);

% Show the locations of the unambiguous detections corresponding to the
% signal returns from the first transmit element.
ax = findaxes('Range-Doppler Map'); hold(ax,'on');
p = plot(ax,dpBins(iDpDet)/prf,rgDet,'rd','DisplayName','Unambiguous detections');
legend(p,'Unambiguous detections','Location','southwest');
```
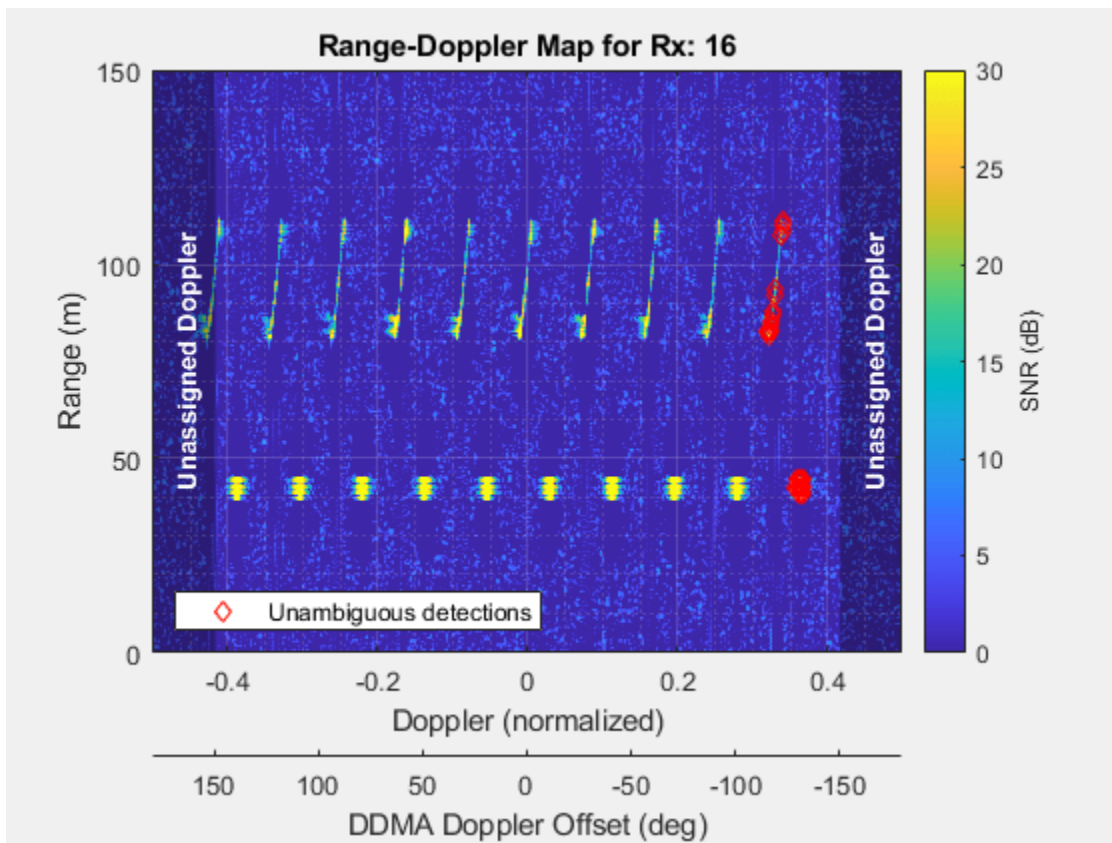

The previous figure shows the locations of the target returns generated by the first transmit element. The merging target vehicle, located at approximately 40 meters in front of the radar is moving at approximately the same velocity as the ego vehicle, so it should have a Doppler offset of zero degrees. However, because a DDMA Doppler offset is applied to the FMCW waveform transmitted from each transmit element, we should expect this DDMA offset in addition to the Doppler offset due to the velocity difference between the ego vehicle and the target vehicle. The returns from the first transmit

element should then include the -135 degrees Doppler offset due to the DDMA modulation applied to that element as well.

Remove DDMA Doppler offset applied to the first transmit element to recover the Doppler offset attributed to the velocity difference between the ego and target vehicles. Compute the range-rate of the target returns from the corrected Doppler offsets.

```
% Apply Doppler correction for DDMA Doppler shift applied on the first
% transmit element.
fd = (0:Ncp-1)/Ncp-0.5;
f1 = dopOffsets(1);
dpDet = mod(fd(iDpDet)+f1+0.5,1)-0.5;

% Compute unambiguous range-rate of detected Doppler bin
prf = 1/xcvrMIMO.Waveform.SweepTime;
dpDet = dpDet*prf;
rrDet = dop2speed(dpDet,lambda)/2;

% Show the range and range-rate extracted from the detected target returns.
fig = findfig('Range-Doppler Detections'); ax = axes(fig);
plot(ax,rrDet,rgDet,'s');
xlabel(ax,'Range-rate (m/s)'); ylabel('Range (m)');
title(ax,'Range-Doppler Detections');
ax.YDir = 'normal';
grid(ax,'on'); grid(ax,'minor'); ax.Layer = 'top';
axis(ax,[dop2speed(prf/2,lambda)/2*[-1 1] 0 covCfg.MaxRange]);
```
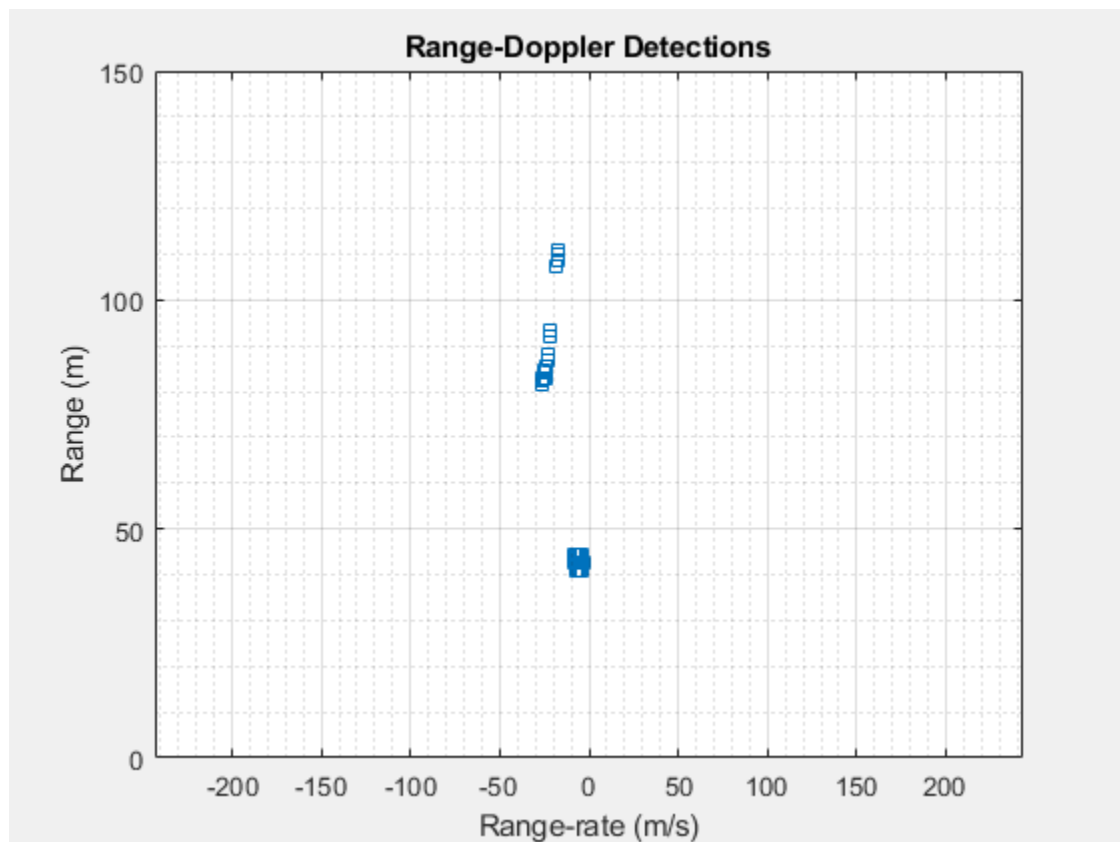
Virtual MIMO Array Beamspace Processing

Use the location of the target returns detected for the first transmit elements to identify the locations of the target returns from the other nine trans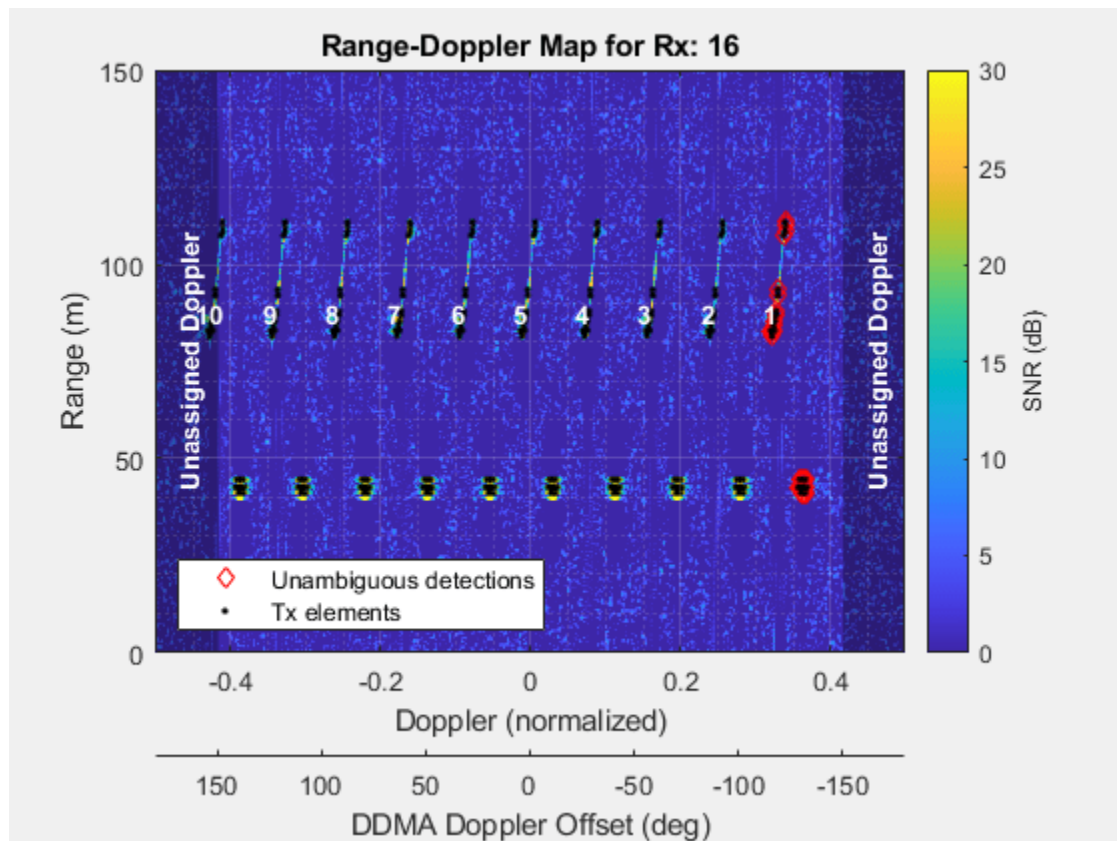mit elements to assemble the virtual elements for the virtual MIMO array for each target detection. The virtual elements correspond to every possible pair of transmit and receive array elements.

```
% Assemble virtual array elements from range-Doppler detections.
dPks = Ncp/Nv; % Number of Doppler bins between modulated DDMA target peaks
iDpTx = mod(iDpDet(:)-(0:Ntx-1)*dPks-1,Ncp)'+1; % Transmit elements are isolated in Doppler.
iRgTx = repmat(iRgDet,1,Ntx)';
```

```
% Plot the locations of the target returns to be extracted for the ten
% transmit elements in the range and Doppler map.
plotTxArrayReturns(rgBins,dpBins,iRgTx,iDpTx,Ntx,prf);
```


The range and Doppler bins corresponding to the target returns from the ten transmit elements have been recovered for each receive element.

Use the range and Doppler bin locations for each of the transmit elements to assemble the elements of the virtual array for each target detection.

```
% Extract virtual elements from range-Doppler data cube for each detection.
iTx = sub2ind([size(rgdpCube,3) size(rgdpCube,1)],iDpTx(:),iRgTx(:));
vxTgts = permute(rgdpCube,[2 3 1]); % Nrx x Ntx x Nrg
vxTgts = vxTgts(:,iTx); % Nrx x (Ntx x Ndet)
vxTgts = reshape(vxTgts,Nrx,Ntx,[]); % Nrx x Ntx x Ndet
```

Perform beamforming by computing a 2D FFT along the elements in the virtual array corresponding to the azimuth and elevation dimensions of the virtual array. Use Hanning windows along the azimuth and elevation dimensions to suppress sidelobes in those directions.

```matlab
% Apply DDMA Doppler offset correction to the virtual array's targets.
wgts = dopMatrix(dopOffsets(1:Ntx));
vxTgts = vxTgts.*wgts';

% Reshape the virtual array target returns so that the elements
% corresponding the azimuth and elevation dimensions lie along the second
% and third dimensions of the data cube.
vxTgts = permute(vxTgts,[3 2 1]); % Ndet x Ntx x Nrx
vxTgts = reshape(vxTgts,[],Naz*Ntx,Nel); % Ndet x Naz x Nel
% Apply azimuth taper along the azimuth dimension of the data cube.
Nvxaz = size(vxTgts,2);
winAz = hanning(Nvxaz);
vxTgts = reshape(winAz,1,Nvxaz,1).*vxTgts;

% Apply elevation taper along the elevation dimension of the data cube.
Nvxel = size(vxTgts,3);
winEl = hanning(Nvxel);
vxTgts = reshape(winEl,1,1,Nvxel).*vxTgts;

% Apply 2D FFT along azimuth and elevation dimensions of data cube.
azelTgts = fft(fft(conj(vxTgts),Nvxaz,2),Nvxel,3);
azelTgts = fftshift(fftshift(azelTgts,2),3);

% Compute azimuth and elevation bins of beamformed data cube.
azBins = asind(2*((0:size(azelTgts,2)-1)/size(azelTgts,2))-1);
elBins = asind(2*((0:size(azelTgts,3)-1)/size(azelTgts,3))-1);
```

Use a CFAR detector to estimate the signal-to-noise ratio (SNR) of the detections in the azimuth and elevation beams.

```matlab
% Estimate signal-to-noise ratio from detection beams
pwrCube = abs(azelTgts).^2;

cfar = phased.CFARDetector( ...
    'NumGuardCells',4, ...
    'NumTrainingCells',16, ...
    'NoisePowerOutputPort',true);

% Reshape data to perform CFAR averaging for each detection across azimuth
% and elevation bins.
pwrCube = permute(pwrCube,[2 1 3]); % Naz x Ndet x Nel
[~,nPow] = cfar(pwrCube(:,:),1:size(pwrCube,1));
nPow = reshape(nPow,size(pwrCube));
pwrCube = permute(pwrCube,[2 1 3]); % Ndet x Naz x Nel
nPow = permute(nPow,[2 1 3]); % Ndet x Naz x Nel
snrCube = pwrCube./nPow;
snrdB = pow2db(snrCube);
```

Use the SNR to compute the weighted average for the detections to estimate the azimuth and elevation of the target returns.

```matlab
% Find the azimuth, elevation, and range bins associated with each target
% detection.
[zBins,yBins] = meshgrid(elBins,azBins);
```

```
xGrid = repmat(rgDet(:),[1 size(yBins)]);
yBins = reshape(yBins,[1 size(yBins)]);
zBins = reshape(zBins,[1 size(zBins)]);
yBins = repmat(yBins,numel(rgDet),1,1);
zBins = repmat(zBins,numel(rgDet),1,1);

% Exclude detections outside of the field of view or with a low SNR.
isGd = snrdB>10 & abs(yBins)<covCfg.FieldOfView(1)/2 & abs(zBins)<covCfg.FieldOfView(2)/2;

% Use a weighted average to estimate the azimuth and elevation of the
% target detections.
wgts = snrCube.*isGd;
wgts = wgts./sum(wgts,3);
azDet = sum(wgts.*yBins,3);
elDet = sum(wgts.*zBins,3);

% Find the max SNR for each target detection.
snrdBDet = max(snrdB,[],3);
```

Use the azimuth and elevation estimates to compute 3D Cartesian point-cloud locations for the target detections.

```
% Compute point-cloud locations from azimuth, elevation, and range bins.
rgs = repmat(rgDet(:),1,size(azDet,2));
[x,y,z] = sph2cart(deg2rad(azDet(:)),deg2rad(elDet(:)),rgs(:));
locDet = [x(:) y(:) z(:)];

% Remove excluded detection points.
isGd = ~isnan(x(:));
locDet = locDet(isGd,:);
snrdBDet = snrdBDet(isGd);

% Plot the point cloud detections in the simulated scenario.
ax = helperPlotPointCloudDetectionsDDMA(locDet,snrdBDet,egoVehicle,xcvrMIMO,covCfg);
campos(ax,[-10 0 5]);
camtarget(ax,[60 0 -10]);
camup(ax,[0.18 0 0.98]);
```
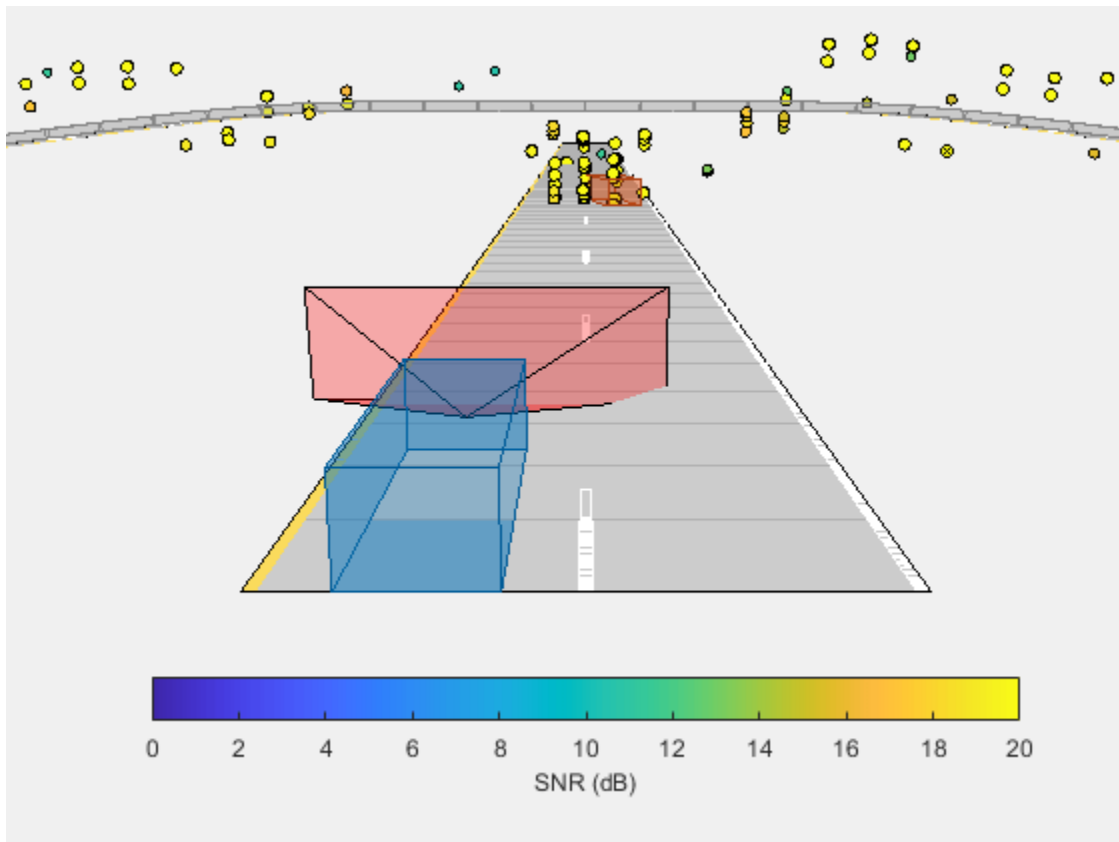
The preceding figure shows the point cloud detections overlayed upon the 3D scenario truth. This figure demonstrates the ability of the 4D MIMO radar to discriminate the difference in elevation between the detections of the merging target vehicle on the road in front of the radar and the overpass crossing over the road.

Reposition the figure's camera better illustrate the locations of the overpass detections.

