Radar Toolbox

User's Guide

MATLAB® & SIMULINK®

R2021b

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

Radar Toolbox User's Guide

Trademarks

Patents

Revision History

March 2021	Online only	New for Version 1.0 (R2021a)
September 2021	Online only	Revised for Version 1.1 (R2021b)

1 Featured Examples

Radar Architecture: System Components and Requirements Allocation (Part 1) 1-3

Radar Architecture: Test Automation and Requirements Traceability (Part 2) 1-19

Benchmark Trajectories for Multi-Object Tracking 1-34

Simulate Radar Ghosts Due to Multipath Return 1-44

Highway Vehicle Tracking with Multipath Radar Reflections 1-61

Track-to-Track Fusion for Automotive Safety Applications 1-72

Track-to-Track Fusion for Automotive Safety Applications in Simulink . 1-87

Automotive Adaptive Cruise Control Using FMCW and MFSK Technology 1-90

Increasing Angular Resolution with Virtual Arrays 1-106

Patch Antenna Array for FMCW Radar 1-115

Simultaneous Range and Speed Estimation Using MFSK Waveform . . 1-126

Automotive Adaptive Cruise Control Using FMCW Technology 1-132

Radar Signal Simulation and Processing for Automated Driving 1-144

Adaptive Tracking of Maneuvering Targets with Managed Radar 1-155

Search and Track Scheduling for Multifunction Phased Array Radar . 1-172

PRF Agility Based on Target Detection 1-191

Interference Mitigation Using Frequency Agility Techniques 1-194

Frequency Agility in Radar, Communications, and EW Systems 1-202

Waveform Scheduling Based on Target Detection 1-210

Label Radar Signals with Signal Labeler 1-215

Pedestrian and Bicyclist Classification Using Deep Learning **1-227**

Radar Target Classification Using Machine Learning and Deep Learning **1-243**

Radar and Communications Waveform Classification Using Deep Learning **1-255**

Spaceborne Synthetic Aperture Radar Performance Prediction **1-268**

Airborne SAR System Design **1-290**

Stripmap Synthetic Aperture Radar (SAR) Image Formation **1-305**

Squinted Spotlight Synthetic Aperture Radar (SAR) Image Formation **1-316**

Synthetic Aperture Radar System Simulation and Image formation . . **1-325**

Processing Radar Reflections Acquired with the Demorad Radar Sensor Platform **1-330**

Detector Performance Analysis Using ROC Curves **1-337**

Monte-Carlo ROC Simulation **1-344**

Assessing Performance with the Tracker Operating Characteristic . . . **1-352**

Modeling Radar Detectability Factors **1-372**

MTI Improvement Factor for Land-Based Radar **1-390**

Radar Link Budget Analysis **1-406**

Planning Radar Network Coverage over Terrain **1-417**

Sea Clutter Simulation for Maritime Radar **1-429**

Radar Scenario Tutorial **1-447**

Use theaterPlot to Visualize Radar Scenario **1-457**

Simulate a Scanning Radar **1-461**

Simulate Passive Radar Sensors and Radar Interferences **1-477**

Introduction to Micro-Doppler Effects **1-495**

Ground Clutter Mitigation with Moving Target Indication (MTI) Radar **1-510**

Simulating a Polarimetric Radar Return for Weather Observation . . . **1-517**

Clutter and Jammer Mitigation with STAP **1-535**

Introduction to Space-Time Adaptive Processing **1-540**

Acceleration of Clutter Simulation Using GPU and Code Generation . **1-556**

Modeling Target Radar Cross Section **1-560**

Simulating a Bistatic Polarimetric Radar **1-573**

Simulating a Bistatic Radar with Two Targets **1-579**

Modeling a Wideband Monostatic Radar in a Multipath Environment **1-583**

Extended Target Tracking with Multipath Radar Reflections in Simulink **1-589**

Radar Vertical Coverage over Terrain **1-599**

Introduction to SAR Target Classification Using Deep Learning **1-608**

Automatic Target Recognition (ATR) in SAR Images **1-617**

Introduction to Pulse Integration and Fluctuation Loss in Radar **1-627**

Introduction to Scanning and Processing Losses in Pulse Radar **1-639**

Modeling Target Position Estimation Errors **1-654**

Modeling the Propagation of Radar Signals **1-665**

Display Micro-Doppler Shift of Moving Bicyclist **1-686**

Radar Performance Analysis Over Terrain **1-689**

Create Physics-Based Radar Model from Statistical Model **1-698**

Doppler Estimation **1-705**

Constant False Alarm Rate (CFAR) Detection **1-711**

Waveform Parameter Extraction from Received Pulse **1-720**

Lidar and Radar Fusion in an Urban Air Mobility Scenario **1-733**

2 Clutter

Clutter Modeling **2-2**
Surface Clutter Overview **2-2**
Approaches for Clutter Simulation or Analysis **2-2**
Considerations for Setting Up a Constant Gamma Clutter Simulation **2-2**
Related Examples **2-4**

3 Interference

Barrage Jammer **3-2**
Support for Modeling Barrage Jammer **3-2**
Model Barrage Jammer Output **3-2**
Model Effect of Barrage Jammer on Target Echo **3-3**

4 Radar Equation

Radar Equation **4-2**
Radar Equation Theory **4-2**
Plot Vertical Coverage Pattern Using Default Parameters **4-3**
Compute Peak Power Using Radar Equation Calculator App **4-4**

5 Model Platform Motion Using Trajectory Objects

Model Platform Motion Using Trajectory Objects **5-2**
Introduction **5-2**
waypointTrajectory **5-2**
geoTrajectory **5-5**
kinematicTrajectory **5-7**
Summary **5-13**

6 Kalman Filters

Linear Kalman Filters **6-2**
State Equations **6-2**
Measurement Models **6-3**
Linear Kalman Filter Equations **6-4**
Filter Loop **6-4**
Constant Velocity Model **6-5**
Constant Acceleration Model **6-6**

Extended Kalman Filters **6-7**
State Update Model **6-7**
Measurement Model **6-7**
Extended Kalman Filter Loop **6-8**
Predefined Extended Kalman Filter Functions **6-9**

1

Featured Examples

- “Radar Architecture: System Components and Requirements Allocation (Part 1)” on page 1-3
- “Radar Architecture: Test Automation and Requirements Traceability (Part 2)” on page 1-19
- “Benchmark Trajectories for Multi-Object Tracking” on page 1-34
- “Simulate Radar Ghosts Due to Multipath Return” on page 1-44
- “Highway Vehicle Tracking with Multipath Radar Reflections” on page 1-61
- “Track-to-Track Fusion for Automotive Safety Applications” on page 1-72
- “Track-to-Track Fusion for Automotive Safety Applications in Simulink” on page 1-87
- “Automotive Adaptive Cruise Control Using FMCW and MFSK Technology” on page 1-90
- “Increasing Angular Resolution with Virtual Arrays” on page 1-106
- “Patch Antenna Array for FMCW Radar” on page 1-115
- “Simultaneous Range and Speed Estimation Using MFSK Waveform” on page 1-126
- “Automotive Adaptive Cruise Control Using FMCW Technology” on page 1-132
- “Radar Signal Simulation and Processing for Automated Driving” on page 1-144
- “Adaptive Tracking of Maneuvering Targets with Managed Radar” on page 1-155
- “Search and Track Scheduling for Multifunction Phased Array Radar” on page 1-172
- “PRF Agility Based on Target Detection” on page 1-191
- “Interference Mitigation Using Frequency Agility Techniques” on page 1-194
- “Frequency Agility in Radar, Communications, and EW Systems” on page 1-202
- “Waveform Scheduling Based on Target Detection” on page 1-210
- “Label Radar Signals with Signal Labeler” on page 1-215
- “Pedestrian and Bicyclist Classification Using Deep Learning” on page 1-227
- “Radar Target Classification Using Machine Learning and Deep Learning” on page 1-243
- “Radar and Communications Waveform Classification Using Deep Learning” on page 1-255
- “Spaceborne Synthetic Aperture Radar Performance Prediction” on page 1-268
- “Airborne SAR System Design” on page 1-290
- “Stripmap Synthetic Aperture Radar (SAR) Image Formation” on page 1-305
- “Squinted Spotlight Synthetic Aperture Radar (SAR) Image Formation” on page 1-316
- “Synthetic Aperture Radar System Simulation and Image formation” on page 1-325
- “Processing Radar Reflections Acquired with the Demorad Radar Sensor Platform” on page 1-330
- “Detector Performance Analysis Using ROC Curves” on page 1-337
- “Monte-Carlo ROC Simulation” on page 1-344
- “Assessing Performance with the Tracker Operating Characteristic” on page 1-352
- “Modeling Radar Detectability Factors” on page 1-372
- “MTI Improvement Factor for Land-Based Radar” on page 1-390
- “Radar Link Budget Analysis” on page 1-406

- “Planning Radar Network Coverage over Terrain” on page 1-417
- “Sea Clutter Simulation for Maritime Radar” on page 1-429
- “Radar Scenario Tutorial” on page 1-447
- “Use theaterPlot to Visualize Radar Scenario” on page 1-457
- “Simulate a Scanning Radar” on page 1-461
- “Simulate Passive Radar Sensors and Radar Interferences” on page 1-477
- “Introduction to Micro-Doppler Effects” on page 1-495
- “Ground Clutter Mitigation with Moving Target Indication (MTI) Radar” on page 1-510
- “Simulating a Polarimetric Radar Return for Weather Observation” on page 1-517
- “Clutter and Jammer Mitigation with STAP” on page 1-535
- “Introduction to Space-Time Adaptive Processing” on page 1-540
- “Acceleration of Clutter Simulation Using GPU and Code Generation” on page 1-556
- “Modeling Target Radar Cross Section” on page 1-560
- “Simulating a Bistatic Polarimetric Radar” on page 1-573
- “Simulating a Bistatic Radar with Two Targets” on page 1-579
- “Modeling a Wideband Monostatic Radar in a Multipath Environment” on page 1-583
- “Extended Target Tracking with Multipath Radar Reflections in Simulink” on page 1-589
- “Radar Vertical Coverage over Terrain” on page 1-599
- “Introduction to SAR Target Classification Using Deep Learning ” on page 1-608
- “Automatic Target Recognition (ATR) in SAR Images” on page 1-617
- “Introduction to Pulse Integration and Fluctuation Loss in Radar” on page 1-627
- “Introduction to Scanning and Processing Losses in Pulse Radar” on page 1-639
- “Modeling Target Position Estimation Errors” on page 1-654
- “Modeling the Propagation of Radar Signals” on page 1-665
- “Display Micro-Doppler Shift of Moving Bicyclist” on page 1-686
- “Radar Performance Analysis Over Terrain” on page 1-689
- “Create Physics-Based Radar Model from Statistical Model” on page 1-698
- “Doppler Estimation” on page 1-705
- “Constant False Alarm Rate (CFAR) Detection” on page 1-711
- “Waveform Parameter Extraction from Received Pulse” on page 1-720
- “Lidar and Radar Fusion in an Urban Air Mobility Scenario” on page 1-733

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

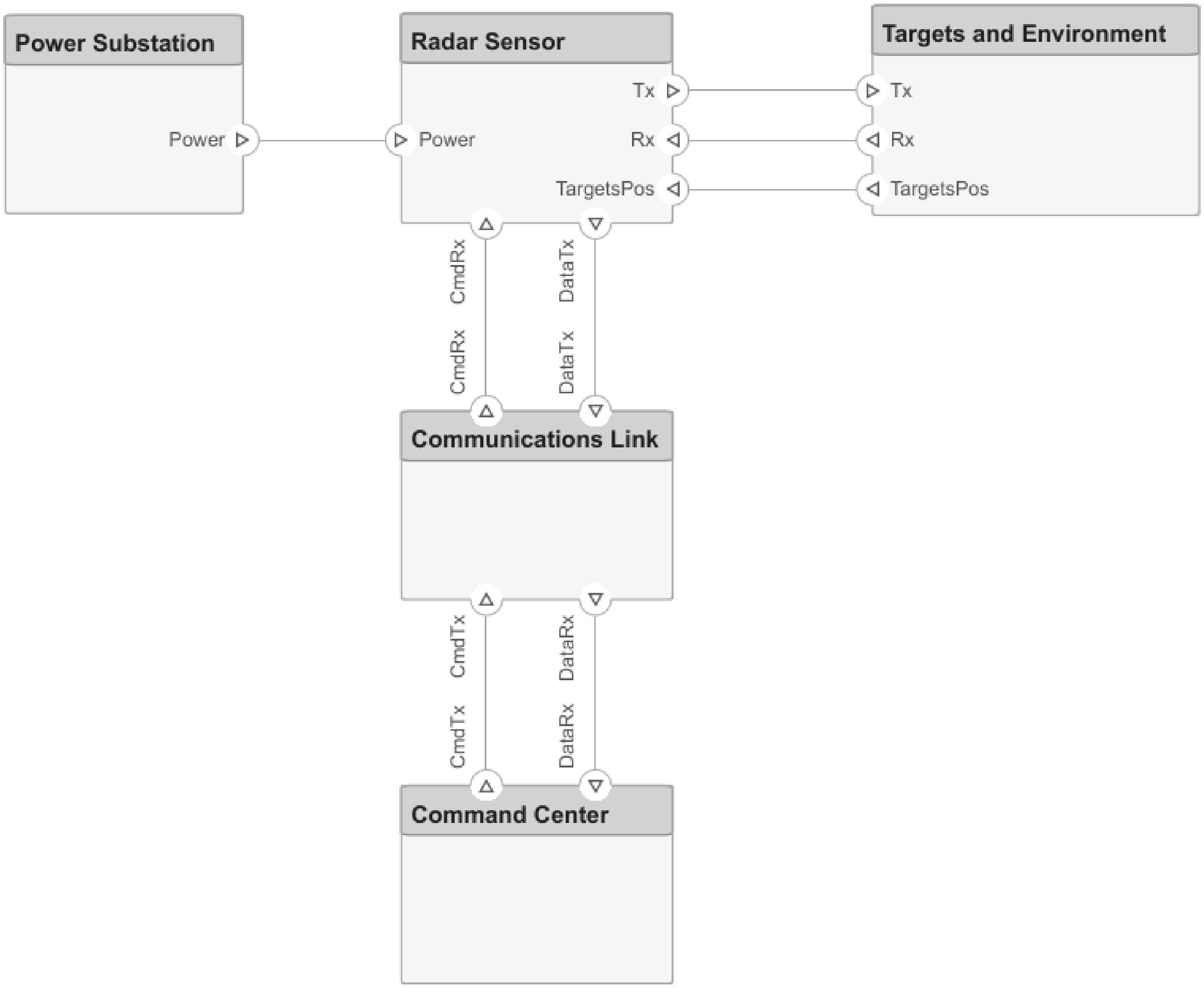
The architecture model specifies only the conceptual components of the system, their interfaces, and links between them. The components of the architecture model are not required to have a concrete implementation. As will be shown further in this example, System Composer allows for defining specific 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.

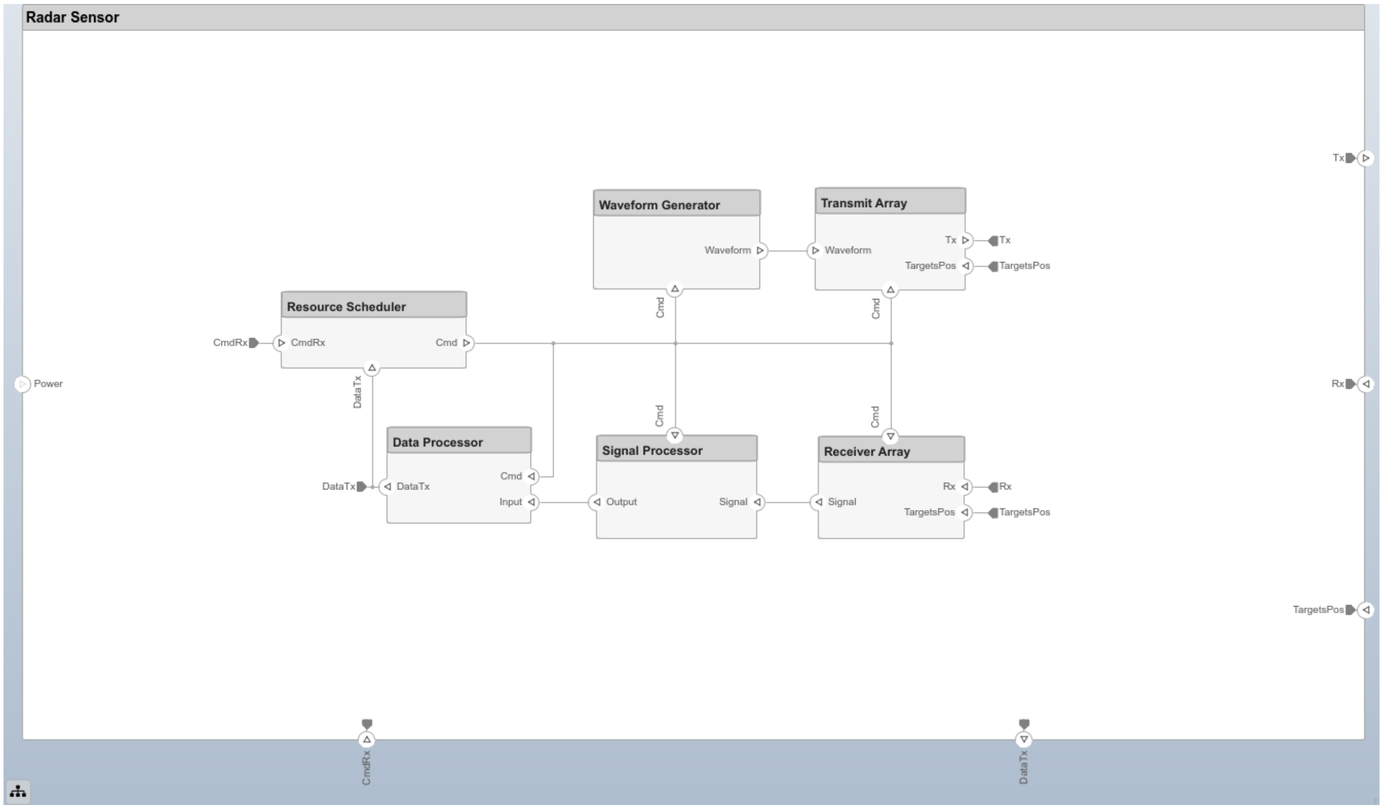
```
open_system('slexRadarArchitectureExample')
```



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`.

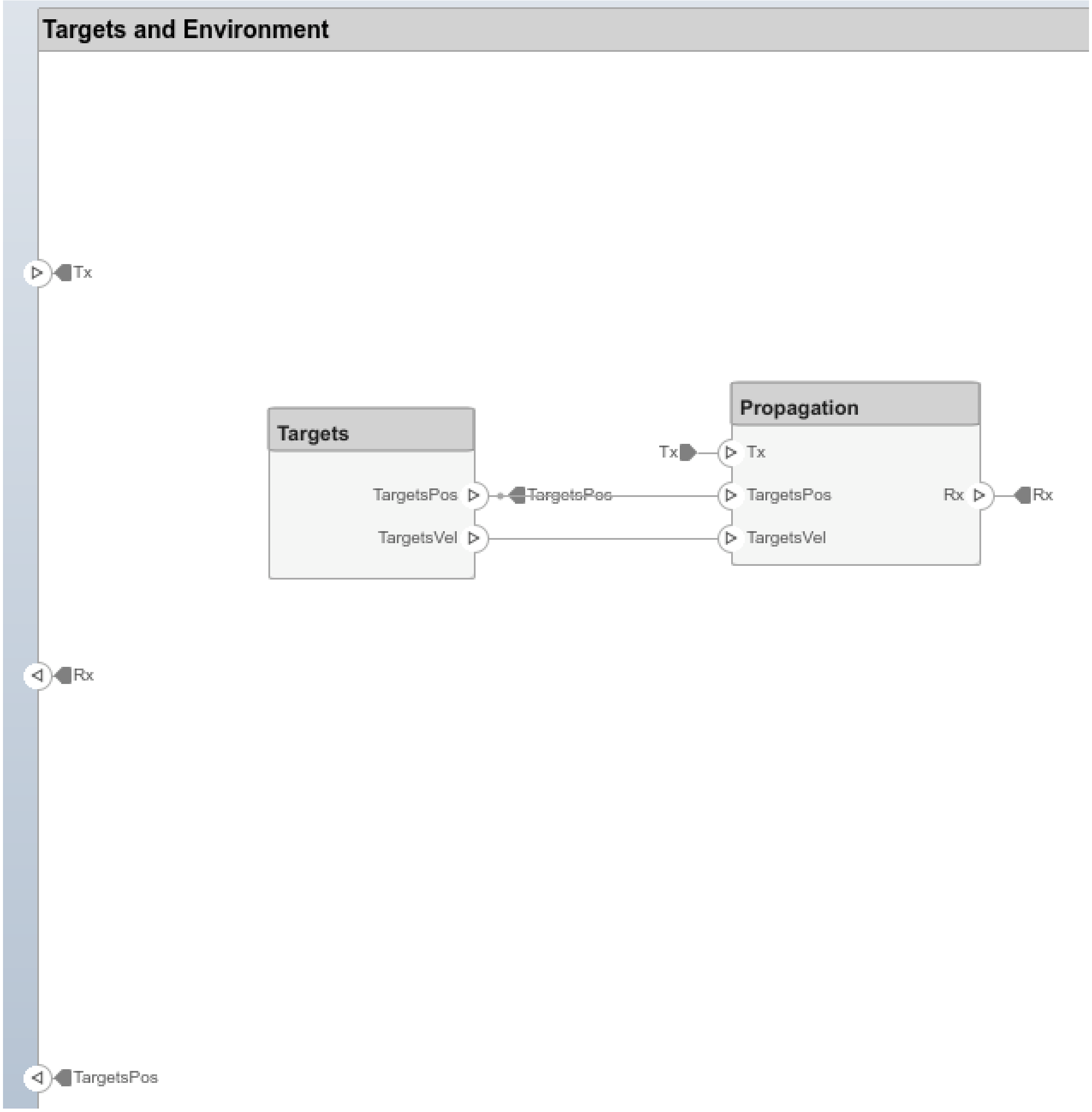
Targets and Environment

`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")
```

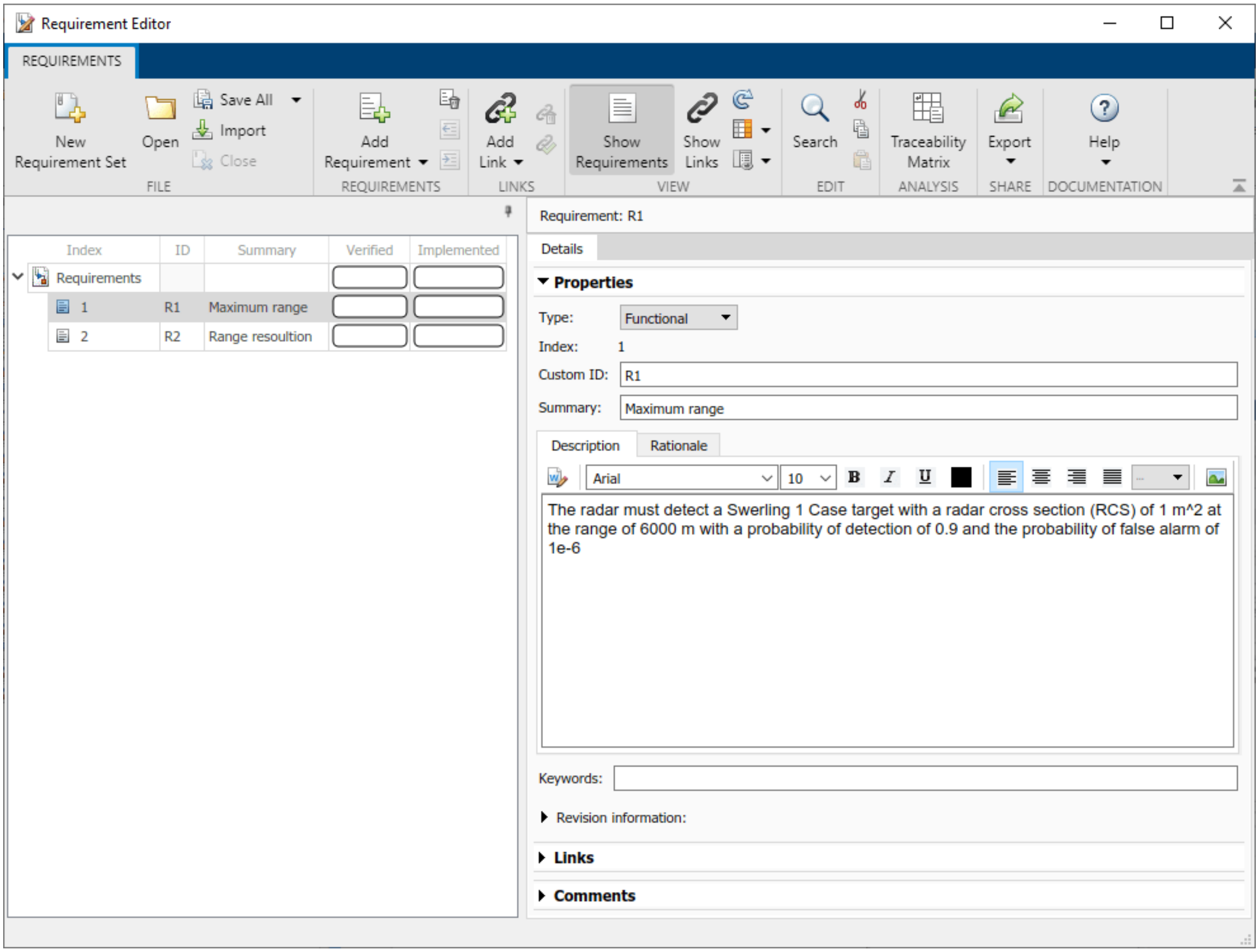


Requirements Traceability

Simulink Requirements™ 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, Simulink Requirements 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')
```



Index	ID	Summary	Verified	Implemented
Requirements				
1	R1	Maximum range		
2	R2	Range resoultion		

Requirement: R1

Type: Functional

Index: 1

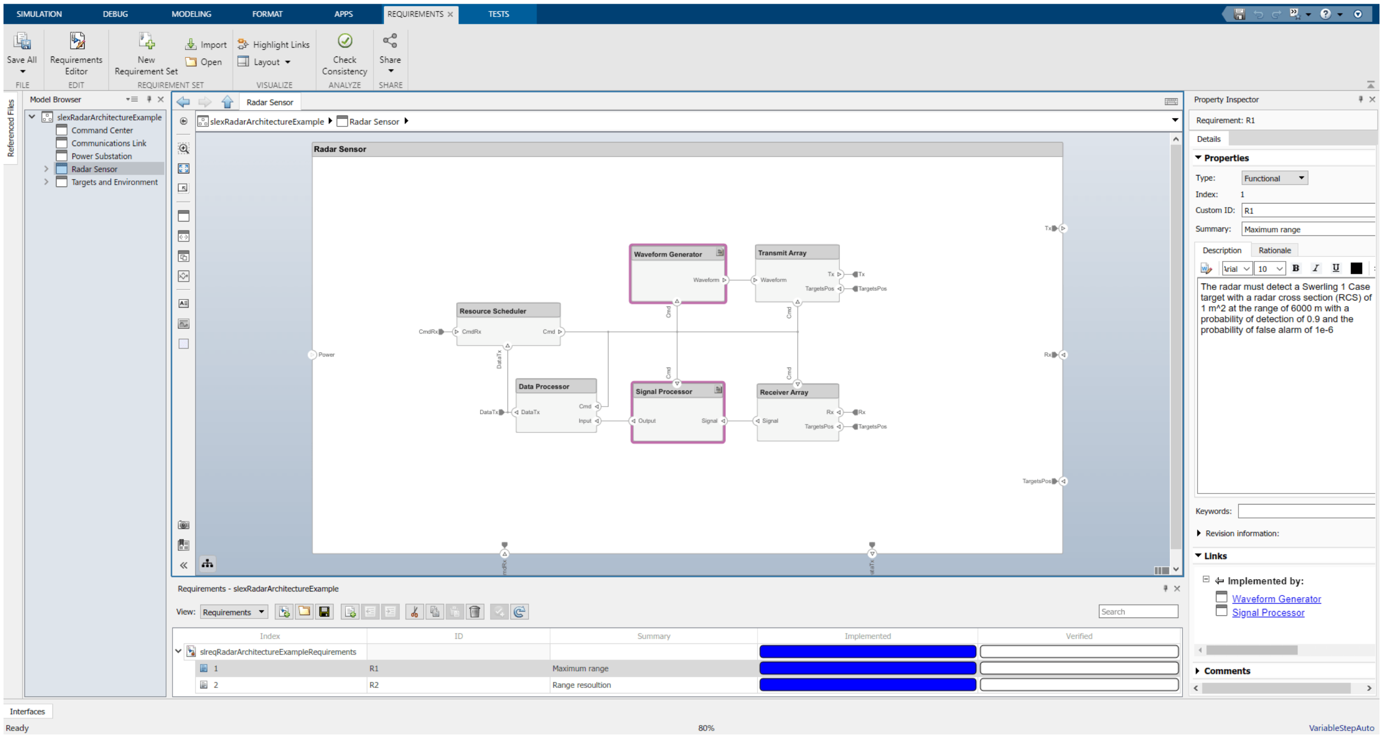
Custom ID: R1

Summary: Maximum range

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

Keywords:

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.



Requirement: R1

Type: Functional

Index: 1

Custom ID: R1

Summary: Maximum range

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

Keywords:

Implemented by:

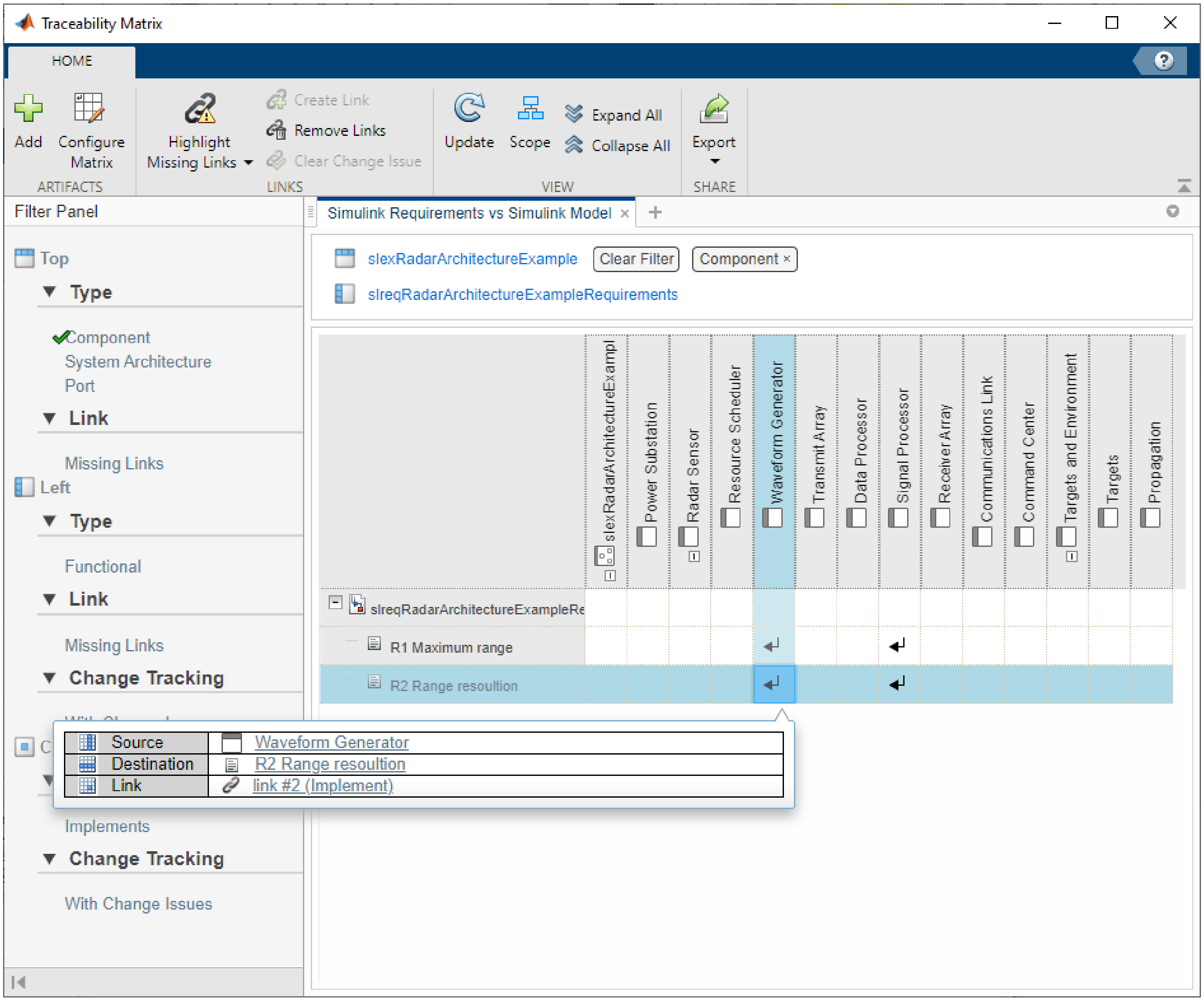
- Waveform Generator
- Signal Processor

Requirements - slexRadarArchitectureExample

View: Requirements

Index	ID	Summary	Implemented	Verified
slreqRadarArchitectureExampleRequirements				
1	R1	Maximum range		
2	R2	Range resoultion		

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.



Source	Waveform Generator
Destination	R2 Range resoultion
Link	link #2 (Implement)

Component Implementation

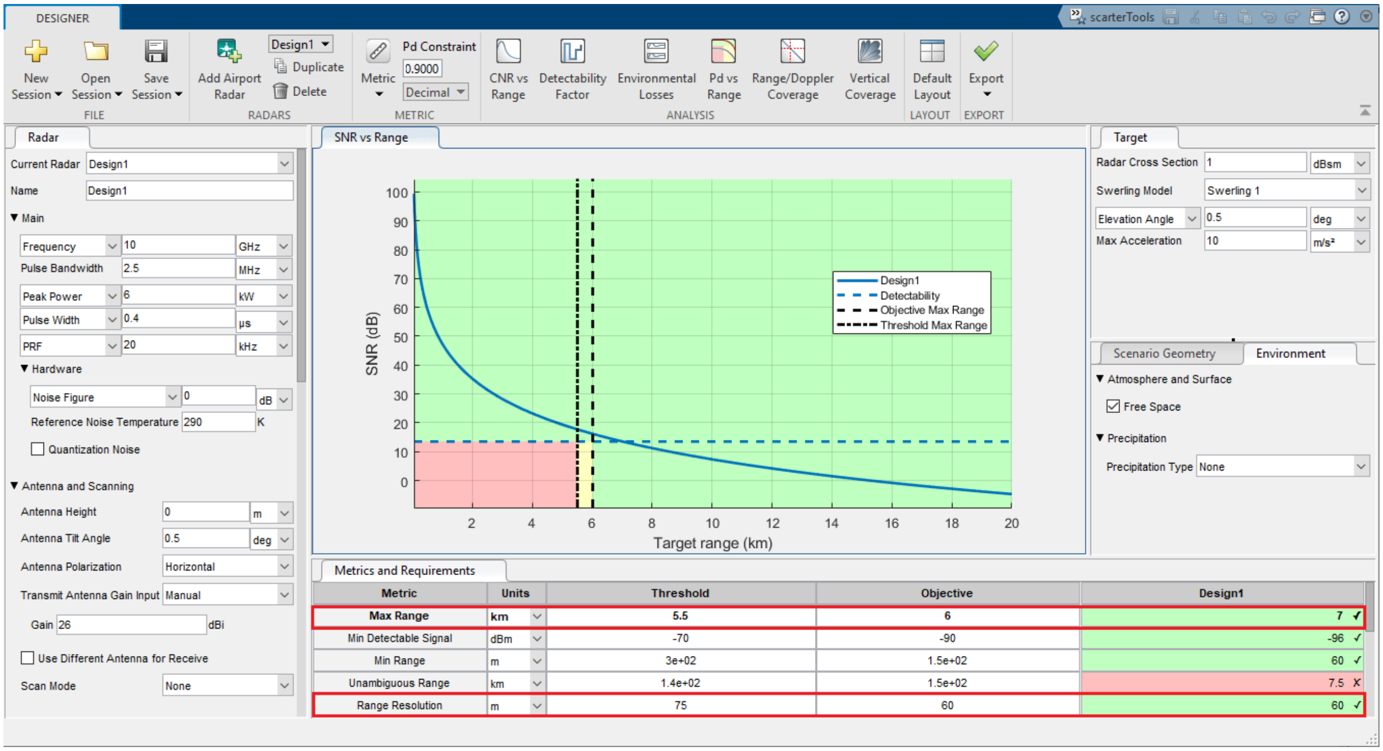
To simulate a radar system, provide a concrete behavior to the components of the architecture model. System Composer allows for you to specify the behavior of some components in Simulink, while leaving the behavior of other components undefined. This provides a lot of flexibility to the design and simulation 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;



Radar

Current Radar: Design1

Name: Design1

▼ Main

Frequency: 10 GHz

Pulse Bandwidth: 2.5 MHz

Peak Power: 6 kW

Pulse Width: 0.4 μs

PRF: 20 kHz

▼ Hardware

Noise Figure: 0 dB

Reference Noise Temperature: 290 K

□ Quantization Noise

▼ Antenna and Scanning

Antenna Height: 0 m

Antenna Tilt Angle: 0.5 deg

Antenna Polarization: Horizontal

Transmit Antenna Gain Input: Manual

Gain: 26 dBi

□ Use Different Antenna for Receive

Scan Mode: None

Target

Radar Cross Section: 1 dBsm

Swerling Model: Swerling 1

Elevation Angle: 0.5 deg

Max Acceleration: 10 m/s²

Scenario Geometry | Environment

▼ Atmosphere and Surface

☑ Free Space

▼ Precipitation

Precipitation Type: None

Metrics and Requirements

Metric	Units	Threshold	Objective	Design1
Max Range	**km**	**5.5**	**6**	**7 ✓**
Min Detectable Signal	dBm	-70	-90	-96 ✓
Min Range	m	3e+02	1.5e+02	60 ✓
Unambiguous Range	km	1.4e+02	1.5e+02	7.5 X
Range Resolution	m	75	60	60 ✓

Open this design in Radar Designer app.

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

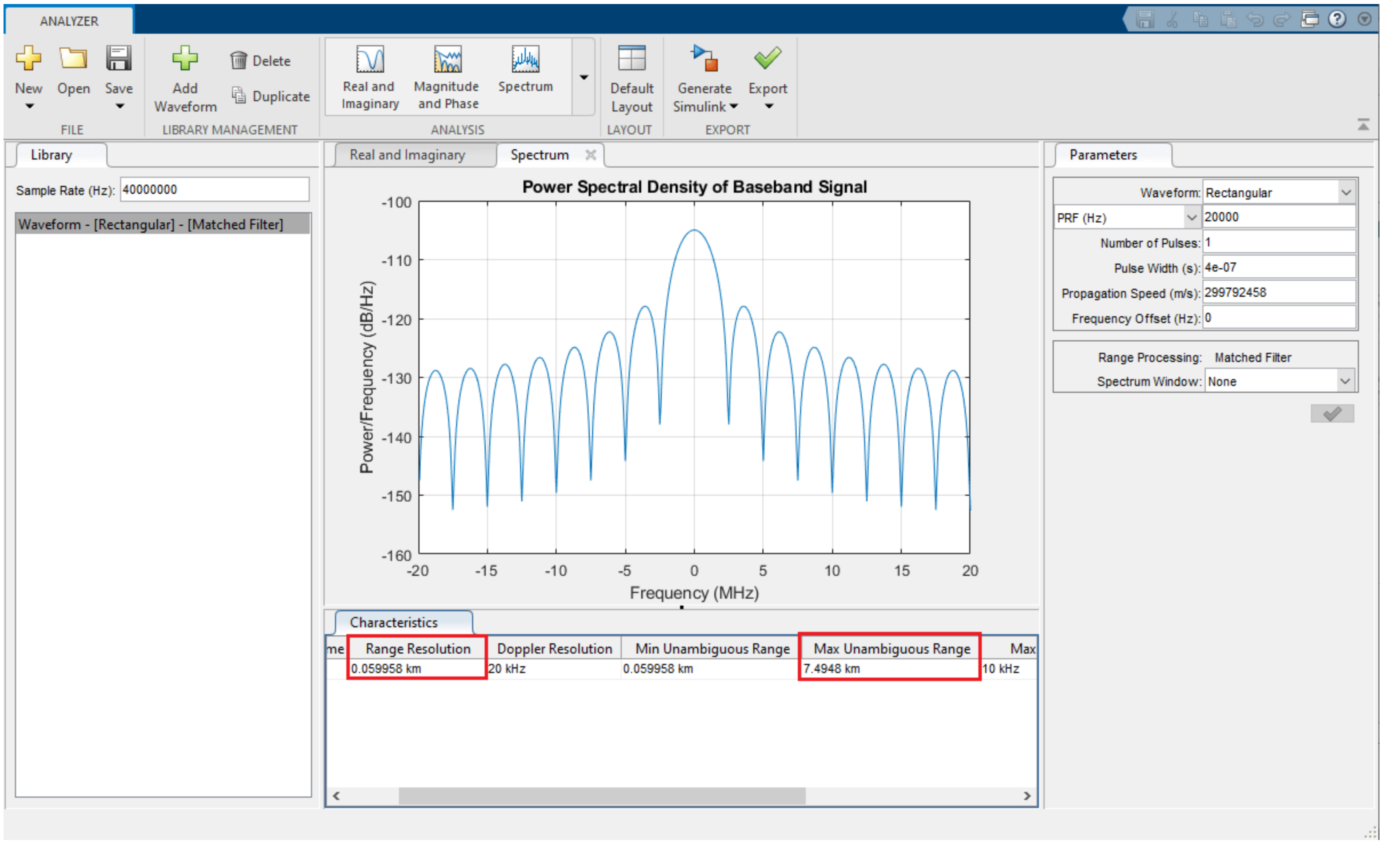
Waveform Generator

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.



Sample Rate (Hz): 40000000

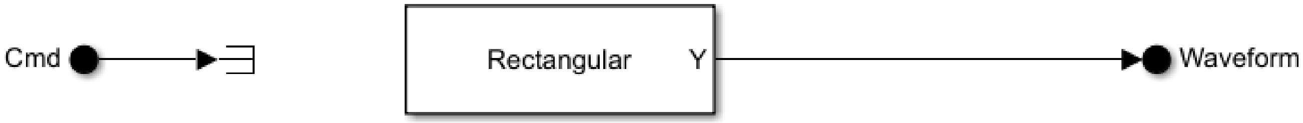
Waveform - [Rectangular] - [Matched Filter]

Waveform: Rectangular
PRF (Hz): 20000
Number of Pulses: 1
Pulse Width (s): 4e-07
Propagation Speed (m/s): 299792458
Frequency Offset (Hz): 0

Range Processing: Matched Filter
Spectrum Window: None

me	Range Resolution	Doppler Resolution	Min Unambiguous Range	Max Unambiguous Range	Max
	0.059958 km	20 kHz	0.059958 km	7.4948 km	10 kHz

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.

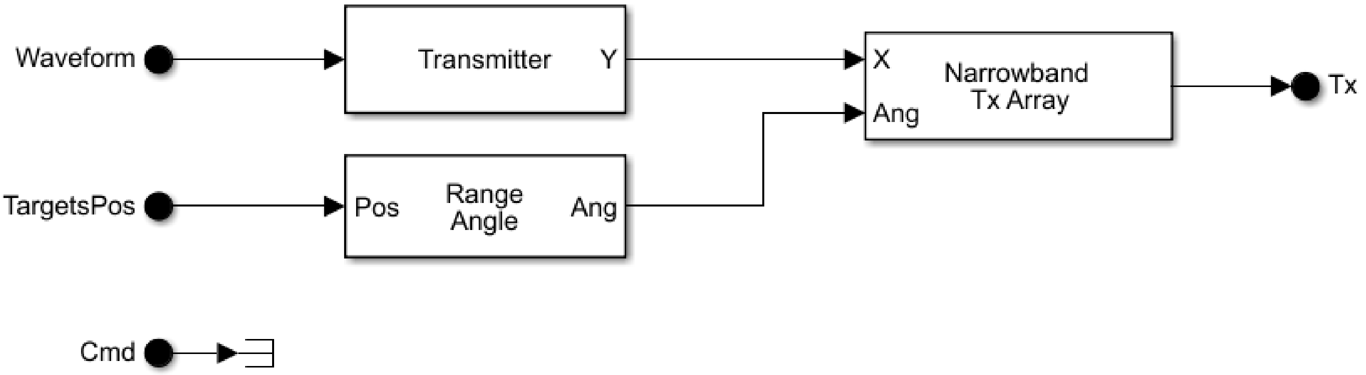


Set the Output signal format property of the block to Pulses. This means that every pulse repetition interval (PRI) of 1/prf seconds, the block produces a column vector of fs/prf complex waveform samples.

Transmit Array

The Transmit Array component comprises the following Simulink blocks:

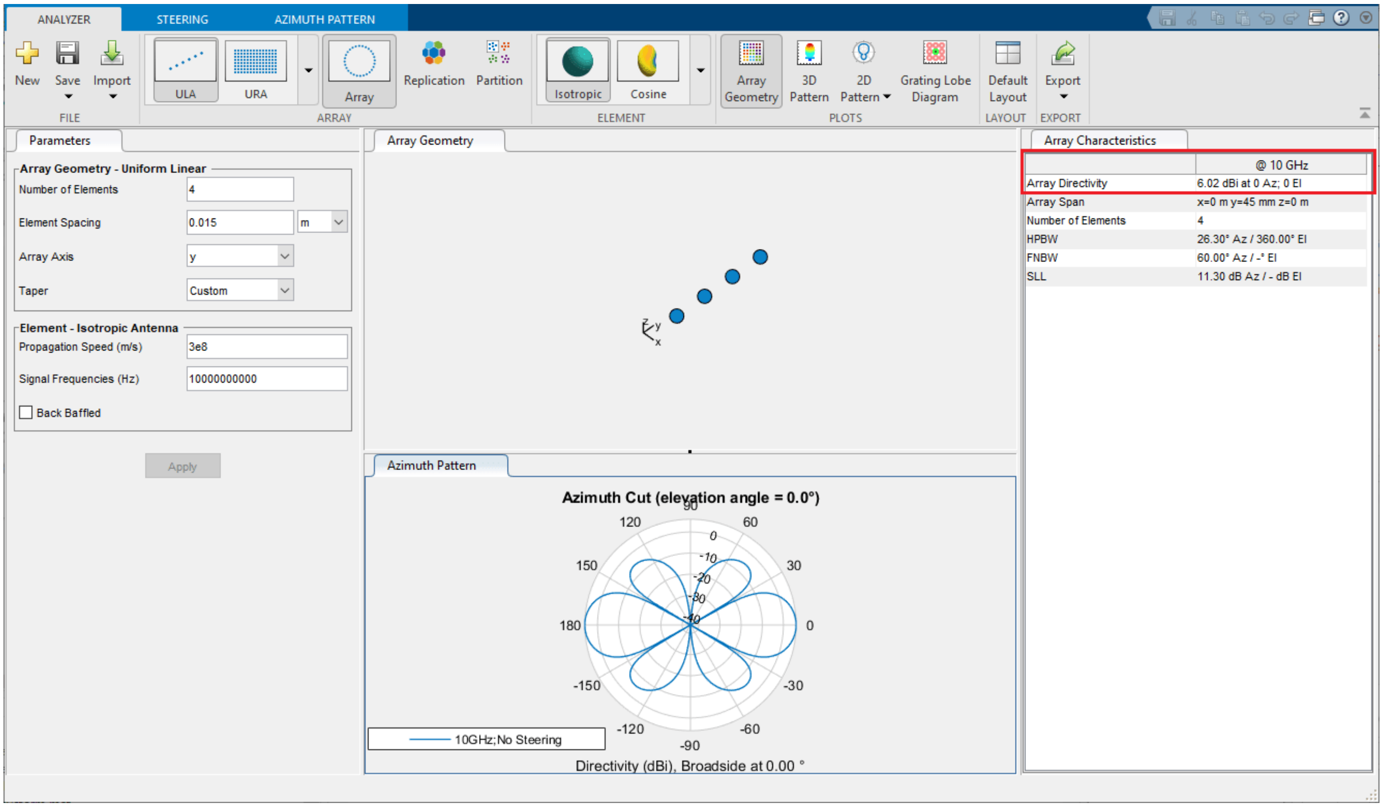
- Transmitter — Transmits the waveform generated by Waveform Generator with the specified peak power and transmit gain.
- Range Angle Calculator — Computes the directions towards the targets assuming the radar is placed on static platform located at the origin. The target directions are used as Ang input 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')
```



Array Geometry - Uniform Linear

Number of Elements	4
Element Spacing	0.015 m
Array Axis	y
Taper	Custom

Element - Isotropic Antenna

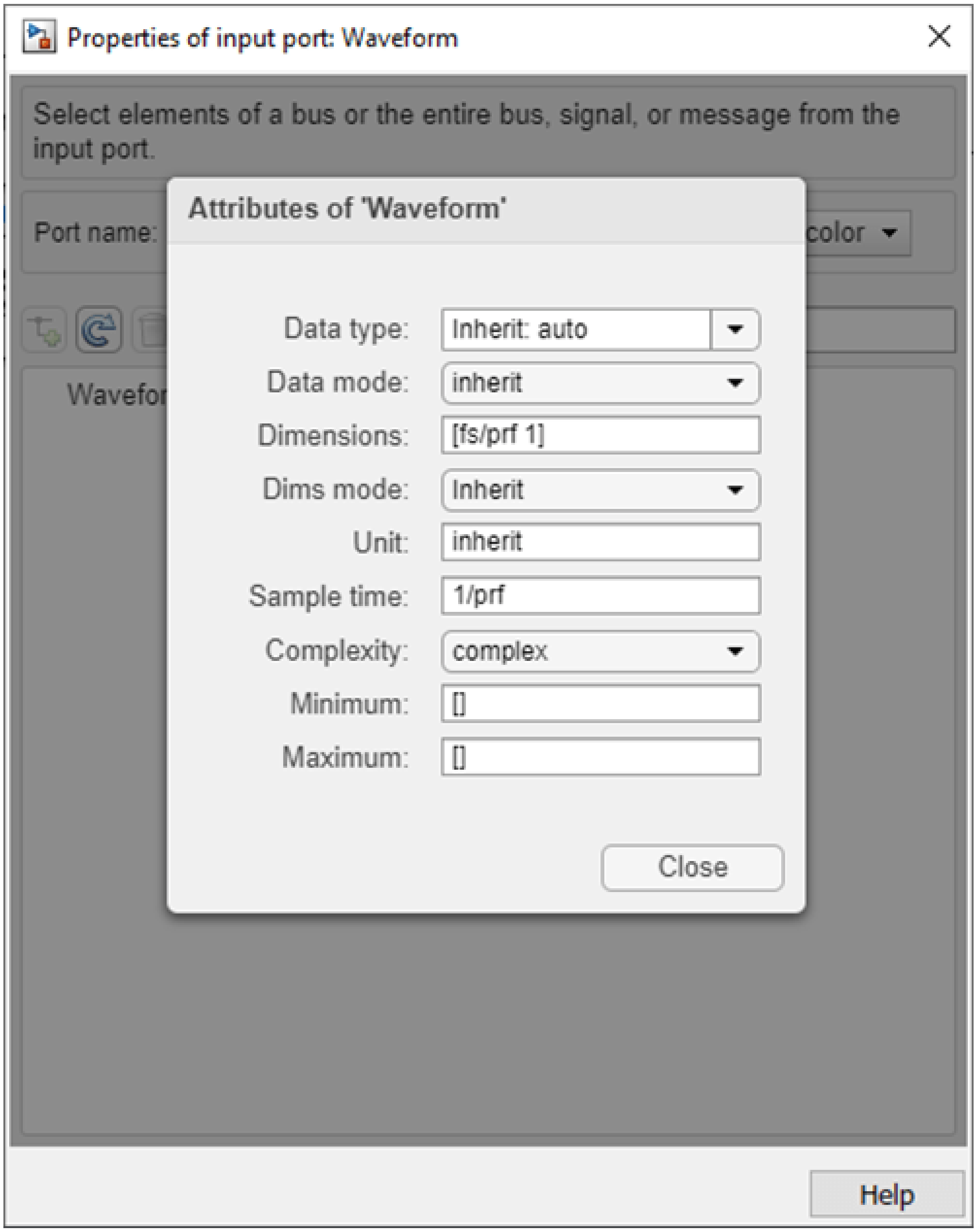
Propagation Speed (m/s)	3e8
Signal Frequencies (Hz)	10000000000

□ Back Baffled

Apply

	@ 10 GHz
Array Directivity	6.02 dBi at 0 Az; 0 El
Array Span	x=0 m y=45 mm z=0 m
Number of Elements	4
HPBW	26.30° Az / 360.00° El
FNBW	60.00° Az / -° El
SLL	11.30 dB Az / - dB El

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.



Properties of input port: Waveform

Select elements of a bus or the entire bus, signal, or message from the input port.

Port name:

Attributes of 'Waveform'

Data type: Inherit: auto

Data mode: inherit

Dimensions: [fs/prf 1]

Dims mode: Inherit

Unit: inherit

Sample time: 1/prf

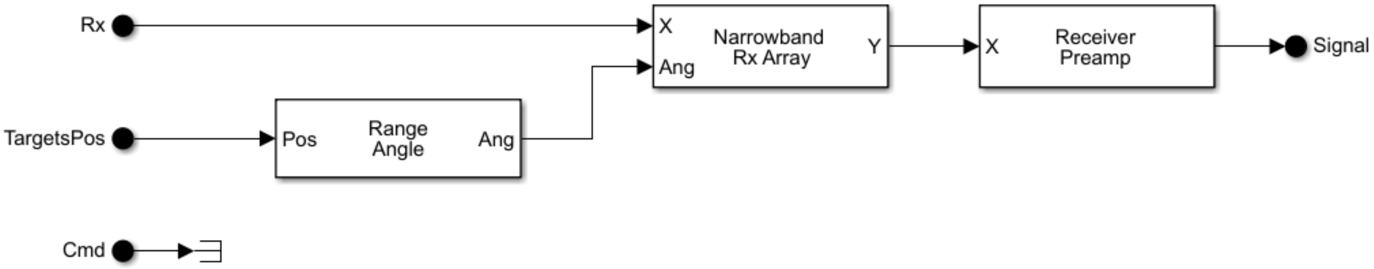
Complexity: complex

Minimum: []

Maximum: []

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.

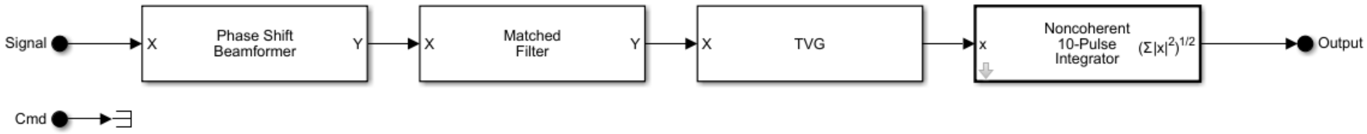


The `Rx` input is a matrix of received waveform samples with columns corresponding to `size(tgtpos,2)` targets. The dimensions of `Rx` must be set to [fs/prf `size(tgtpos,2)`], the sample time to `1/prf`, and the complexity to `'complex'`.

Signal Processor

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

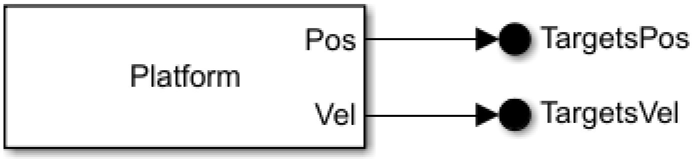
- `Phase Shift Beamformer` — Combines the received signals at each array element. This example sets the beamforming direction to the broadside.
- `Matched Filter` — Performs matched filtering to improve SNR. The coefficients of the matched filter are set to match the transmitted waveform.
- `Time Varying 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

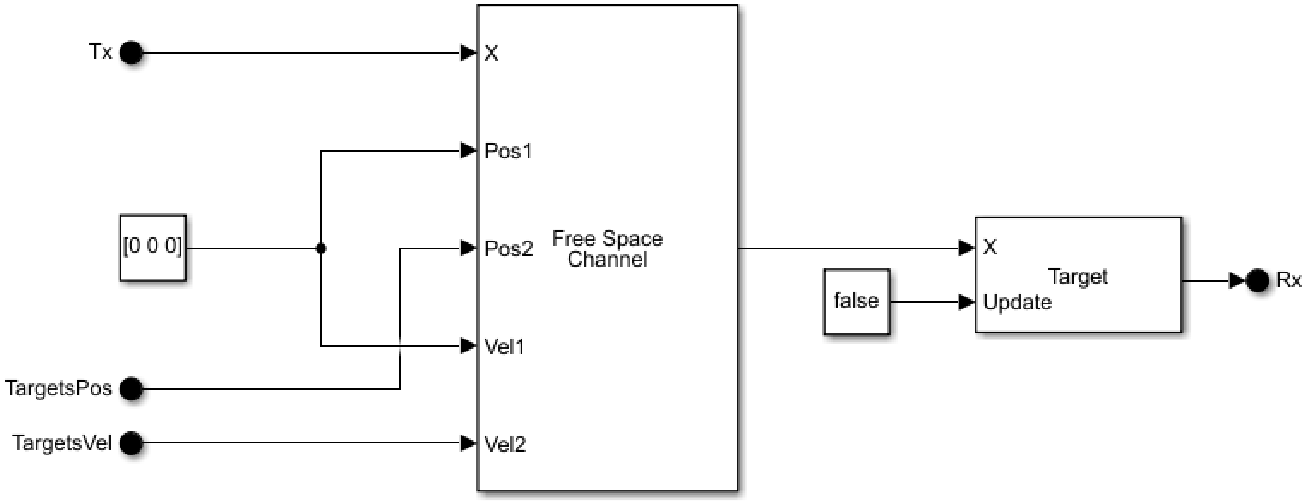
The `Targets` component is implemented using a single `Platform` block.



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

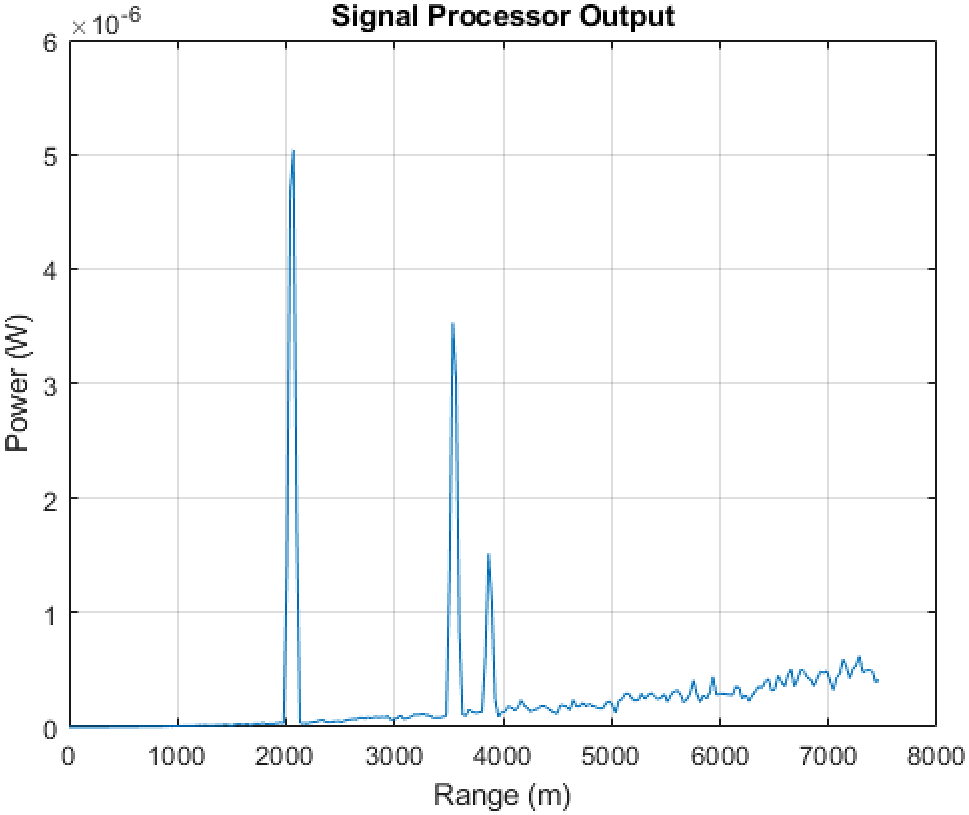
Specifying these blocks in Simulink is enough to obtain a model of a radar system that can produce radar detections. Prior to proceeding with testing the model and verifying the specific performance requirements, run the simulation and check whether it generates the results as expected. Consider three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

```
% 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 Simulink Requirements™ to link the requirements to the components of the architecture. It also shows how to implement the individual components of the architecture using Simulink.

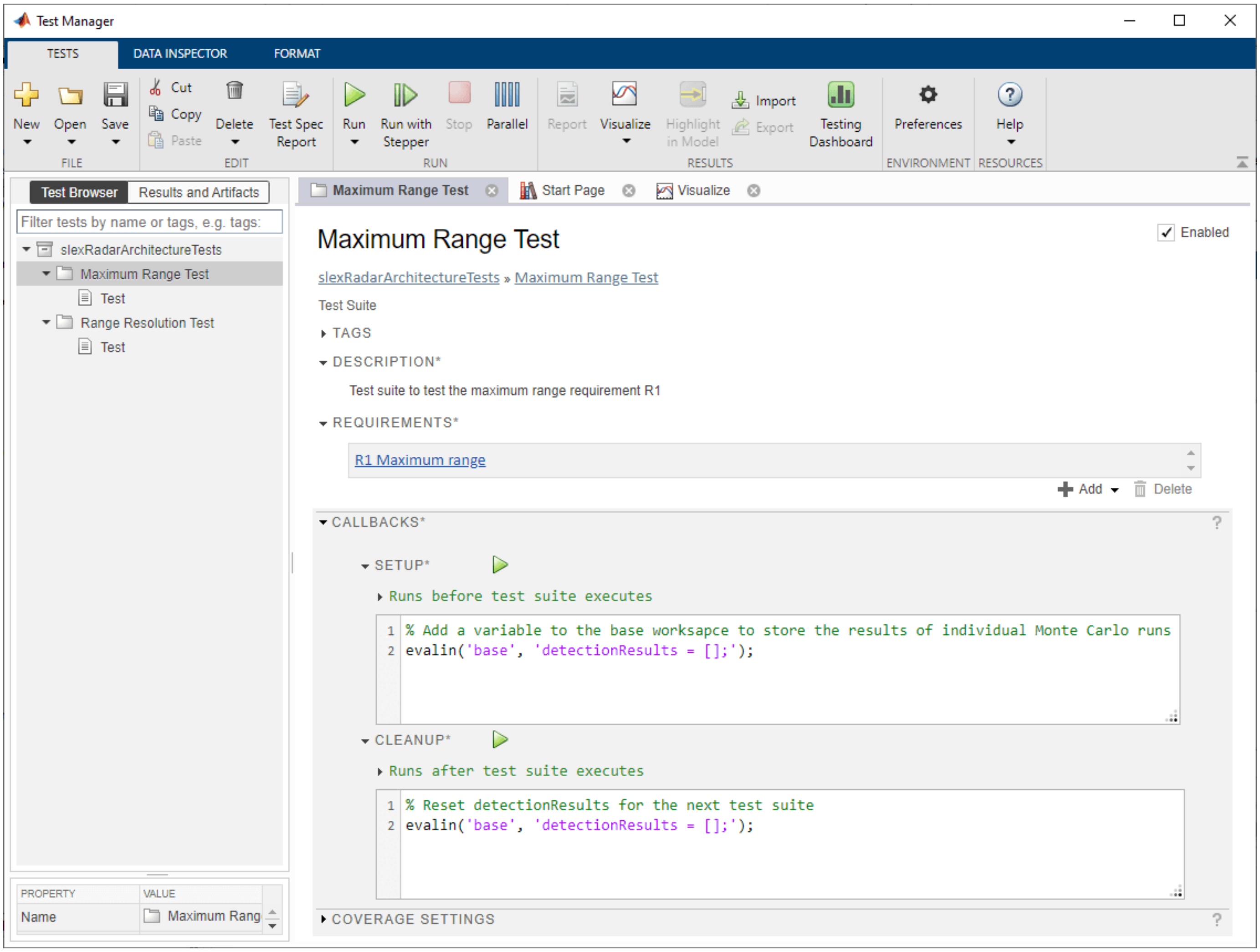
Automated Testing

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.



Maximum Range Test

slexRadarArchitectureTests » Maximum Range Test

Test Suite

Test suite to test the maximum range requirement R1

R1 Maximum range

Runs before test suite executes

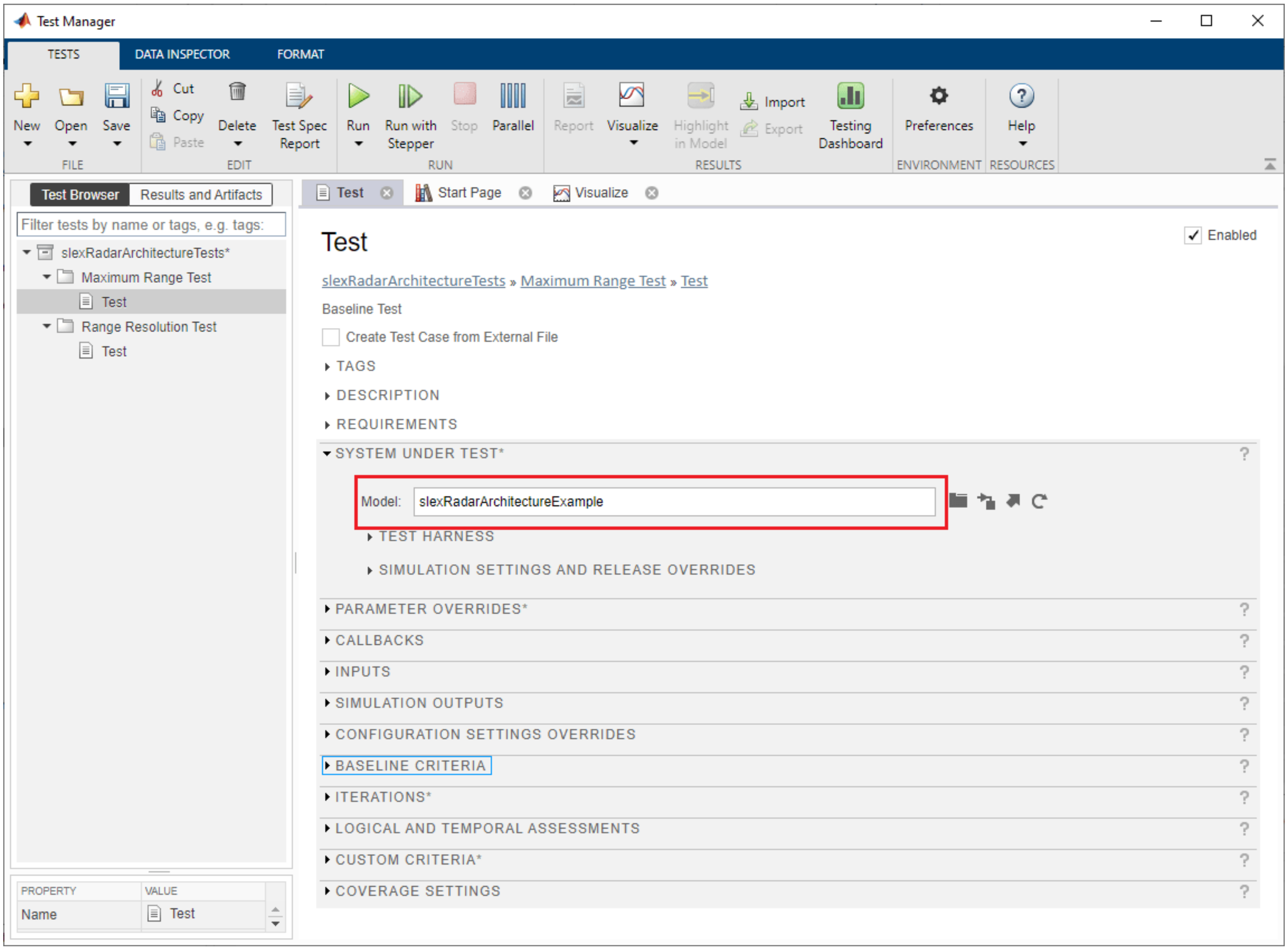
```
% Add a variable to the base worksapce to store the results of individual Monte Carlo runs
evalin('base', 'detectionResults = [];');
```

Runs after test suite executes

```
% Reset detectionResults for the next test suite
evalin('base', 'detectionResults = [];');
```

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*.



- slexRadarArchitectureTests*
  - Maximum Range Test
    - Test
  - Range Resolution Test
    - Test

PROPERTY	VALUE
Name	Test

Test

☑ Enabled

slexRadarArchitectureTests » Maximum Range Test » Test

Baseline Test

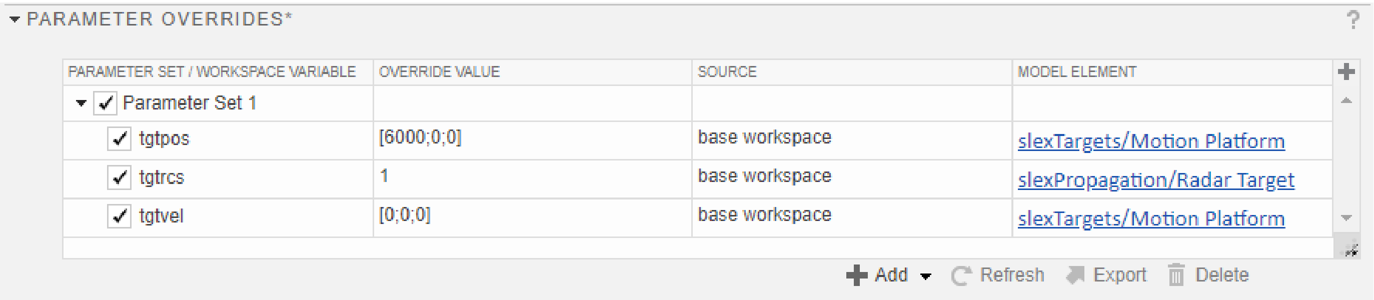
□ Create Test Case from External File

SYSTEM UNDER TEST*

Model: slexRadarArchitectureExample

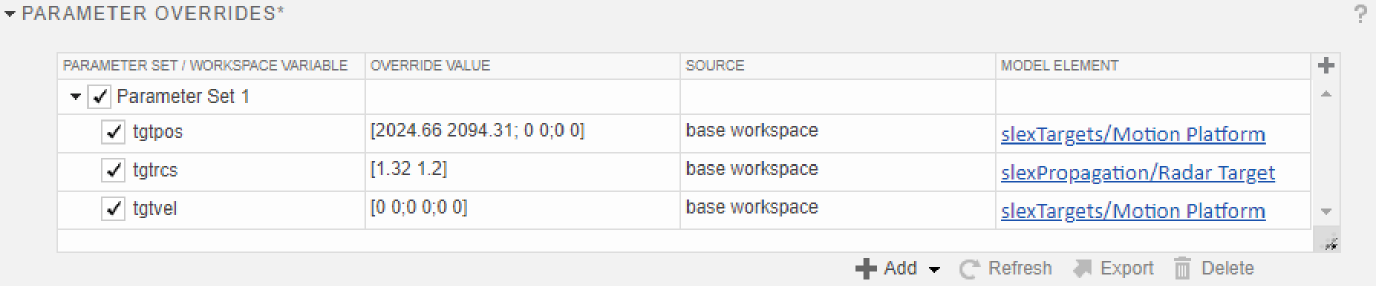
- The Parameter Overrides section is used to assign different values to the parameters in the base workspace during a test execution. Use this section to specify the targets parameters for the maximum range test and the range resolution test.

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.



PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
☑ Parameter Set 1			
☑ tgtpos	[6000;0;0]	base workspace	slexTargets/Motion Platform
☑ tgtrcs	1	base workspace	slexPropagation/Radar Target
☑ tgtvel	[0;0;0]	base workspace	slexTargets/Motion Platform

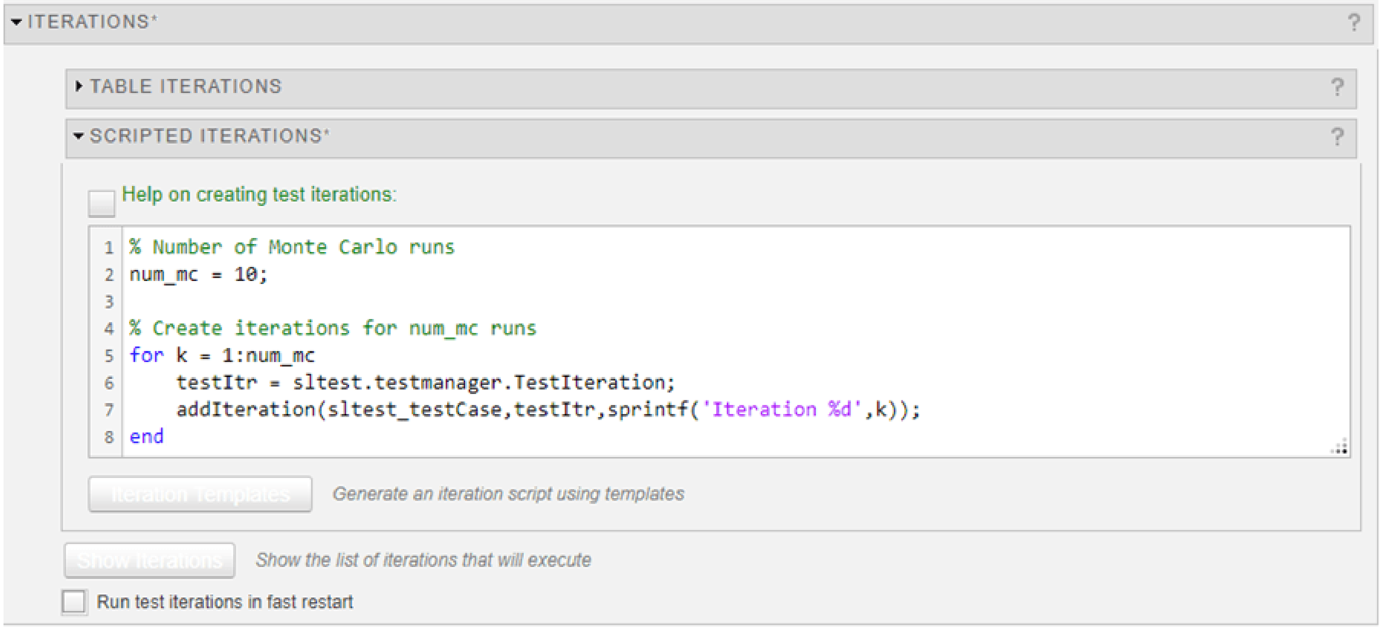
For the range resolution test, specify two targets with different RCS separated in range by 70 m as required by R2.



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

- 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.



□ Help on creating test iterations:

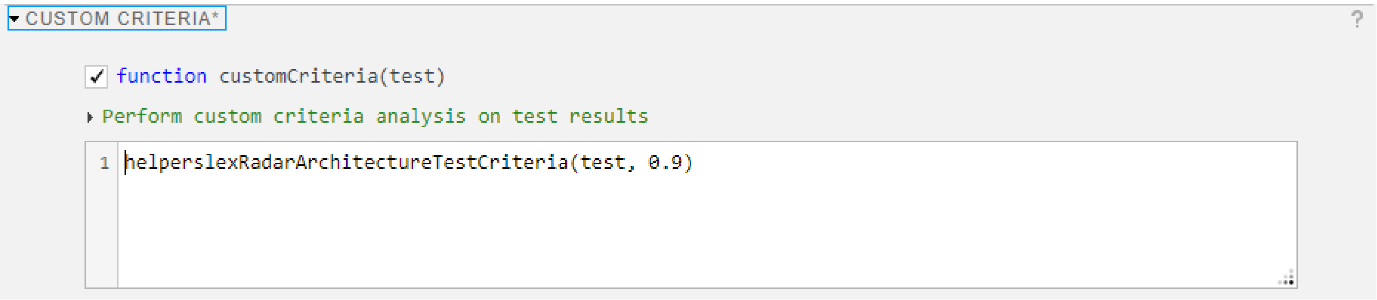
```
% Number of Monte Carlo runs
num_mc = 10;

% Create iterations for num_mc runs
for k = 1:num_mc
    testItr = sltest.testmanager.TestIteration;
    addIteration(sltest_testCase,testItr,sprintf('Iteration %d',k));
end
```

Generate an iteration script using templates

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.

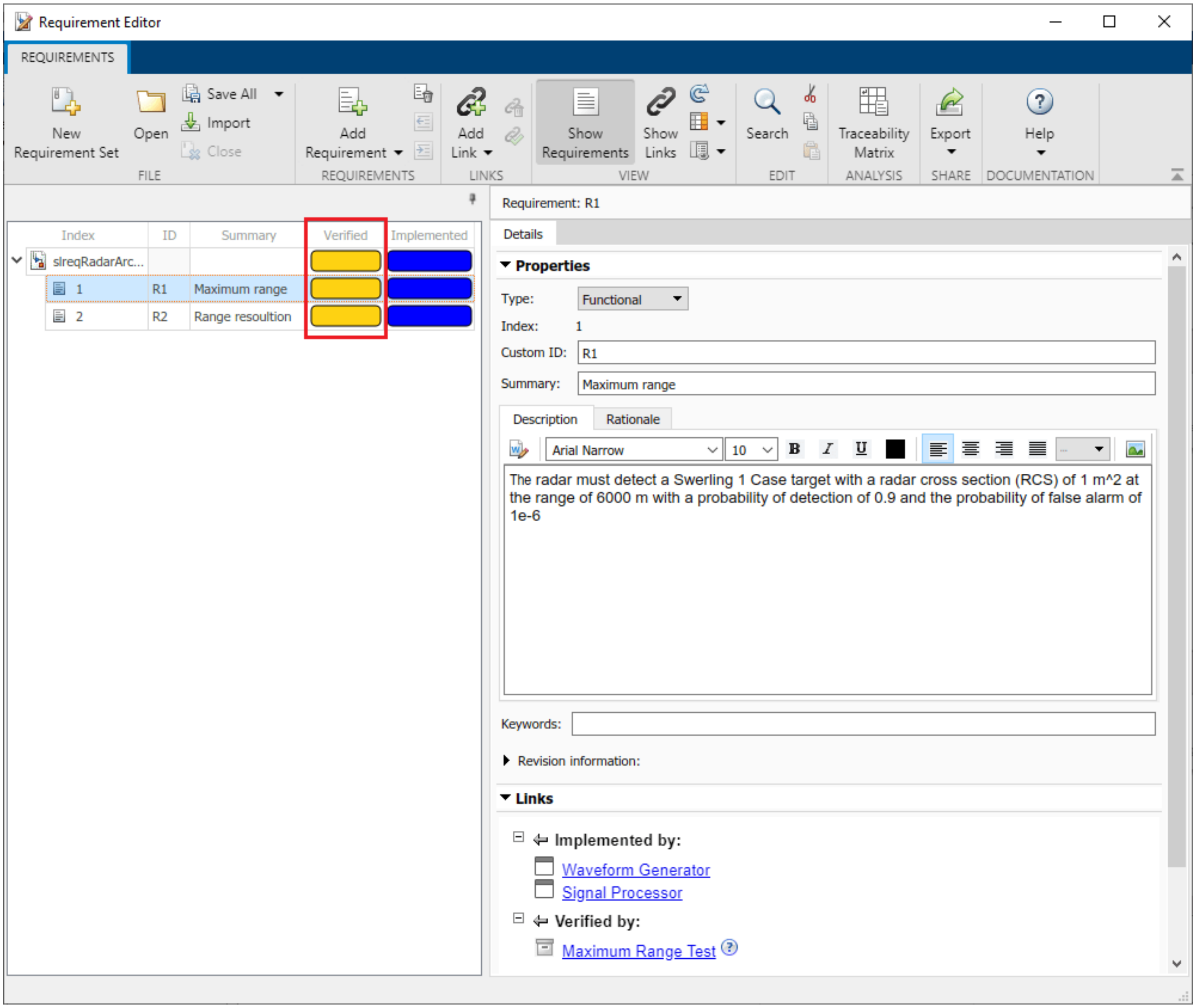


```
☑ function customCriteria(test)
Perform custom criteria analysis on test results
helperslexRadarArchitectureTestCriteria(test, 0.9)
```

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.



Index	ID	Summary	Verified	Implemented
slreqRadarArc...			🟨	🟦
1	R1	Maximum range	🟨	🟦
2	R2	Range resoultion	🟨	🟦

Requirement: R1

Properties

Type: Functional

Index: 1

Custom ID: R1

Summary: Maximum range

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

Keywords:

Revision information:

Links

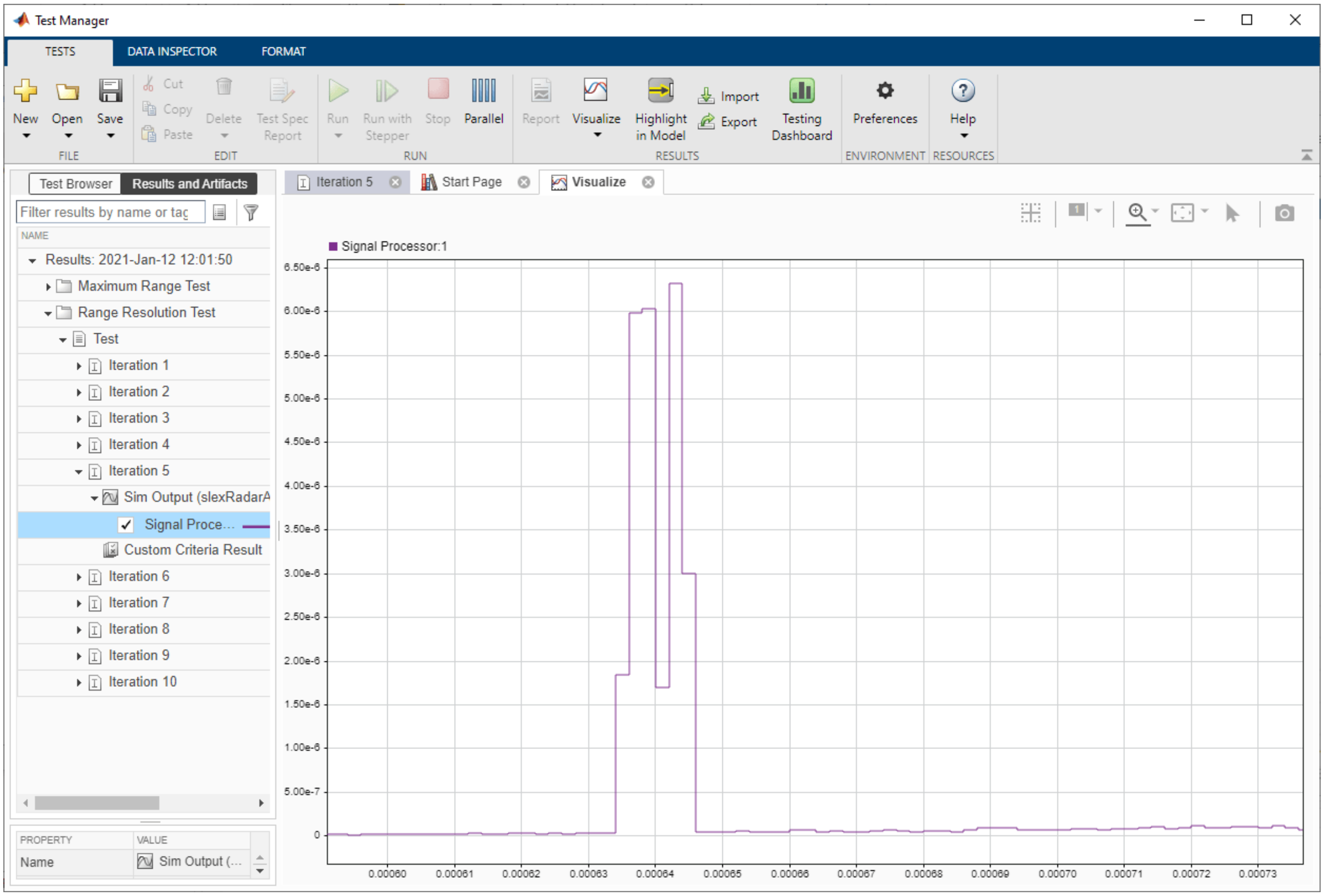
Implemented by:

- Waveform Generator
- Signal Processor

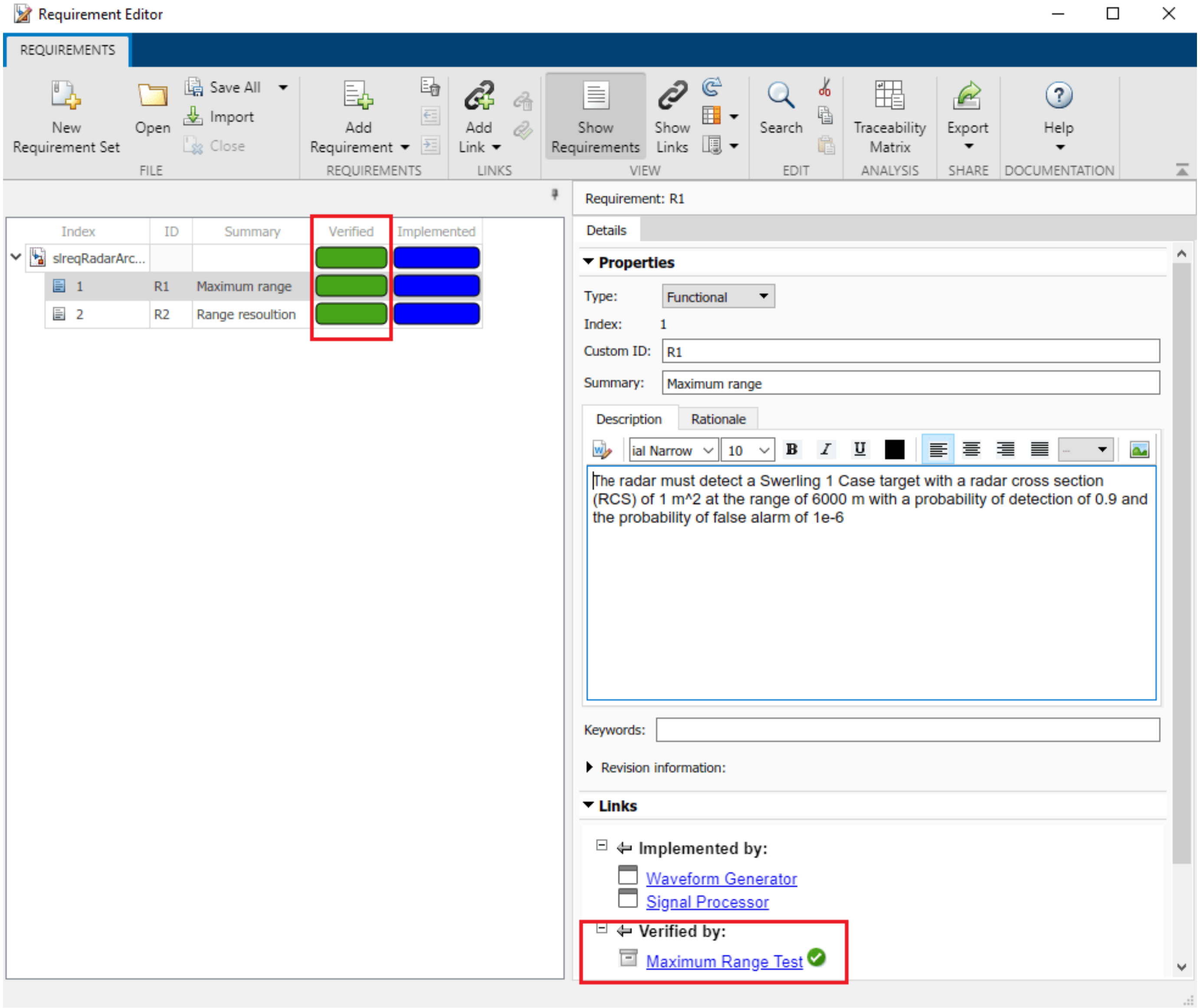
Verified by:

- Maximum Range Test

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.



Index	ID	Summary	Verified	Implemented
slreqRadarArc...			🟩	🟦
1	R1	Maximum range	🟩	🟦
2	R2	Range resoultion	🟩	🟦

Requirement: R1

Properties

Type: Functional

Index: 1

Custom ID: R1

Summary: Maximum range

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

Keywords:

Revision information:

Links

Implemented by:

Waveform Generator

Signal Processor

Verified by:

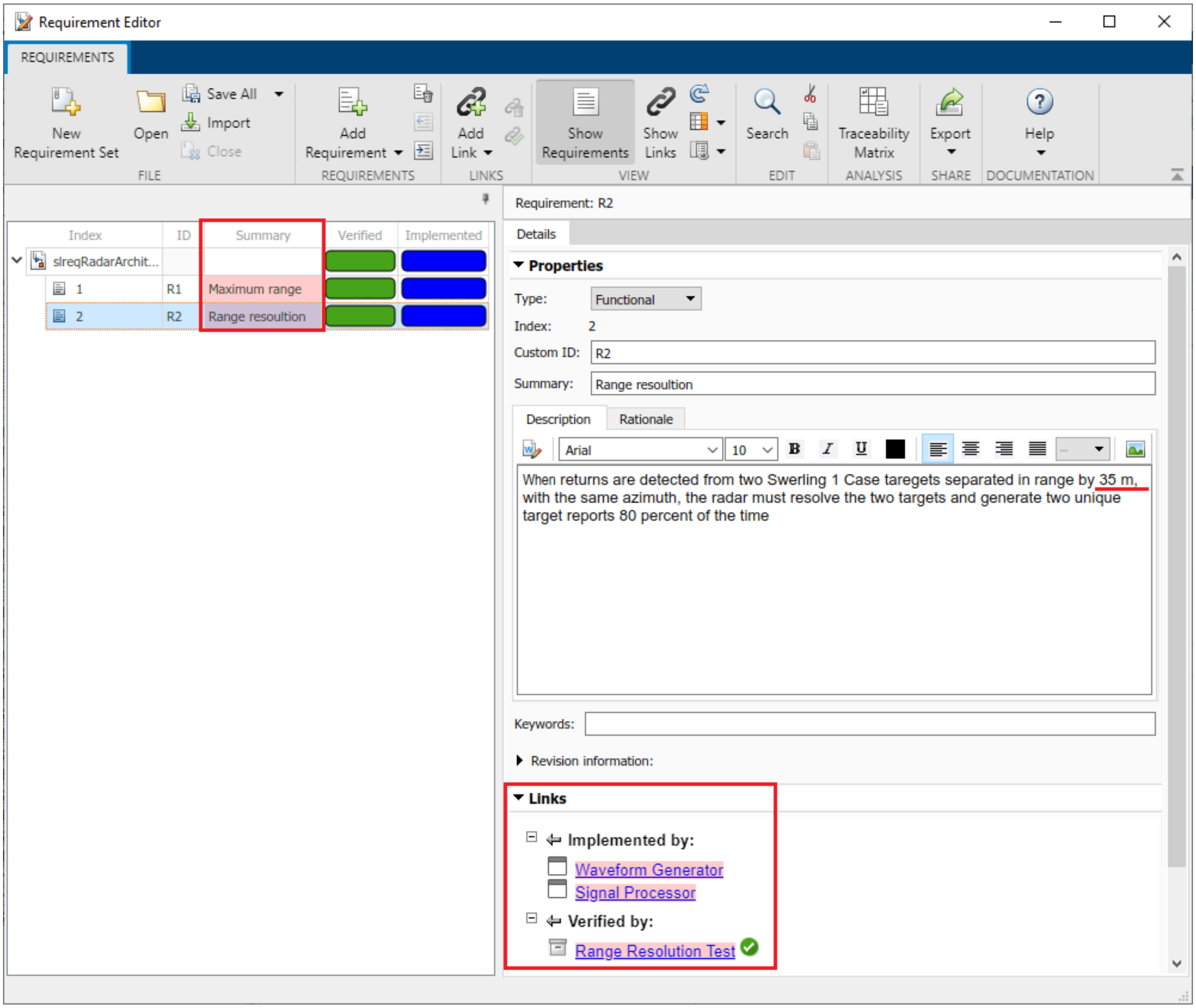
Maximum Range Test ✓

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.



Index	ID	Summary	Verified	Implemented
slreqRadarArchit...			🟩	🟦
1	R1	Maximum range	🟩	🟦
2	R2	Range resoultion	🟩	🟦

Requirement: R2

Type: Functional

Index: 2

Custom ID: R2

Summary: Range resoultion

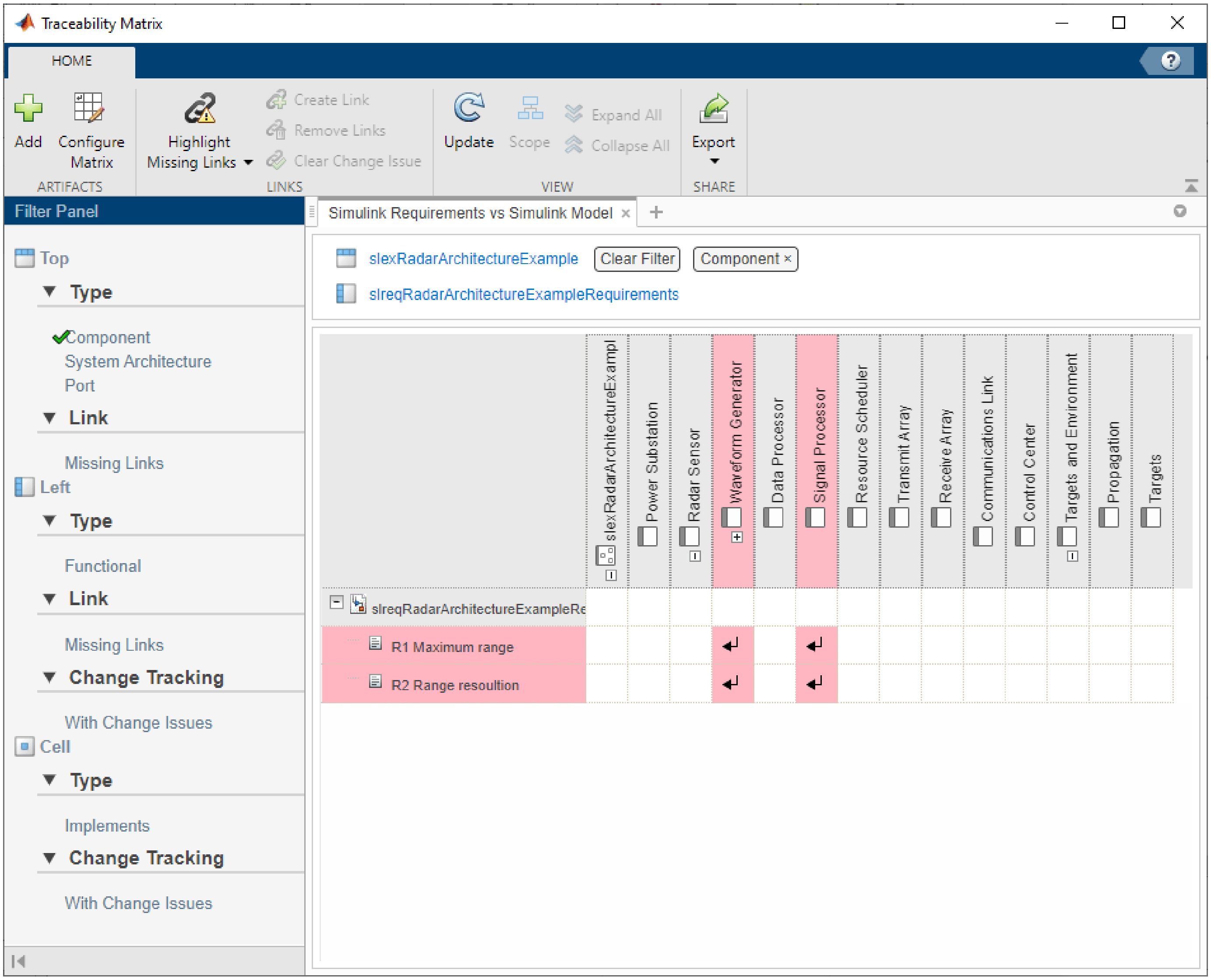
When returns are detected from two Swerling 1 Case taregets separated in range by 35 m, with the same azimuth, the radar must resolve the two targets and generate two unique target reports 80 percent of the time

Keywords:

Implemented by:
- Waveform Generator
- Signal Processor

Verified by:
- Range Resolution Test ✓

To monitor the changes in the requirements or in the implementations of the system components use the requirements Traceability Matrix.



	slexRadarArchitectureExampl	Power Substation	Radar Sensor	Waveform Generator	Data Processor	Signal Processor	Resource Scheduler	Transmit Array	Receive Array	Communications Link	Control Center	Targets and Environment	Propagation	Targets
slreqRadarArchitectureExampleRe														
R1 Maximum range				↵		↵								
R2 Range resoultion				↵		↵								

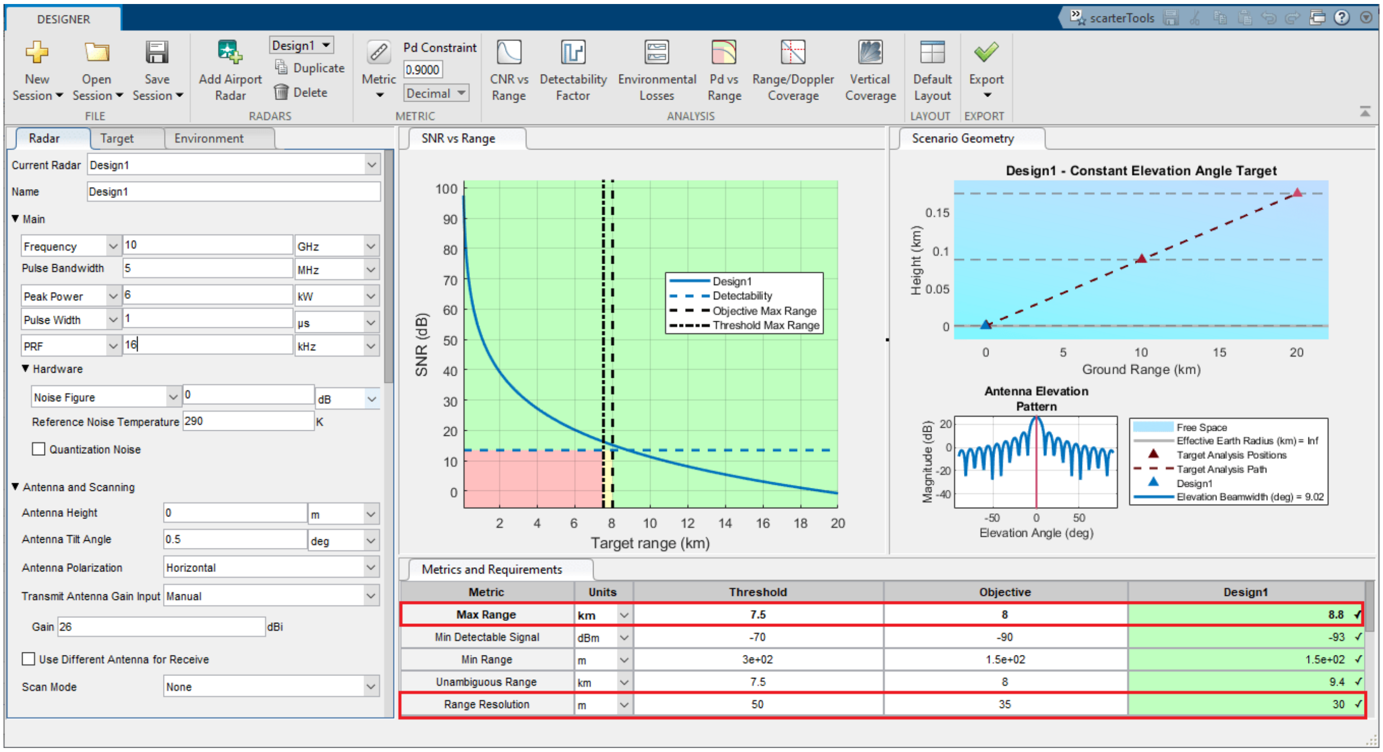
Updated System Parameters

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')
```



Current Radar: Design1

Name: Design1

▼ Main

Frequency: 10 GHz

Pulse Bandwidth: 5 MHz

Peak Power: 6 kW

Pulse Width: 1 μs

PRF: 16 kHz

▼ Hardware

Noise Figure: 0 dB

Reference Noise Temperature: 290 K

□ Quantization Noise

▼ Antenna and Scanning

Antenna Height: 0 m

Antenna Tilt Angle: 0.5 deg

Antenna Polarization: Horizontal

Transmit Antenna Gain Input: Manual

Gain: 26 dBi

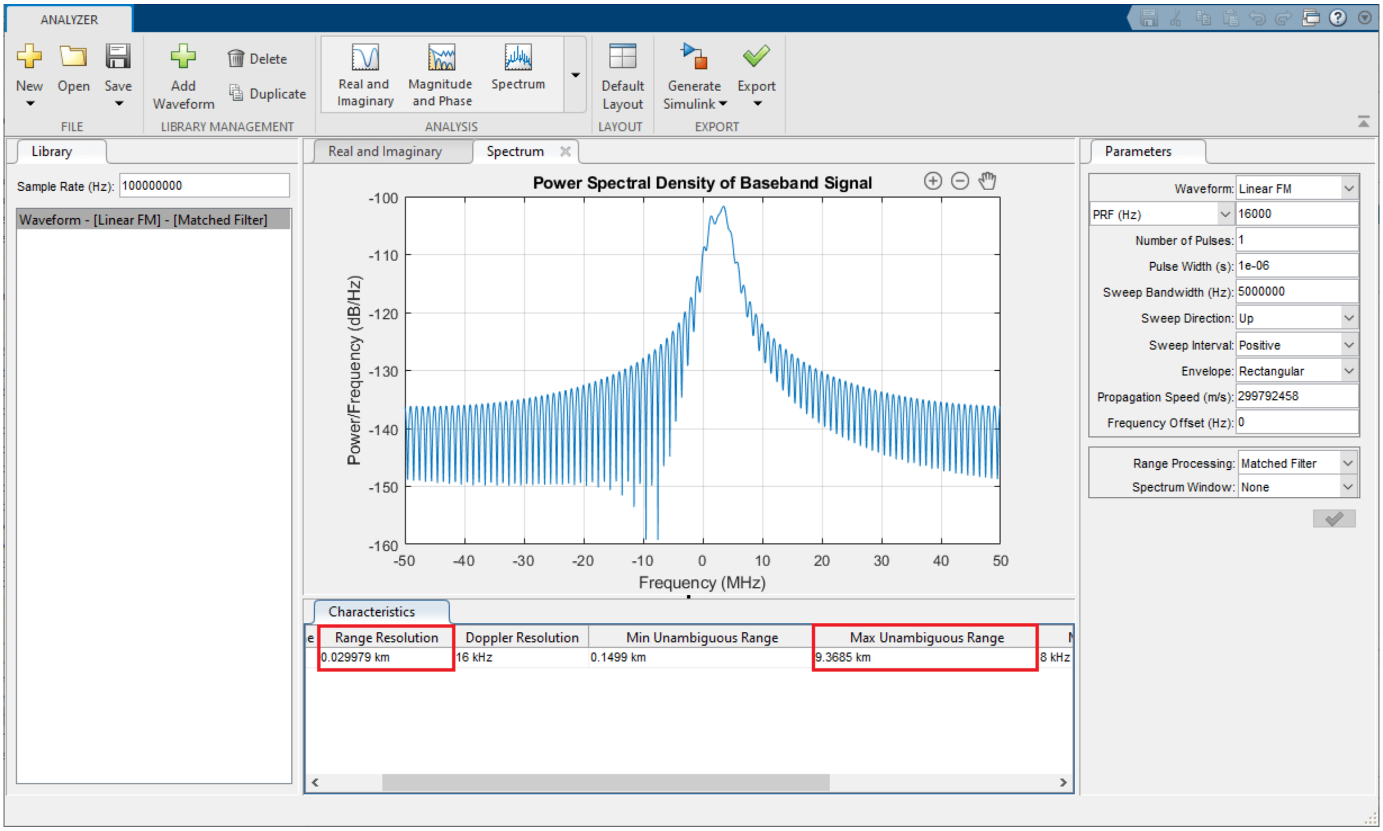
□ Use Different Antenna for Receive

Scan Mode: None

Metric	Units	Threshold	Objective	Design1
Max Range	**km**	**7.5**	**8**	**8.8** ✓
Min Detectable Signal	dBm	-70	-90	-93 ✓
Min Range	m	3e+02	1.5e+02	1.5e+02 ✓
Unambiguous Range	km	7.5	8	9.4 ✓
Range Resolution	m	50	35	30 ✓

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')
```



Sample Rate (Hz): 100000000

Waveform - [Linear FM] - [Matched Filter]

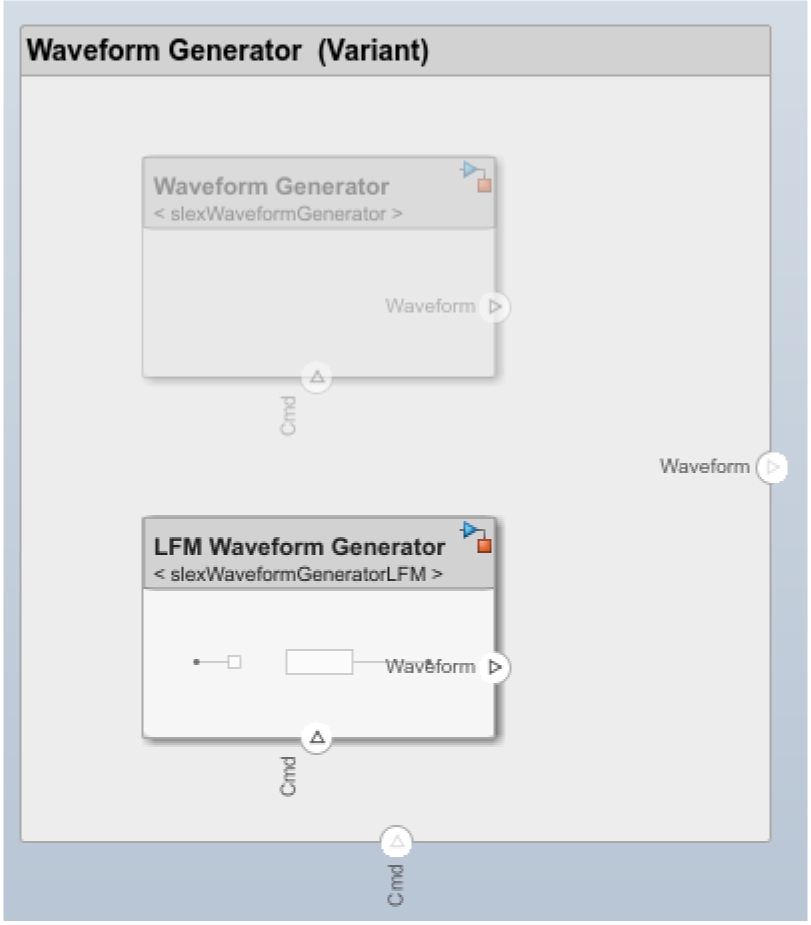
Waveform: Linear FM
PRF (Hz): 16000
Number of Pulses: 1
Pulse Width (s): 1e-06
Sweep Bandwidth (Hz): 5000000
Sweep Direction: Up
Sweep Interval: Positive
Envelope: Rectangular
Propagation Speed (m/s): 299792458
Frequency Offset (Hz): 0

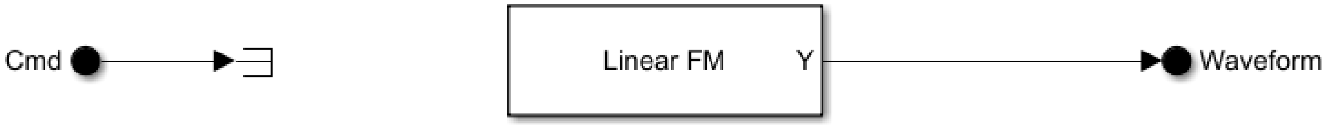
Range Processing: Matched Filter
Spectrum Window: None

e	Range Resolution	Doppler Resolution	Min Unambiguous Range	Max Unambiguous Range	M
	0.029979 km	16 kHz	0.1499 km	9.3685 km	8 kHz

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.

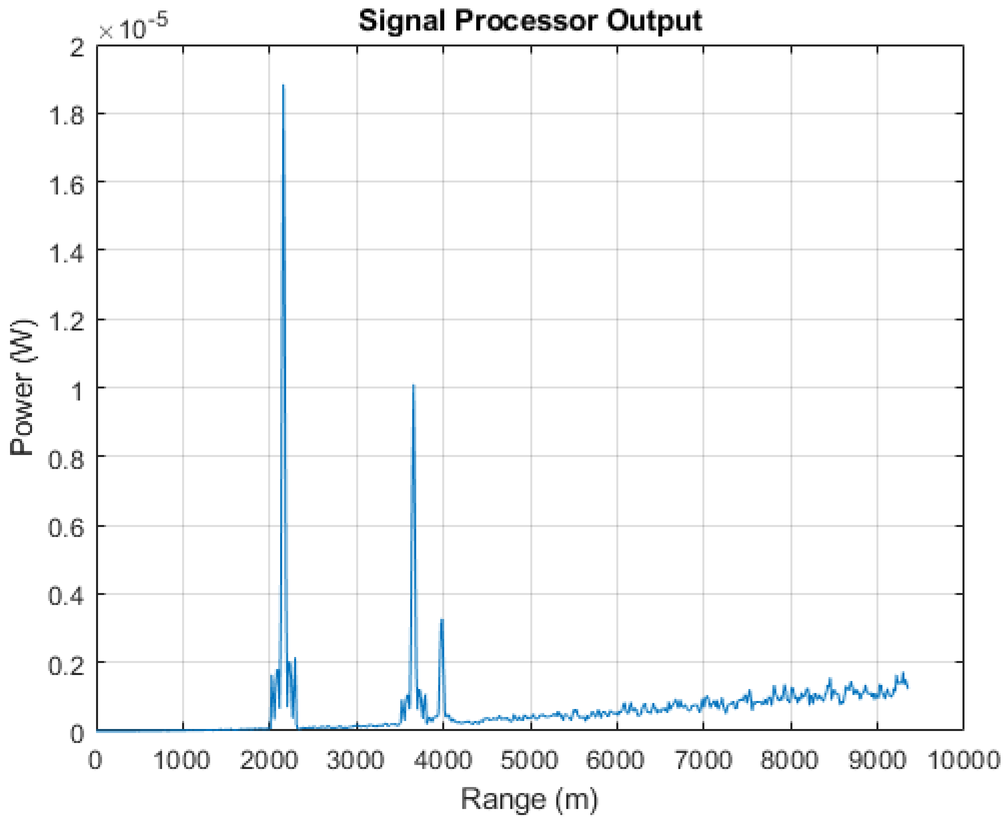

Waveform Generator (Variant)
Waveform Generator
< slexWaveformGenerator >
Waveform
Cmd
Waveform
LFM Waveform Generator
< slexWaveformGeneratorLFM >
Waveform
Cmd
Cmd



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));
```

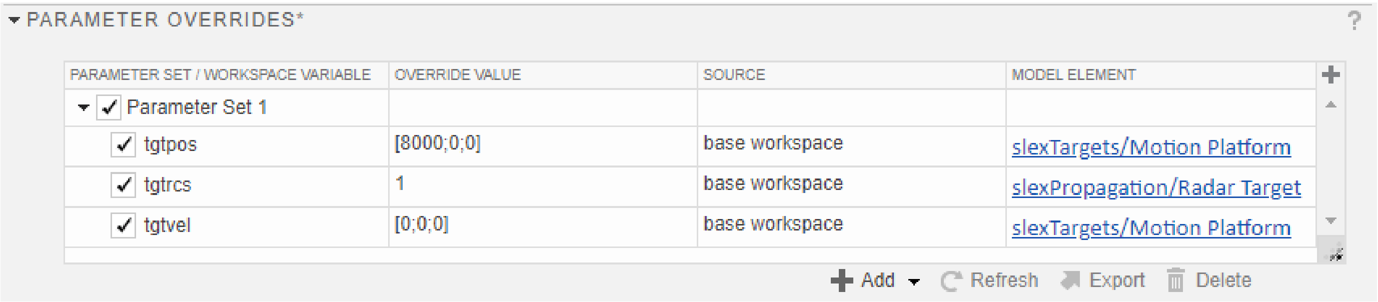
```
xlabel('Range (m)');
ylabel('Power (W)');
title('Signal Processor Output');

grid on;
```



Updated Tests

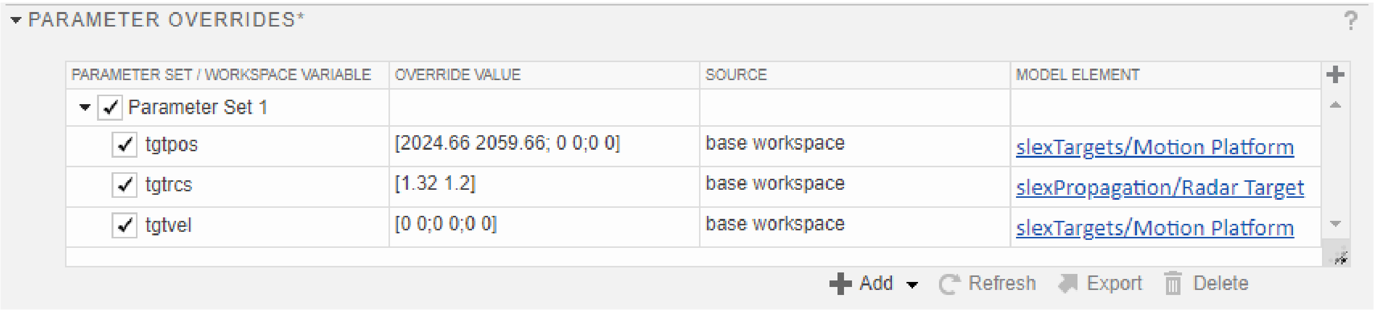
Before verifying that the radar system with LFM can satisfy the updated requirements, make corresponding modifications to the tests by updating the targets positions.

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



PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
☑ Parameter Set 1			
☑ tgtpos	[8000;0;0]	base workspace	slexTargets/Motion Platform
☑ tgtrcs	1	base workspace	slexPropagation/Radar Target
☑ tgtvel	[0;0;0]	base workspace	slexTargets/Motion Platform

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



PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
☑ Parameter Set 1			
☑ tgtpos	[2024.66 2059.66; 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

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 Simulink Requirements 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 `radarScenario` and `waypointTrajectory`.

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 us 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 us 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 we are using 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.

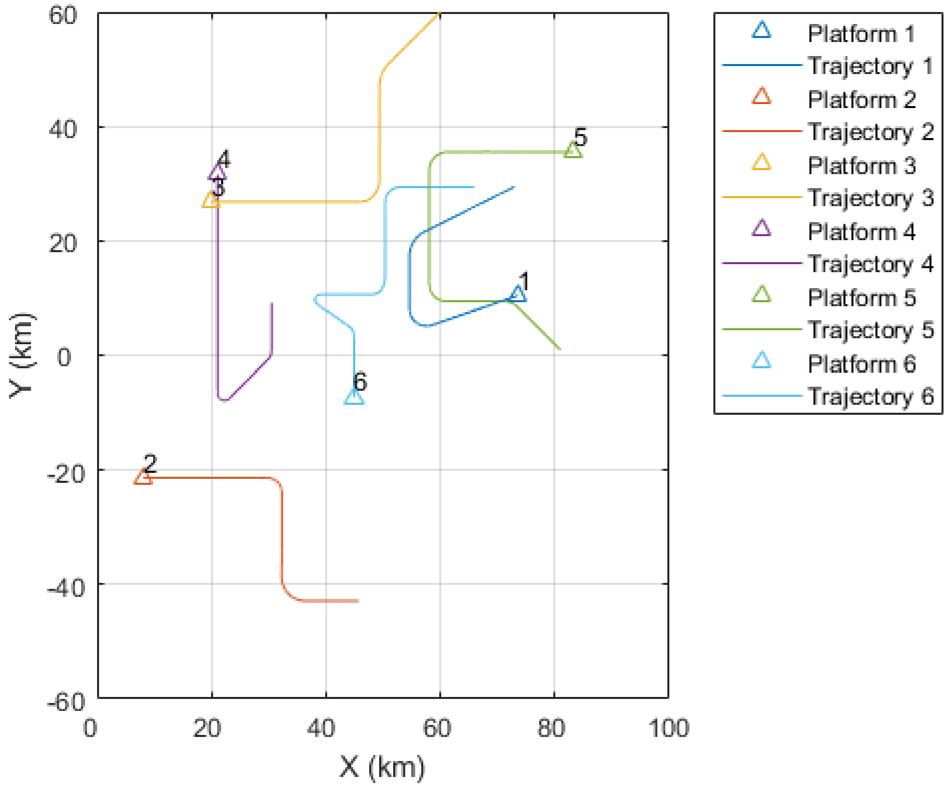
To use the control points, you can create a scenario with six platforms and assign a trajectory to each:

```
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

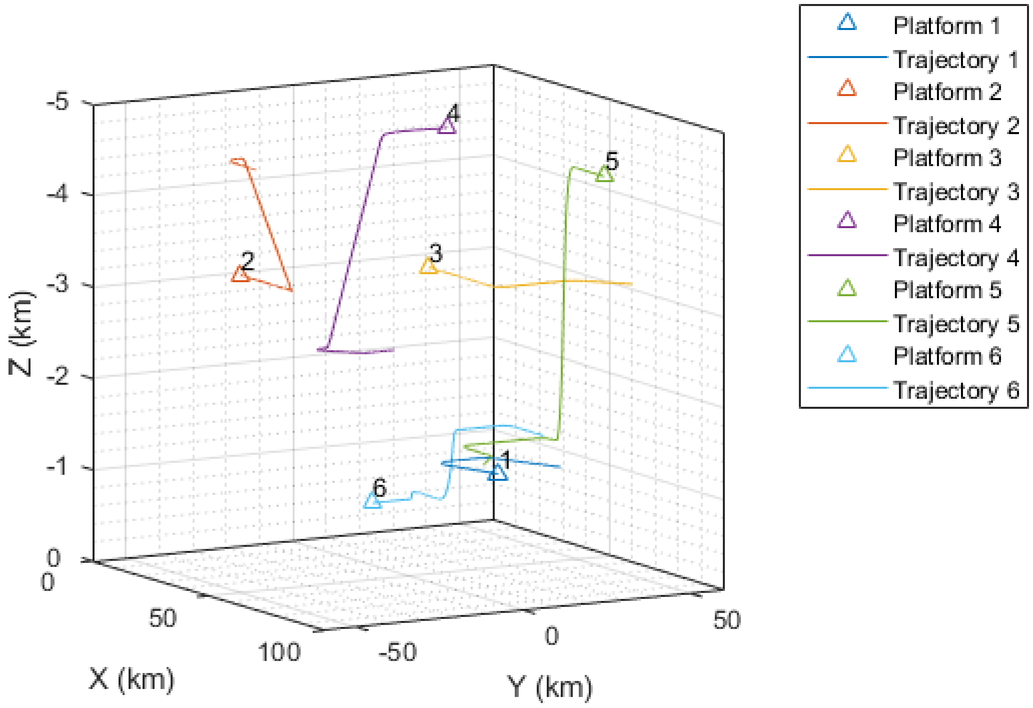
Once you have the scenario and plotter set up, you can set up a `theaterPlot` to create an animated view of the locations of the aircraft as time progresses.

```
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 NED (north-east-down) coordinate system, elevation above ground has a negative z component.

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



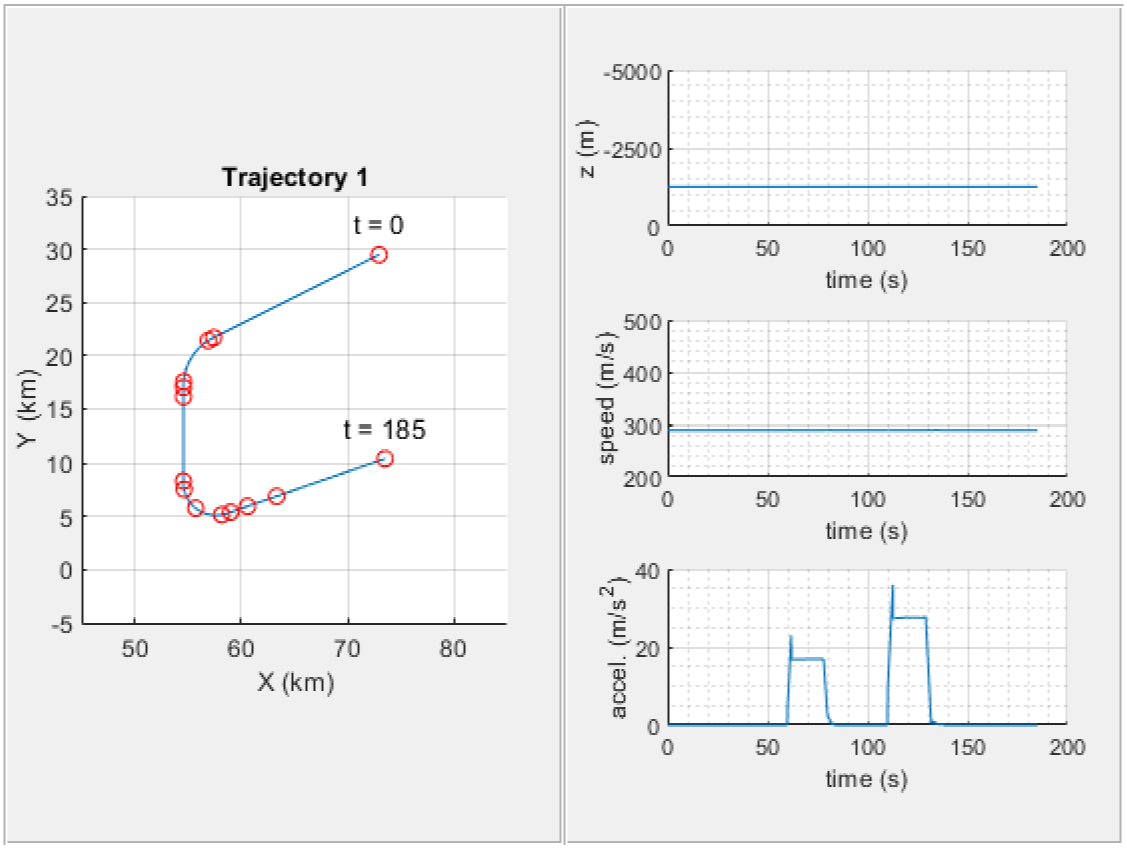
Trajectory 1

It may 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.

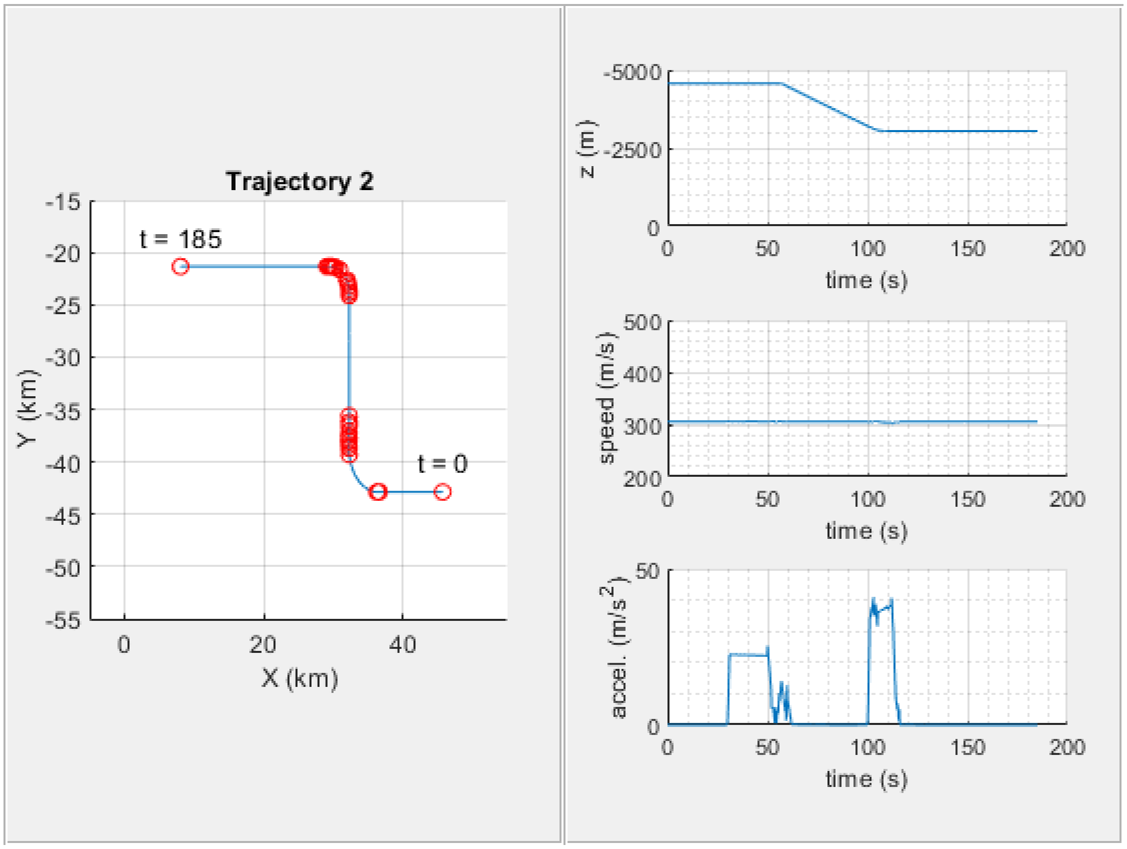
```
[time, position, velocity, acceleration] = cumulativeHistory(helperPlot);
helperTrajectoryViewer(1, time, position, velocity, acceleration, trajTable);
```



Trajectory 2

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

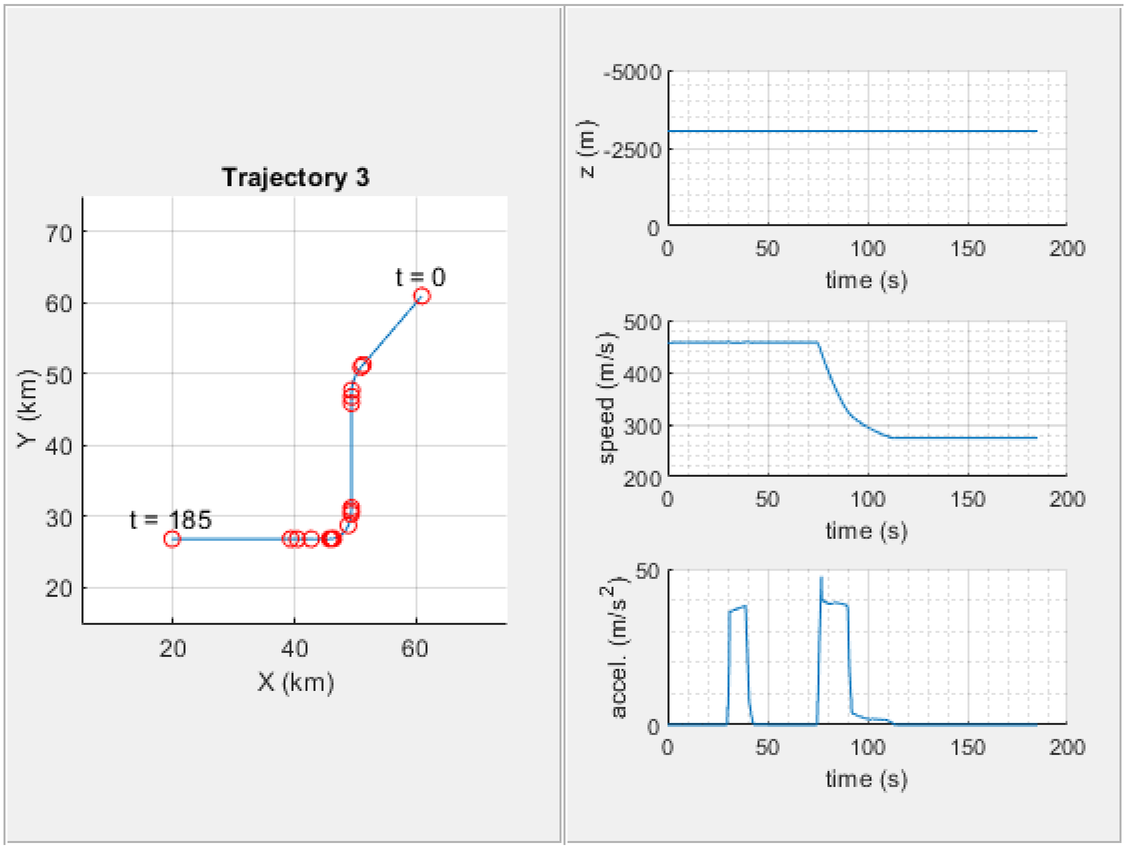
```
helperTrajectoryViewer(2, time, position, velocity, acceleration, trajTable);
```



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.

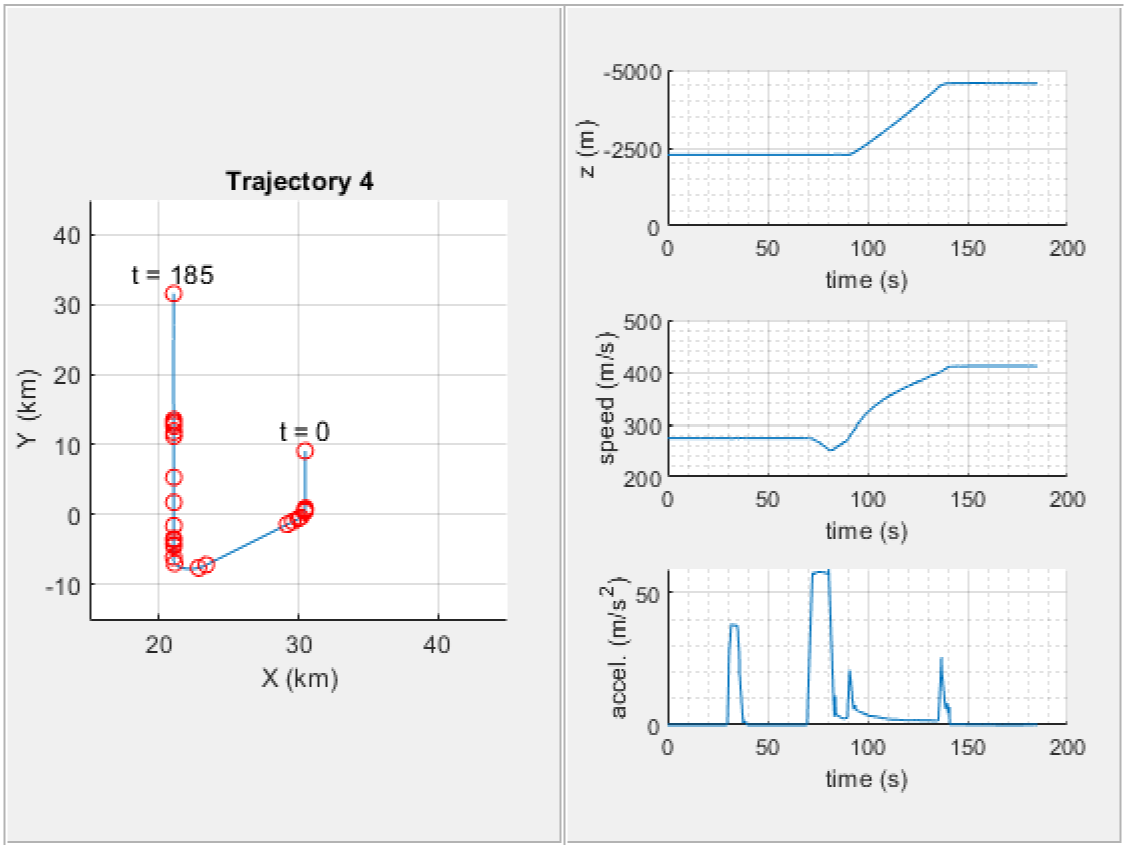
```
helperTrajectoryViewer(3, time, position, velocity, acceleration, trajTable);
```



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.

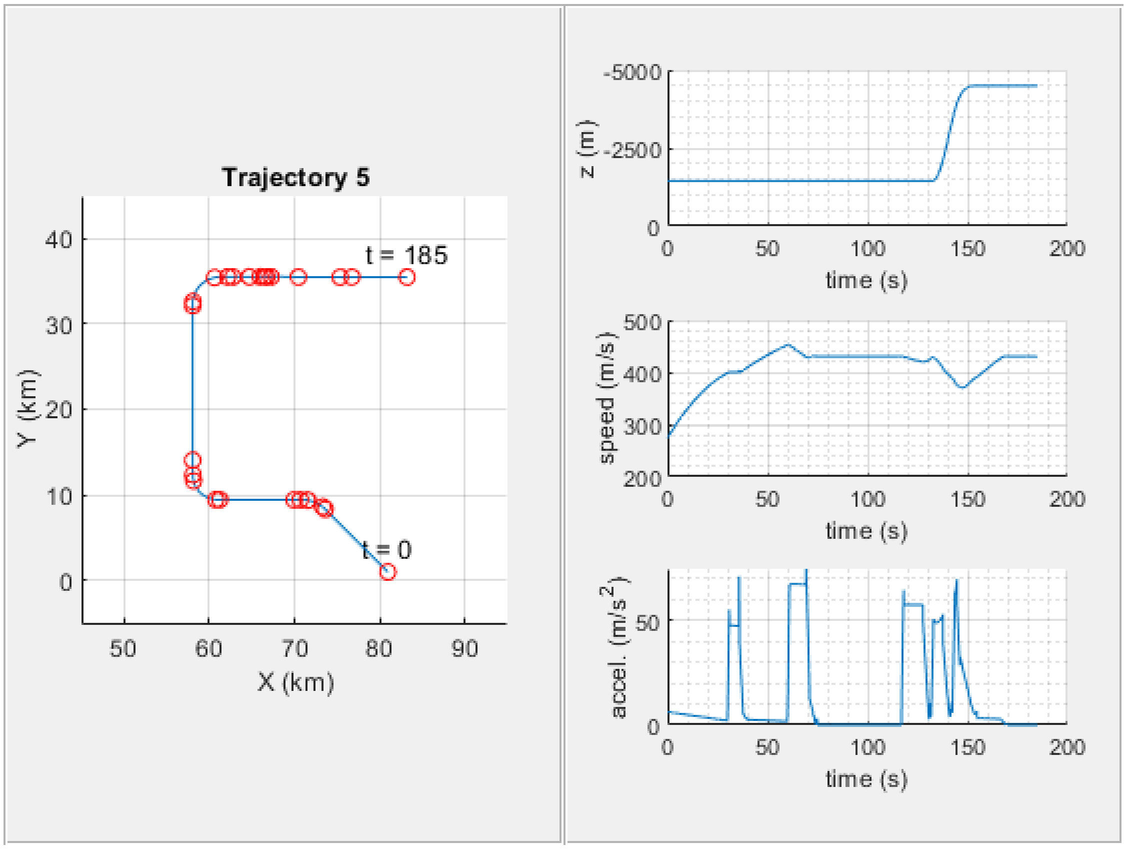
```
helperTrajectoryViewer(4, time, position, velocity, acceleration, trajTable);
```



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.

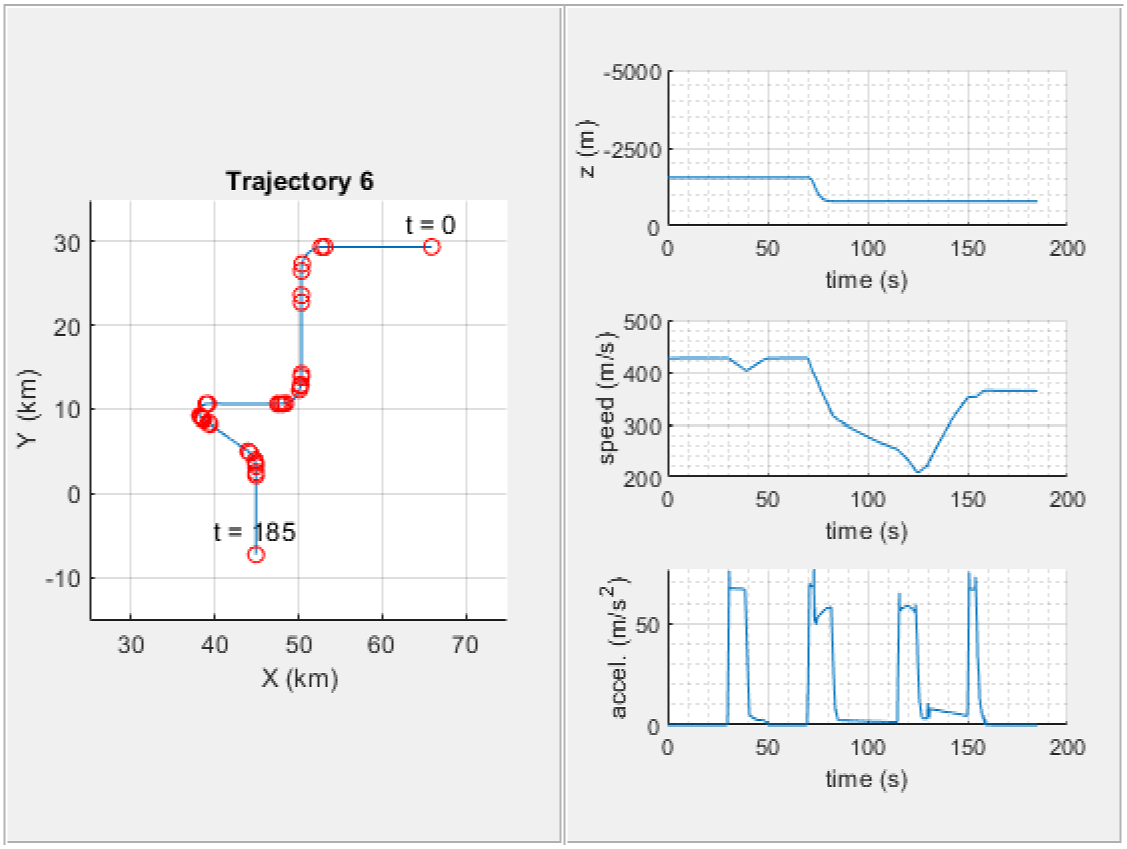
```
helperTrajectoryViewer(5, time, position, velocity, acceleration, trajTable);
```



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` 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 W.D. Blair, G. A. Watson, T. Kirubarajan, Y. Bar-Shalom, "Benchmark for Radar Allocation and Tracking in ECM." Aerospace and Electronic Systems IEEE Trans on, vol. 34. no. 4. 1998

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.

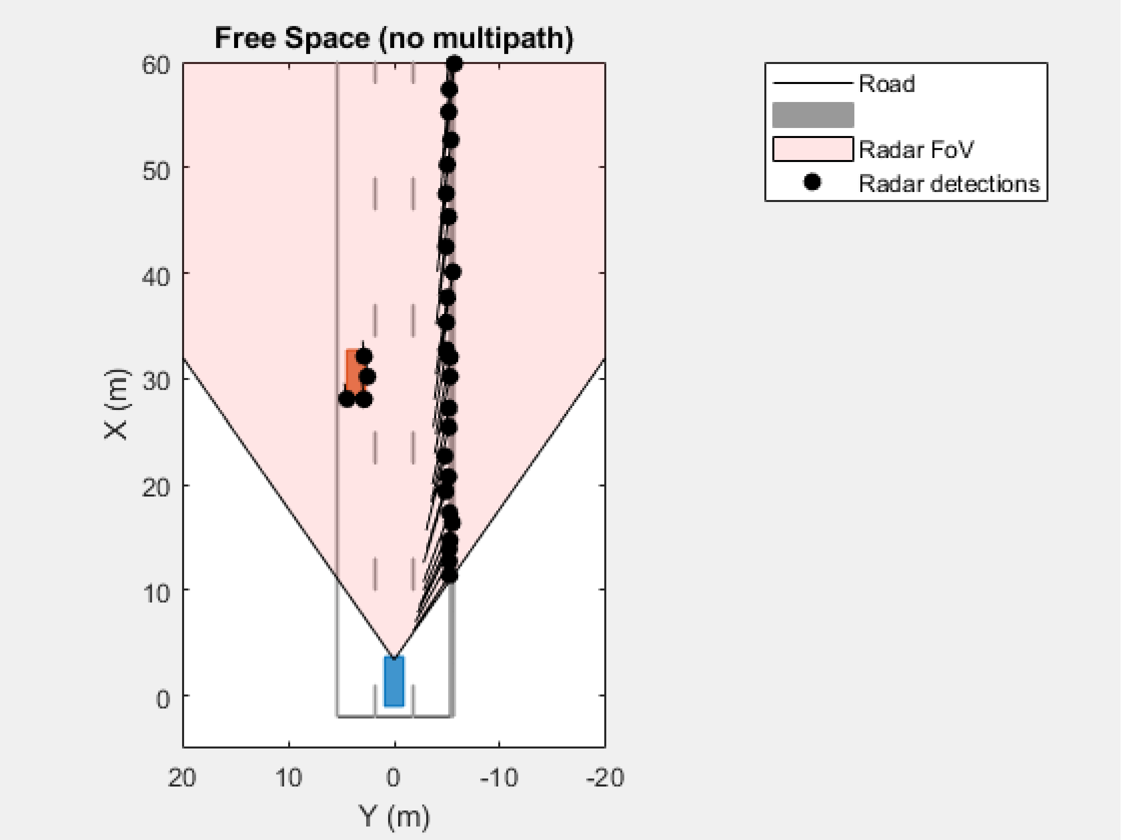
Motivation

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 100 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);
```

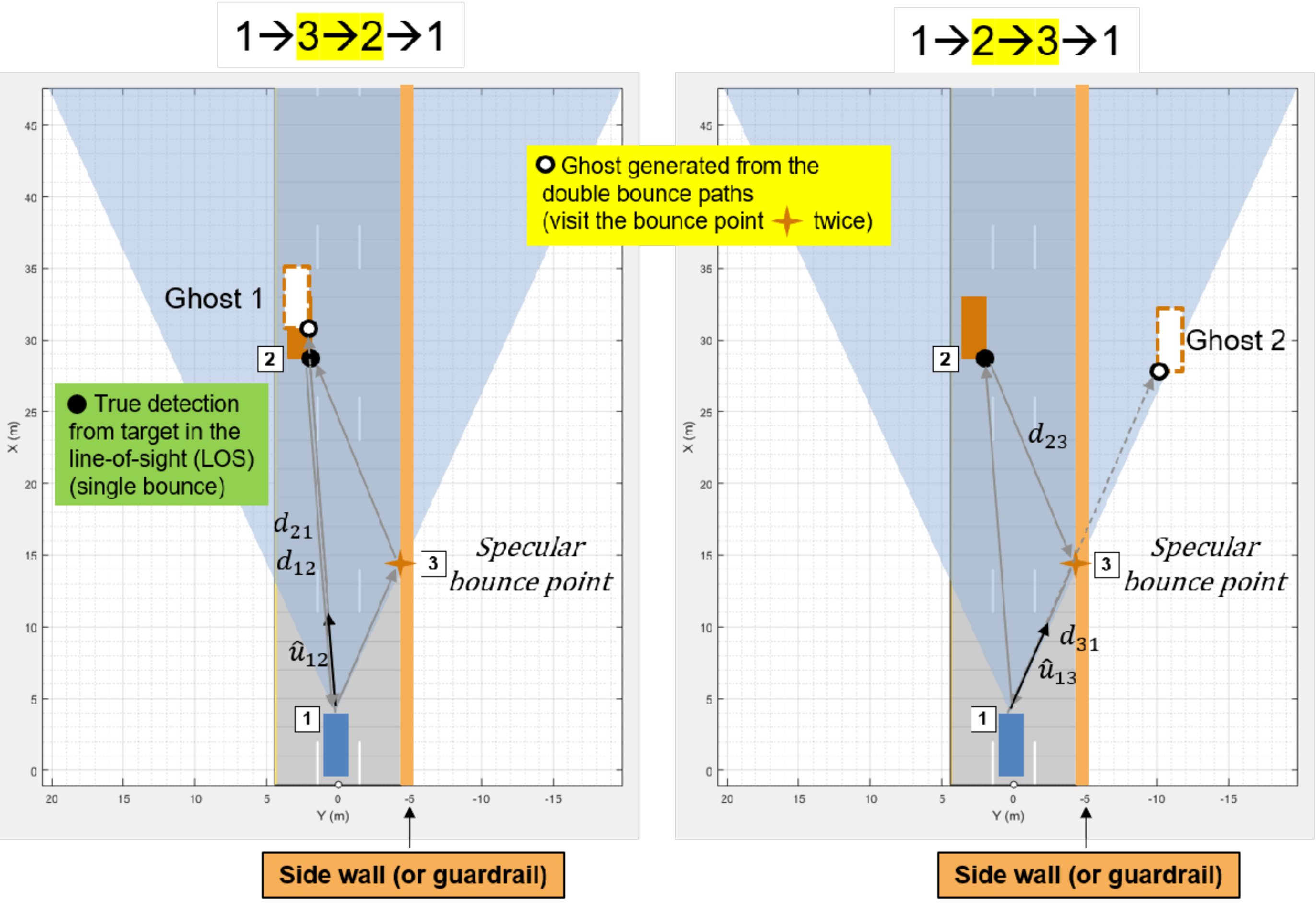


This figure shows the locations of the detections along the target vehicle as well as along the side of the barrier. However, detections are not always so well-behaved. One phenomenon that can pose considerable challenges to radar engineers is multipath. Multipath is when the signal not only propagates 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

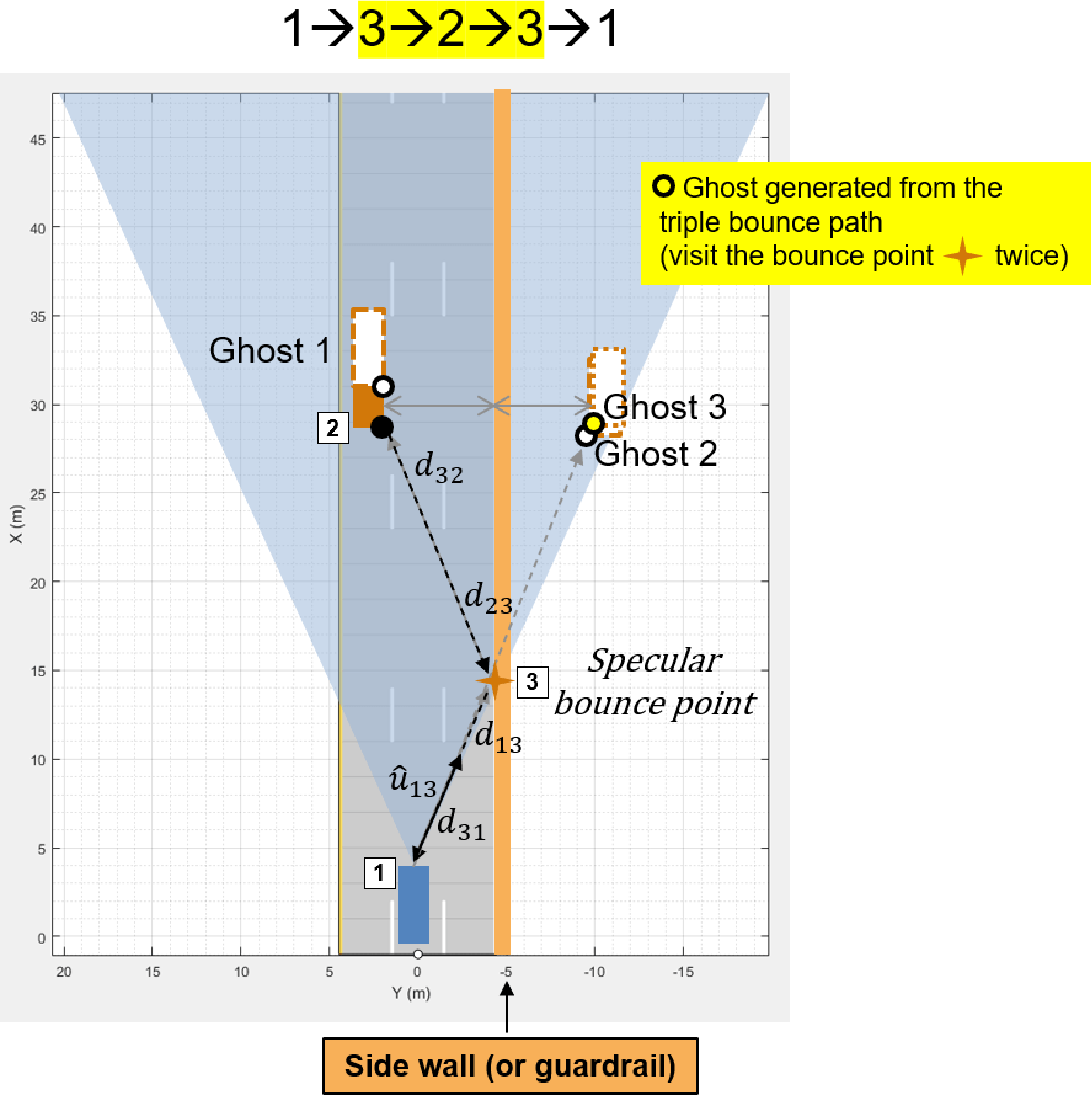
In this case, there are two unique propagation paths that consist of two bounces.

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

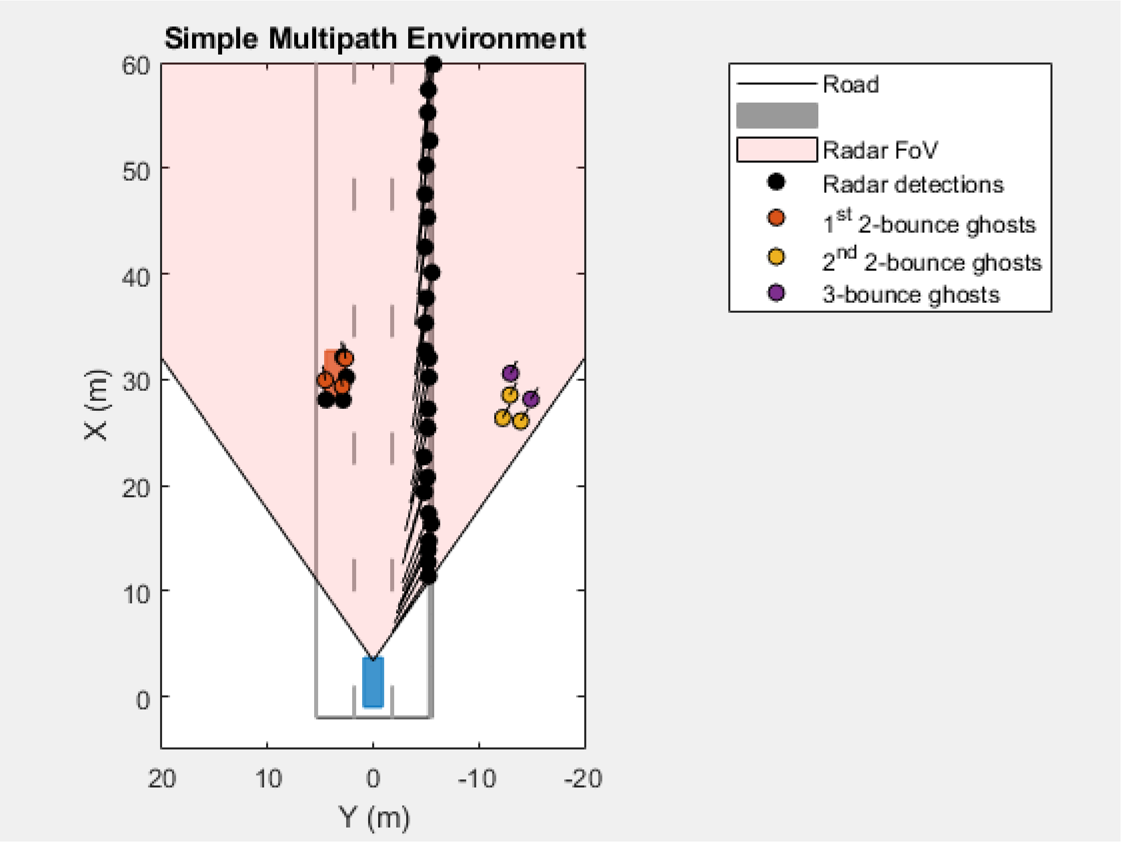


The three-bounce path reflects off the barrier twice. This path never propagates directly to the target or directly back to the radar. The three-bounce ghost detections appear on the other side of the reflecting surface as the radar receives the reflected signal in that direction. Additionally, it has 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');
```



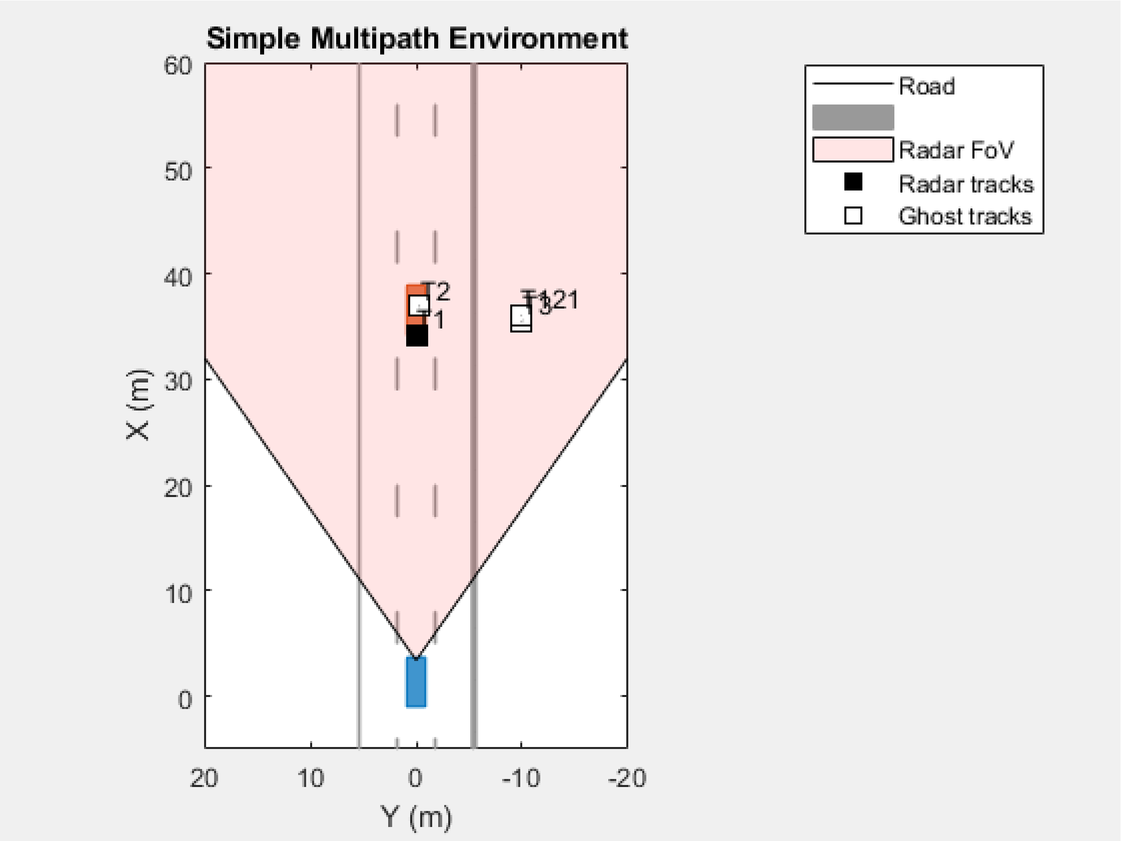
This figure reproduces the analysis of the three propagation paths. The first two-bounce ghosts lie in the direction of the target at a slightly longer range than the direct-path detections. The second two-bounce and three-bounce ghosts lie in the direction of the mirrored image of the target generated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

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

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

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

Construct the equivalent `radarTransceiver` directly from the `radarDataGenerator`.

```
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';

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: [1×1 phased.RectangularWaveform]
             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: [3.4000 0 0.2000]
```

```
          MountingAngles: [0 0 0]
    NumRepetitionsSource: 'Property'
          NumRepetitions: 512
```

```
rdg.AzimuthResolution = azRes;
```

The `radarTransceiver` has only one transmit and receive element, but the statistical radar uses a uniform linear array (ULA) to form multiple beams. Attach a ULA array to the receive antenna of the `radarTransceiver`. The number or elements in the ULA is determined by the radar's azimuth resolution and wavelength.

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

```
numRxElmt = 26
```

```
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

   456    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);
```

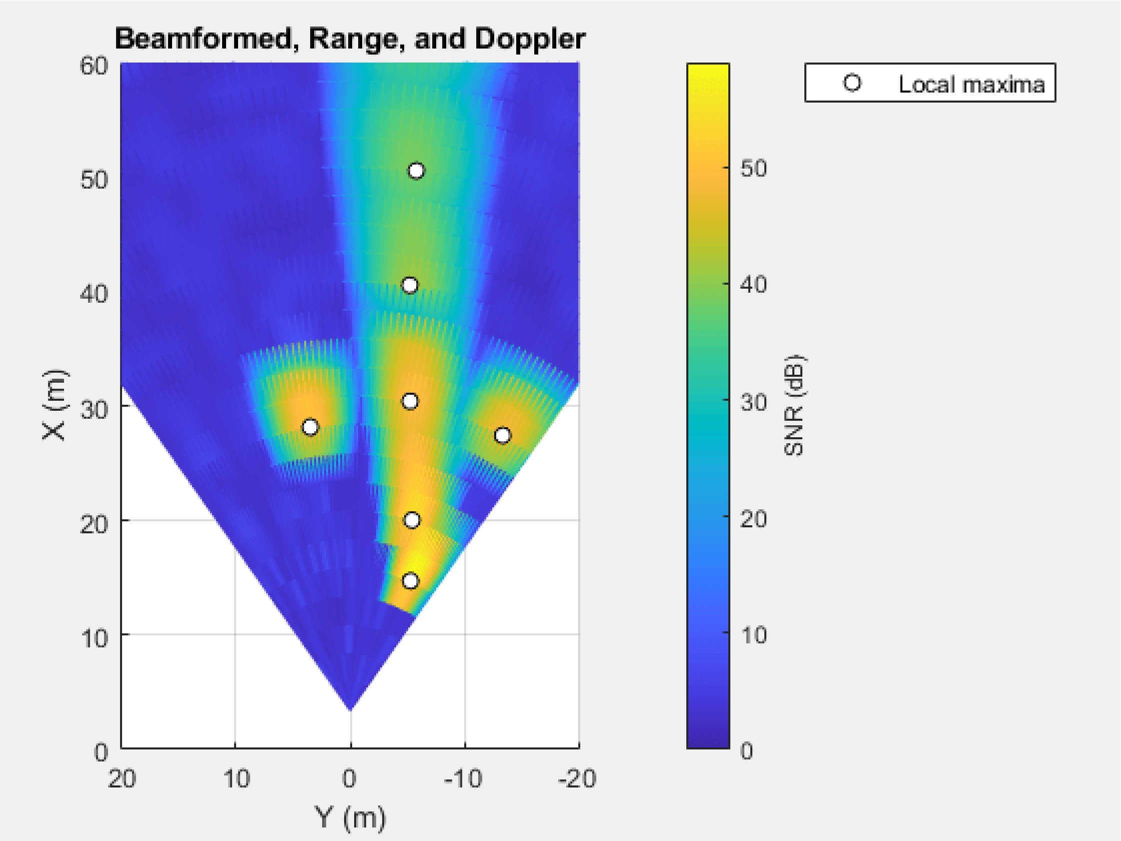
Beamforming

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.

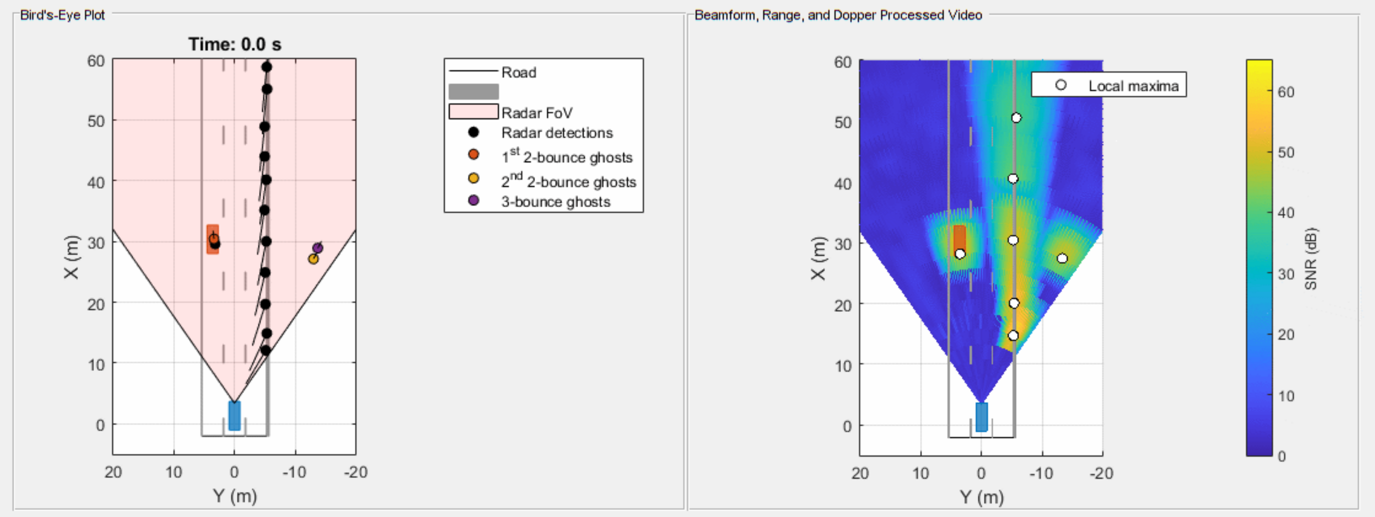
```
helperPlotBeamformedRangeDoppler(Xbfmrngdop,rggrid,anggrid,rtxrx);
```



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.

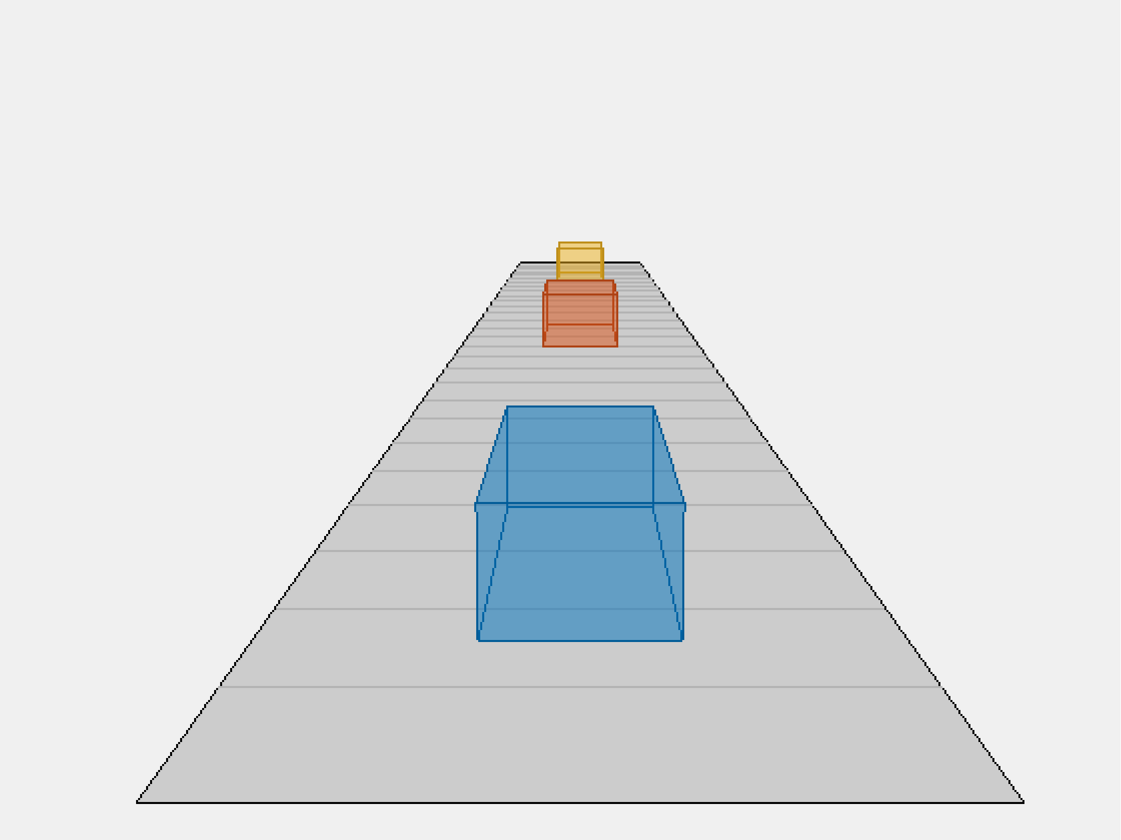
```
helperPlayStatAndIQMovie(scenario,egoVehicle,rtxrx,rdg,rngdopproc,beamformer,bmfwin);
```



Multipath Ground Bounce

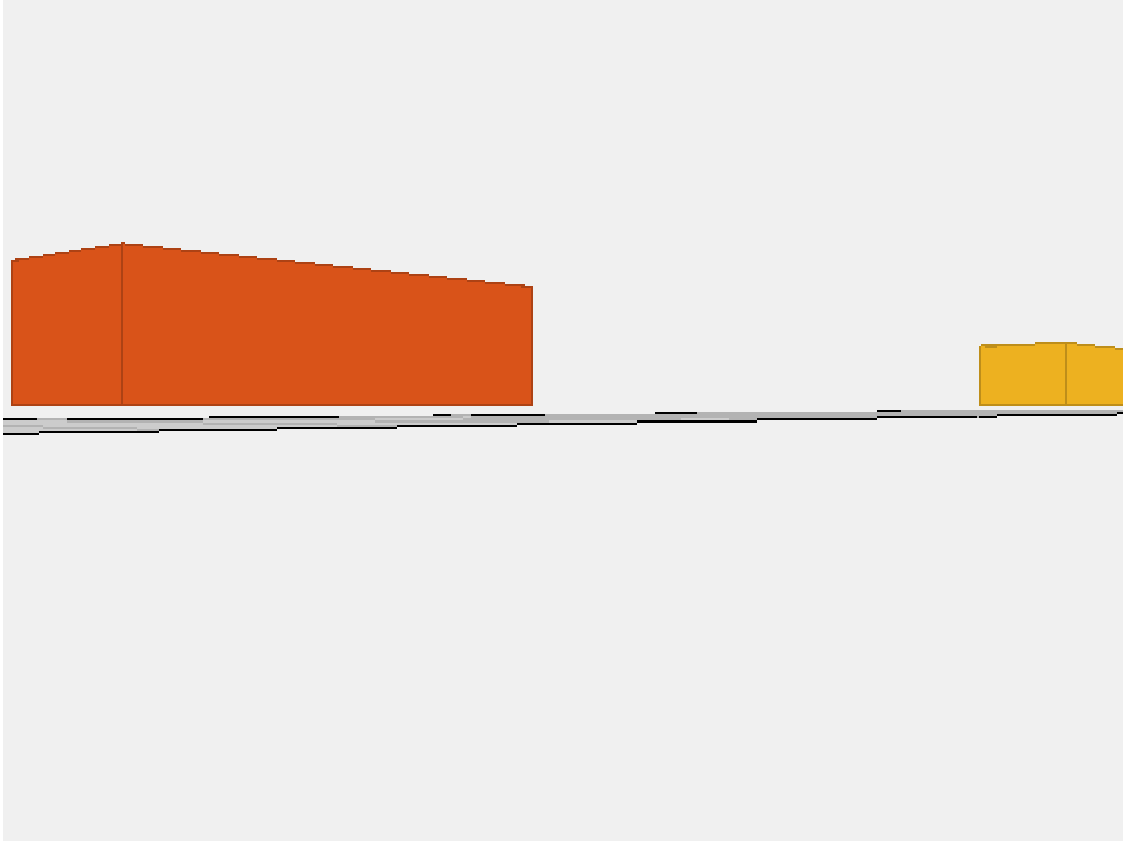
Multipath ghost detections can be used at times to see objects in the road that would otherwise not be detected by the radar due to occlusion. One example is the detection of an occluded vehicle due to multipath off of the road surface. Use the helperGroundBounceScenarioDSD function to create a scenario 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);
```

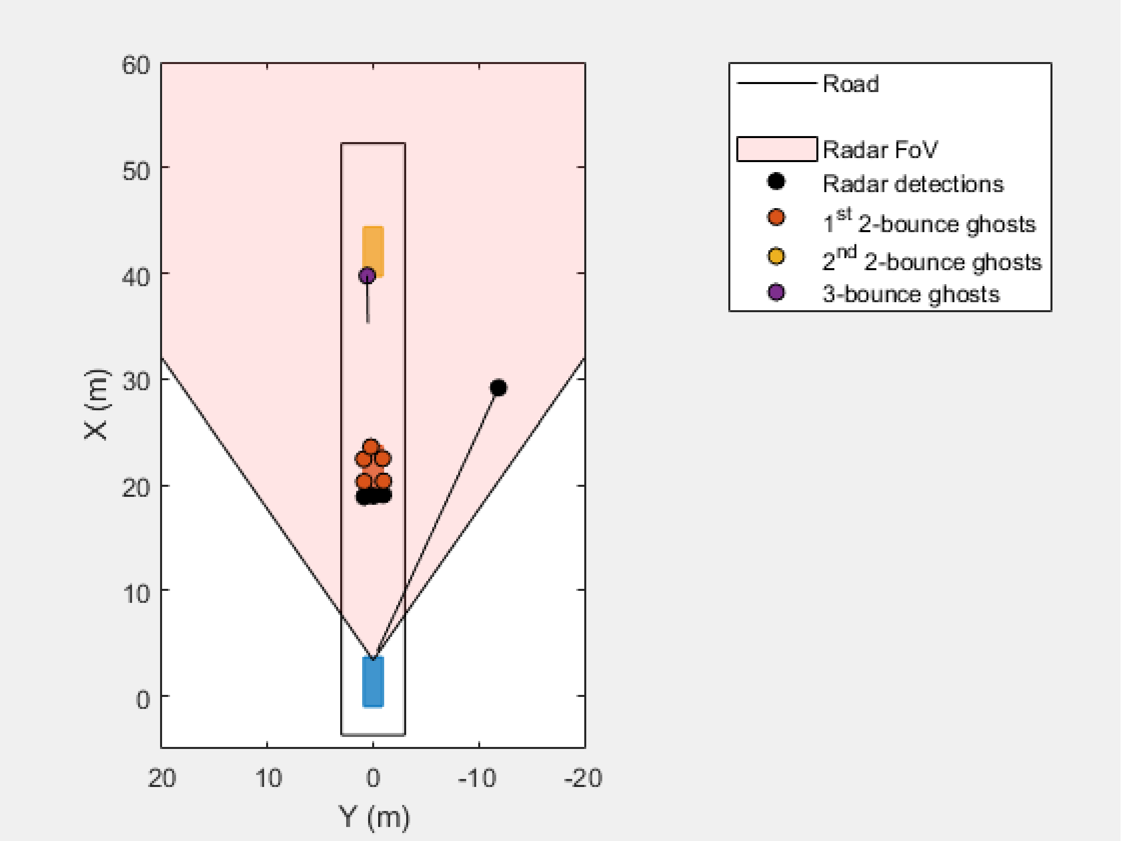


Multipath can pass through the space between the underside of a car and the surface of the road.

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.