```
campos(ax,[145 60 30]);
camtarget(ax,[60 0 0]);
camup(ax,[-0.35 -0.25 0.90]);
```
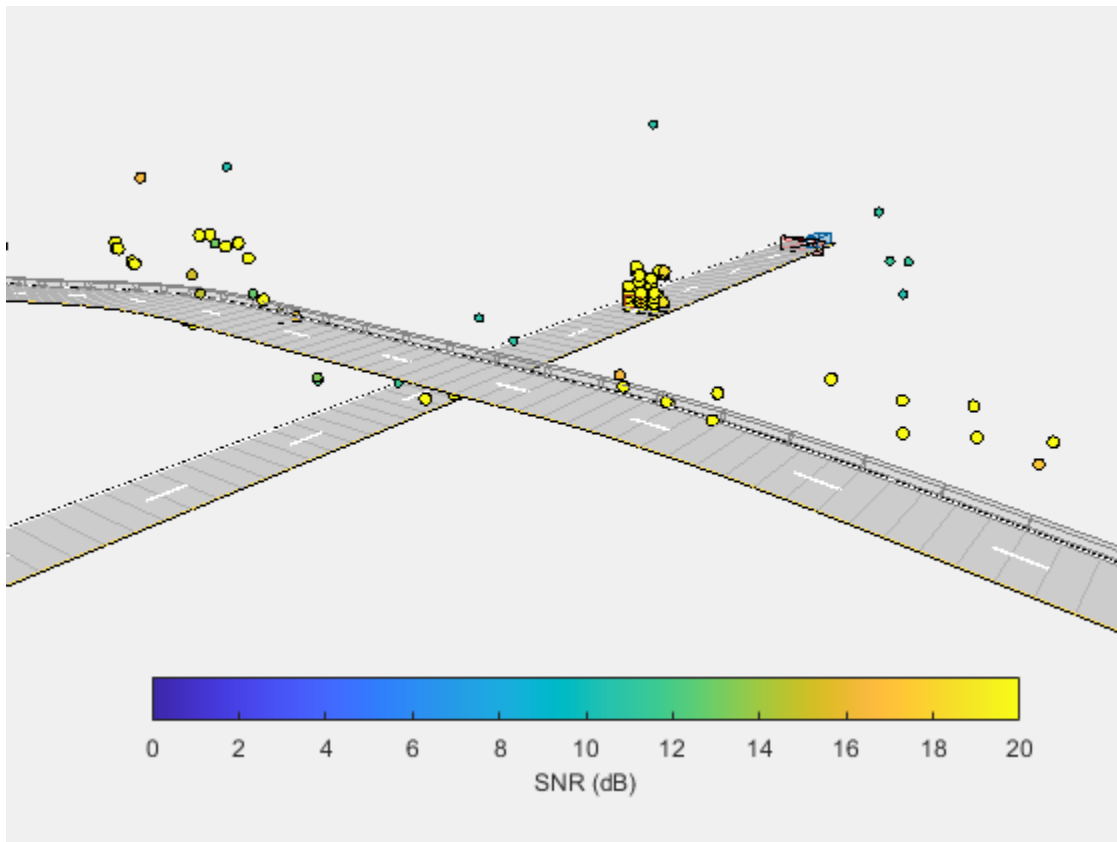
The preceding figure shows the same scenario and point cloud detections viewed from above the overpass and looking back towards the ego vehicle and radar. The overpass detections are correctly located along the elevated road's surface of the overpass. The target vehicle's detections are also observed to lie on the surface of the road passing beneath the overpass.

Summary

In this example, you learned how to model a high resolution 4D radar using DDMA MIMO techniques. By implementing a DDMA MIMO radar, you were able to simultaneously transmit across all the transmit array elements while maintaining the maximum ambiguous velocity associated with the FMCW waveform transmitted by the radar. By using a virtual array, you were also able to form beams in azimuth and elevation with a beamwidth of 2 degrees with a total of only 260 combined transmit and receive elements, demonstrating a significant reduction in array size over what would have been required if a full array had been used. Lastly, you used a highway overpass scenario to demonstrate the 4D radar's ability to resolve the difference in target height between targets on the same road as the ego vehicle and detections from the overpass crossing above the road being traveled.

References

[1] F. Jansen, "Automotive radar Doppler division MIMO with velocity ambiguity resolving capabilities," in 2019 16th European Radar Conference (EuRAD), Paris, France, Oct. 2019, pp. 245–248.

[2] F. Xu, S. A. Vorobyov, and F. Yang, "Transmit beamspace DDMA based automotive MIMO radar," IEEE Trans. Veh. Technol., vol. 71, no. 2, pp. 1669–1684, Feb. 2022.

Supporting Functions

```matlab
function [dopOff,Mv] = DDMAOffsets(Ntx,Nvtx)

% Ensure even number of bins to avoid a Doppler bin at fd = 0 or fd = 1
Mv = Ntx+Nvtx;
Mv = Mv+mod(Mv,2);
Nvtx = Mv-Ntx;

mdx = (1:Mv).';
dopOff = (mdx-0.5)/Mv;

% Shift modulated Doppler frequencies from -PRF/2 to PRF/2 and prevent
% separation of signals at ±PRF/2 into two parts.
dopOff = dopOff-1/2+Nvtx/Mv/2;
end

function dft = dopMatrix(fd,N)
if nargin<2
    n = 1;
else
    n = 1:N;
end
dft = exp(-1i*2*pi*fd(:).*n);
end

function y = even(x)
y = round(x/2)*2;
end

function plotDDMAOffsets(dopOffsets,Ntx)
fig = findfig('DDMA Tx Doppler offsets'); ax = polaraxes(fig);
h = helperPolarQuiver(ax,dopOffsets(1:Ntx)*2*pi,1);
hold(ax,'on');
hvx = helperPolarQuiver(ax,dopOffsets(Ntx+1:end)*2*pi,1,'k','Color',0.7*[1 1 1]);
title(ax,fig.Name);
str = strsplit(strtrim(sprintf('Tx_{%i} ',1:Ntx)));
legend([h;hvx(1)],str{:},'Tx_{virtual}');
ax.ThetaLim = [-180 180];
end

function plotTxArrayReturns(rgBins,dpBins,iRgTx,iDpTx,Ntx,prf)
ax = findaxes('Range-Doppler Map'); hold(ax,'on');
p = findobj(ax,'Type','line');
p = [p;plot(ax,dpBins(iDpTx(:))/prf,rgBins(iRgTx(:)),'k.','DisplayName','Tx elements')];
legend(p(1:2),'Unambiguous detections','Tx elements','Location','southwest','AutoUpdate','off');

for m = 1:Ntx
    text(ax,dpBins(iDpTx(m))/prf,rgBins(iRgTx(m)),sprintf('%i',m), ...
        'Color','w','FontWeight','bold','FontSize',10, ...
        'HorizontalAlignment','center','VerticalAlignment','bottom');
end
end

function fig = findfig(name)
```

```matlab
fig = findobj(groot,'Type','figure','Name',name);
if isempty(fig) || ~any(ishghandle(fig))
    fig = figure('Name',name);
    fig.Visible = 'on';
    fig.MenuBar = 'figure';
elseif any(ishghandle(fig))
    fig = fig(ishghandle(fig));
end
fig = fig(1); clf(fig); figure(fig);
end

function ax = findaxes(name)
fig = findobj(groot,'Type','figure','Name',name);
if isempty(fig) || ~any(ishghandle(fig))
    fig = figure('Name',name);
    fig.Visible = 'on';
elseif any(ishghandle(fig))
    fig = fig(ishghandle(fig));
end
fig = fig(1);
ax = fig.CurrentAxes;
if isempty(ax)
    ax = axes(fig);
end
end
```

Waveform Design and Signal Processing of Stepped Frequency Modulated Radar

This example shows how to design a stepped frequency modulated (FM) waveform, generate I/Q data for a stepped FM radar system, and process the I/Q data for range processing.

Introduction

In some radar systems, such as automotive radar, the radar is required to transceive a waveform with a large bandwidth to achieve fine range resolution, but the sample rate at the radar receiver is low. A very common and effective technique is the linear FM waveform. However, generating linear FM waveform with good linearity over a large bandwidth places high demands on the transmitter analog components.

The stepped FM waveform is an alternative technique for achieving fine range resolution with low sample rate, while avoiding analog FM modulations. A stepped FM waveform is composed of multiple single-frequency signals with frequencies linearly and equally stepped across a large bandwidth. This example demonstrates how to design, simulate and process the stepped FM waveform to satisfy design requirements. The design requirements include:

- Range requirements: achieve a maximum unambiguous range of 500 meters and a range resolution of 2 meters.
- Detection requirements: detect non-fluctuating targets with a radar cross section (RCS) greater than or equal to 1 square meter, up to the maximum unambiguous range, with a probability of detection of 0.99 and a probability of false alarm of 1e-4.

Waveform and Transceiver Definitions

Define Stepped FM Waveform

Assume that the radar operates at 1 GHz in a free space environment. Use the `helperSteppedFMParameter` helper function to derive the stepped FM waveform parameters from the specified range requirements.

```
% Physical parameters
rng('default');                                 % Set random number generator for repeatable resu
c = physconst('LightSpeed');                    % Propagation speed (m/s)
fc = 1e9;                                        % Carrier frequency (Hz)
lambda = freq2wavelen(fc,c);                    % Carrier Wavelength (m)

% Range requirements
rangeMax = 500;                                 % Maximum unambiguous range (m)
rangeRes = 2;                                   % Required range resolution (m)

% Waveform parameters
dutyCyc = 0.5;                                  % Duty cycle
[Nstep,fstep,fs,prf,pulseWidth] = helperSteppedFMParameter(rangeMax,rangeRes,dutyCyc);
```

Define the stepped FM waveform using `phased.SteppedFMWaveform` object. Configure the stepped FM waveform using the derived waveform parameters.

```
% Configure the stepped FM waveform
sfmwav = phased.SteppedFMWaveform('NumSteps',Nstep,'FrequencyStep',fstep,...
    'SampleRate',fs,'PRF',prf,'PulseWidth',pulseWidth);
```

Plot the spectrum of the first 15 steps.

```
helperViewSteppedFMSpectrum(sfmwav);
```
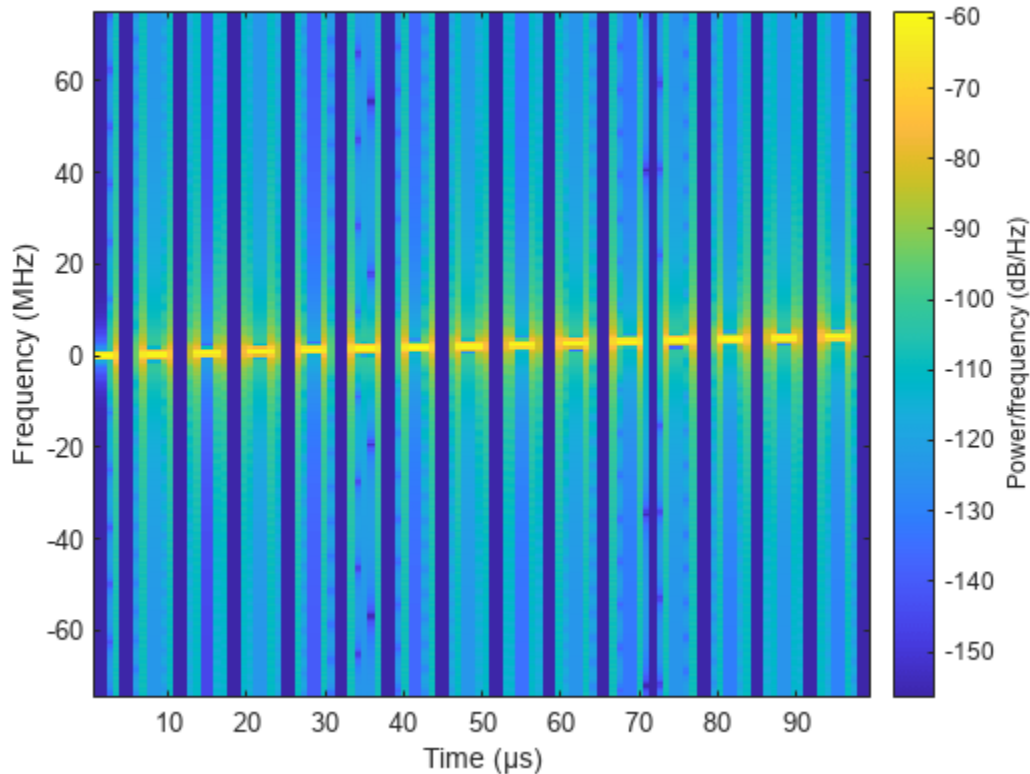

Model Radar Transceivers

The radar uses a single isotropic antenna to transmit and uses a single isotropic antenna to receive the radar signals. Determine the radar transmit peak power according to the detection requirements. For more information on the peak transmit power specification, please refer to the example "Simulating Test Signals for a Radar Receiver".

```
% Construct transmit and receive antenna element
antElmnt = phased.IsotropicAntennaElement;    % Isotropic antenna
antGain = 20;                                  % Antenna gain (dB)

% Detection requirements
pd = 0.99;                                      % Probability of detection
pfa = 1e-4;                                     % Probability of false alarm

% Determine minimum required SNR from detection requirements
snrMin = albersheim(pd,pfa);                   % Minimum required SNR (dB)

% Specify minimum target RCS
minRCS = 1;                                      % Minimum target RCS (m^2)

% Signal duration in stepped FM waveform
sigtau = Nstep*pulseWidth;                      % Signal duration in waveform (s)
```

```
% Determine radar transmit peak power from radar equation
txPkPower = radareqpow(lambda,...
    rangeMax,snrMin,sigtau,...
    'RCS',minRCS,'Gain',antGain);                % Transmit peak power (W)
```

Model the radar's monostatic transceiver using `radarTransceiver` and configure the radar to use a stepped FM waveform.

```
% Model radar monostatic transceiver
radar = radarTransceiver('Waveform',sfmwav);
radar.Transmitter = phased.Transmitter('PeakPower',txPkPower,'Gain',antGain);
radar.TransmitAntenna = phased.Radiator('Sensor',antElmnt,'OperatingFrequency',fc);
radar.ReceiveAntenna = phased.Collector('Sensor',antElmnt,'OperatingFrequency',fc);
radar.Receiver = phased.ReceiverPreamp('Gain',antGain,'SampleRate',sfmwav.SampleRate);
```

Scenario Setup

Define a scenario with a stationary radar and 3 moving targets using `radarScenario`. The scenario updates in each frequency step, and the scenario duration is the duration of a stepped FM waveform.

```
% Create a radar scenario
scene = radarScenario('UpdateRate',sfmwav.PRF,'StopTime',(sfmwav.NumSteps-1)/sfmwav.PRF);
```
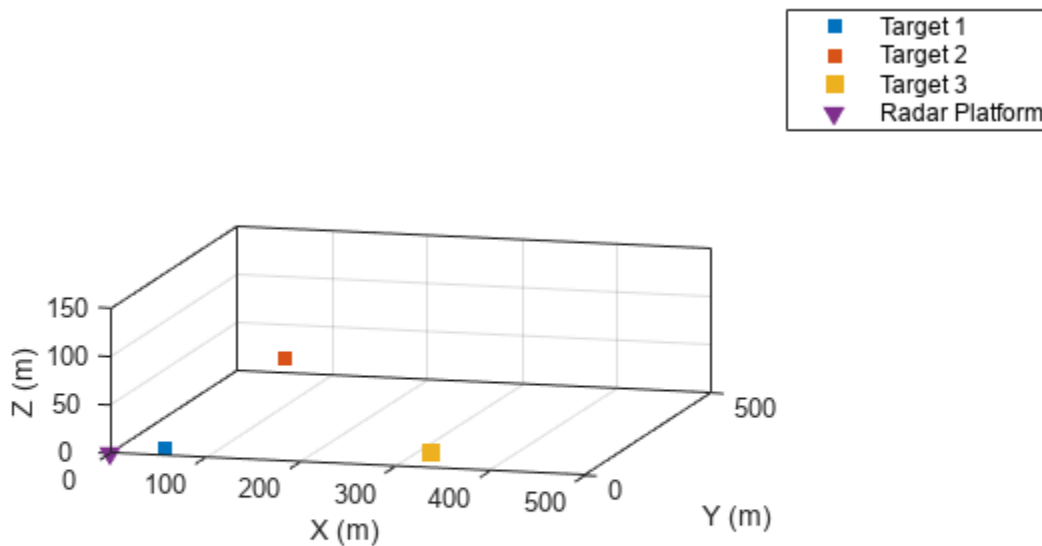
Set and visualize the positions and velocities of radar and targets below.

```
% Set positions of radar and targets
[radarPlatform,tgtPlatform] = helperPlatformSetup(radar,scene);

% Visualize positions of radar and targets
helperViewRadarScenario(radarPlatform,tgtPlatform);
```

Show the initial positions of the targets relative to the radar.

```
% Targets' initial positions
[tgtRange,tgtDOA] = helperViewInitialPosition(radarPlatform,tgtPlatform)
```

tgtRange = *1×3*

   62   214   327

tgtDOA = *2×3*

   42.0000   0   16.0000
        0   30.0000   0

Receiver Signal Processing

The receiver signal processing flow is summarized using the following diagram. First, each step of the received signal is passed to its matched filter. Next, sample one point on the matched filter output in each step with the sample interval equal to the pulse repetition interval (PRI) of the stepped FM waveform. Finally, perform fast Fourier transform (FFT) on the samples to obtain the range response and use a constant false alarm rate (CFAR) detector to detect targets' ranges.
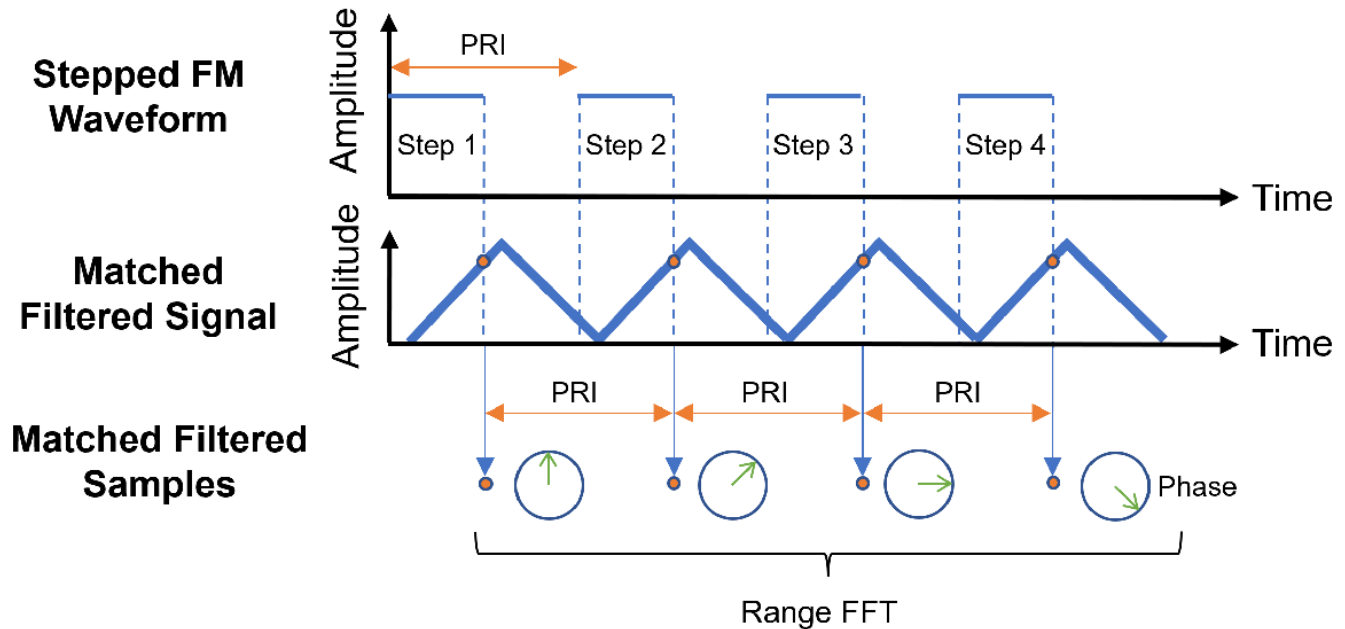
Receiving I/Q data

To receive the I/Q data, advance the radar scenario with the duration equal to the PRI using `advance` and get the received I/Q signal in each step from the radar scenario using `receive`.