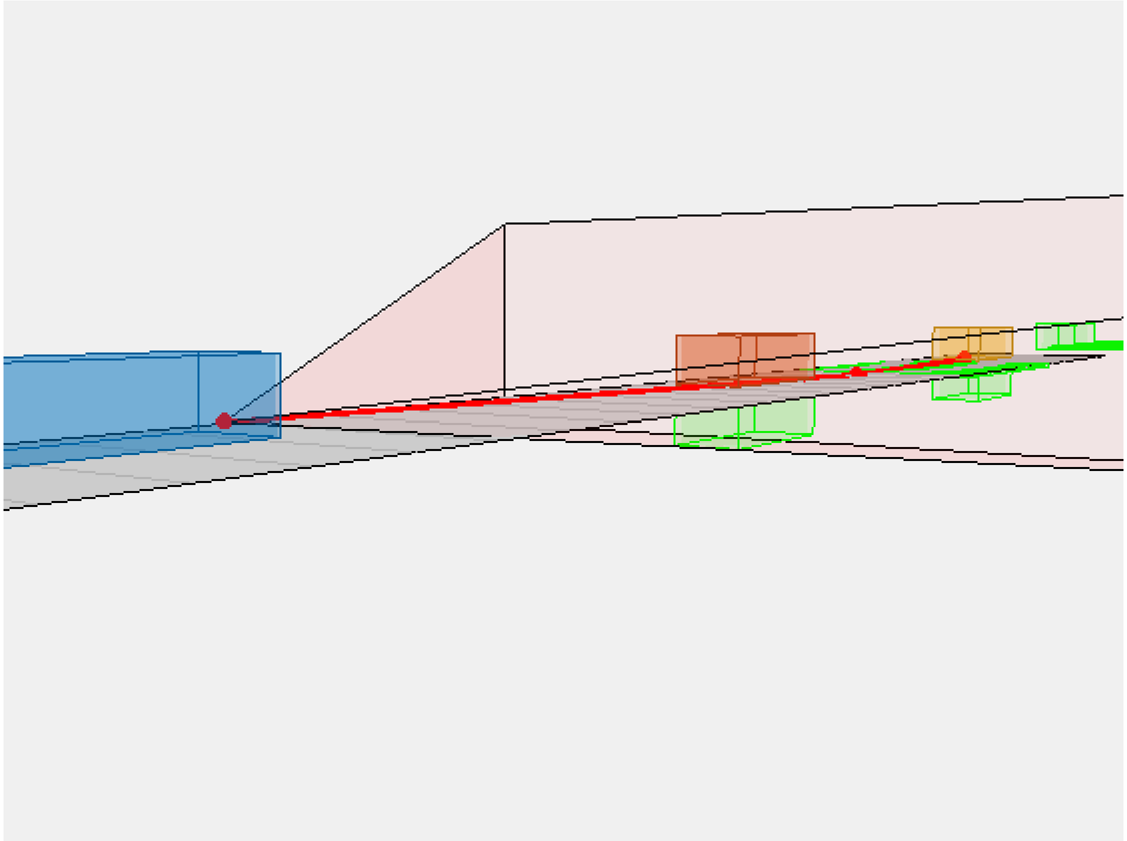
```
release(rdg);
rdg.RangeRateResolution = 0.5;
rdg.FieldOfView(2) = 10;
rdg.TargetReportFormat = 'Detections';

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);
```



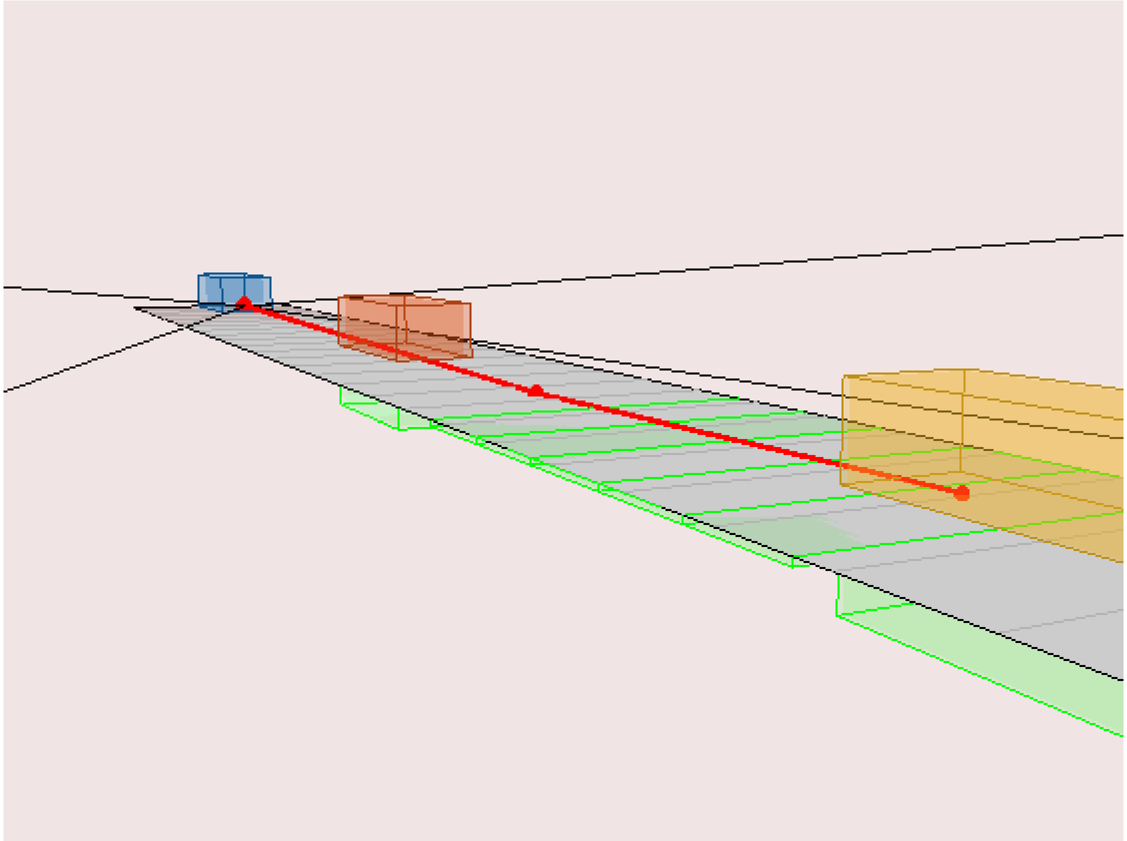
The detection from the occluded car is possible due to the three-bounce path that exists between the road surface and the underside of the red car.

```
% 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.
```

```
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

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

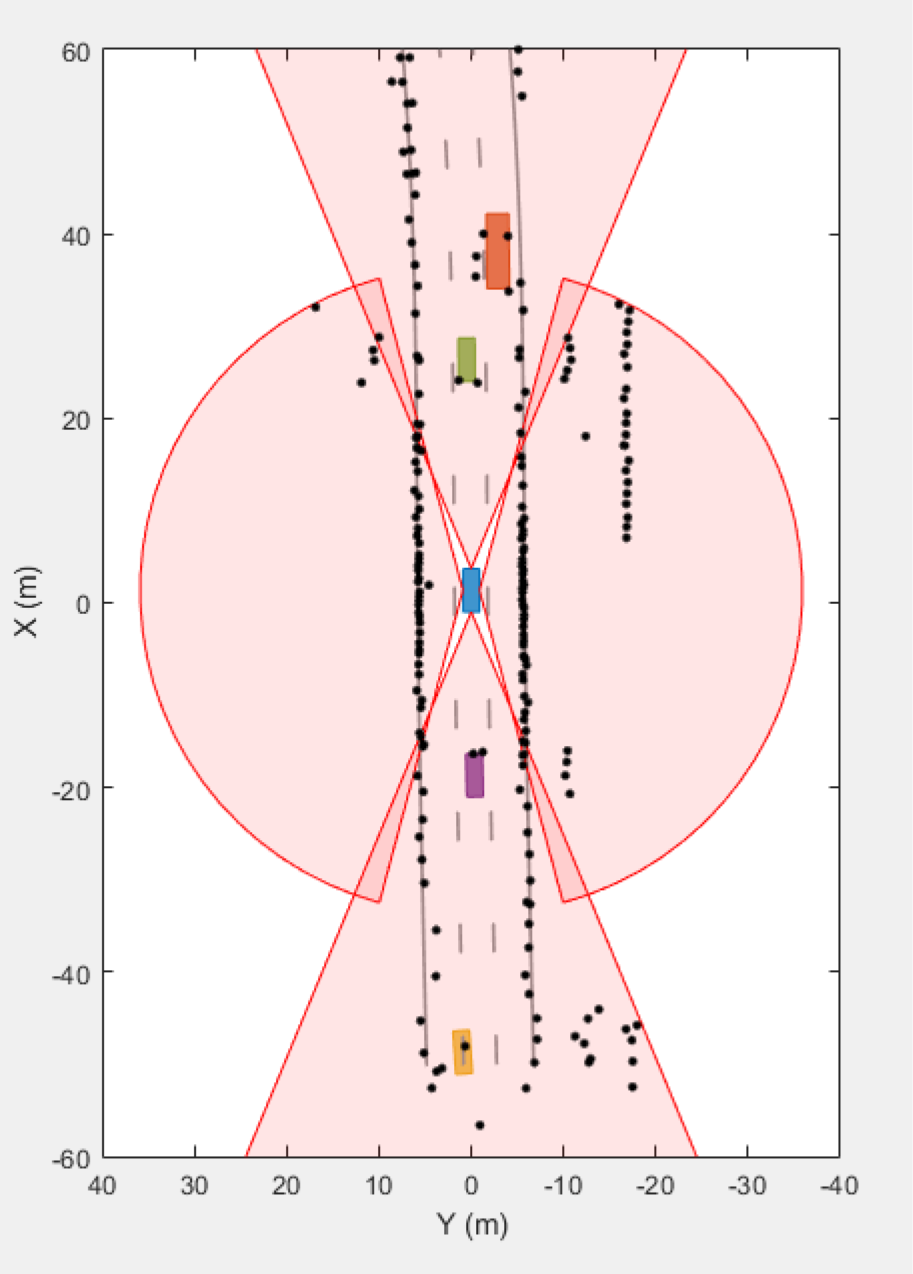
Highway Vehicle Tracking with Multipath Radar Reflections

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 1-44 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

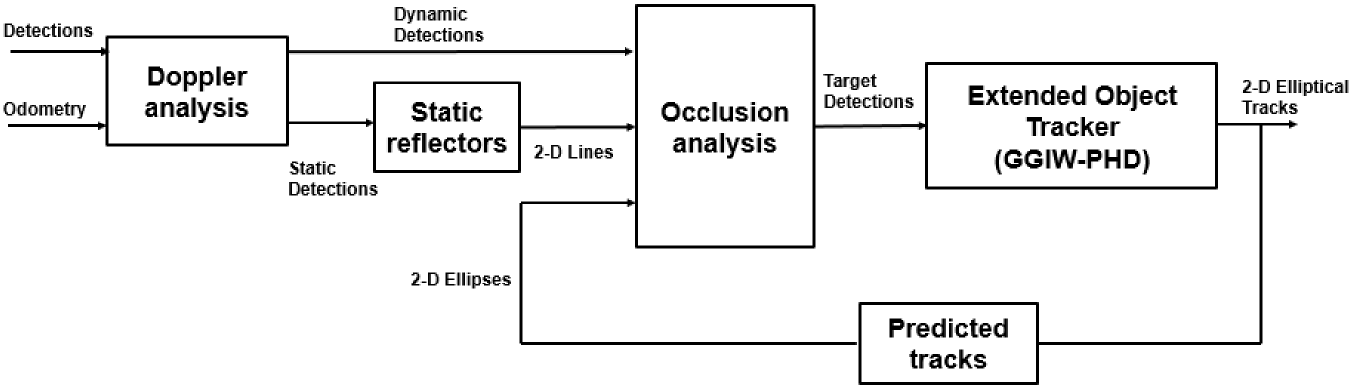
The scenario used in this example is created using the `drivingScenario` class. You use the `radarDataGenerator` System object™ to simulate radar returns from a direct path and from reflections in the scenario. The `HasGhosts` property of the sensor is specified as `true` to simulate multipath reflections. 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-2,...
    'BirthRate',1e-2,...
    'HasSensorConfigurationsInput',true...
    );
```

Run Scenario and Track Objects

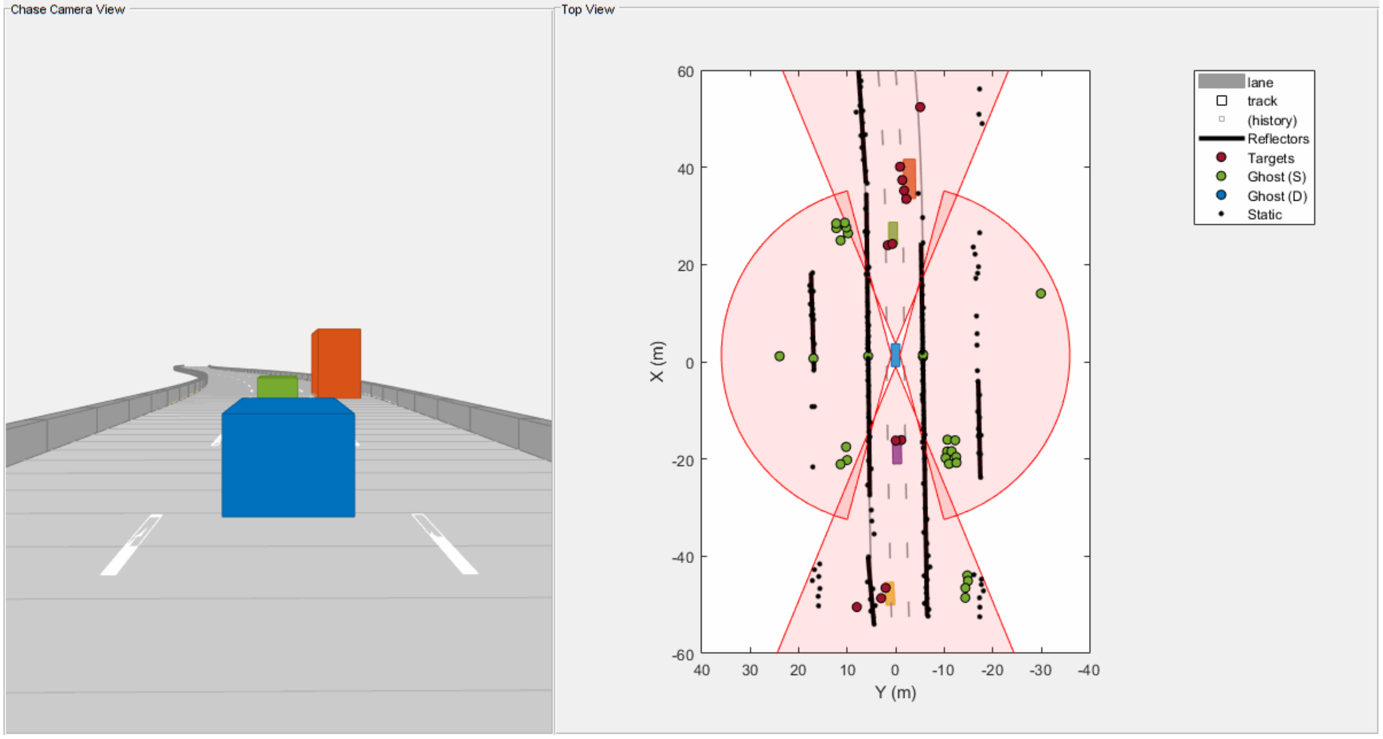
Next, you advance the scenario, use the recorded measurements from the sensors, and process them using the previously described algorithm. You analyze the performance of the tracking algorithm by using the generalized optimal subpattern assignment (GOSPA) metric. You also analyze the performance of the 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, ref

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

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

Results

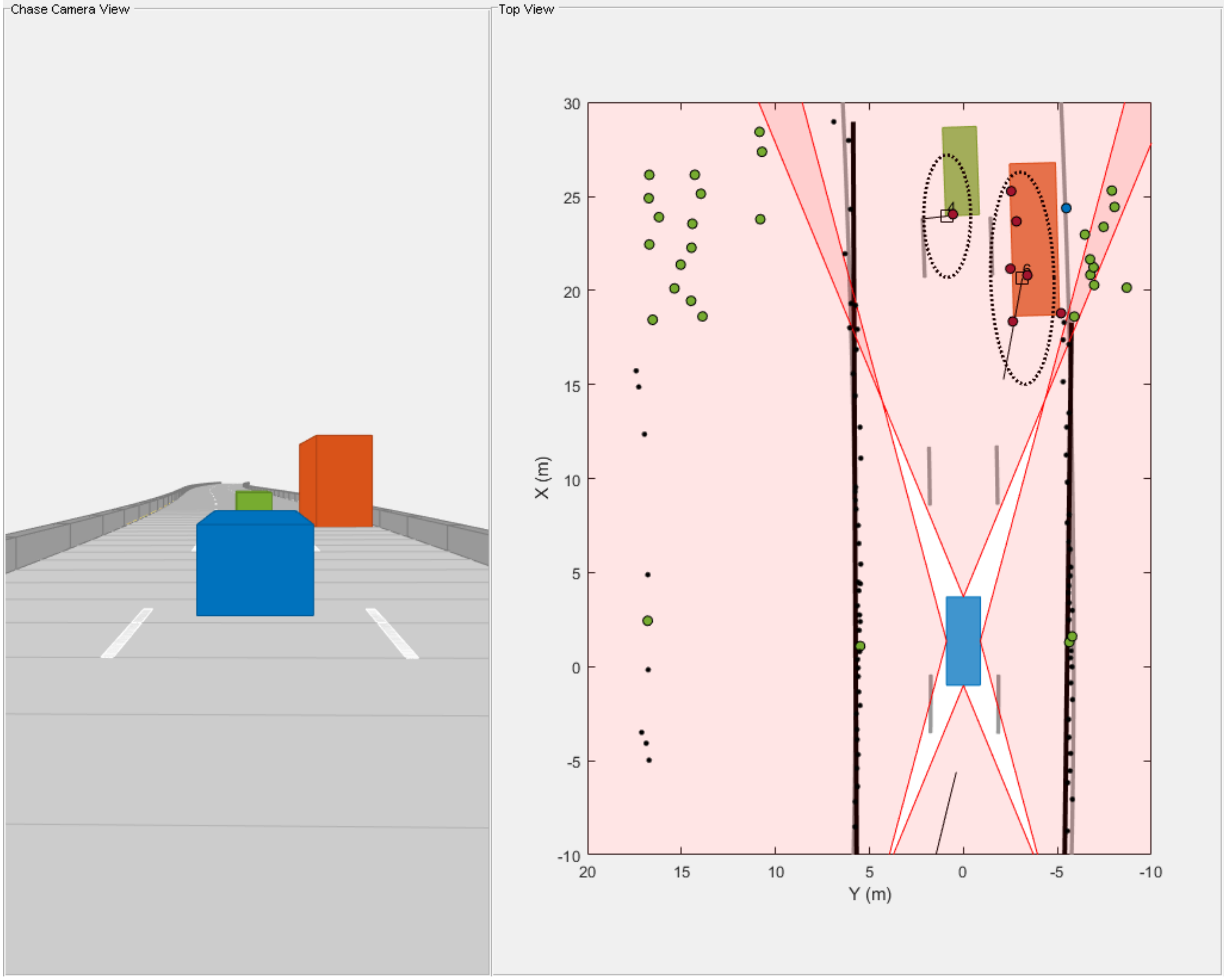
Animation and snapshot analysis

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.



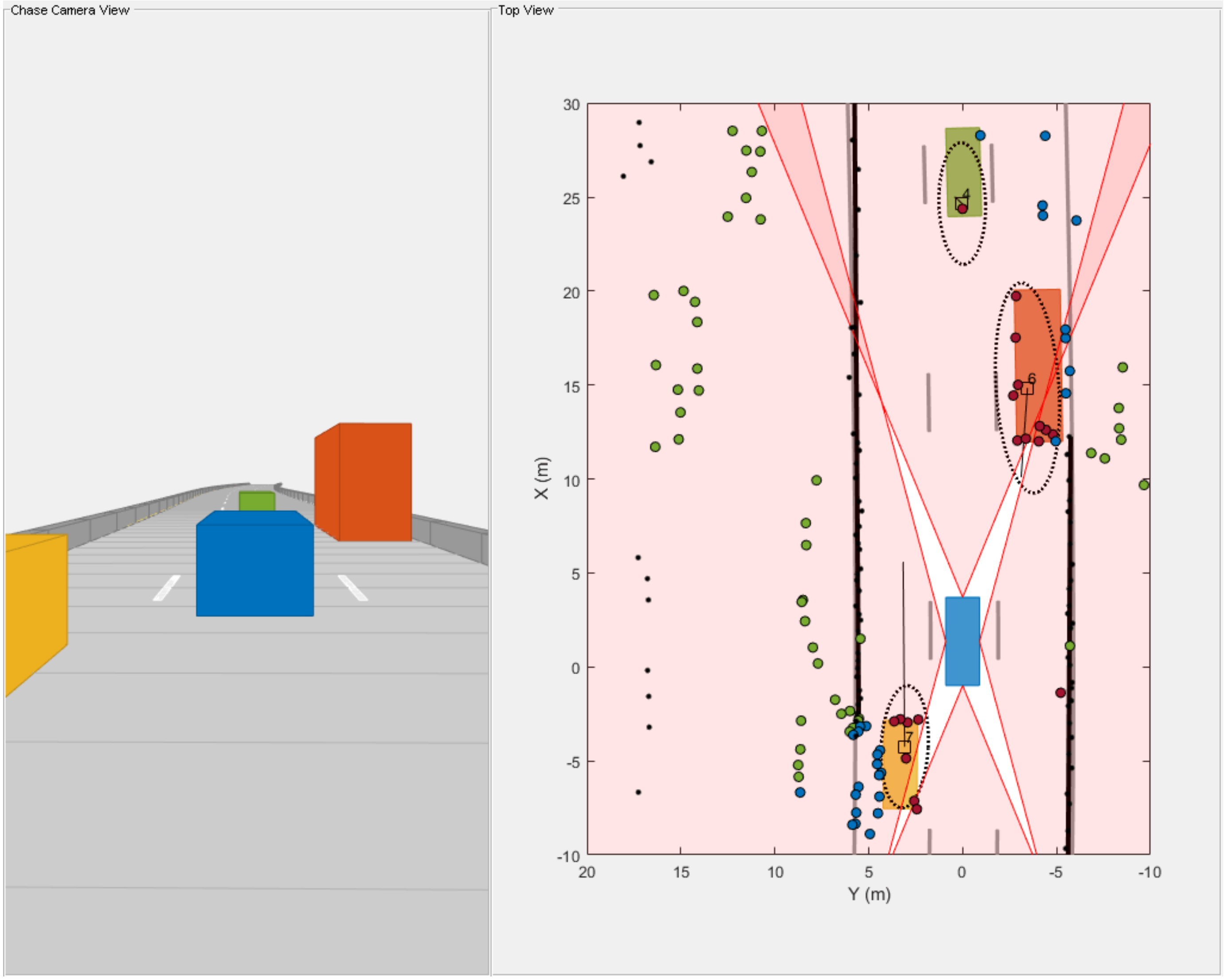
Next, you analyze the performance of the algorithm using different snapshots captured during the simulation. The snapshot below is captured at 3 seconds and shows the situation in front of the ego vehicle. At this time, the ego vehicle is approaching the slow-moving truck, and the left radar sensor observes 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
```

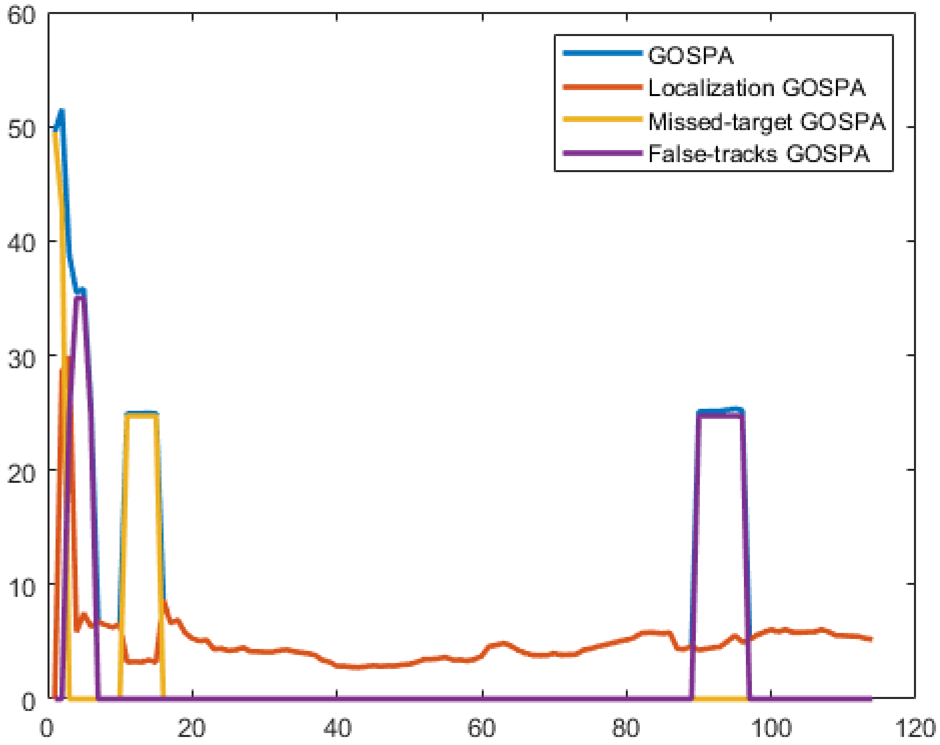


Also notice the passing vehicle denoted by the yellow car on the left of the ego vehicle. The detections, which seem to originate from the nonvisible surface of the yellow vehicle, are two-bounce detections of the barriers, reflected via the front face of the passing vehicle. These ghost detections are 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',{'Ta
disp('True Information');disp(numElemsTable);
```

```
True Information
                   Number of Detections
                   ____________________

    Targets                1974
    Ghost (S)              3203
    Ghost (D)               847
    Environment           27039
    Clutter                 123
```

```
% 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         90.53        0.61         8.36          0.51          0   
    Ghost (S)        3.18       86.26        10.24          0.31          0   
    Ghost (D)          17           0           83             0          0   
    Environment      1.05        2.88         3.96          92.1          0   
    Clutter         13.82       67.48        17.07          1.63          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 1-0 . 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.

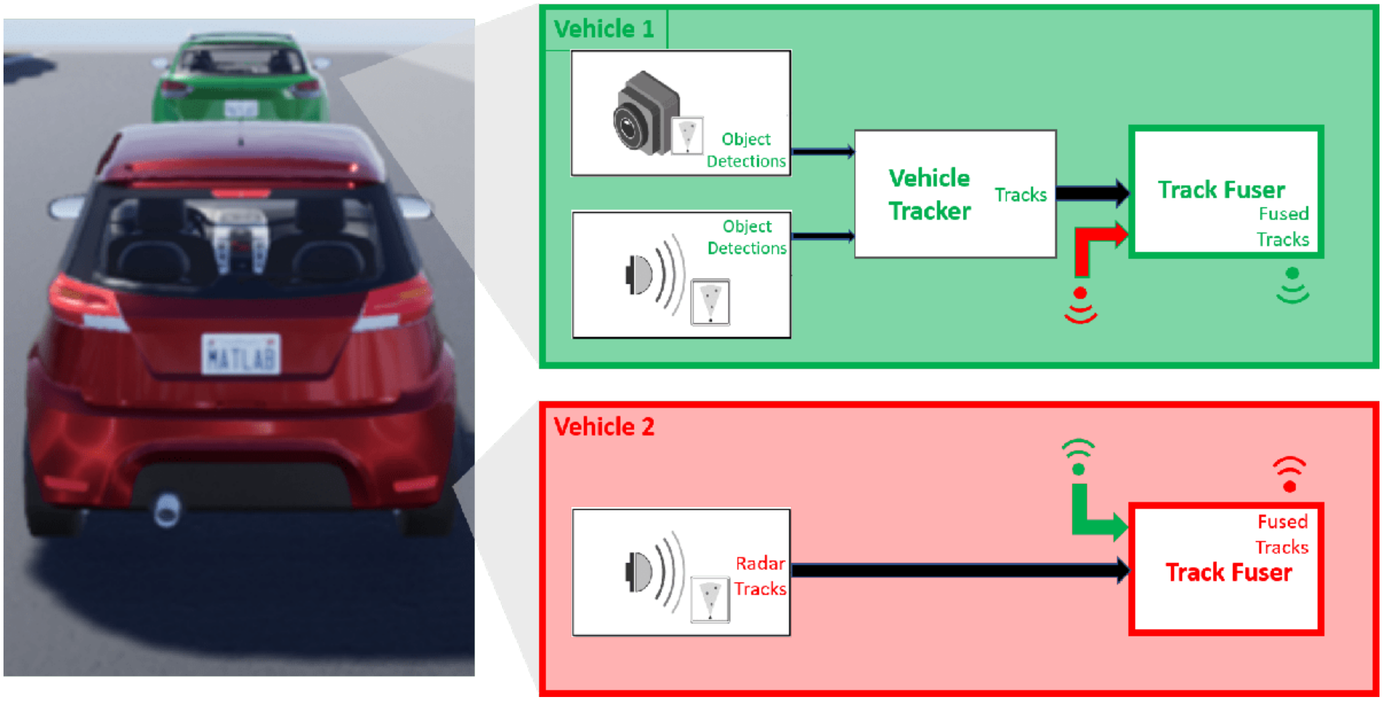
While sensor fusion across multiple vehicles is beneficial, most vehicles are required to meet certain safety requirements even if only internal sensors are available. Therefore, the vehicle is likely to be equipped with a tracker, a track fuser, or both. These tracking algorithms provide situational 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', [10
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

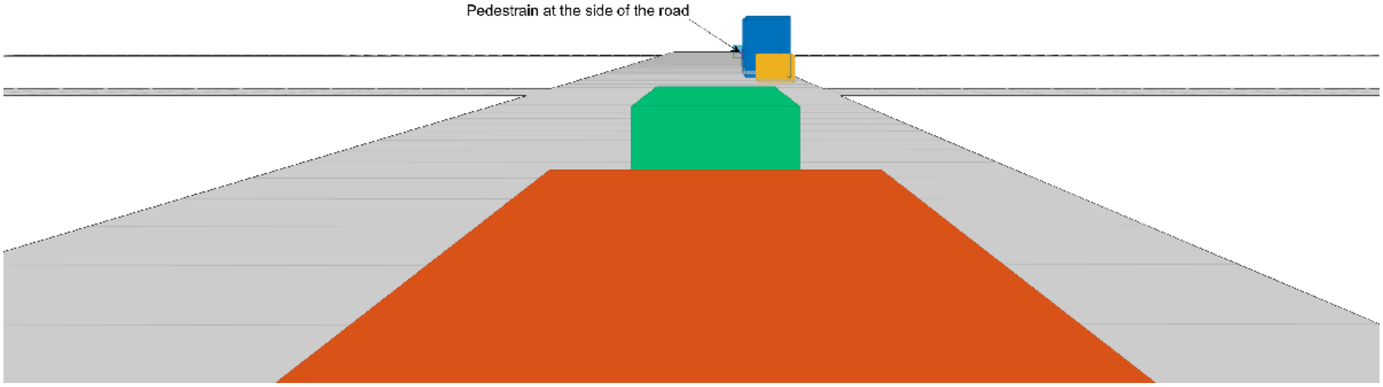
The following scenario shows two vehicles driving down a street. Vehicle 1 is the lead vehicle and is equipped with two forward-looking sensors: a short-range radar sensor and a vision sensor. Vehicle 2, driving 10 meters behind vehicle 1, is equipped with a long-range radar. The right side of the street 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.



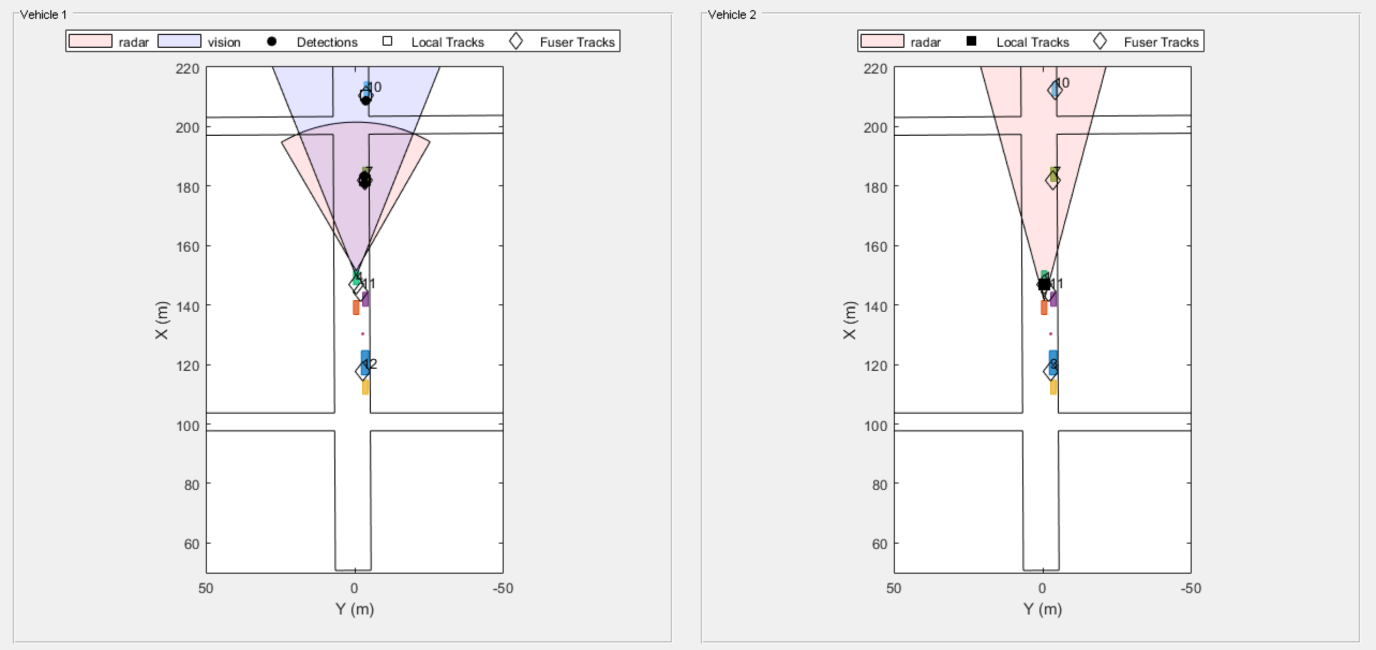
```
% Define each vehicle as a combination of an actor, sensors, a tracker, and plotters
v1=struct('Actor',{vehicle1},'Sensors',{sensors(attachedVehicle==1)},'Tracker',{v1Tracker},'DetP
v2=struct('Actor',{vehicle2},'Sensors',{sensors(attachedVehicle==2)},'Tracker',{{}},'DetPlotter'
```

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;
```

```
        % 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
```

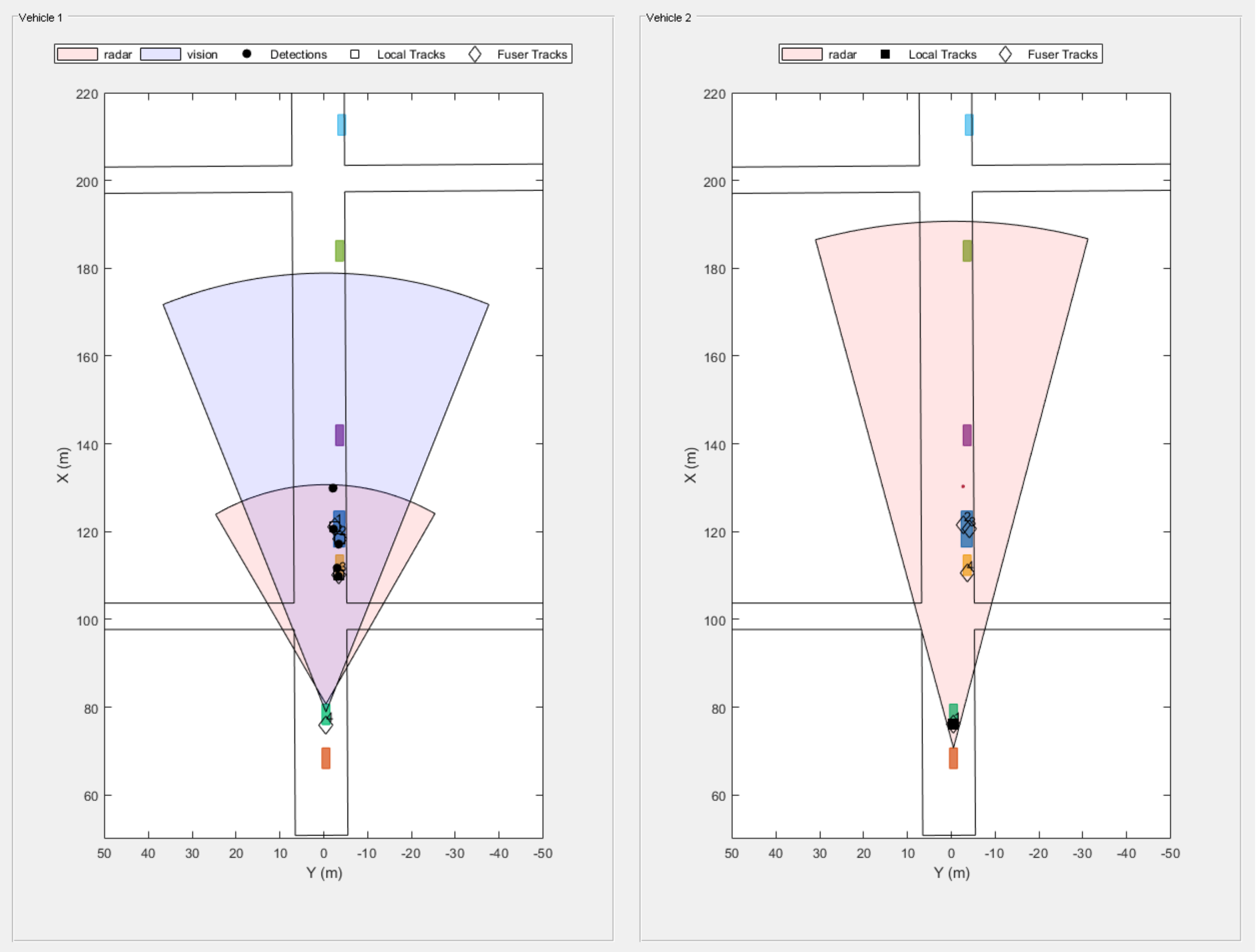


The figure shows the scene and tracking results at the end of the scenario. Subsequent sections of this example analyze the tracking results at key times.

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.

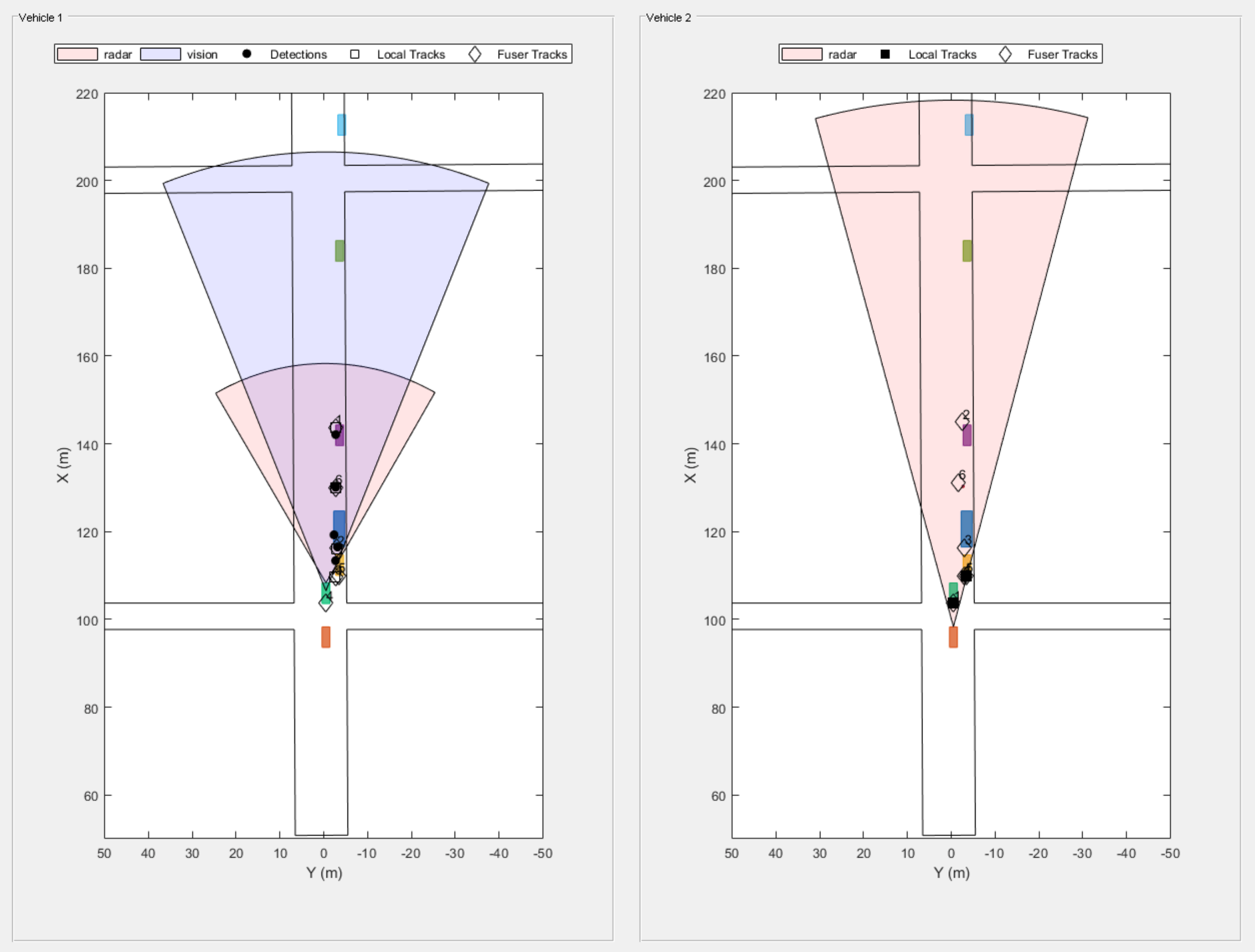
```
showsnap(snaps,1)
```



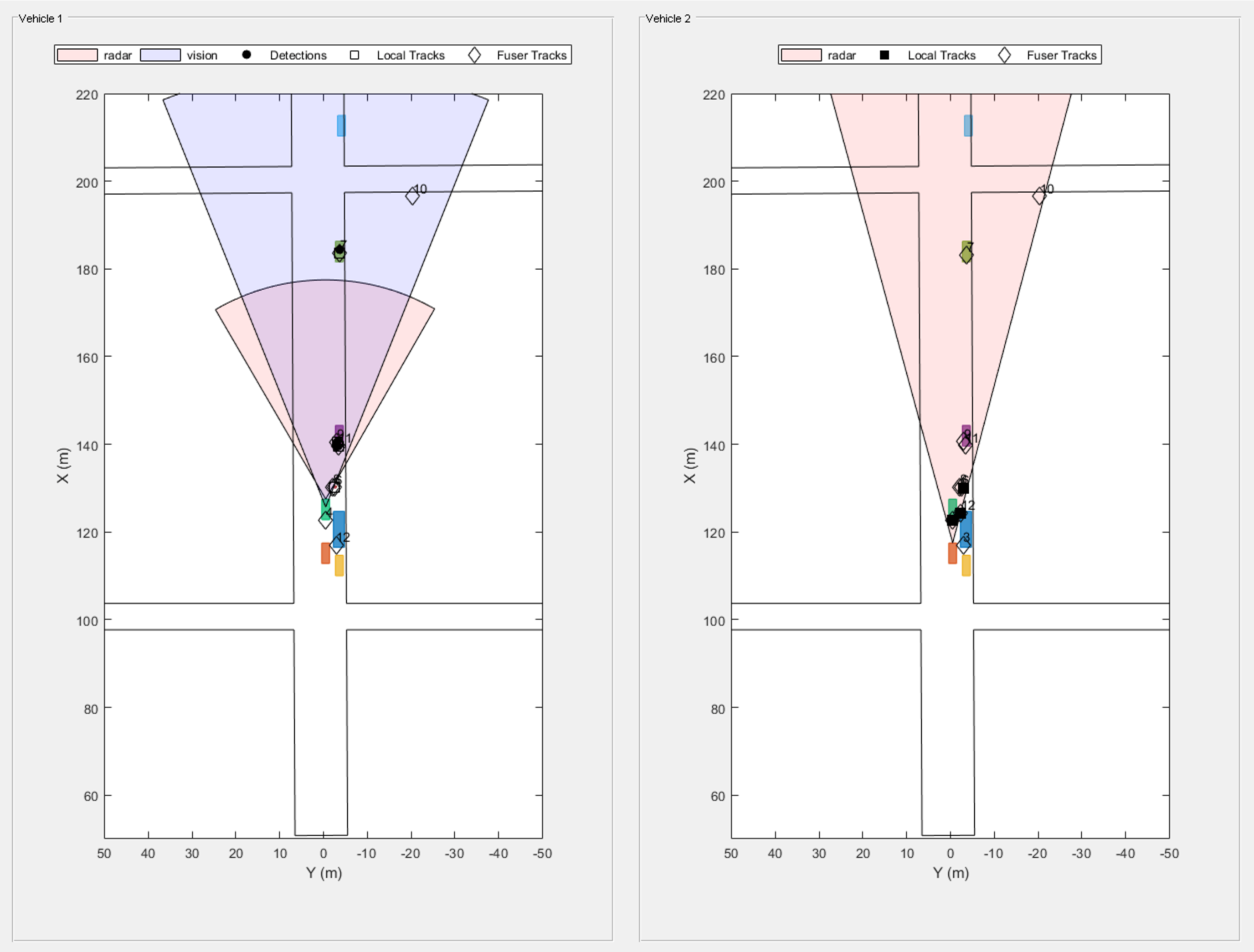
Analyze Tracking of Pedestrian at Side of Street

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)
```

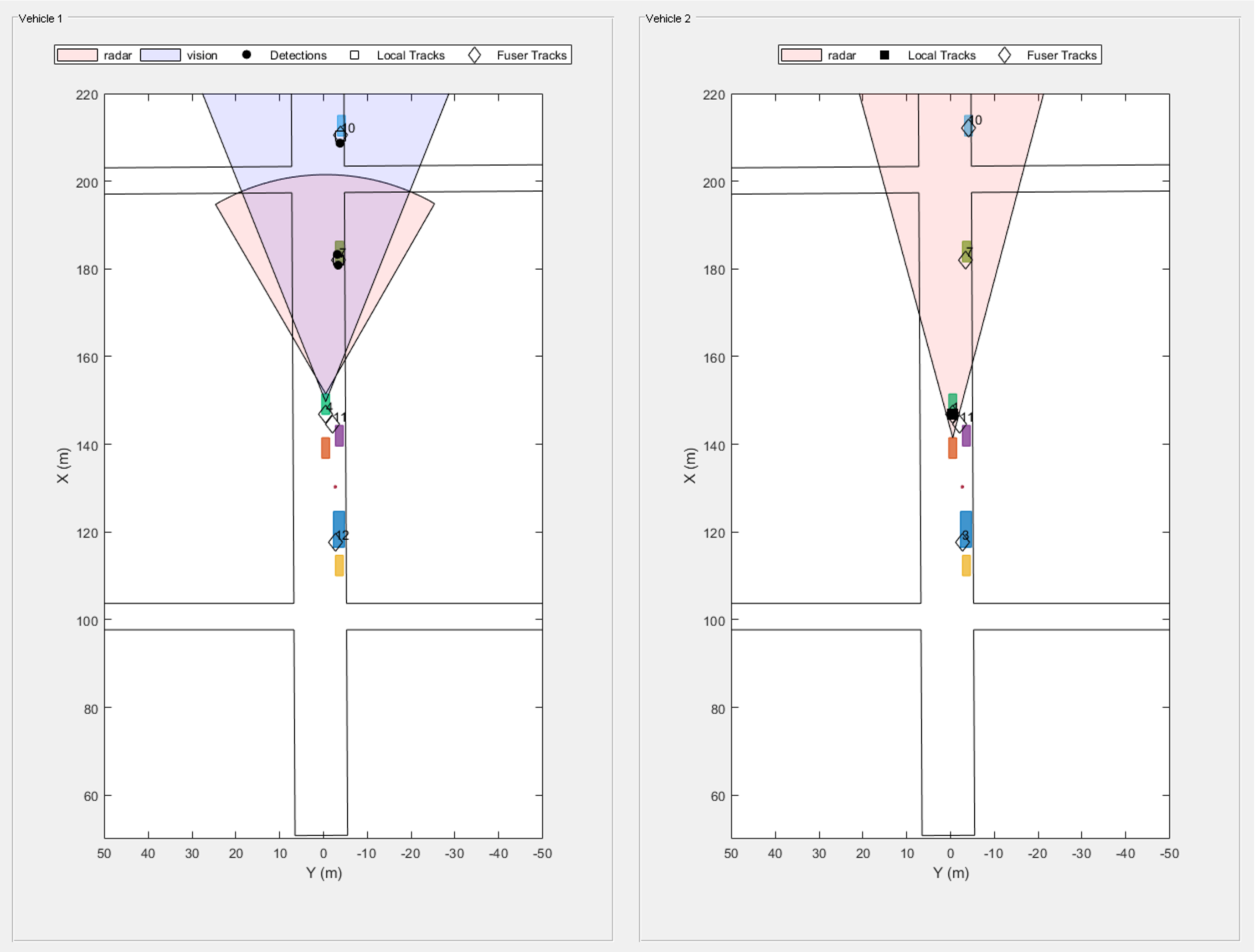


Avoiding Rumor Propagation

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.

```
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);
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.

```
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;
        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
```

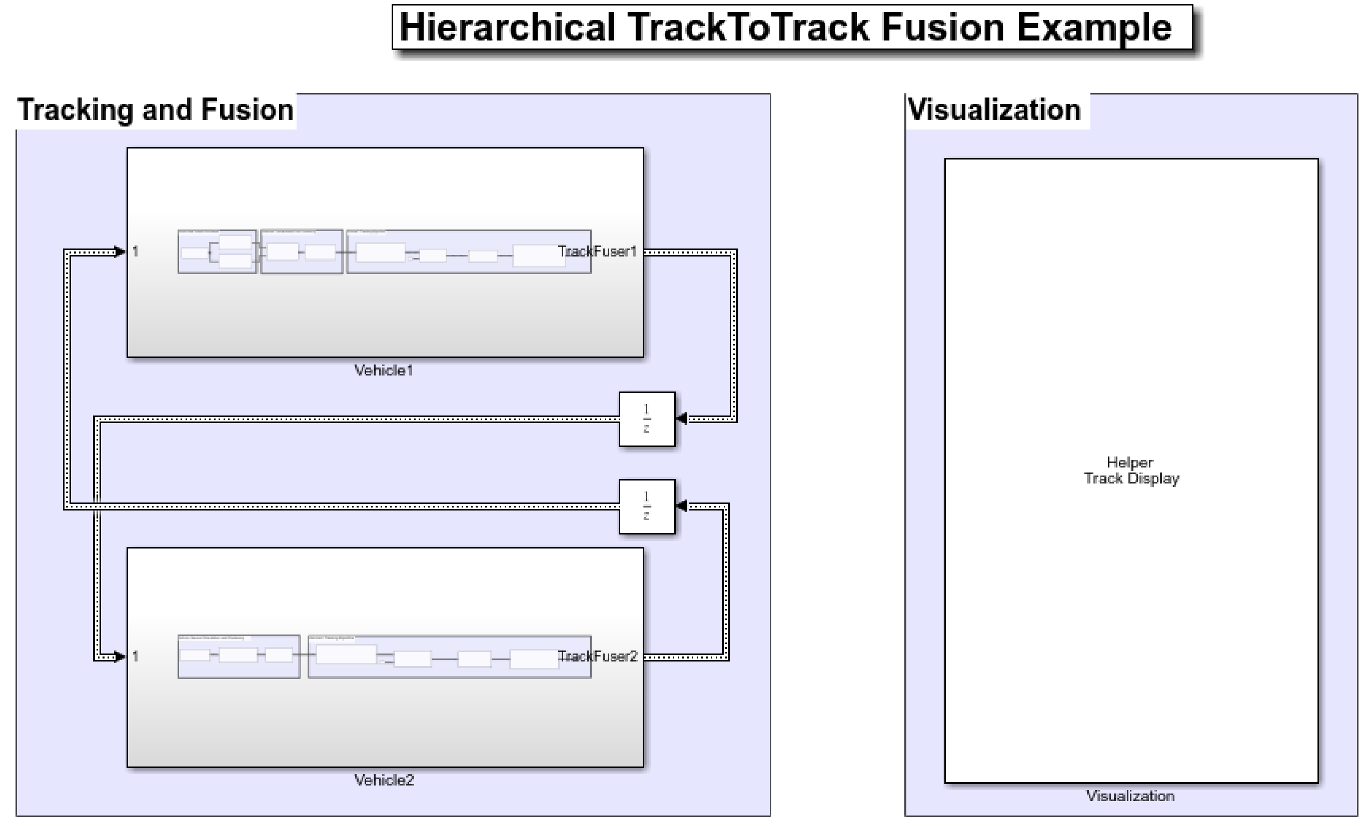
Track-to-Track Fusion for Automotive Safety Applications in Simulink

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 fuser block. In the example, each vehicle perform tracking independently as well as fuse 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

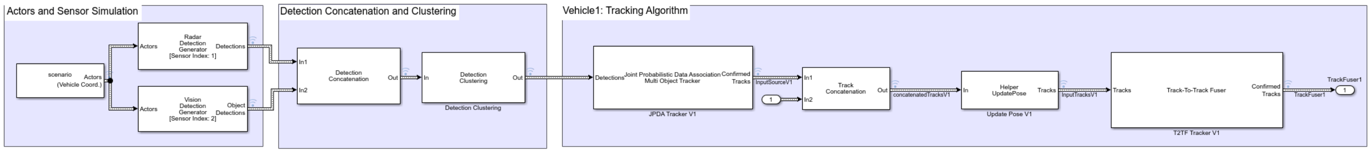


Prior to running this example, the `drivingScenario` object was used to create the same scenario defined in “Track-to-Track Fusion for Automotive Safety Applications” (Sensor Fusion and Tracking Toolbox). The roads and actors from this scenario were then saved to the scenario object file `TrackToTrackFusionScenario.mat`.

Tracking and Fusion

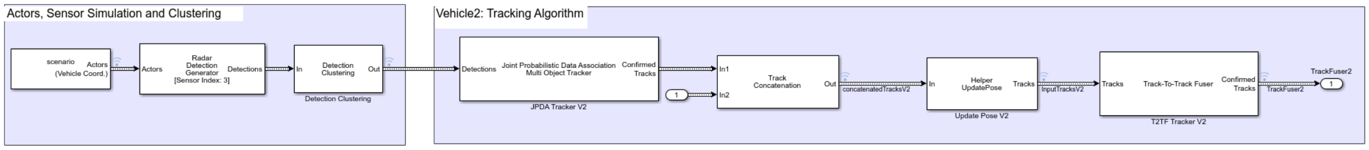
In the Tracking and Fusion section of the model there are two subsystems which implements the target tracking and fusion capabilities of `Vehicle1` and `Vehicle2` in this scenario.

Vehicle1



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 other vehicle's track fuser. 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 these 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 `trackFuser` (Sensor Fusion and Tracking Toolbox) `T2TF Tracker V1` block 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 to be fed into the update of the other vehicle's track fuser in the next time stamp.

Vehicle2



`Vehicle2` subsystem follows similar setup as `Vehicle1` subsystem as described above.

Visualization

The Visualization block is implemented using the MATLAB System block and is defined using `HelperTrackDisplay` block. The block uses `RunTimeObject` parameter of the blocks to display their outputs. See “Access Block Data During Simulation” (Simulink) for further information on how to access block outputs during simulation.

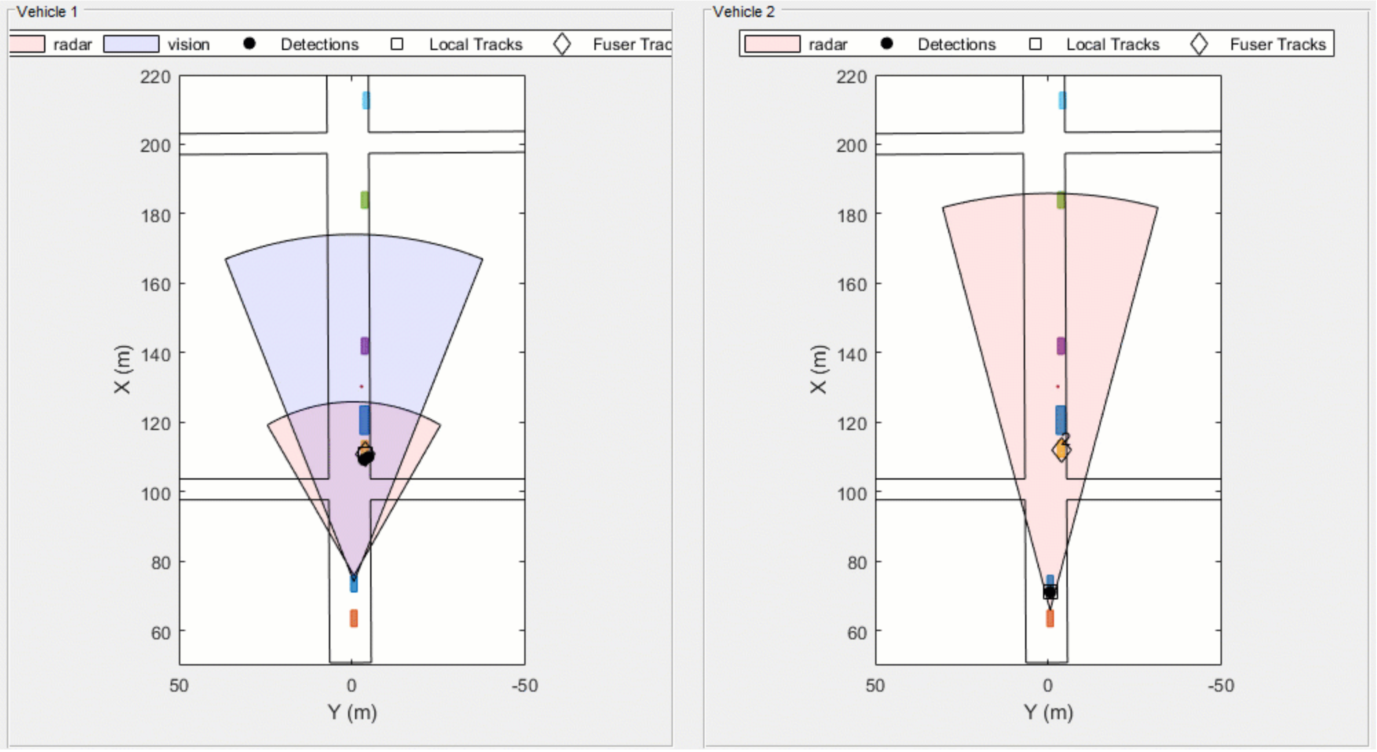
Results

After running the model, you visualize the results as on the figure. The animation below shows the results for this simulation.

The visualization includes two panels. The left panel shows the detections, local tracks, and fused tracks that Vehicle1 generated during the simulation and represents the situational awareness of the Vehicle1. The right panel shows the situational awareness of Vehicle2.

The recorded detections are represented by black circles. The local and fused tracks from Vehicle1 are represented by square and diamond respectively. The local and fused tracks from Vehicle2 represented by a solid black square and diamond. Notice that during the start of simulation, Vehicle1 detects vehicles parked on the right side of the street, and tracks associated with the parked vehicles are confirmed. Currently Vehicle2 only detects Vehicle1 which is immediately in front of it. As the simulation continues the confirmed tracks from Vehicle1 are broadcasts to the fuser on Vehicle2. After fusing the tracks, vehicle2 becomes aware of the objects prior to detecting these objects on its own. Similarly, Vehicle2 tracks are broadcasts to Vehicle1. Vehicle1 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 car on the right side of the street is detected and tracked by Vehicle1. Vehicle2 first becomes aware of the pedestrian by fusing the track from Vehicle1 at around 0.8 seconds. It takes Vehicle2 roughly 3 seconds before it starts detecting the pedestrian using its own sensor. The ability to track a pedestrian based on inputs from Vehicle1 allows Vehicle2 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 used a JPDA tracker block to generate the local tracks.

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.

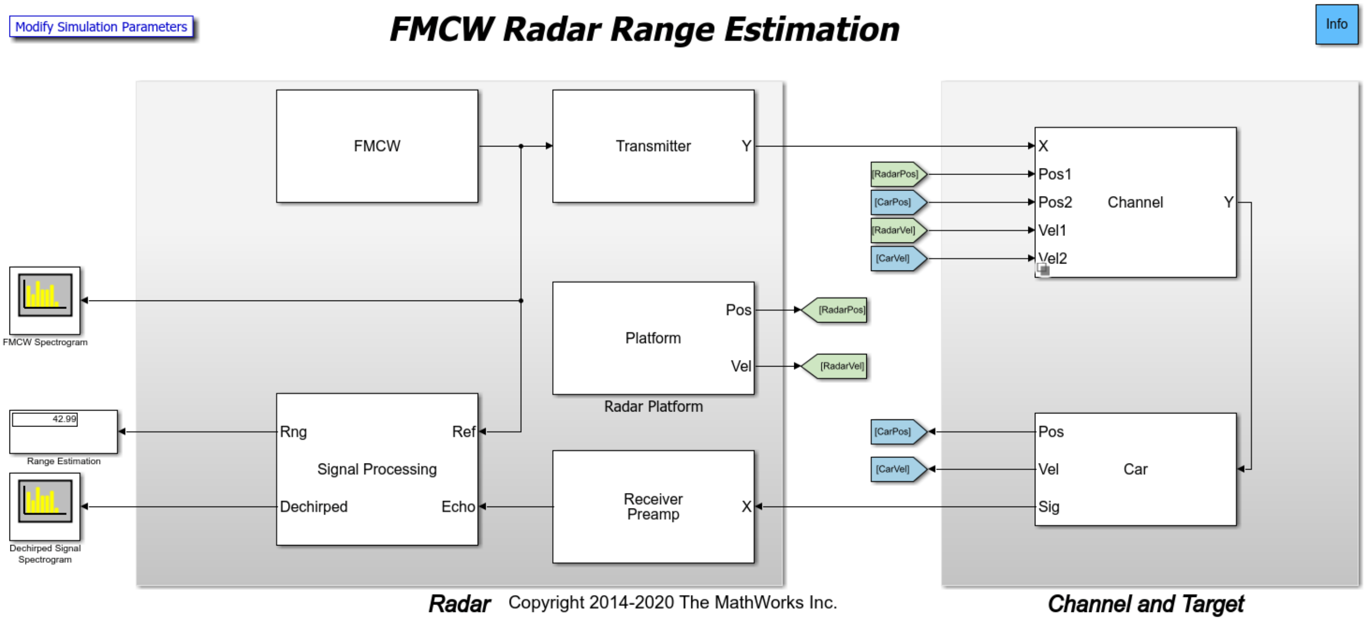
Available Example Implementations

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 1-132 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.



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.

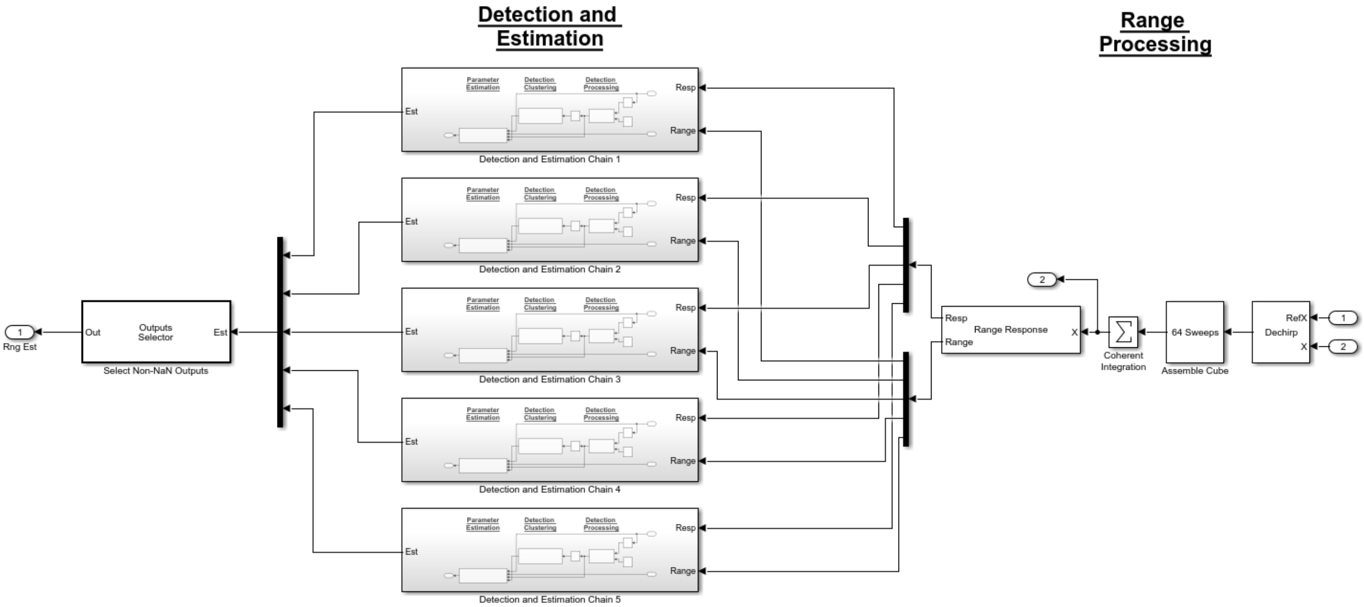
Radar

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.

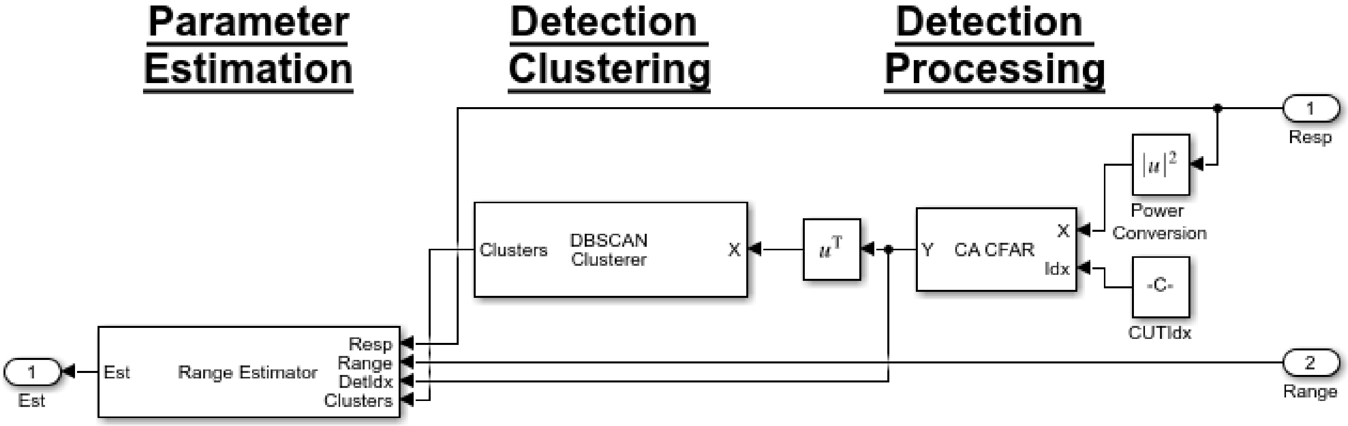
Within the **Radar**, the target echo goes through several signal processing steps before the target range can be estimated. The signal processing subsystem consists of two high-level processing stages.

- 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.

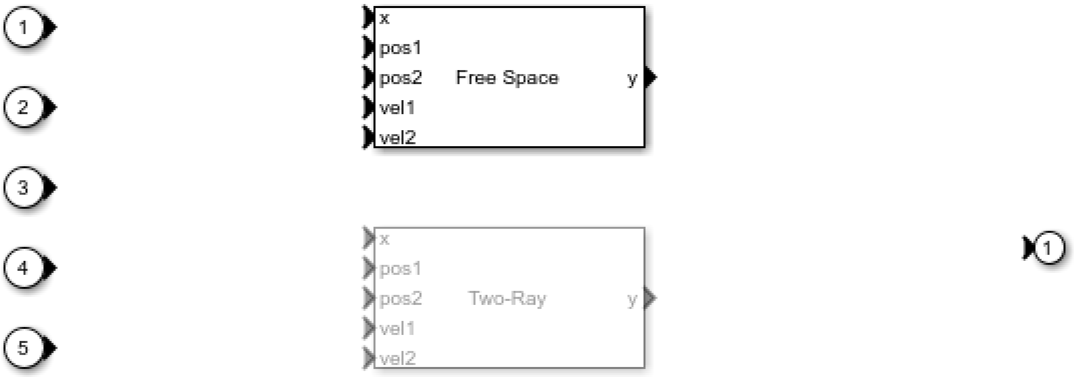
- Detection Processing: The radar data is first passed to a 1-dimensional cell-averaging (CA) constant false alarm rate (CFAR) detector that operates in the range dimension. This block identifies detections or hits.
- Detection Clustering: The detections are then passed to the next step where they are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



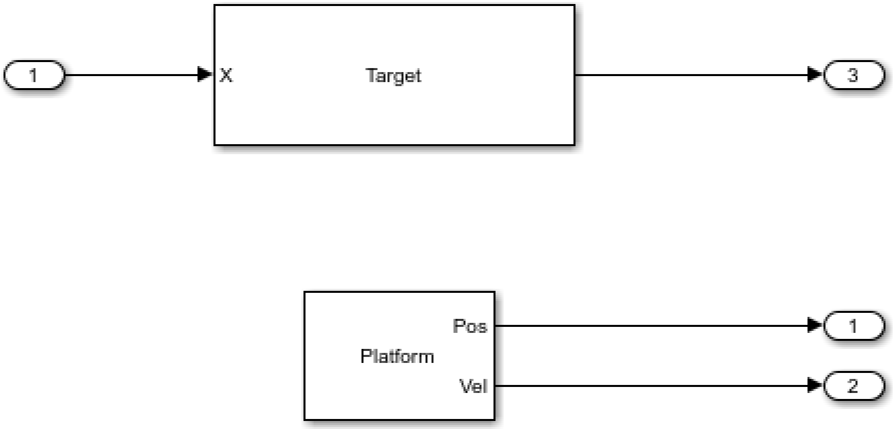
Channel and Target

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.

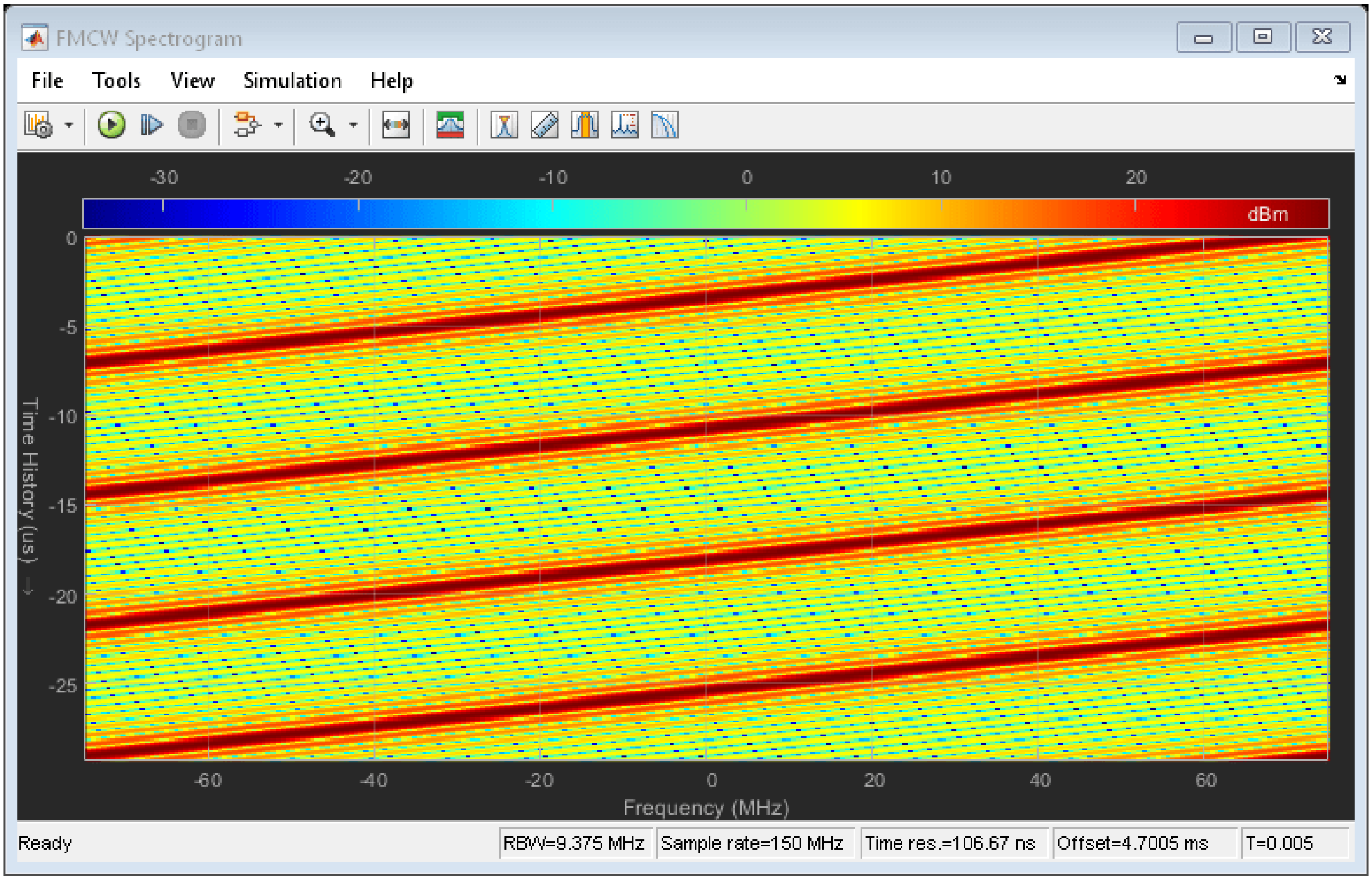
In this model's scenario, the radar vehicle starts at the origin, traveling at 100 km/h (27.8 m/s), while the target vehicle starts at 43 meters in front of the radar vehicle, traveling at 96 km/h (26.7 m/s). The positions and velocities of both the radar and the target vehicles are used in the propagation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

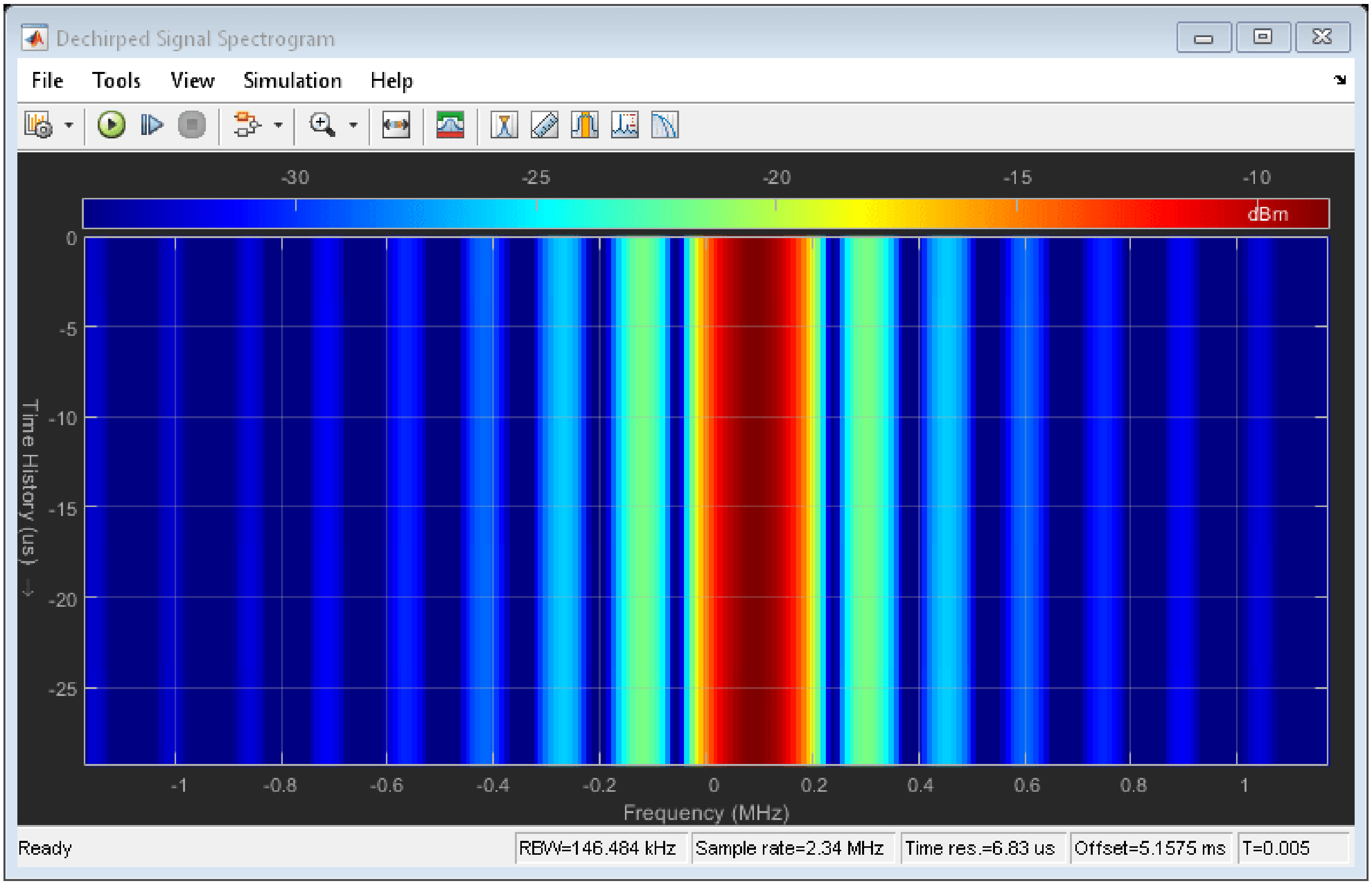
Results and Displays

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.



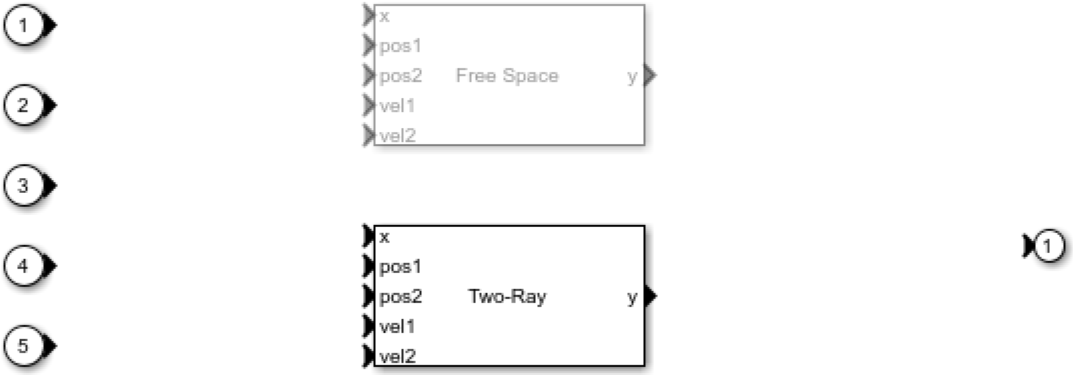
Ready | RBW=9.375 MHz | Sample rate=150 MHz | Time res.=106.67 ns | Offset=4.7005 ms | T=0.005

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.

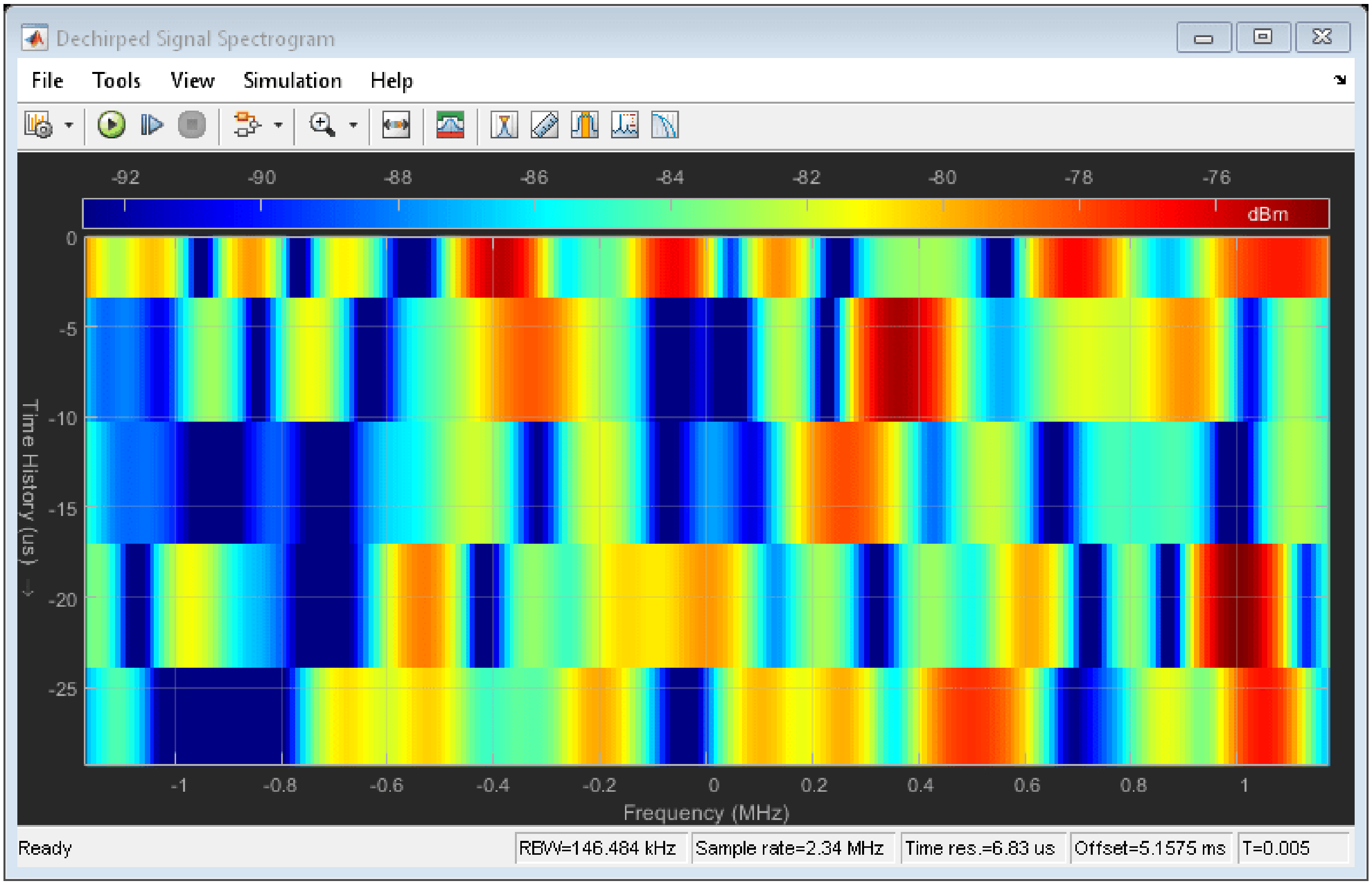


Ready | RBW=146.484 kHz | Sample rate=2.34 MHz | Time res.=6.83 us | Offset=5.1575 ms | T=0.005

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.

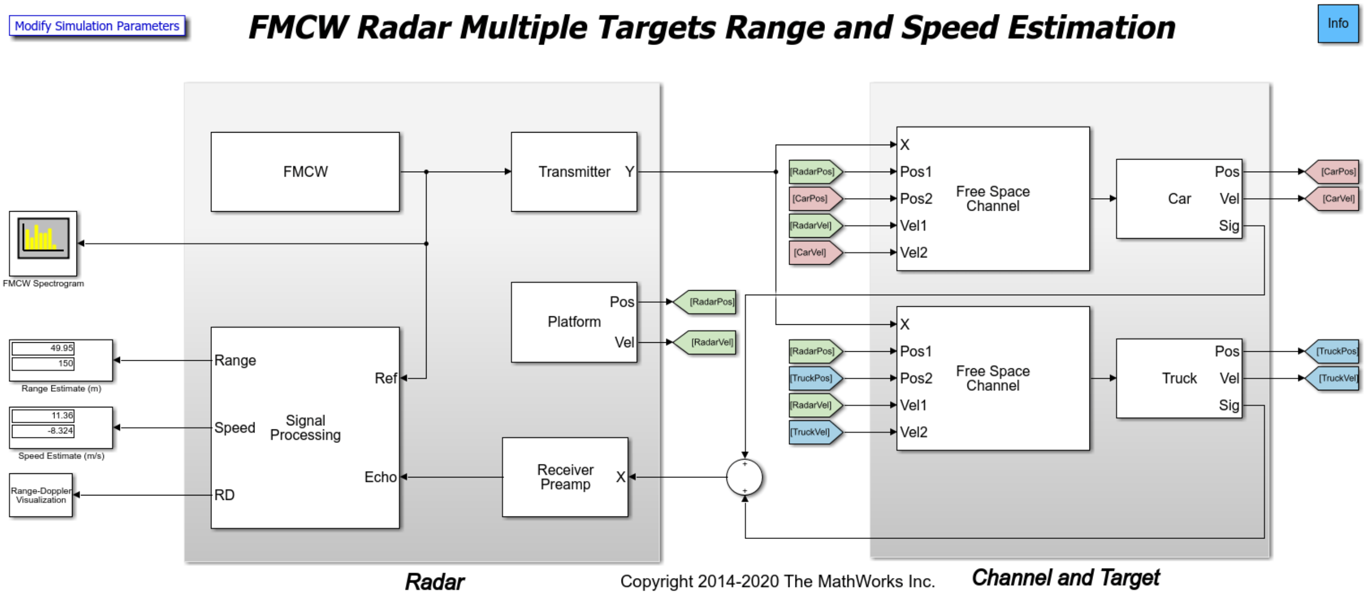


RBW=146.484 kHz | Sample rate=2.34 MHz | Time res.=6.83 us | Offset=5.1575 ms | T=0.005

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.



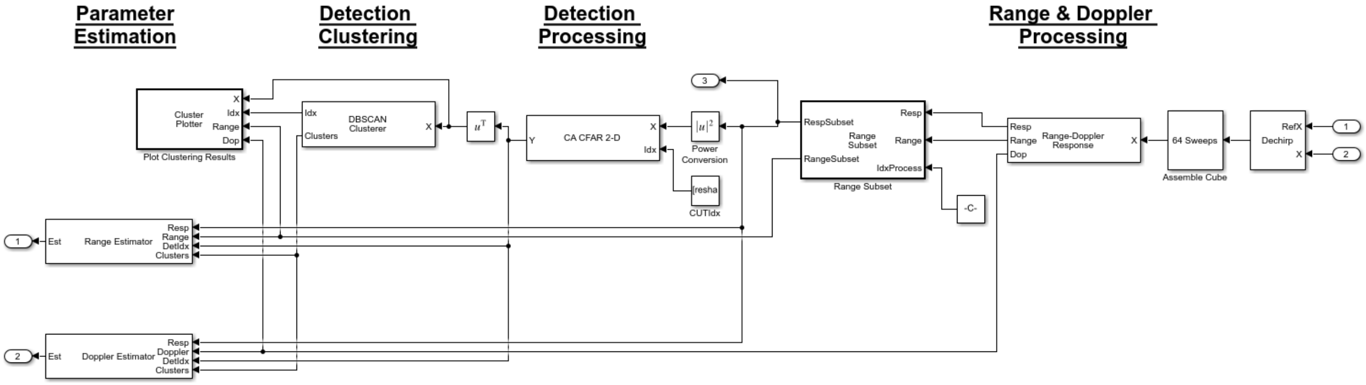
The model is essentially the same as the previous example with 4 primary differences. This model:

- 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

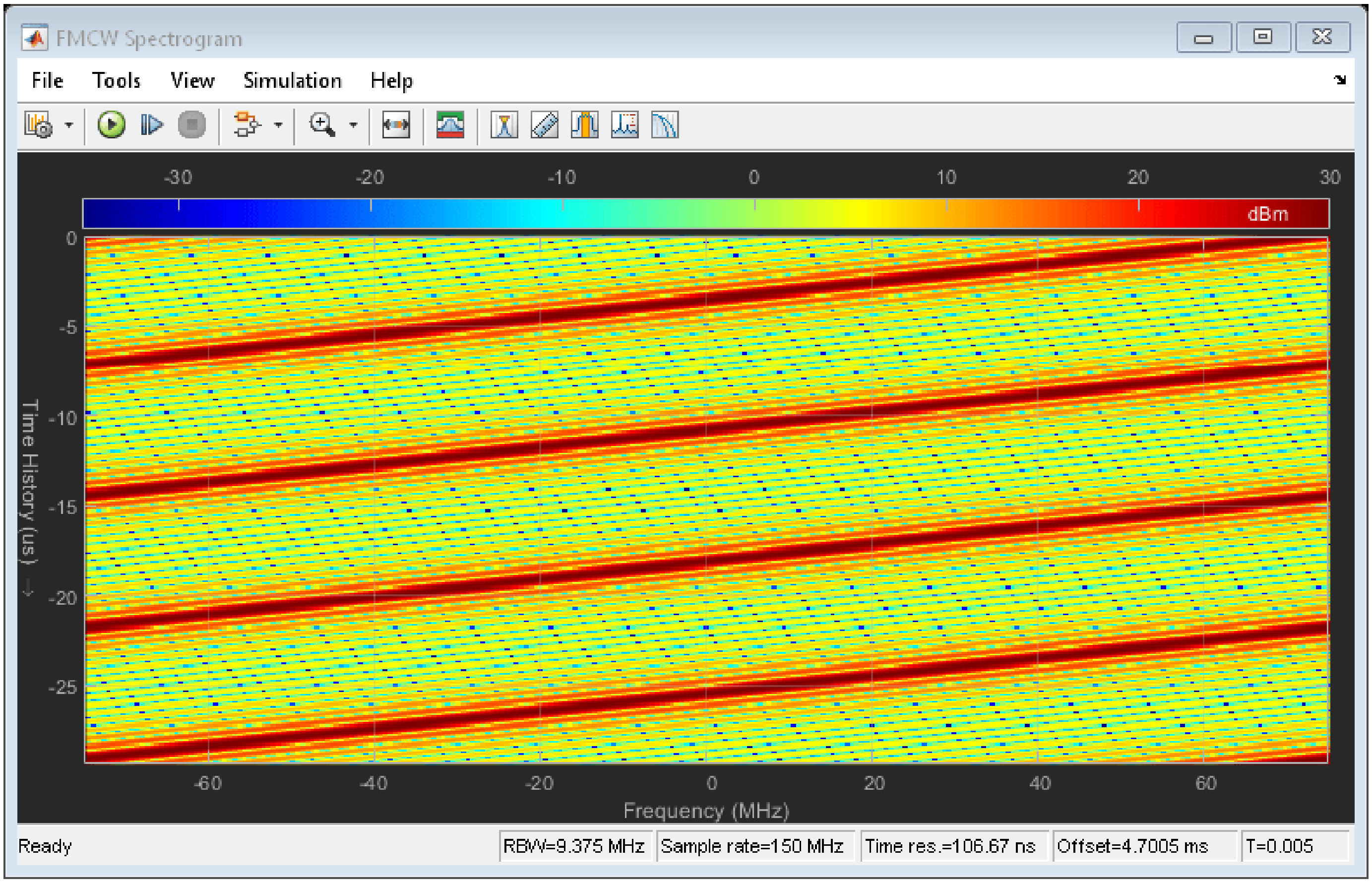
There are now two target vehicles in the scene, labeled as Car and Truck, and each vehicle has an associated propagation channel. The Car starts 50 meters in front of the radar vehicle and travels at a speed of 60 km/h (16.7 m/s). The Truck starts at 150 meters in front of the radar vehicle and travels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

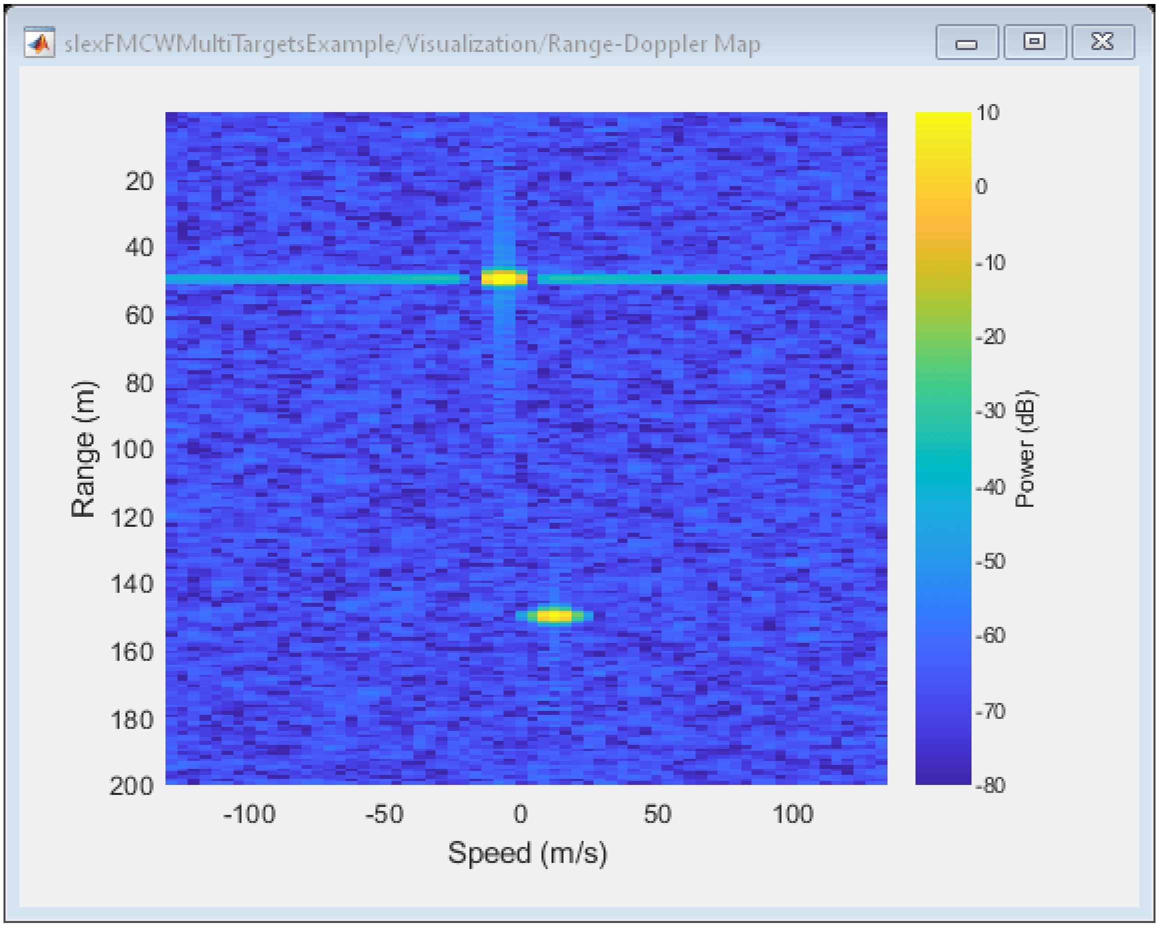
Results and Displays

The FMCW signal shown below is the same as in the previous model.



RBW=9.375 MHz | Sample rate=150 MHz | Time res.=106.67 ns | Offset=4.7005 ms | T=0.005

The two targets can be visualized in the range-Doppler map below.

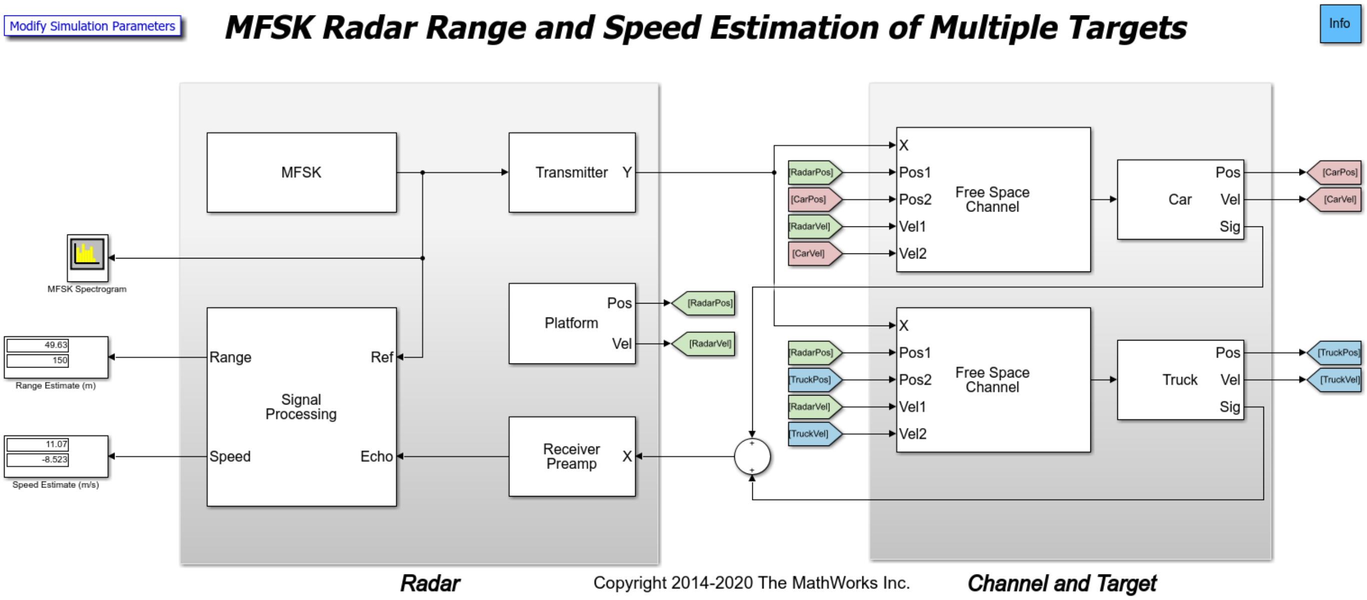


The map correctly shows two targets: one at 50 meters and one at 150 meters. Because the radar can only measure the relative speed, the expected speed values for these two vehicles are 11.1 m/s and -8.3 m/s, respectively, where the negative sign indicates that the Truck is moving away from the radar 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.

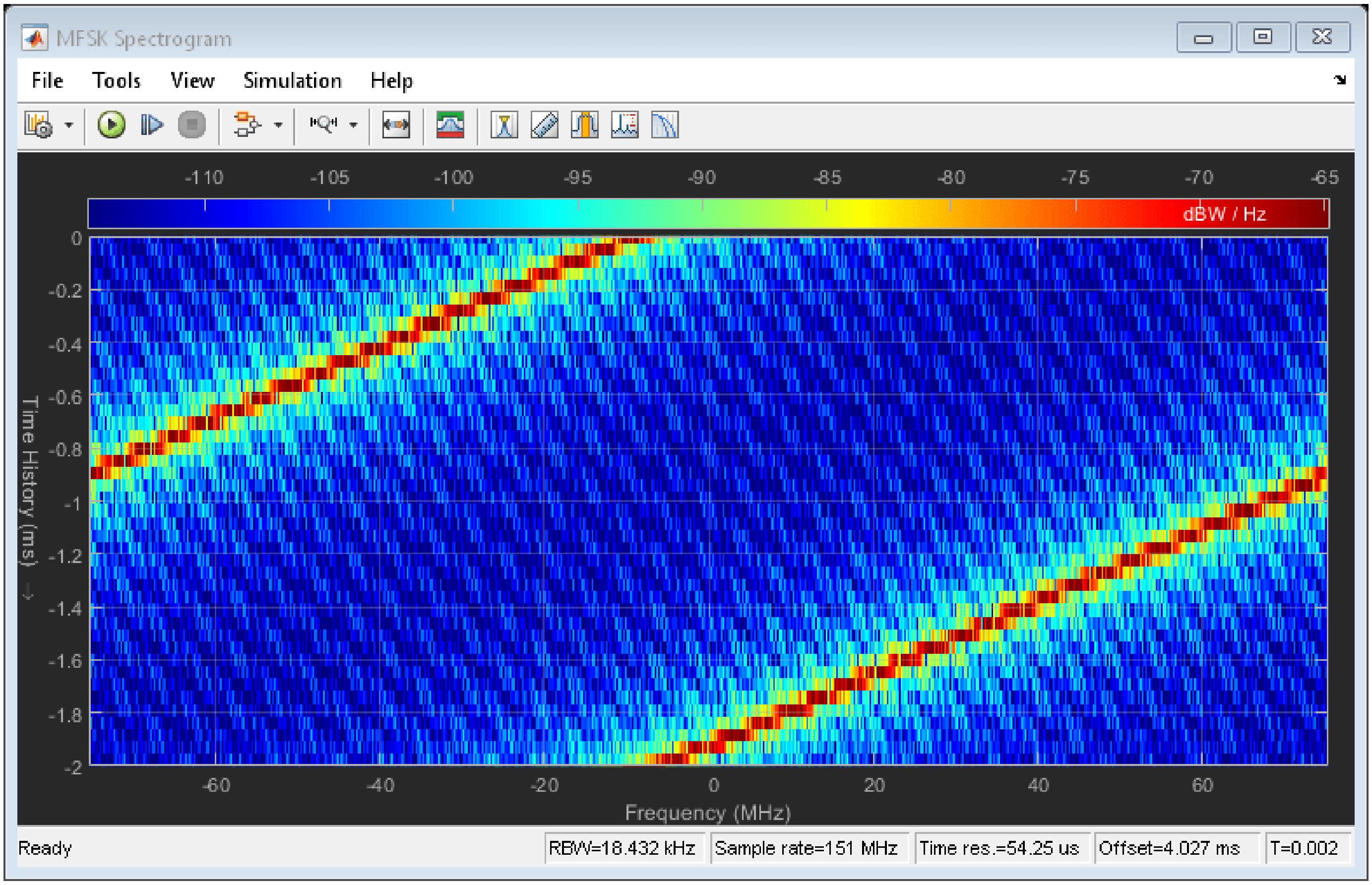
MFSK Radar Range and Speed Estimation of Multiple Targets

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.

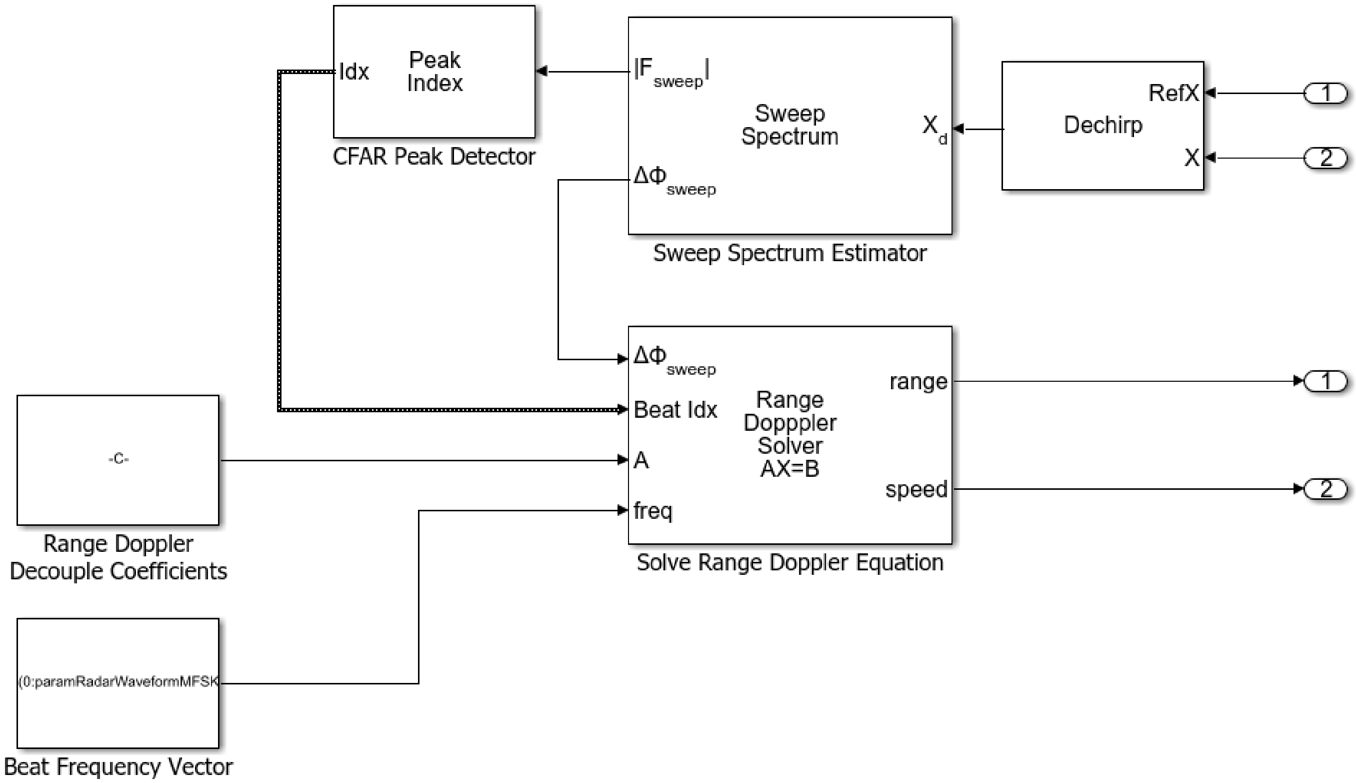


The primary differences between this model and the previous are in the waveform block and the signal processing subsystem. The MFSK waveform essentially consists of two FMCW sweeps with a fixed frequency offset. The sweep in this case happens at discrete steps. From the parameters of the MFSK waveform 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.



RBW=18.432 kHz | Sample rate=151 MHz | Time res.=54.25 us | Offset=4.027 ms | T=0.002

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

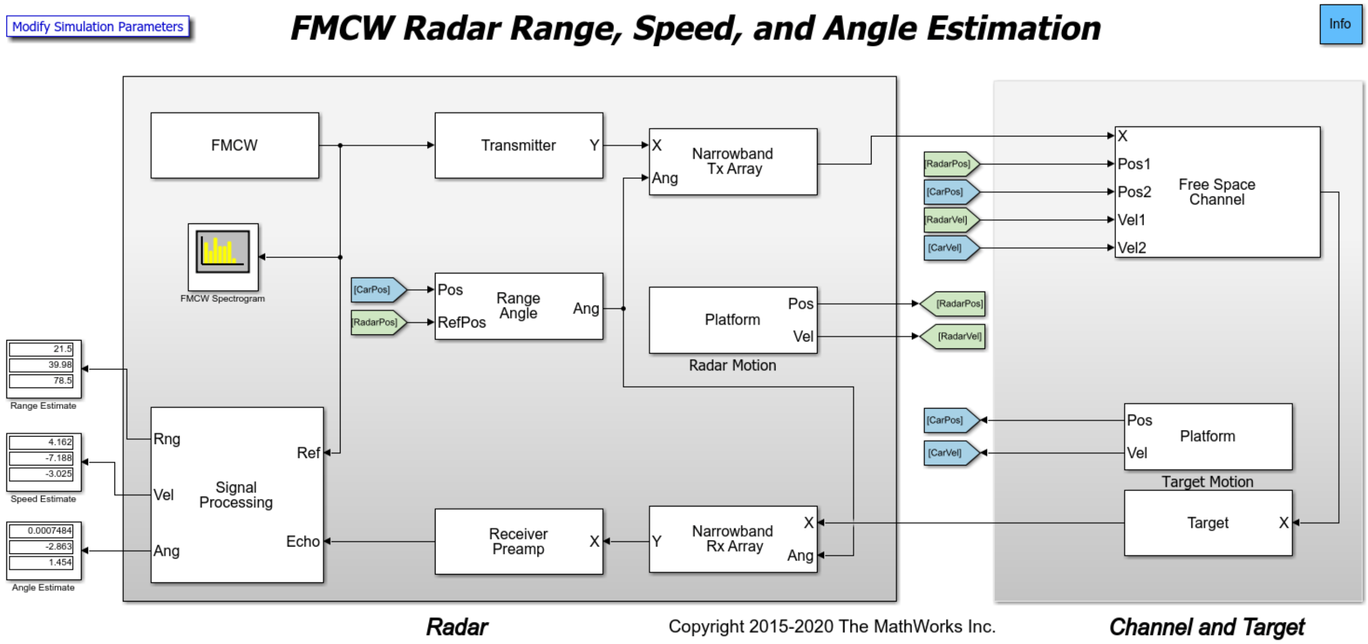
Several dialog parameters of the model are calculated by the helper function helperslexMFSKMultiTargetsParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields 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.

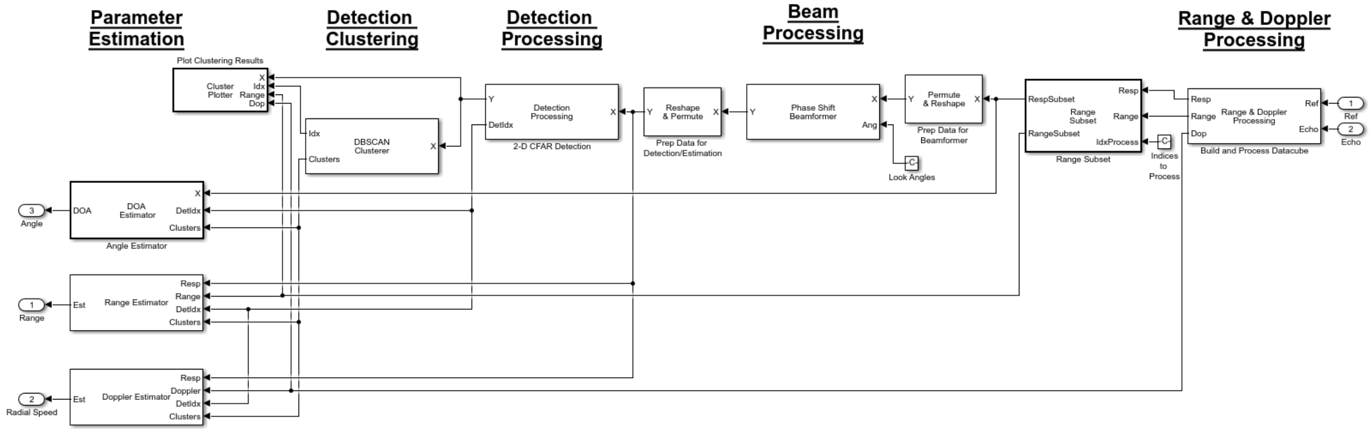


In this scenario, the radar is traveling in the center lane of a highway at 100 km/h (27.8 m/s). The first target vehicle is traveling 20 meters ahead in the same lane as the radar at 85 km/h (23.6 m/s). The second target vehicle is traveling at 125 km/h (34.7 m/s) in the right lane and is 40 meters 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 1-132 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.

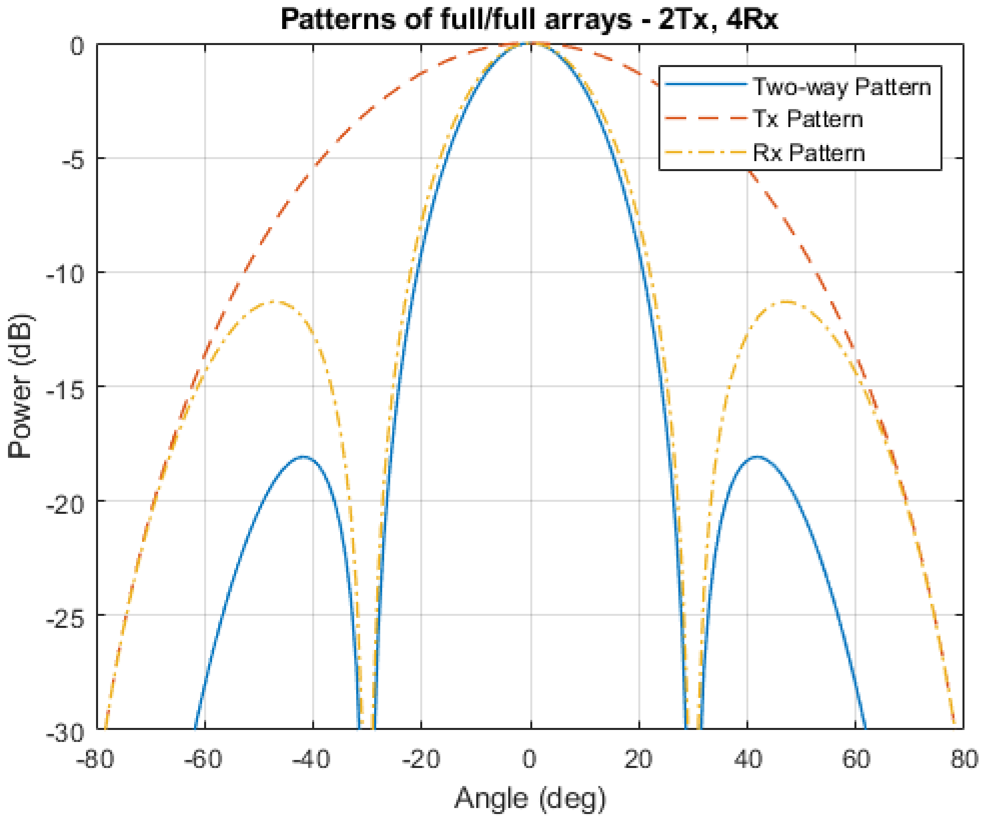
Virtual Array

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;
```

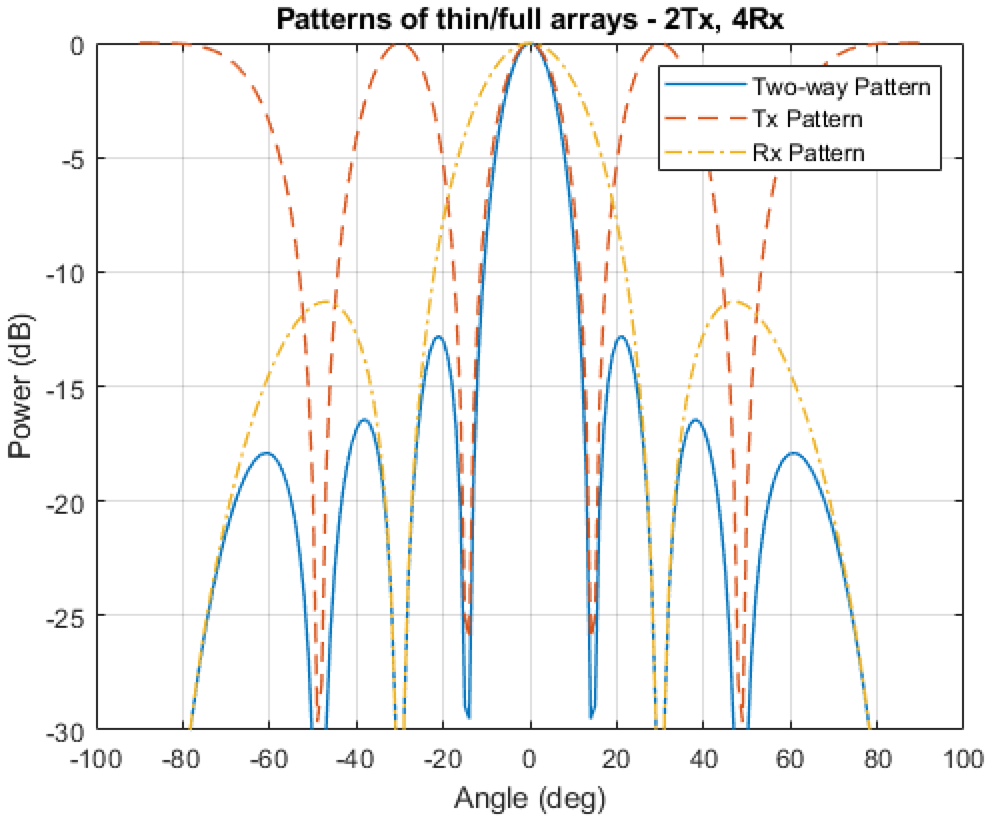
If both arrays have half-wavelength spacing, which are sometimes referred to as full arrays, then the two-way pattern is close to the receive array pattern.

```
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',...
    {'-','--','-.'});
```

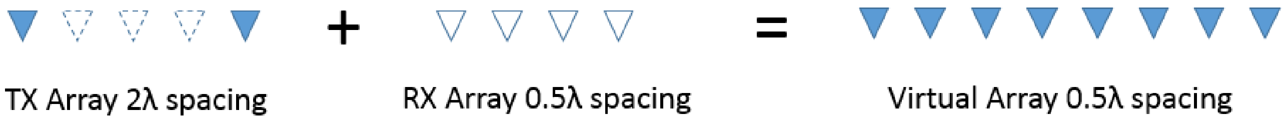


If the full transmit array is replaced with a thin array, meaning the element spacing is wider than half wavelength, then the two-way pattern has a narrower beamwidth. Notice that even though the thin transmit array has grating lobes, those grating lobes are not present in the two-way pattern.

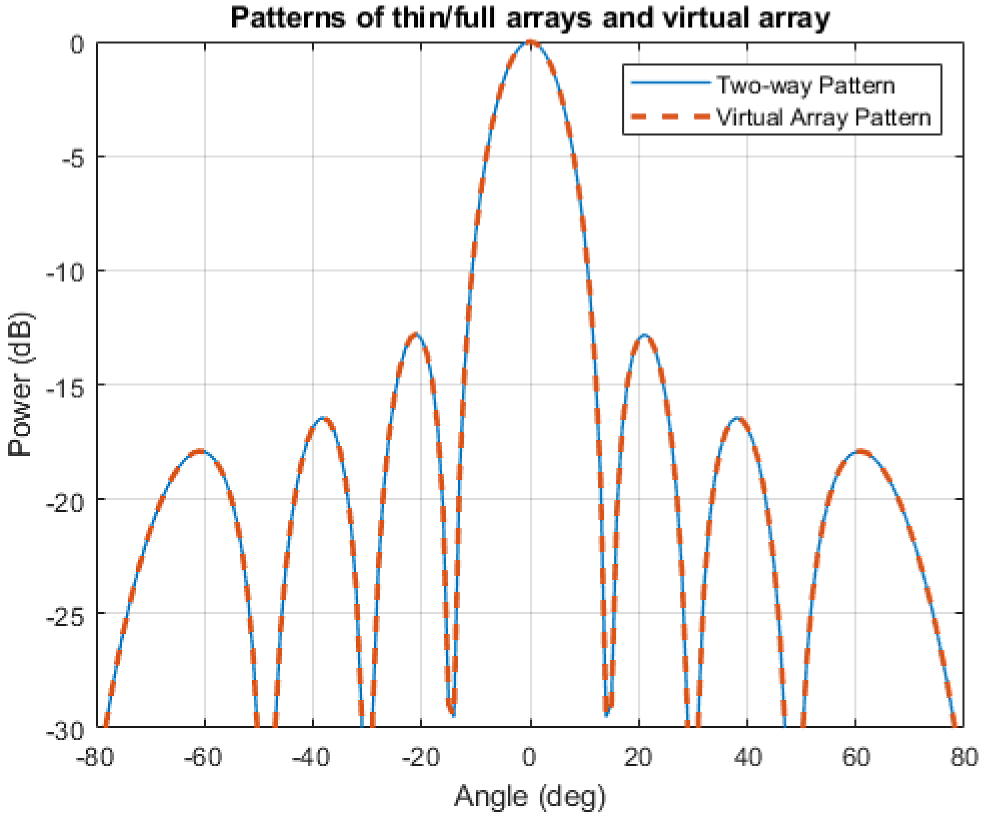
```
dt = Nr*lambda/2;
txarray = phased.ULA(Nt,dt);
pattx = pattern(txarray,fc,ang,0,'Type','powerdb');
pat2way = pattx+patrx;
helperPlotMultipledBPattern(ang,[pat2way pattx patrx],[-30 0],...
    {'Two-way Pattern','Tx Pattern','Rx Pattern'},...
    'Patterns of thin/full arrays - 2Tx, 4Rx',...
    {'-','--','-.'});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



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.

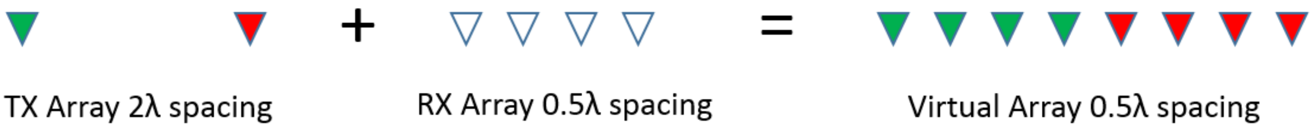


```
varray = phased.ULA(Nt*Nr,dr);
patv = pattern(varray,fc,ang,0,'Type','powerdb');
helperPlotMultipledBPattern(ang,[pat2way patv],[-30 0],...
    {'Two-way Pattern','Virtual Array Pattern'},...
    'Patterns of thin/full arrays and virtual array',...
    {'-','--'},[1 2]);
```



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 1-132 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
```

Virtual Array Processing

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.

```
Nvsweep = Nsweep/2;
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:

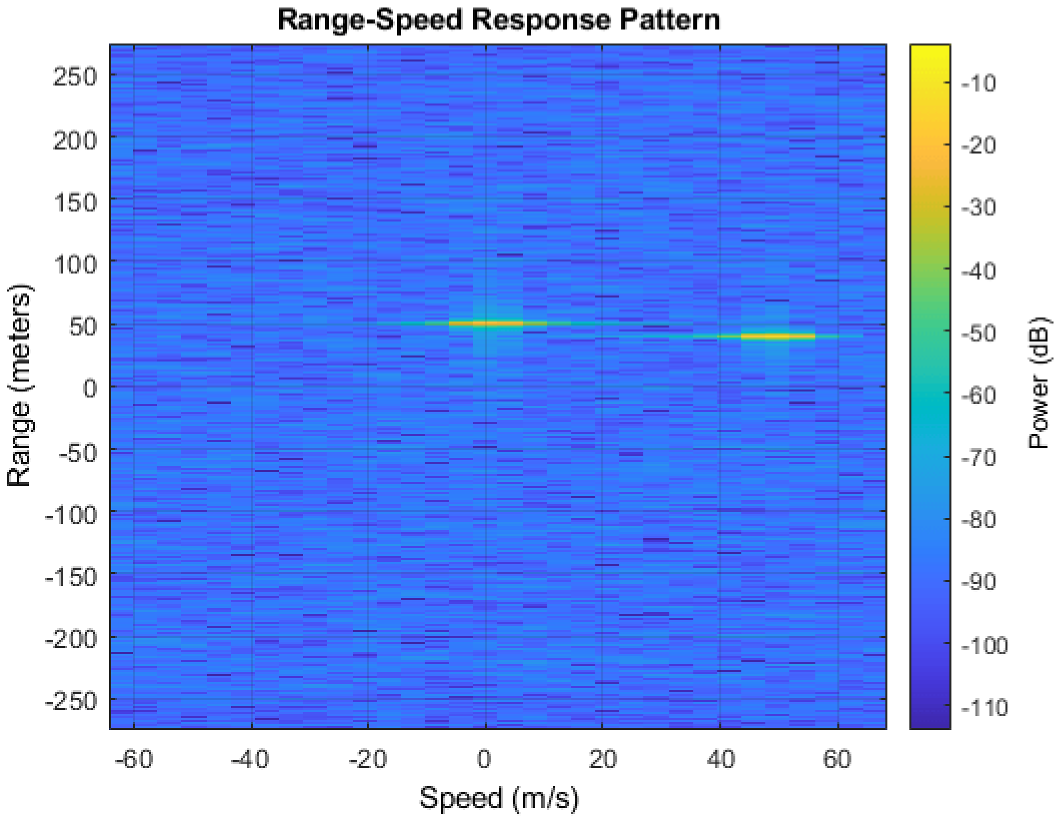
```
xrv = cat(2,xr1,xr2);
```

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,:)));
```



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);
```

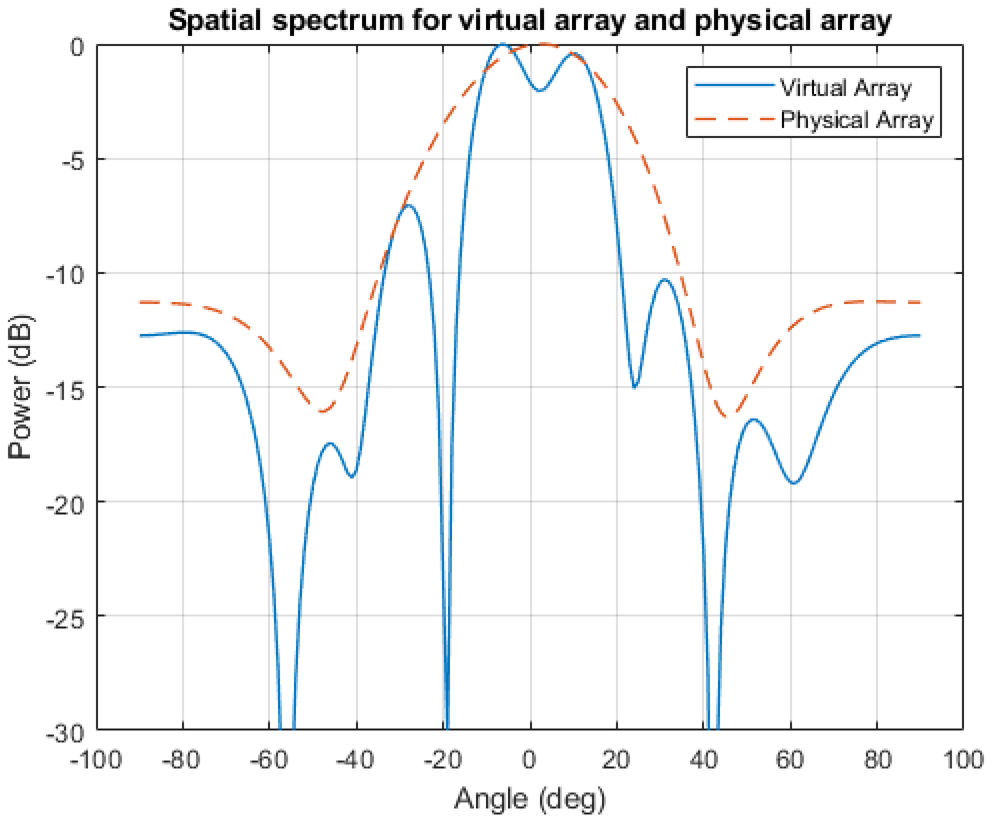
```
fprintf('target_az_est = [%s]\n',num2str(target_az_est));
```

```
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.

Using the same configuration as the example, one 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.

MIMO radars can also adopt phase-coded waveforms in MIMO radar. 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.

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

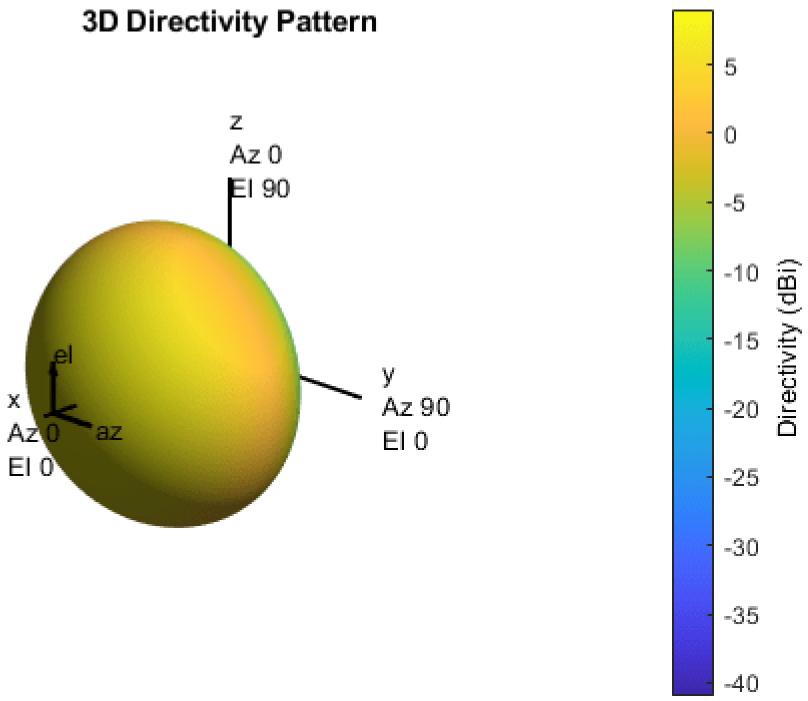
This example shows how to model a 77 GHz 2x4 antenna array for Frequency-Modulated Continuous-Wave (FMCW) radar applications. The presence of antennas and antenna arrays in and around vehicles has become commonplace with the introduction of wireless collision detection, collision avoidance, and lane 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

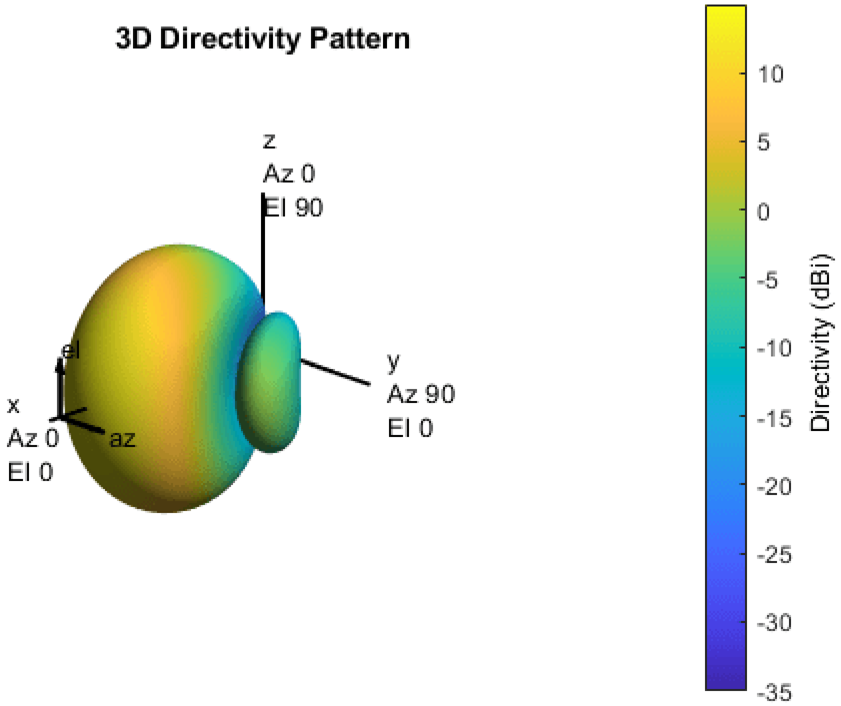
The FMCW antenna array is intended for a forward radar system designed to look for and prevent a collision. Therefore, a cosine antenna pattern is an appropriate choice for the initial design since it does not radiate any energy backwards. Assume that the radar system operates at 77 GHz with a 700 MHz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

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

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



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)
```



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);
```

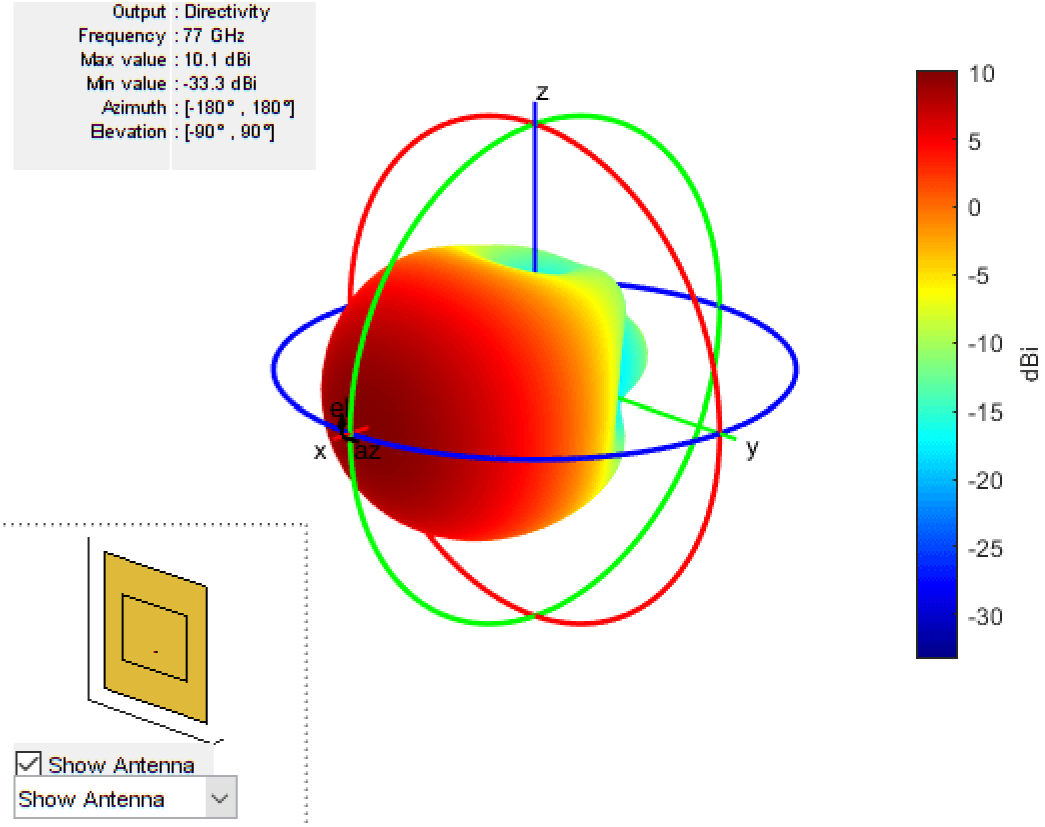
Because the default patch antenna geometry has its maximum radiation directed towards zenith, rotate the patch antenna by 90 degrees about the y-axis so that the maximum would now occur along the x-axis.

```
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 6-9 dBi.

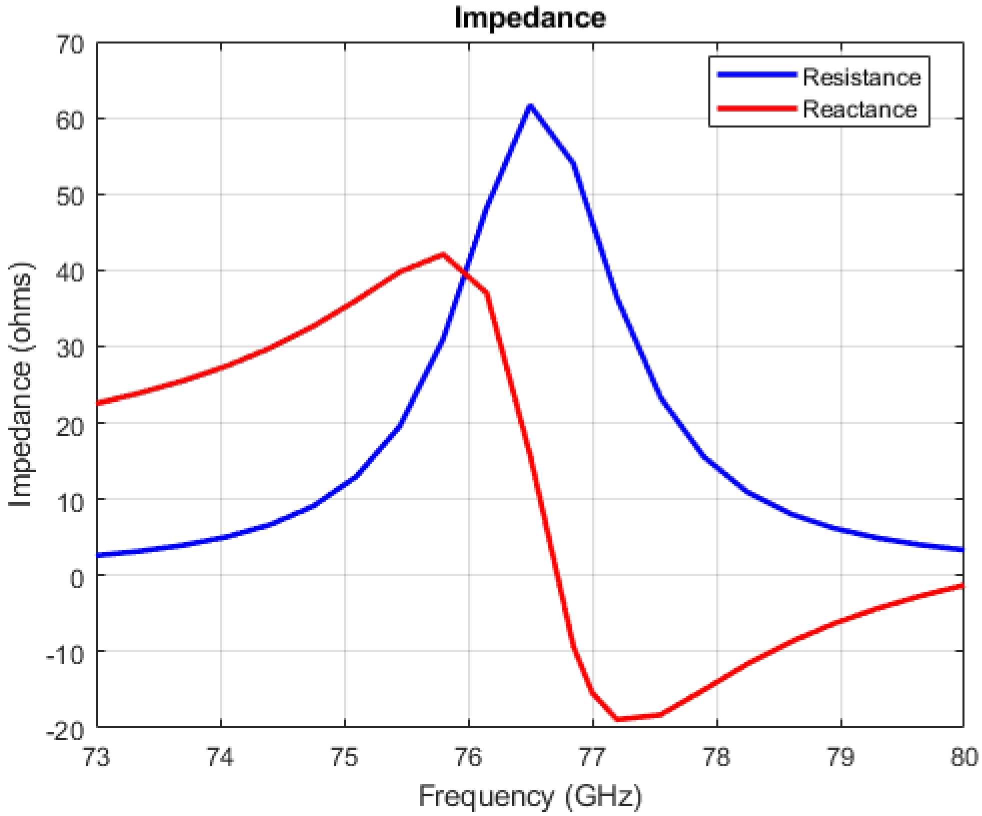
```
myFigure = gcf;
myFigure.Color = 'w';
pattern(patchElement,fc)
```



```
   Output : Directivity
Frequency : 77 GHz
Max value : 10.1 dBi
Min value : -33.3 dBi
  Azimuth : [-180° , 180°]
Elevation : [-90° , 90°]
```

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 antenna's input impedance behavior.

```
Numfreqs = 21;
freqsweep = unique([linspace(fmin,fmax,Numfreqs) fc]);
impedance(patchElement,freqsweep);
```

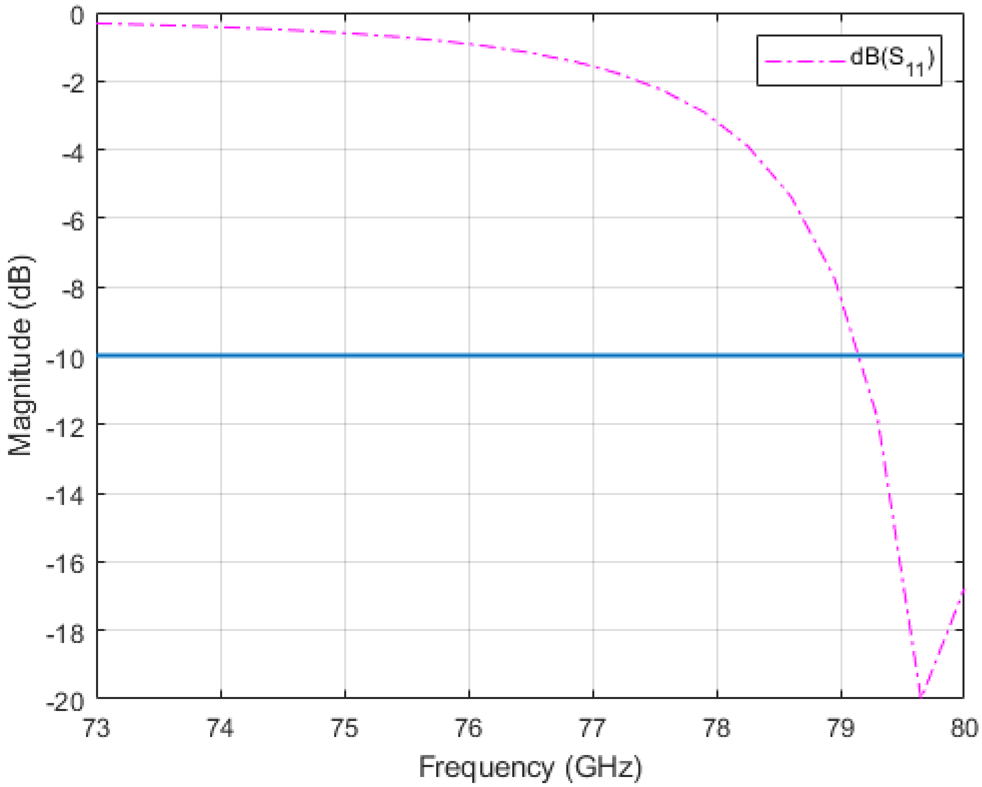


According to the figure, the patch antenna has its first resonance (parallel resonance) at 74 GHz. It is a common practice to shift this resonance to 77 GHz by scaling the length of the patch.

```
act_resonance = 74e9;
lambda_act = vp/act_resonance;
scale = lambda/lambda_act;
patchElement.Length = scale*patchElement.Length;
```

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.

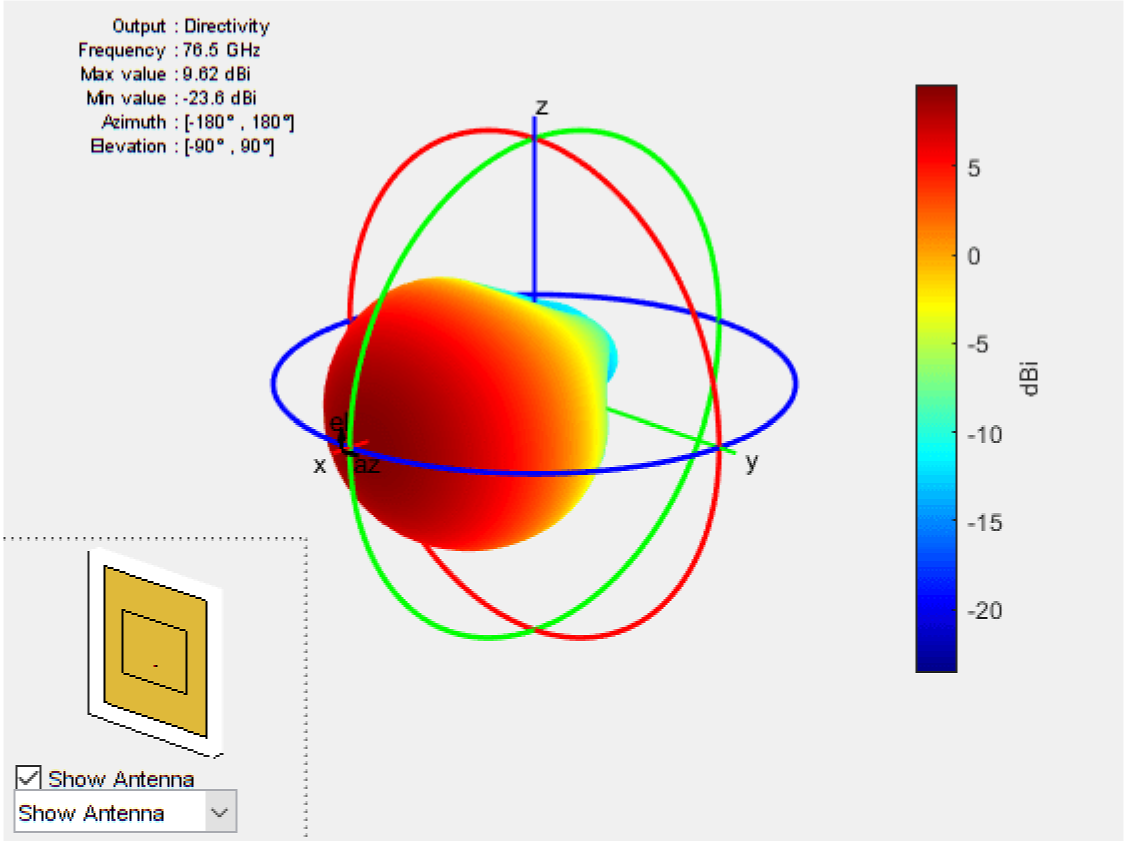
```
s = sparameters(patchElement,freqsweep);
rfplot(s,'m-.')
hold on
line(freqsweep/1e9,ones(1,numel(freqsweep))*-10,'LineWidth',1.5)
hold off
```



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.

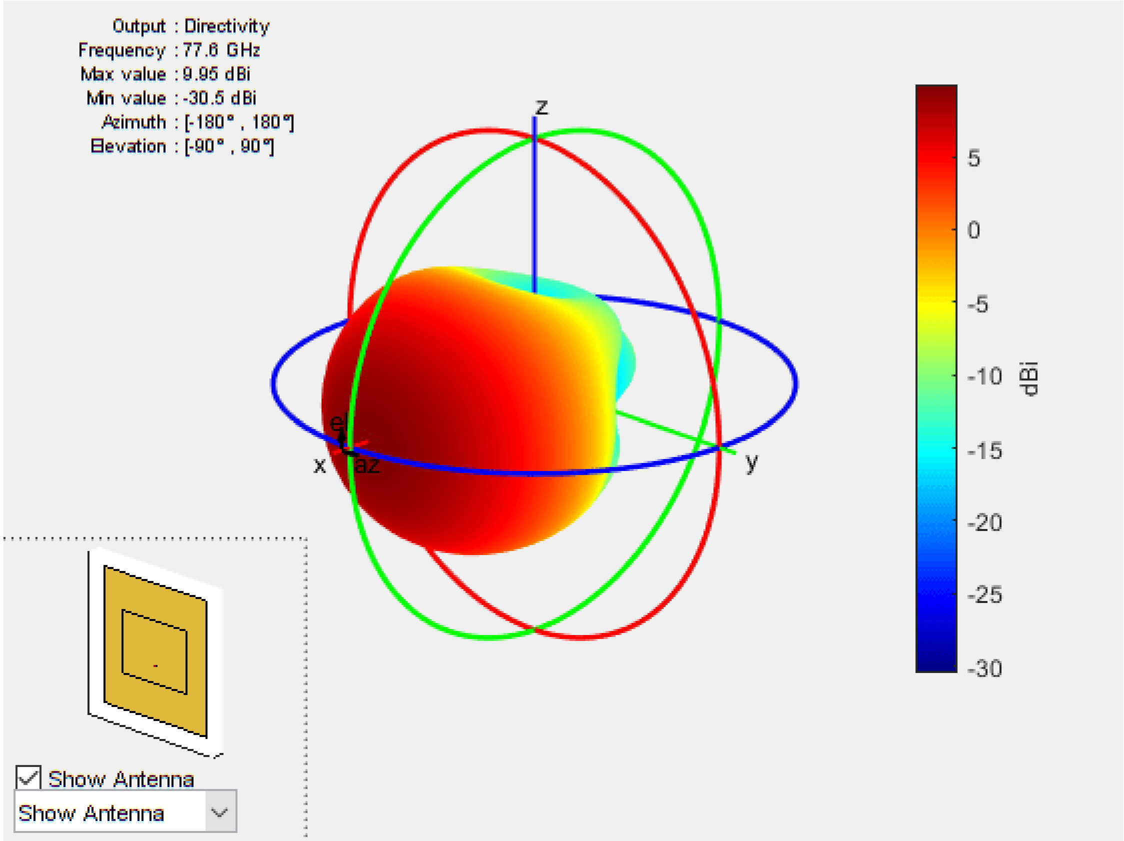
```
pattern(patchElement,76.5e9)
```



Output : Directivity
Frequency : 76.5 GHz
Max value : 9.62 dBi
Min value : -23.6 dBi
Azimuth : [-180° , 180°]
Elevation : [-90° , 90°]

☑ Show Antenna

```
pattern(patchElement,77.6e9)
```



Output : Directivity
Frequency : 77.6 GHz
Max value : 9.95 dBi
Min value : -30.5 dBi
Azimuth : [-180° , 180°]
Elevation : [-90° , 90°]

In general, it is a good practice to check pattern behavior over the frequency band of interest.

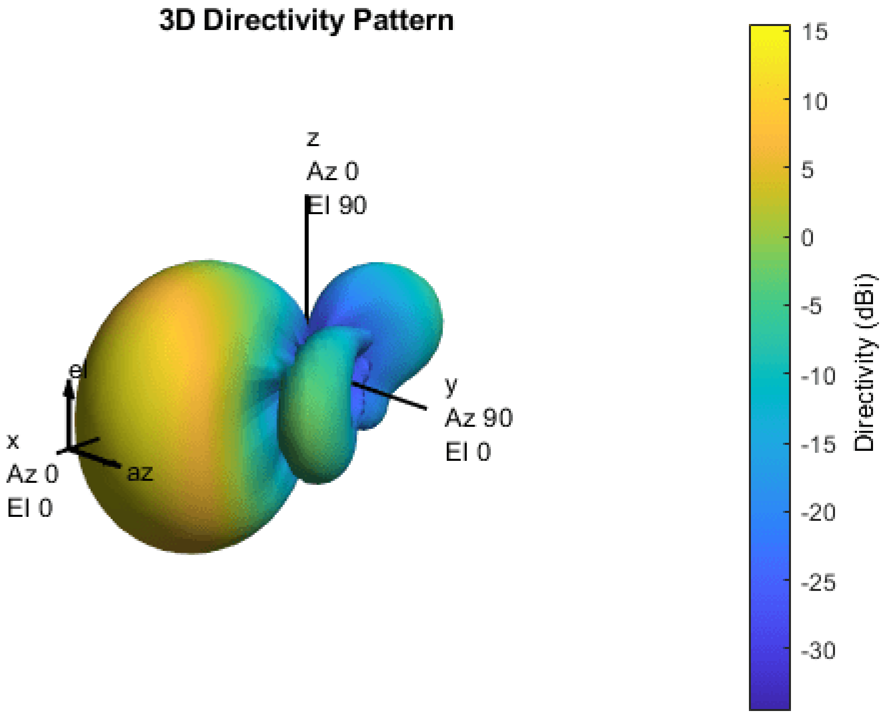
Create Array from Isolated Radiators and Plot Pattern

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.

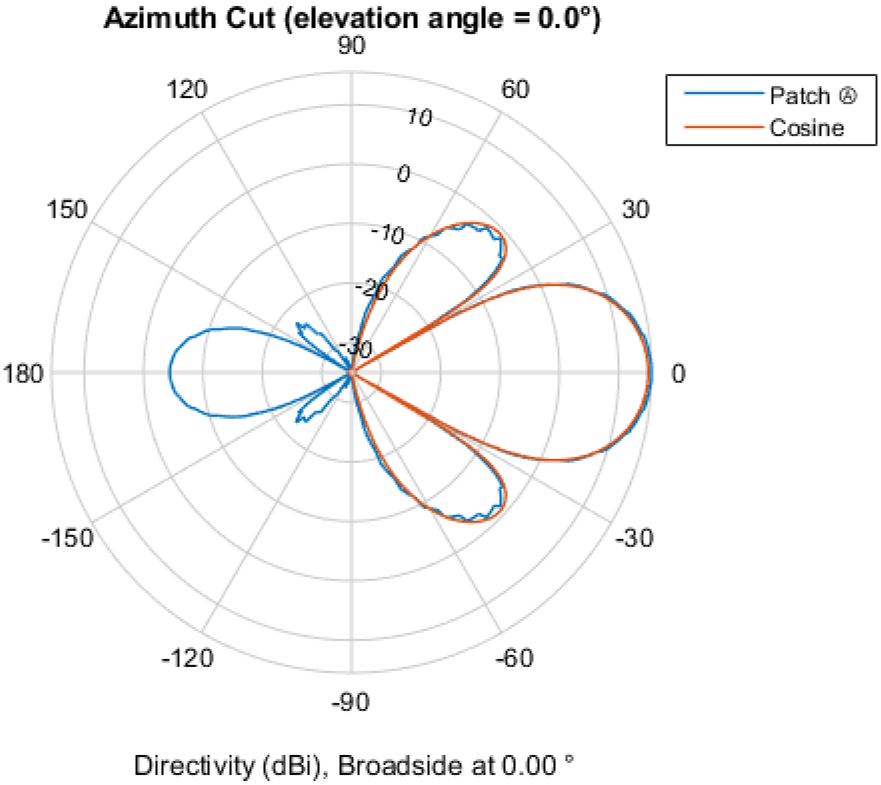
```
az = -180:5:180;
el = -90:5:90;
clf
pattern(fmcwPatchArray,fc,az,el)
```



Plots below compare the pattern variation in 2 orthogonal planes for the patch antenna array and the cosine element array. Note that both arrays ignore mutual coupling effect.

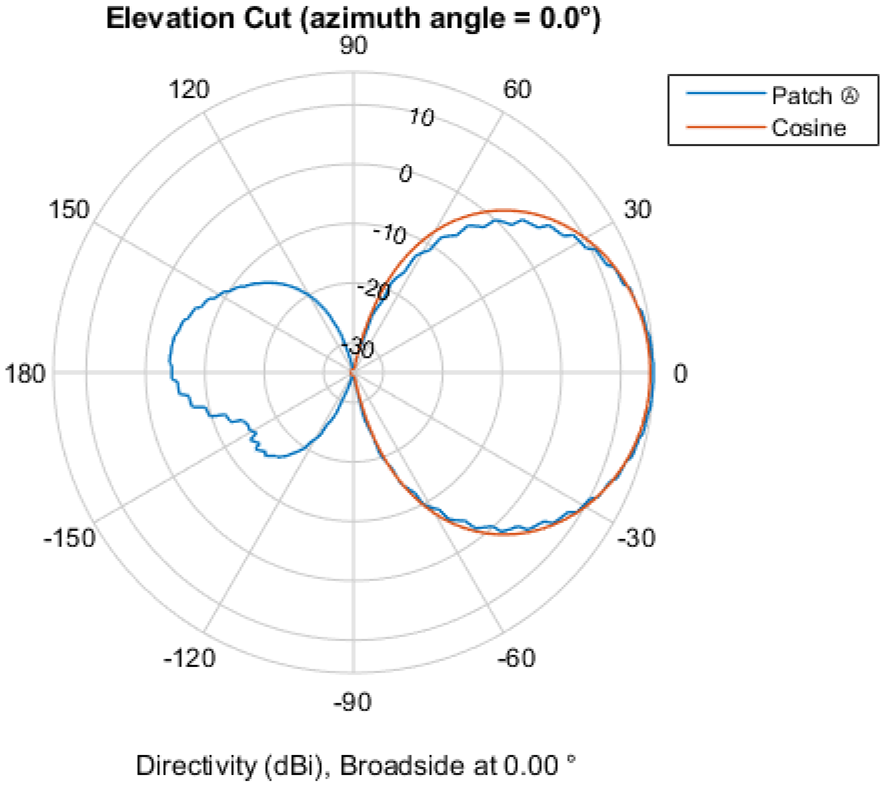
First, plot the patterns along the azimuth direction.

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



Then, plot the patterns along the elevation direction.

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



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 would have on the array pattern.

However, analysis of realistic arrays must also consider mutual coupling effect. Since this is a small array (8 elements in 2x4 configuration), 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`

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 multi-target scenarios because it does not introduce ghost targets.

Triangle Sweep FMCW Waveform

In example “Automotive Adaptive Cruise Control Using FMCW Technology” on page 1-132, 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 solves the range-Doppler coupling issue elegantly for a single target, its processing becomes complicated in multi-target situations. Next section shows how a triangle sweep FMCW waveform behaves when two targets are present.

The scene includes a car 50 meters away from the radar, traveling at 96 km/h along the same direction as the radar, and a truck at 55 meters 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.

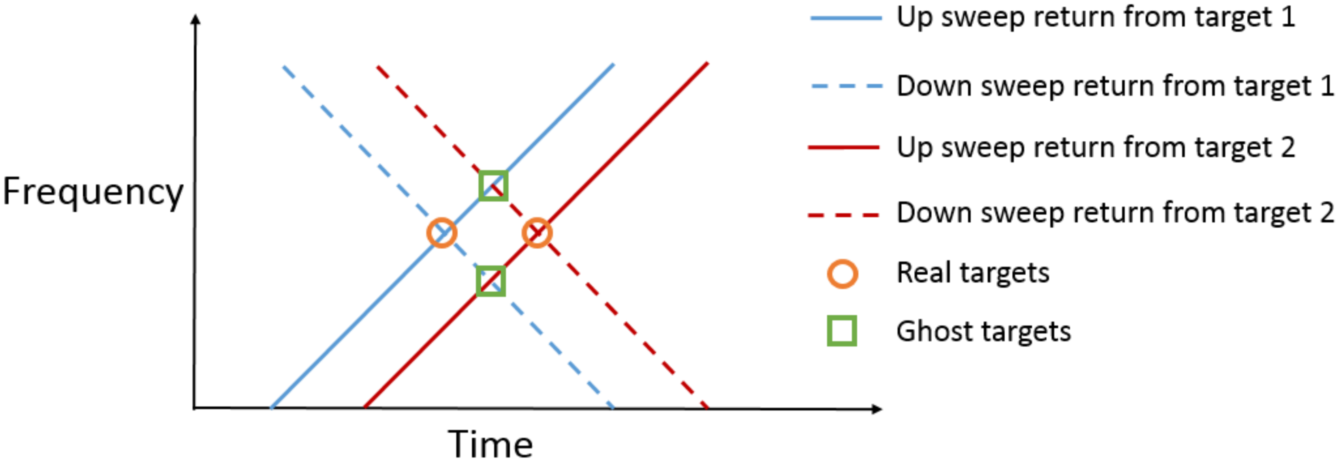
```
dfactor = ceil(fs/maxbeatfreq)/2;
fs_d = fs/dfactor;
fbu_rng = rootmusic(decimate(xr(:,1),dfactor),2,fs_d);
fbd_rng = rootmusic(decimate(xr(:,2),dfactor),2,fs_d);
```

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
```

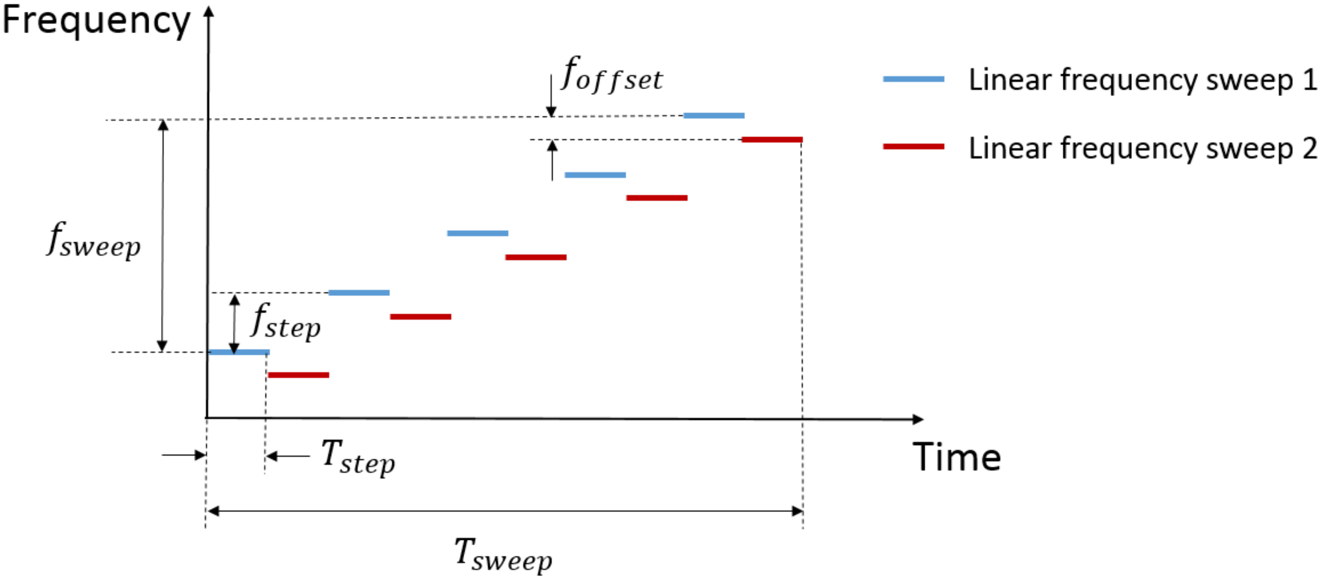
The remaining two are what often referred to as the ghost targets. The relationship between real targets and ghost targets can be better explained using time-frequency representation.


Frequency
Time
Up sweep return from target 1
Down sweep return from target 1
Up sweep return from target 2
Down sweep return from target 2
Real targets
Ghost targets

As shown in the figure, each intersection of a 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 1-0 is designed for automotive radar to achieve simultaneous range and Doppler estimation under 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 the number of targets present in the scene.

The following sections simulates the previous example again, but uses an MFSK waveform instead.

End-to-end Radar System Simulation Using MFSK Waveform

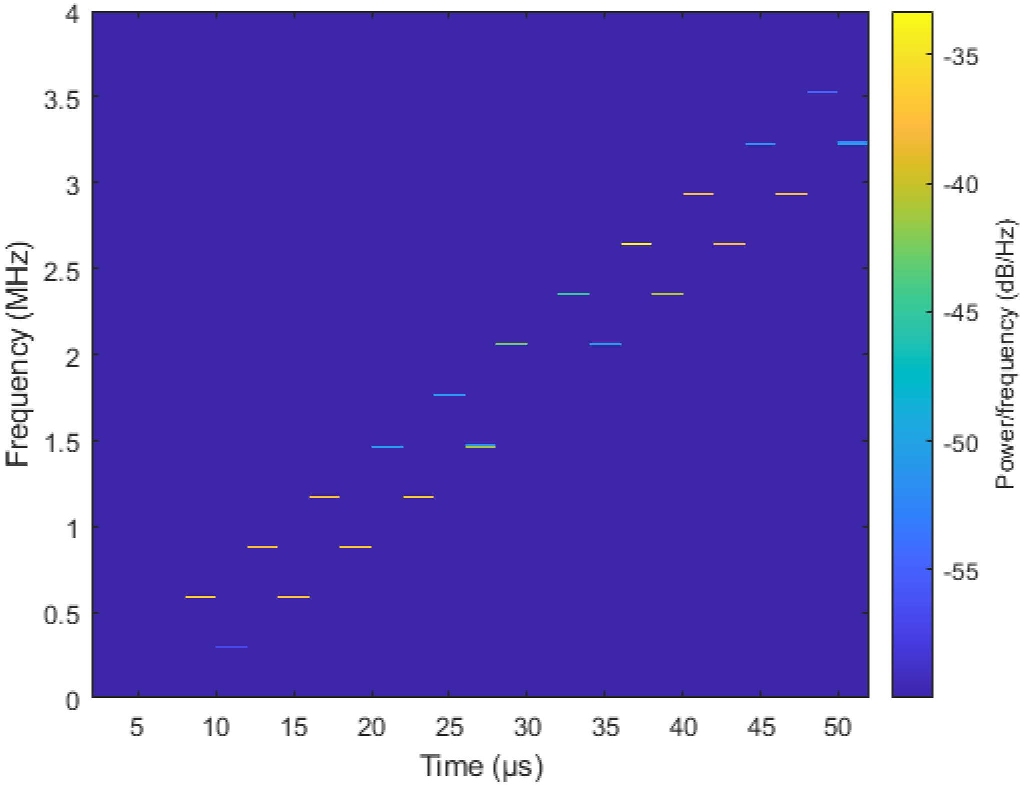
First, parameterize the MFSK waveform to satisfy the system requirement specified in [1] on page 1-0 . Because the range resolution is 1 meter, the sweep bandwidth is set at 150 MHz. In addition, the frequency offset is set at -294 kHz as specified in [1] on page 1-0 . Each step lasts about 2 microseconds 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 1 sweep is needed to estimate the range and Doppler.

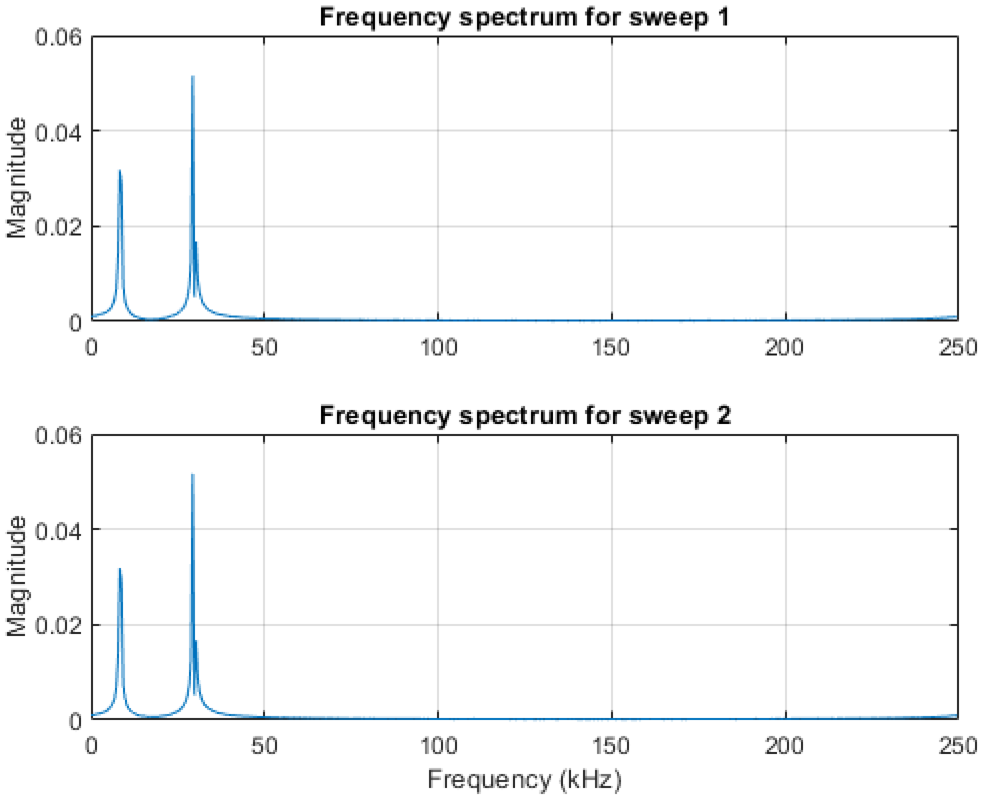
```
Nsweep = 1;
release(channel);
channel.SampleRate = mfskwaveform.SampleRate;
release(receiver);
receiver.SampleRate = mfskwaveform.SampleRate;

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. Next figures shows the frequency spectrums of the received echos 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 identical locations in both returns so there is 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 one constructs the phase difference, the equations are slightly different. For the approach shown in this example, it can be shown that the range and speed satisfies 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 below:

```
sweep_slope = mfskwaveform.SweepBandwidth/...
    (mfskwaveform.StepsPerSweep*mfskwaveform.StepTime);
temp = ...
    [1 sweep_slope;mfskwaveform.StepTime mfskwaveform.FrequencyOffset]\...
    [Fbeat phi/(2*pi)].';

r_est = c*temp(2,:)/2
```

```
r_est = 1×2

   54.8564   49.6452
```

```
v_est = lambda*temp(1,:)/(-2)
```

```
v_est = 1×2

   36.0089   -9.8495
```

The estimated range and speed match the true range and speed values, as tabulated below, 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. It is shown that MFSK waveform have an advantage over FMCW waveform when multiple targets are present since it does not introduce ghost targets during the processing.

References

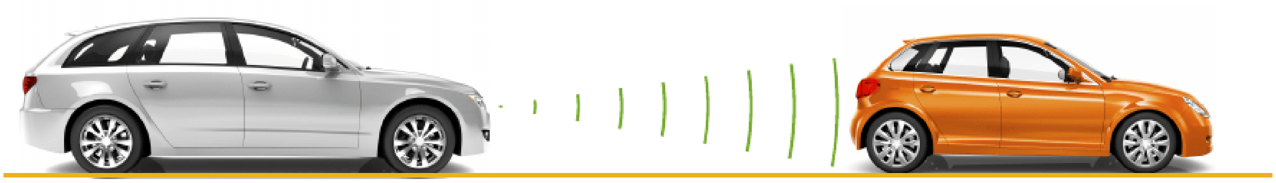
[1] Rohling, H. and M. Meinecke. *Waveform Design Principle for Automotive Radar Systems*, Proceedings of CIE International Conference on Radar, 2001.

Automotive Adaptive Cruise Control Using FMCW Technology

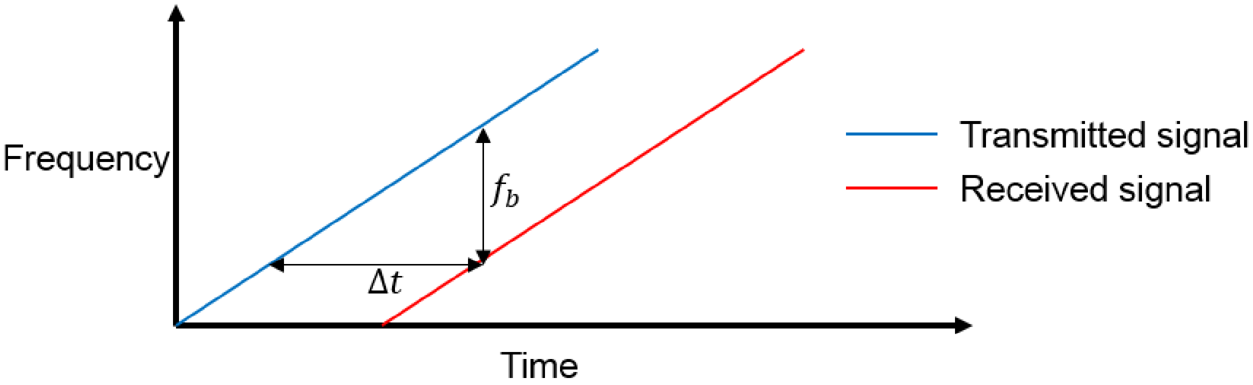
This example shows how to model an automotive adaptive cruise control system using the frequency modulated continuous wave (FMCW) technique. This example performs range and Doppler estimation of a moving vehicle. Unlike pulsed radar systems that are commonly seen in the defense industry, automotive radar 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. The figure below 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, one 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, one 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 5 to 6 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 = range2bw(range_res,c);
sweep_slope = bw/tm;
```

Because an FMCW signal often occupies a huge bandwidth, setting the sample rate blindly to twice the bandwidth often stresses the capability of A/D converter hardware. To address this issue, one can often choose a lower sample rate. Two things can be considered 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 given by

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

In addition, 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
```

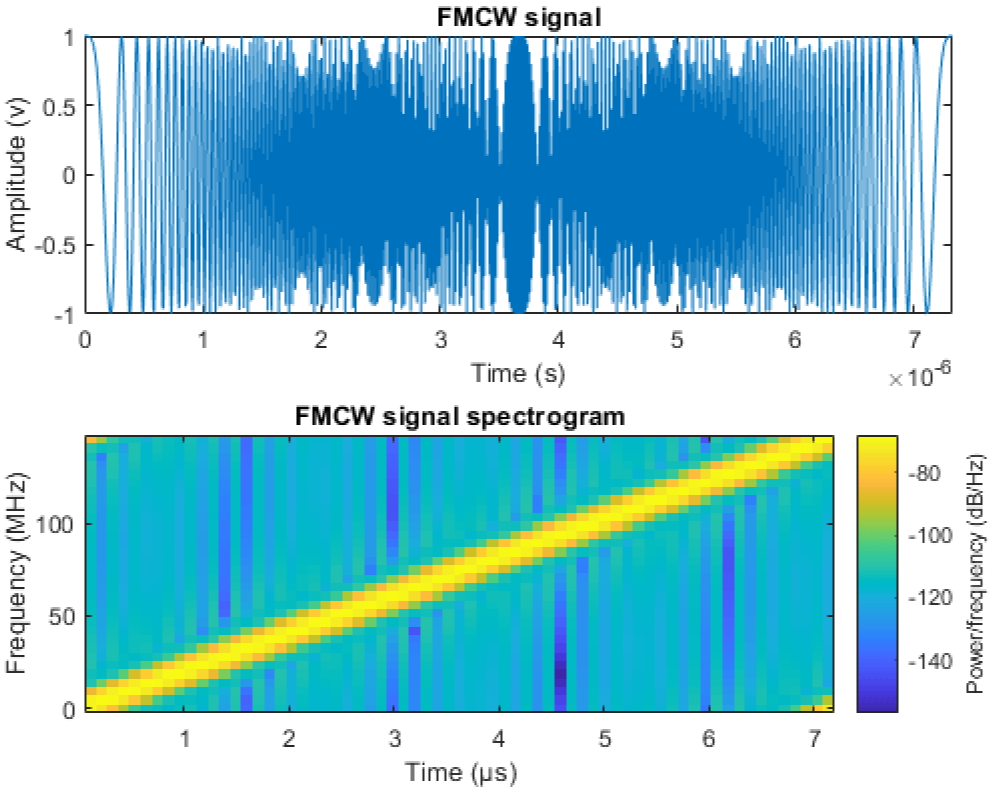
```
Sweep time (microseconds)    7.33
Sweep bandwidth (MHz)        150
Maximum beat frequency (MHz) 27.30
Sample rate (MHz)            150
```

With all the information above, one can set up the FMCW waveform used in the radar system.

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

This is a up-sweep linear FMCW signal, often referred to as sawtooth shape. One can 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 50 m ahead of the car with the radar, at a speed of 96 km/h along the x-axis.

The radar cross section of a car, according to [1], can be computed 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]);
```

The propagation model is assumed 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. One 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.

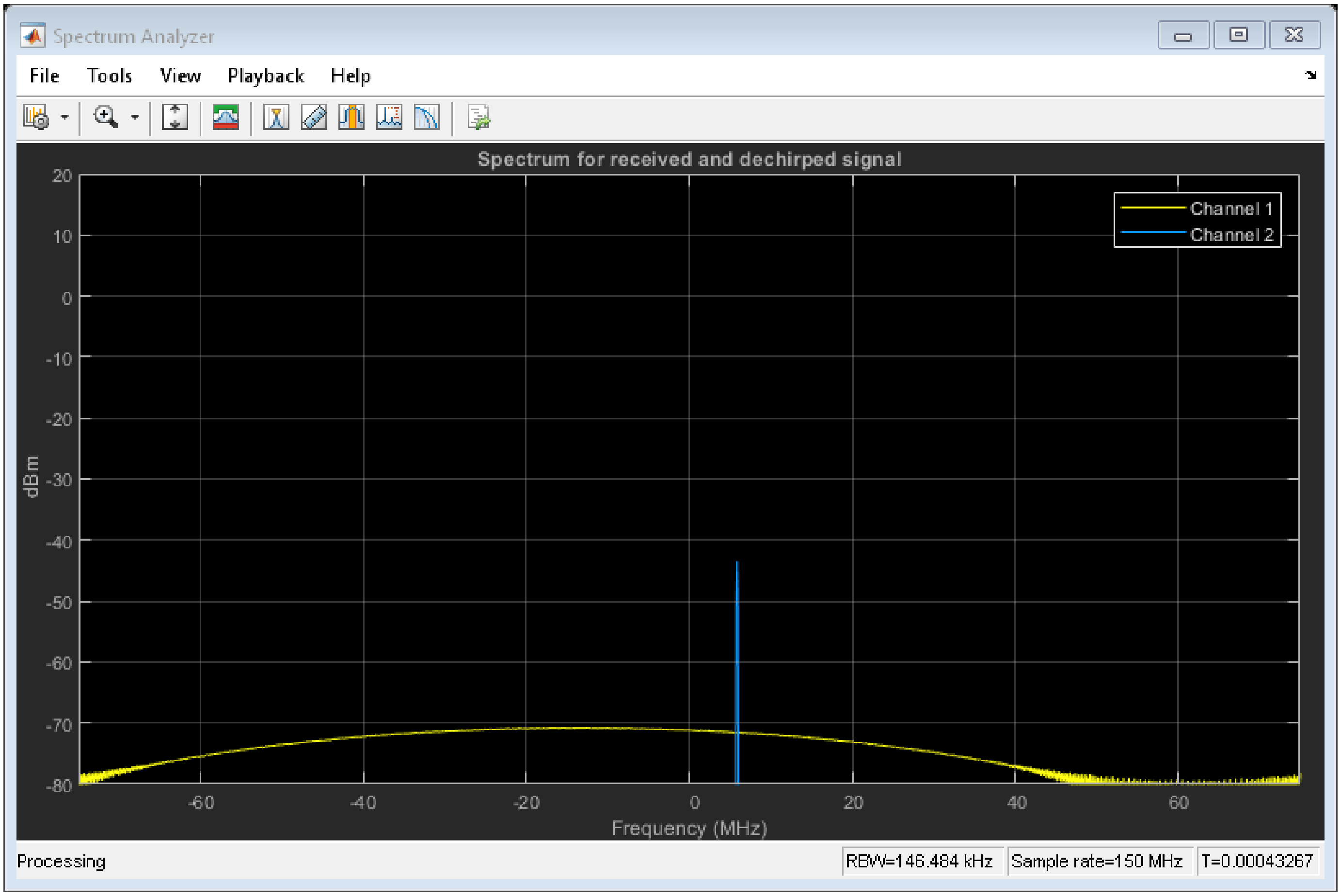
The next section simulates the process outlined above. A total of 64 sweeps are simulated and a range Doppler response is generated at the end.

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

```
specanalyzer = dsp.SpectrumAnalyzer('SampleRate',fs,...
    'PlotAsTwoSidedSpectrum',true,...
    'Title','Spectrum for received and dechirped signal',...
    'ShowLegend',true);
```

Next, run the simulation loop.