```
% Number of samples in a PRI
Npri = floor(sfmwav.SampleRate/prf);

% Assemble the received I/Q data
Xrec = zeros(Npri,Nstep);

% Initialize step index
stepIdx = 1;

% Advance radar scenario for in each frequency step
while advance(scene)
    % Get the received signal
    rxSig = receive(scene);

    % Record the received signal
    Xrec(:,stepIdx) = rxSig{1};

    % Update step index
    stepIdx = stepIdx + 1;
end
```

Matched Filtering

A simple pulse matched filter is applied to process the received data in each step. Then, sample the matched filter signal in each step at the matched filter delay and the sample rate is equal to the pulse repetition frequency (PRF) of the stepped FM waveform. The matched filtered samples have significantly higher amplitudes than the received raw I/Q data.

```
% Get matched filter coefficient for the stepped FM waveform
mfCoeff = getMatchedFilter(sfmwav);

% Assemble the matched filtered data
Xmf = zeros(Npri,Nstep);

% Match filtering for each step
for stepIdx = 1:Nstep
    % Define matched filter for each step
    mf = phased.MatchedFilter('Coefficients',mfCoeff(:,stepIdx));

    % Perform matched filtering
    Xmf(:,stepIdx) = mf(Xrec(:,stepIdx));
end

% Get matched filter delay
mfDelay = size(mfCoeff,1);

% Sample in each frequency step at the matched filter delay
XmfSample = reshape(Xmf(mfDelay,:),Nstep,[]);
```

Range Processing

The linear increase of the frequency in different steps introduces linear phase shifts on the matched filter samples for each target. The linear phase shift change speed of a target is linearly proportional to the target's range. This allows the stepped FM radar to measure the ranges of the targets via measuring the phase shifts, which can be efficiently implemented using range FFT.

Apply range FFT on the sampled data using `phased.RangeResponse`. As the sampled data are sampled with intervals equal to the PRI of the stepped FM waveform, the sample rate configured in `phased.RangeResponse` is the PRF of the stepped FM waveform. The sweep slope of the stepped FM waveform is defined and configured as the step frequency divided by the PRI.

```
% Number of range samples
Nrange = 2^nextpow2(Nstep);

% Define range response
rngresp = phased.RangeResponse('RangeMethod','FFT',...
    'RangeFFTLengthSource','Property','RangeFFTLength',Nrange,...
    'SampleRate',sfmwav.PRF,...
    'SweepSlope',sfmwav.FrequencyStep*sfmwav.PRF,...
    'ReferenceRangeCentered',false);

% Calculate the range response of the sampled I/Q data
[Xrng,rngGrid] = rngresp(conj(XmfSample));
```

CFAR Detection

Apply the CFAR detector on the range response for detecting the ranges of the targets.

```
% CFAR detector setup
cfarDetector = phased.CFARDetector('NumGuardCells',6,...
    'NumTrainingCells',10,'ProbabilityFalseAlarm',pfa,...
    'ThresholdOutputPort',true);

% Range spectrum
rngSpectrum = abs(Xrng).^2;
```

```
% CFAR detection on range spectrum
[cfarDetectResult,cfarDetectThresh] = cfarDetector(rngSpectrum,1:Nrange);

% Plot range response pattern and CFAR detection result
helperViewDetectionResult(rngSpectrum,rngGrid,cfarDetectResult,cfarDetectThresh);
```
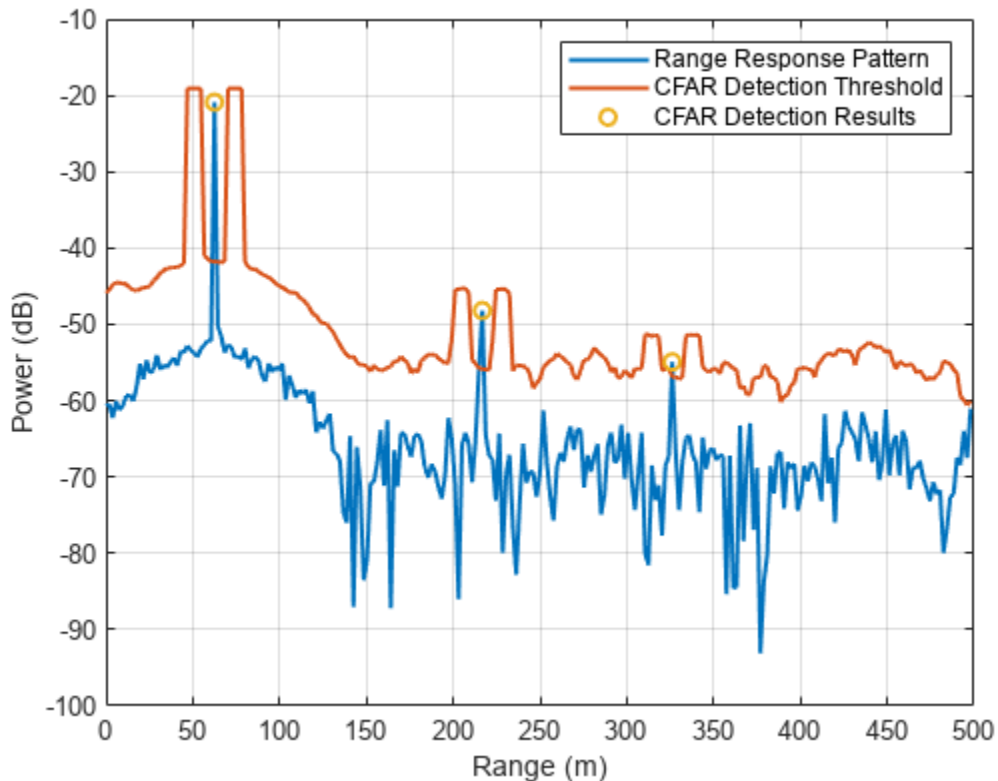

From the CFAR detection result, we can clearly observe 3 targets. The detected ranges are displayed below.

```
rngDetected = rngGrid(cfarDetectResult)'
```

rngDetected = *1×3*

    62.5000   216.7969   326.1719

Summary

In this example, you learned how to design a stepped FM waveform to meet desired performance requirements. You learned how to generate and process the stepped FM I/Q data to obtain range response.

Reference

[1] M. A. Richards. *Fundamentals of Radar Signal Processing,* Second., New York: McGraw Hill, 2014, pp. 188–194.

[2] M. A. Richards, W. L. Melvin, J. A. Scheer, J. Scheer. *Principles of Modern Radar: Advanced Techniques, Volume 2*. Institution of Engineering and Technology, 2012, pp. 58-69.

[3] N. Levanon. "Stepped-frequency Pulse-train Radar Signal," in IEE Proceedings-Radar Sonar and Navigation 149, no. 6, 2002.

[4] D. A. Noon. "Stepped-frequency Radar Design and Signal Processing Enhances Ground Penetrating Radar Performance," 1996.

Helper Functions

helperSteppedFMParameter Function

```
function [nstep,fstep,fs,prf,pulseWidth] = helperSteppedFMParameter(rangeMax,rangeRes,dutyCyc)
c = physconst('LightSpeed');                    % Propagation speed (m/s)

% Determine step frequency from the required maximum range
fstep = rangeres2bw(rangeMax,c);                % Step frequency (Hz)

% Determine the waveform bandwidth from the required range resolution
bw = rangeres2bw(rangeRes,c);                   % Bandwidth (Hz)

% Determine RF sample rate from bandwidth
fs = 2*bw;                                       % RF sample rate (Hz)

% Determine number of frequency steps in the stepped FM waveform
nstep = round(bw/fstep);                         % Number of frequency steps

% Determine pulse width
pulseWidth = 1/fstep;                            % Pulse width (s)

% Determine fast-time PRF
prf = dutyCyc/pulseWidth;                        % Fast-time PRF (Hz)
end
```

helperPlatformSetup Function

```
function [radarPlatform,tgtPlatform] = helperPlatformSetup(radar,scene)
% Initialize radar position and velocity
radarPos = [0; 0; 0];
radarVel = [0; 0; 0];

% Define radar platform
radarTrajectory = kinematicTrajectory('Position',radarPos,'Velocity',radarVel);
radarPlatform = platform(scene,'Trajectory',radarTrajectory,'Sensors',radar);

% Initialize targets' positions and velocities
ntgt = 3;                                        % Number of targets
tgtRcs = [2.2 1.6 2.8];                          % Targets' RCSs (m^2)
tgtRng = [62 214 327];                           % Targets' ranges (m)
tgtAz  = [42 0 16];                              % Targets' azimuth angle (deg)
tgtEl  = [0 30 0];                               % Targets' elevation angle (deg)
[tgtx, tgtY, tgtZ] = sph2cart(deg2rad(tgtAz),deg2rad(tgtEl),tgtRng);
tgtPos = [tgtx; tgtY; tgtZ];
tgtVel = [30 120 -60;0 10 0;0 0 0];

% Define target platform
tgtPlatform = cell(ntgt, 1);
```

```matlab
fc = radar.TransmitAntenna.OperatingFrequency;
for tgtIdx = 1:ntgt
    tgtTrajectory = kinematicTrajectory('Position',tgtPos(:,tgtIdx),'Velocity', tgtVel(:,tgtIdx)
    tgtRcsSignature = rcsSignature('FluctuationModel','Swerling0',...
        'Pattern',[tgtRcs(tgtIdx),tgtRcs(tgtIdx)],...
        'Azimuth',[-180 180],'Elevation',[-90 90],'Frequency',[fc fc]);
    tgtPlatform{tgtIdx} = platform(scene, 'Trajectory', tgtTrajectory, ...
        'Signature', tgtRcsSignature);
end
end
```

# Simulink Examples

Using the Radar Data Generator Block

Open and run the Radar Data Generator block in Simulink™.

```
% Create a Tracking scenario object.
scene = trackingScenario;

% Create a stationary radar platform 10 meters above the origin.
rdrPlat = platform(scene,'Position',[0 0 10]);

% Create a moving target 100 meters away and also 10 meters
% above the surface. The target moves at 10 m/s away from the radar.
tgtPlat = platform(scene,'Position',[100 0 10]);
tgtPlat.Trajectory.Velocity = [10 0 0];

% Open and run the Simulink model.
% Display the detections.
model = 'simpleModel';
open_system(model);
sim(model);
```
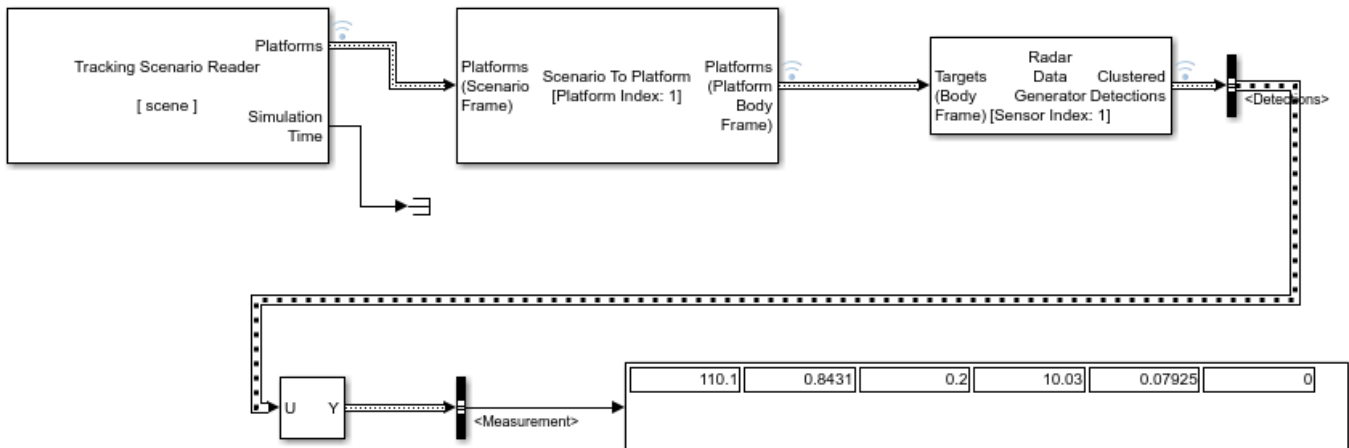
Clutter

Clutter Modeling

In this section...
"Surface Clutter Overview" on page 3-2
"Approaches for Clutter Simulation or Analysis" on page 3-2
"Considerations for Setting Up a Constant Gamma Clutter Simulation" on page 3-2
"Related Examples" on page 3-4

Surface Clutter Overview

Surface clutter refers to reflections of a radar signal from land, sea, or the land-sea interface. When trying to detect or track targets moving on or above the surface, you must be able to distinguish between clutter and the targets of interest. For example, a ground moving target indicator (GMTI) radar application should detect targets on the ground while accounting for radar reflections from trees or houses.

If you are simulating a radar system, you might want to incorporate surface clutter into the simulation to ensure the system can overcome the effects of surface clutter. If you are analyzing the statistical performance of a radar system, you might want to incorporate clutter return distributions into the analysis.

Approaches for Clutter Simulation or Analysis

Radar Toolbox software offers these tools to help you incorporate surface clutter into your simulation or analysis:

* `constantGammaClutter`, a System object™ that simulates clutter returns using the constant gamma model
* Utility functions to help you implement your own clutter models:

  * `billingsleyicm`
  * `depressionang`
  * `effearthradius`
  * `grazingang`
  * `horizonrange`
  * `surfclutterrcs`
  * `surfacegamma`

Considerations for Setting Up a Constant Gamma Clutter Simulation

When you use `constantGammaClutter`, you must configure the object for the situation you are simulating, and confirm that the assumptions the software makes are valid for your system.

Physical Configuration Properties

The `constantGammaClutter` object has properties that correspond to physical aspects of the situation you are modeling. These properties include:

- Propagation speed, sample rate, and pulse repetition frequency of the signal
- Operating frequency of the system
- Altitude, speed, and direction of the radar platform
- Depression angle of the broadside of the radar antenna array

Clutter-Related Properties

The object has properties that correspond to the clutter characteristics, location, and modeling fidelity. These properties include:

- Gamma parameter that depends on the terrain type and system's operating frequency.
- Azimuth coverage and maximum range for the clutter simulation.
- Azimuth span of each clutter patch. The software internally divides the clutter ring into a series of adjacent, non-overlapping clutter patches.
- Clutter coherence time. This value indicates how frequently the software changes the set of random numbers in the clutter simulation.

    In the simulation, you can use identical random numbers over a time interval or uncorrelated random numbers. Simulation behavior slightly differs from reality, where a moving platform produces clutter returns that are correlated with each other over small time intervals.

Working with Samples or Pulses

The `constantGammaClutter` object has properties that let you obtain results in a convenient format. Using the `OutputFormat` property, you can choose to have the `step` method produce a signal that represents:

- A fixed number of pulses. You indicate the number of pulses using the `NumPulses` property of the object.
- A fixed number of samples. You indicate the number of samples using the `NumSamples` property of the object. Typically, you use the number of samples in one pulse. In staggered PRF applications, you might find this option more convenient because the `step` output always has the same matrix size.

Assumptions

The clutter simulation that `constantGammaClutter` provides is based on these assumptions:

- The radar system is monostatic.
- The propagation is in free space.
- The terrain is homogeneous.
- The clutter patch is stationary during the coherence time. Coherence time indicates how frequently the software changes the set of random numbers in the clutter simulation.
- Because the signal is narrowband, the spatial response and Doppler shift can be approximated by phase shifts.
- The radar system maintains a constant height during simulation.
- The radar system maintains a constant speed during simulation.

Related Examples

- "Ground Clutter Mitigation with Moving Target Indication (MTI) Radar" on page 1-525
- "Introduction to Space-Time Adaptive Processing"
- "DPCA Pulse Canceller to Reject Clutter"
- "Adaptive DPCA Pulse Canceller To Reject Clutter and Interference"
- "Sample Matrix Inversion Beamformer"

Interference

Barrage Jammer

In this section...
"Support for Modeling Barrage Jammer" on page 4-2
"Model Barrage Jammer Output" on page 4-2
"Model Effect of Barrage Jammer on Target Echo" on page 4-3

Support for Modeling Barrage Jammer

The `barrageJammer` object models a broadband jammer. The output of `barrageJammer` is a complex white Gaussian noise sequence. The modifiable properties of the barrage jammer are:

- `ERP` — Effective radiated power in watts
- `SamplesPerFrameSource` — Source of number of samples per frame
- `SamplesPerFrame` — Number of samples per frame
- `SeedSource` — Source of seed for random number generator
- `Seed` — Seed for random number generator

The real and imaginary parts of the complex white Gaussian noise sequence each have variance equal to 1/2 the effective radiated power in watts. Denote the effective radiated power in watts by *P*. The barrage jammer output is:

$$w[n] = \sqrt{\frac{P}{2}}x[n] + j\sqrt{\frac{P}{2}}y[n]$$

In this equation, `x[n]` and `y[n]` are uncorrelated sequences of zero-mean Gaussian random variables with unit variance.

Model Barrage Jammer Output

This example examines the statistical properties of the barrage jammer output and how they relate to the effective radiated power *(ERP)*. Create a barrage jammer using an effective radiated power of 5000 watts. Generate output at 500 samples per frame. Then call the `step` function once to generate a single frame of complex data. Using the `histogram` function, show the distribution of barrage jammer output values. The `BarrageJammer` System object uses a random number generator. In this example, the random number generator seed is fixed for illustrative purposes and can be removed.

```
rng default
jammer = barrageJammer('ERP',5000,...
    'SamplesPerFrame',500);
y = jammer();
subplot(2,1,1)
histogram(real(y))
title('Histogram of Real Part')
subplot(2,1,2)
histogram(imag(y))
title('Histogram of Imaginary Part')
xlabel('Watts')
```
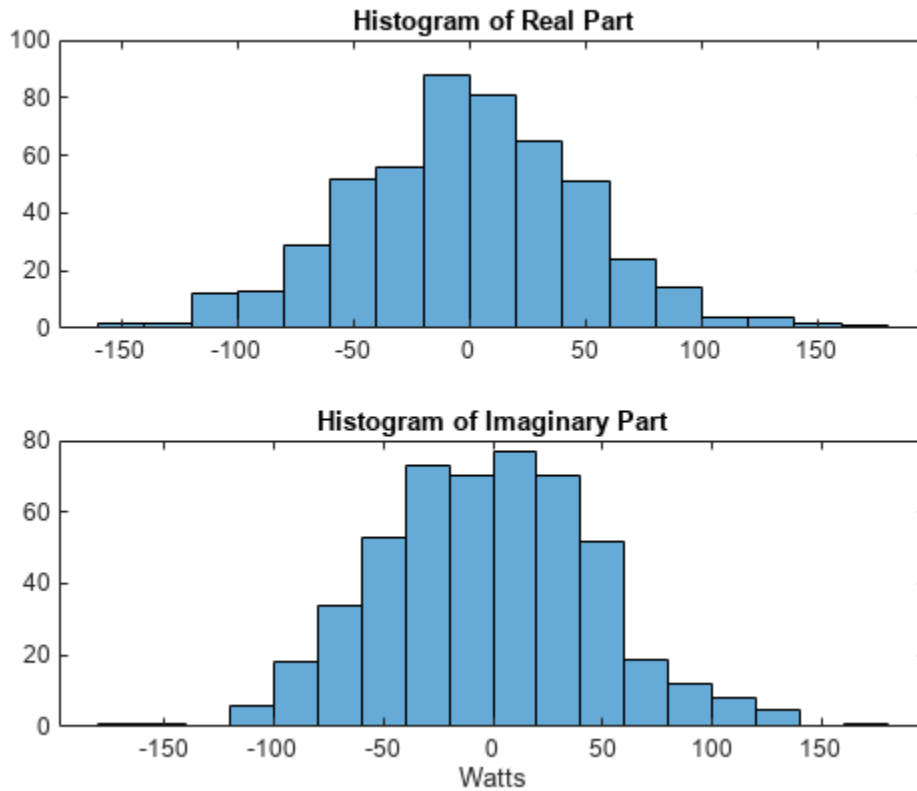
The mean values of the real and imaginary parts are

```
mean(real(y))
```

ans = -1.0961

```
mean(imag(y))
```

ans = -2.1671

which are effectively zero. The standard deviations of the real and imaginary parts are

```
std(real(y))
```

ans = 50.1950

```
std(imag(y))
```

ans = 49.7448

which agree with the predicted value of $\sqrt{ERP/2}$.

Model Effect of Barrage Jammer on Target Echo

This example demonstrates how to simulate the effect of a barrage jammer on a target echo. First, create the required objects. You need an array, a transmitter, a radiator, a target, a jammer, a

collector, and a receiver. Additionally, you need to define two propagation paths: one from the array to the target and back, and the other path from the jammer to the array.