```
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
```

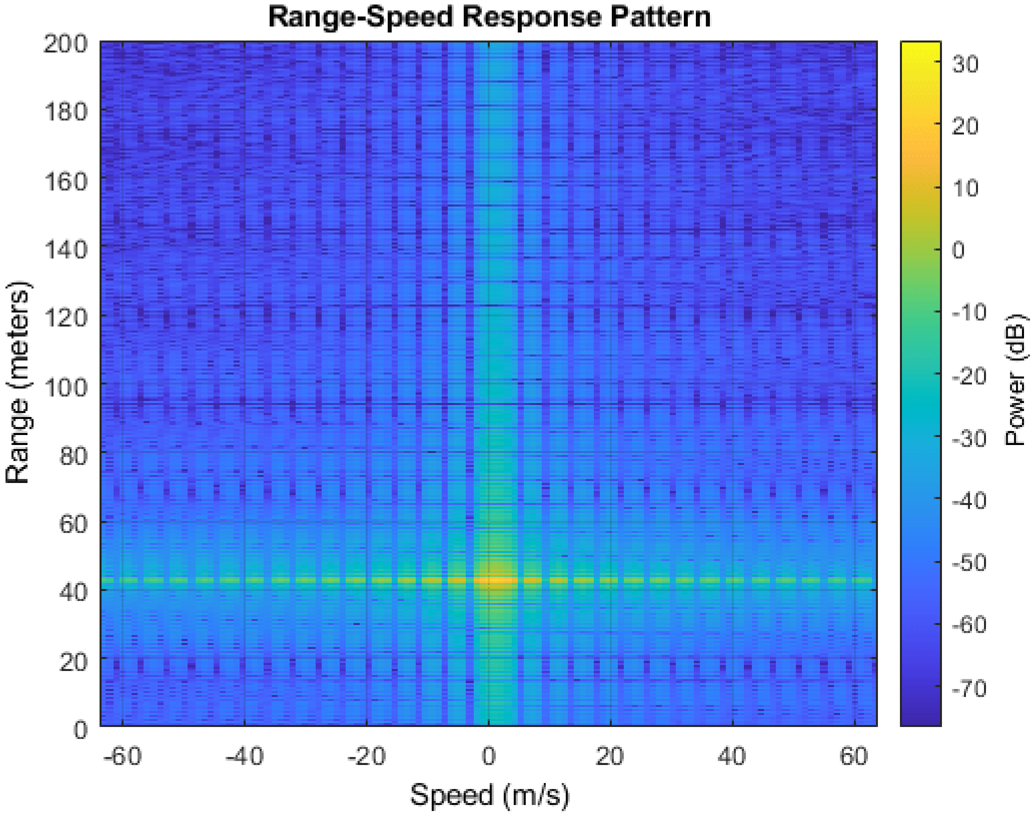


From the spectrum scope, one can see that although the received signal is wideband (channel 1), sweeping through the entire bandwidth, the dechirped signal becomes narrowband (channel 2).

Range and Doppler Estimation

Before estimating the value of the range and Doppler, it may be a good idea to 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;
```



From the range Doppler response, one 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, one 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, one only needs to sample it 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 snippet 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, the beat frequency is estimated 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, the Doppler shift is estimated 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

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

```
deltaR =

   -0.0041
```

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

Even though the current design is achieving 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 may 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, one may think of 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 doesn't work very well with the long sweep time. The longer sweep time results in a lower sampling rate across the sweeps, thus reducing the radar's capability of 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

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

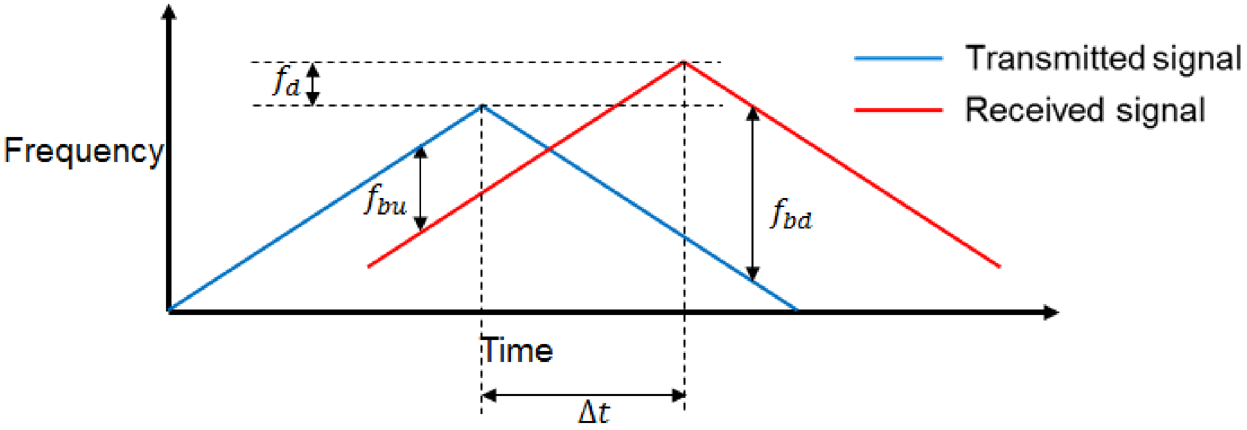
```
v_unambiguous =

    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 are 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, one 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.

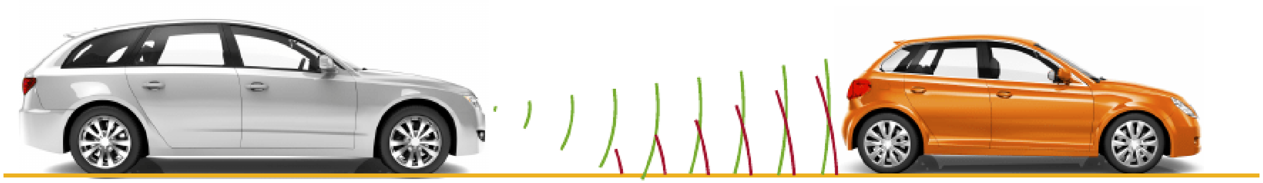
```
fd = -(fbu_rng+fbd_rng)/2;
v_est = dop2speed(fd,lambda)/2
```

```
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

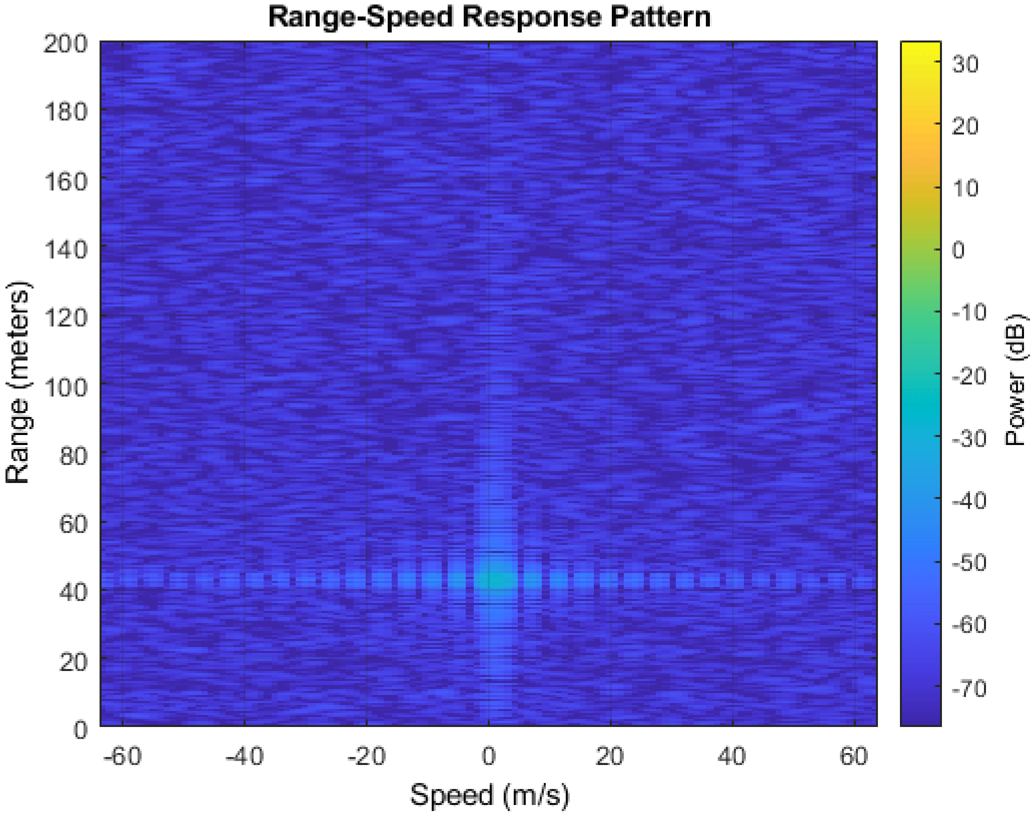
To complete the discussion, in reality, the actual signal propagation between the radar and the target vehicle is more complicated than what is modeled so far. For example, the radio wave can also arrive at the target vehicle via reflections. A simple yet widely used model to describe such a multipath 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 impacts the phase of the signal and the receiving signal at the target vehicle is a coherent combination of the signals via the two paths. Same thing happens on the return trip too where the reflected signal off 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 fluctuating 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.

```
txchannel = phased.TwoRayChannel('PropagationSpeed',c,...
    'OperatingFrequency',fc,'SampleRate',fs);
rxchannel = phased.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);
```



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 discussed 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, C. et al.. *77 GHz ACC Radar Simulation Platform*, IEEE International Conferences on Intelligent Transport Systems Telecommunications (ITST), 2009.

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

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 1-44. 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 1-132. 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
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 = range2bw(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.

```
% 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 1-132.

```
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);
```

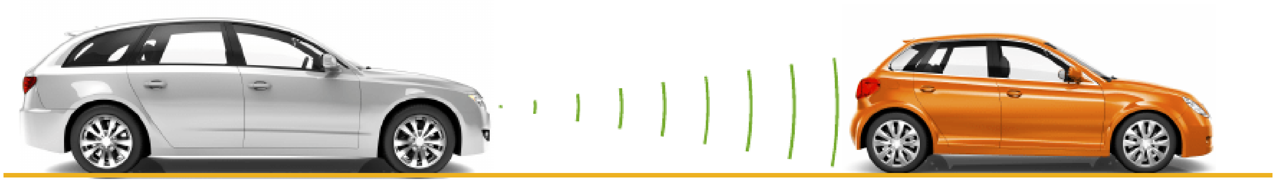
To further improve the precision of the estimated vehicle locations, pass the radar detections to a tracker. Configure the tracker to use an extended Kalman filter (EKF), which converts the spherical radar measurements into the Cartesian coordinate frame of the ego vehicle. Also configure the tracker 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

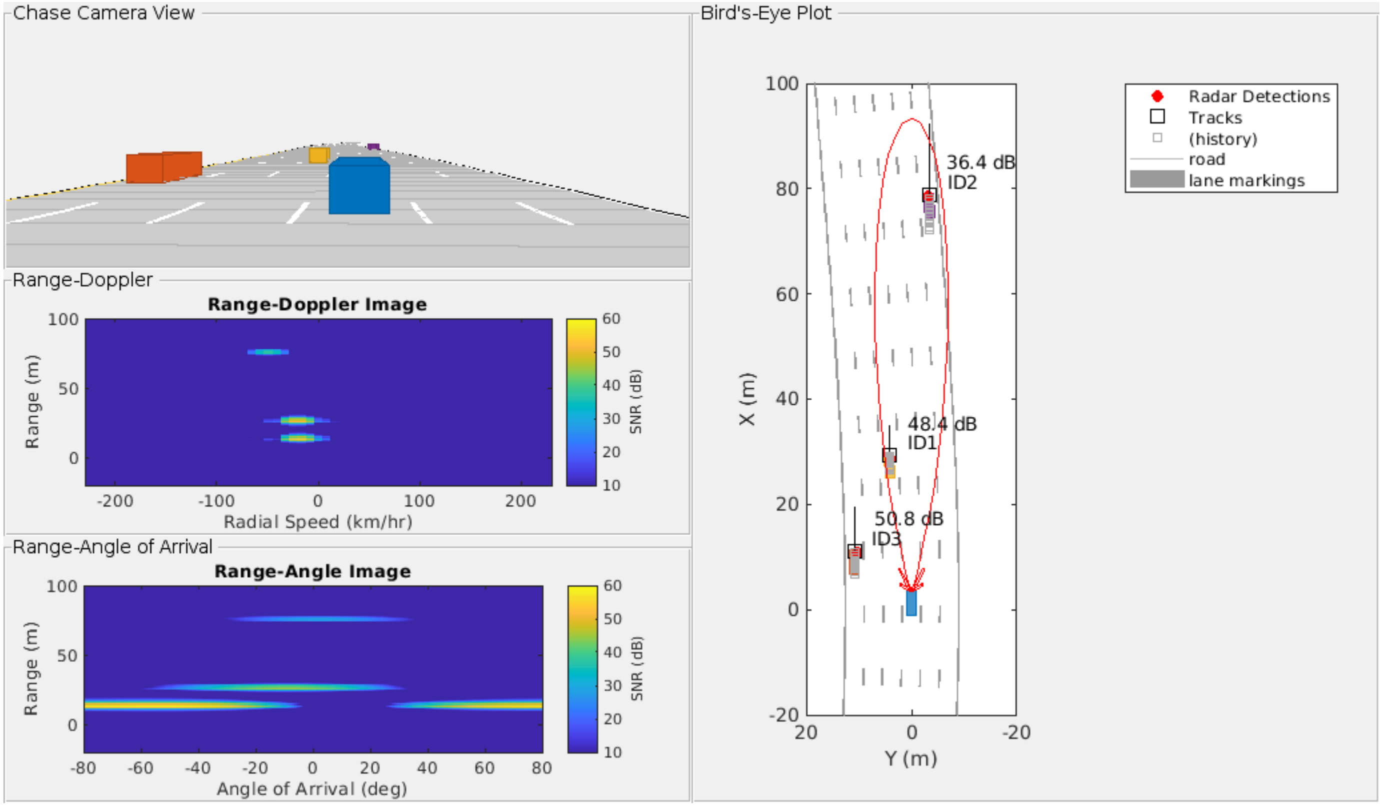
Create a highway driving scenario with three vehicles traveling in the vicinity of the ego vehicle. The vehicles are modeled as cuboids and have different velocities and positions defined in the driving scenario. The ego vehicle is moving with a velocity of 80 km/hr and the other three cars are moving 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
    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).

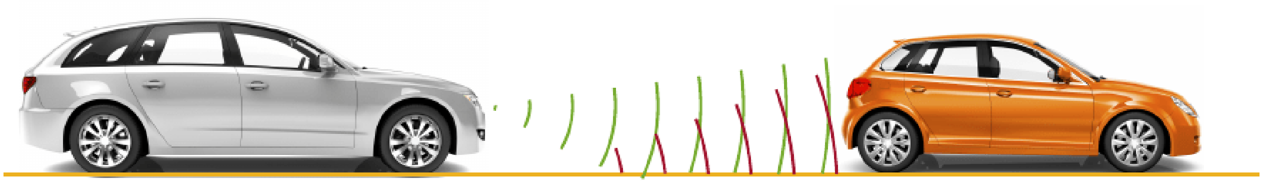
The right side of the figure shows the bird's-eye plot, which presents a top down perspective of the scenario. All of the vehicles, detections, and tracks are shown in the coordinate reference frame of the ego vehicle. The estimated signal-to-noise ratio (SNR) for each radar measurement is printed next 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

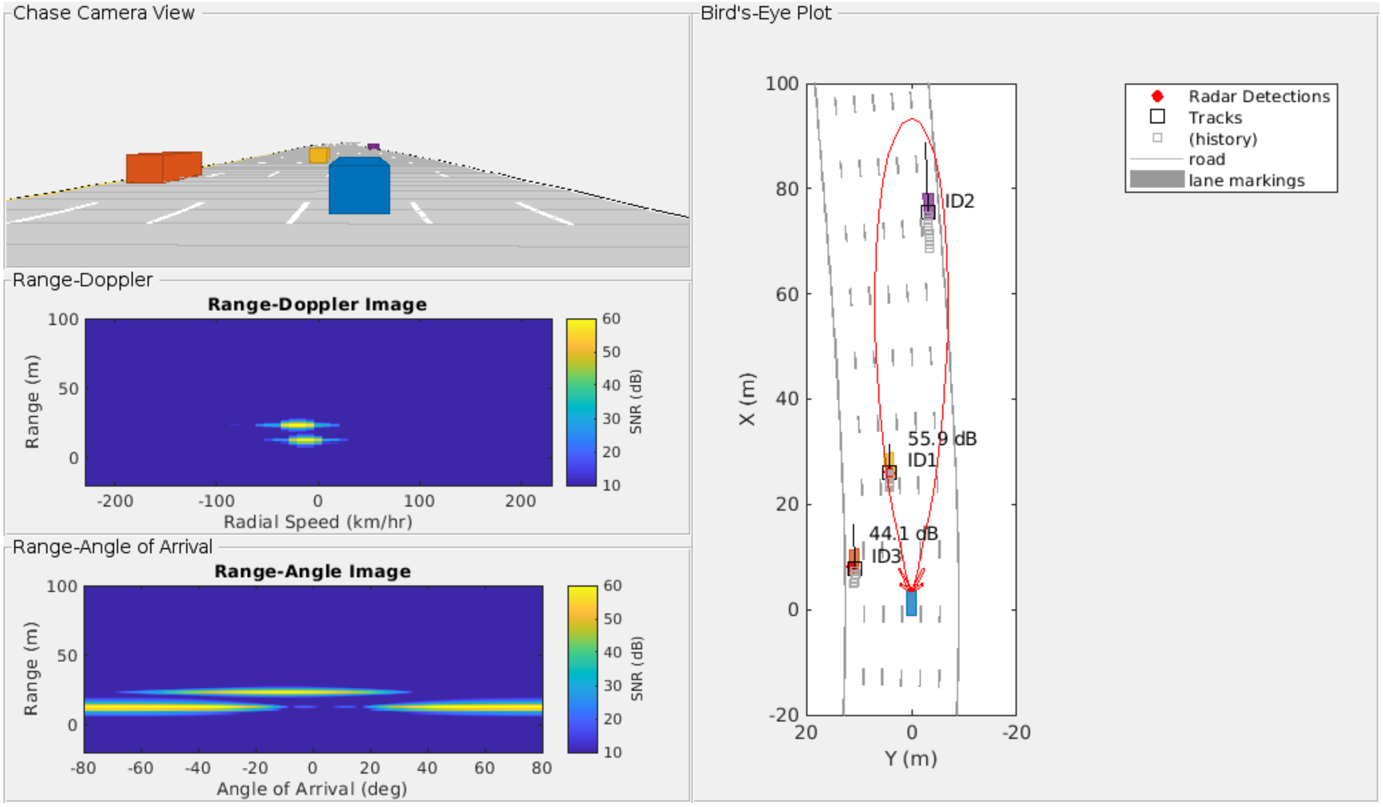
The previous driving scenario simulation used free space propagation. This is a simple model that models only direct line-of-sight propagation between the radar and each of the targets. In reality, the radar signal propagation is much more complex, involving reflections from multiple obstacles before 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.

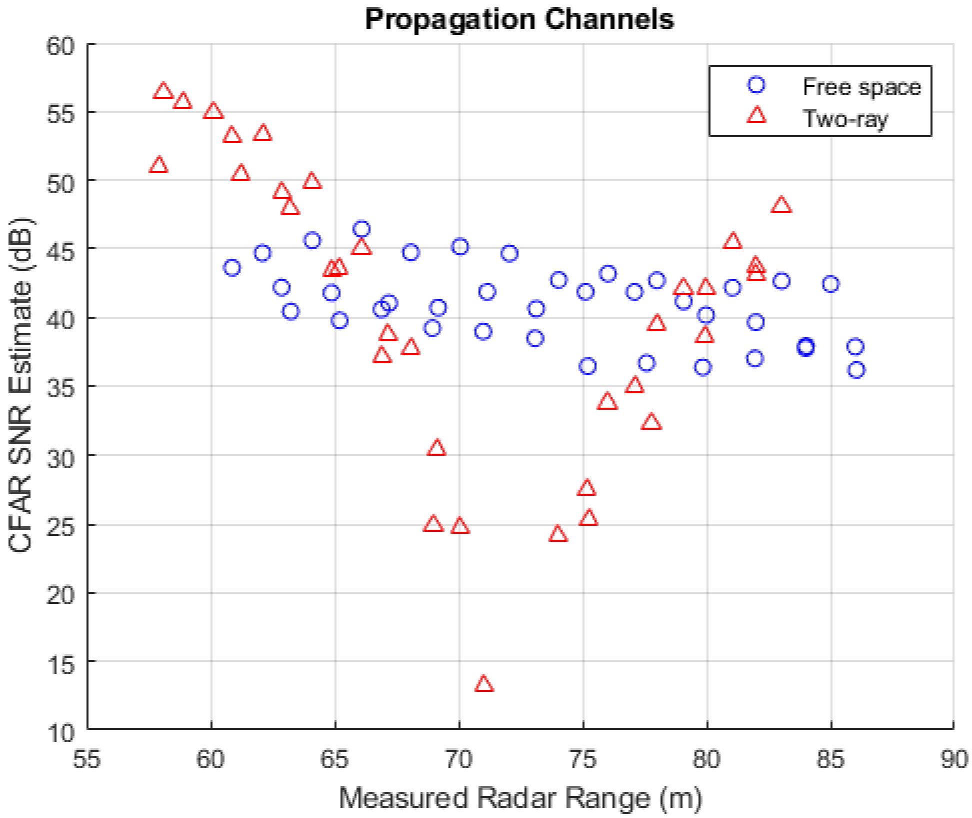
```
% Reset the driving scenario
[scenario,egoCar,radarParams,pointTgts] = ...
    helperAutoDrivingRadarSigProc('Setup Scenario',c,fc);

% 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);
```



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 1-172 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 1-172 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.

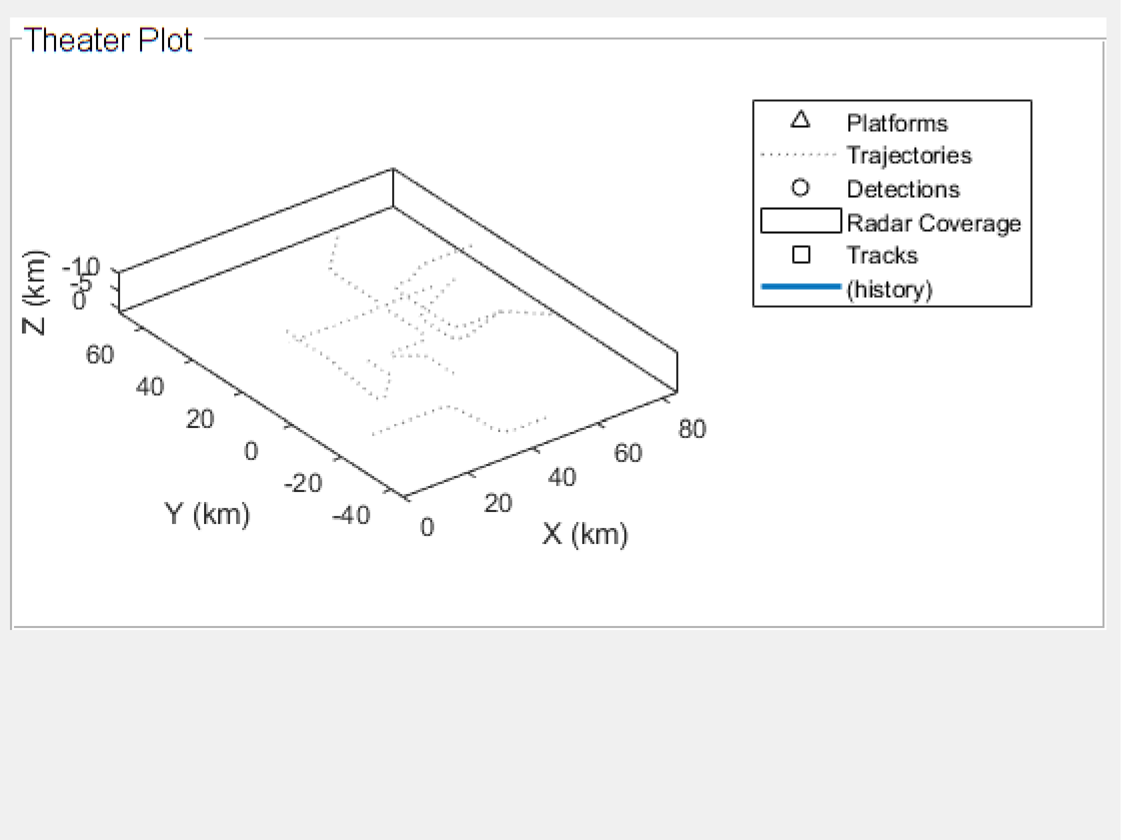
Active tracking treats all the tracks in the same way, which makes it a *mode*-based resource management algorithm. A more sophisticated way to manage the radar is based on the properties of each track. For example, use track properties such as the size of state uncertainty covariance, whether the track 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.

```
% 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], ...
    'FieldOfView', [3;10], ...
    'AzimuthResolution', 1.5, ...
    '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.

```
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 1-155 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 `FieldOfView` property. Negative elevation angles mean that the radar points the beam from the horizon up.

```
AzimuthLimits = [-90 60];
ElevationLimits = [-9.9 0];
azscanspan   = diff(AzimuthLimits);
numazscan    = floor(azscanspan/radar.FieldOfView(1))+1;
azscanangles = linspace(AzimuthLimits(1),AzimuthLimits(2),numazscan)+radar.MountingAngles(1);
elscanspan   = diff(ElevationLimits);
numelscan    = floor(elscanspan/radar.FieldOfView(2))+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 1-155 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.

```
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.

```
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.

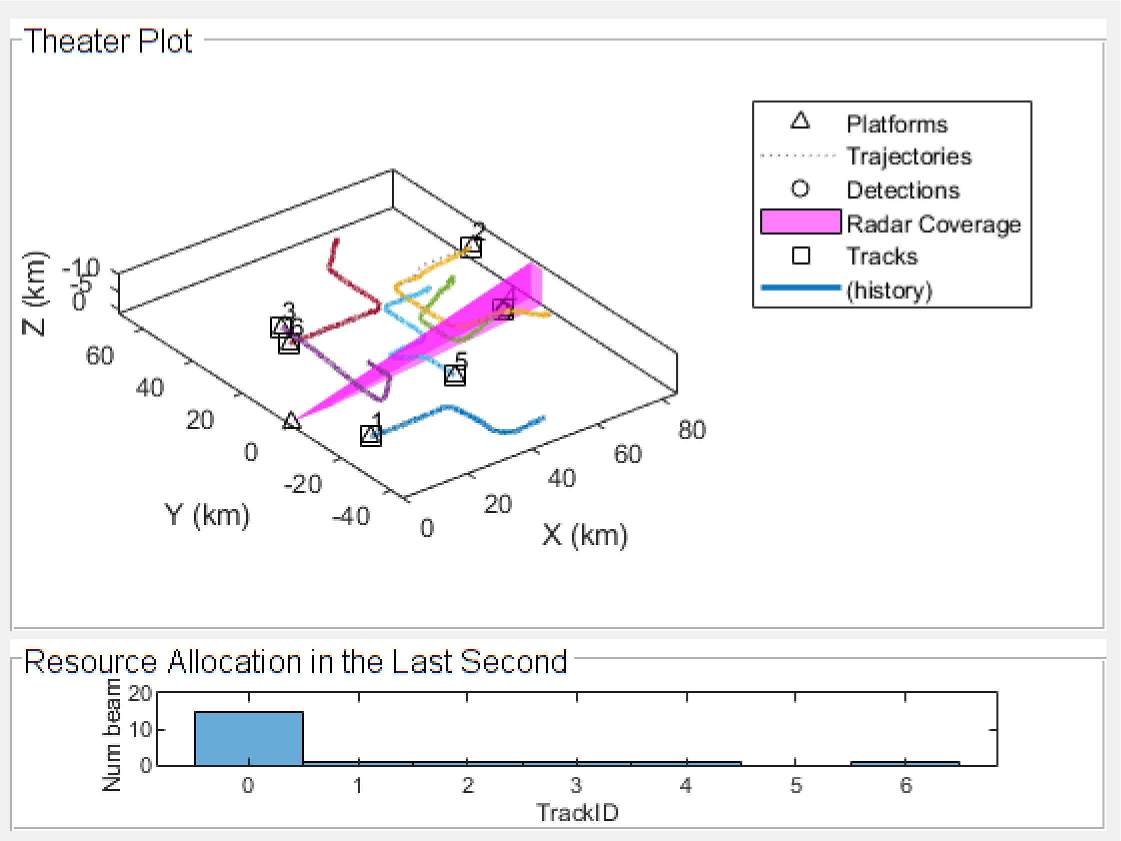
The Analyze Results section shows the results of other options.

```
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.

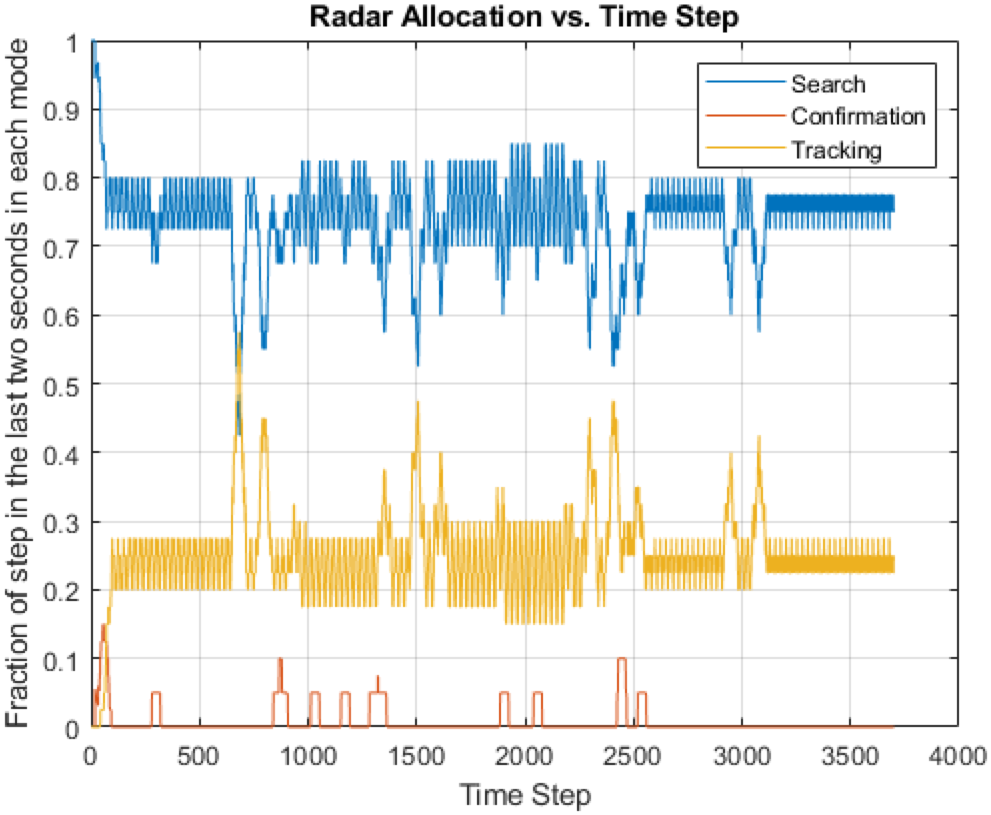
```
% Create a radar resource allocation display
rp = uipanel('Parent',f,'Title','Resource Allocation in the Last Second','FontSize',12,...
    'BackgroundColor','white','Position',[.01 0.01 0.98 0.23]);
rax = axes(rp);

% Run the scenario
allocationType = helperAdaptiveTrackingSim(scenario, thp, rax, tracker, resetJobQ, managerPrefere
```



Analyze Results

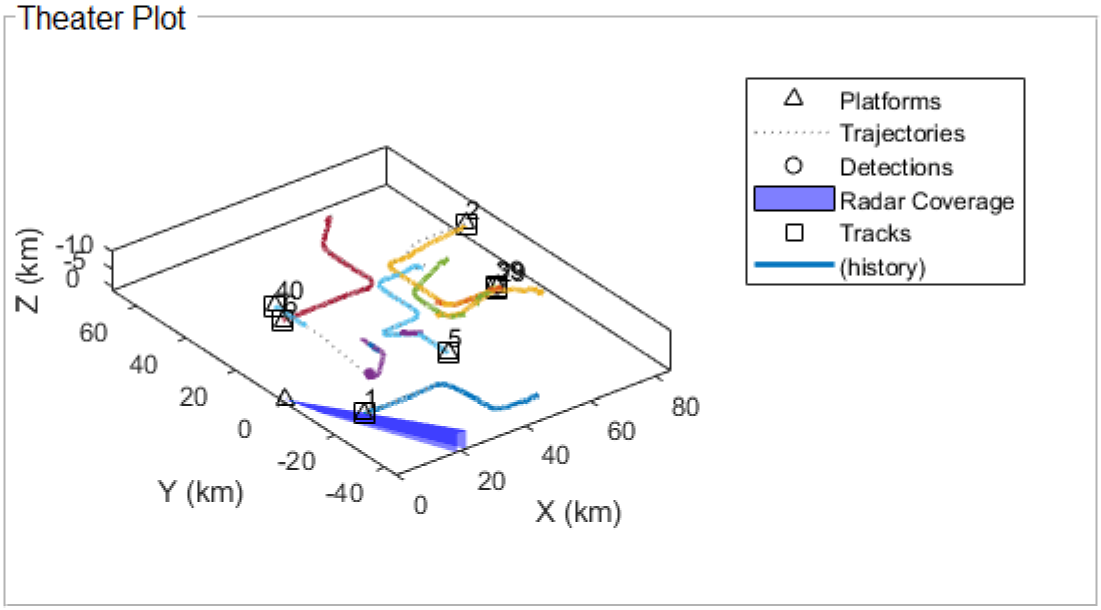
Analyze the radar task load and its division between search, confirmation, and tracking jobs. The graph shows that most of the time the radar allocates about 75% search jobs and 25% tracking jobs, which is as expected when the targets are not maneuvering. When the targets are maneuvering, the resource 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.

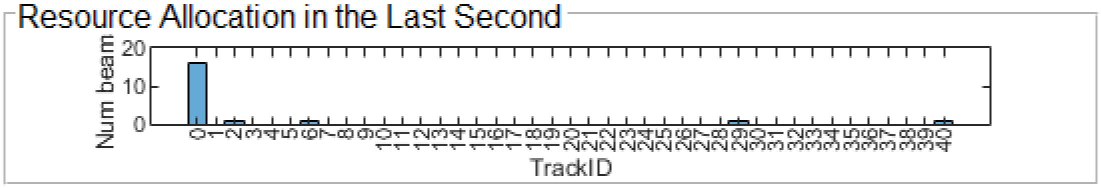
```
numSteps = numel(allocationType);
allocationSummary = zeros(3,numSteps);
for i = 1:numSteps
    for jobType = 1:3
        allocationSummary(jobType,i) = sum(allocationType(1,max(1,i-2*updateRate+1):i)==jobType),
    end
end
figure;
plot(1:numSteps,allocationSummary(:,1:numSteps))
title('Radar Allocation vs. Time Step');
xlabel('Time Step');
ylabel('Fraction of step in the last two seconds in each mode');
legend('Search','Confirmation','Tracking');
grid on
```

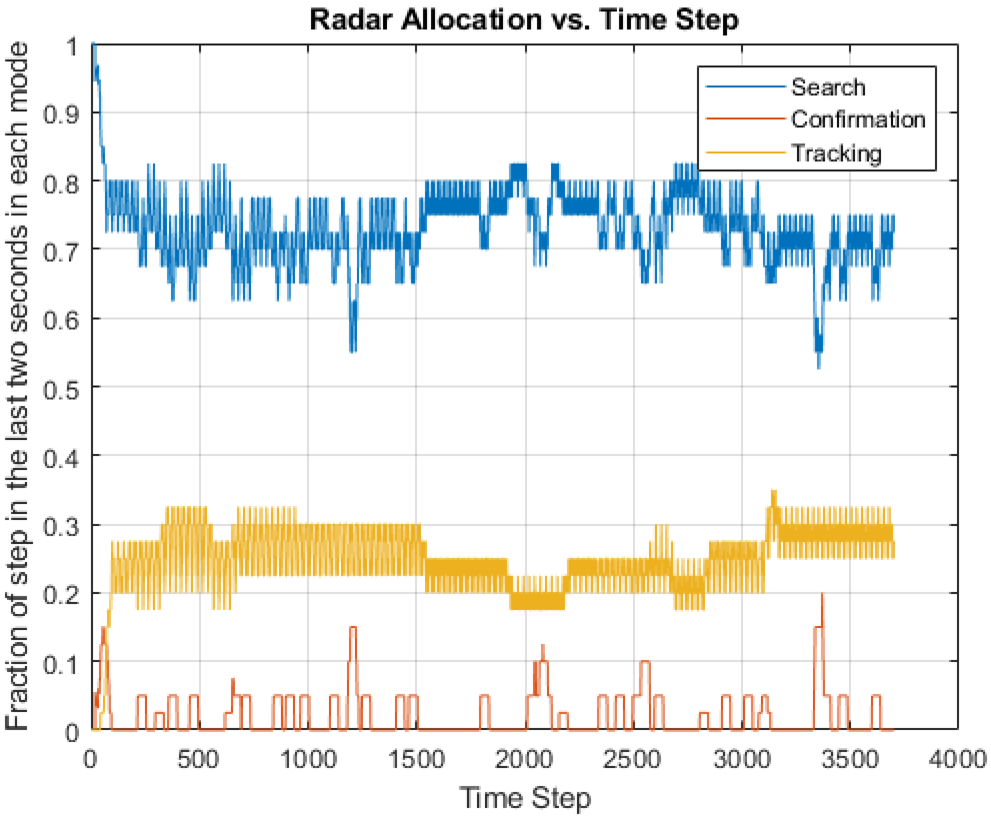


Compare this result with the result of Active Tracking at 0.8 Hz track revisit rate. The following figures show the tracking results and radar allocation graph for the Active Tracking case. The tracking results show that some tracks were lost and broken, but the radar resource allocation graph shows 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.

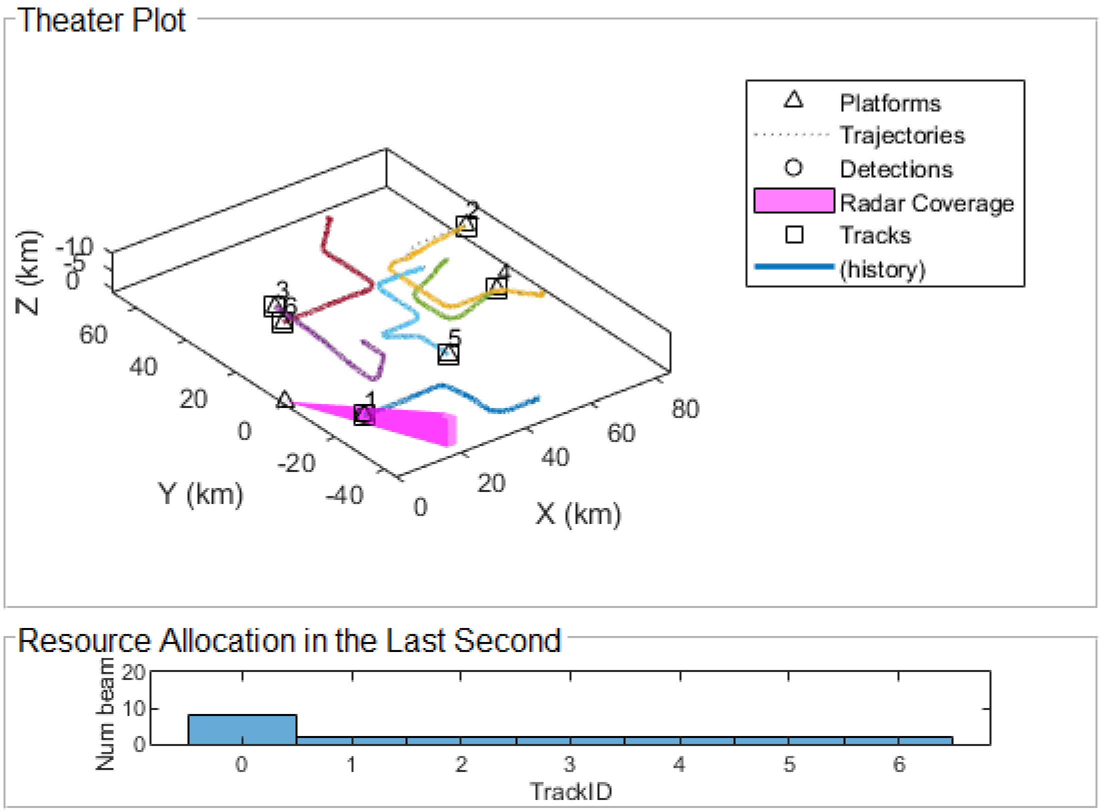
```
clearData(plp);
clearData(dtp);
clearData(trp);
reset(tracker);
restart(scenario);

% Modify manager preferences to 1 Hz revisit rate for both non-maneuvering and maneuvering targe
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
```

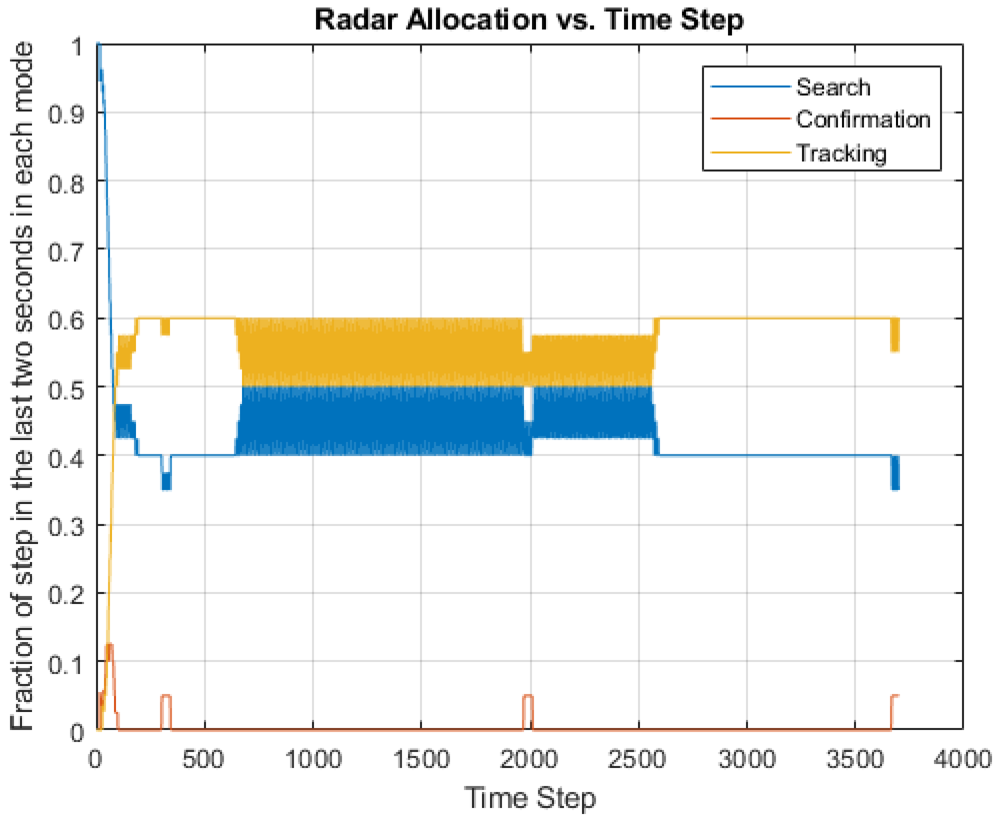

Theater Plot
Platforms
Trajectories
Detections
Radar Coverage
Tracks
(history)
Z (km)
-10
-5
0
60
40
20
0
-20
-40
Y (km)
0
20
40
60
80
X (km)
2
29
40
6
5
1


Resource Allocation in the Last Second
Num beam
20
10
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
TrackID

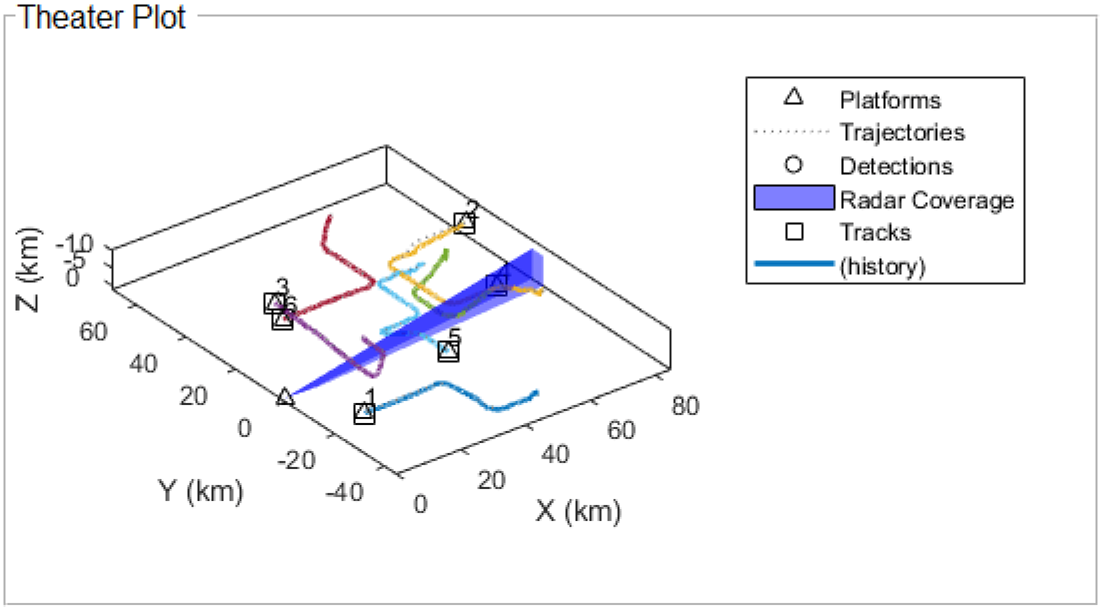

Radar Allocation vs. Time Step
Search
Confirmation
Tracking
Fraction of step in the last two seconds in each mode
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500
3000
3500
4000
Time Step

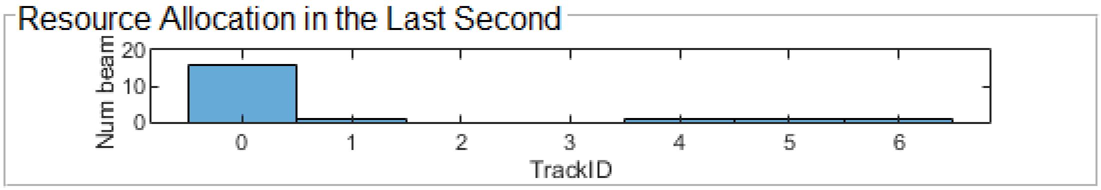
Clearly, a higher tracking revisit rate is needed for Active Tracking. The following two graphs show that increasing the track revisit rate to 2 Hz improves the tracking of maneuvering targets. However, the cost is that the radar dedicates more than 50% of its time to tracking tasks even when the tracks are not maneuvering. If the number of targets were greater, the radar would become overwhelmed.

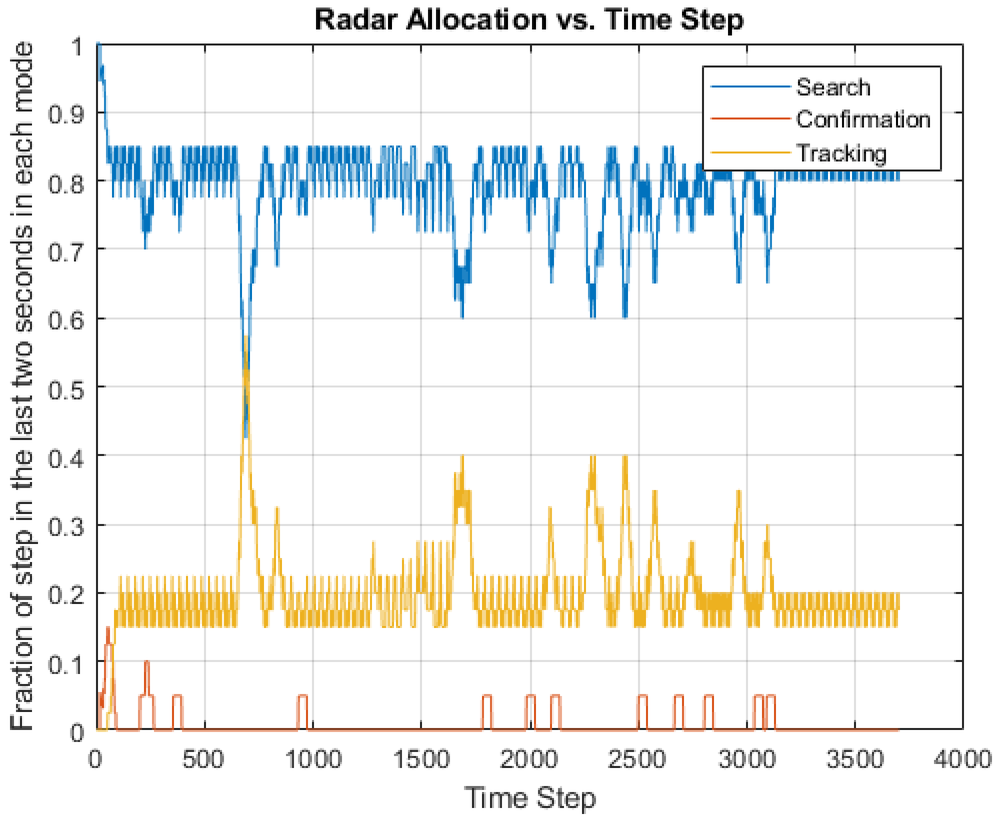

Theater Plot
△ Platforms
Trajectories
○ Detections
Radar Coverage
□ Tracks
(history)
Z (km)
-10
-5
0
60
40
20
0
-20
-40
Y (km)
0
20
40
60
80
X (km)
1
2
3
4
5
6
Resource Allocation in the Last Second
Num beam
20
10
0
0
1
2
3
4
5
6
TrackID


Radar Allocation vs. Time Step
Search
Confirmation
Tracking
Fraction of step in the last two seconds in each mode
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500
3000
3500
4000
Time Step

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.


Theater Plot
Platforms
Trajectories
Detections
Radar Coverage
Tracks
(history)
Z (km)
-10
-5
0
60
40
20
0
-20
-40
Y (km)
0
20
40
60
80
X (km)
1
2
3
4
5
6


Resource Allocation in the Last Second
Num beam
20
10
0
0
1
2
3
4
5
6
TrackID


Radar Allocation vs. Time Step
Fraction of step in the last two seconds in each mode
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
500
1000
1500
2000
2500
3000
3500
4000
Time Step
Search
Confirmation
Tracking

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);

% 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 = [0 0 1];
    else
        detectableTracks = currentJob.TrackID;
        resourceAllocation(min([currentStep,updateRate])) = currentJob.TrackID;
        cvp.Color = [1 0 1];
        if strcmpi(currentJob.JobType,'Confirm')
            allocationType(currentStep) = 2;
        else
```

```
            allocationType(currentStep) = 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, posit
    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 = bw2range(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);
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.

```
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.

```
% 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

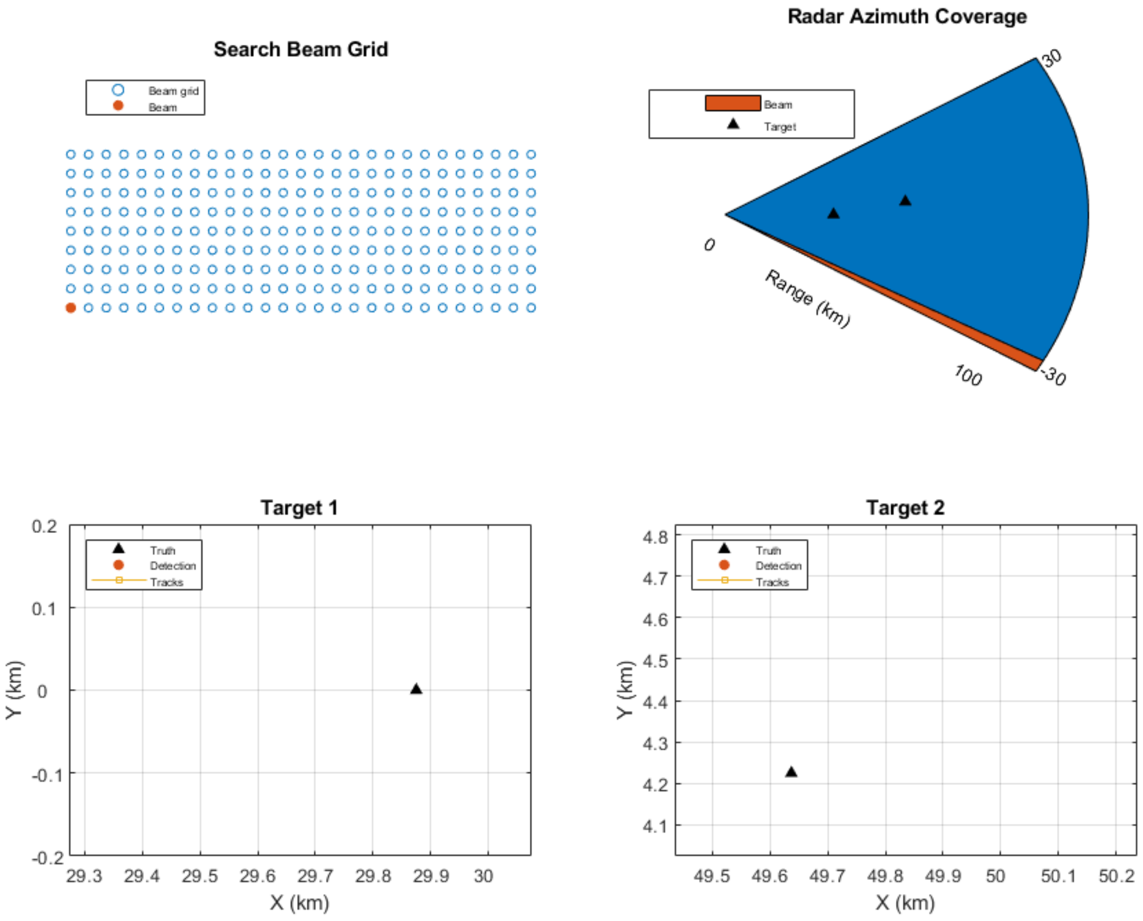
The search tasks can be considered as deterministic. In this example, a raster scan is used to cover the desired airspace. If no other tasks exist, the radar scans the space one angular cell at a time. The size of an angular cell is determined by the beam width of the antenna array.

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);
```



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.

```
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.

```
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.

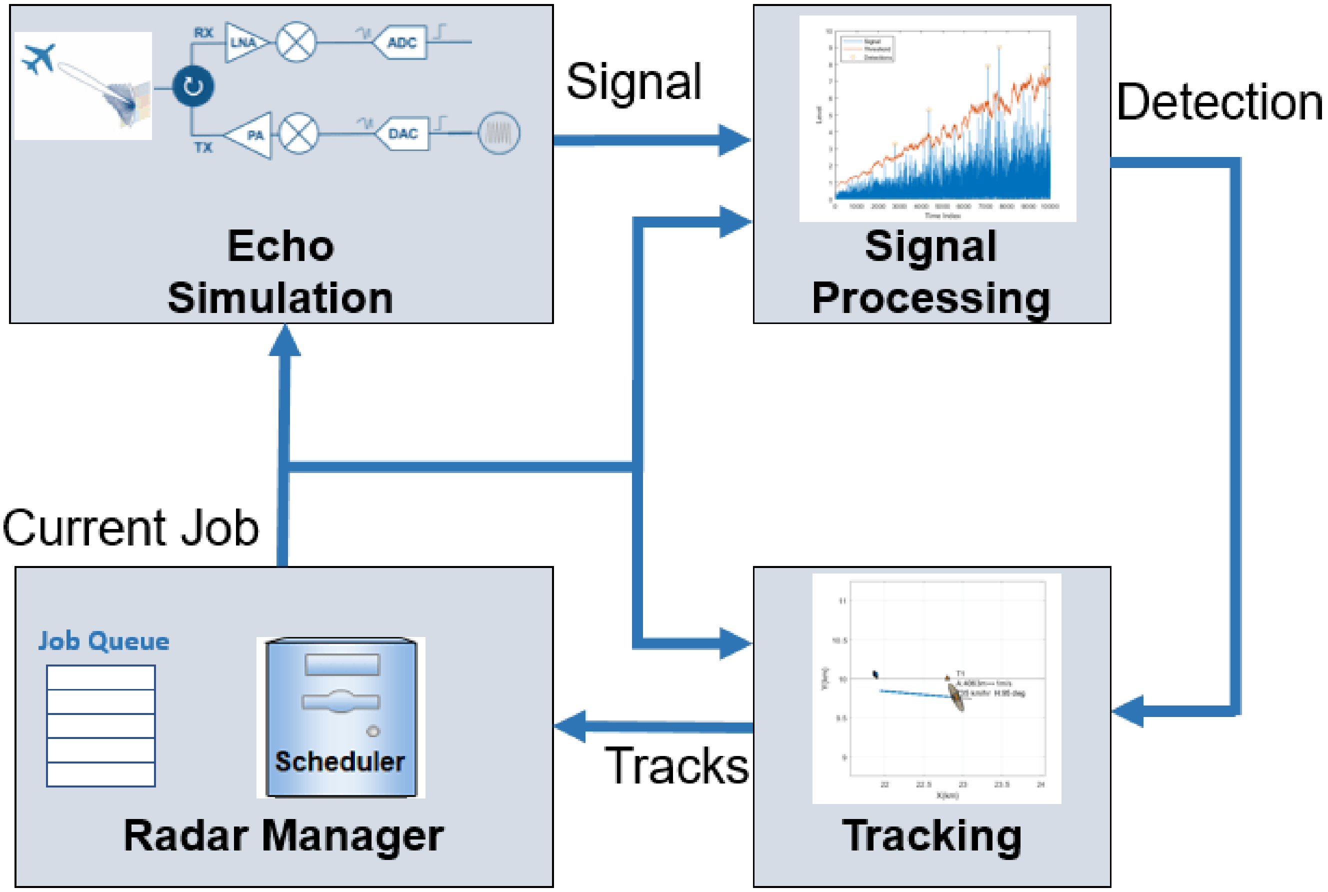
The size limit for the queue in this example is set to 10.

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.

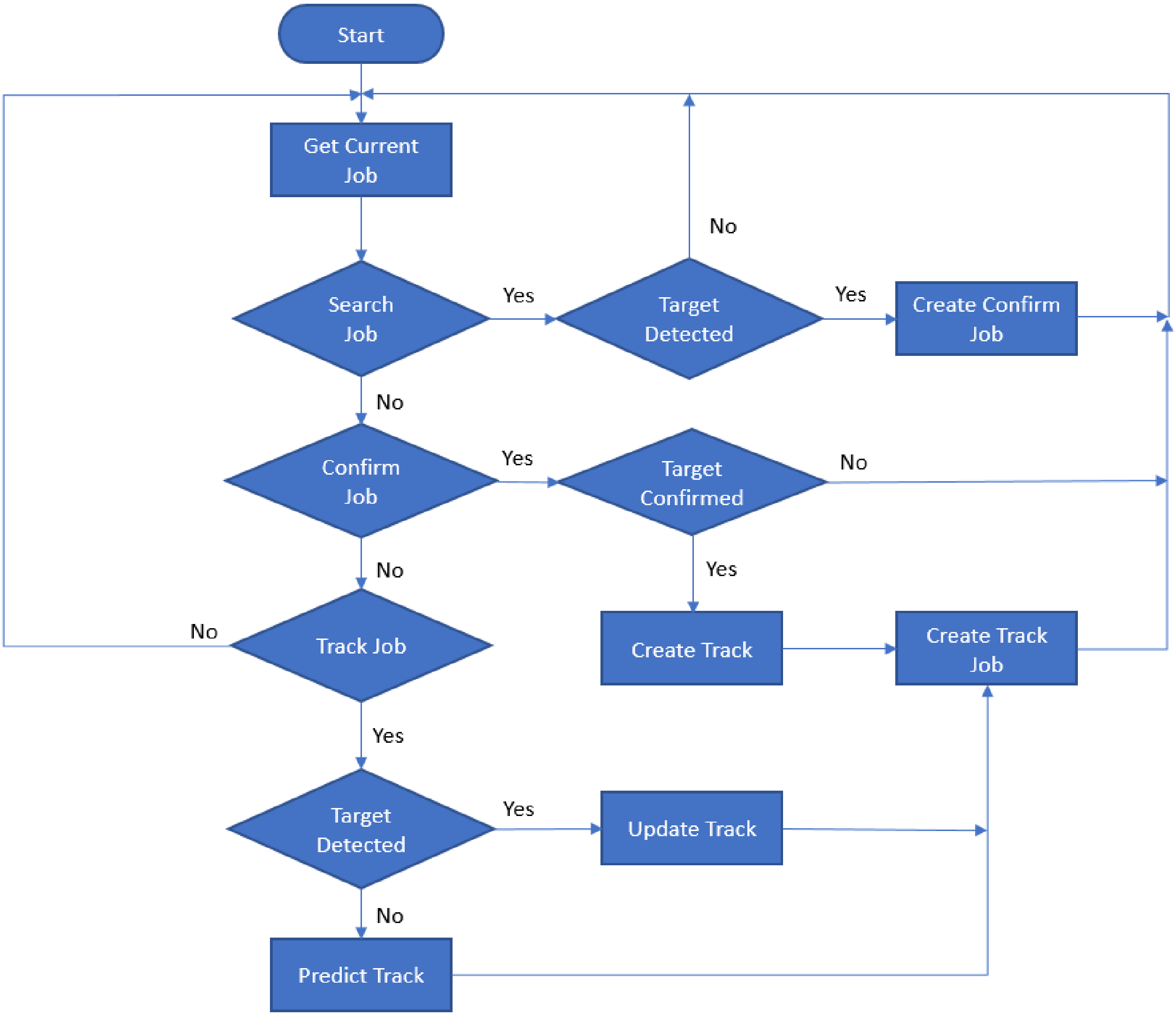
Simulation

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);
```

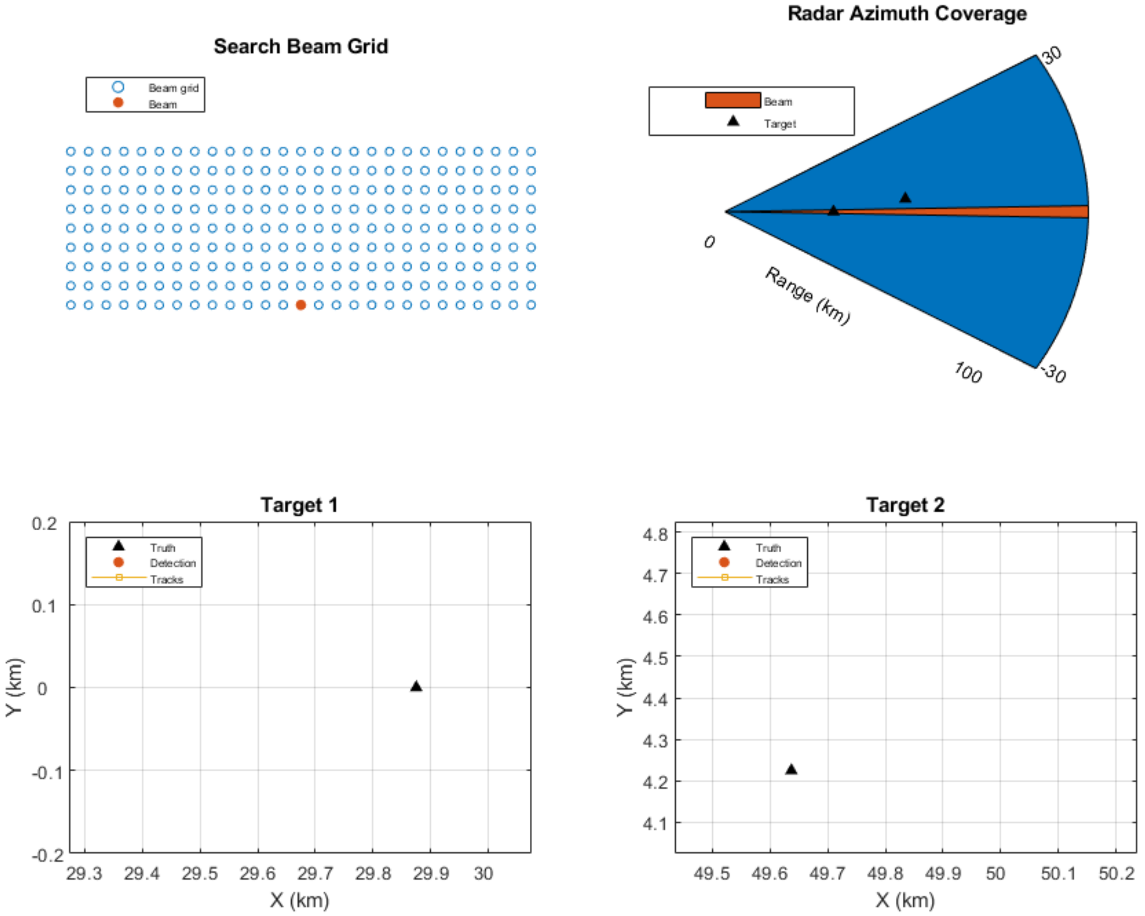
You can run the example in its entirety to see the plots being dynamically updated during execution. In the top two plots, the color of the beams indicates the types of the current job: red for search, yellow for confirm, and purple for track. The two plots below show the true locations (triangle), detections (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
```

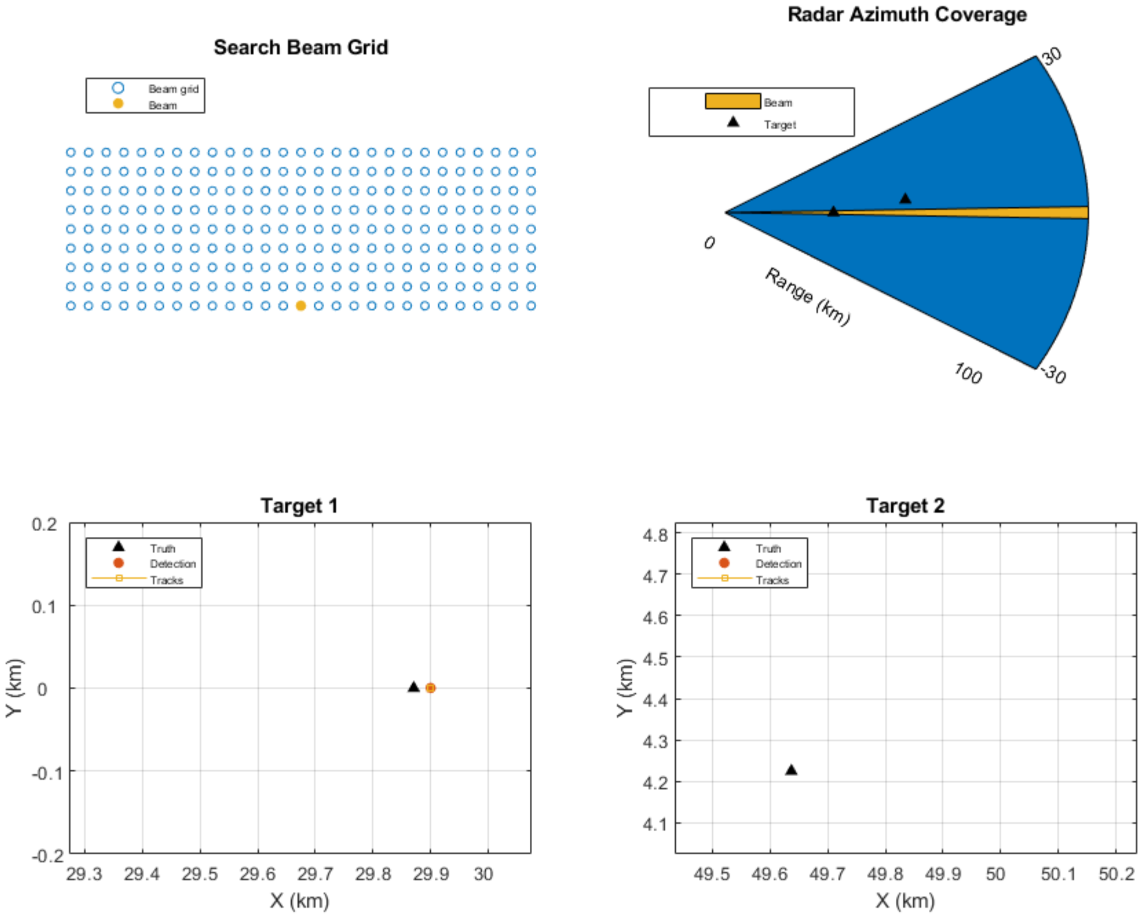


As expected, the radar gets a detection when the radar beam illuminates the target, as shown in the figure. When this happens, the radar sends a confirmation beam immediately to make sure it is not a false detection.

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
```

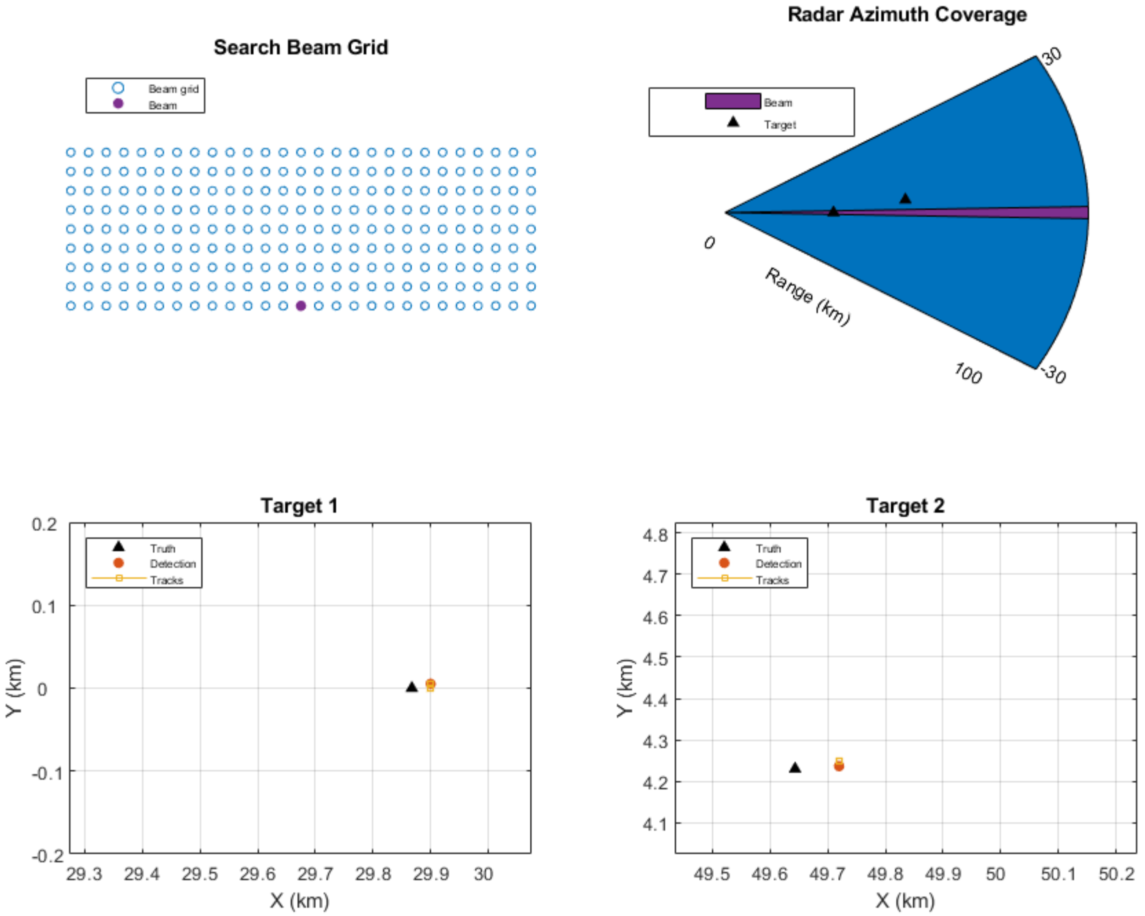


The figure now shows the confirmation beam. Once the detection is confirmed, a track is established for the target, and a track job is scheduled to execute after a short interval.

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
```



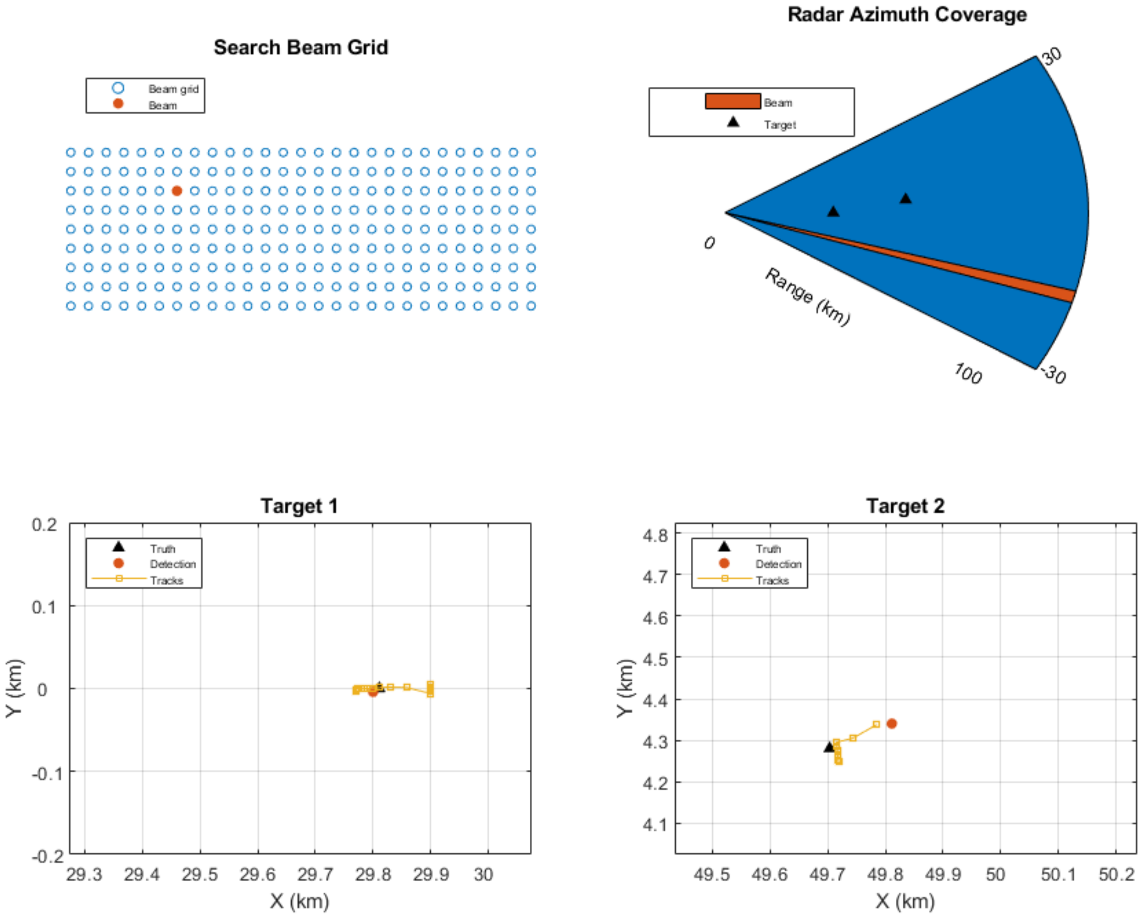
The results show that the simulation stops at a track beam. The zoomed-in figures around the two targets show how the tracks are updated based on the detection and measurements. A new track job for the next revisit is also added to the job queue during the execution of a track job.

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
```

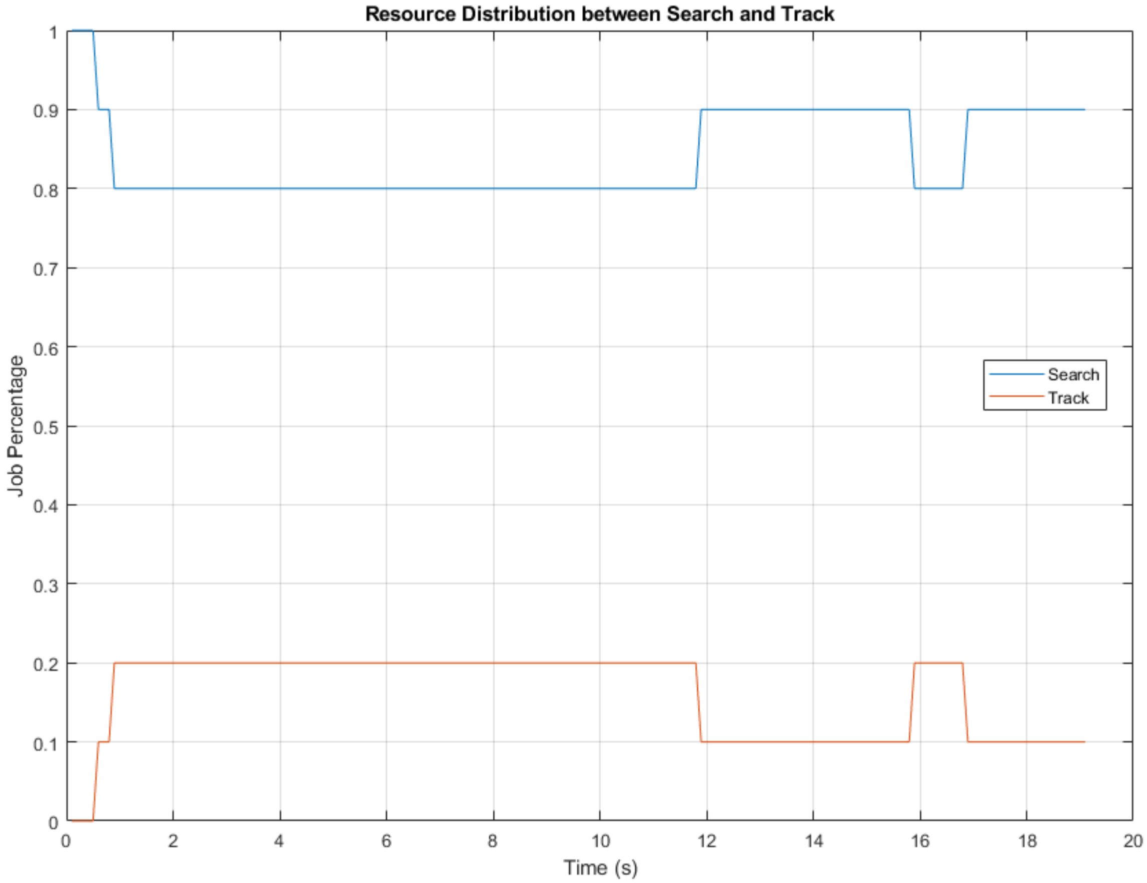
```
0.570000 sec:    Track    [4.905253 -0.023520]    Track 2 at 49900.000000 m
0.640000 sec:    Track    [-0.009124 0.012105]    Track 1 at 29900.000000 m
0.670000 sec:    Track    [4.908435 -0.012902]    Track 2 at 49900.000000 m
0.740000 sec:    Track    [-0.014700 -0.007442]    Track 1 at 29800.000000 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.

```
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.