```
antenna = phased.ULA(4);
Fs = 1e6;
fc = 1e9;
rng('default')
waveform = phased.RectangularWaveform('PulseWidth',100e-6,...
    'PRF',1e3,'NumPulses',5,'SampleRate',Fs);
transmitter = phased.Transmitter('PeakPower',1e4,'Gain',20,...
    'InUseOutputPort',true);
radiator = phased.Radiator('Sensor',antenna,'OperatingFrequency',fc);
jammer = barrageJammer('ERP',1000,...
    'SamplesPerFrame',waveform.NumPulses*waveform.SampleRate/waveform.PRF);
target = phased.RadarTarget('Model','Nonfluctuating',...
    'MeanRCS',1,'OperatingFrequency',fc);
targetchannel = phased.FreeSpace('TwoWayPropagation',true,...
    'SampleRate',Fs,'OperatingFrequency', fc);
jammerchannel = phased.FreeSpace('TwoWayPropagation',false,...
    'SampleRate',Fs,'OperatingFrequency', fc);
collector = phased.Collector('Sensor',antenna,...
    'OperatingFrequency',fc);
amplifier = phased.ReceiverPreamp('EnableInputPort',true);
```

Assume that the array, target, and jammer are stationary. The array is located at the global origin, *(0,0,0)*. The target is located at *(1000,500,0)*, and the jammer is located at *(2000,2000,100)*. Determine the directions from the array to the target and jammer.

```
targetloc = [1000 ; 500; 0];
jammerloc = [2000; 2000; 100];
[~,tgtang] = rangeangle(targetloc);
[~,jamang] = rangeangle(jammerloc);
```

Finally, transmit the rectangular pulse waveform to the target, reflect it off the target, and collect the echo at the array. Simultaneously, the jammer transmits a jamming signal toward the array. The jamming signal and echo are mixed at the receiver. Generate waveform

```
wav = waveform();
% Transmit waveform
[wav,txstatus] = transmitter(wav);
% Radiate pulse toward the target
wav = radiator(wav,tgtang);
% Propagate pulse toward the target
wav = targetchannel(wav,[0;0;0],targetloc,[0;0;0],[0;0;0]);
% Reflect it off the target
wav = target(wav);
% Collect the echo
wav = collector(wav,tgtang);
```

Generate the jamming signal

```
jamsig = jammer();
% Propagate the jamming signal to the array
jamsig = jammerchannel(jamsig,jammerloc,[0;0;0],[0;0;0],[0;0;0]);
% Collect the jamming signal
jamsig = collector(jamsig,jamang);

% Receive target echo alone and target echo + jamming signal
```

```
pulsewave = amplifier(wav,~txstatus);
pulsewave_jamsig = amplifier(wav + jamsig,~txstatus);
```

Plot the result, and compare it with received waveform with and without jamming.

```
subplot(2,1,1)
t = unigrid(0,1/Fs,size(pulsewave,1)*1/Fs,'[)');
plot(t*1000,abs(pulsewave(:,1)))
title('Magnitudes of Pulse Waveform Without Jamming--Element 1')
ylabel('Magnitude')
subplot(2,1,2)
plot(t*1000,abs(pulsewave_jamsig(:,1)))
title('Magnitudes of Pulse Waveform with Jamming--Element 1')
xlabel('millisec')
ylabel('Magnitude')
```
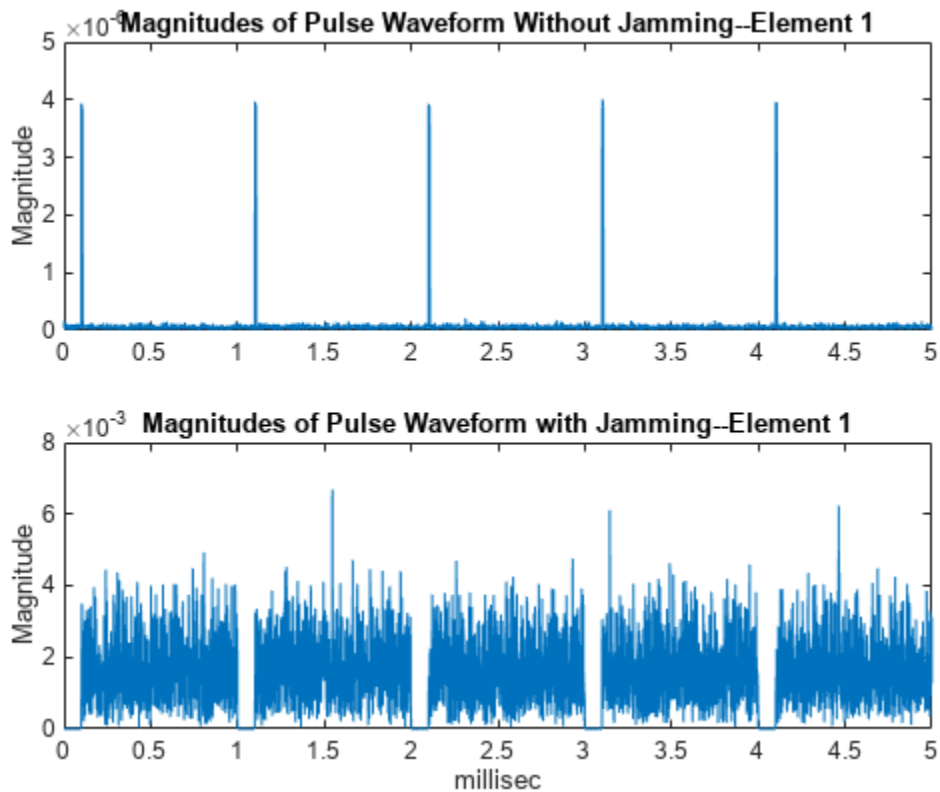
Radar Equation

Radar Equation

In this section...
"Radar Equation Theory" on page 5-2
"Plot Vertical Coverage Pattern Using Default Parameters" on page 5-3
"Compute Peak Power Using Radar Equation Calculator App" on page 5-4

Radar Equation Theory

The point target radar range equation estimates the power at the input to the receiver for a target of a given radar cross section at a specified range. In this equation, the signal model is assumed to be deterministic. The equation for the power at the input to the receiver is

$$P_r = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R_t^2 R_r^2 L},$$

where the terms in the equation are:

- P_r — Received power in watts.
- P_t — Peak transmit power in watts.
- G_t — Transmitter gain.
- G_r — Receiver gain.
- λ — Radar operating frequency wavelength in meters.
- σ — Target's nonfluctuating radar cross section in square meters.
- L — General loss factor to account for both system and propagation loss.
- R_t — Range from the transmitter to the target.
- R_r — Range from the receiver to the target. If the radar is monostatic, the transmitter and receiver ranges are identical.

The equation for the power at the input to the receiver represents the signal term in the signal-to-noise ratio (SNR). To model the noise term, assume the thermal noise in the receiver has a white noise power spectral density (PSD) given by

$$P(f) = kT,$$

where k is the Boltzmann constant and T is the effective noise temperature. The receiver acts as a filter to shape the white noise PSD. Assume that the magnitude squared receiver frequency response approximates a rectangular filter with bandwidth equal to the reciprocal of the pulse duration, $1/\tau$. The total noise power at the output of the receiver is

$$N = \frac{kTF_n}{\tau},$$

where F_n is the receiver *noise figure*.

The product of the effective noise temperature and the receiver noise factor is referred to as the *system temperature* and is denoted by T_s, so that $T_s = TF_n$.

Using the equation for the received signal power and the output noise power, the receiver output SNR is

$$\frac{P_r}{N} = \frac{P_t \tau G_t G_r \lambda^2 \sigma}{(4\pi)^3 k T_s R_t^2 R_r^2 L}.$$

Solving for the required peak transmit power,

$$P_t = \frac{P_r (4\pi)^3 k T_s R_t^2 R_r^2 L}{N \tau G_t G_r \lambda^2 \sigma}.$$

## Plot Vertical Coverage Pattern Using Default Parameters

Set the frequency to 100 MHz, the antenna height to 10 m, and the free-space range to 200 km. The antenna pattern, surface roughness, antenna tilt angle, and field polarization assume their default values as specified in the `AntennaPattern`, `SurfaceRoughness`, `TiltAngle`, and `Polarization` properties.

Obtain an array of vertical coverage pattern values and angles.

```
freq = 100e6;
ant_height = 10;
rng_fs = 200;
[vcp,vcpangles] = radarvcd(freq,rng_fs,ant_height);
```

To see the vertical coverage pattern, omit the output arguments.

```
radarvcd(freq,rng_fs,ant_height);
```

Blake Chart
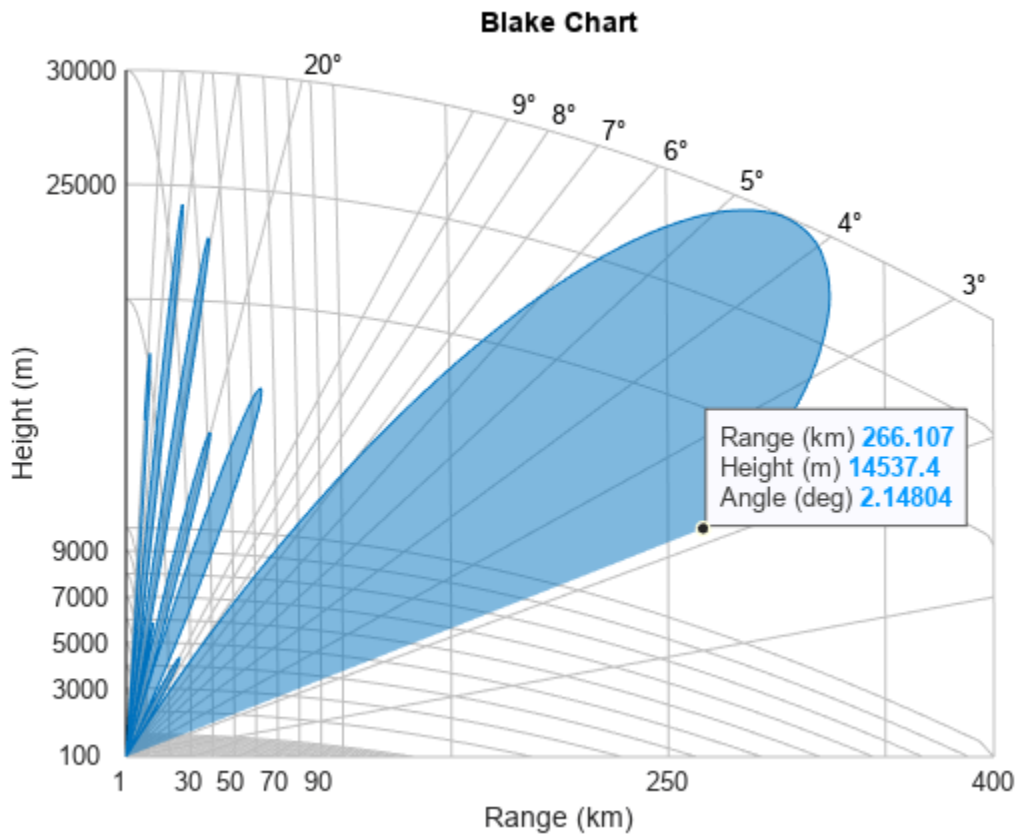
Compute Peak Power Using Radar Equation Calculator App

The `radarEquationCalculator` app lets you determine key radar characteristics such as detection range, required peak transmit power, and SNR. The App works for monostatic and bistatic radars.

Open radarEquationCalculator App

When you type `radarEquationCalculator` from the command line or select the app from the **App Toolstrip**, an interactive window opens. The default window shows a calculation of target range from SNR, power, and other parameters. You can then select various options to compute different radar parameters.