```
function [xrsint,xrdazint,xrdelint,mfradar] = generateEchos(mfradar,env,current_job)

% Number of pulses and operating frequency
Npulses = 10;
fc = mfradar.TxAnt.OperatingFrequency;

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.

```
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.Bear
    % 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;
            [~,~,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_ti
                '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_ti
                '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_ti
                '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.

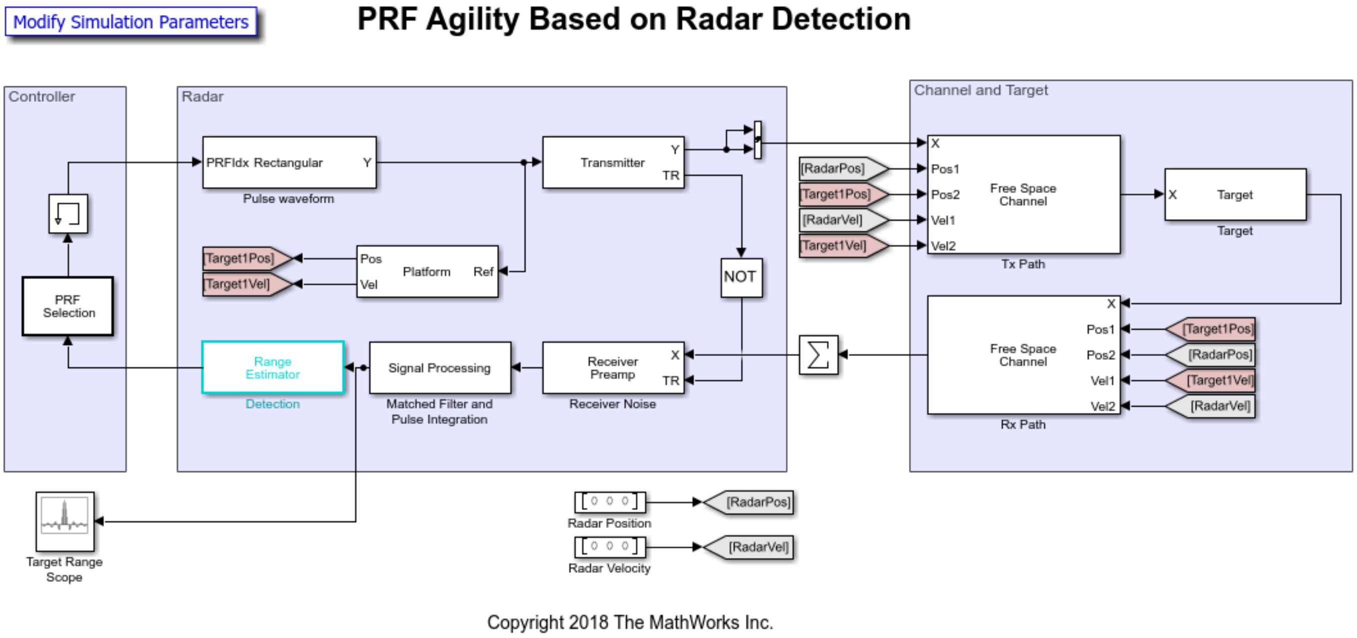
Available Example Implementations

This example includes one 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.



The system is very similar to what is used in the “Simulating Test Signals for a Radar Receiver in Simulink” 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. 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 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.

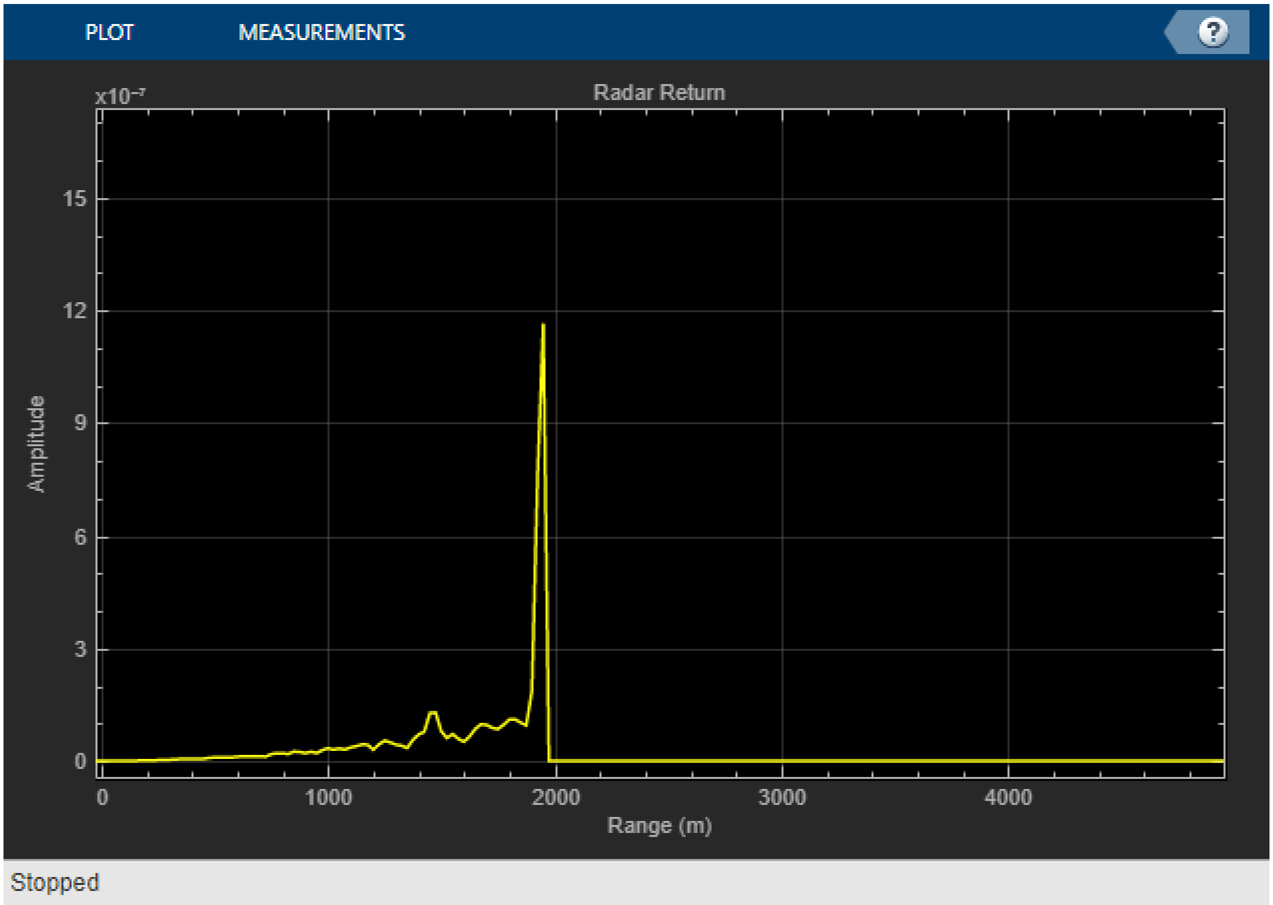
Exploring the Example

Several dialog parameters of the model are calculated by the helper function helperslexPRFSchedulingSim. 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.

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

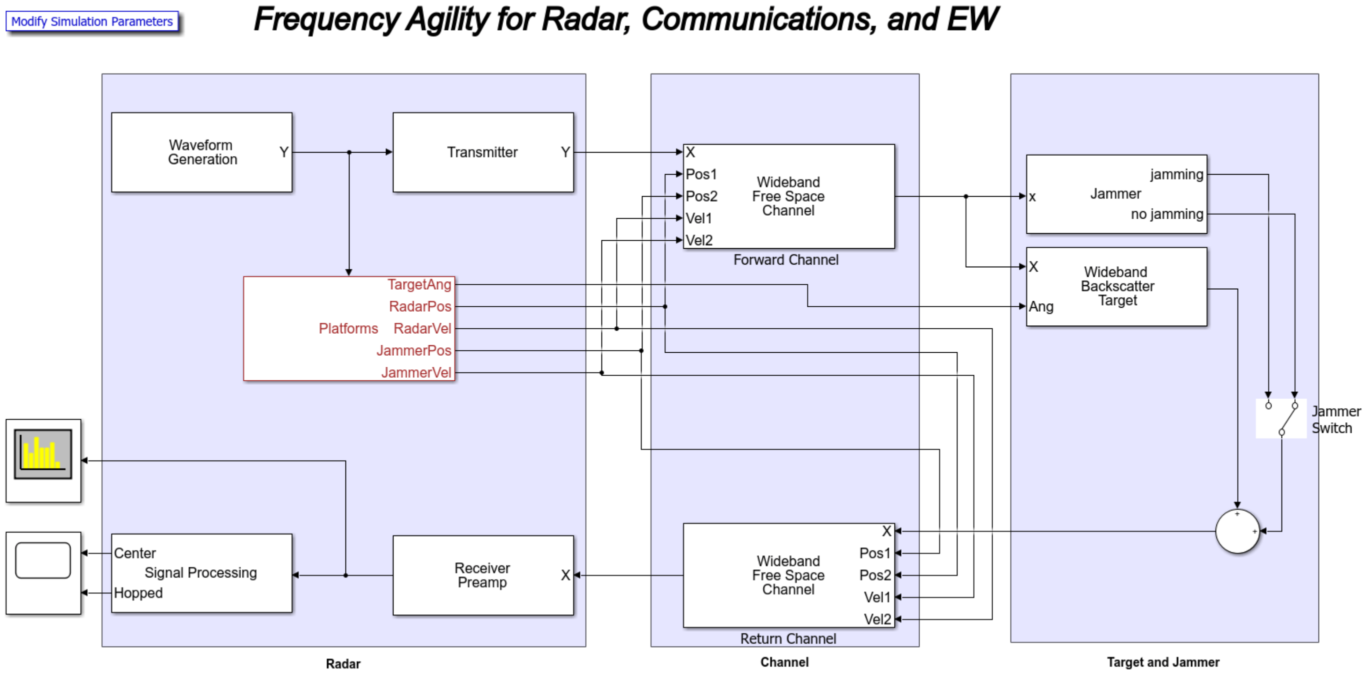
This example shows how to build a radar system in Simulink® that dynamically changes its PRF based on the target detection range. A staggered PRF system can be modeled similarly.

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 EW systems. Using Simulink, a scenario is created with a ground based radar and an approaching aircraft who can also emit jamming signals. A similar example in MATLAB can be found in “Frequency Agility in Radar, Communications, and EW Systems” on page 1-202.

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 waveform with different operating frequencies to mitigate the jamming effect. The model includes blocks for waveform generation, signal propagation, and radar signal processing. It shows how the radar and jammer interact and gain advantage over each other.

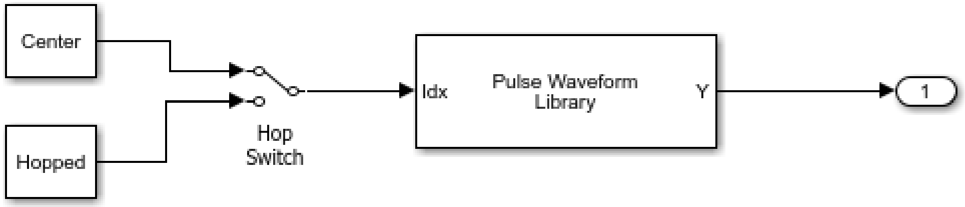


Copyright 2017-2020 The MathWorks Inc.

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 linear FM (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 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.

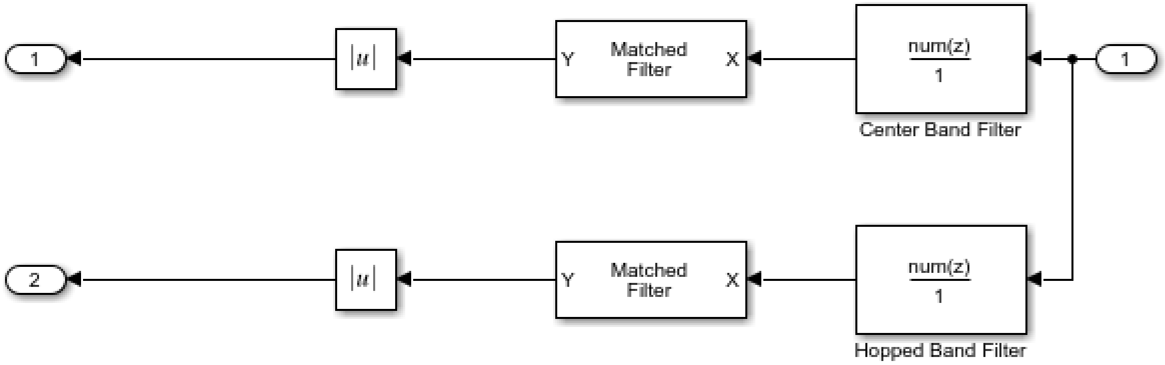


Propagation Channels

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 send back a jamming signal to confuse the radar system. That jamming signal is also propagated through the return channel. Because different signals may 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 of them 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 SNR and be ready for detection.

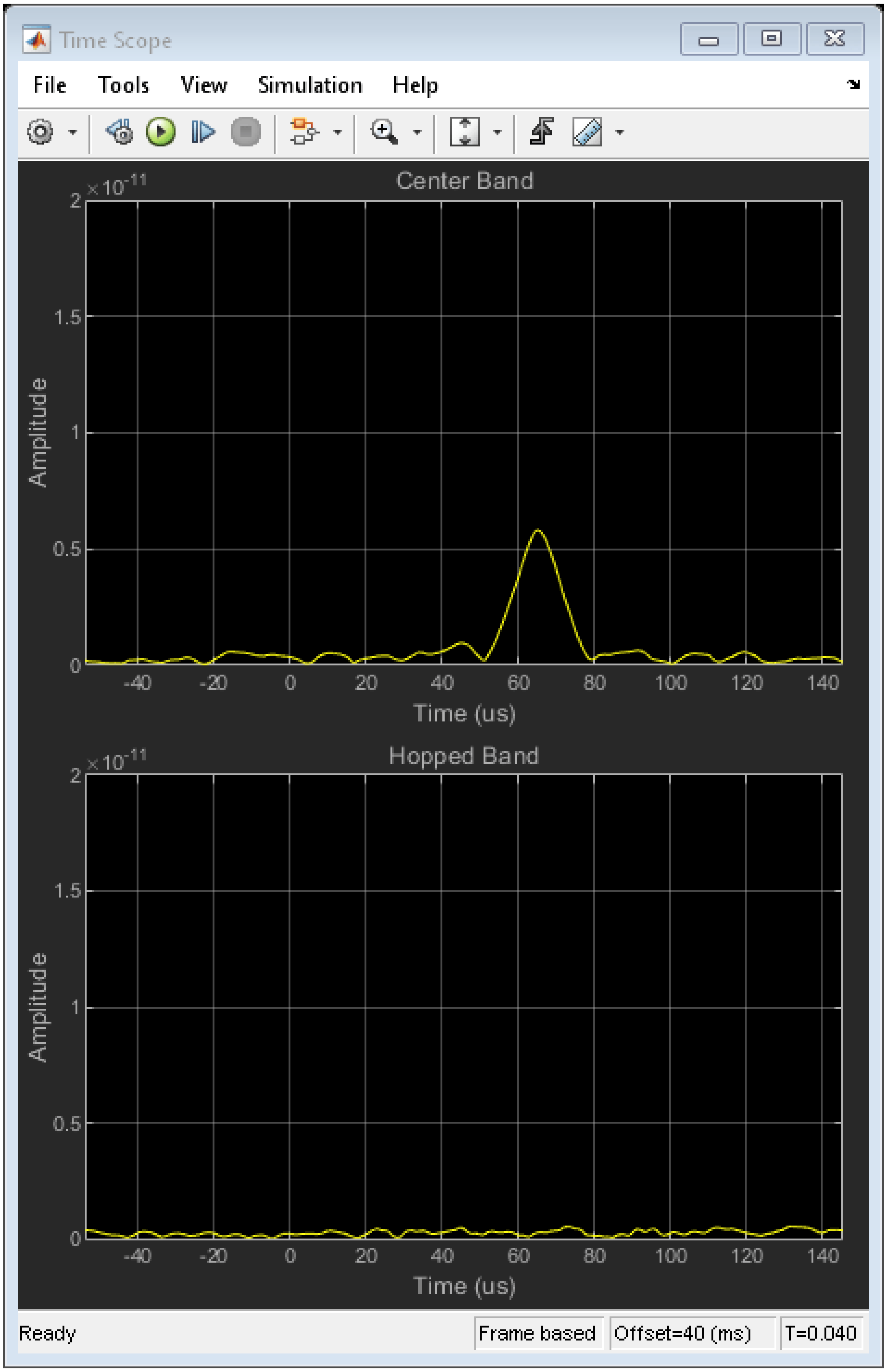


Exploring the Example

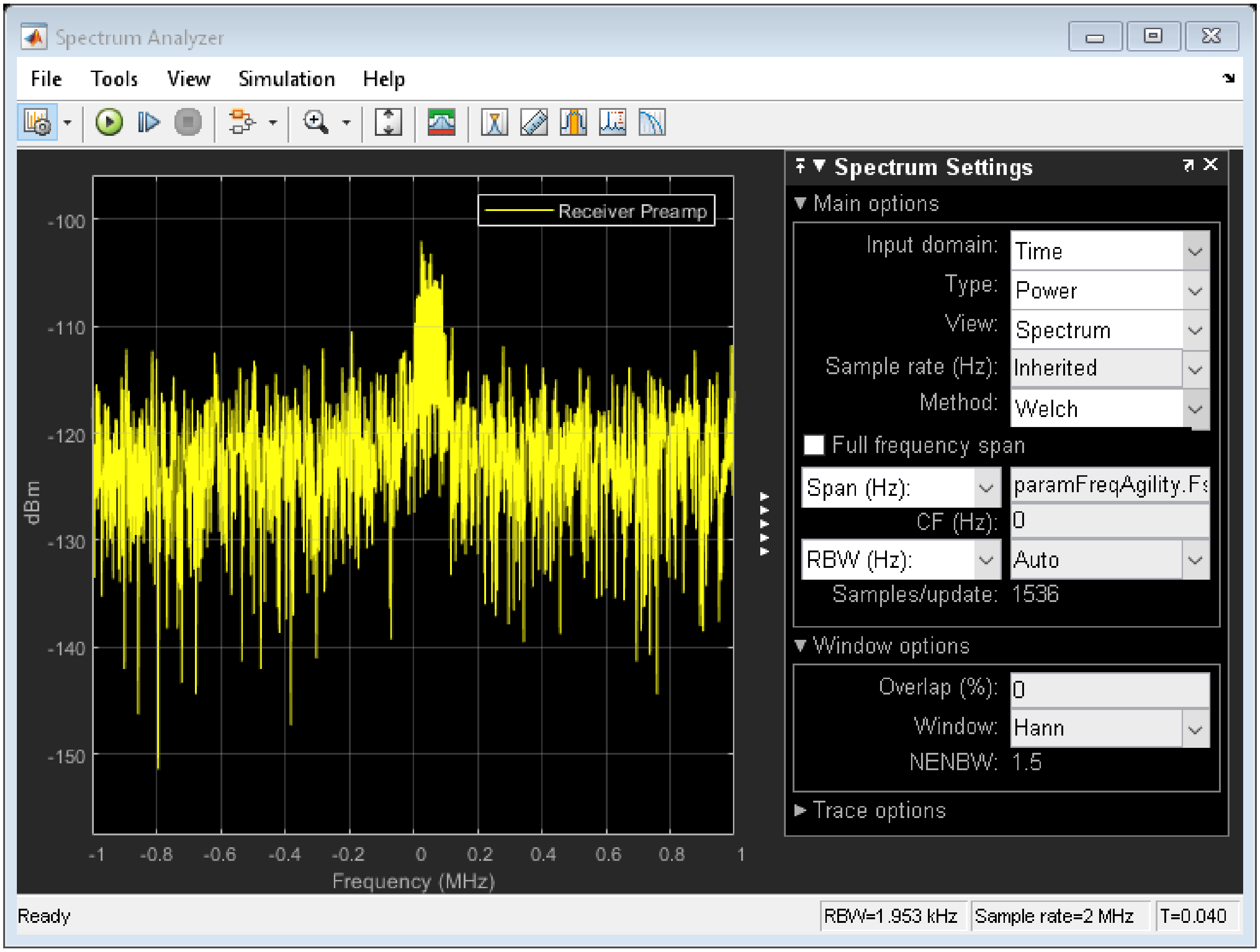
Several dialog parameters of the model are calculated by the helper function `helperslexFrequencyAgilityParam`. To open the function from the model, click on Modify Simulation Parameters block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields 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

First run the model for the case when there is no jamming signal. The scopes 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.



Spectrum Settings

▼ Main options

Input domain: Time
Type: Power
View: Spectrum
Sample rate (Hz): Inherited
Method: Welch
□ Full frequency span
Span (Hz): paramFreqAgility.Fs
CF (Hz): 0
RBW (Hz): Auto
Samples/update: 1536

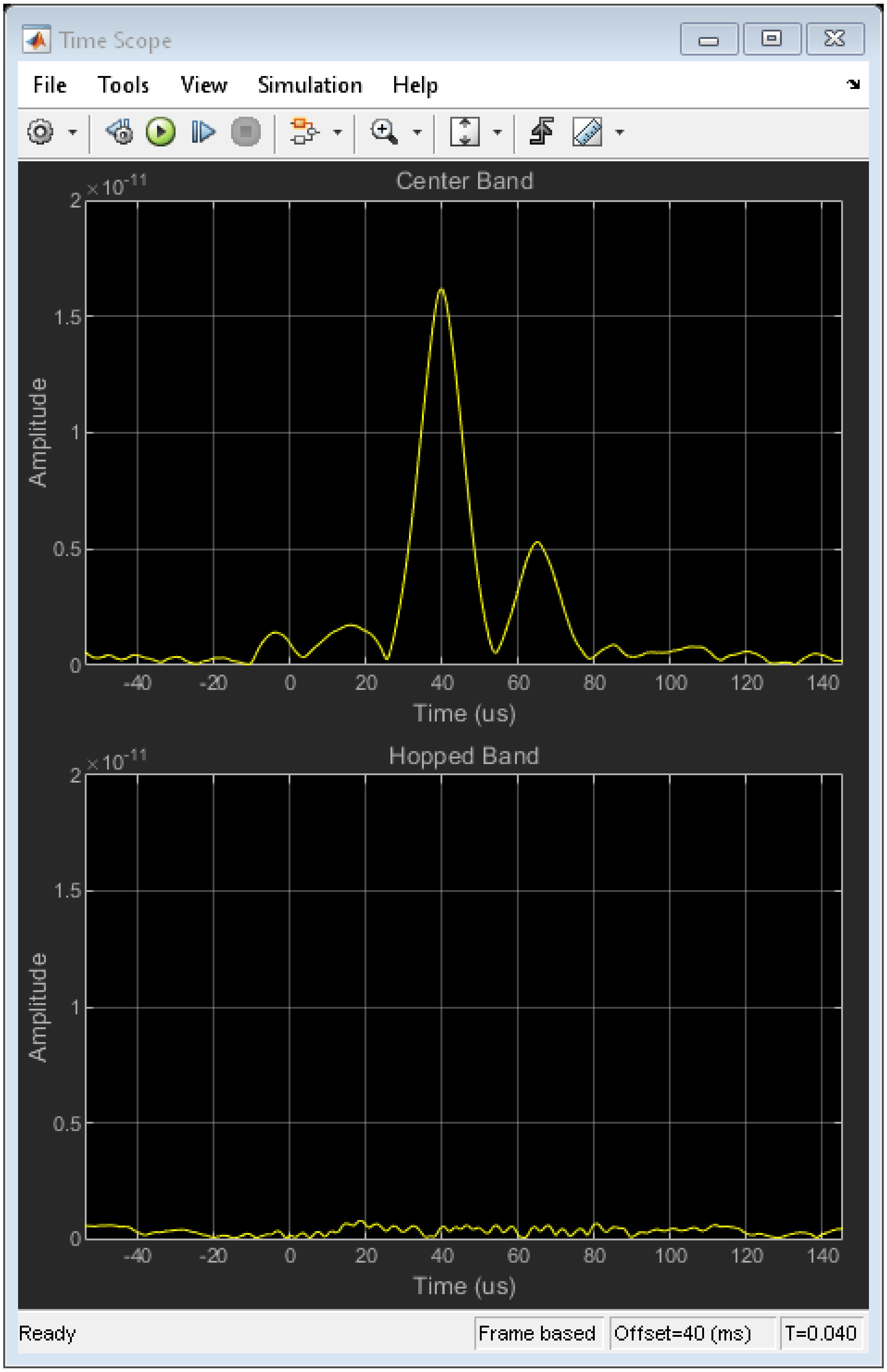
▼ Window options

Overlap (%): 0
Window: Hann
NENBW: 1.5

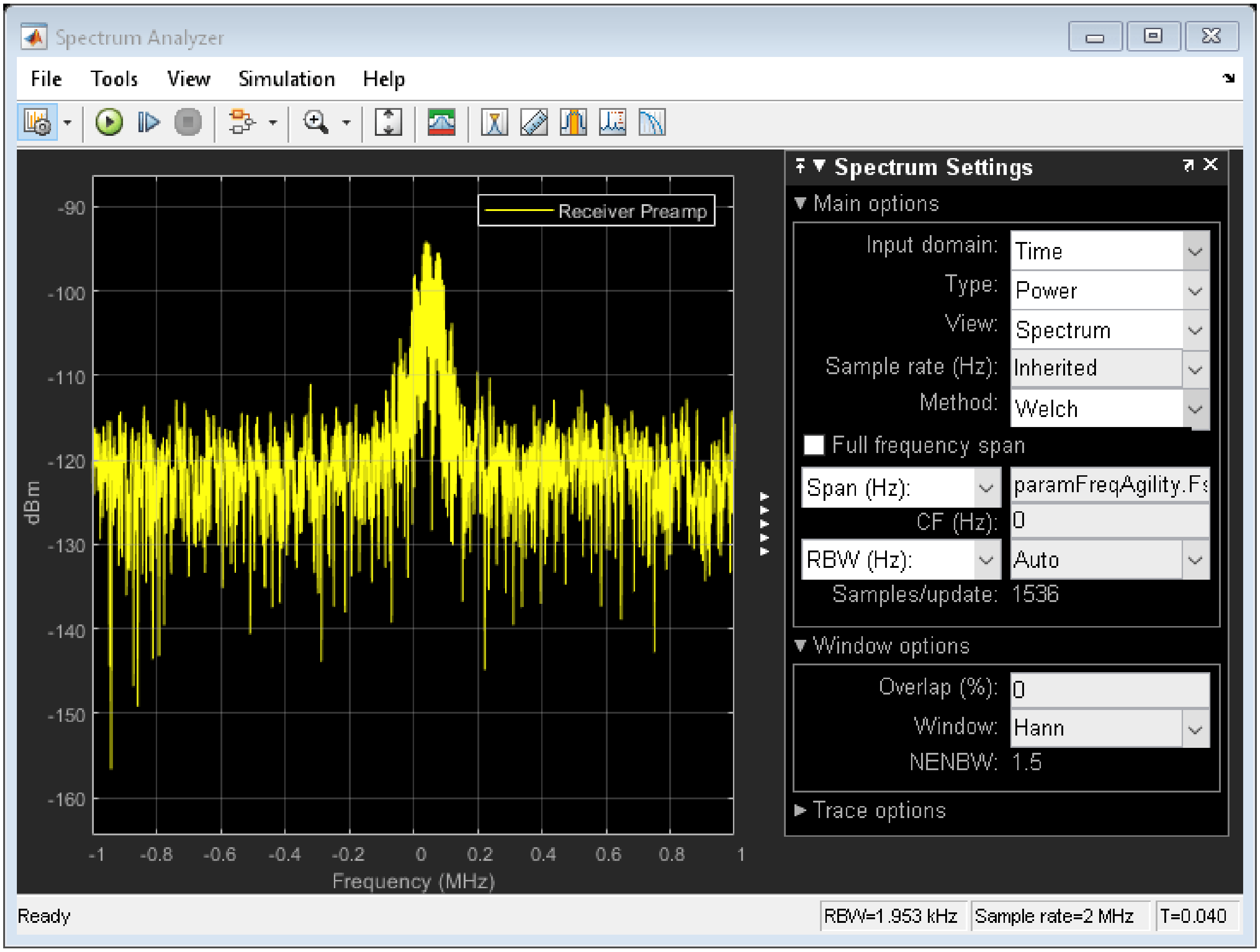
▶ Trace options

Ready | RBW=1.953 kHz | Sample rate=2 MHz | T=0.040

Now enable jammer by clicking the Jammer Switch block. In this situation, the target intercepts the signal, amplify 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.



Input domain: Time

Type: Power

View: Spectrum

Sample rate (Hz): Inherited

Method: Welch

□ Full frequency span

Span (Hz): paramFreqAgility.Fs

CF (Hz): 0

RBW (Hz): Auto

Samples/update: 1536

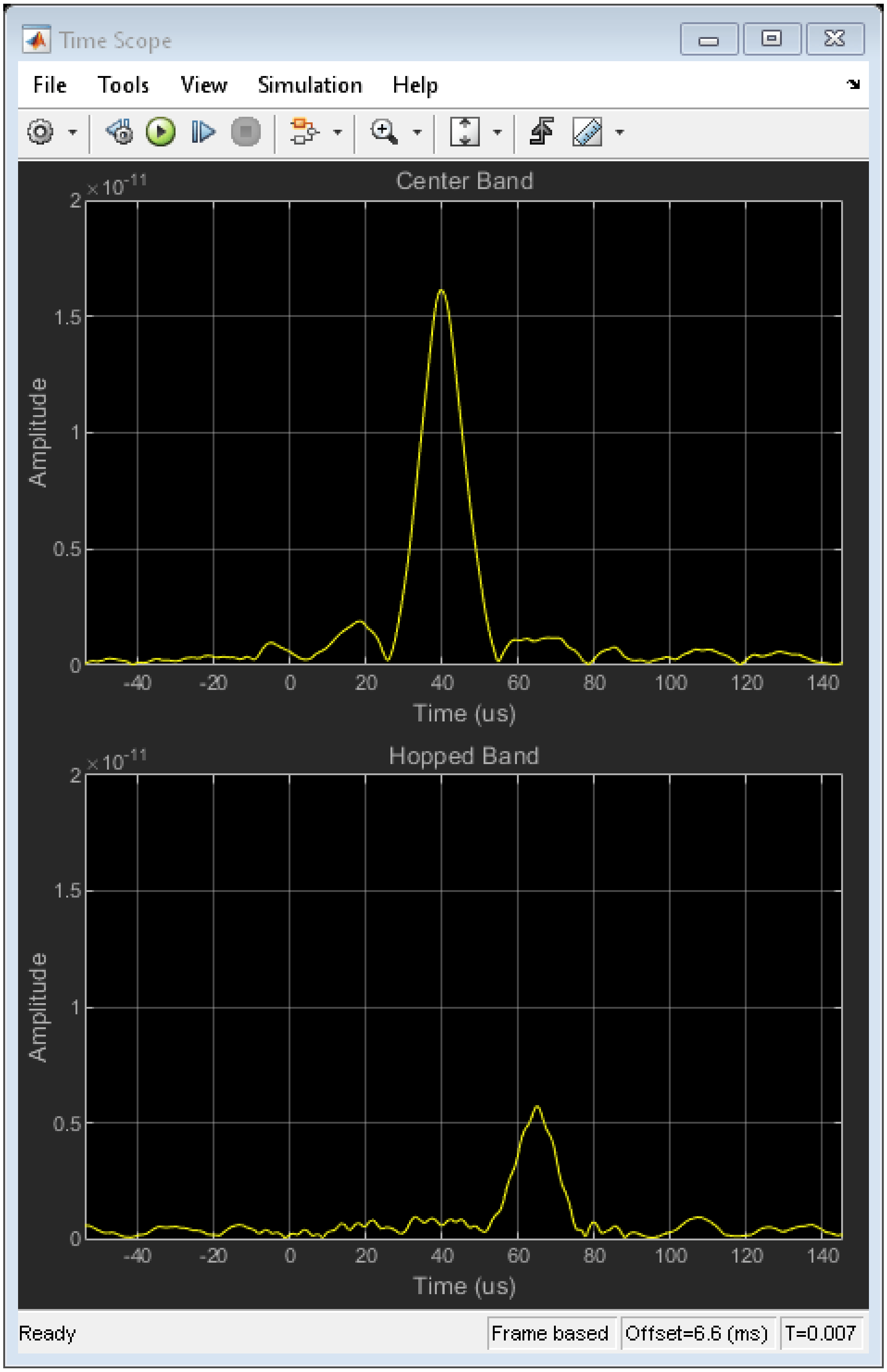
Overlap (%): 0

Window: Hann

NENBW: 1.5

Ready RBW=1.953 kHz Sample rate=2 MHz T=0.040

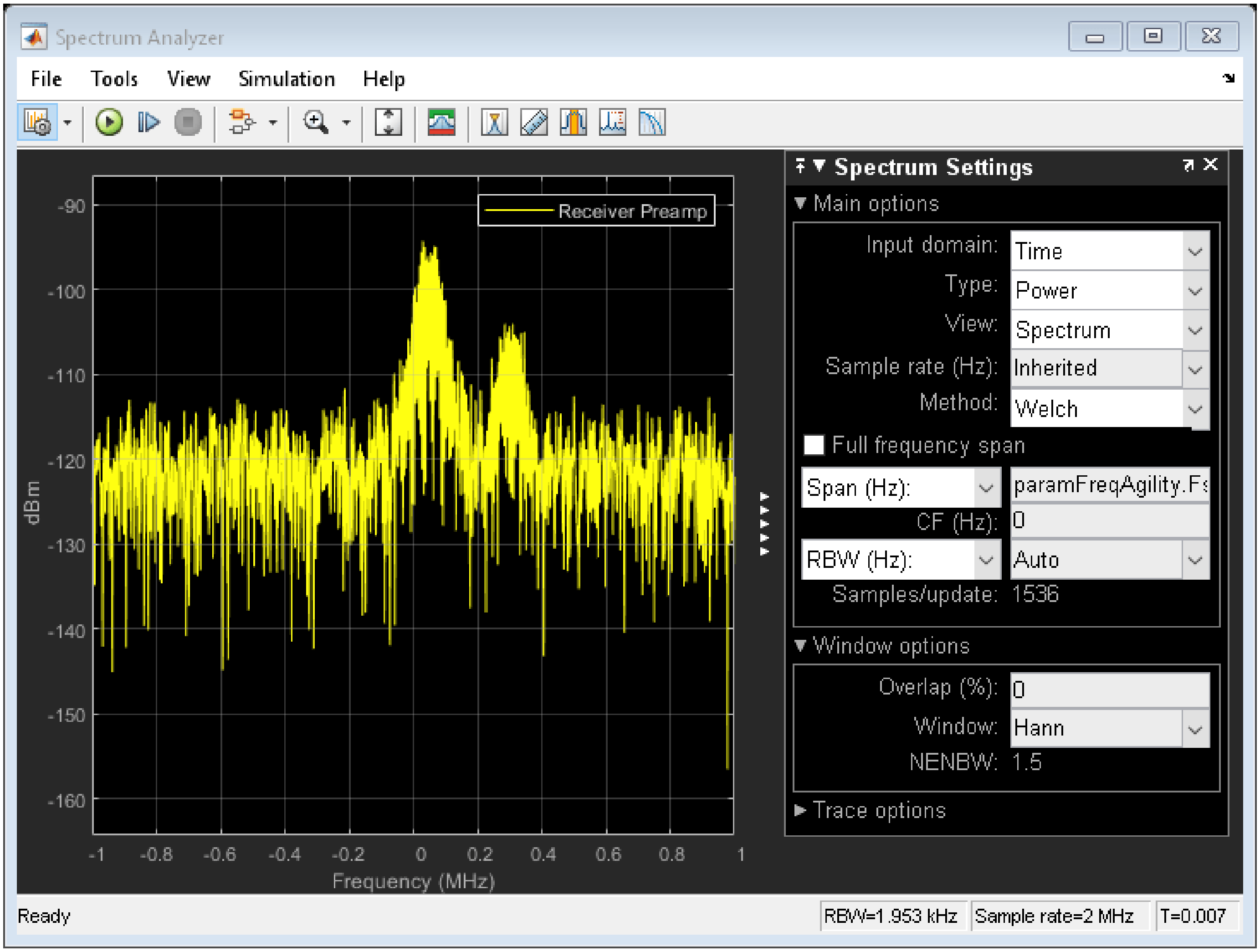
If the radar has a pre-scheduled frequency hopping schedule or is smart enough to know that it might have been confused by a jamming signal, it could switch to a different frequency band to operate. Such scenario can be simulated by clicking the Hop Switch block so the radar signal is transmitted in the hopped band.



Ready | Frame based | Offset=6.6 (ms) | T=0.007

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 hasn't figured out the radar's new operating band yet 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.



Input domain: Time

Type: Power

View: Spectrum

Sample rate (Hz): Inherited

Method: Welch

□ Full frequency span

Span (Hz): paramFreqAgility.F

CF (Hz): 0

RBW (Hz): Auto

Samples/update: 1536

Overlap (%): 0

Window: Hann

NENBW: 1.5

Ready | RBW=1.953 kHz | Sample rate=2 MHz | T=0.007

Summary

This 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 EW 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.

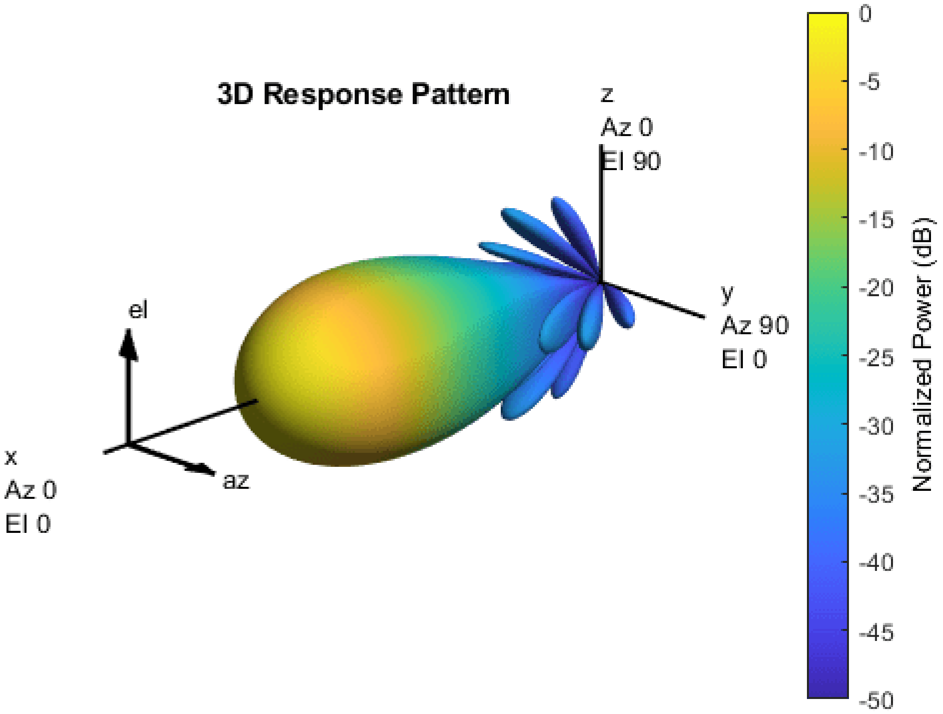
```
fc = 10e9;       % Define frequency of operation for X-band Radar
fs = 2e6;        % Define sample rate for the systems
c = 3e8;
lambda = c/fc;

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 (8x8) 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. The transmitter and receiver specifications are:

```
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);
```

The environment and target are described below. Wideband propagation channels are used to allow us 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);
```

In this example, two one-way propagation channels are used because the jamming signal only propagates through the return channel.

The received echo can be simulated as

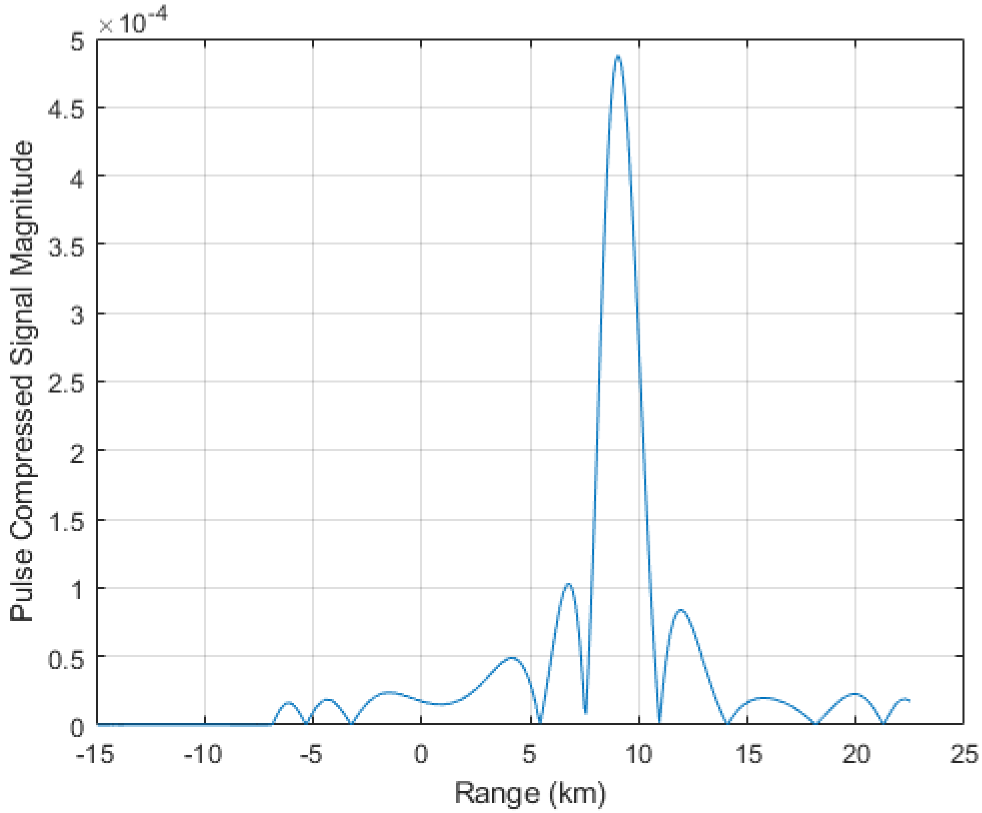
```
rng(2017);
[tgtRng, tgtAng] = rangeangle(target_pos, radar_pos);
x = wav(0);                                                % waveform
xt = tx(x);                                                % transmit
xtarray = txArray(xt, tgtAng);                             % radiate
yp = envout(xtarray,radar_pos,target_pos,radar_vel,target_vel); % 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
```

We can perform a direction of arrival estimate using a 2D beam scan and used 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);
```

The beamformed signal can then be passed through matched filter and detector.

```
mfcoeff1 = getMatchedFilter(wav);
mf1 = phased.MatchedFilter('Coefficients',mfcoeff1);
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');
```



The figure shows that the target produces a dominant peak in the received signal.

Signal Analysis at the Target/Jammer

The radar works very well in the above example. However, in a complex environment, interferences can affect the radar performance. The interferences 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 not closely located in angular space, beamforming may 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 the 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.

The detected signal characteristics are displayed below

```
pw = (pulsewidth(abs(yp), fs));
prf = round(1/pulseperiod(abs([yp;yp]), fs));
bw= obw(yp,fs,[],95);

fprintf('Waveform Characteristics:\n');
```

```
Waveform Characteristics:
```

```
fprintf('Pulse width:\t\t%f\n',pw);
```

```
Pulse width:        0.000100
```

```
fprintf('PRF:\t\t\t%f\n',prf);
```

```
PRF:            4000.000000
```

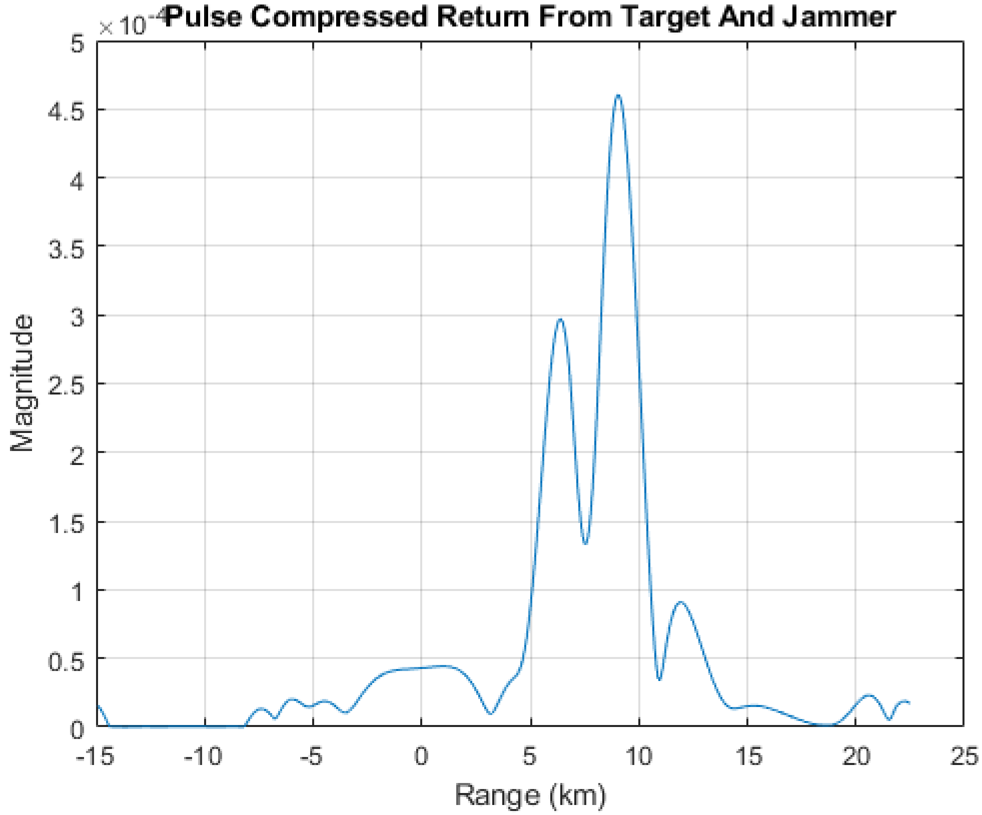
```
fprintf('Sweep bandwidth:\t%f\n',bw);
```

```
Sweep bandwidth:    112041.098255
```

The jammer needs some time to do these analysis and prepare for the jamming signal so it is hard to create an effective spoofing signal right away, but generally within several pulses intervals the jamming signal is ready and the jammer can put it within arbitrary position in a pulse to make the spoofing target look closer or father 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 out, 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.

```
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 on to the closest target thinking that one is the most prominent threat and spend less resource on the true target.

Frequency Agility to Counter Interference

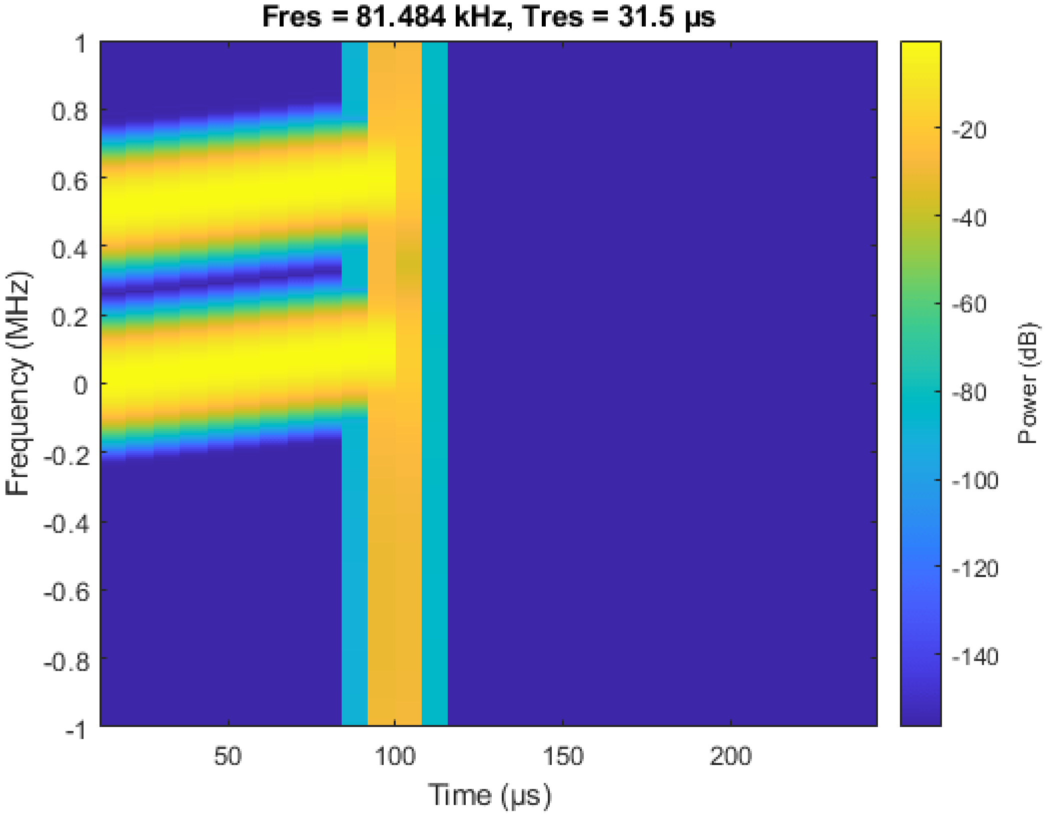
One possible approach to mitigate the jamming effect at the radar receiver is to adopt a predefined frequency hopping schedule. In this case, the waveform transmitted from radar may change carrier frequency from time to time. Since the hopping sequence is only known to the radar, the jammer would not 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 jammer to be able to handle transmission of signals over a broader bandwidth. Thus, the frequency hop can create a time interval that radar operates 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 becomes

```
deltaf = fs/4;
xh = wav(deltaf);  % hopped signal
```

The figure below 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')
```



Using similar approach outlined in 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 are suppressed. In addition, when the waveform hops, the matched filter needs to be updated accordingly.

Let us now apply the corresponding bandpass filters and matched filters to the received signal.

First, create a bandpass filter using the signal's bandwidth.

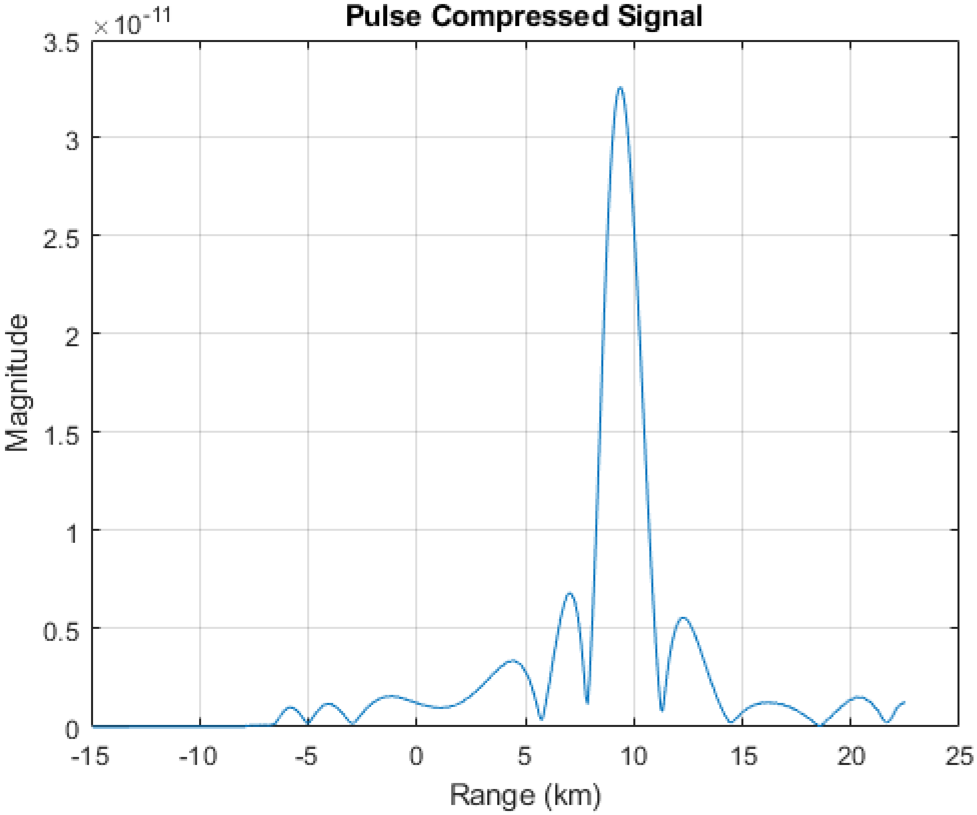
```
buttercoef = butter(9,bw/fs);
```

Then, we can modulate the resulting bandpass filter with a carrier to obtain the bandpass filter around that carrier frequency.

```
bf2 = buttercoef.*exp(1i*2*pi*deltaf*(0:numel(buttercoef)-1)/fs);
```

Similarly, the matched filter coefficient needs to be modulated too.

```
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');
```



The 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 where 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 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

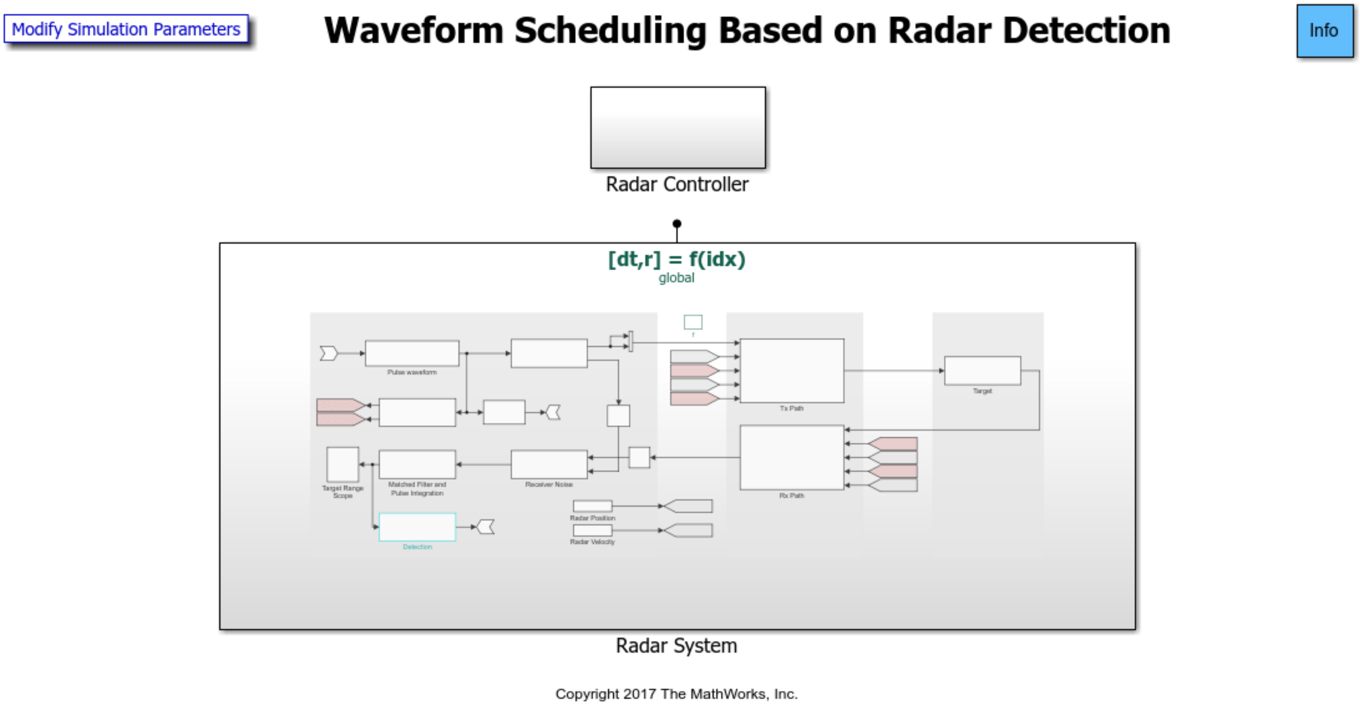
This example includes one Simulink® model:

- Dynamic PRF Selection Based on Radar Detection: slexPRFSelectionSEExample.slx

Dynamic PRF Selection Based on Radar Detection

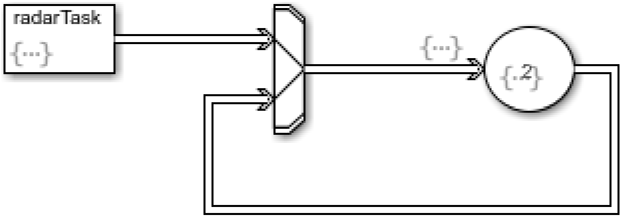
This model simulates a monostatic radar that searches for targets with an unambiguous range of 5 km. If the radar detects a target within 2 km, then it will switch to a higher PRF to only look for targets with 2 km range and enhance its capability to detect high speed targets.

```
helperslexPRFSelectionSim('openModel');
```



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.

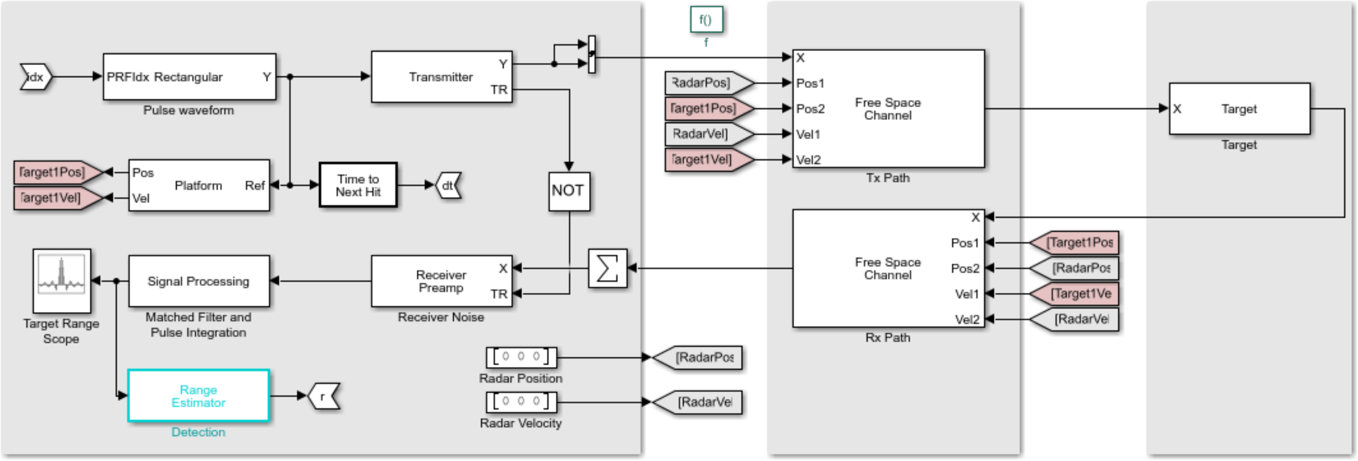
```
helperslexPRFSelectionSim('showController');
```



Radar System

The radar system resides in a Simulink function block and is shown in the following figure.

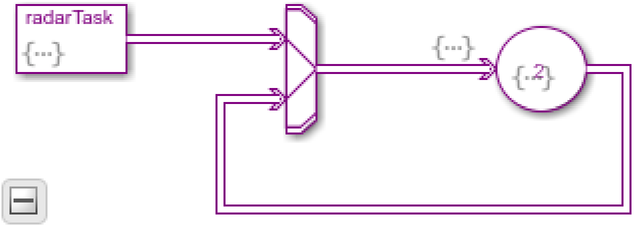
```
helperslexPRFSelectionSim('showRadar');
```



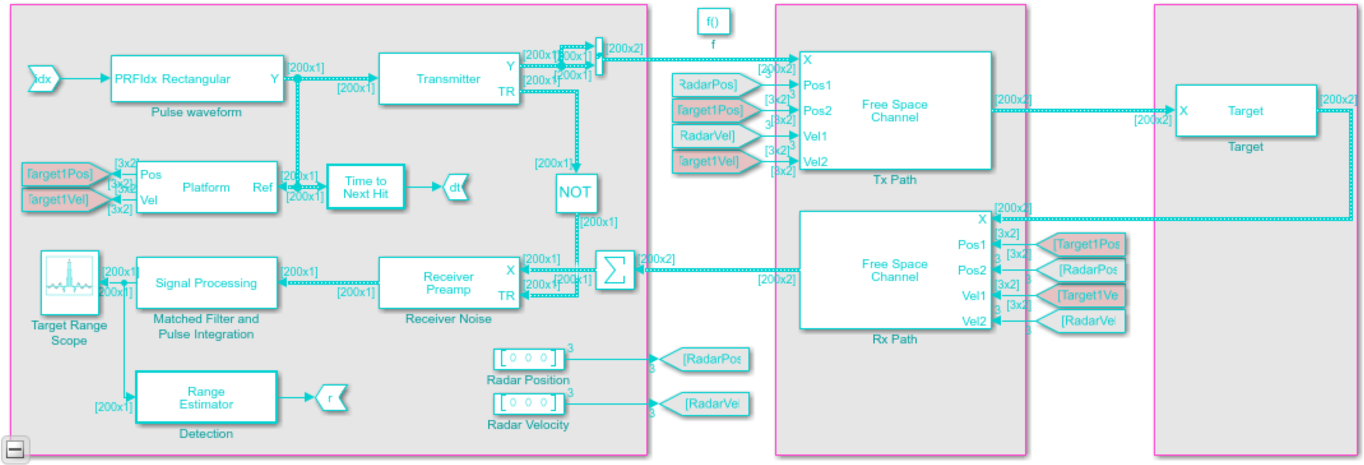
The system is very similar to what is used in the “Waveform Scheduling Based on Target Detection” example with the following notable difference:

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');
```



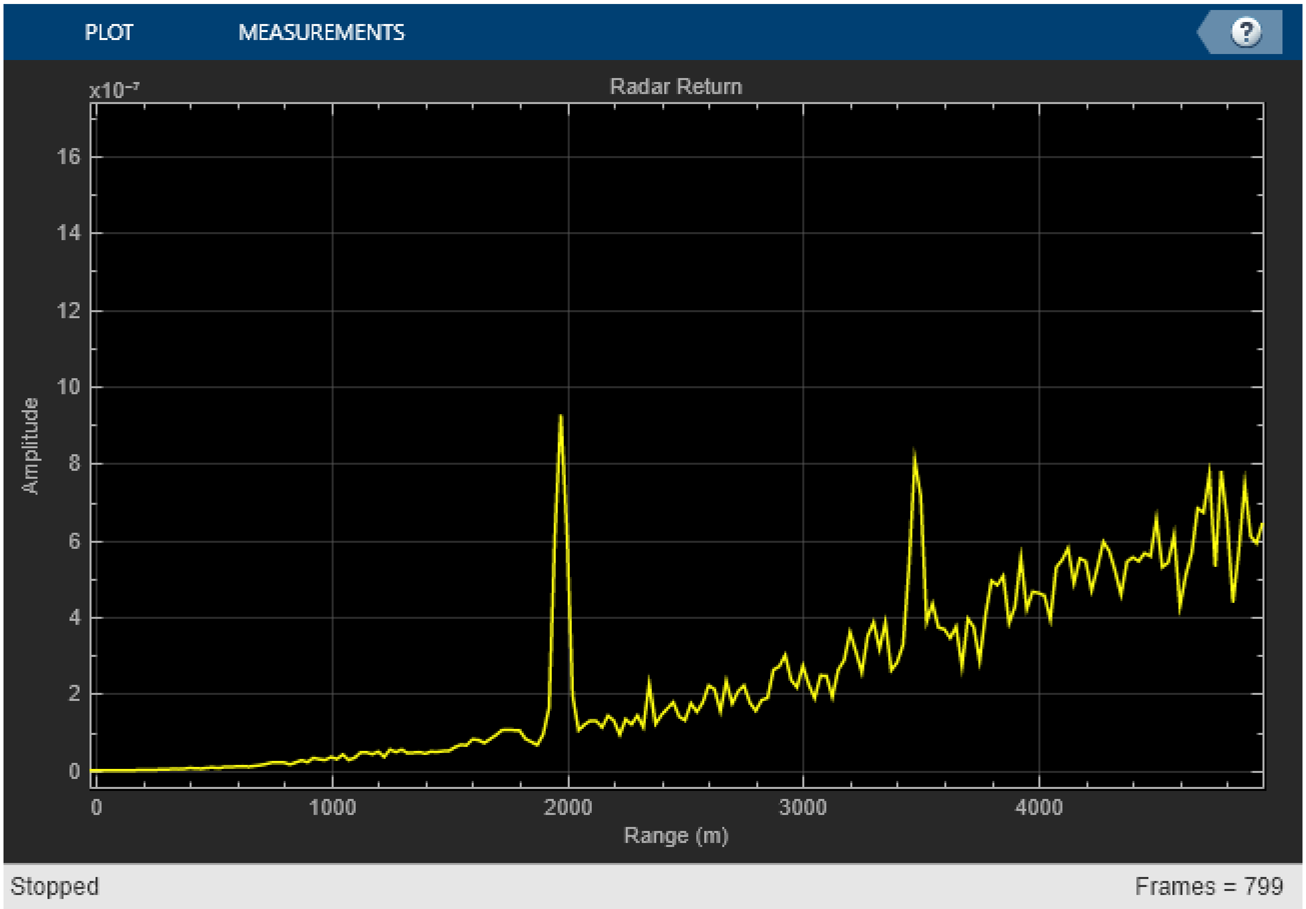
Exploring the Example

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

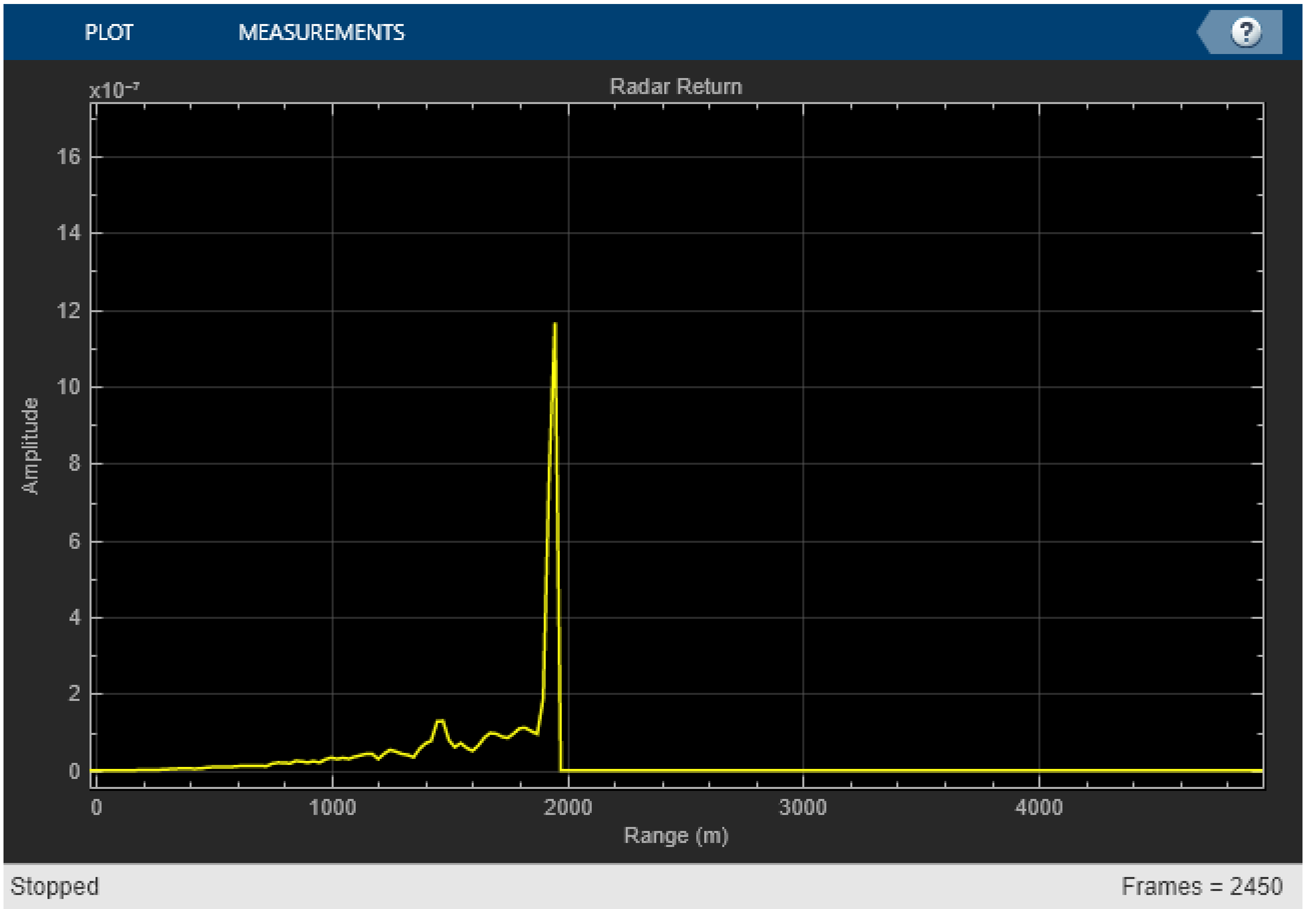
The figure below shows the detected ranges of the targets. Target ranges are computed from the round-trip time delay of the reflected signals from the targets. At the simulation start, the radar detects two targets, one is slightly over 2 km away and the other one is at approximately 3.5 km away.

```
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.

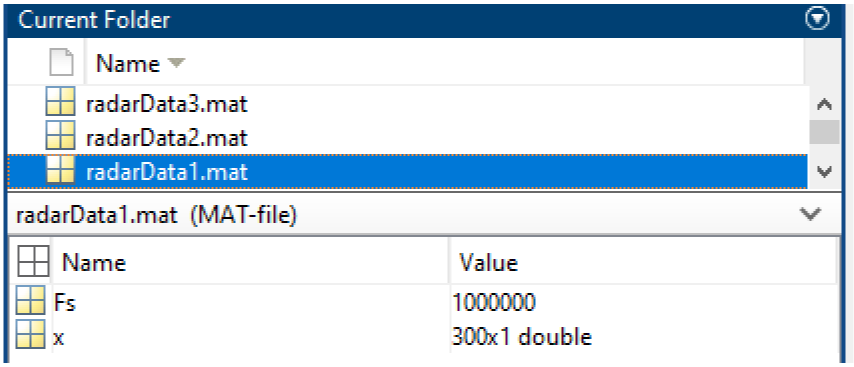
```
helperslexPRFSelectionSim('closeModel');
```

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.

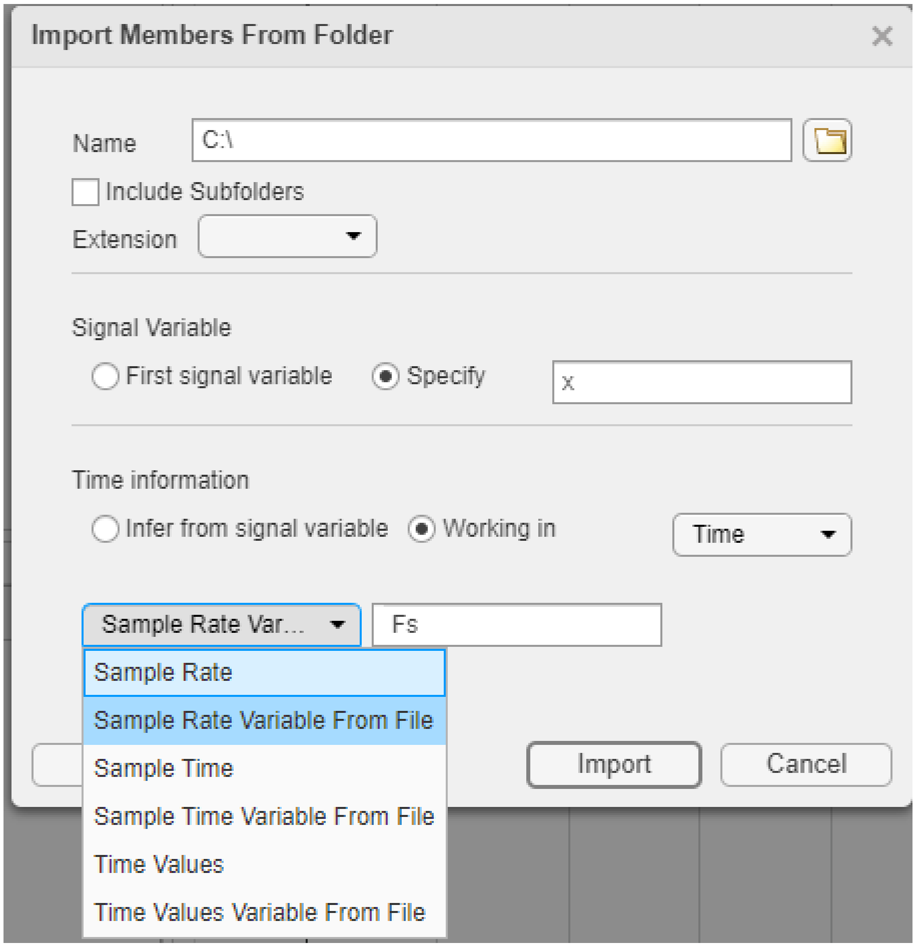
Import Data into Signal Labeler

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`.



Name	Value
Fs	1000000
x	300x1 double

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 signals by selecting the check boxes next to their names.



Import Members From Folder

Name C:\

□ Include Subfolders

Extension

Signal Variable

○ First signal variable ● Specify x

Time information

○ Infer from signal variable ● Working in Time

Sample Rate Var... Fs

Sample Rate
Sample Rate Variable From File
Sample Time
Sample Time Variable From File
Time Values
Time Values Variable From File
Import
Cancel

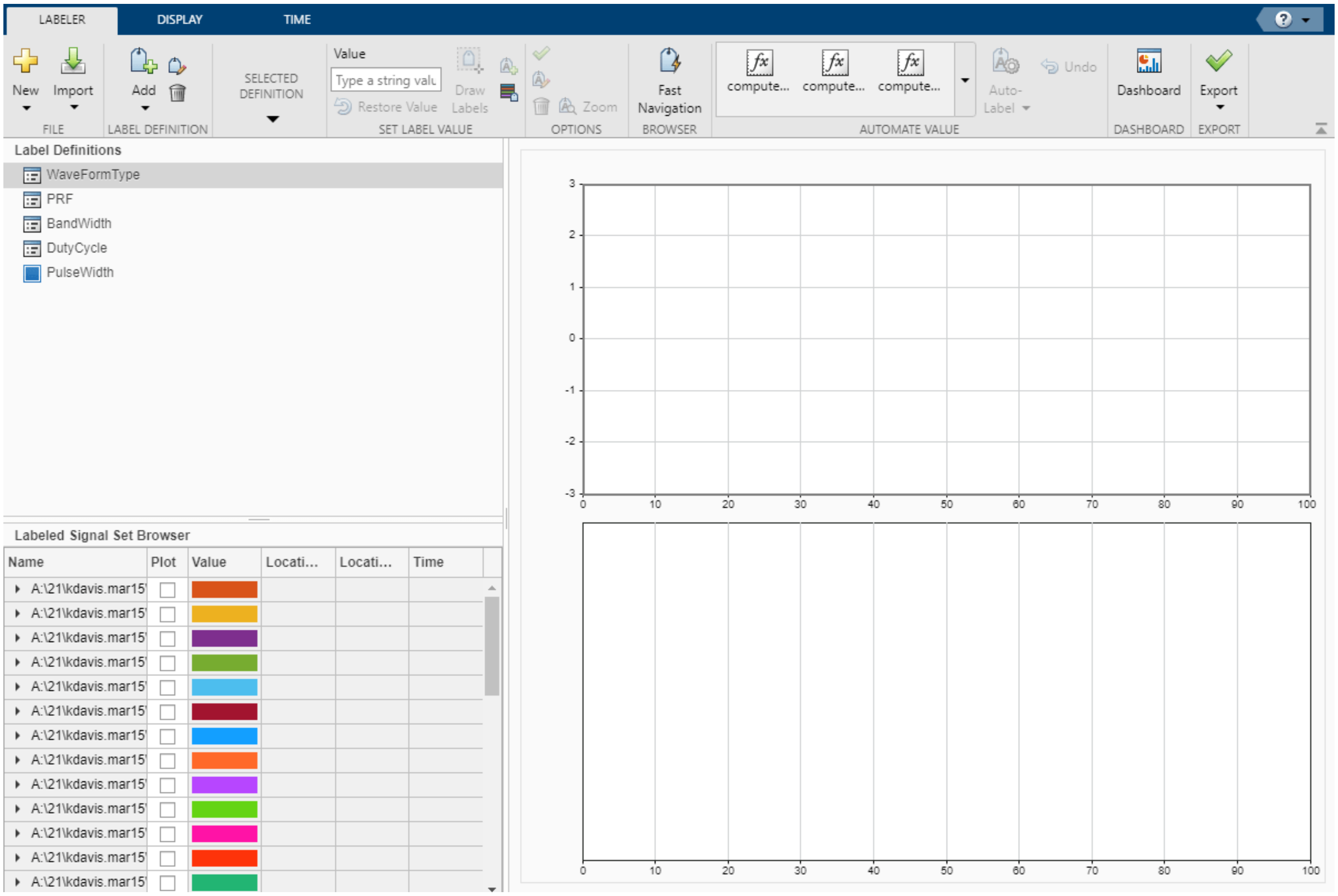
Define Labels

Create a label definition for the signal waveform type.

1 Create a string attribute label to label the waveform type. Click **Add** on the **Labeler** tab and select `Add Label Definition`.
2 In the dialog box, specify the **Label Name** as `WaveFormType and` the **Label Type** as Attribute. Set **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 as `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
- BandWidth
- DutyCycle
- PulseWidth

Labeled Signal Set Browser

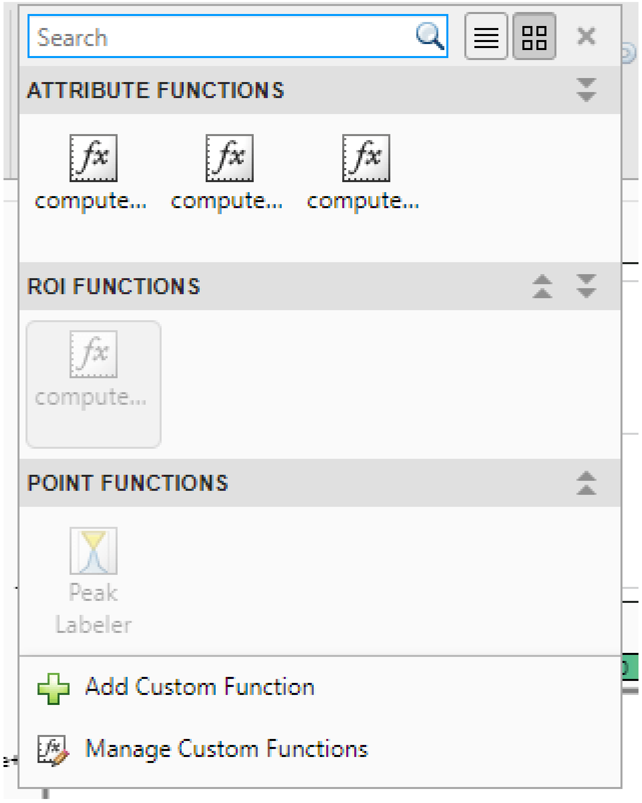
Name	Plot	Value	Locati...	Locati...	Time
A:\21\kdavis.mar15'	□	🟧			
A:\21\kdavis.mar15'	□	🟨			
A:\21\kdavis.mar15'	□	🟪			
A:\21\kdavis.mar15'	□	🟩			
A:\21\kdavis.mar15'	□	🟦			
A:\21\kdavis.mar15'	□	🟥			
A:\21\kdavis.mar15'	□	🟦			
A:\21\kdavis.mar15'	□	🟧			
A:\21\kdavis.mar15'	□	🟪			
A:\21\kdavis.mar15'	□	🟩			
A:\21\kdavis.mar15'	□	🟪			
A:\21\kdavis.mar15'	□	🟥			
A:\21\kdavis.mar15'	□	🟩			

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 section of the example. To create each function, in 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 1-0 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 1-0 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 1-0 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 1-0 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.

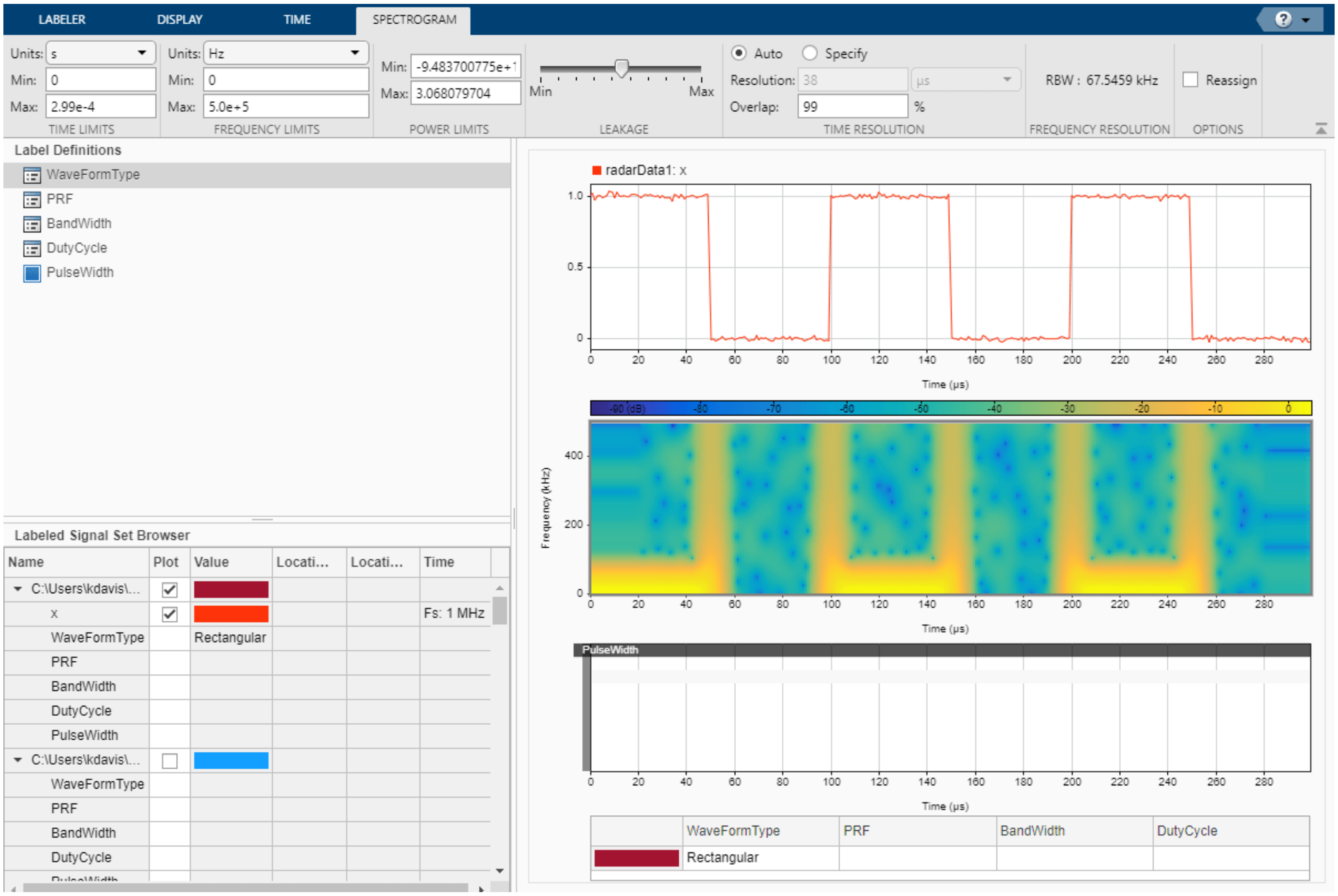


Label Waveform Type, PRF, Bandwidth, Duty Cycle, and Pulsewidth

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 `Rectangular` waveform follows.



TIME LIMITS: Units: s; Min: 0; Max: 2.99e-4

FREQUENCY LIMITS: Units: Hz; Min: 0; Max: 5.0e+5

POWER LIMITS: Min: -9.483700775e+1; Max: 3.068079704

LEAKAGE: Min – Max

TIME RESOLUTION: ● Auto ○ Specify; Resolution: 38 μs; Overlap: 99 %

FREQUENCY RESOLUTION: RBW : 67.5459 kHz

OPTIONS: □ Reassign

Label Definitions

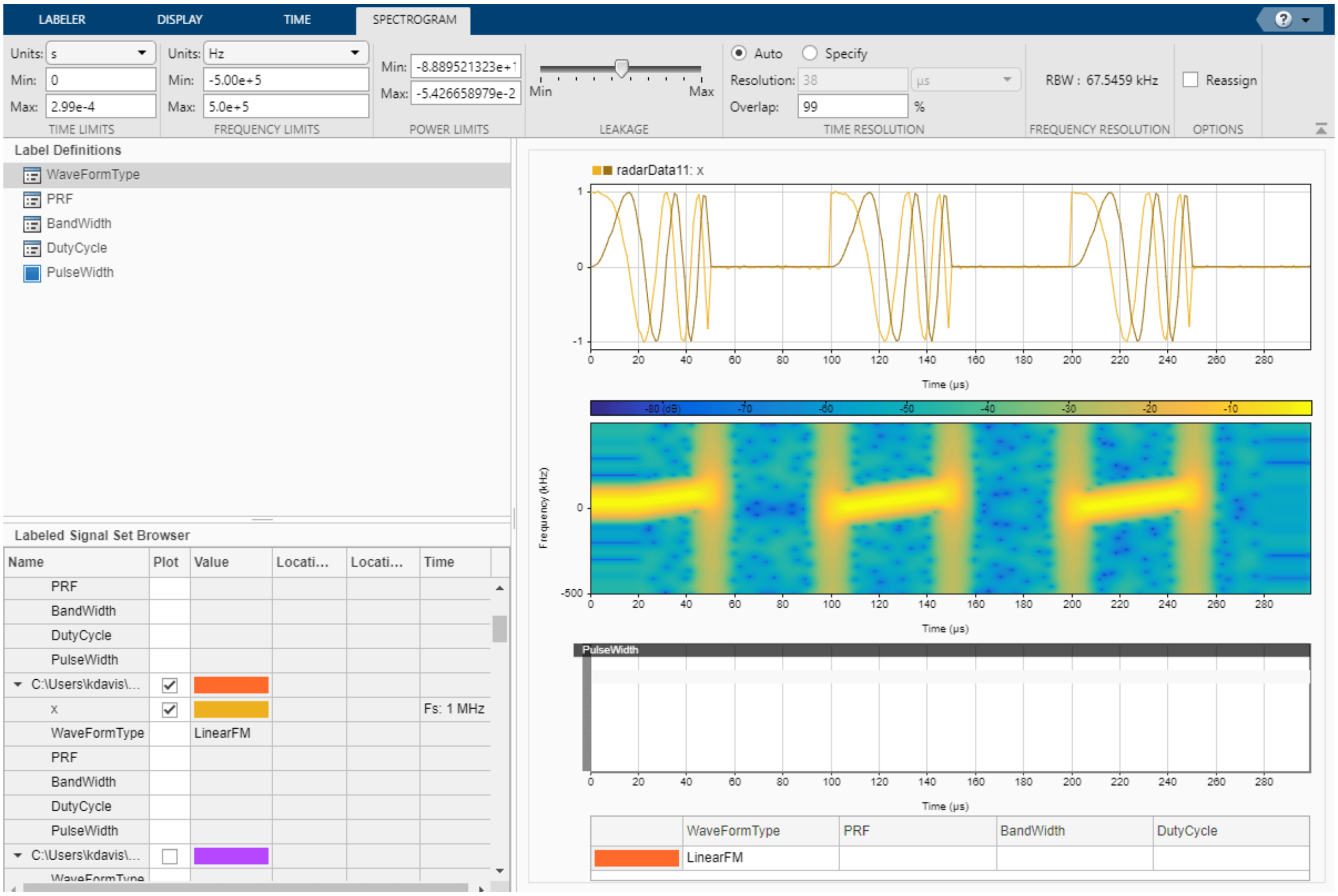
- WaveFormType
- PRF
- BandWidth
- DutyCycle
- PulseWidth

Labeled Signal Set Browser

Name	Plot	Value	Locati...	Locati...	Time
C:\Users\kdavis\...	☑	🟥			
x	☑	🟥			Fs: 1 MHz
WaveFormType		Rectangular			
PRF					
BandWidth					
DutyCycle					
PulseWidth					
C:\Users\kdavis\...	□	🟦			
WaveFormType					
PRF					
BandWidth					
DutyCycle					
PulseWidth					

	WaveFormType	PRF	BandWidth	DutyCycle
🟥	Rectangular			

An example of a `LinearFM` waveform follows.



Units: s
Min: 0
Max: 2.99e-4
TIME LIMITS

Units: Hz
Min: -5.00e+5
Max: 5.0e+5
FREQUENCY LIMITS

Min: -8.889521323e+1
Max: -5.426658979e-2
POWER LIMITS

● Auto ○ Specify
Resolution: 38 μs
Overlap: 99 %
TIME RESOLUTION

RBW : 67.5459 kHz
FREQUENCY RESOLUTION

□ Reassign
OPTIONS

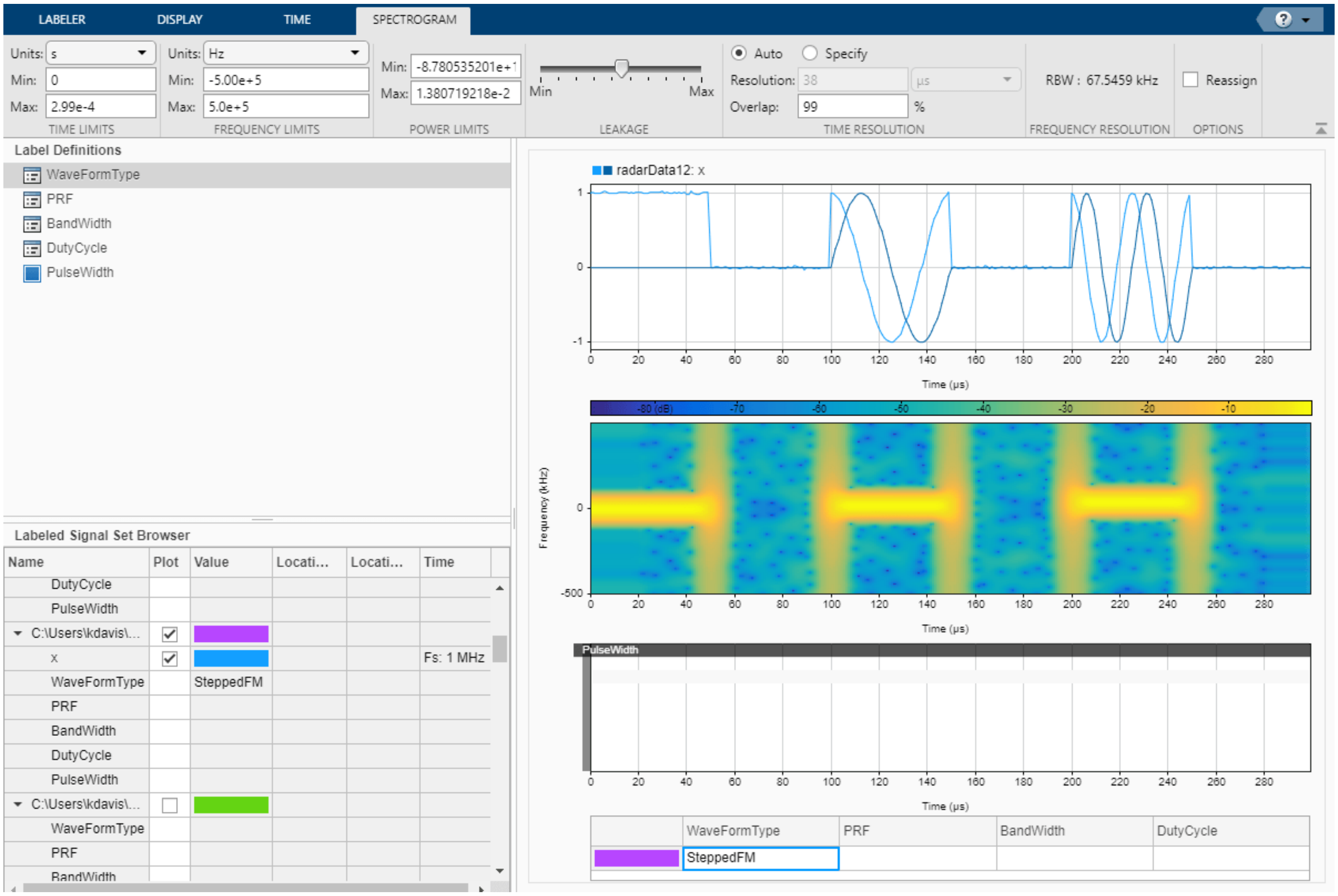
Label Definitions
- WaveFormType
- PRF
- BandWidth
- DutyCycle
- PulseWidth

Labeled Signal Set Browser

Name	Plot	Value	Locati...	Locati...	Time
PRF					
BandWidth					
DutyCycle					
PulseWidth					
C:\Users\kdavis\...	☑	🟧			
x	☑	🟨			Fs: 1 MHz
WaveFormType		LinearFM			
PRF					
BandWidth					
DutyCycle					
PulseWidth					
C:\Users\kdavis\...	□	🟪			
WaveFormType					

	WaveFormType	PRF	BandWidth	DutyCycle
🟧	LinearFM			

An example of a `SteppedFM` waveform follows.



TIME LIMITS: Units: s; Min: 0; Max: 2.99e-4

FREQUENCY LIMITS: Units: Hz; Min: -5.00e+5; Max: 5.0e+5

POWER LIMITS: Min: -8.780535201e+1; Max: 1.380719218e-2

TIME RESOLUTION: ● Auto ○ Specify; Resolution: 38 μs; Overlap: 99 %

FREQUENCY RESOLUTION: RBW : 67.5459 kHz

OPTIONS: □ Reassign

Label Definitions

- WaveFormType
- PRF
- BandWidth
- DutyCycle
- PulseWidth

Labeled Signal Set Browser

Name	Plot	Value	Locati...	Locati...	Time
DutyCycle					
PulseWidth					
C:\Users\kdavis\...	☑	🟪			
x	☑	🟦			Fs: 1 MHz
WaveFormType		SteppedFM			
PRF					
BandWidth					
DutyCycle					
PulseWidth					
C:\Users\kdavis\...	□	🟩			
WaveFormType					
PRF					
BandWidth					

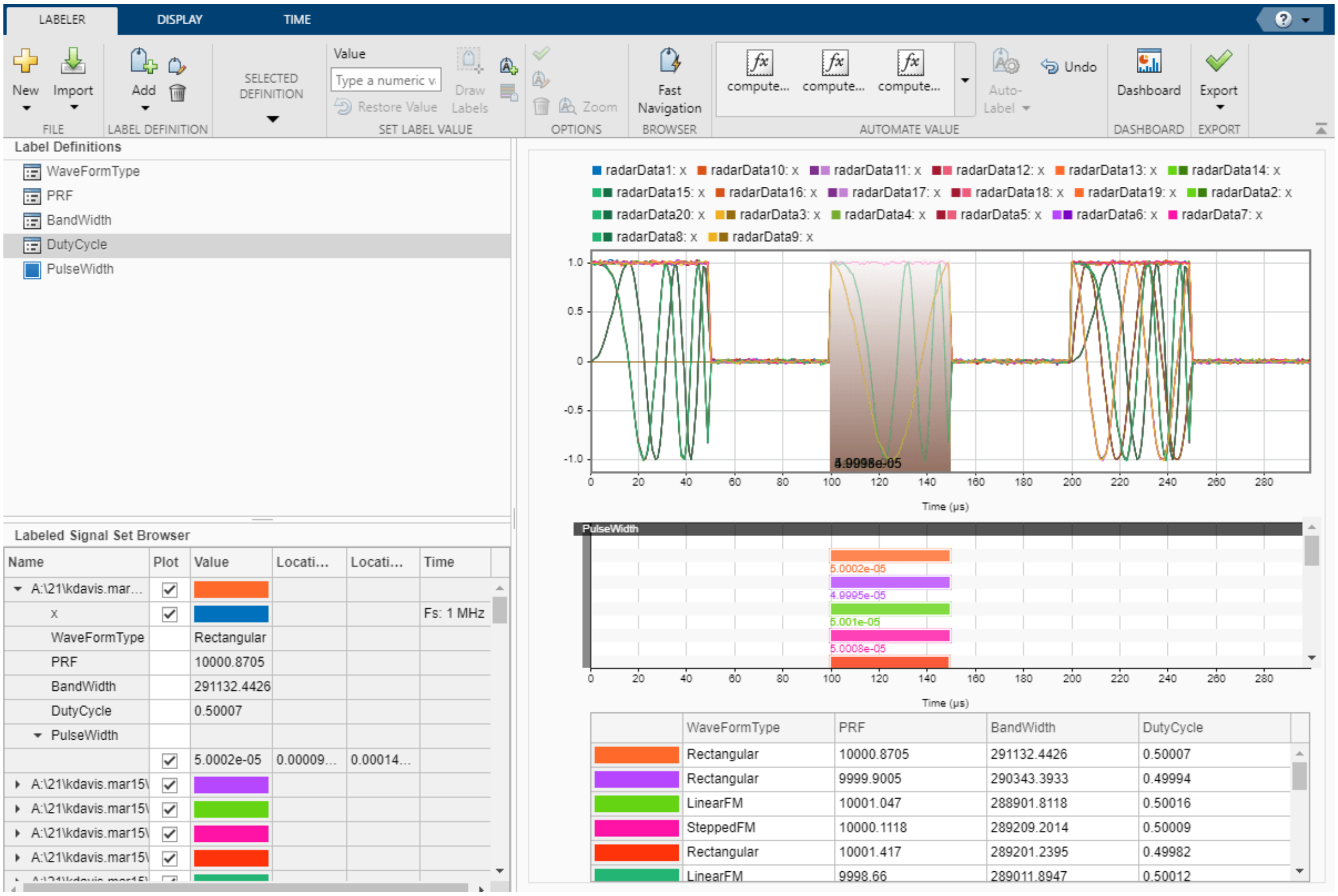
	WaveFormType	PRF	BandWidth	DutyCycle
🟪	SteppedFM			

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.



Labeled Signal Set Browser

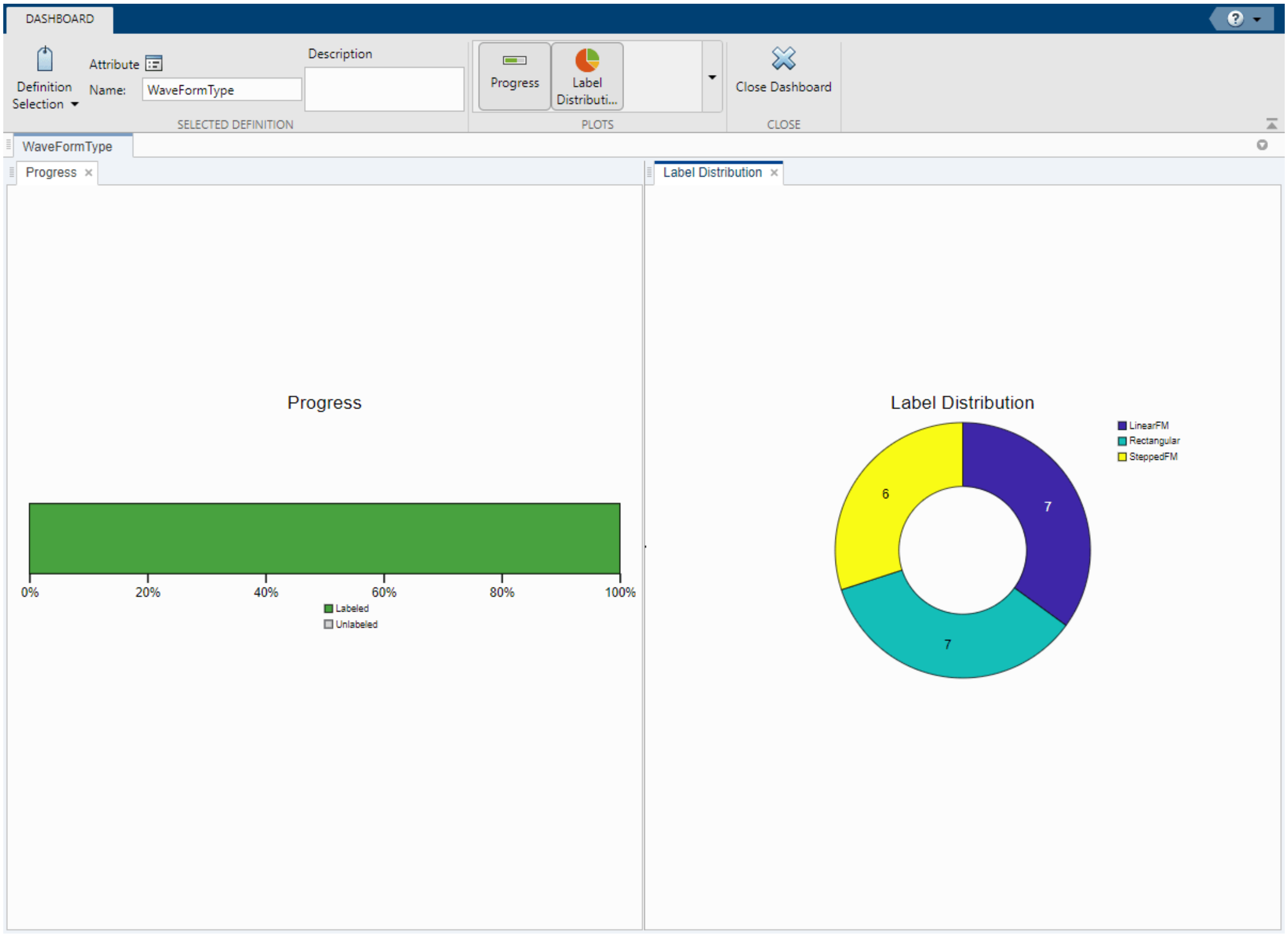
Name	Plot	Value	Locati...	Locati...	Time
A:\21\kdavis.mar...	☑	🟧			
x	☑	🟦			Fs: 1 MHz
WaveFormType		Rectangular			
PRF		10000.8705			
BandWidth		291132.4426			
DutyCycle		0.50007			
PulseWidth					
	☑	5.0002e-05	0.00009...	0.00014...	
A:\21\kdavis.mar15\	☑	🟪			
A:\21\kdavis.mar15\	☑	🟩			
A:\21\kdavis.mar15\	☑	🟪			
A:\21\kdavis.mar15\	☑	🟥			

	WaveFormType	PRF	BandWidth	DutyCycle
🟧	Rectangular	10000.8705	291132.4426	0.50007
🟪	Rectangular	9999.9005	290343.3933	0.49994
🟩	LinearFM	10001.047	288901.8118	0.50016
🟪	SteppedFM	10000.1118	289209.2014	0.50009
🟥	Rectangular	10001.417	289201.2395	0.49982
🟩	LinearFM	9998.66	289011.8947	0.50012

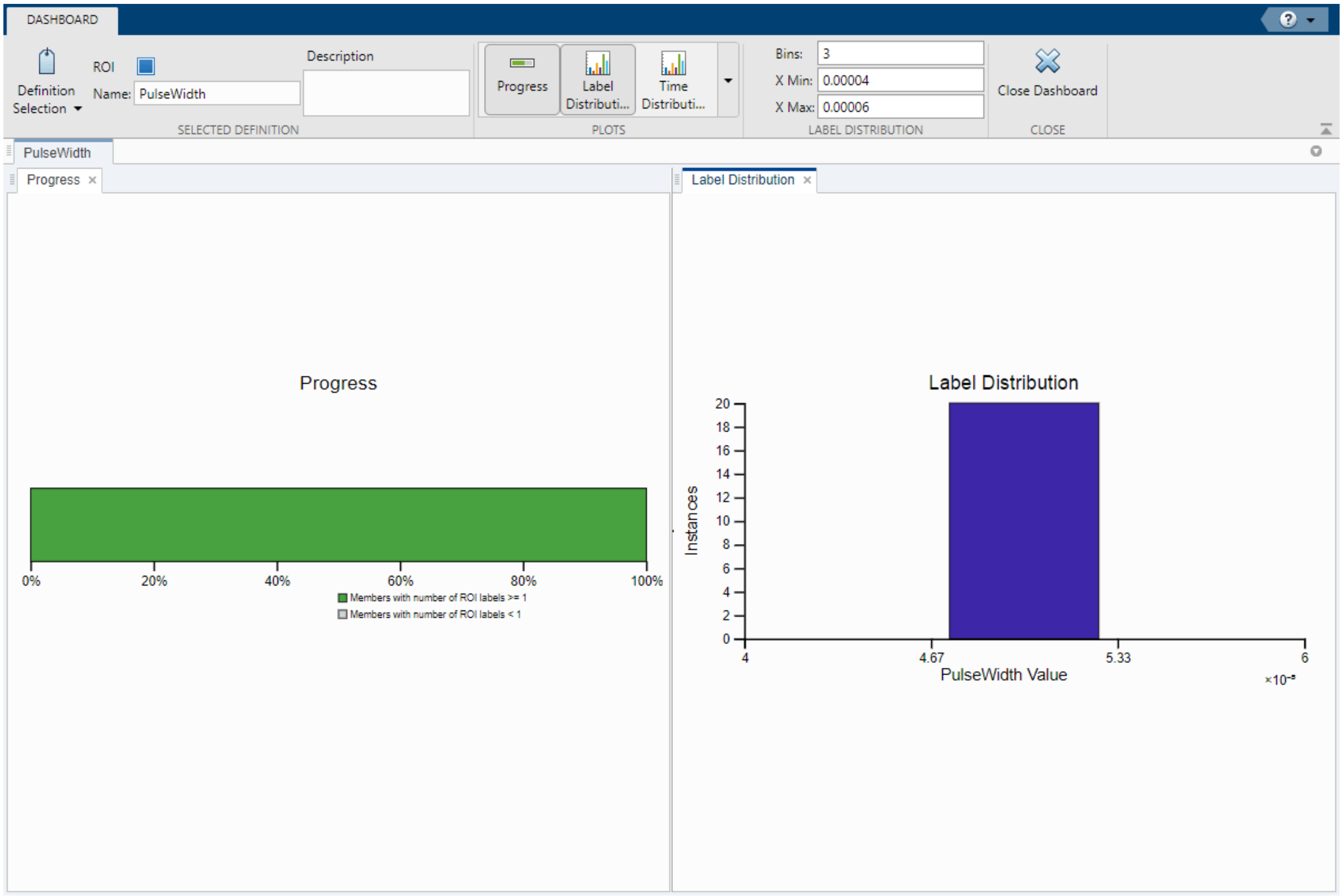
Validate Labeled Signals

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 pulsewidth label values are distributed around 5e-5. To look at the time distribution of the pulsewidth 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 pulsewidth distributed around 5e-5.



Name: PulseWidth

Bins: 3

X Min: 0.00004

X Max: 0.00006

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.

```
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.

```
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.

```
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.

```
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);
```

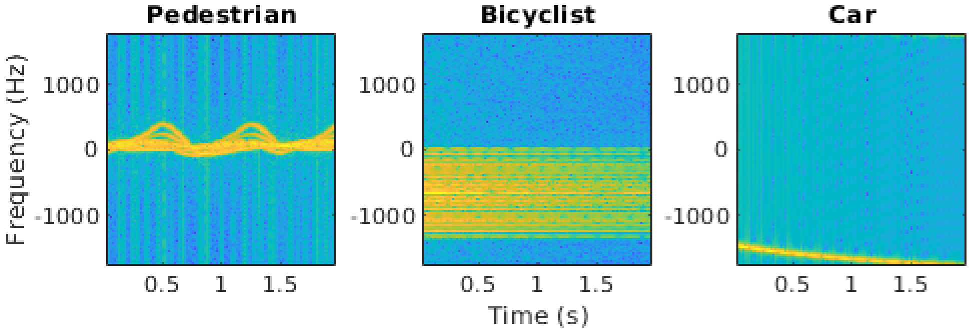
The helper function `helperDopplerSignatures` computes the short-time Fourier transform (STFT) of a radar return to generate the micro-Doppler signature. To obtain the micro-Doppler signatures, use the helper functions to apply the STFT and a preprocessing method to each signal.

```
[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

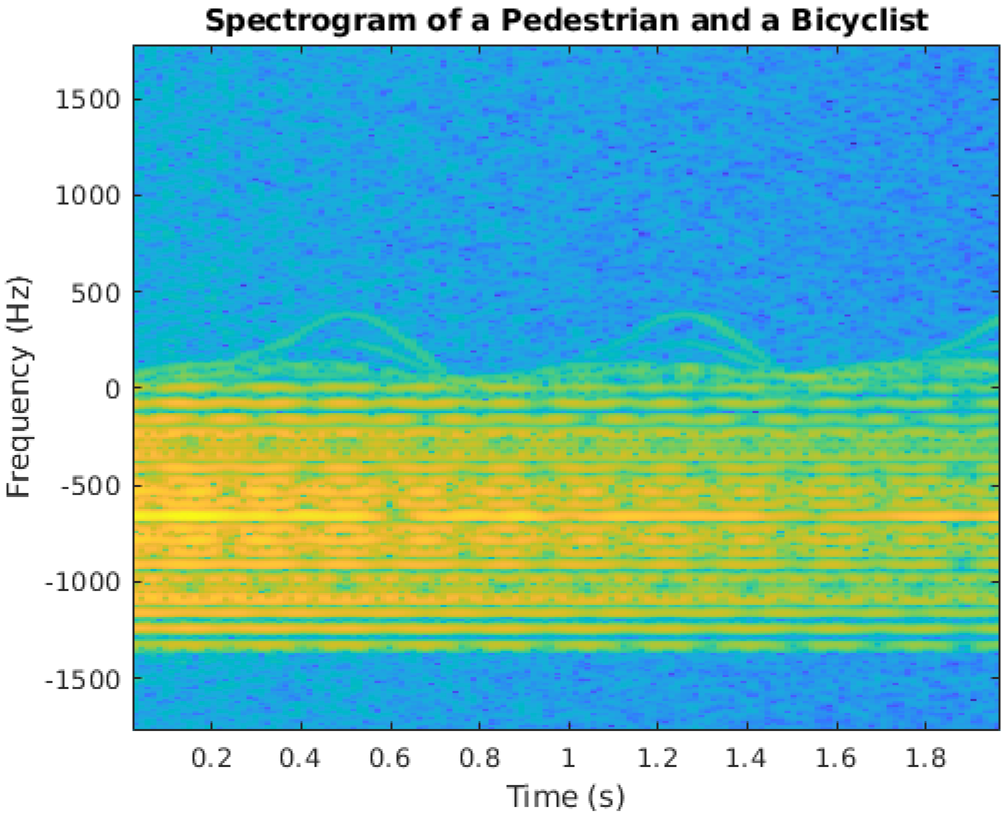
Classifying a single realization as a pedestrian or bicyclist is relatively simple because the pedestrian and bicyclist micro-Doppler signatures are dissimilar. However, classifying multiple overlapping pedestrians or bicyclists, with the addition of Gaussian noise or car noise, is much more difficult.

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.

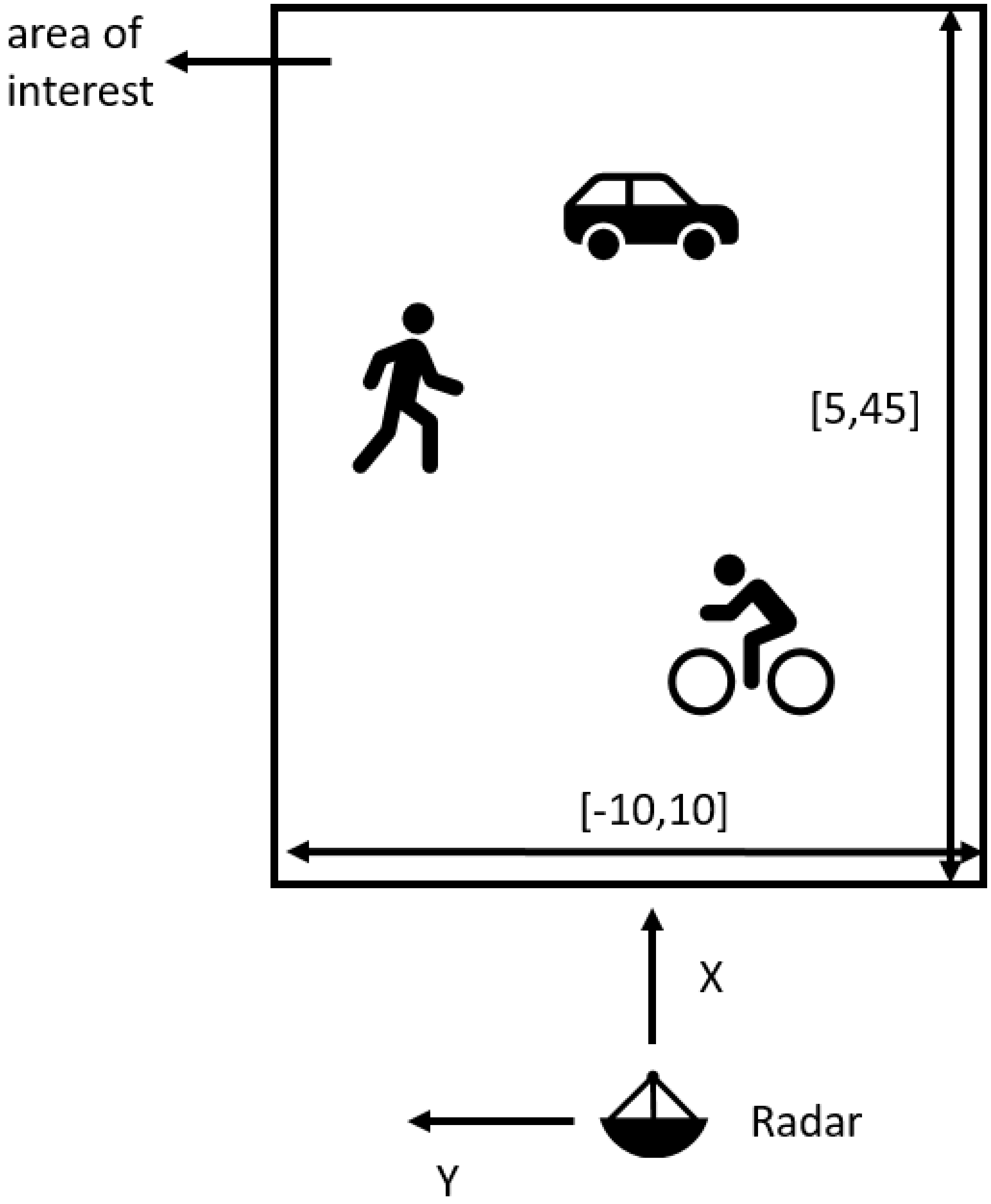
```
[S,~,~] = helperDopplerSignatures(xRadarRec,Tsamp);

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, 1.4*h*] meters/second, where *h* is the height value

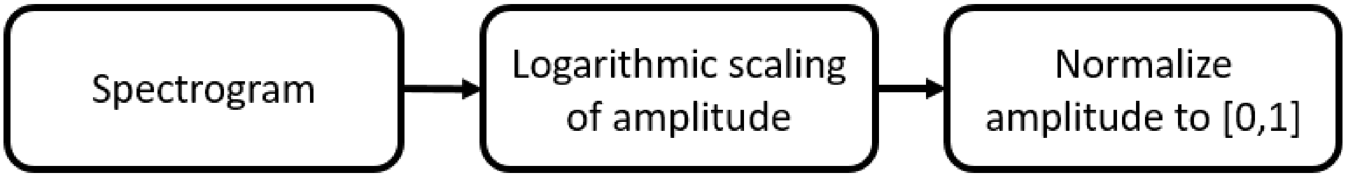
2) Bicyclists

- 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:


Spectrogram
Logarithmic scaling of amplitude
Normalize amplitude to [0,1]

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.

```
% 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).

```
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
```

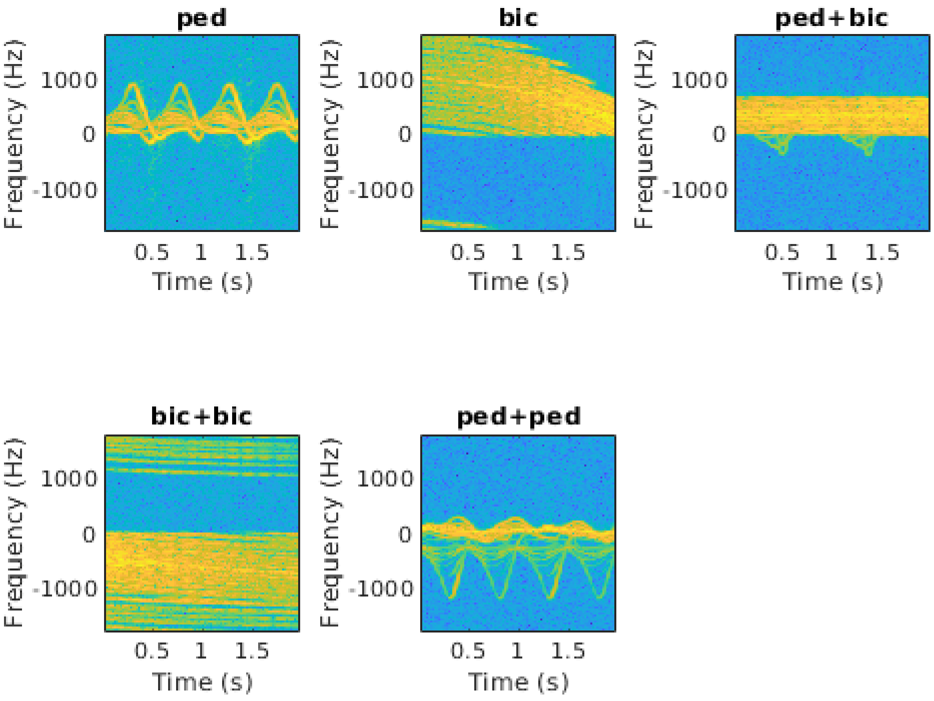
Specify the optimization solver and the hyperparameters to train the CNN using `trainingOptions`. This example uses the ADAM optimizer and a mini-batch size of 128. Train the network using either a CPU or GPU. Using a GPU requires Parallel Computing Toolbox™. To see which GPUs are supported, see “GPU Support by Release” (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

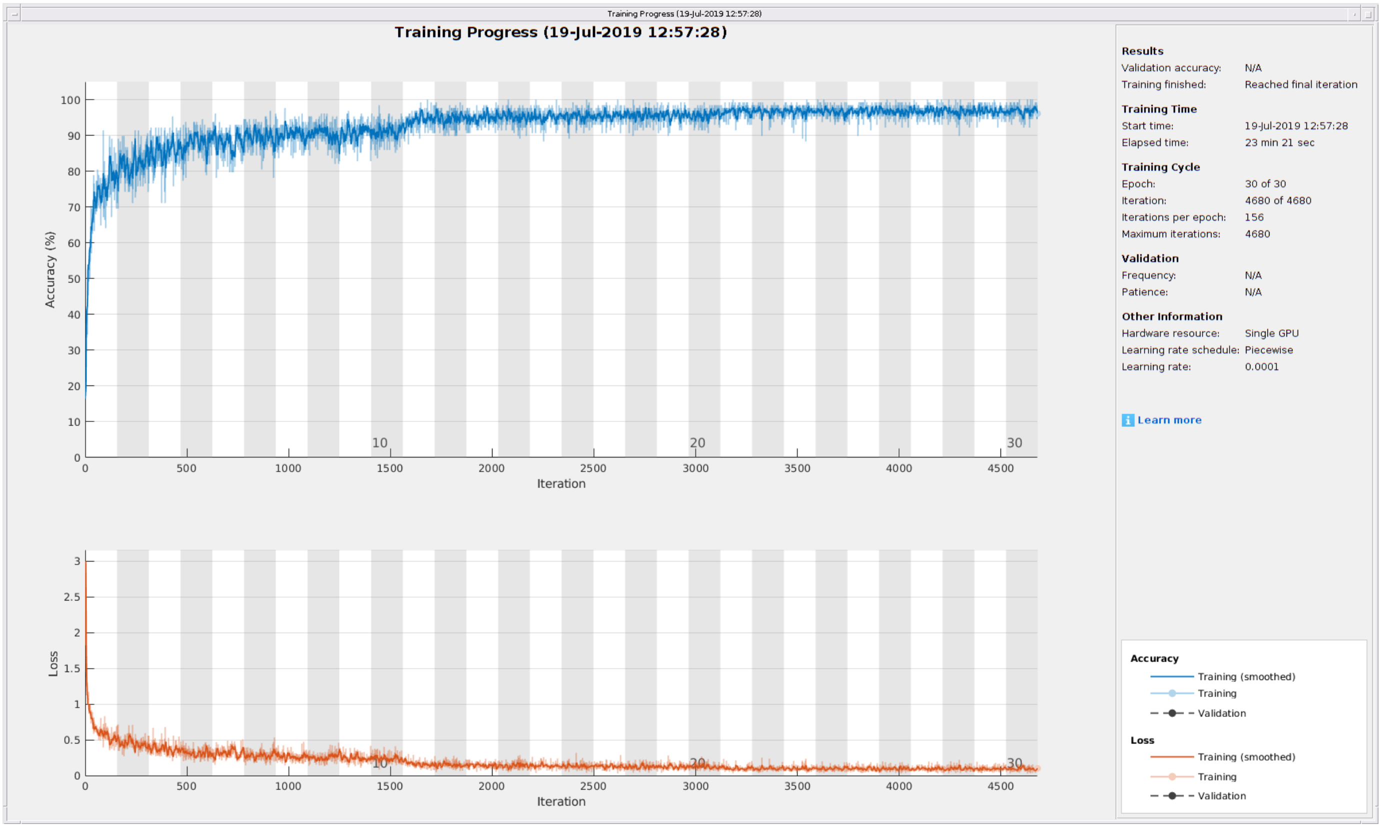
Load the data set without car noise and use the helper function `helperPlotTrainData` to plot one example of each of the five categories in the training data set,

```
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);
```



Training Progress (19-Jul-2019 12:57:28)

Results
Validation accuracy: N/A
Training finished: Reached final iteration

Training Time
Start time: 19-Jul-2019 12:57:28
Elapsed time: 23 min 21 sec

Training Cycle
Epoch: 30 of 30
Iteration: 4680 of 4680
Iterations per epoch: 156
Maximum iterations: 4680

Validation
Frequency: N/A
Patience: N/A

Other Information
Hardware resource: Single GPU
Learning rate schedule: Piecewise
Learning rate: 0.0001

Accuracy
Training (smoothed)
Training
Validation

Loss
Training (smoothed)
Training
Validation

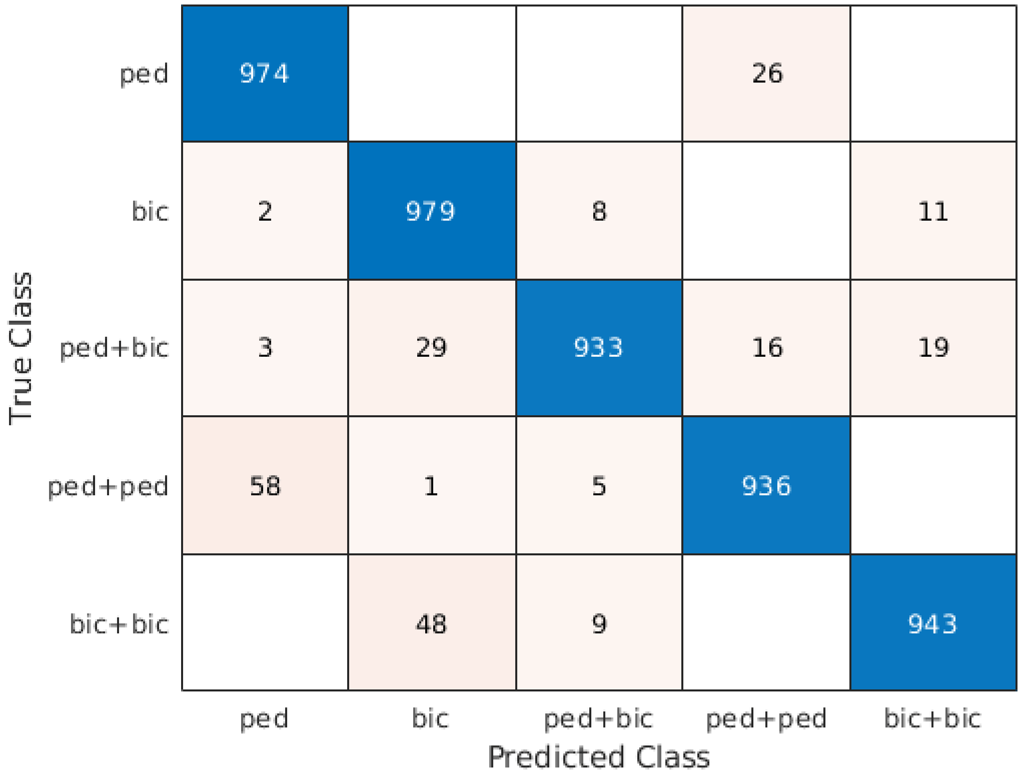
Use the trained network and the `classify` function to obtain the predicted labels for the test data set `testDataNoCar`. The variable `predTestLabel` contains the network predictions. The network achieves about 95% accuracy for the test data set without the car noise.

```
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'))
```

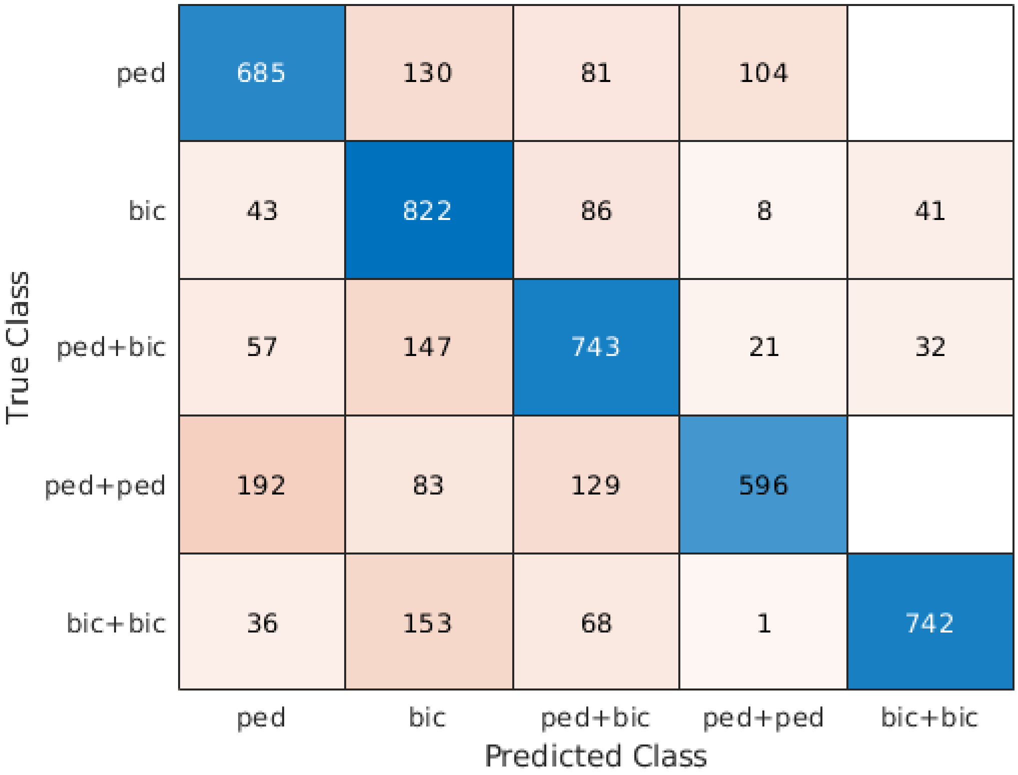
Input the car-noise-corrupted test data set to the network. The prediction accuracy for the test data set with the car noise drops significantly, to around 70%, because the network never saw training samples containing car noise.

```
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);
```



True Class \ Predicted Class	ped	bic	ped+bic	ped+ped	bic+bic
ped	685	130	81	104	
bic	43	822	86	8	41
ped+bic	57	147	743	21	32
ped+ped	192	83	129	596	
bic+bic	36	153	68	1	742

Car noise significantly impedes the performance of the classifier. To solve this problem, train the CNN using data that contains car noise.

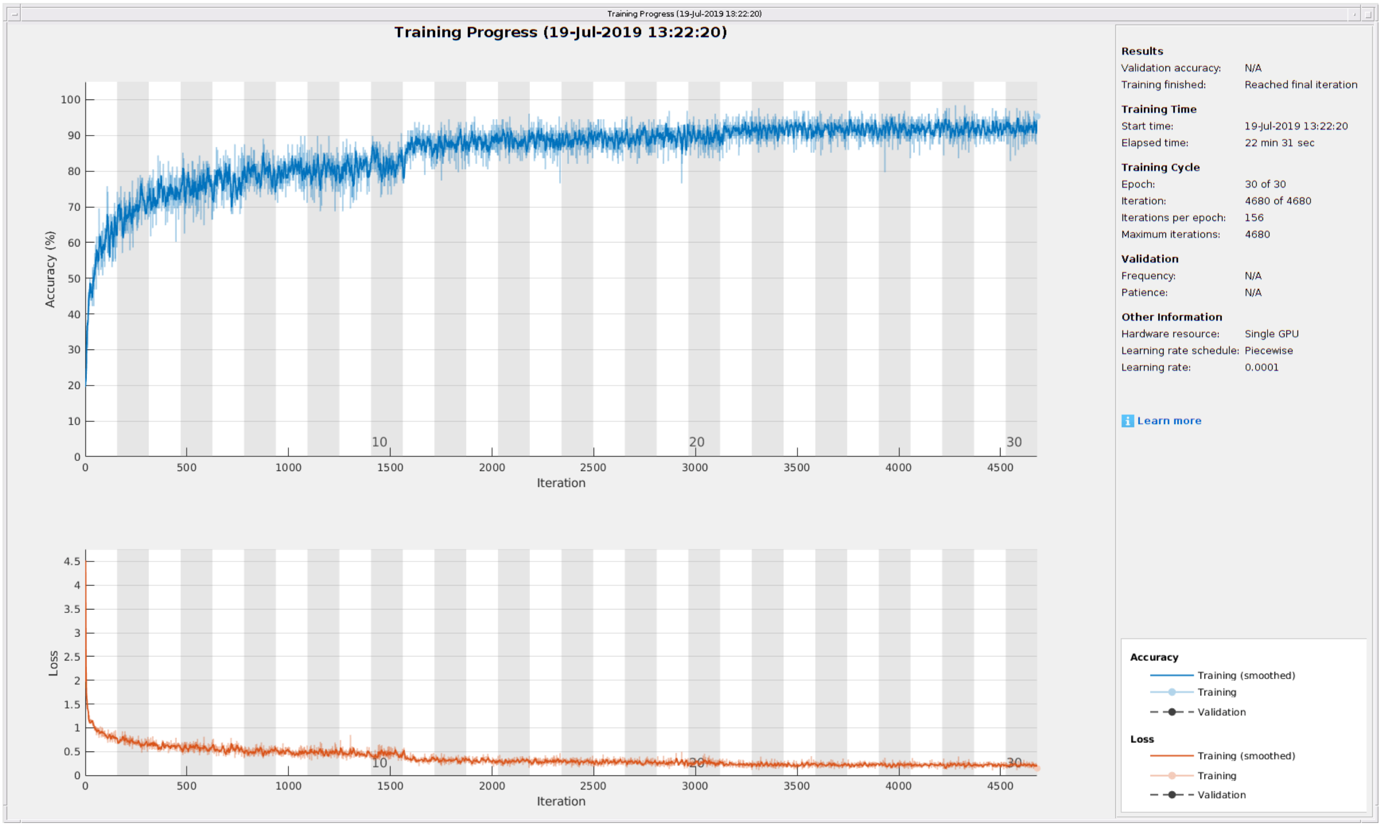
Retrain CNN by Adding Car Noise to Training Data Set

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);
```



Results
Validation accuracy: N/A
Training finished: Reached final iteration

Training Time
Start time: 19-Jul-2019 13:22:20
Elapsed time: 22 min 31 sec

Training Cycle
Epoch: 30 of 30
Iteration: 4680 of 4680
Iterations per epoch: 156
Maximum iterations: 4680

Validation
Frequency: N/A
Patience: N/A

Other Information
Hardware resource: Single GPU
Learning rate schedule: Piecewise
Learning rate: 0.0001

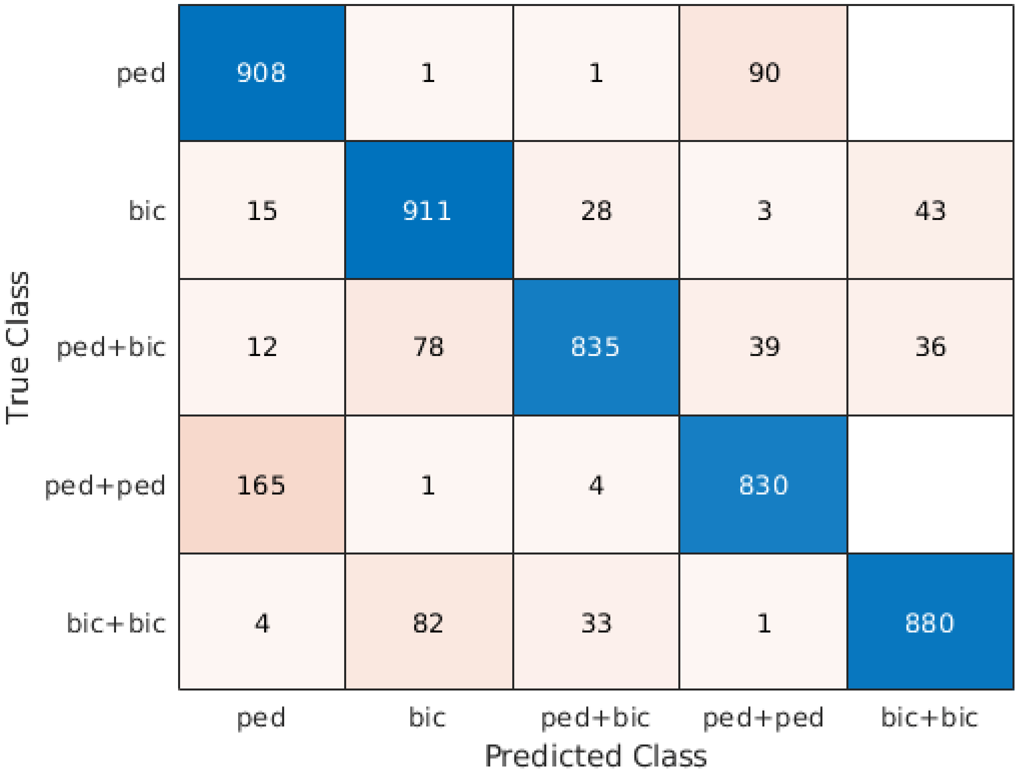
Input the car-noise-corrupted test data set to the network `trainedNetCarNoise`. The prediction accuracy is about 87%, which is approximately 15% higher than the performance of the network trained without car noise samples.

```
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);
```

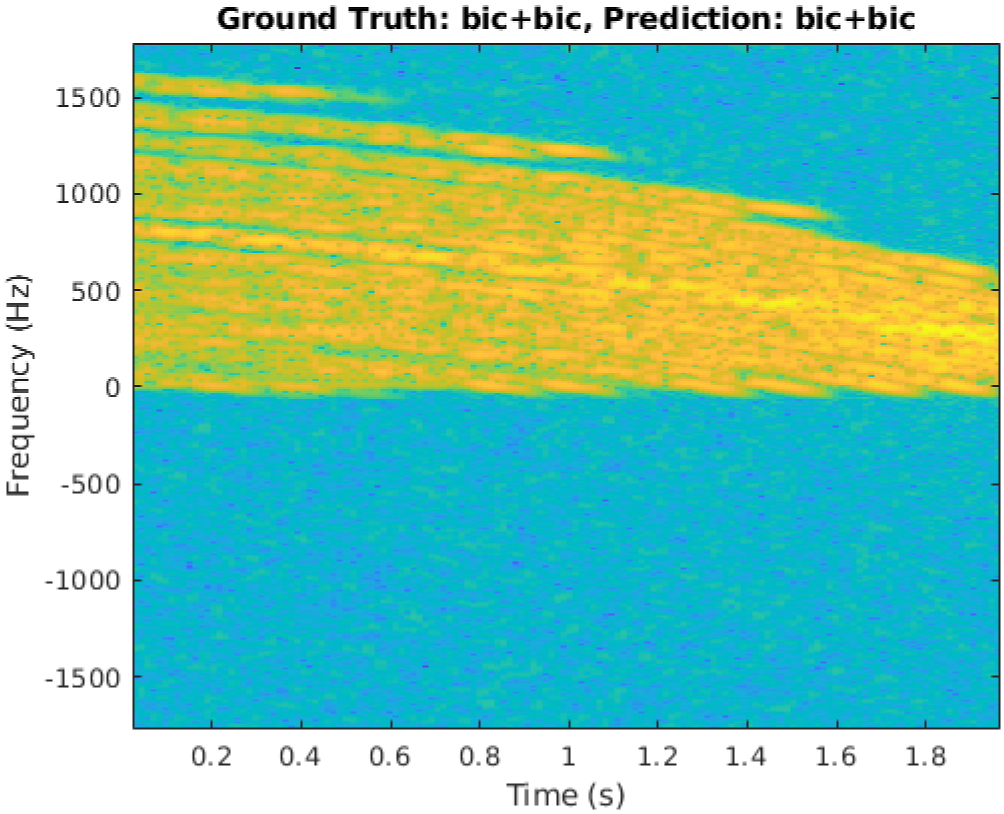


Case Study

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.

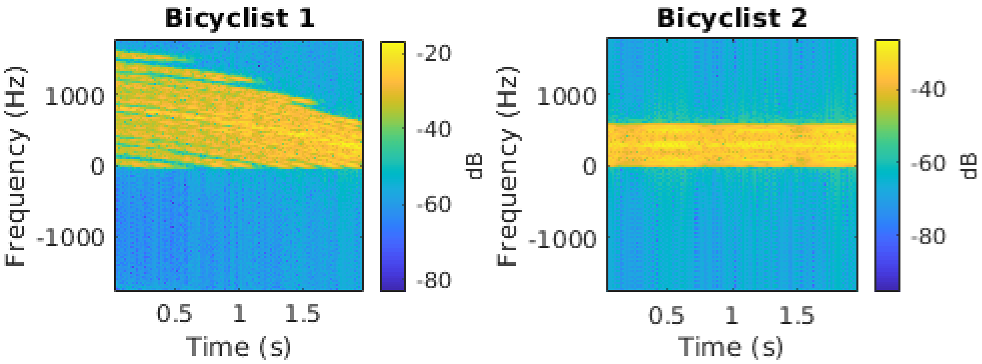
For example, signature #4 of the car-noise-corrupted test data, which does not have car noise, has two bicyclists with overlapping micro-Doppler signatures. The network correctly predicts that the scene has two bicyclists.

```
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)))
```



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.

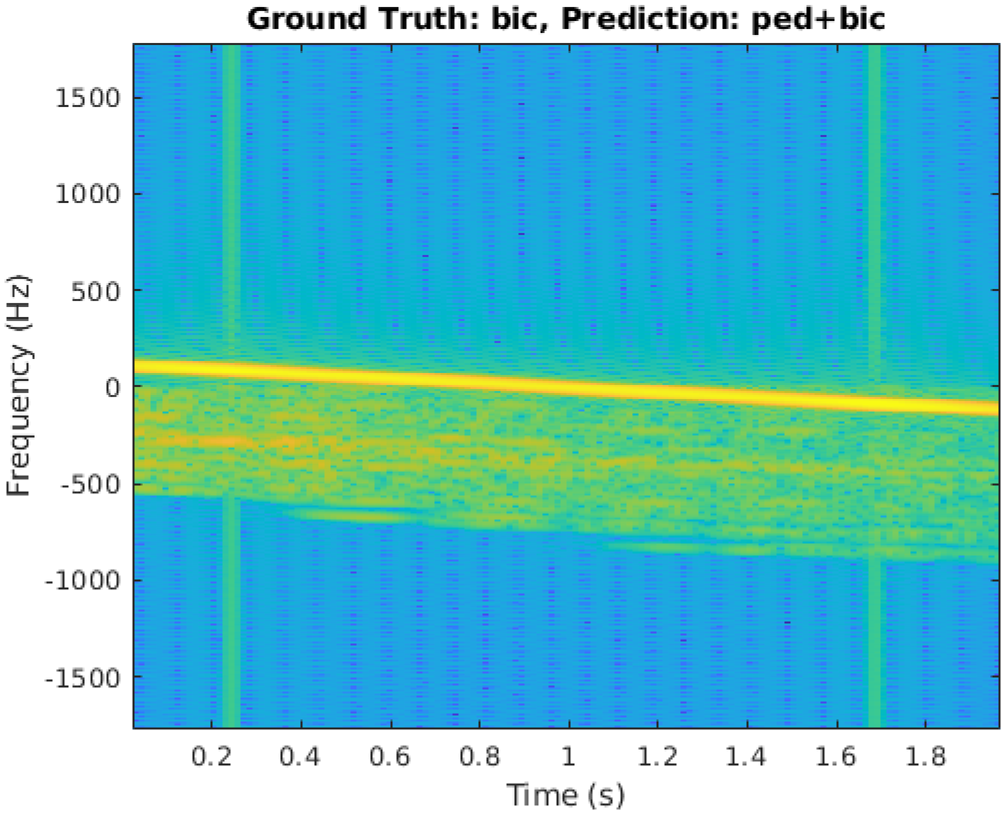
```
load CaseStudyData.mat
M = 200; % FFT window length
beta = 6; % window parameter
w = kaiser(M,beta); % kaiser window
R = floor(1.7*(M-1)/(beta+1)); % ROUGH estimate
noverlap = M-R; % overlap length

[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)))
```



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

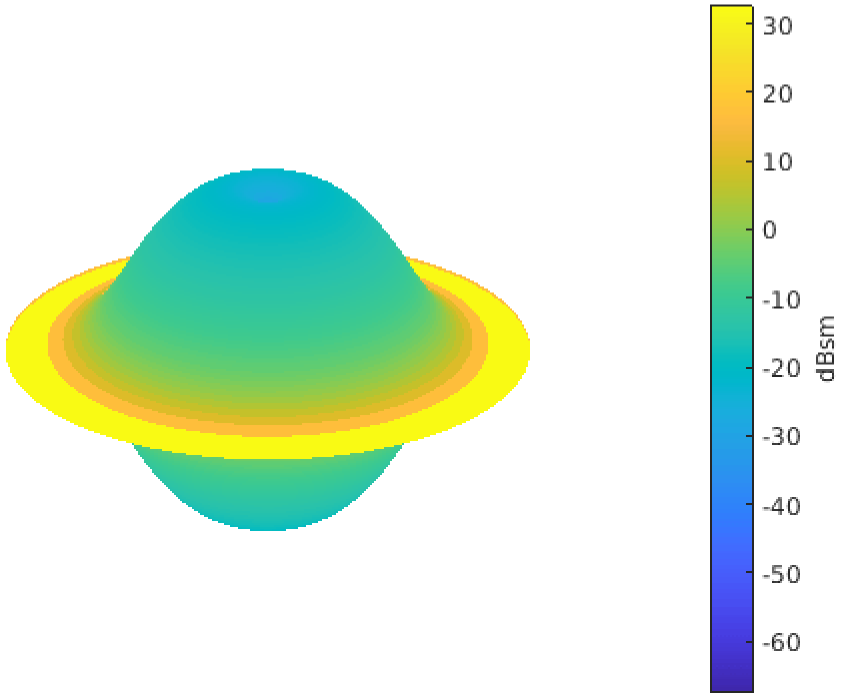
Target classification is an important function in modern radar systems. This example uses machine and deep learning to classify radar echoes from a cylinder and a cone. Although this example uses the synthesized I/Q samples, the workflow is applicable to real radar returns.

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);
```

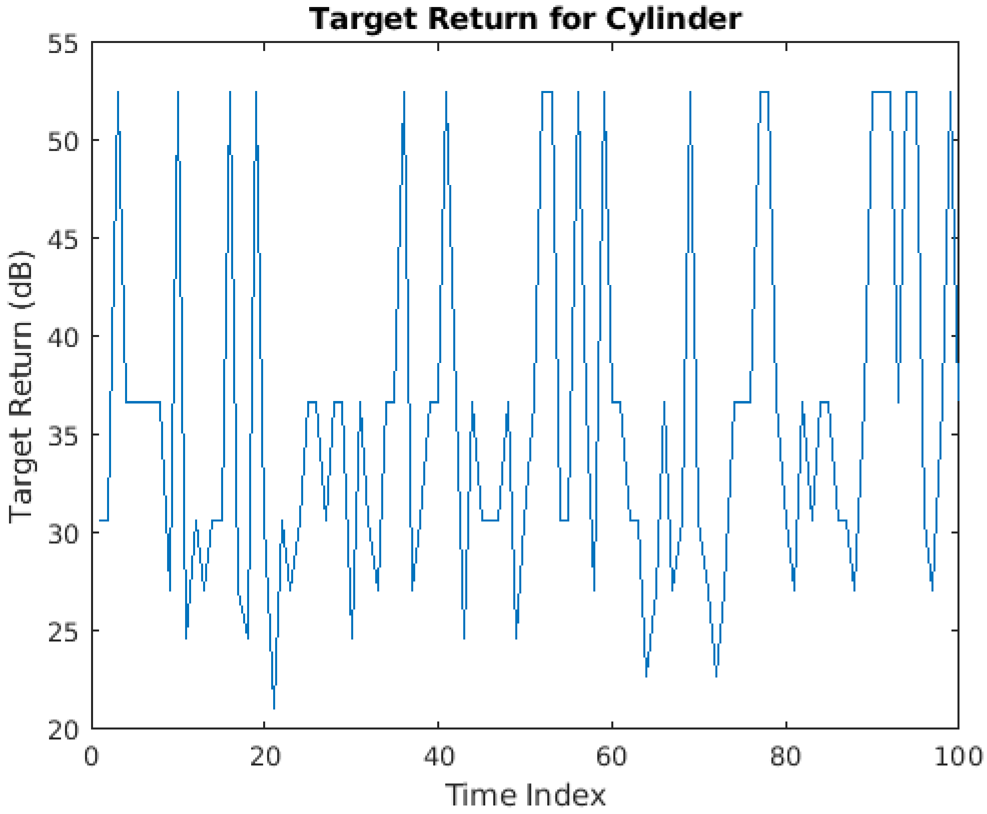


The pattern can then be applied to a backscatter radar target to simulate returns from different aspects angles.

```
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');
```



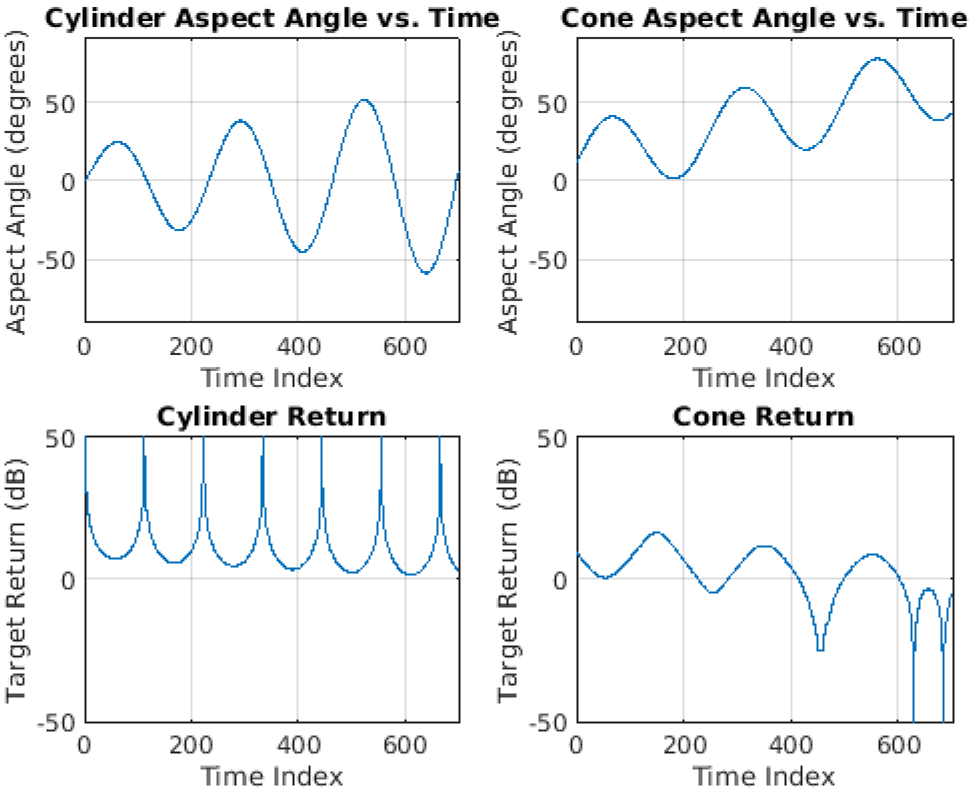
The return of the cone can be generated similarly. To create the training set, the above process is repeated for 5 arbitrarily selected cylinder radii. In addition, for each radius, 10 motion profiles are simulated by varying the incident angle following 10 randomly generated sinusoid curve around boresight. 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

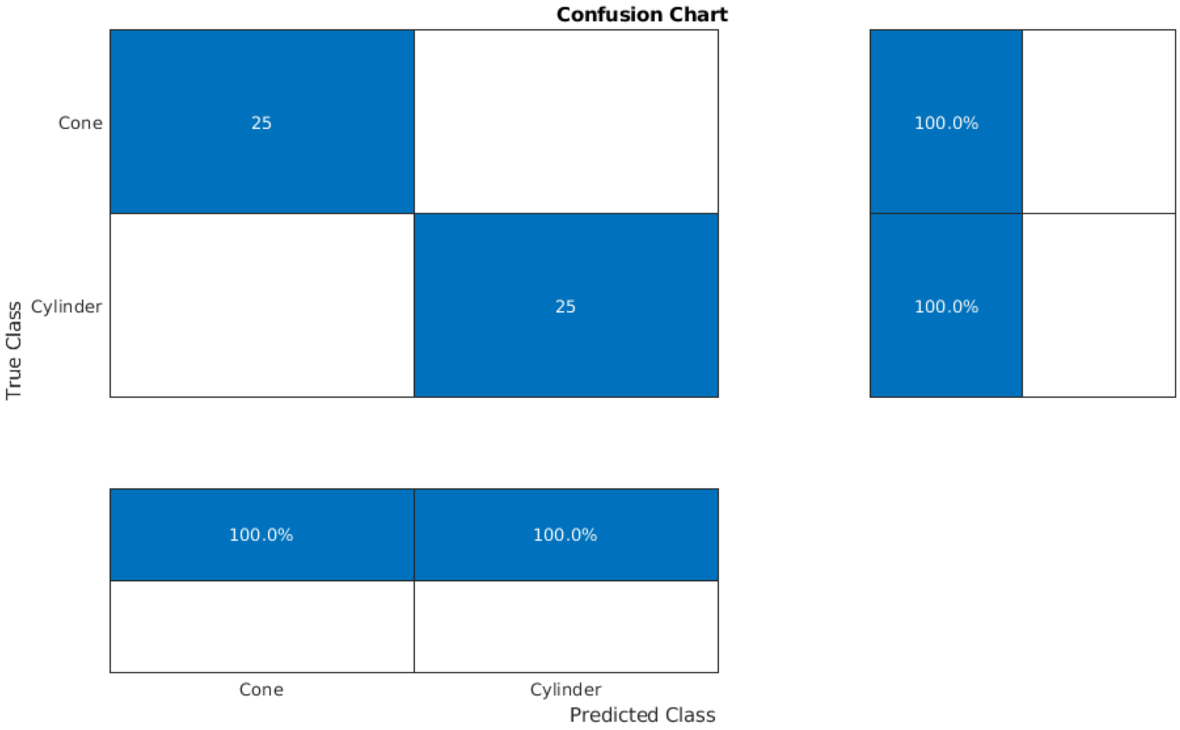
Using the trained SVM, classify the scattering features obtained from the test set.

```
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';
```



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.

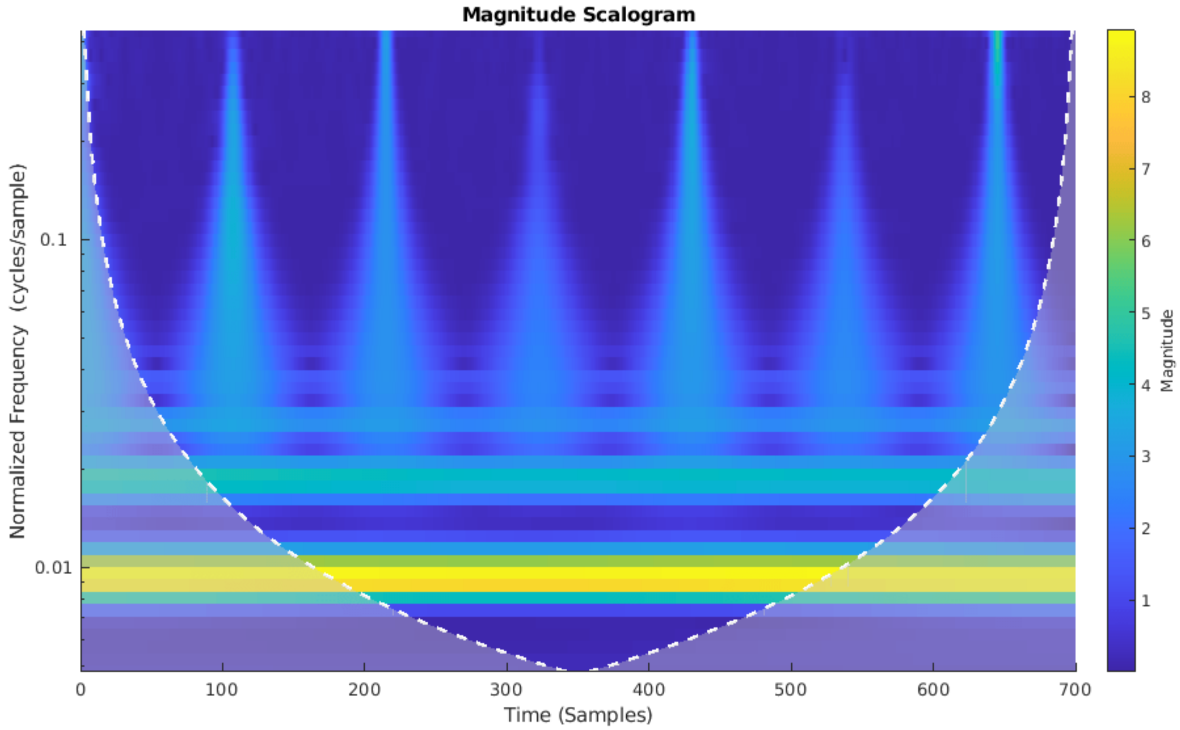
Continuous Wavelet Transform

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;
```

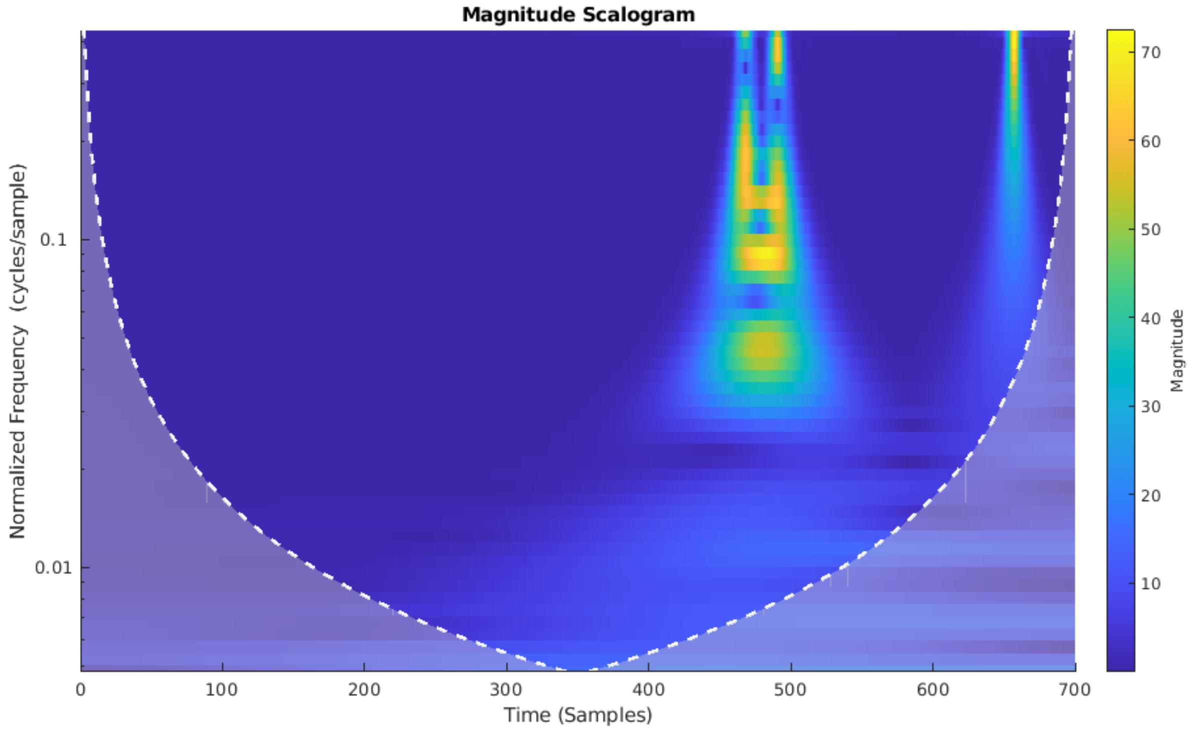
It is evident that the radar returns previously shown are characterized by slowing varying changes punctuated by large transient decreases as described earlier. A wavelet transform is ideally suited to sparsely representing such signals. Wavelets shrink to localize transient phenomena with high temporal 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  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      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 |

|========================================================================================|
```

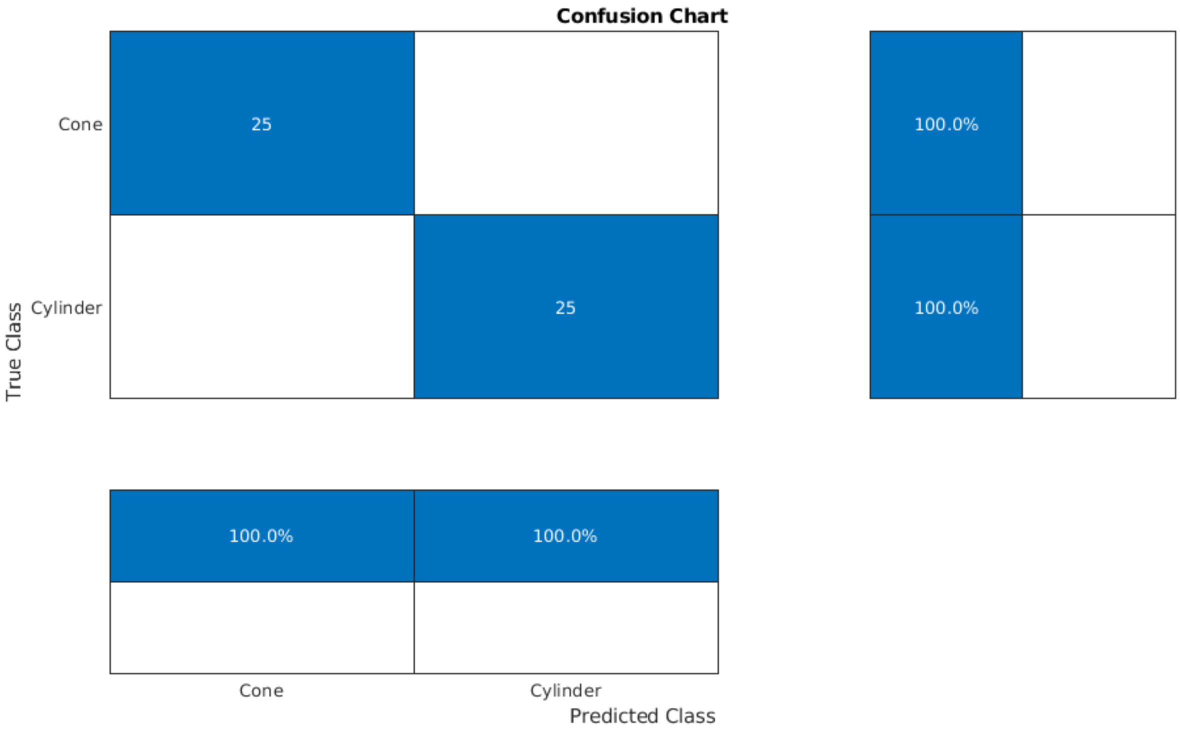
Use the trained network to predict target returns in the held-out test set.

```
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.

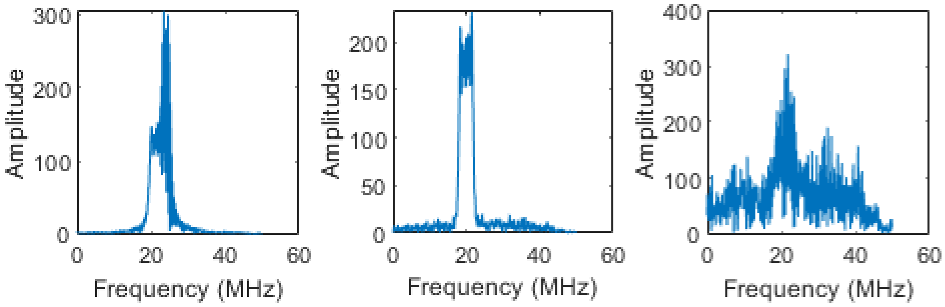
Each signal has unique parameters and is augmented with various impairments to make it more realistic. For each waveform, the pulse width and repetition frequency will be randomly generated. For LFM waveforms, the sweep bandwidth and direction are randomly generated. For Barker waveforms, the chip width 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 provided 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.

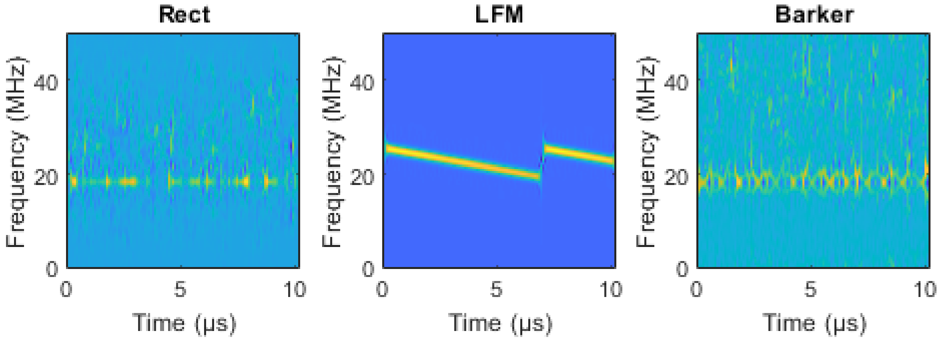
```
idLFM = find(modType == "LFM",3);
nfft = 2^nextpow2(length(wav{1}));
f = (0:(nfft/2-1))/nfft*100e6;

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 each signal's modulation type 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);
```

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` will transform 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 our needs, 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 = 
  6x1 Layer array with layers:

     1   'drop9'                             Dropout                       50% dropout
     2   'conv10'                            Convolution                   1000 1×1×512 convoluti
     3   'relu_conv10'                       ReLU                          ReLU
     4   'pool10'                            2-D Global Average Pooling    2-D global average poo
     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
```

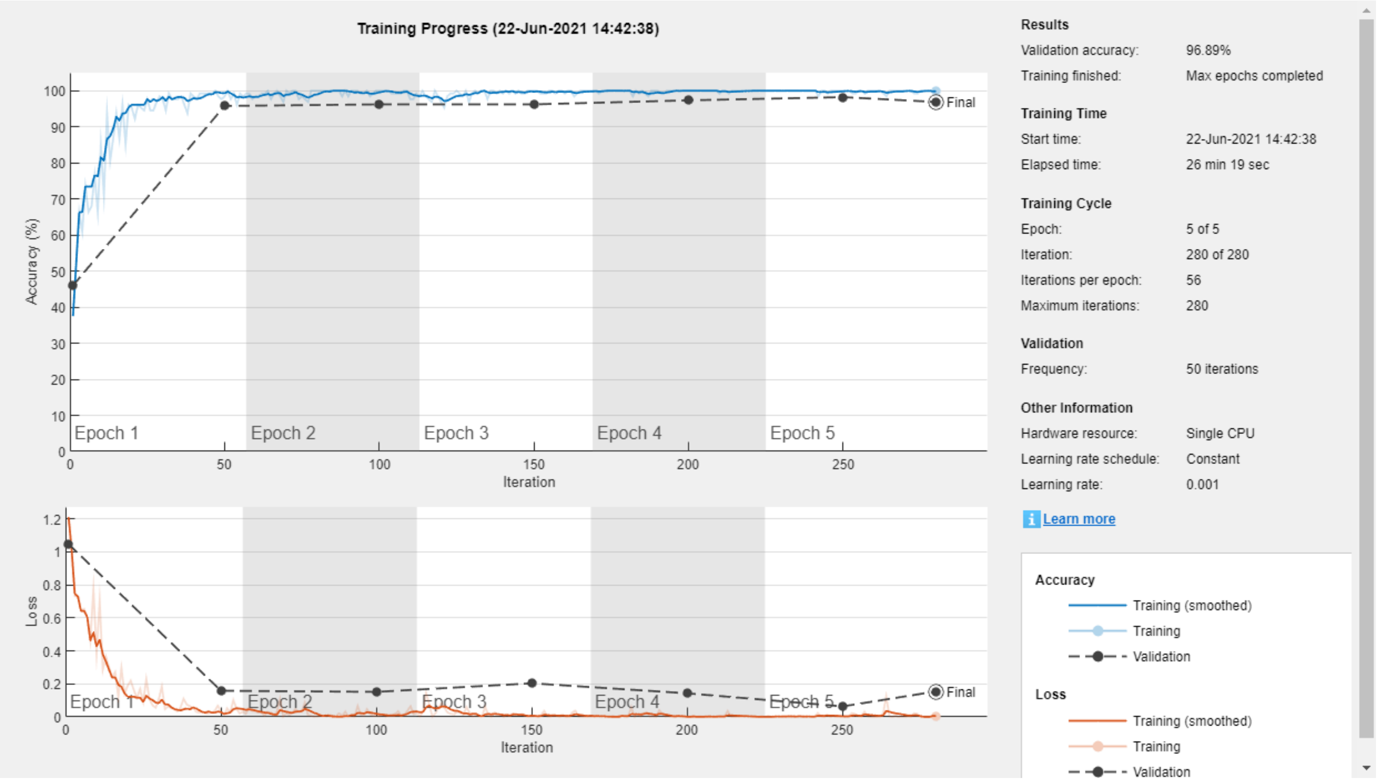
Choose options for the training process that ensures good network performance. Refer to the `trainingOptions` documentation for a description of each option.

```
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 of the dataset's large size, the process may 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);
```



Results

Validation accuracy: 96.89%

Training finished: Max epochs completed

Training Time

Start time: 22-Jun-2021 14:42:38

Elapsed time: 26 min 19 sec

Training Cycle

Epoch: 5 of 5

Iteration: 280 of 280

Iterations per epoch: 56

Maximum iterations: 280

Validation

Frequency: 50 iterations

Other Information

Hardware resource: Single CPU

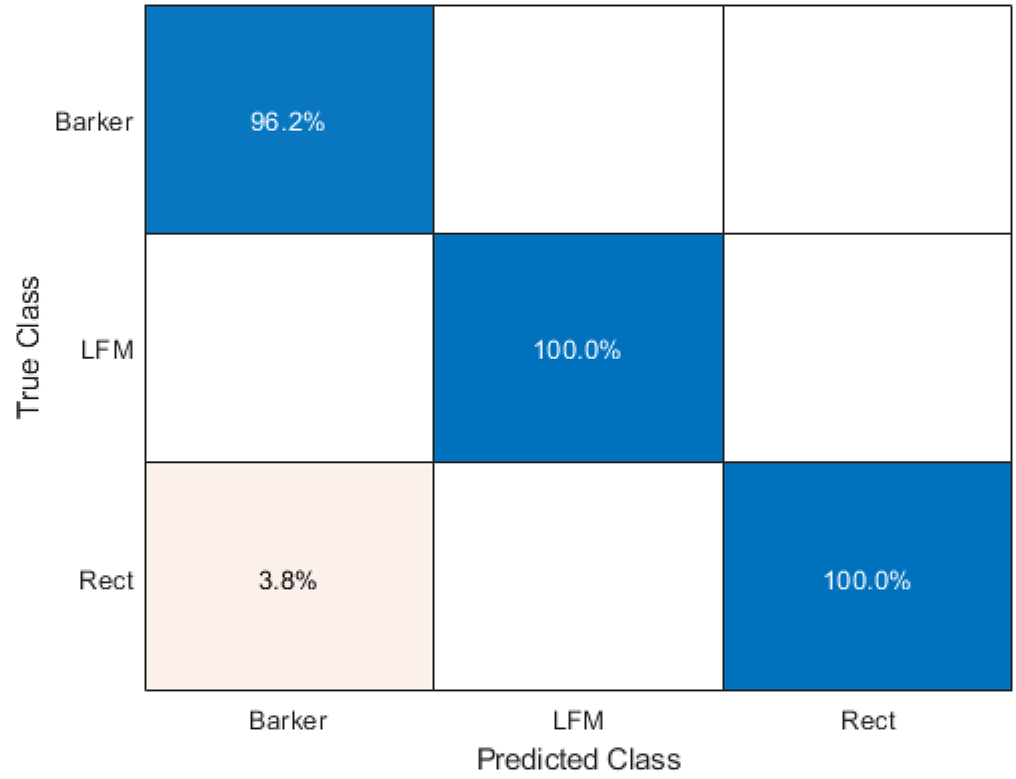
Learning rate schedule: Constant

Learning rate: 0.001

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')
```



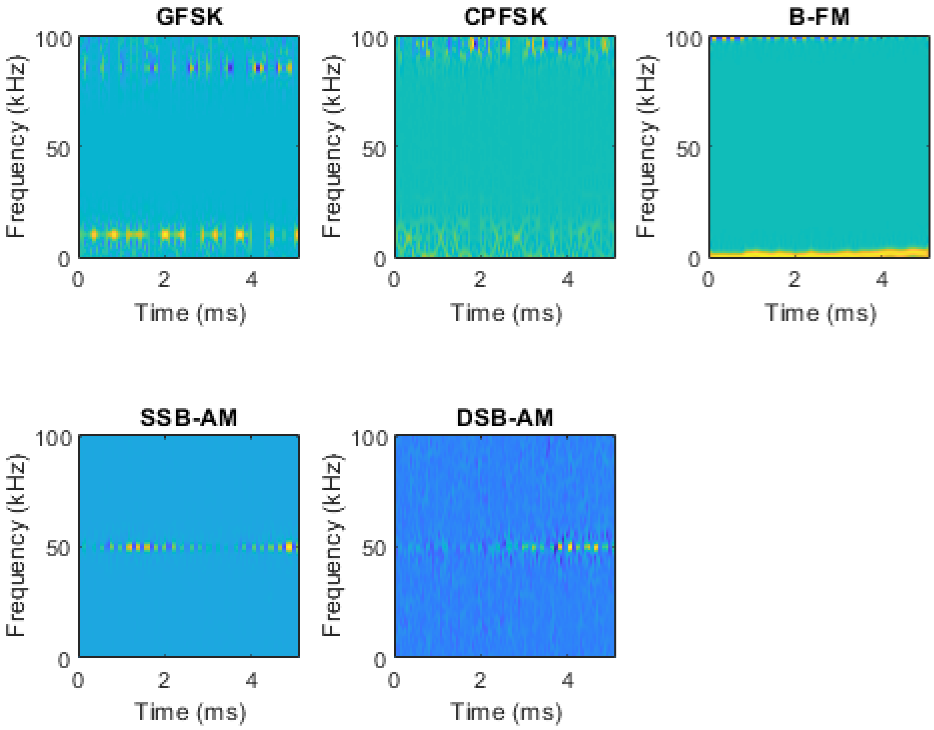
Generate Communications Waveforms and Extract Features

The frequency spectrum of a radar classification system must compete with other transmitted sources. Let's 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 are 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-A
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.

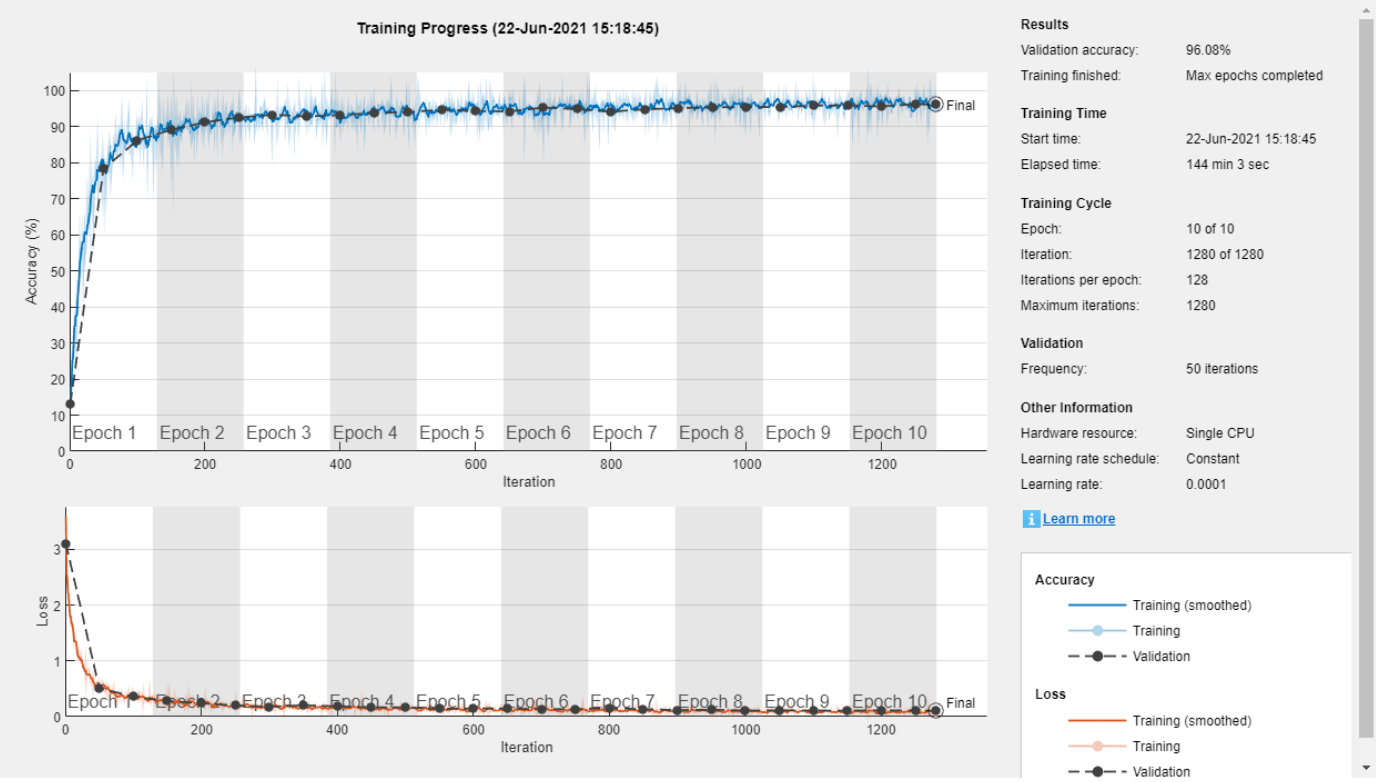
```
numClasses = 8;
net = squeezenet;
lgraphSqz = layerGraph(net);

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);
```



Results

Validation accuracy: 96.08%

Training finished: Max epochs completed

Training Time

Start time: 22-Jun-2021 15:18:45

Elapsed time: 144 min 3 sec

Training Cycle

Epoch: 10 of 10

Iteration: 1280 of 1280

Iterations per epoch: 128

Maximum iterations: 1280

Validation

Frequency: 50 iterations

Other Information

Hardware resource: Single CPU

Learning rate schedule: Constant

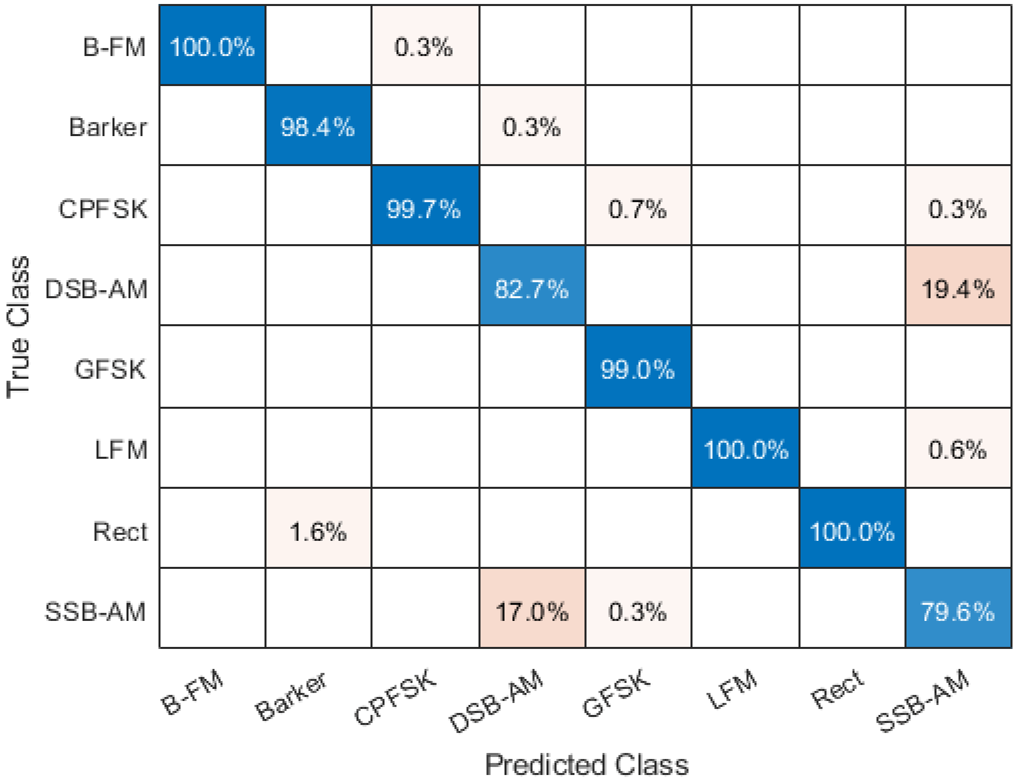
Learning rate: 0.0001

Learn more

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')
```



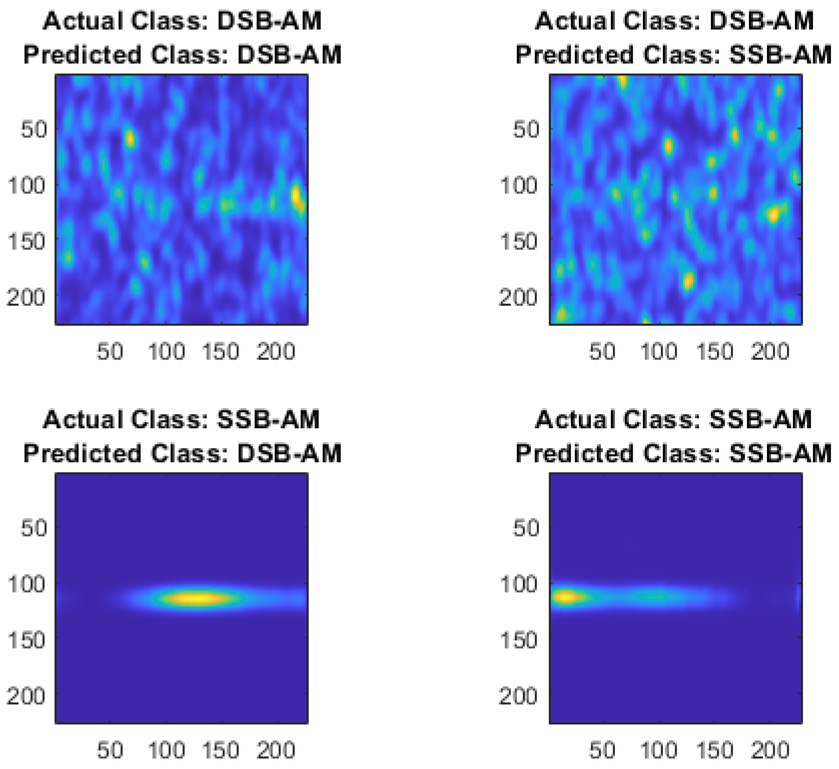
	B-FM	Barker	CPFSK	DSB-AM	GFSK	LFM	Rect	SSB-AM
B-FM	100.0%		0.3%					
Barker		98.4%		0.3%				
CPFSK			99.7%		0.7%			0.3%
DSB-AM				82.7%				19.4%
GFSK					99.0%			
LFM						100.0%		0.6%
Rect		1.6%					100.0%	
SSB-AM				17.0%	0.3%			79.6%

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.

Let us 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'})
```



Actual Class: DSB-AM
Predicted Class: DSB-AM

Actual Class: DSB-AM
Predicted Class: SSB-AM

Actual Class: SSB-AM
Predicted Class: DSB-AM

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.

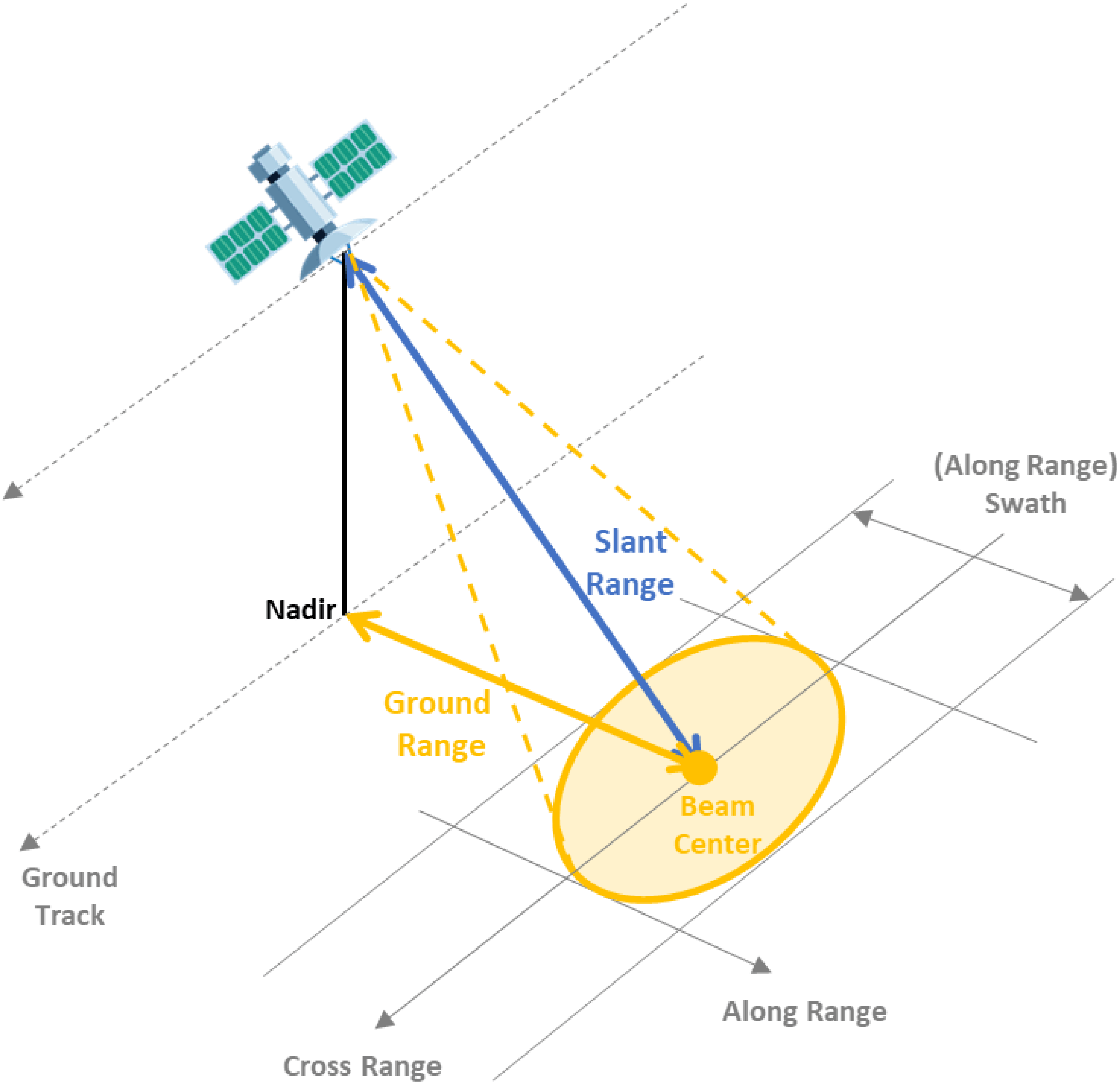
[3] `Wang, Chao, Jian Wang, and Xudong Zhang. "Automatic radar waveform recognition based on time-frequency analysis and convolutional neural network." *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2017.

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
```

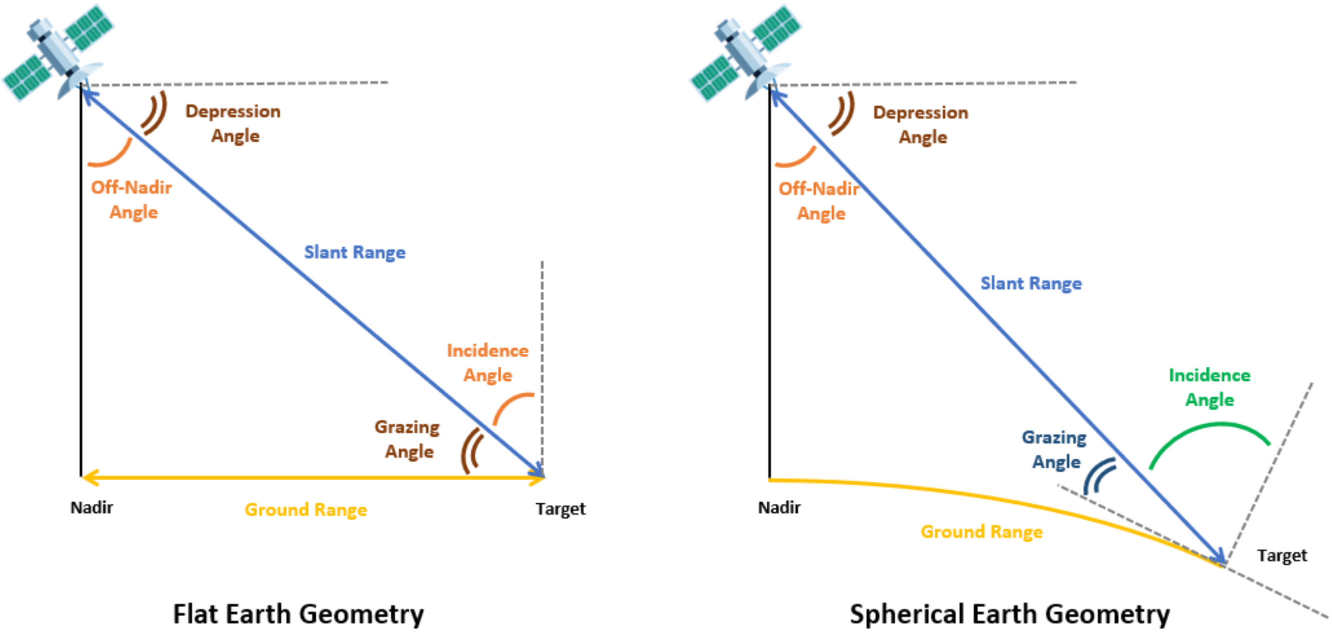
Use the `aperture2gain` function to calculate the antenna gain.

```
antGain = aperture2gain(daz*del, lambda) % dBi
```

```
antGain = 43.8369
```

Antenna Orientation

The depression angle is often used to define the antenna pointing direction.