```
radarEquationCalculator
```

Compute Required Peak Transmit Power of Monostatic Radar

As an example, use the app to compute the required peak transmit power for a monostatic radar to detect a large target at 100 km. The radar operates at 10 GHz with a 40 dB antenna gain. Set the probability of detection to 0.9 and the probability of false alarm to 0.0001.

1 From the **Calculation Type** drop-down list, choose `Peak Transmit Power`

2 Set the **Wavelength** to 3 cm

3 Specify the **Pulse Width** as 2 microseconds

4 Assume total **System Losses** of 5 dB

5 Assuming the target is a large airplane, set **Target Radar Cross Section** value to $100 \ \text{m}^2$

6 Choose **Configuration** as `Monostatic`

7 Set the **Gain** to be 40 dB

    8   Open the **SNR** box

    9   Specify the **Probability of Detections** as `0.9`

   10   Specify the **Probability of False Alarm** as `0.0001`

The app shows that the required peak transmit power is 0.2095 W.

Measurement Accuracy and Resolution

Measurement Accuracy, Bias, and Resolution

Definition of Terms

Measurement *accuracy* and *resolution* are related concepts that are often confused with one another. Accuracy expresses how close a measurement is to its true value. Resolution is the granularity, or fineness, of a measurement. One description of resolution expresses is how close two objects can be before they can no longer be distinguished as different objects. Resolution used in this sense is sometimes referred to as *Rayleigh resolution*. Resolution need not refer to two different objects. You can use it to describe errors in how large an object is or how far has an object has moved before its motion can be detected. Accuracy and resolution may apply to radar measurements such as range, azimuth and elevation, and range rate.

Resolution

Resolution is the ability to distinguish between two objects. For example, range resolution is the ability of radar system to distinguish between two or more targets on the same bearing but at different ranges. The range resolution depends on the width of the transmitted pulse. A radar system should be able to distinguish targets separated by one-half the pulse width.

You can think of resolution as an instrumentation limitation such as signal bandwidth or antenna aperture size. Each measurement has a characteristic quantity that determines that limitation. It is further assumed that the measurement error associated with a particular parameter is independent of the errors in any of the other parameters, the accuracy is limited only by receiver noise, and that all bias errors are accounted for separately. This table lists the defining characteristic for each measured quantity.

Radar Measurement	Characteristic Property	Resolution (Δ)
Range	Bandwidth	1/BW
Angle	Antenna or array aperture	λ/D
Speed (Doppler)	Coherent integration time	λ/T

For range measurements, range resolution is inversely dependent on the signal bandwidth *BW*. Larger bandwidth provides greater range resolution. Range-rate resolution depends is inversely proportional to the coherent integration time of *N* pulses. Azimuth and elevation angle resolution is inversely proportional to the antenna or array aperture.

- The range resolution Δ_R is the minimum range between two targets the can be resolved. Range resolution depends on the signal bandwidth *BW*.

$$\Delta R = \frac{c}{2 \cdot BW}$$

  where *c* is the speed of light and *BW* is the signal bandwidth. The `RangeResolution` property of the `radarDataGenerator` object lets you specify the range resolution.

- Azimuth resolution Δ_azim is primarily determined by antenna or array aperture size, signal frequency, and array tapering. The `AzimuthResolution` property of the `radarDataGenerator` object defines the minimum separation in azimuth angle at which the radar can distinguish between two targets. The azimuth resolution is typically the half-power beamwidth of the azimuth angle beamwidth of the radar. The same considerations apply to elevation resolution Δ_elev.

Elevation resolution Δ_{elev} is also determined by antenna or array aperture, frequency, and array tapering. The `ElevationResolution` property defines the minimum separation in elevation angle at which a radar can distinguish between two targets. The elevation resolution is typically the half-power beamwidth of the elevation angle beamwidth of the radar.

- Range rate resolution Δv defines the minimum separation in range rate at which the radar can distinguish between two targets.

$$\delta v = \lambda \cdot PRF/(2N)$$

where Δv is the range rate resolution, PRF is the pulse repetition frequency, and N is the number of pulses. The `RangeRateResolution` property defines the minimum separation in range rate at which the radar can distinguish between two moving targets.
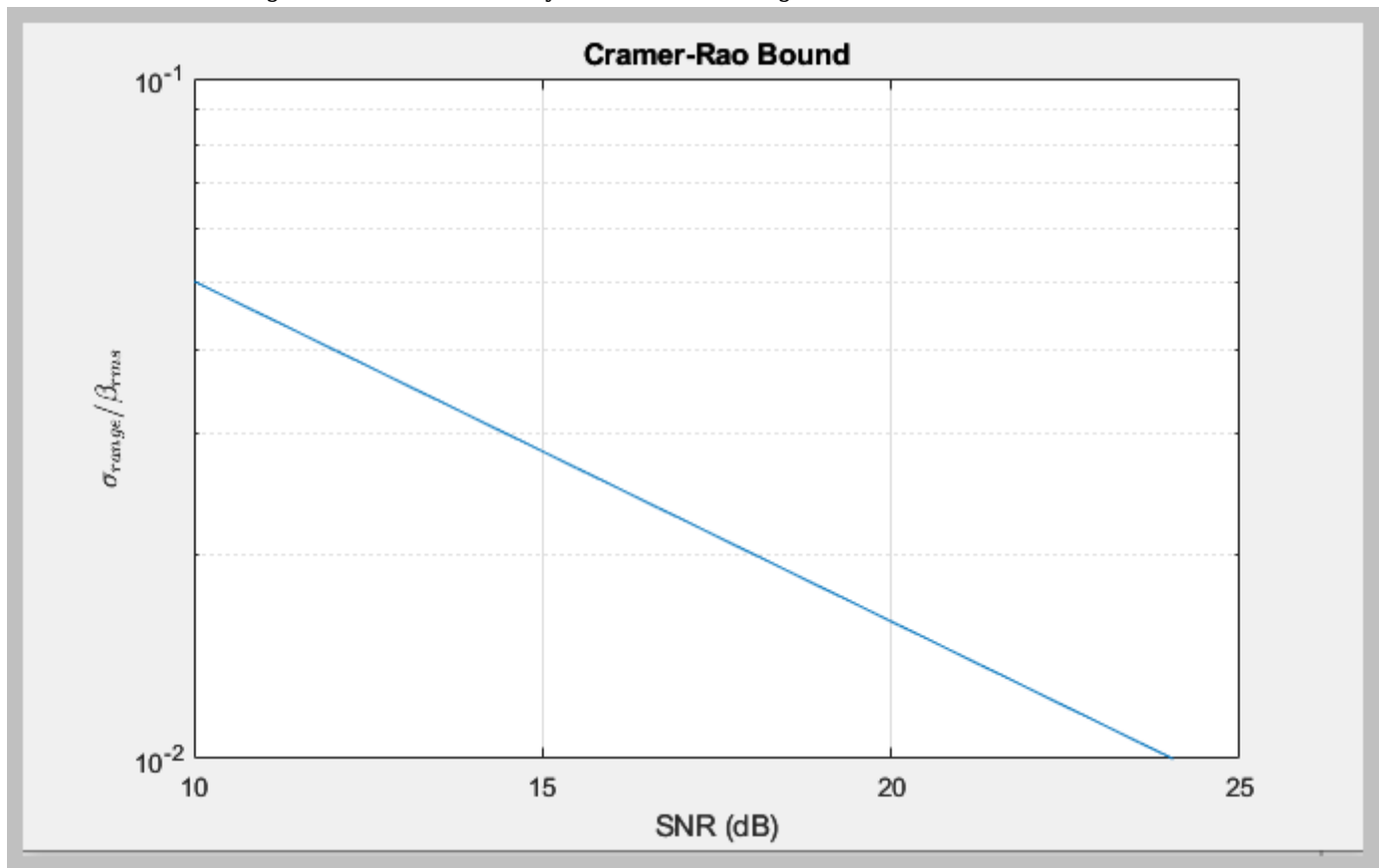
Accuracy

The accuracy σ_{meas} of a radar measurement states how precisely the measurement can be made. Accuracy is calculated as the root-mean square value (rms) of the difference between an estimated value of a quantity and its true value. Generally, accuracy is inversely proportional to the square-root of the signal-to-noise ratio and therefore improves with SNR. Accuracy is also a function of resolution. A general relation between resolution and accuracy is

$$\sigma_{meas} = \frac{k\Delta}{\sqrt{2\chi_{meas}}}$$

where χ_{meas} is the received signal-to-noise ratio (in linear units) and k is a constant with a value close to unity. A measurement may also have a bias which is a constant offset to its measured value.

The figure below shows the normalized accuracy of a detected object's measurement plotted against received signal SNR. The accuracy is calculated using the Cramer-Rao bound.

The following list shows the general formulas for estimation accuracy of radar parameters

- Range estimation accuracy is:

$$\sigma_R = \frac{1}{BW\sqrt{\chi}}\frac{c}{2\pi}\sqrt{\frac{3}{2}}$$

- Range-rate estimation accuracy is

$$\sigma_V = \frac{1}{N \cdot PRI\sqrt{\chi}}\frac{\lambda}{4\pi}\sqrt{6}$$

  where N is the number pulses, λ is the wavelength, and PRI is the pulse repetition interval.

- Azimuth and elevation angle estimation accuracies have the same form

$$\sigma_\varphi = \frac{1}{\sqrt{M \cdot \chi}}\frac{\varphi}{2\pi k}\sqrt{6}$$

The accuracy improvement factor k_{meas} represents how much the accuracy of the measurements is improved over the resolution. The equation here expresses the accuracy in terms of the k_{meas}, SNR, and measurement resolution Δ_{meas}.

$$\sigma_{meas} = \frac{k\Delta_{meas}}{\sqrt{2\chi_{meas}}} = f_{meas}\Delta_{meas}$$

This table gives typical accuracy improvement factors for range, range rate, and angle measurements:

Measurement type	Improvement factor k_{meas}
range (derived for LFM chirp)	sqrt(3)/π ≅ 0.5513
range rate	sqrt(3)/π ≅ 0.5513
angle (azimuth or elevation)	0.628

This figure shows the accuracy improvement factor using parameters specified in the table above for range, range rate, and angle measurements as a functions of SNR. Without biases, the accuracy approaches zero as the SNR increases.
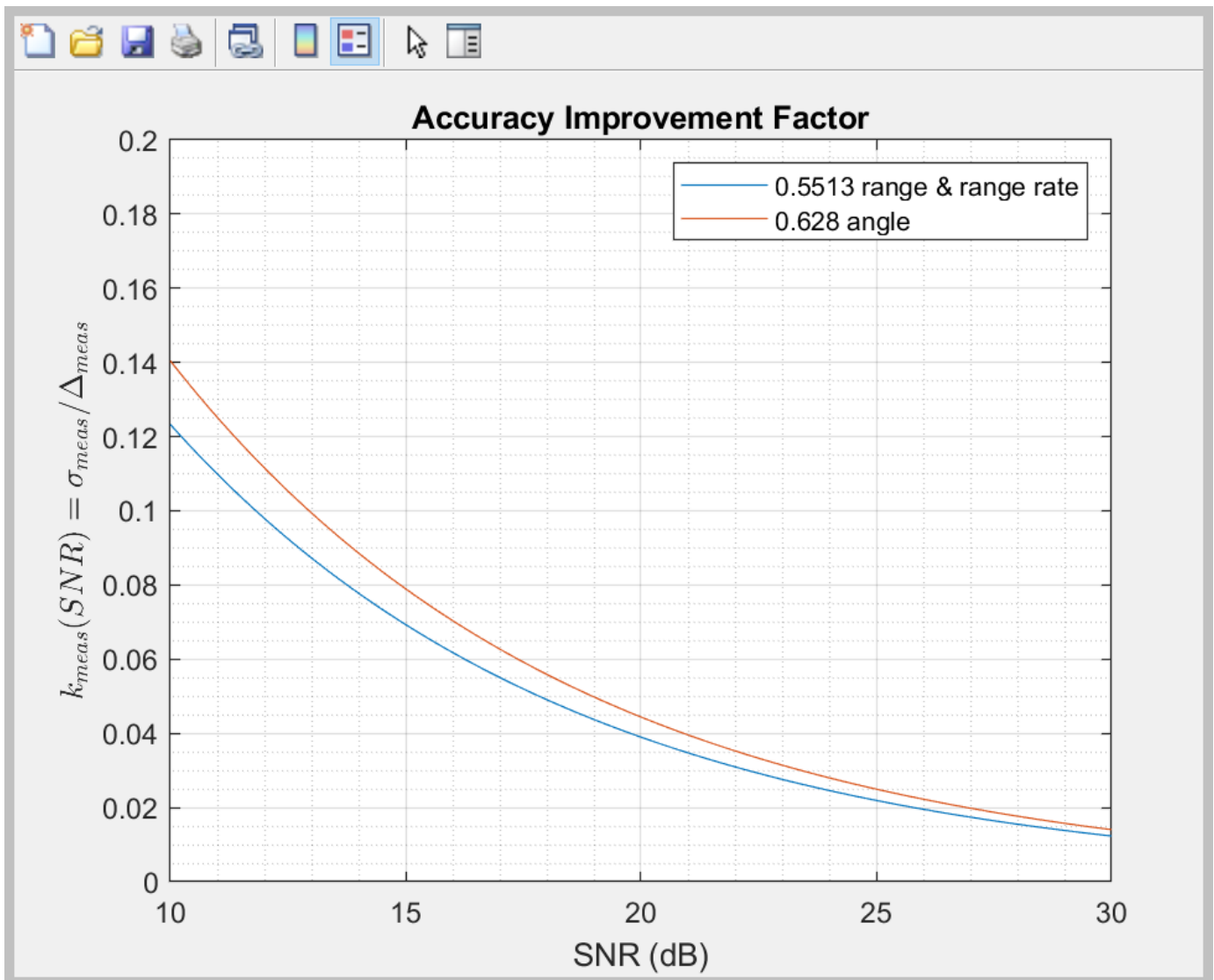
Bias

Additional terms may be added to the accuracies to account for biases in the estimate of radar parameter. Biases place a lower limit on the estimated accuracies. They may be due to, for example, instrumentation or timing errors.

- Range bias, b_R, can be specified as a fraction of the range resolution `RangeResolution` using the `RangeBiasFraction` property.

$$\sigma_R = \sqrt{\frac{3c^2}{8\pi^2 \cdot \chi \cdot BW^2} + b_R^2}$$

- The range rate bias, b_v, can be expressed as a fraction of the range-rate resolution `RangeRateResolution` using the `RangeRateBiasFraction`.

$$\sigma_V = \sqrt{\frac{6\lambda^2}{(N \cdot PRI)^2 \cdot \chi \cdot 16\pi^2} + b_V^2}$$

- The `AzimuthBiasFraction` property is the azimuth error bias is expressed as a fraction of the azimuth resolution `AzimuthResolution` property. This sets a lower bound on the azimuth accuracy of the radar. The `ElevationBiasFraction` property is the elevation error bias expressed as a fraction of the elevation resolution `ElevationResolution` property. This sets a lower bound on the elevation accuracy of the radar.

$$\sigma_\varphi = \sqrt{\frac{6\varphi^2}{4\pi^2 \cdot \chi \cdot Mk^2} + b_\varphi^2}$$

This figure shows the accuracy improvement factor using parameters specified in the table above for the range, range rate, and angle measurements as a functions of SNR when there are biases present in the measurements.
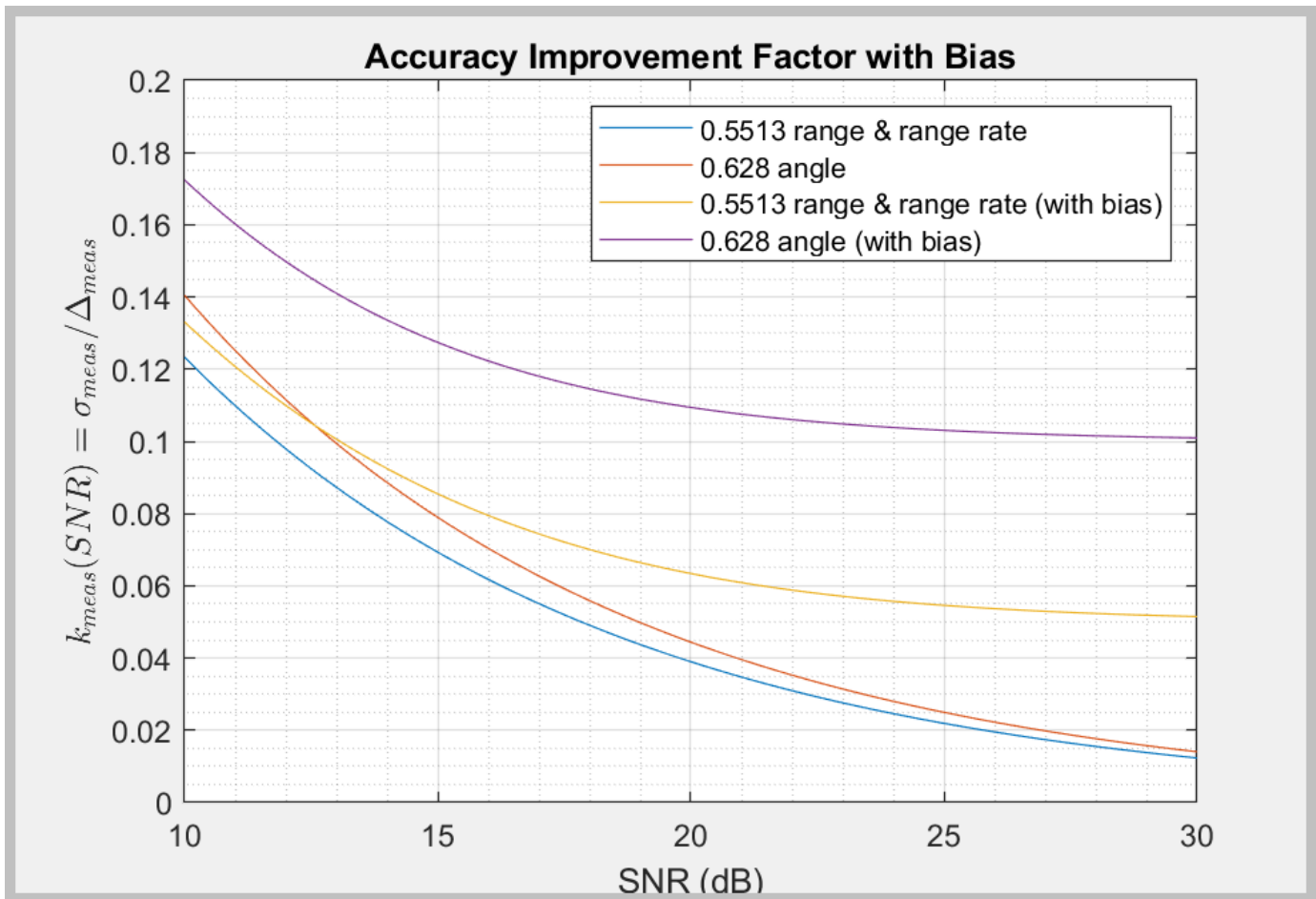
With biases present, the accuracy never approaches zero as the SNR increases.

Model Platform Motion Using Trajectory Objects

Model Platform Motion Using Trajectory Objects

This topic introduces how to use three different trajectory objects to model platform trajectories, and how to choose between them.

Introduction

Radar Toolbox provides three System objects that you can use to model trajectories of platforms including ground vehicles, ships, aircraft, and spacecraft. You can choose between these trajectory objects based on the available trajectory information and the distance span of the trajectory.

- `waypointTrajectory` — Defines a trajectory using a few waypoints in Cartesian coordinates that the trajectory must pass through. The trajectory assumes the reference frame is a fixed North-East-Down (NED) or East-North-Up (ENU) frame. Since the trajectory interpolation assumes that the gravitational acceleration expressed in the trajectory reference frame is constant, `waypointTrajectory` is typically used for a trajectory defined within an area that spans only tens or hundreds of kilometers.

- `geoTrajectory` — Defines a trajectory using a few waypoints in geodetic coordinates (latitude, longitude, and altitude) that the trajectory must pass through. Since the waypoints are expressed in geodetic coordinates, `geoTrajectory` is typically used for a trajectory from hundreds to thousands of kilometers of distance span.

- `kinematicTrajectory` — Defines a trajectory using kinematic properties, such as acceleration and angular acceleration, expressed in the platform body frame. You can use `kinematicTrajectory` to generate a trajectory of any distance span as long as the kinematic information of the trajectory is available. The object assumes a Cartesian coordinate reference frame.

The two waypoint-based trajectory objects (`waypointTrajectory` and `geoTrajectory`) can automatically calculate the linear velocity information of the platform, but you can also explicitly specify the linear velocity using the `Velocity` property or a combination of the `Course`, `GroundSpeed`, and `ClimbRate` properties.

The trajectory of a platform is composed of rotational motion and translational motion. By default, the two waypoint-based trajectory objects (`waypointTrajectory` and `geoTrajectory`) automatically generate the orientation of the platform at each waypoint by aligning the yaw angle with the path of the trajectory, but you can explicitly specify the orientation using the `Orientation` property. Alternately, you can use the `AutoPitch` and `AutoBank` properties to enable automatic pitch and roll angles, respectively. For `kinematicTrajectory`, you need to use the `Orientation` property and the angular velocity input to specify the rotational motion of the trajectory.

waypointTrajectory

The `waypointTrajectory` System object defines a trajectory that smoothly passes through waypoints expressed in Cartesian coordinates. Generally, you can use `waypointTrajectory` to model vehicles travelling within hundreds of kilometers. These vehicles include automobiles, surface marine craft, and commercial aircraft (helicopters, planes, and quadcopters). You can choose the reference frame as a fixed ENU or NED frame using the `ReferenceFrame` property. For more details on how the object generates the trajectory, see the "Algorithms" (Sensor Fusion and Tracking Toolbox) section of `waypointTrajectory`.

waypointTrajectory Example for Aircraft Landing

Define the trajectory of a landing aircraft using a `waypointTrajectory` object.

```
waypoints = [-421 -384 2000;
    47 -294 1600;
    1368 174 1300;
    995 1037 900;
    -285 293 600;
    -1274 84 350;
    -2328 101 150;
    -3209 83 0];
timeOfArrival =  [0; 16.71; 76.00; 121.8; 204.3; 280.31; 404.33; 624.6];
aircraftTraj = waypointTrajectory(waypoints,timeOfArrival);
```

Create a `theaterPlot` object to visualize the trajectory and the aircraft.

```
minCoords = min(waypoints);
maxCoords = max(waypoints);
tp = theaterPlot('XLimits',1.2*[minCoords(1) maxCoords(1)], ...
    'YLimits',1.2*[minCoords(2) maxCoords(2)], ...
    'ZLimits',1.2*[minCoords(3) maxCoords(3)]);

% Create a trajectory plotter and a platform plotter
tPlotter = trajectoryPlotter(tp,'DisplayName','Trajectory');
pPlotter = platformPlotter(tp,'DisplayName','Aircraft');
```

Obtain the Cartesian waypoints of the trajectory using the `lookupPose` object function.

```
sampleTimes = timeOfArrival(1):timeOfArrival(end);
numSteps = length(sampleTimes);
[positions,orientations] = lookupPose(aircraftTraj,sampleTimes);
plotTrajectory(tPlotter,{positions})
axis equal
```

Plot the platform motion using an airplane mesh object.

```
mesh = scale(rotate(tracking.scenario.airplaneMesh,[0 0 180]),15); % Exaggerated scale for better
view(20.545,-20.6978)
for i = 1:numSteps
    plotPlatform(pPlotter,positions(i,:),mesh,orientations(i))
    % Uncomment the next line to slow the aircraft motion animation
    % pause(1e-7)
end
```

In the animation, the yaw angle of the aircraft aligns with the trajectory by default.

Create a second aircraft trajectory with the same waypoints as the first aircraft trajectory, but set its `AutoPitch` and `AutoBank` properties to `true`. This generates a trajectory more representative of the possible aircraft maneuvers.

```
aircraftTraj2 = waypointTrajectory(waypoints,timeOfArrival, ...
    'AutoPitch',true, ...
    'AutoBank',true);
```

Plot the second trajectory and observe the change in aircraft orientation.

```
[positions2,orientations2] = lookupPose(aircraftTraj2,sampleTimes);
for i = 1:numSteps
    plotPlatform(pPlotter,positions2(i,:),mesh,orientations2(i));
    % Uncomment the next line to slow the aircraft motion animation
    % pause(1e-7)
end
```
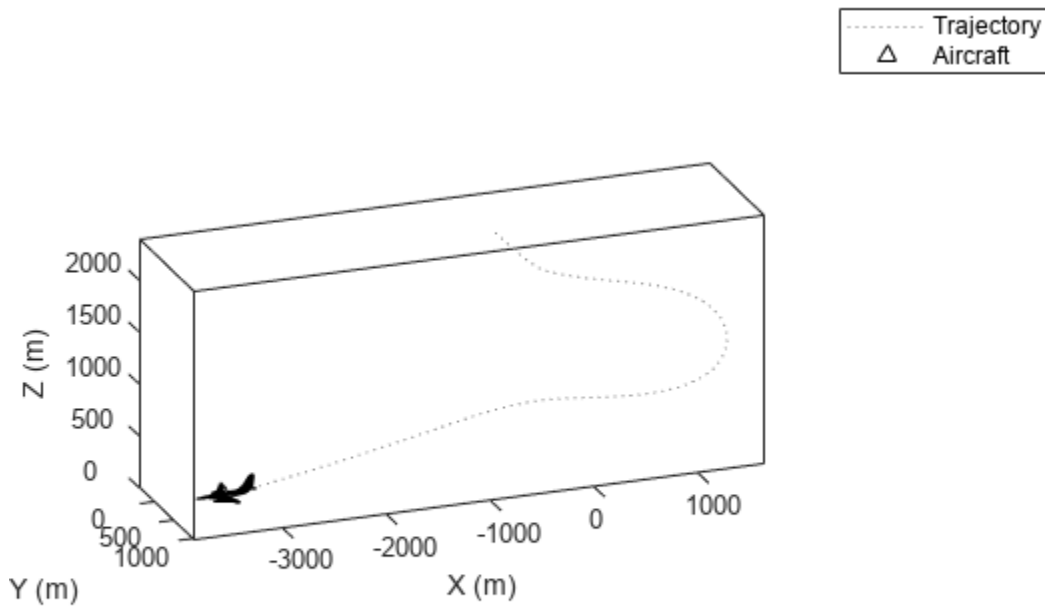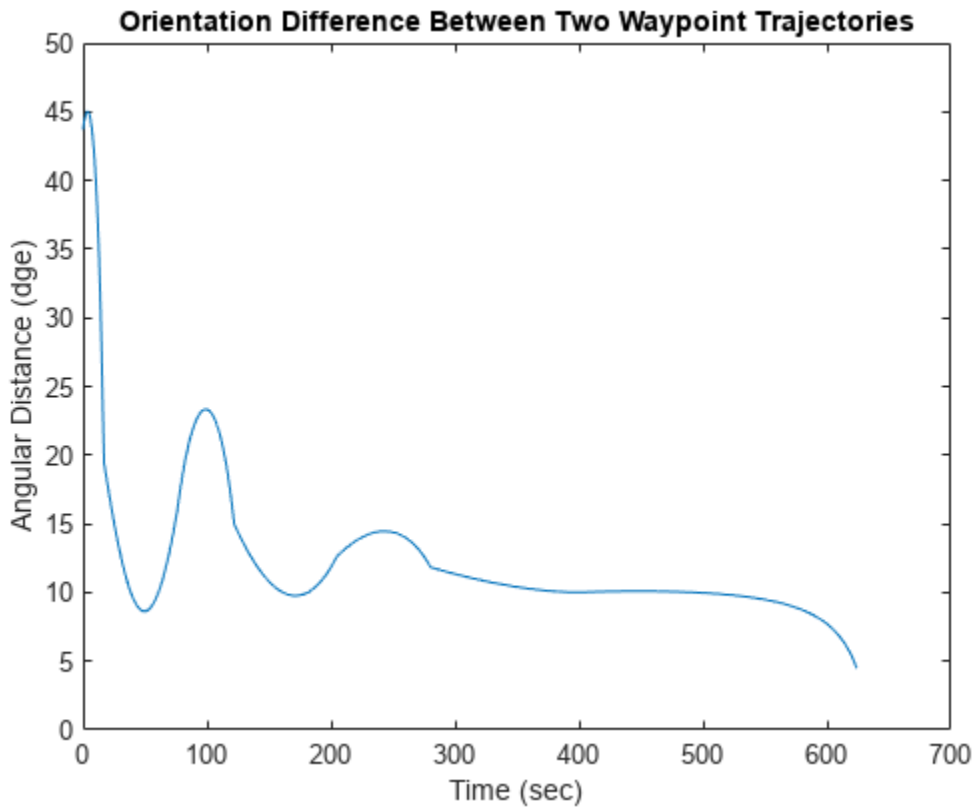

Visualize the orientation differences between the two trajectories in angles.

```
distance = dist(orientations2,orientations);
figure
plot(sampleTimes,distance*180/pi)
xlabel('Time (sec)')
ylabel('Angular Distance (dge)')
title('Orientation Difference Between Two Waypoint Trajectories')
```

Orientation Difference Between Two Waypoint Trajectories

geoTrajectory

The `geoTrajectory` System object generates a trajectory using waypoints in a similar fashion as the `waypointTrajectory` object, but it has two major differences in how to specify waypoints and velocity inputs.

- When specifying waypoints for `geoTrajectory`, express each waypoint in the geodetic coordinates of latitude, longitude, and altitude above the WG84 ellipsoid model. Using geodetic coordinates, you can conveniently specify long-range trajectories, such as airplane flight trajectory on a realistic Earth model.
- When specifying the orientation and velocity information for each waypoint, the reference frame for orientation and velocity is the local NED or ENU frame defined under the current trajectory waypoint. For example, the N_1-E_1-D_1 frame shown in the figure is a local NED reference frame.
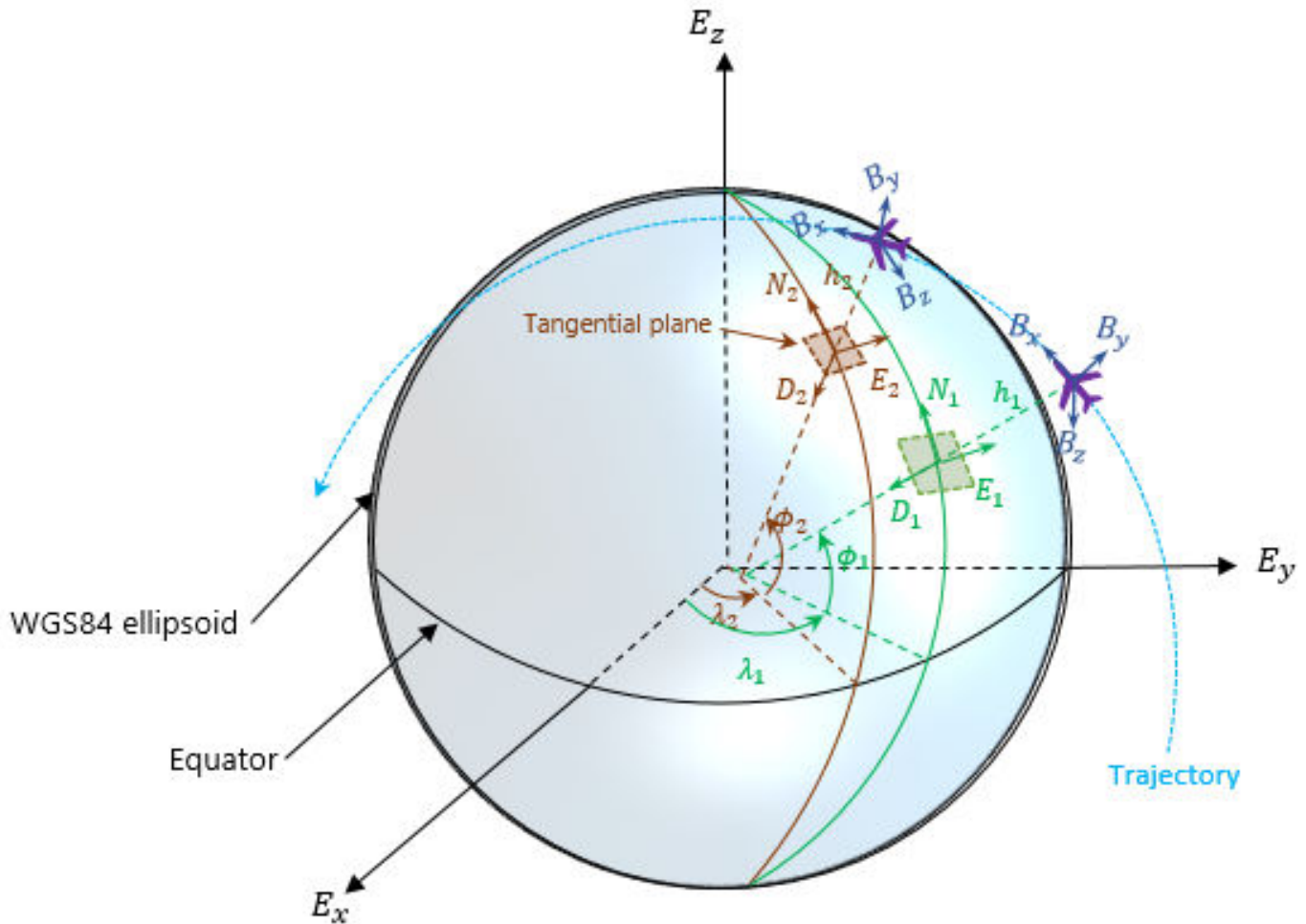
In the figure,

- E_x, E_y, and E_z are the three axes of the Earth-centered Earth-fixed (ECEF) frame, which is fixed on the Earth.
- (λ_1, ϕ_1, h_1) and (λ_2, ϕ_2, h_2) are the geodetic coordinates of the plane at the two waypoints.
- (N_1, E_1, D_1) and (N_2, E_2, D_2) are the two local NED frames corresponding to the two trajectory waypoints.
- B_x, B_y, and B_z are the three axes of the platform body frame, which is fixed on the platform.

geoTrajectory Example For Flight Trajectory

Load the flight data of a flight trajectory from Los Angeles to Boston. The data contains flight information including flight time, geodetic coordinates for each waypoint, course, and ground speed.

```
load flightData.mat
```

Create a `geoTrajectory` object based on the flight data.

```
planeTraj = geoTrajectory([latitudes longitudes heights],timeOfArrival, ...
    'Course',courses,'GroundSpeed',speeds);
```

Look up the Cartesian coordinates of the waypoints in the ECEF frame.

```
sampleTimes = 0:1000:3600*10;
positionsCart = lookupPose(planeTraj,sampleTimes,'ECEF');
```

Show the trajectory using the `helperGlobeView` class, which approximates the Earth sphere.

```
viewer = helperGlobeView;
plot3(positionsCart(:,1),positionsCart(:,2),positionsCart(:,3),'r')
```


You can further explore the trajectory by querying other outputs of the trajectory.

kinematicTrajectory

Unlike the two waypoint trajectory objects, the `kinematicTrajectory` System object uses kinematic attributes to specify a trajectory. Think of the trajectory as a numerical integration of the pose (position and orientation) and linear velocity of the platform, based on the linear acceleration and angular acceleration information. The pose and linear velocity are specified with respected to a chosen, fixed scenario frame, whereas the linear acceleration and angular velocity are specified with respected to the platform body frame.

kinematicTrajectory Example For UAV Path

Create a `kinematicTrajectory` object for simulating a UAV path. Specify the time span of the trajectory as 120 seconds.

```
traj = kinematicTrajectory('SampleRate',1, ...
    'AngularVelocitySource','Property');

tStart = 0;
tFinal = 120;
tspan = tStart:tFinal;

numSteps = length(tspan);
positions = NaN(3,numSteps);
velocities = NaN(3,numSteps);
vel = NaN(1,numSteps);
```

To form a square path covering a small region, separate the UAV trajectory into six segments:

- Taking off and ascending in the z-direction

- Moving in the positive x-direction

- Moving in the positive y-direction

- Moving in the negative x-direction

- Moving in the negative y-direction

- Descending in the z-direction and landing

In each segment, the UAV accelerates in one direction and then decelerates in that direction with the same acceleration magnitude. As a result, at the end of each segment, the velocity of the UAV is zero.

```
segSteps = floor(numSteps/12);
accelerations = zeros(3,numSteps);
acc = 1;
% Acceleration for taking off and ascending in the z-direction
accelerations(3,1:segSteps) = acc;
accelerations(3,segSteps+1:2*segSteps) = -acc;

% Acceleration for moving in the positive x-direction
accelerations(1,2*segSteps+1:3*segSteps) = acc;
accelerations(1,3*segSteps+1:4*segSteps) = -acc;

% Acceleration for moving in the positive y-direction
accelerations(2,4*segSteps+1:5*segSteps) = acc;
accelerations(2,5*segSteps+1:6*segSteps) = -acc;

% Acceleration for moving in the negative x-direction
accelerations(1,6*segSteps+1:7*segSteps) = -acc;
accelerations(1,7*segSteps+1:8*segSteps) = acc;

% Acceleration for moving in the negative y-direction
accelerations(2,8*segSteps+1:9*segSteps) = -acc;
accelerations(2,9*segSteps+1:10*segSteps) = acc;

% Descending in the z-direction and landing
accelerations(3,10*segSteps+1:11*segSteps) =-acc;
accelerations(3,11*segSteps+1:end) = acc;
```

Simulate the trajectory by calling the `kinematicTrajectory` object with the specified acceleration.

```
for i = 1:numSteps
    [positions(:,i),~,velocities(:,i)] = traj(accelerations(:,i)');
```

```
        vel(i) = norm(velocities(:,i));
end
```

Visualize the trajectory using `theaterPlot`.

```
% Create a theaterPlot object and create plotters for the trajectory and the
% UAV Platform.
figure
tp = theaterPlot('XLimits',[-30 130],'YLimits',[-30 130],'ZLimits',[-30 130]);
tPlotter = trajectoryPlotter(tp,'DisplayName','UAV trajectory');
pPlotter = platformPlotter(tp,'DisplayName','UAV','MarkerFaceColor','g');

% Plot the trajectory.
plotTrajectory(tPlotter,{positions'})
view(-43.18,56.49)

% Use a cube to represent the UAV platform.
dims = struct('Length',10, ...
    'Width',5, ...
    'Height',3, ...
    'OriginOffset',[0 0 0]);

% Animate the UAV platform position.
for i = 1:numSteps
    plotPlatform(pPlotter,positions(:,i)',dims,eye(3))
    pause(0.01)
end
```
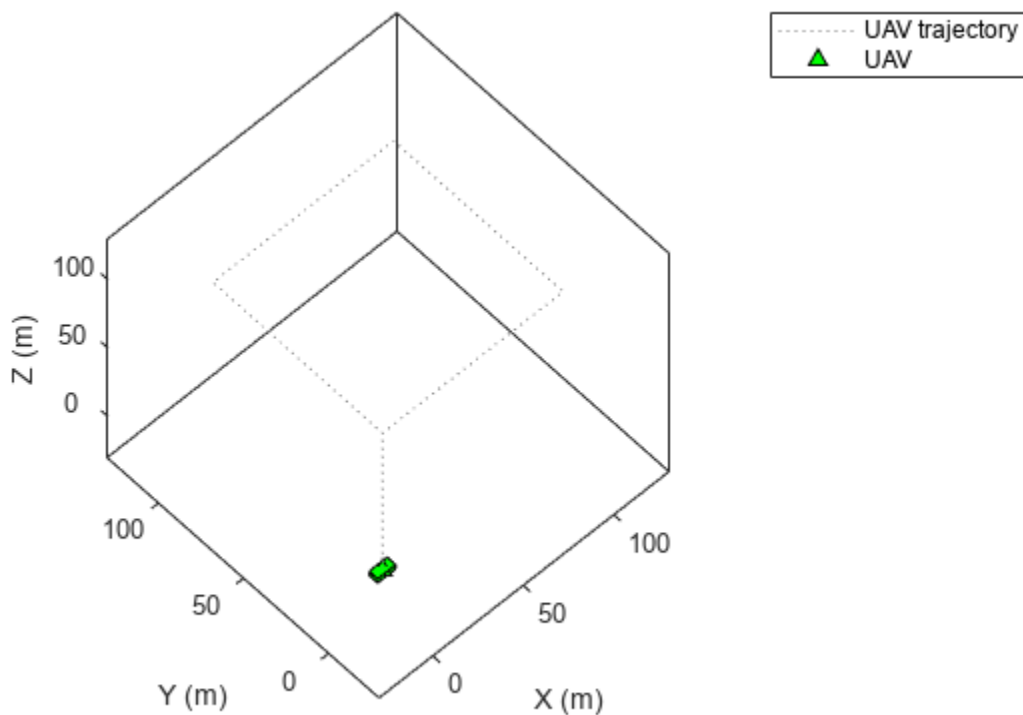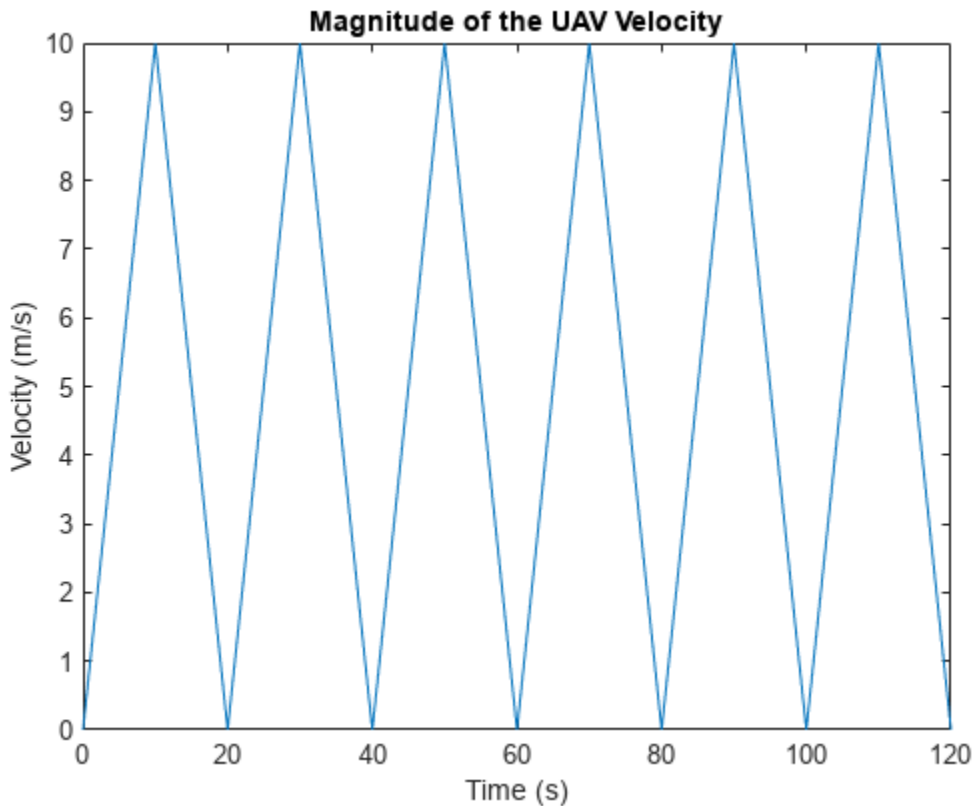
Show the velocity magnitude of the UAV platform.

```
figure
plot(tspan,vel)
xlabel('Time (s)')
ylabel('Velocity (m/s)')
title('Magnitude of the UAV Velocity')
```


kinematicTrajectory Example For Spacecraft Trajectory

Use `kinematicTrajetory` to specify a circular spacecraft trajectory. The orbit has these elements:

- Orbital radius (r) — 7000 km
- Inclination (i) — 60 degrees
- Argument of ascending node (Ω) — 90 degrees. The ascending node direction is aligned with the Y-direction.
- True anomaly (ν) — –90 degrees

In the figure, X-Y-Z is the Earth-centered inertial (ECI) frame, which has a fixed position and orientation in space. x-y-z is the spacecraft body frame, fixed on the spacecraft. $\vec{r}$ and $\vec{v}$ are the initial position and velocity of the spacecraft.
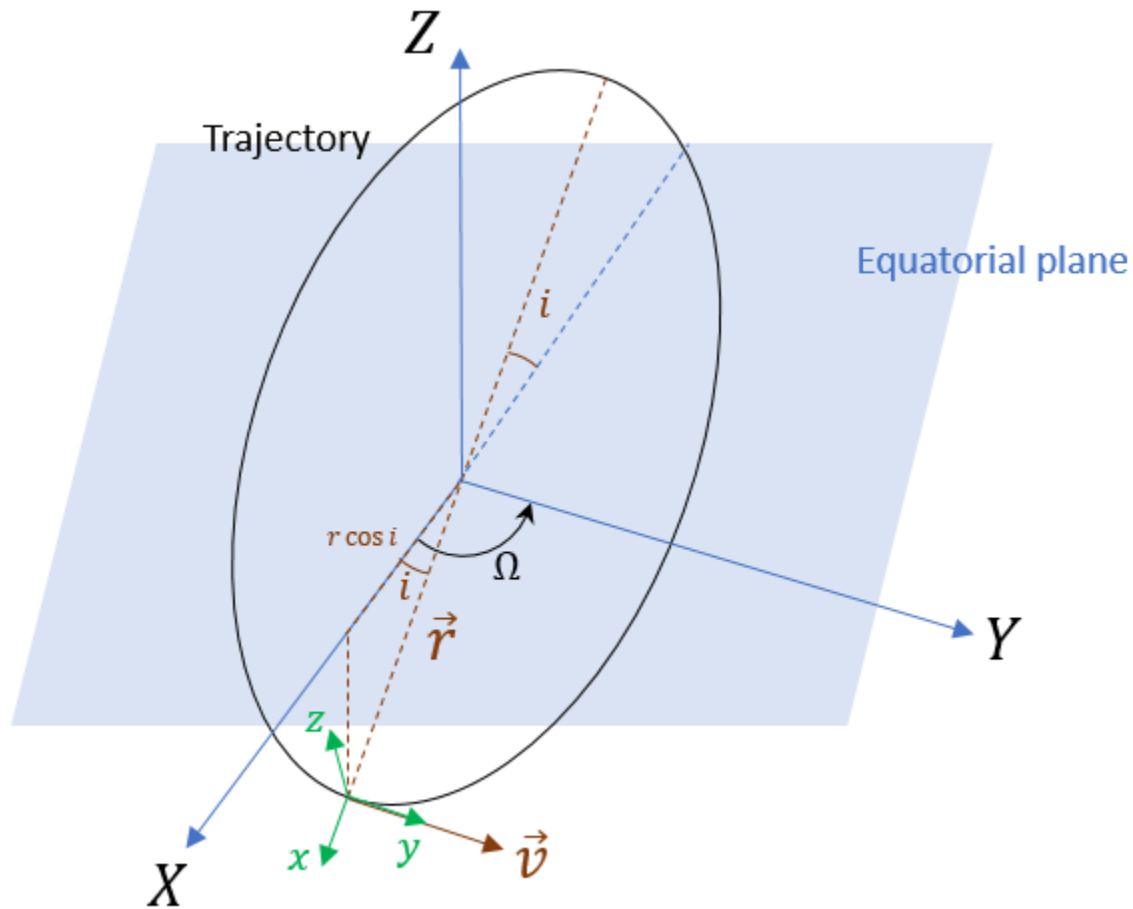
To specify the circular orbit using `kinematicTrajectory`, you need to provide the initial position, initial velocity, and initial orientation of the spacecraft with respect to the ECI frame. For the chosen true anomaly (v = –90˚), the spacecraft velocity is aligned with the *Y*-direction.

```
inclination = 60; % degrees
mu = 3.986e14; % standard earth gravitational parameter
radius = 7000e3;% meters
v = sqrt(mu/radius); % speed
initialPosition = [radius*cosd(inclination),0,-radius*sind(inclination)]';
initialVelocity = [0 v 0]';
```

Assume the x-direction of the spacecraft body frame is the radial direction, the z-direction is the normal direction of the orbital plane, and the y-direction completes the right-hand rule. Use the assumptions to specify the orientation of the body frame at the initial position.

```
orientation = quaternion([0 inclination 0],'eulerd','zyx','frame');
```

Express the angular velocity and the angular acceleration of the trajectory in the platform body frame.

```
omega = v/radius;
angularVelocity = [0 0 omega]';
```

```matlab
a = v^2/radius;
acceleration = [-a 0 0]';
```

Set a simulation time of one orbital period. Specify a simulation step as 2 seconds.

```matlab
tFinal = 2*pi/omega;
dt = 2;
sampleRate = 1/dt;
tspan = 0:dt:tFinal;
numSteps = length(tspan);
```

Create the spacecraft trajectory. Since the acceleration and angular velocity of the spacecraft remain constant with respect to the spacecraft body frame, specify them as constants. Generate position and orientation outputs along the trajectory by using the `kinematicTrajectory` object.

```matlab
traj = kinematicTrajectory('SampleRate',sampleRate, ...
    'Position',initialPosition, ...
    'Velocity',initialVelocity, ...
    'Orientation',orientation, ...
    'AngularVelocity',angularVelocity, ...
    'Acceleration',acceleration, ...
    'AccelerationSource','Property', ...
    'AngularVelocitySource','Property');

% Generate position and orientation outputs.
positions = NaN(3,numSteps);
orientations = zeros(numSteps,1,'quaternion');
for i = 1:numSteps
    [positions(:,i),orientations(i)] = traj();
end
```

Use the `helperGlobeView` class and `theaterPlot` to show the trajectory.

```matlab
viewer = helperGlobeView(0,[60 0]);
tp = theaterPlot('Parent',gca,...
    'XLimits',1.2*[-radius radius],...
    'YLimits',1.2*[-radius radius],...
    'ZLimits',1.2*[-radius radius]);
tPlotter = trajectoryPlotter(tp,'LineWidth',2);
pPlotter = platformPlotter(tp,'MarkerFaceColor','m');
legend(gca,'off')

plotTrajectory(tPlotter,{positions'})
% Use a cube with exaggerated dimensions to represent the spacecraft.
dims = struct('Length',8e5,'Width',4e5,'Height',2e5,'OriginOffset',[0 0 0]);

for i = 1:numSteps
    plotPlatform(pPlotter,positions(:,i)',dims,orientations(i))
    % Since the reference frame is the ECI frame, earth rotates with respect to it.
    rotate(viewer,dt)
end
```

Summary

In this topic, you learned how to use three trajectory objects to customize your own trajectories based on the available information. In addition, you learned the fundamental differences in applying them. This table highlights the main attributes of these trajectory objects.

Trajectory Object	Position Inputs	Linear Velocity Inputs	Orientation	Acceleration and Angular Velocity Inputs	Recommended Distance Span
waypointTrajectory	Cartesian waypoints expressed in a fixed frame (NED or ENU)	One of these options: • Automatically generate velocity for a smooth trajectory, by default • Specify velocity in the fixed frame at each waypoint • Specify course, ground speed, and climb rate in the fixed frame at each waypoint	One of these options: • Auto yaw by default, auto pitch by selection, and auto bank by selection • Specify orientation in the fixed frame	Cannot specify	From within tens to hundreds of kilometers

Trajectory Object	Position Inputs	Linear Velocity Inputs	Orientation	Acceleration and Angular Velocity Inputs	Recommended Distance Span
geoTrajectory	Geodetic waypoints in the ECEF frame	One of the these options: • Automatically generate velocity for a smooth trajectory, by default • Specify velocity in the local frame (NED or ENU) for each waypoint • Specify course, ground speed, and climb rate in the local frame (NED or ENU) for each waypoint	One of these options: • Auto yaw by default, auto pitch by selection, and auto bank by selection • Specify orientation in the local frame	Cannot specify	From hundreds to thousands of kilometers
kinematicTrajectory	Initial position expressed in a chosen, fixed frame	Only initial velocity in the fixed frame	Only initial orientation in the fixed frame	Specify acceleration and angular velocity in the platform body frame	Unlimited distance span

8

Kalman Filters

- "Linear Kalman Filters" on page 8-2
- "Extended Kalman Filters" on page 8-10

Linear Kalman Filters

Kalman filters track an object using a sequence of detections or measurements to estimate the state of the object based on the motion model of the object. In a motion model, state is a collection of quantities that represent the status of an object, such as its position, velocity, and acceleration. An object motion model is defined by the evolution of the object state.

The linear Kalman filter (`trackingKF`) is an optimal, recursive algorithm for estimating the state of an object if the estimation system is linear and Gaussian. An estimation system is linear if both the motion model and measurement model are linear. The filter works by recursively predicting the object state using the motion model and correcting the state using measurements.

Motion Model

For most types of objects tracked in the toolbox, the state vector consists of one-, two-, or three-dimensional positions and velocities.

Consider an object moving in the x-direction at a constant acceleration. You can write the equation of motion, using Newtonian equations, as:

$$m\ddot{x} = f$$

$$\ddot{x} = \frac{f}{m} = a$$

Furthermore, if you define the state as:

$$x_1 = x$$

$$x_2 = \dot{x},$$

you can write the equation of motion in state-space form as:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}a$$

In most cases, a motion model does not fully model the motion of an object, and you need to include the process noise to compensate the uncertainty in the motion model. For the constant velocity model, you can add process noise as an acceleration term.

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}a + \begin{bmatrix} 0 \\ 1 \end{bmatrix}v_k$$

Here, v_k is the unknown noise perturbation of the acceleration. For the filter to be optimal, you must assume the process noise is zero-mean, white Gaussian noise.

You can extend this type of equation to more than one dimension. In two dimensions, the equation has the form:

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ a_x \\ 0 \\ a_y \end{bmatrix} + \begin{bmatrix} 0 \\ v_x \\ 0 \\ v_y \end{bmatrix}$$

The 4-by-4 matrix in this equation is the state transition matrix. For independent x- and y-motions, this matrix is block diagonal.

When you convert a continuous time model to a discrete time model, you integrate the equations of motion over the length of the time interval. In the discrete form, for a sample interval of T, the state representation becomes:

$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x_{1,k} \\ x_{2,k} \end{bmatrix} + \begin{bmatrix} \frac{1}{2}T^2 \\ T \end{bmatrix}a + \begin{bmatrix} \frac{1}{2}T^2 \\ T \end{bmatrix}\tilde{v}$$

where x_{k+1} is the state at discrete time $k+1$, and x_k is the state at the earlier discrete time k. If you include noise, the equation becomes more complicated, because the integration of noise is not straightforward. For details on how to obtain the discretized process noise from a continuous system, See [1].

You can generalize the state equation to:

$$x_{k+1} = A_k x_k + B_k u_k + G_k v_k$$

where A_k is the state transition matrix and B_k is the control matrix. The control matrix accounts for any known forces acting on the object. v_k represents discretized process noise, following a Gaussian distribution of mean 0 and covariance Q_k. G_k is the process noise gain matrix.

Measurement Models

Measurements are what you observe or measure in a system. Measurements depend on the state vector, but are usually not the same as the state vector. For instance, in a radar system, the measurements can be spherical coordinates such as range, azimuth, and elevation, while the state vector is the Cartesian position and velocity. A linear Kalman filter assumes the measurements are a linear function of the state vector. To apply nonlinear measurement models, you can choose to use an extended Kalman filter (`trackingEKF`) or an unscented Kalman filter (`trackingUKF`).

You can represent a linear measurement as:

$$z_k = H_k x_k + w_k$$

Here, H_k is the measurement matrix and w_k represents measurement noise at the current time step. For an optimal filter, the measurement noise must be zero-mean, Gaussian white noise. Assume the covariance matrix of the measurement noise is R_k .

Filter Loop

The filter starts with best estimates of the state $x_{0|0}$ and the state covariance $P_{0|0}$. The filter performs these steps in a recursive loop.

1   Propagate the state to the next step using the motion equation:

$$x_{k+1|k} = F_k x_{k|k} + B_k u_k \,.$$

Propagate the covariance matrix as well:

$$P_{k+1|k} = F_k P_{k|k} F_k^T + G_k Q_k G_k^T \,.$$

The subscript notation $k+1|k$ indicates that the corresponding quantity is the estimate at the $k+1$ step propagated from step k. This estimate is often called the *a priori* estimate. The predicted measurement at the $k+1$ step is

$$z_{k+1|k} = H_{k+1} x_{k+1|k}$$

2   Use the difference between the actual measurement and the predicted measurement to correct the state at the $k+1$ step. To correct the state, the filter must compute the Kalman gain. First, the filter computes the measurement prediction covariance (innovation) as:

$$S_{k+1} = H_{k+1} P_{k+1|k} H_{k+1}^T + R_{k+1}$$

Then, the filter computes the Kalman gain as:

$$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$$

3   The filter corrects the predicted estimate by using the measurement. The estimate, after correction using the measurement z_{k+1}, is

$$x_{k+1|k+1} = x_{k+1|k} + K_{k+1}(z_{k+1} - z_{k+1|k})$$

where K_{k+1} is the Kalman gain. The corrected state is often called the *a posteriori* estimate of the state, because it is derived after including the measurement.

The filter corrects the state covariance matrix as:

$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1} S_{k+1} K_{k+1}^T$$

This figure summarizes the Kalman loop operations. Once initialized, a Kalman filter loops between prediction and correction until reaching the end of the simulation.

Initialize

Predict

$$x_{k+1|k} = A_k x_{k|k} + B_k u_k$$
$$P_{k+1|k} = A_k P_{k|k} A_k^T + G_k Q_k G_k^T$$
$$z_{k+1|k} = H_{k+1} x_{k+1|k}$$

$x_{0|0}, P_{0|0}$

Correct

$$S_{k+1} = H_{k+1} P_{k+1|k} H_{k+1}^T + R_{k+1}$$
$$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$$
$$x_{k+1|k+1} = x_{k+1|k} + K_{k+1}(z_{k+1} - z_{k+1|k})$$
$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1} S_{k+1} K_{k+1}^T$$
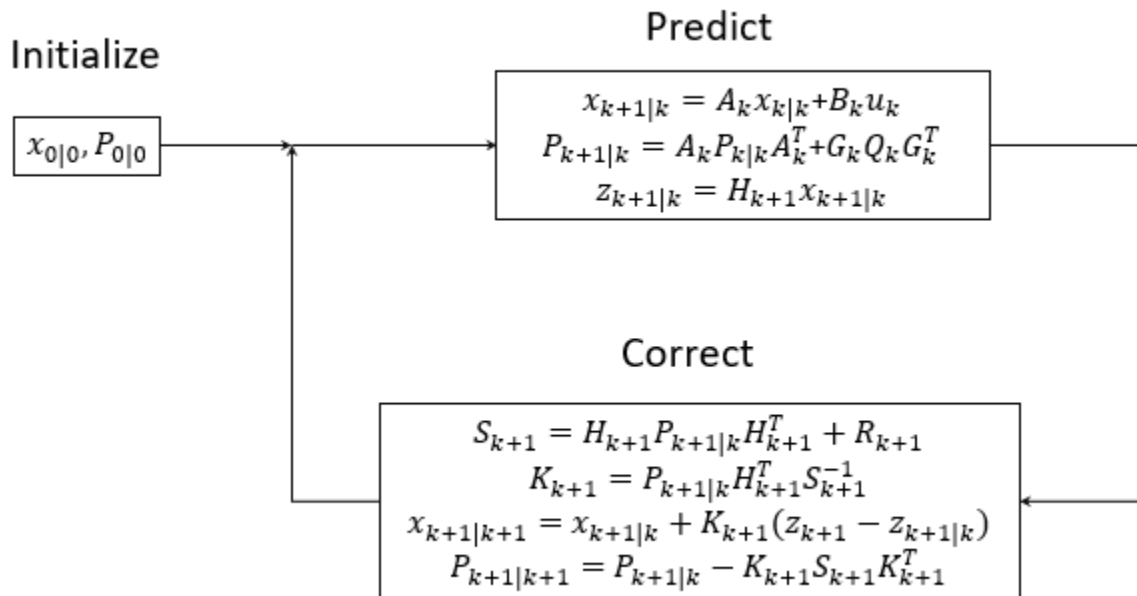
Built-In Motion Models in trackingKF

When you only need to use the standard 1-D, 2-D, or 3-D constant velocity or constant acceleration motion models, you can specify the `MotionModel` property of `trackingKF` as one of these:

- `"1D Constant Velocity"`
- `"1D Constant Acceleration"`
- `"2D Constant Velocity"`
- `"2D Constant Acceleration"`
- `"3D Constant Velocity"`
- `"3D Constant Acceleration"`

To customize your own motion model, specify the `MotionModel` property as `"Custom"`, and then specify the state transition matrix in the `StateTransitionModel` property of the filter.

For the 3-D constant velocity model, the state equation is:

$$
\begin{bmatrix} x_{k+1} \\ v_{x,k+1} \\ y_{k+1} \\ v_{y,k+1} \\ z_{k+1} \\ v_{z,k+1} \end{bmatrix}
=
\begin{bmatrix} 1 & T & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & T & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} x_k \\ v_{x,k} \\ y_k \\ v_{y,k} \\ z_k \\ v_{z,k} \end{bmatrix}
$$

For the 3-D constant acceleration model, the state equation is:

$$
\begin{bmatrix} x_{k+1} \\ v_{x,k+1} \\ a_{x,k+1} \\ y_{k+1} \\ v_{y,k+1} \\ a_{y,k+1} \\ z_{k+1} \\ v_{z,k+1} \\ a_{z,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & T & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ a_{x,k} \\ y_k \\ v_{y,k} \\ a_{y,k} \\ z_k \\ v_{z,k} \\ a_{z,k} \end{bmatrix}
$$

Example: Estimate 2-D Target States Using trackingKF

Initialize Estimation Model

Specify an initial position and velocity for a target that you assume moving in 2-D. The simulation lasts 20 seconds with a sample time of 0.2 seconds.

```
rng(2021);      % For repeatable results
dt = 0.2;       % seconds
simTime = 20; % seconds
tspan = 0:dt:simTime;
trueInitialState = [30; 2; 40; 2]; % [x;vx;y;vy]
processNoise = diag([0; 1; 0; 1]); % Process noise matrix
```

Create a measurement noise covariance matrix, assuming that the target measurements consist of its position states.

```
measureNoise = diag([4 4]); % Measurement noise matrix
```

The matrix specifies a standard deviation of 2 meters in both the x- and y-directions.

Preallocate variables in which to save estimation results.

```
numSteps = length(tspan);
trueStates = NaN(4,numSteps);
trueStates(:,1) = trueInitialState;
estimateStates = NaN(size(trueStates));
```

Obtain True States and Measurements

Propagate the constant velocity model, and generate the measurements with noise.

```
F = [1 dt 0  0;
     0  1 0  0;
     0  0 1 dt;
     0  0 0  1];
H = [1 0 0 0;
     0 0 1 0];
for i = 2:length(tspan)
    trueStates(:,i) = F*trueStates(:,i-1) + sqrt(processNoise)*randn(4,1);
```

```
end
measurements = H*trueStates + sqrt(measureNoise)*randn(2,numSteps);
```

Plot the true trajectory and the measurements.

```
figure
plot(trueStates(1,1),trueStates(3,1),"r*",DisplayName="True Initial")
hold on
plot(trueStates(1,:),trueStates(3,:),"r",DisplayName="Truth")
plot(measurements(1,:),measurements(2,:),"kx",DisplayName="Measurements")
xlabel("x (m)")
ylabel("y (m)")
axis image
```
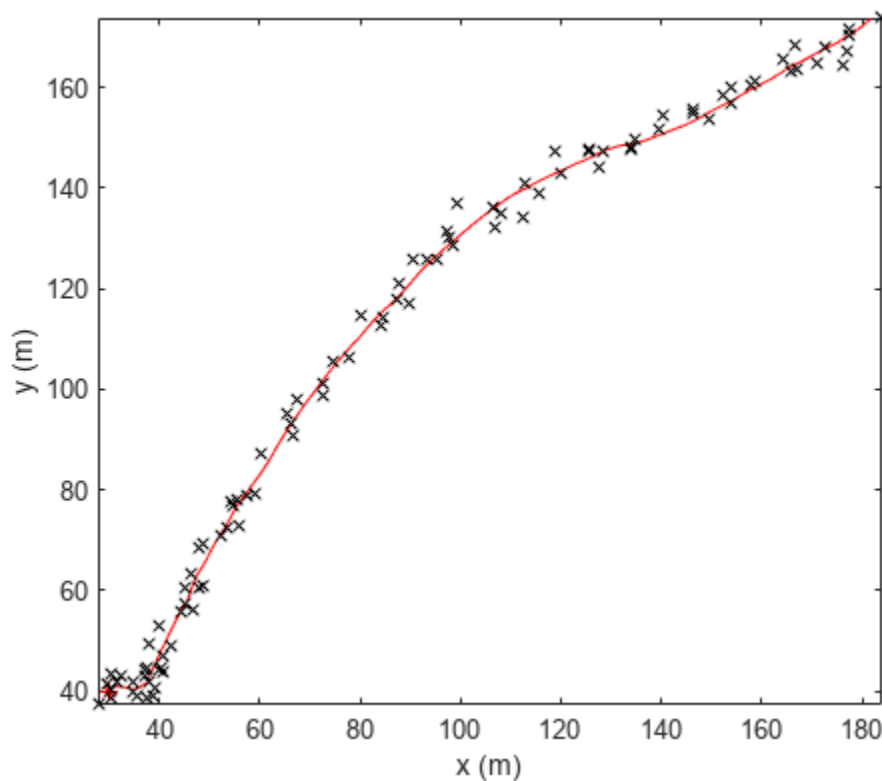

Initialize Linear Kalman Filter

Initialize the filter with a state of [40; 0; 160; 0], which is far from the true initial state. Normally, you can use the initial measurement to construct an initial state as [measurements(1,1);0 ; measurements(2,1); 0]. Here, you use an erroneous initial state, which enables you to test if the filter can quickly converge on the truth.

```
filter = trackingKF(MotionModel="2D Constant Velocity",State=[40; 0; 160;0], ...
    MeasurementModel=H,MeasurementNoise=measureNoise)

filter =
  trackingKF with properties:

                State: [4x1 double]
```

```
         StateCovariance: [4x4 double]

             MotionModel: '2D Constant Velocity'
            ProcessNoise: [2x2 double]

       MeasurementModel: [2x4 double]
       MeasurementNoise: [2x2 double]

          MaxNumOOSMSteps: 0

          EnableSmoothing: 0
```

```
 estimateStates(:,1) = filter.State;
```

Run Linear Kalman Filter and Show Results

Run the filter by recursively calling the `predict` and `correct` object functions. From the results, the estimates converge on the truth quickly. In fact, the linear Kalman filter has an exponential convergence speed.

```matlab
for i=2:length(tspan)
    predict(filter,dt)
    estimateStates(:,i) = correct(filter,measurements(:,i));
end
plot(estimateStates(1,1),estimateStates(3,1),"g*",DisplayName="Initial Estimates")
plot(estimateStates(1,:),estimateStates(3,:),"g",DisplayName="Estimates")
legend(Location="southeast")
```
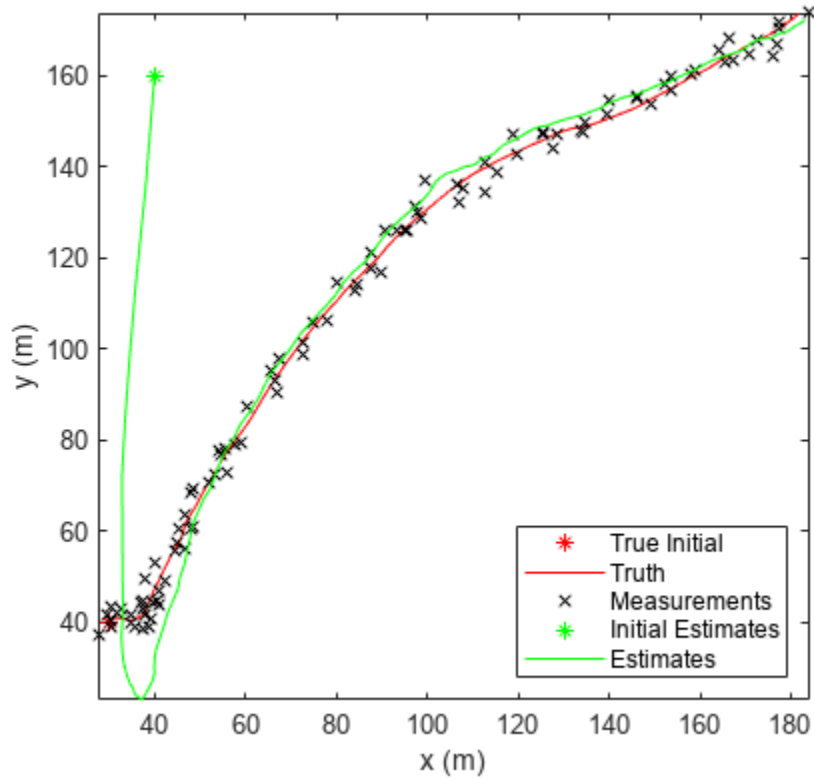
See Also

`trackingKF` | `trackingEKF` | `trackingUKF` | "Extended Kalman Filters" on page 8-10

References

[1] Li, X. Rong, and Vesselin P. Jilkov. "Survey of Maneuvering Target Tracking: Dynamic Models". Edited by Oliver E. Drummond, 2000, pp. 212–35. DOI.org (Crossref), https://doi.org/ 10.1117/12.391979.

Extended Kalman Filters

In this section...
"State Update Model" on page 8-10
"Measurement Model" on page 8-11
"Extended Kalman Filter Loop" on page 8-11
"Predefined Extended Kalman Filter Functions" on page 8-12
"Example: Estimate 2-D Target States with Angle and Range Measurements Using trackingEKF" on page 8-13

When you use a filter to track objects, you use a sequence of detections or measurements to estimate the state of an object based on the motion model of the object. In a motion model, state is a collection of quantities that represent the status of an object, such as its position, velocity, and acceleration. Use an extended Kalman filter (`trackingEKF`) when object motion follows a nonlinear state equation or when the measurements are nonlinear functions of the state. For example, consider using an extended Kalman filter when the measurements of the object are expressed in spherical coordinates, such as azimuth, elevation, and range, but the states of the target are expressed in Cartesian coordinates.

The formulation of an extended Kalman is based on the linearization of the state equation and measurement equation. Linearization enables you to propagate the state and state covariance in an approximately linear format, and requires Jacobians of the state equation and measurement equation.

Note If your estimate system is linear, you can use the linear Kalman filter (`trackingKF`) or the extended Kalman filter (`trackingEKF`) to estimate the target state. If your system is nonlinear, you should use a nonlinear filter, such as the extended Kalman filter or the unscented Kalman filter (`trackingUKF`).

State Update Model

Assume a closed-form expression for the predicted state as a function of the previous state x_k, controls u_k, noise w_k, and time t.

$$x_{k+1} = f(x_k, u_k, w_k, t)$$

The Jacobian of the predicted state with respect to the previous state is obtained by partial derivatives as:

$$F^{(x)} = \frac{\partial f}{\partial x}.$$

The Jacobian of the predicted state with respect to the noise is:

$$F^{(w)} = \frac{\partial f}{\partial w}.$$

These functions take simpler forms when the noise is additive in the state update equation:

$$x_{k+1} = f(x_k, u_k, t) + w_k$$

In this case, $F^{(w)}$ is an identity matrix.

You can specify the state Jacobian matrix using the `StateTransitionJacobianFcn` property of the `trackingEKF` object. If you do not specify this property, the object computes Jacobians using numeric differencing, which is slightly less accurate and can increase the computation time.

Measurement Model

In an extended Kalman filter, the measurement can also be a nonlinear function of the state and the measurement noise.

$$z_k = h(x_k, v_k, t)$$

The Jacobian of the measurement with respect to the state is:

$$H^{(x)} = \frac{\partial h}{\partial x}.$$

The Jacobian of the measurement with respect to the measurement noise is:

$$H^{(v)} = \frac{\partial h}{\partial v}.$$

These functions take simpler forms when the noise is additive in the measurement equation:
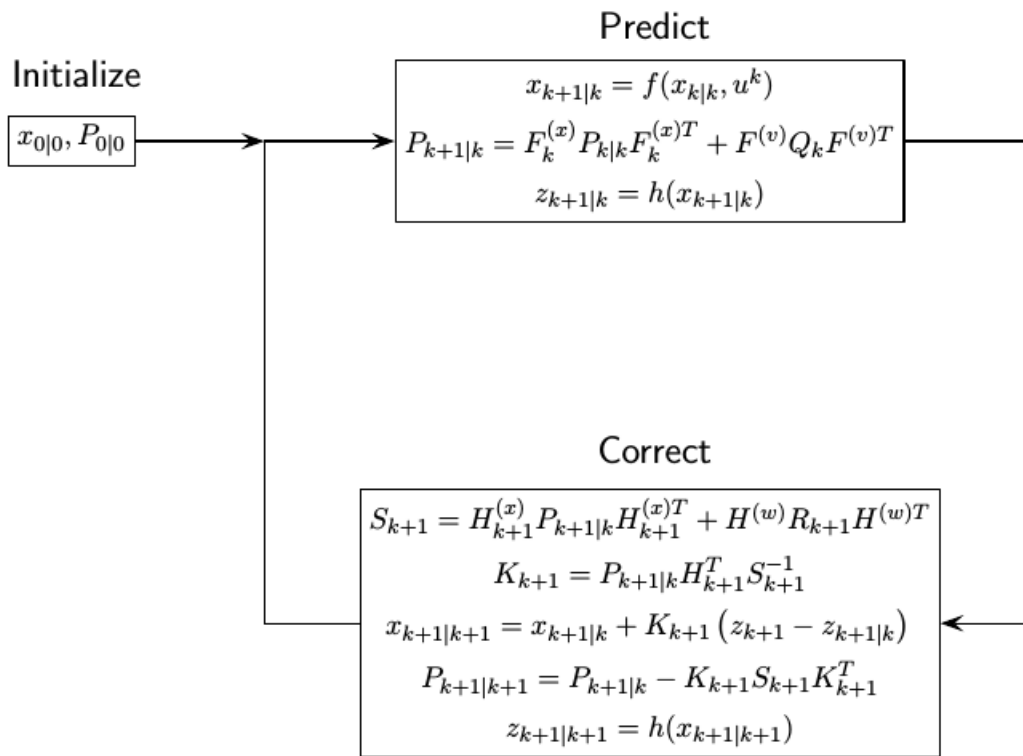
$$z_k = h(x_k, t) + v_k$$

In this case, $H^{(v)}$ is an identity matrix.

In `trackingEKF`, you can specify the measurement Jacobian matrix using the `MeasurementJacobianFcn` property. If you do not specify this property, the object computes the Jacobians using numeric differencing, which is slightly less accurate and can increase the computation time.

Extended Kalman Filter Loop

The extended Kalman filter loop is almost identical to the loop of "Linear Kalman Filters" on page 8-2 except that:

- The filter uses the exact nonlinear state update and measurement functions whenever possible.
- The state Jacobian replaces the state transition matrix.
- The measurement jacobian replaces the measurement matrix.

Predict

$$x_{k+1|k} = f(x_{k|k}, u^k)$$
$$P_{k+1|k} = F_k^{(x)} P_{k|k} F_k^{(x)T} + F^{(v)} Q_k F^{(v)T}$$
$$z_{k+1|k} = h(x_{k+1|k})$$

Initialize

$$x_{0|0}, P_{0|0}$$

Correct

$$S_{k+1} = H_{k+1}^{(x)} P_{k+1|k} H_{k+1}^{(x)T} + H^{(w)} R_{k+1} H^{(w)T}$$
$$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$$
$$x_{k+1|k+1} = x_{k+1|k} + K_{k+1} \left(z_{k+1} - z_{k+1|k} \right)$$
$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1} S_{k+1} K_{k+1}^T$$
$$z_{k+1|k+1} = h(x_{k+1|k+1})$$

Predefined Extended Kalman Filter Functions

The toolbox provides predefined state update and measurement functions to use in `trackingEKF`.

Motion Model	Function Name	Function Purpose	State Representation
Constant velocity	`constvel`	Constant-velocity state update model	• 1-D — `[x;vx]` • 2-D — `[x;vx;y;vy]` • 3-D — `[x;vx;y;vy;z;vz]` where • x, y, and z represents the position in the x-, y-, and z-directions, respectively. • vx, vy, and vz represents the velocity in the x-, y-, and z-directions, respectively.
	`constveljac`	Constant-velocity state update Jacobian	
	`cvmeas`	Constant-velocity measurement model	
	`cvmeasjac`	Constant-velocity measurement Jacobian	

Motion Model	Function Name	Function Purpose	State Representation
Constant acceleration	`constacc`	Constant-acceleration state update model	• 1-D — `[x;vx;ax]` • 2-D — `[x;vx;ax;y;vy;ay]` • 3-D — `[x;vx;ax;y;vy;ay;z;vz;az]` where • `ax`, `ay`, and `az` represents the acceleration in the *x*-, *y*-, and *z*-directions, respectively.
	`constaccjac`	Constant-acceleration state update Jacobian	
	`cameas`	Constant-acceleration measurement model	
	`cameasjac`	Constant-acceleration measurement Jacobian	
Constant turn rate	`constturn`	Constant turn-rate state update model	• 2-D — `[x;vx;y;vy;omega]` • 3-D — `[x;vx;y;vy;omega;z;vz]` where `omega` represents the turn-rate.
	`constturnjac`	Constant turn-rate state update Jacobian	
	`ctmeas`	Constant turn-rate measurement model	
	`ctmeasjac`	Constant turn-rate measurement Jacobian	

Example: Estimate 2-D Target States with Angle and Range Measurements Using trackingEKF

Initialize Estimation Model

Assume a target moves in 2D with the following initial position and velocity. The simulation lasts 20 seconds with a sample time of 0.2 seconds.

```
rng(2022);      % For repeatable results
dt = 0.2;       % seconds
simTime = 20; % seconds
tspan = 0:dt:simTime;
trueInitialState = [30; 1; 40; 1]; % [x;vx;y;vy]
initialCovariance = diag([100,1e3,100,1e3]);
processNoise = diag([0; .01; 0; .01]); % Process noise matrix
```

Assume the measurements are the azimuth angle relative to the positive-x direction and the range to from the origin to the target. The measurement noise covariance matrix is:

```
measureNoise = diag([2e-6;1]); % Measurement noise matrix. Units are m^2 and rad^2.
```

Preallocate variables in which to save results.

```
numSteps = length(tspan);
trueStates = NaN(4,numSteps);
trueStates(:,1) = trueInitialState;
estimateStates = NaN(size(trueStates));
measurements = NaN(2,numSteps);
```

Obtain True States and Measurements

Propagate the constant velocity model and generate the measurements with noise.

```
for i = 2:length(tspan)
    if i ~= 1
        trueStates(:,i) = stateModel(trueStates(:,i-1),dt) + sqrt(processNoise)*randn(4,1);
    end
    measurements(:,i) = measureModel(trueStates(:,i)) + sqrt(measureNoise)*randn(2,1);
end
```
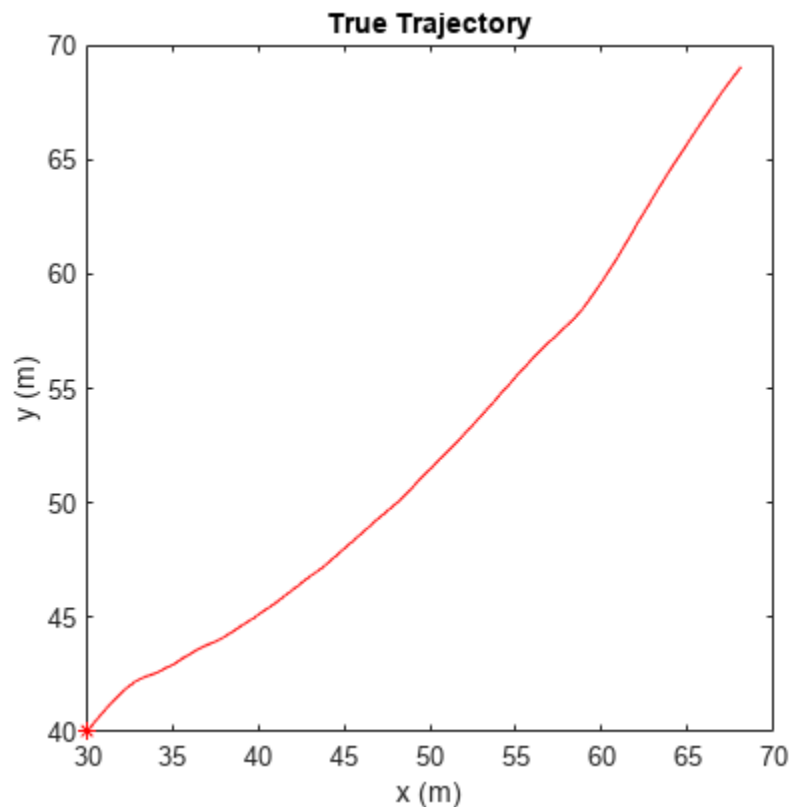
Plot the true trajectory and the measurements.

```
figure(1)
plot(trueStates(1,1),trueStates(3,1),"r*",DisplayName="Initial Truth")
hold on
plot(trueStates(1,:),trueStates(3,:),"r",DisplayName="True Trajectory")
xlabel("x (m)")
ylabel("y (m)")
title("True Trajectory")
axis square
```
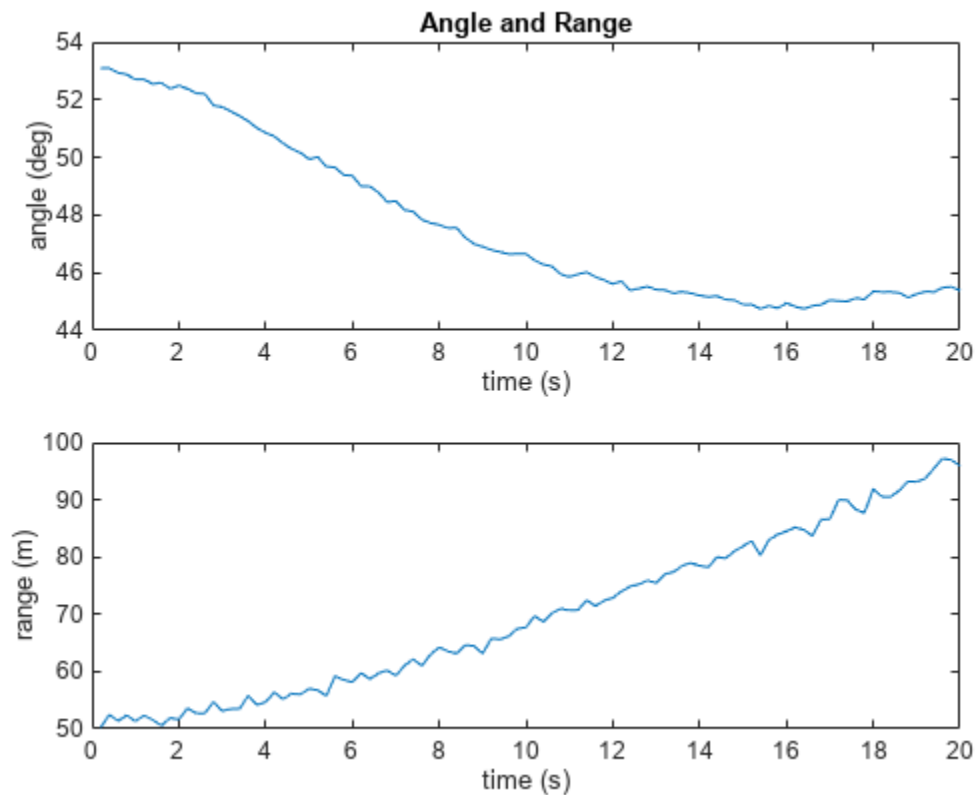
```
figure(2)
subplot(2,1,1)
plot(tspan,measurements(1,:)*180/pi)
xlabel("time (s)")
ylabel("angle (deg)")
title("Angle and Range")
subplot(2,1,2)
plot(tspan,measurements(2,:))
xlabel("time (s)")
ylabel("range (m)")
```


Initialize Extended Kalman Filter

Initialize the filter with an initial state estimate at `[35; 0; 45; 0]`.

```
filter = trackingEKF(State=[35; 0; 45; 0],StateCovariance=initialCovariance, ...
    StateTransitionFcn=@stateModel,ProcessNoise=processNoise, ...
    MeasurementFcn=@measureModel,MeasurementNoise=measureNoise);
estimateStates(:,1) = filter.State;
```

Run Extended Kalman Filter And Show Results

Run the filter by recursively calling the `predict` and `correct` object functions.
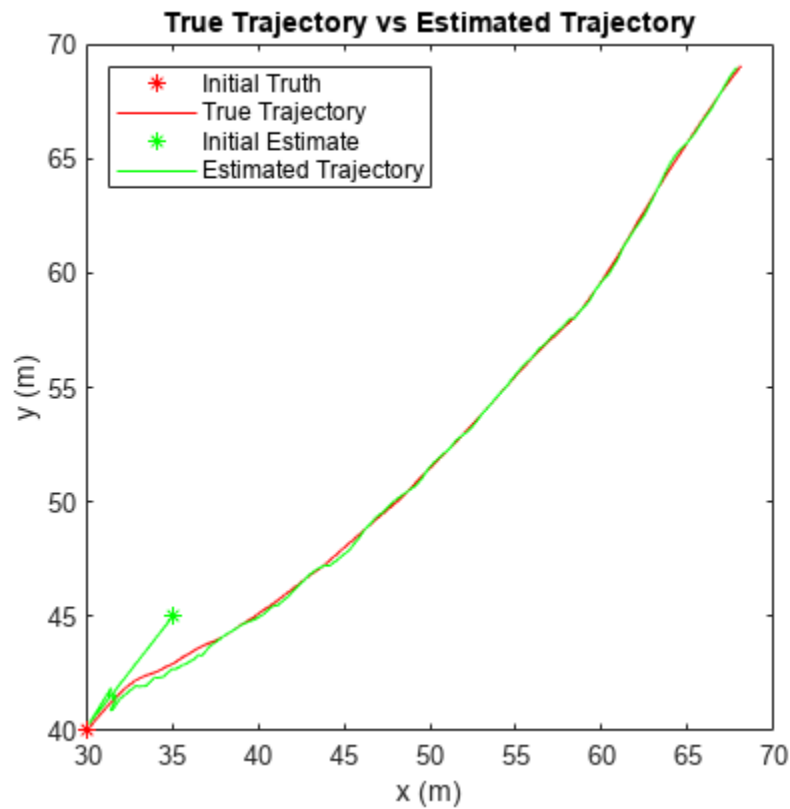
```
for i=2:length(tspan)
    predict(filter,dt);
    estimateStates(:,i) = correct(filter,measurements(:,i));
end
```

```
figure(1)
plot(estimateStates(1,1),estimateStates(3,1),"g*",DisplayName="Initial Estimate")
plot(estimateStates(1,:),estimateStates(3,:),"g",DisplayName="Estimated Trajectory")
legend(Location="northwest")
title("True Trajectory vs Estimated Trajectory")
```


Helper Functions

`stateModel` modeled constant velocity motion without process noise.

```
function stateNext = stateModel(state,dt)
    F = [1 dt 0  0;
         0  1 0  0;
         0  0 1 dt;
         0  0 0  1];
    stateNext = F*state;
end
```

`meausreModel` models range and azimuth angle measurements without noise.

```
function z = measureModel(state)
    angle = atan(state(3)/state(1));
    range = norm([state(1) state(3)]);
```

```
    z = [angle;range];
end
```

See Also

trackingKF | trackingEKF | trackingUKF | "Linear Kalman Filters" on page 8-2