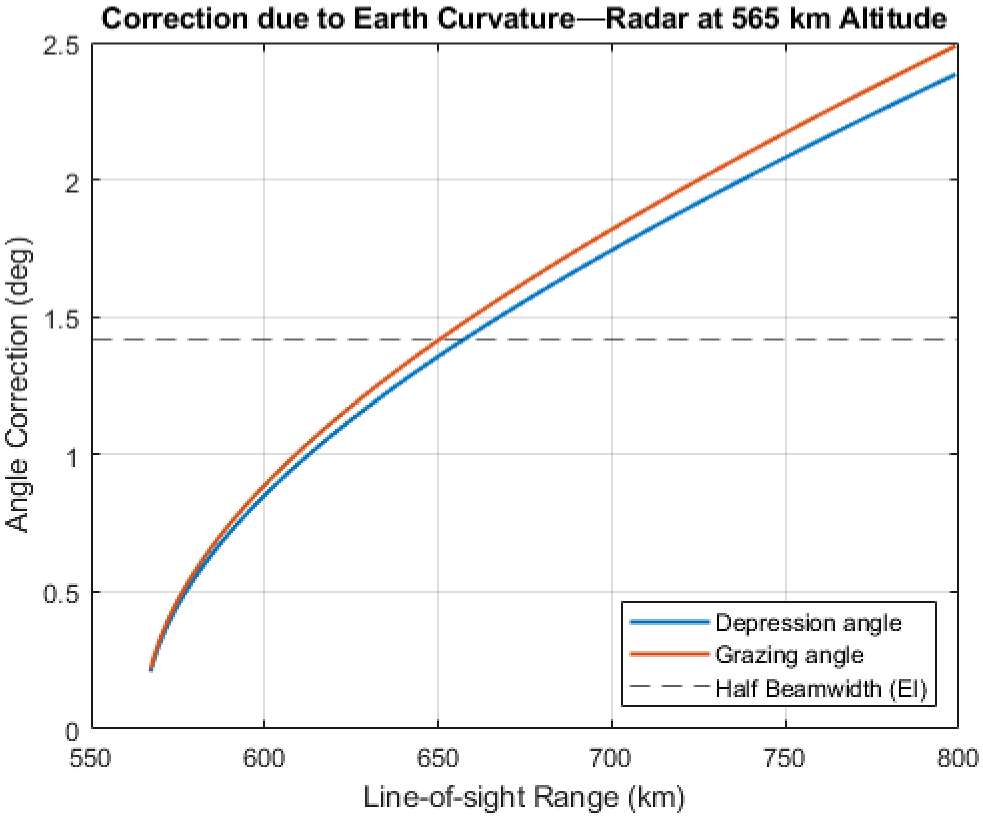
Earth Curvature Effects

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'
```

```
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'
```

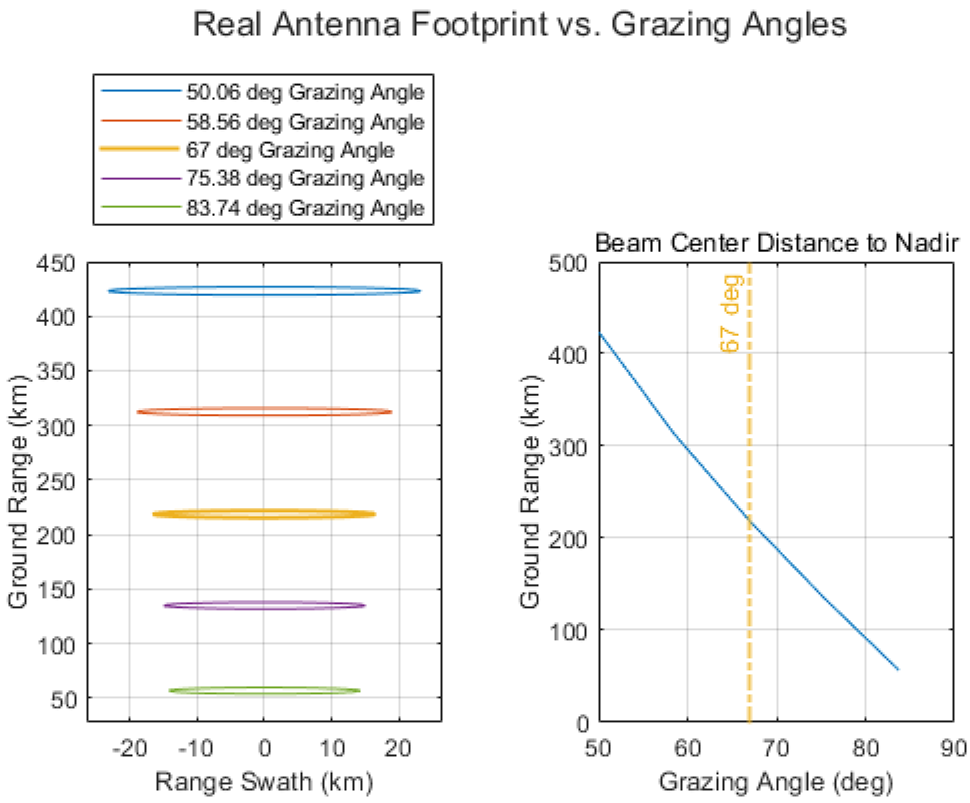
Footprint Sensitivity to Frequency and Angle Variations

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);
```

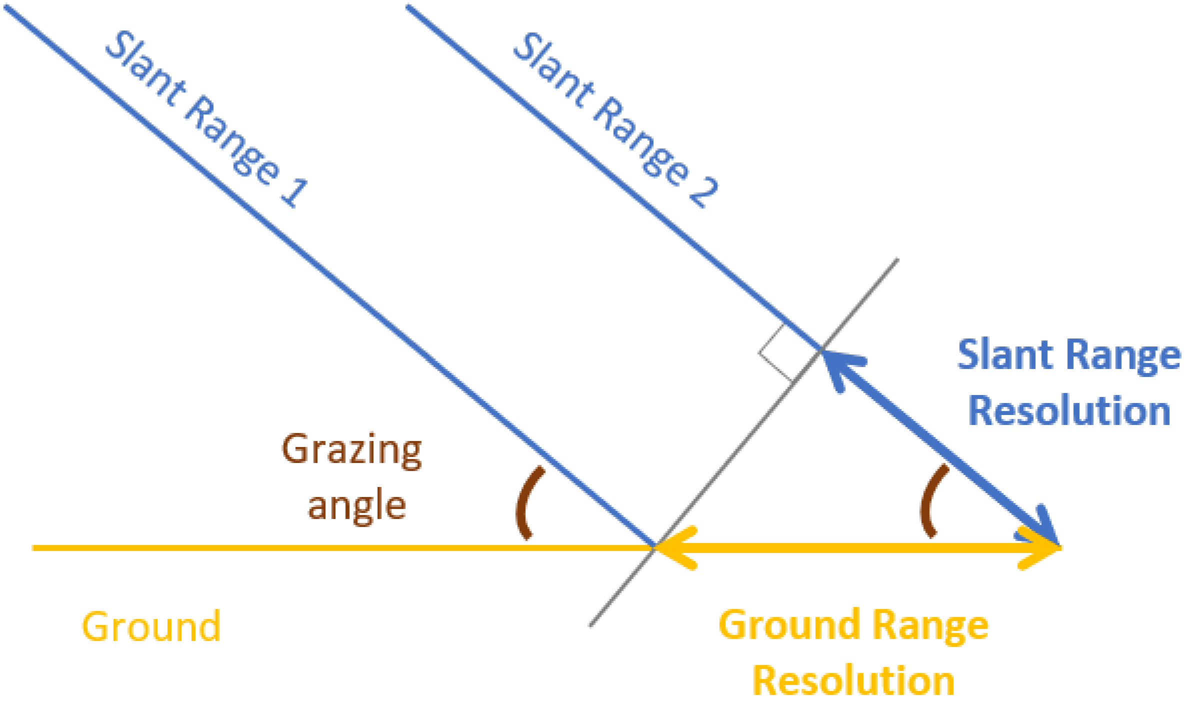


Real Antenna Resolution

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.

```
rngres  = slant2grndrangeres(slantrngres,grazang);
disp(['Real antenna ground range resolution: ', num2str(round(rngres,2)), ' m'])
```

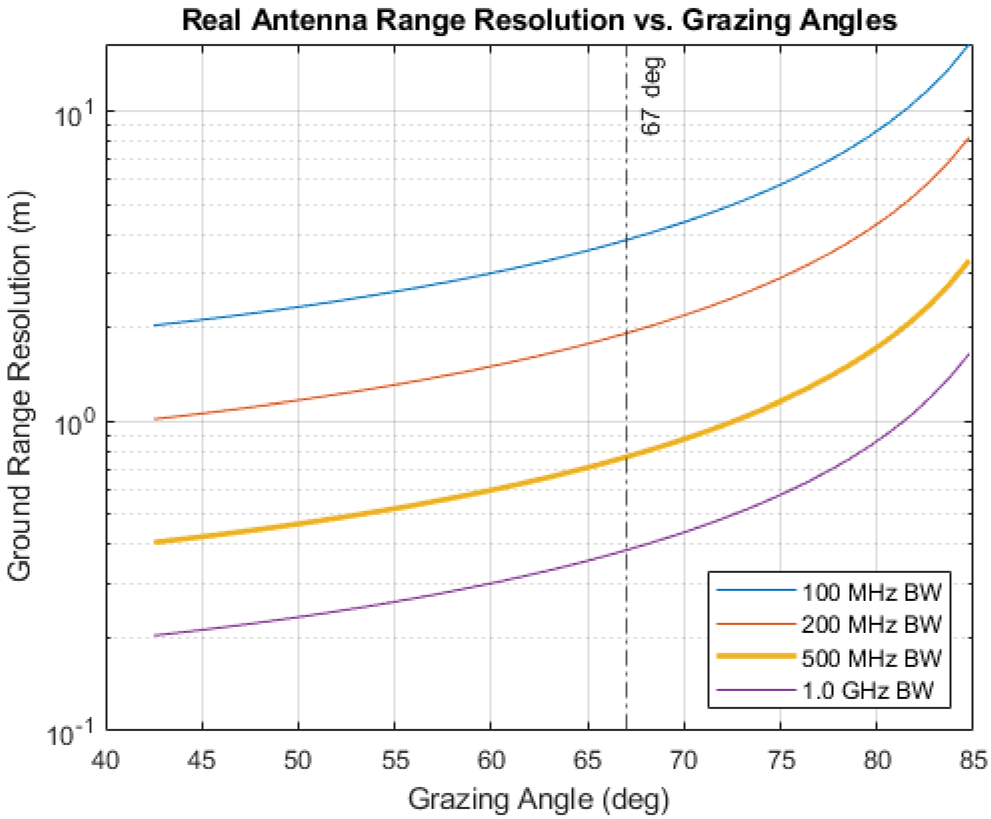
```
Real antenna ground range resolution: 0.77 m
```

```
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.

```
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) r
integrationTime = sarinttime(idealSyntheticApertureLength,v);           % Ideal integration time
```

```
Parameters = ["Synthetic Aperture Length";"Range Resolution";"Cross Range Resolution";"Integratio
IdealSAR = [round(idealSyntheticApertureLength/1e3,1);rngres;round(azResolution,1);round(integrat
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.

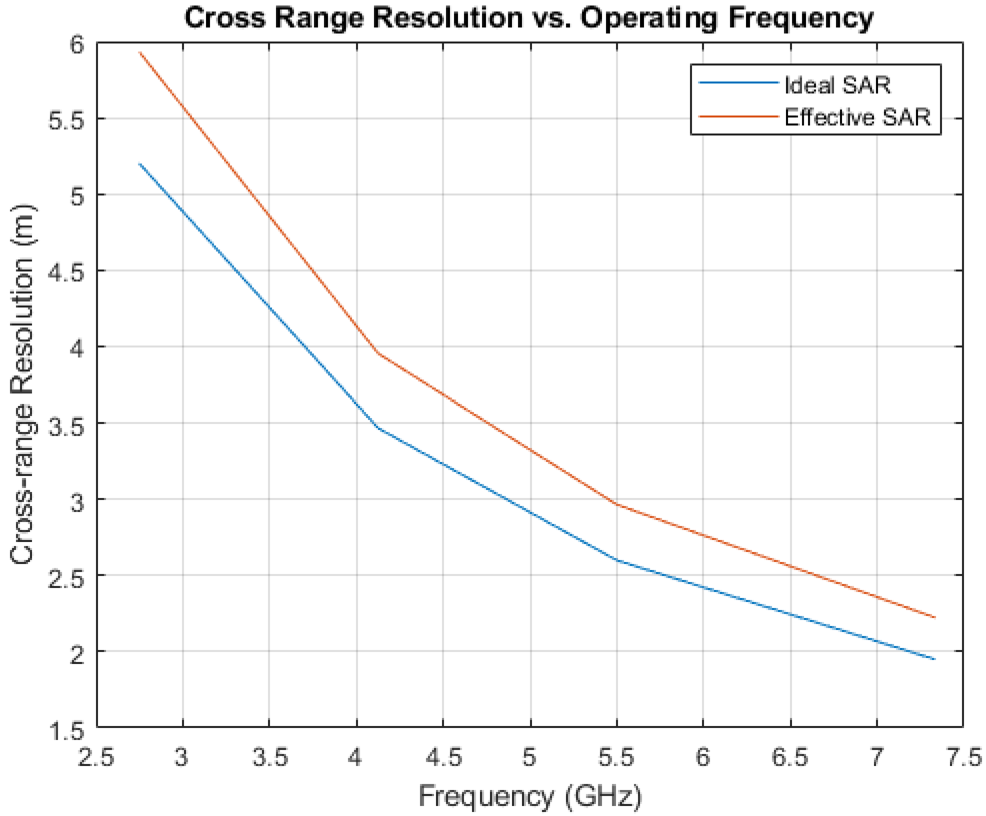
```
effSyntheticApertureLength = sarlen(v,proctime);                           % Take processing time c
effAzResolution = sarazres(lambda,slantrng,effSyntheticApertureLength);  % Effective azimuth reso

RealAntenna = [NaN; rngres; round(crngres); NaN];
EffectiveSAR = [round(effSyntheticApertureLength/1e3,1);rngres;round(effAzResolution,1);round(pro

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')
```



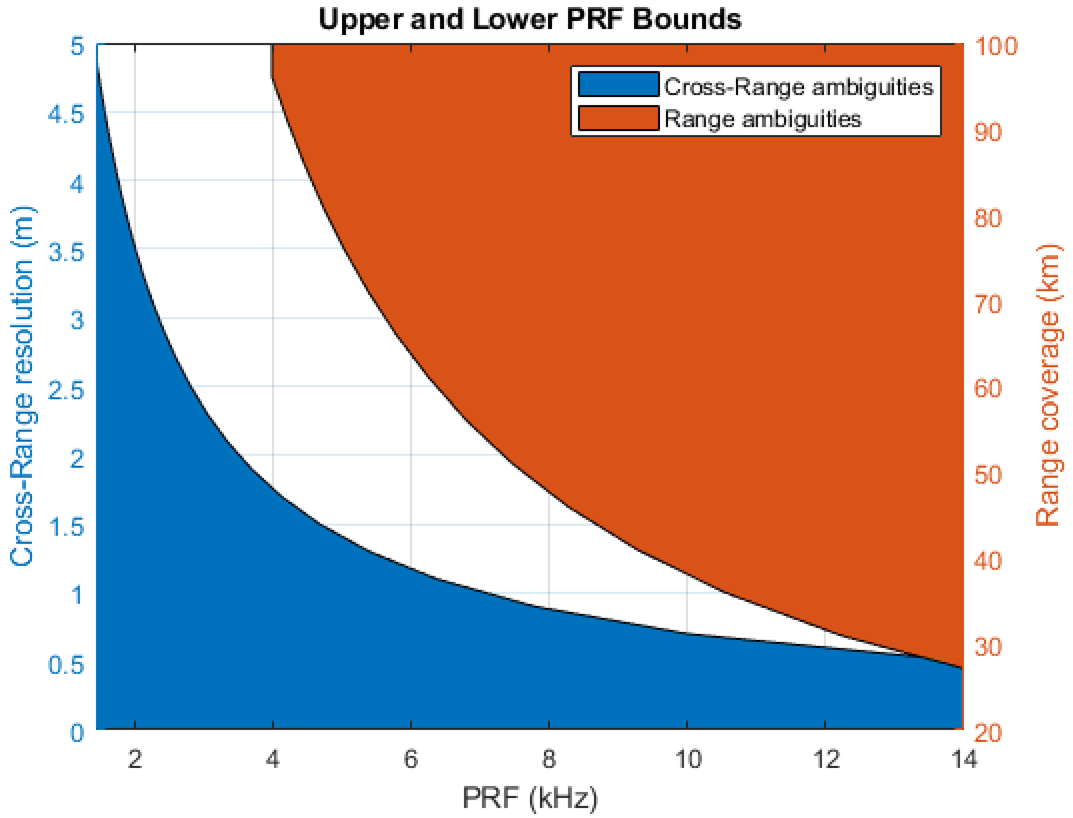
Range and Cross-Range Ambiguities in SAR Images

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',
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)
```

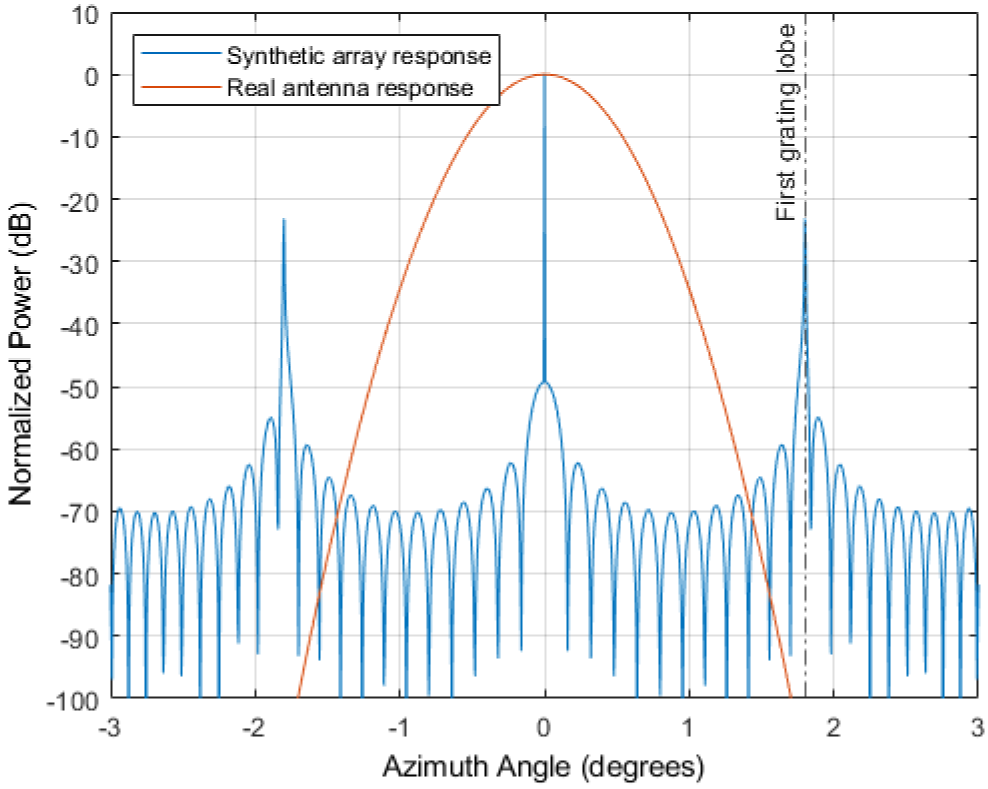
```
ans = 0.6006
```

```
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';
```



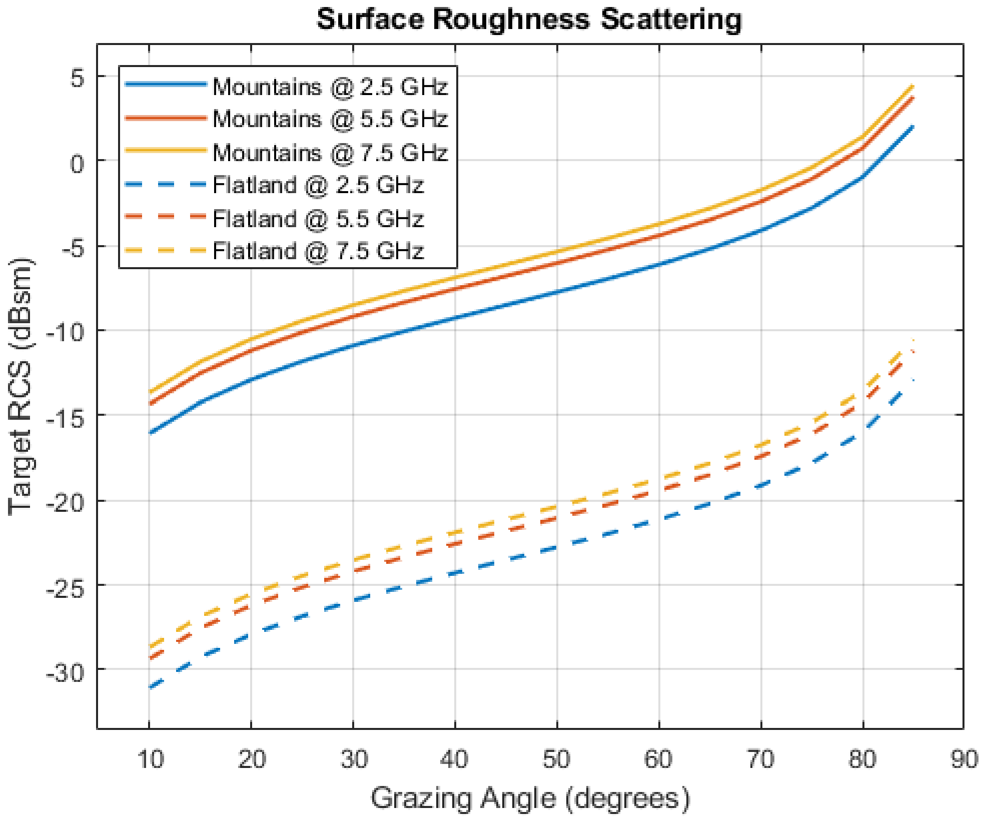
SAR Image SNR and Noise Equivalent Reflectivity

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

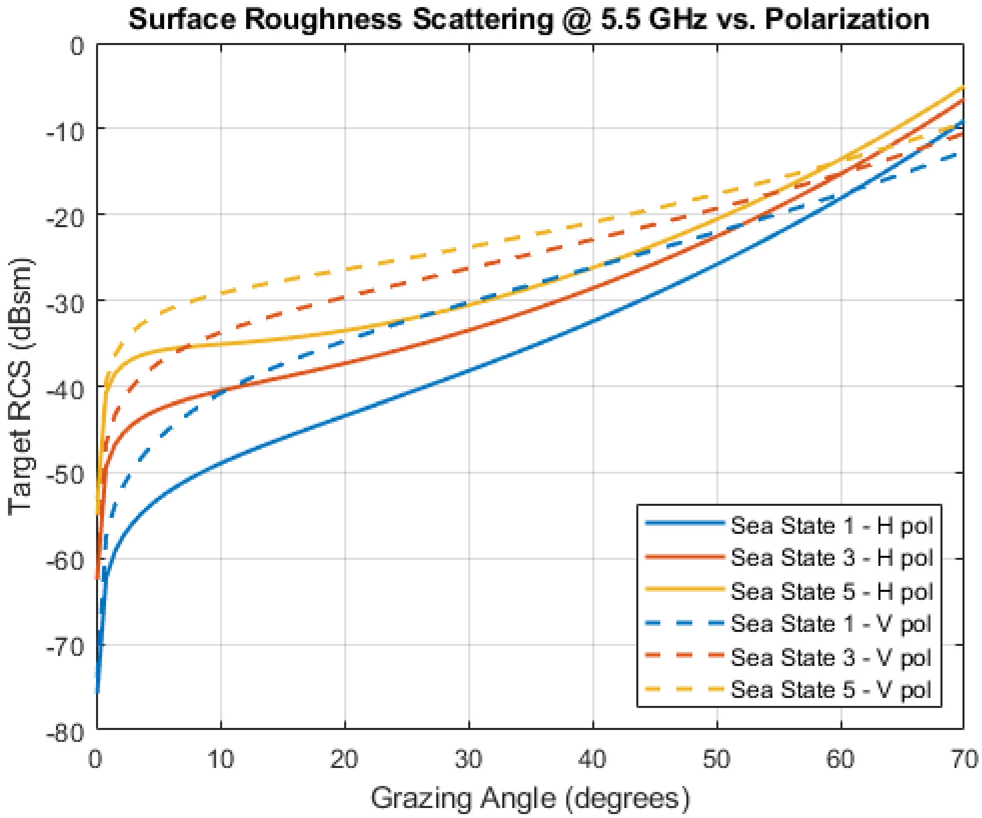
Use the `landreflectivity` function to calculate the reflectivity or the normalized radar cross section (NRCS) for a given grazing angle and operating frequency. The target RCS in the ground image plane is calculated using the `sarSurfaceRCS function` and taking the radar resolution into account. In 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 = 10:5:85;              % 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
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.

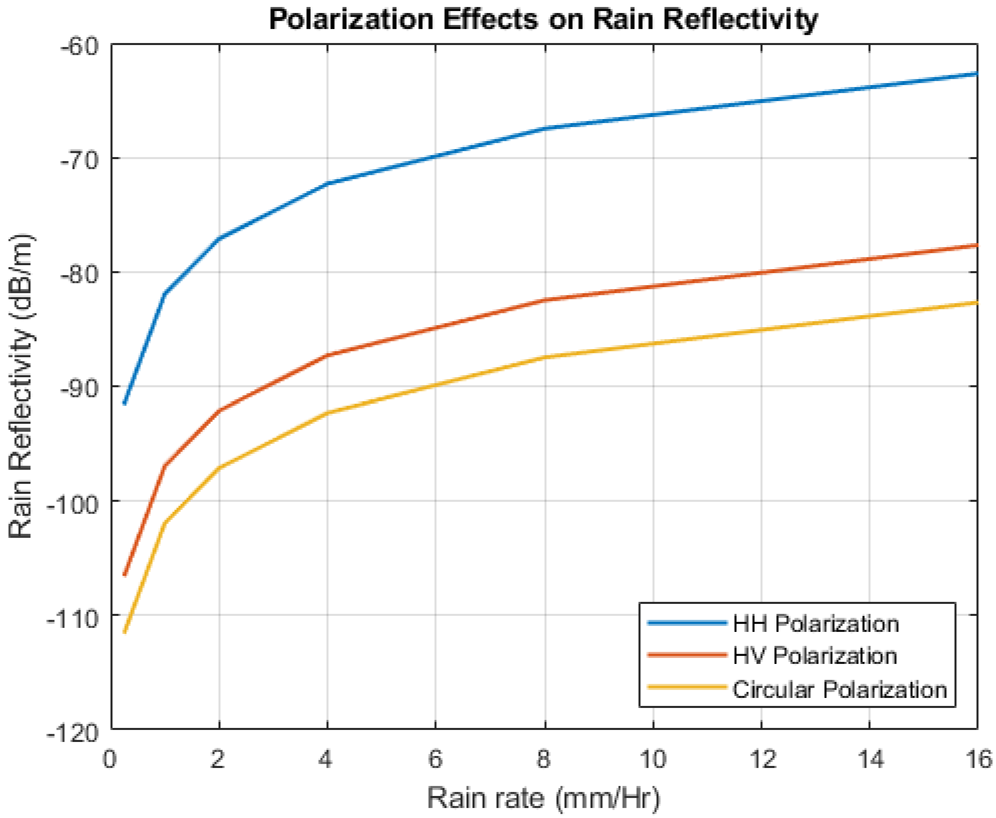
```
pol = ["H","V"];
seaStates = [1 3 5];
grazvpol = linspace(0.1,70,100); % Grazing angle (deg)
tgtrcsv = zeros(numel(grazvpol),numel(pol),numel(seaStates));
for n=1:numel(seaStates)          % Sea states
    for j=1:numel(pol)            % Polarizations
        for u =1:numel(grazvpol) % Grazing angles
            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)
```



Rain Reflectivity and Signal-to-Clutter Ratio (SCR)

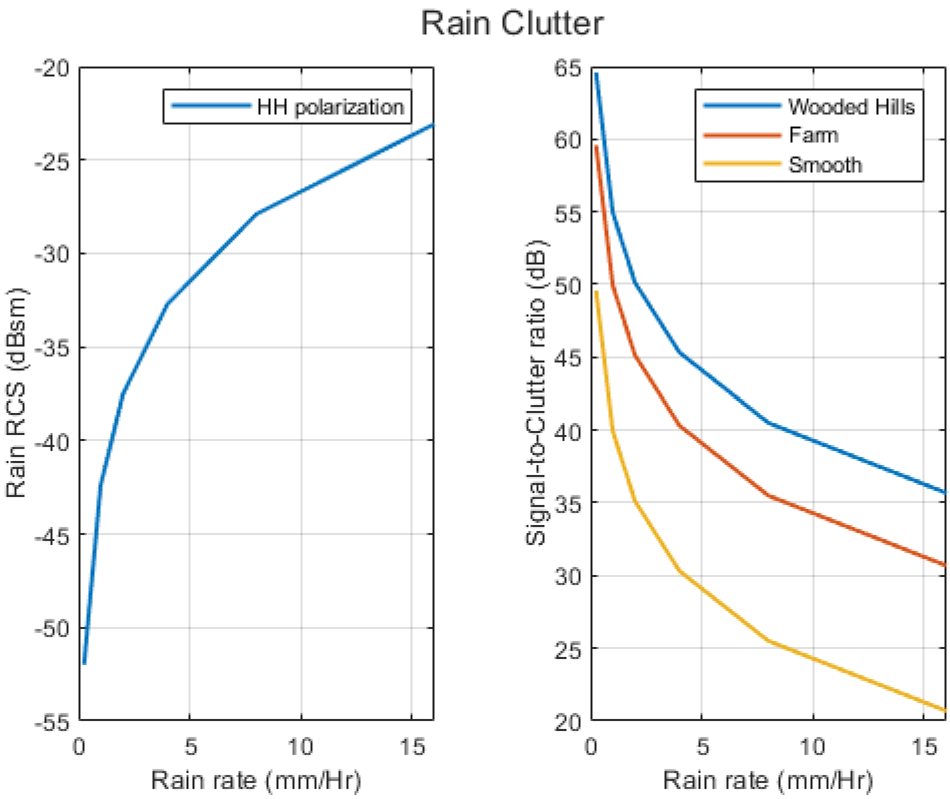
Competing echoes from undesired sources such as rain can cause degradation in SAR images. Rain falling in the vicinity of a target scene clutters the image of that scene. Use the `rainreflectivity` function to analyze rain reflectivity under varying rain rates and polarizations. Observe that the rain 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');
```



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

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 of smooth
    tgtrcs(j) = sarSurfaceRCS(nrcsv(j),[slantrngres effAzResolution],grazang);
    scr(:,j) = rainscr(lambda,rrcs,tgtrcs(j),proctime)';   % Signal to rain clutter ratio
end

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 = 0.0439
```

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.

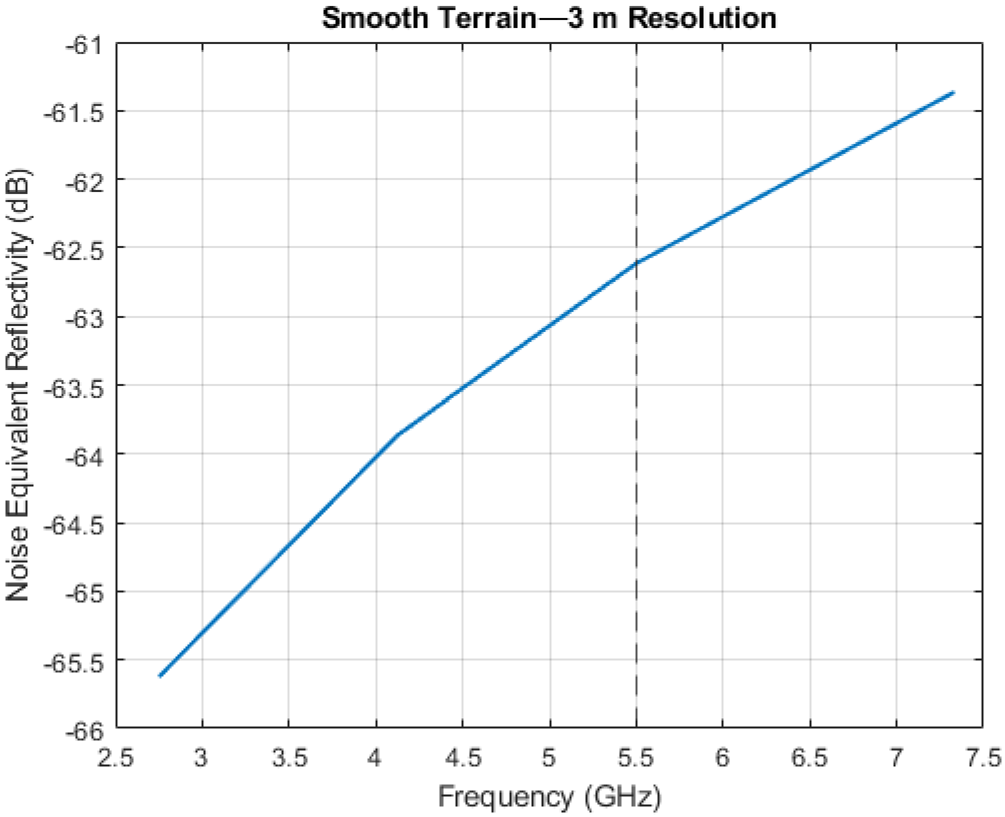
```
Lb = beamloss;
customLoss = 5;        % dB
Pt = 5e3;              % Peak power (W)
sntemp = systemp(cnf); % Noise Temperature
imgsnr = radareqsarsnr(slantrng,lambda,Pt,pw,rnggain,azgain,'Gain',antGain,'RCS',tgtrcs(3),...
    'AtmosphericLoss',atmoLoss,'Loss',cnf+rainLoss,'PropagationFactor',propf,...
    'Ts',sntemp,'CustomFactor',-Lb-customLoss)
```

```
imgsnr = 35.9554
```

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.

```
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

```
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

```
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
% 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

```
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

```
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

```
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
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

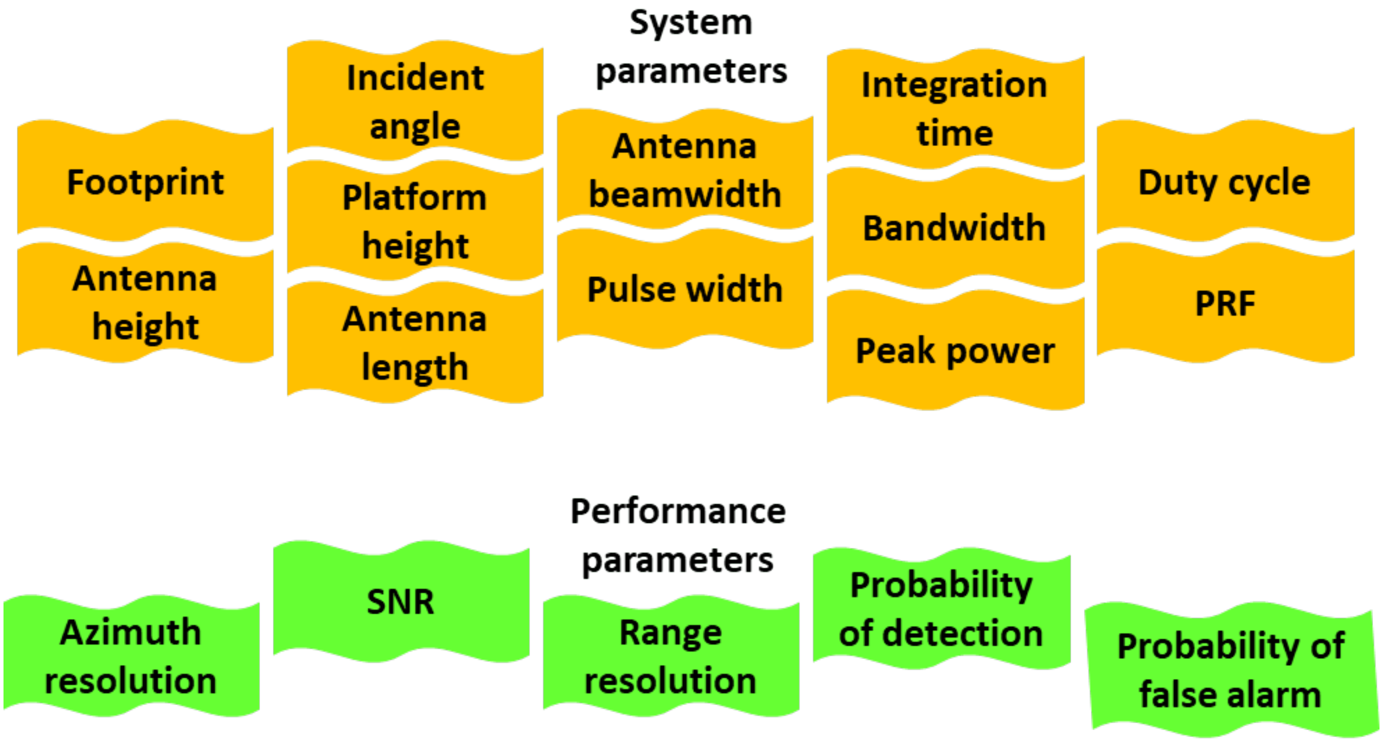
```
function plotRainSCR(rainRate,rrcs,scr,landType)
t = tiledlayout(1,2);
nexttile
% Plot rain RCS vs rain rate
plot(rainRate,pow2db(rrcs),'LineWidth',1.5)
grid on
xlabel('Rain rate (mm/Hr)')
ylabel('Rain RCS (dBsm)')
legend('HH polarization')

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.

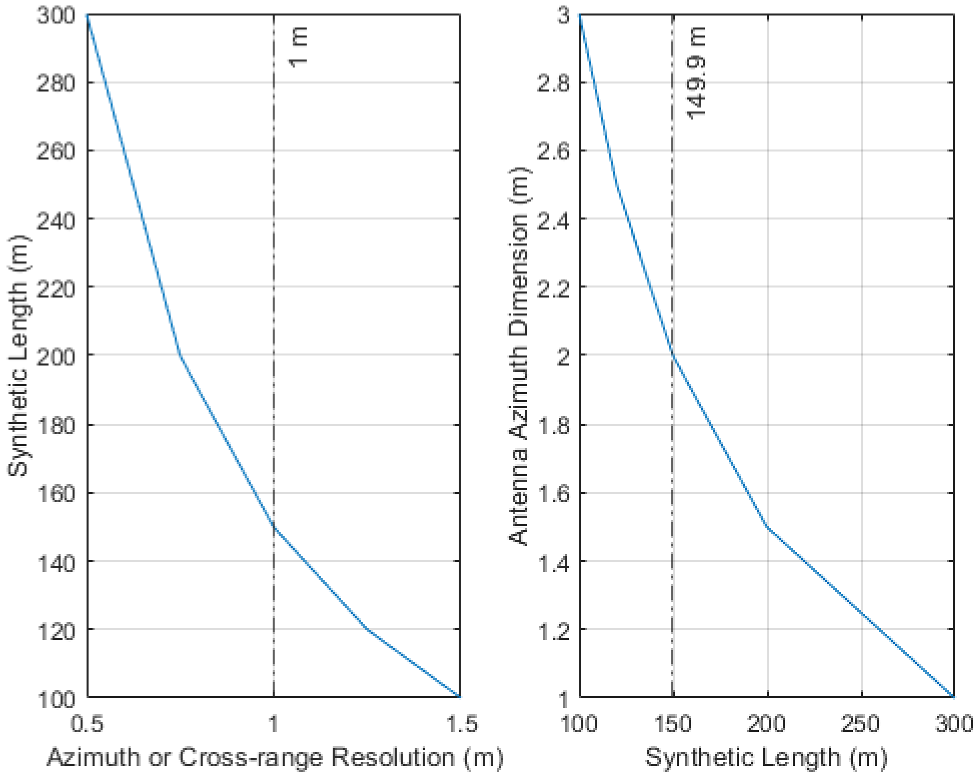
```
grazang = grazingang(h,maxrng,'Flat')  % Grazing angle (in degrees)
```

```
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
```

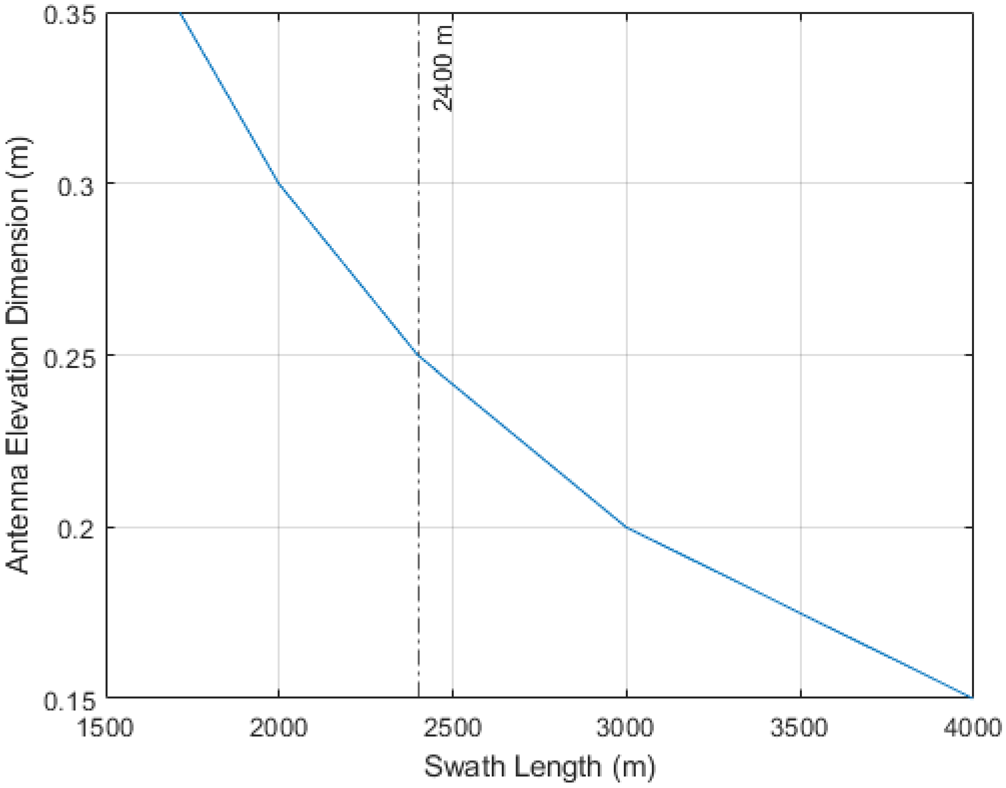
```
striplen = 149.9000
```

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 Swath Length";...
    "Upper Bound on Area Coverage Rate"];
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 Swath Length"                NaN           173.1     "km^2/s"
    "Upper Bound on Area Coverage Rate"          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
```

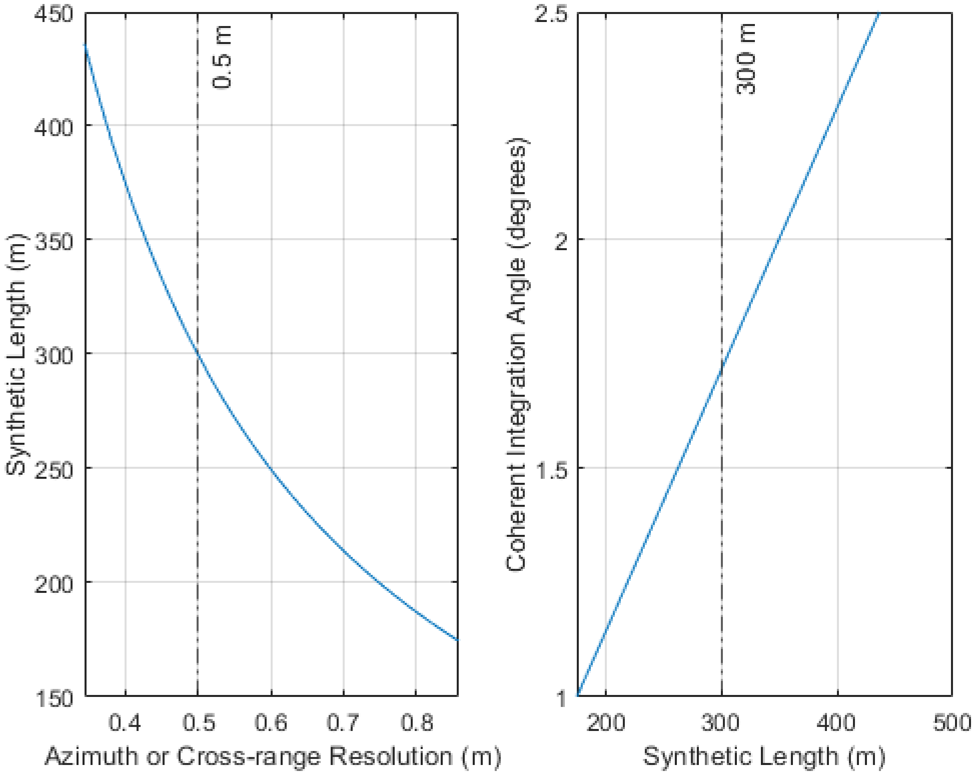
```
len = sarlen(maxrng,'CoherentIntegrationAngle',ciang)   % (m)
```

```
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 Swath Length"               173.1            86.5     "km^2/s" 
    "Upper Bound on Area Coverage Rate"          1731             865     "km"     
```

Azimuth Chirp Signal Parameters

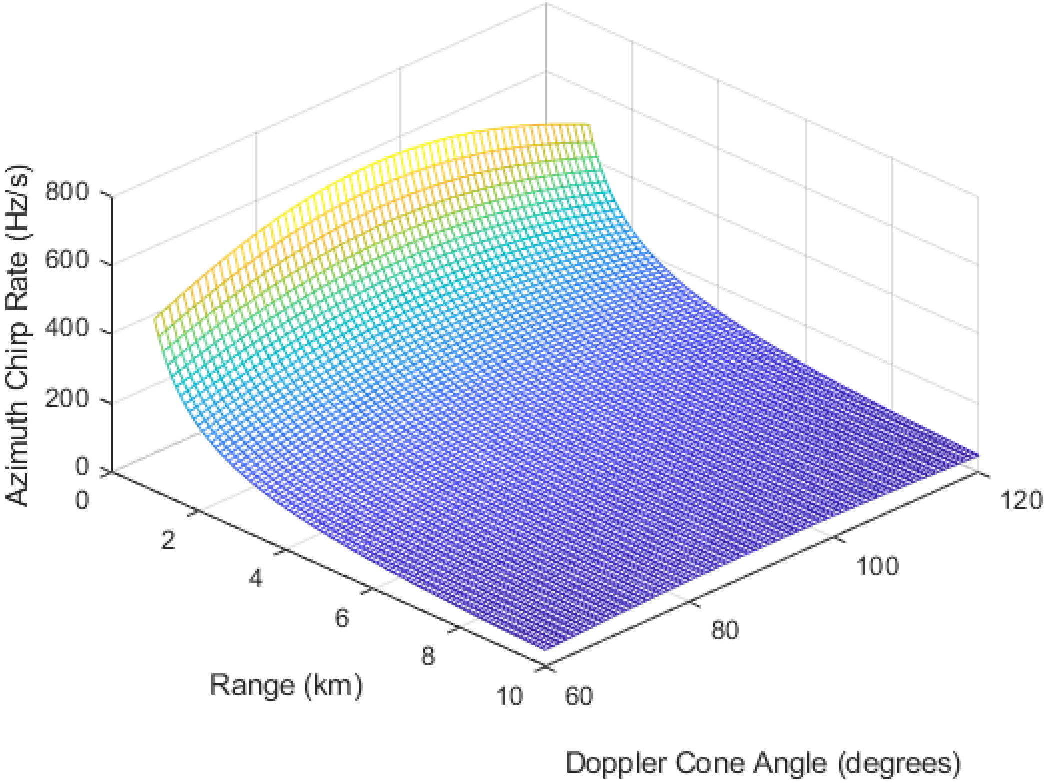
Determine the azimuth chirp signal parameters which are the azimuth chirp rate, Doppler bandwidth, beam compression ratio, and azimuth bandwidth after dechirp. You can derive the azimuth time-bandwidth product. These are important for designing an accurate synthetic aperture processing mechanism in azimuth.

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]);
```

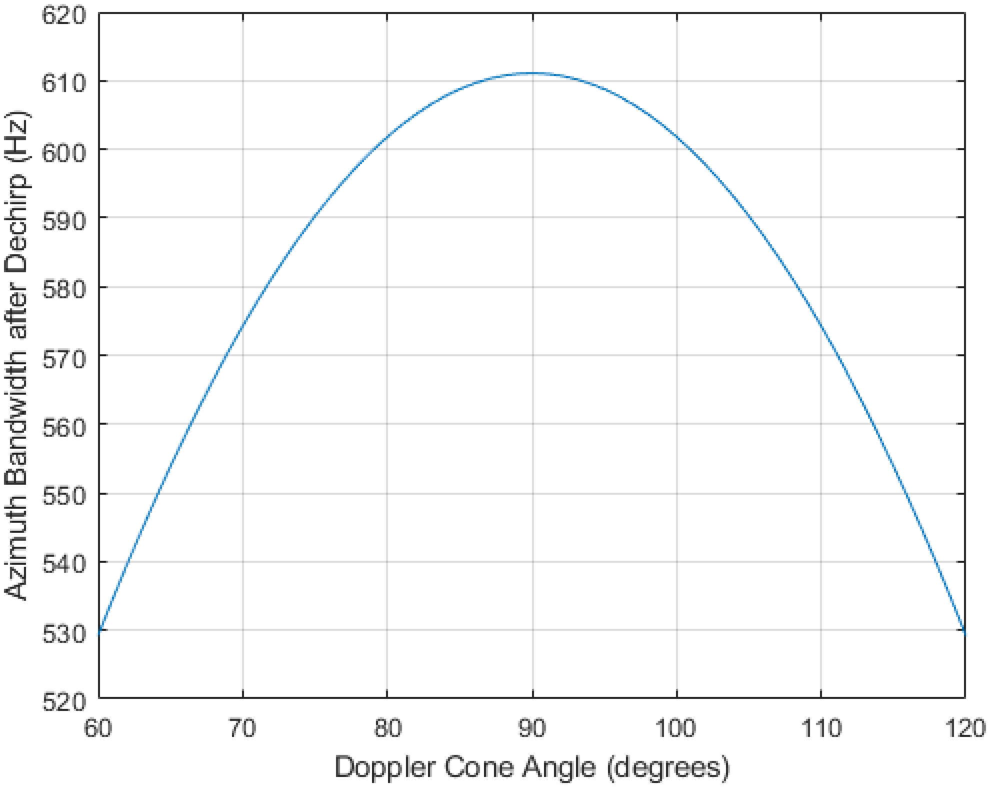


Use the sarscenedopbw function to calculate the scene bandwidth after azimuth dechirping. Assume a scene size of 916 m.

```
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"
```

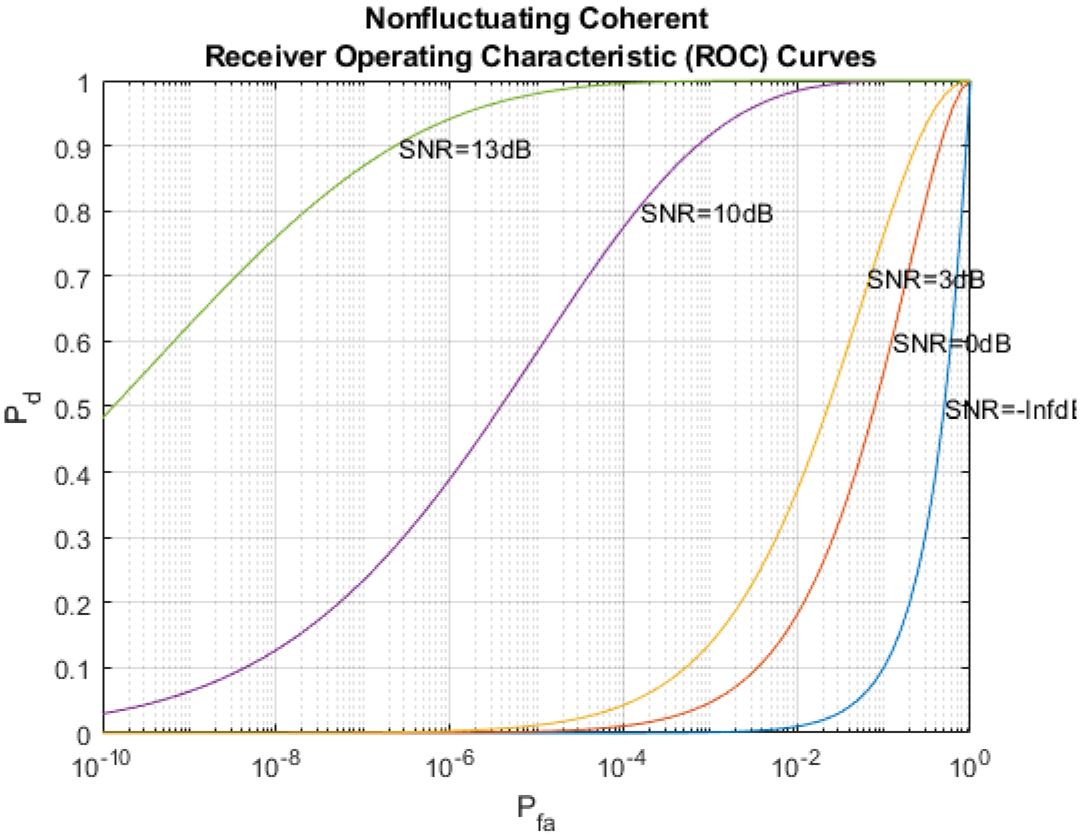
SAR Power Calculation

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 = 0.0733
```

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 = 535.1030
```

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 Swath Length";
    "Upper Bound on Area Coverage Rate";"Coherent Integration Angle";"Doppler Bandwidth from Poi
    "Azimuth Time-Bandwidth Product";"Beam Compression Ratio";"Azimuth Chirp Rate";"Azimuth Band
Stripmap = [daz;del;striplen;azres;stripsynbw;stripinttime;round(stripcovrate/1e6,1);round(strip
    NaN;stripbwchirp;striptbwaz;stripbcr;round(azchirp);bwdechirp];
Spotlight = [daz;del;spotlen;spotazres;spotsynbw;spotinttime;round(spotcovrate/1e6,1);round(spot
    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 Swath Length"                   173.1         86.5    "km^2/s"
    "Upper Bound on Area Coverage Rate"              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
```

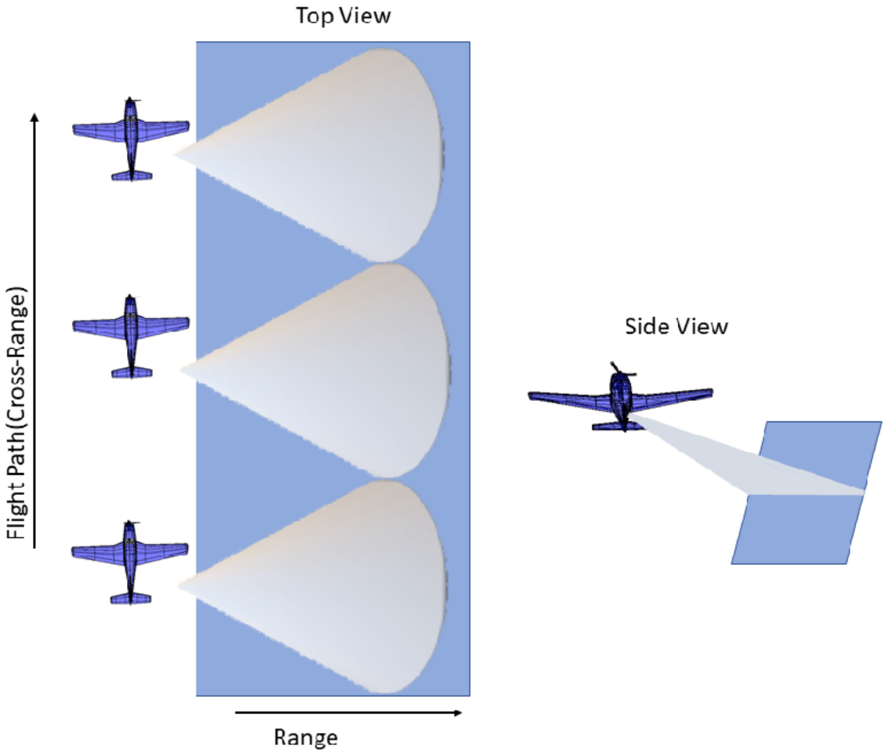
Stripmap Synthetic Aperture Radar (SAR) Image Formation

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, for example. 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 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 beam-width 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 beam-width 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 will help 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 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.



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 the figure above. 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 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.

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);
```

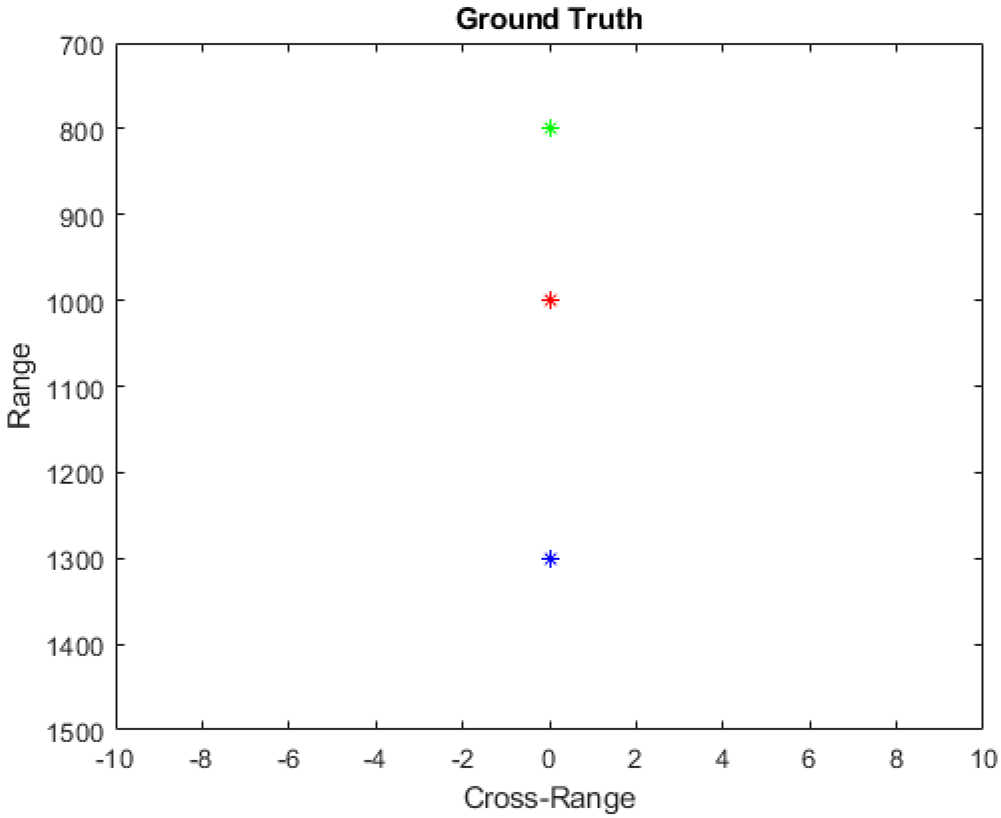
Configure the propagation channel.

```
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

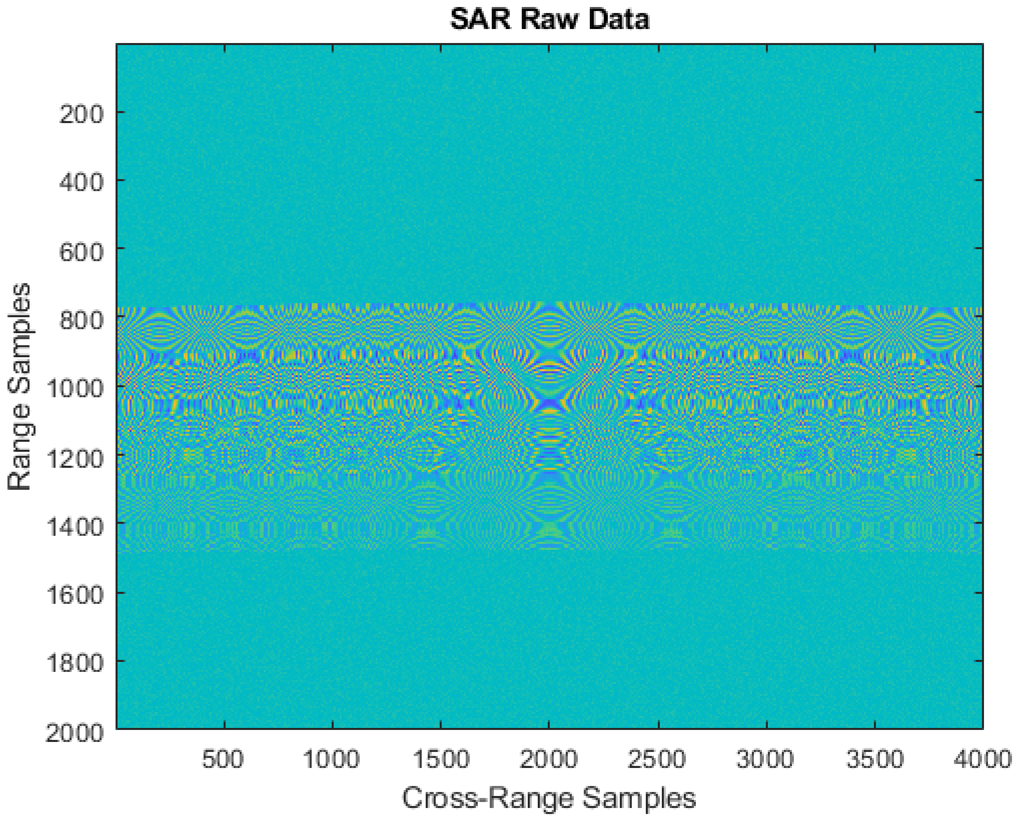
The following section describes how the system operates based on the above configuration. Specifically, the section below shows how the data collection is performed for a SAR platform. As the platform moves in the cross-range direction, pulses are transmitted and received in directions orthogonal to 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')
```



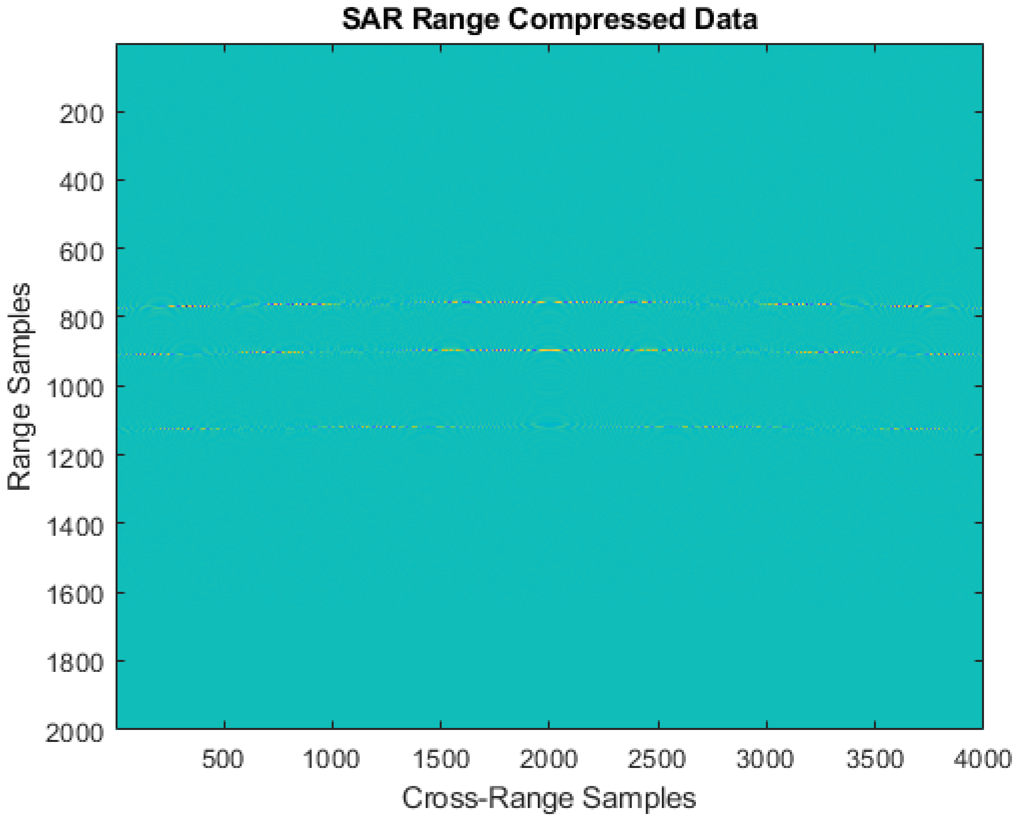
Perform range compression.

Each row of the received signal, which contains all the information from each pulse, can be matched filtered to get the de-chirped/range compressed signal.

```
pulseCompression = phased.RangeResponse('RangeMethod', 'Matched filter', 'PropagationSpeed', c,
matchingCoeff = getMatchedFilter(waveform);
[cdata, rnggrid] = pulseCompression(rxsig, matchingCoeff);
```

The figure below 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')
```



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. Two such techniques are the range migration algorithm and the back-projection algorithm which have been 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
```

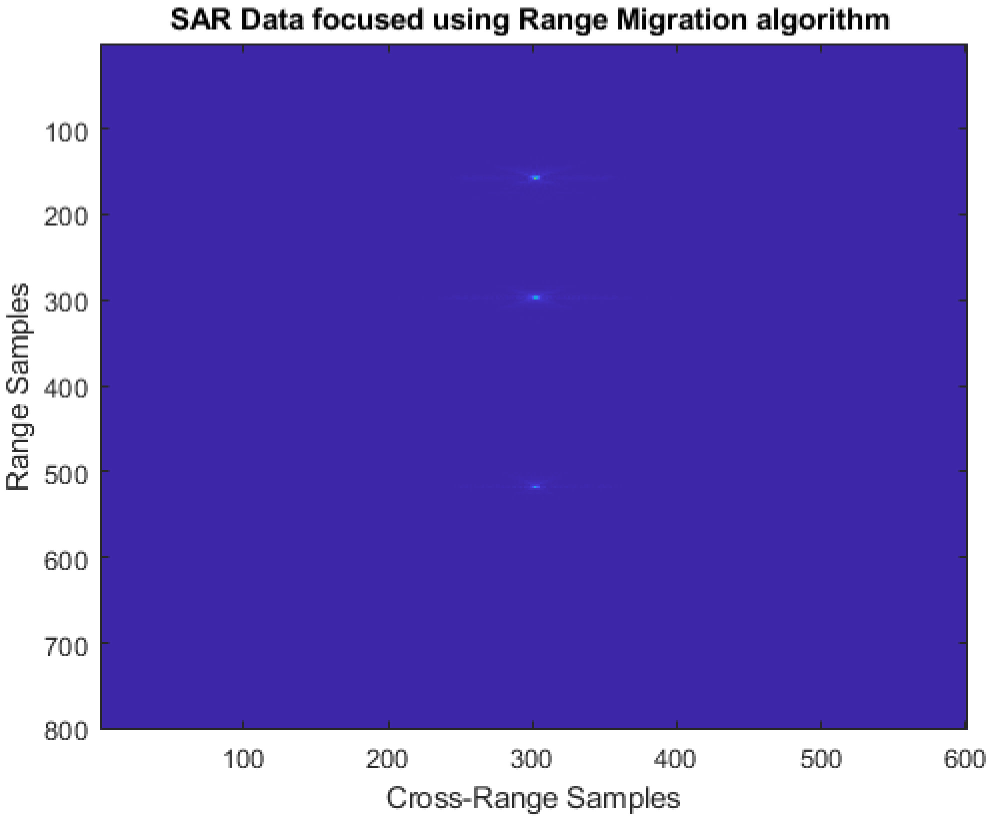
Visualize the final SAR image.

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.

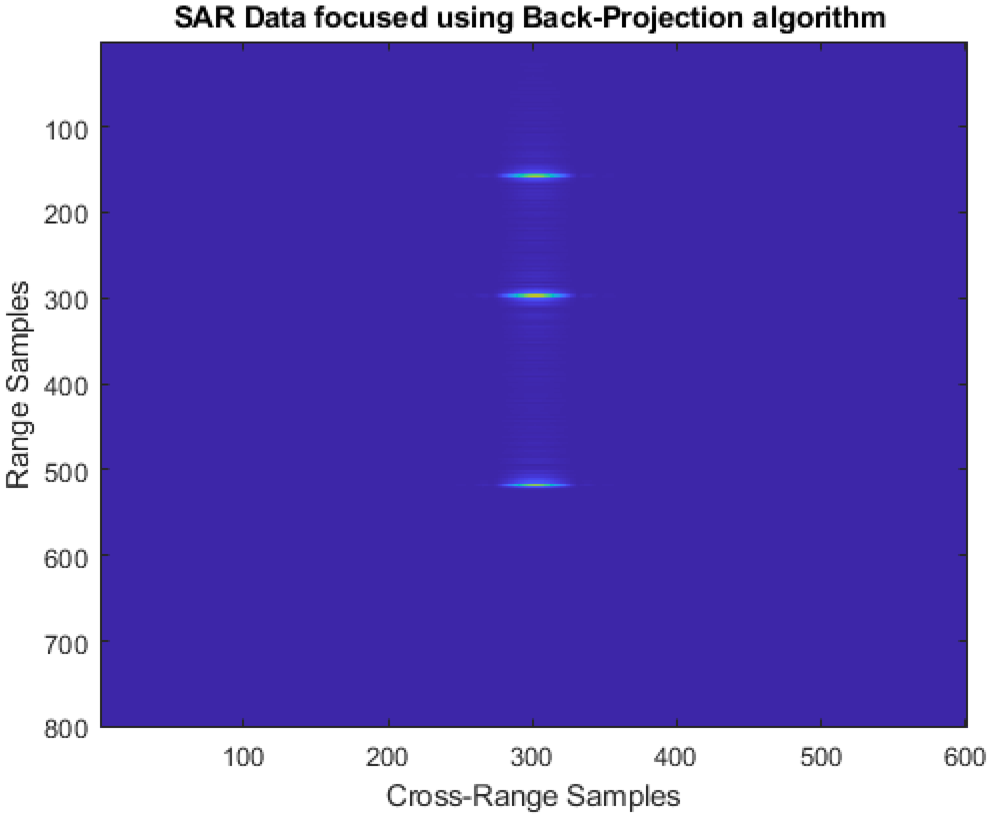
The range migration and accurate form of the backprojection algorithm as shown by [2] and [3] provides theoretical resolution in the cross-track as well as along-track direction. Since the back-projection used here is of the approximate form, the spread in azimuth direction is evident in case of the 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')
```



```
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')
```



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., Prati, C. and Rocca, F., 1991. SAR data focusing using seismic migration techniques. IEEE transactions on aerospace and electronic systems, 27(2), pp.194-207.
2. Cumming, I., Bennett, J., 1979. Digital Processing of Seasat SAR data. IEEE International Conference on Acoustics, Speech, and Signal Processing.
3. Na Y., Lu Y., Sun H.,A Comparison of Back-Projection and Range Migration Algorithms for Ultra-Wideband SAR Imaging, Fourth IEEE Workshop on Sensor Array and Multichannel Processing, 2006., Waltham, MA, 2006, pp. 320-324.
4. Yegulalp, A.F., 1999. Fast backprojection algorithm for synthetic aperture radar. Proceedings of the 1999 IEEE Radar Conference.

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 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

```
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,crossRangeResoluti
```

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.

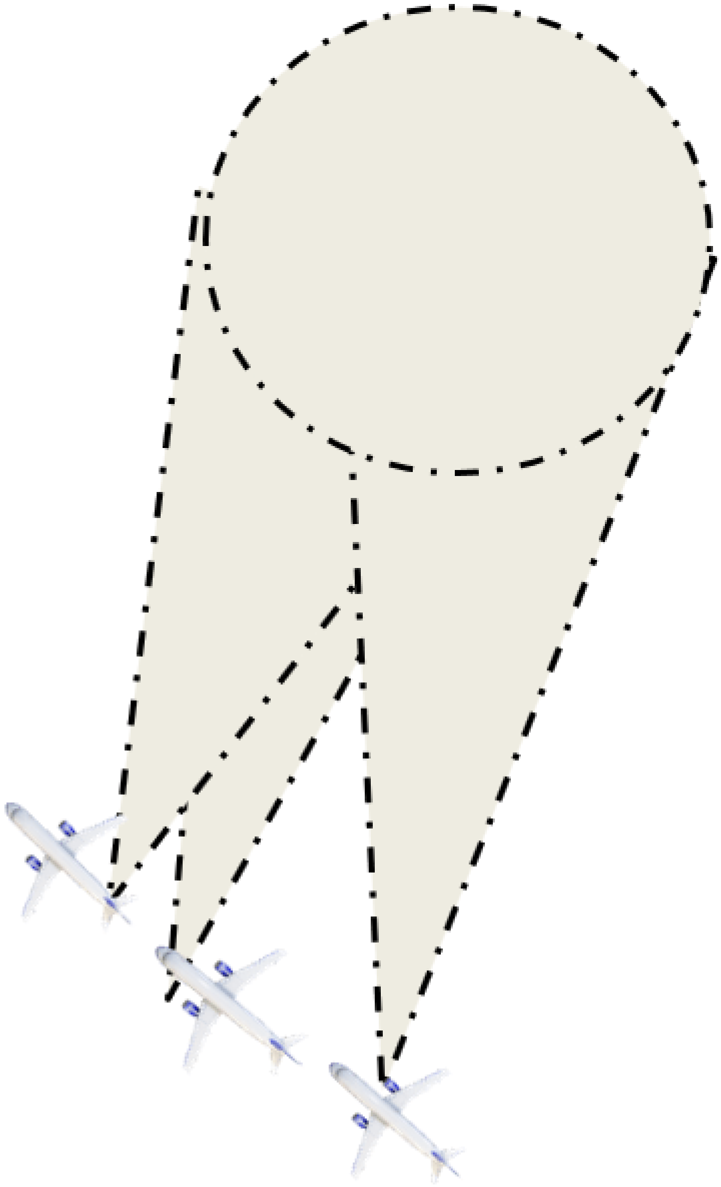
```
rangeIdx =  [find(rnggrid>rangelims(1), 1) find(rnggrid<rangelims(2),1,'last')];
```

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 1-305.



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);
```

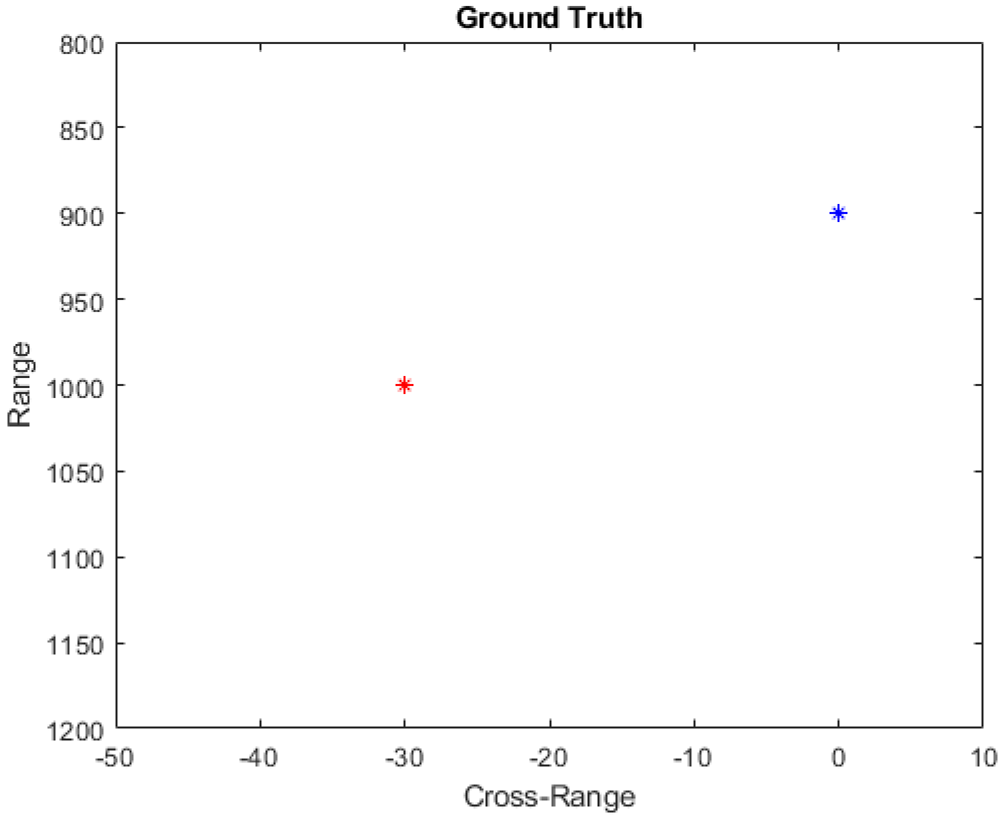
Scene Configuration

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

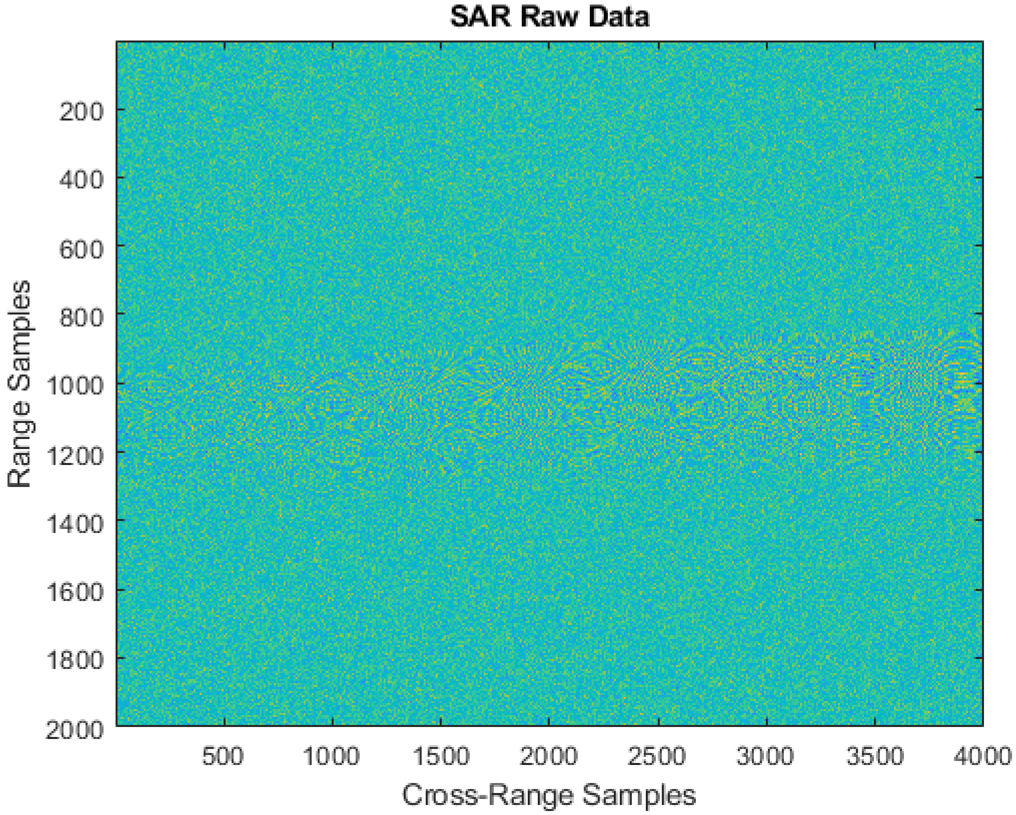
The following section describes how the system operates based on the above configuration. Specifically, the section below shows how the data collection is performed for a SAR platform. As the platform moves in the cross-range direction, pulses are transmitted and received in directions defined by the 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')
```



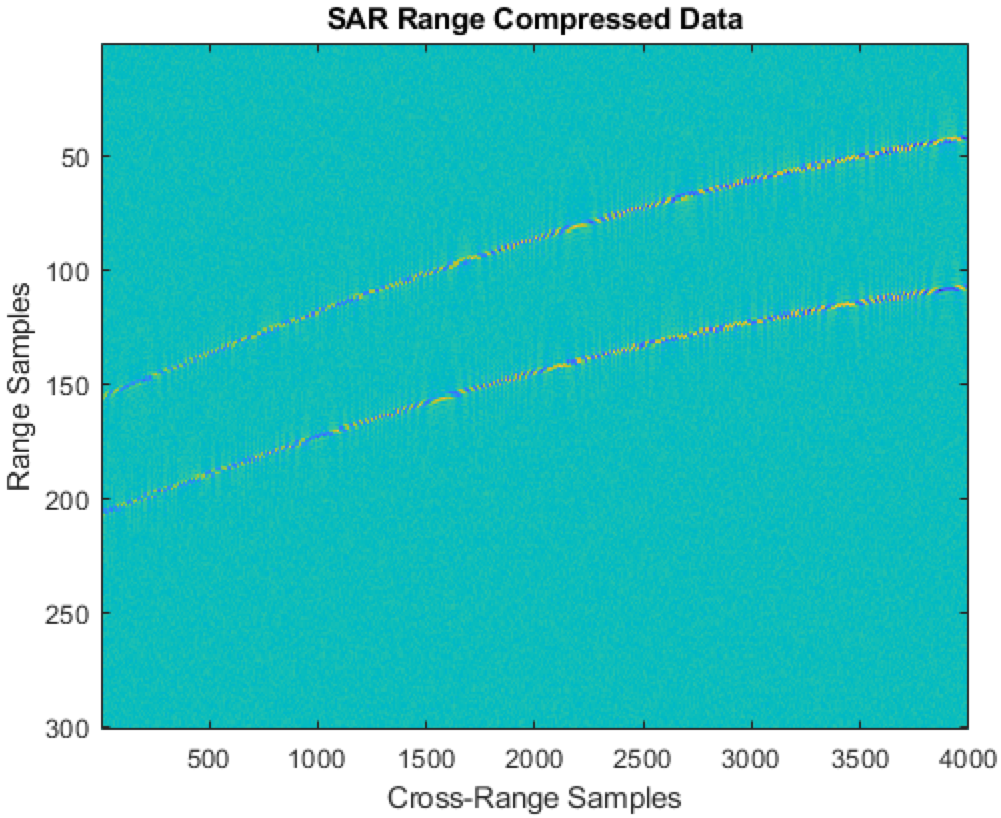
Perform range compression.

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

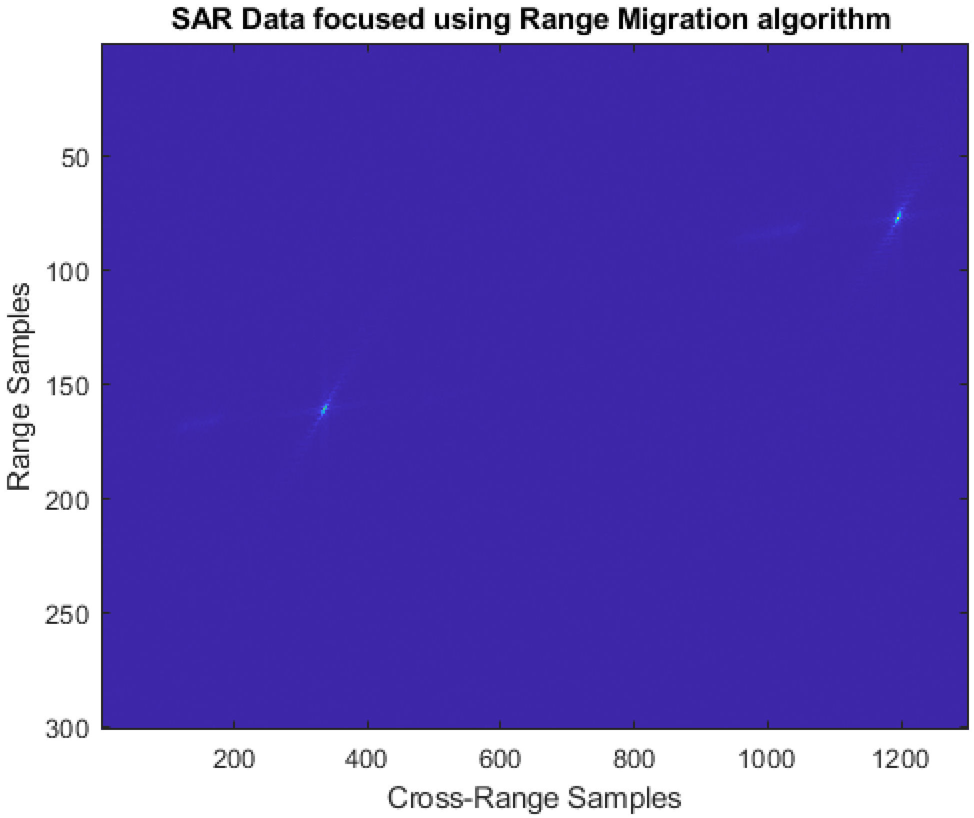
There are multiple techniques to process the cross-range data and get the final image from the SAR raw data once range compression has been achieved. In essence, the range compression helps achieve resolution in the fast-time or range direction and the resolution in the cross-range direction is achieved 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')
```



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., Prati, C. and Rocca, F., 1991. SAR data focusing using seismic migration techniques. IEEE transactions on aerospace and electronic systems, 27(2), pp.194-207.

2 Soumekh, M., 1999. Synthetic Aperture Radar Signal Processing with MATLAB algorithms. John Wiley & Sons, Inc.

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

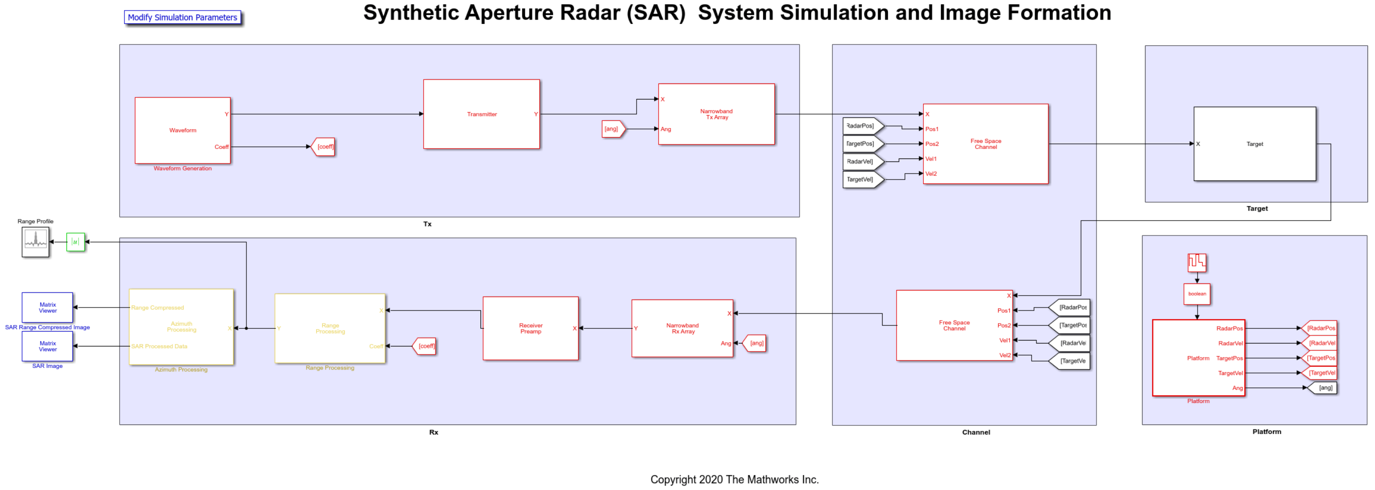
This example demonstrates how to model a Synthetic Aperture RADAR (SAR) system using Stepped Frequency Modulated (SFM) waveform and generate a SAR image in Simulink®. In this example a SAR platform is defined along with the waveform it transmits. The receiver is modeled to handle the matched filtering and azimuth processing of different sub pulses of the SFM waveform. A similar example in MATLAB using Linear Frequency Modulated (LFM) waveform can be found in “Stripmap Synthetic Aperture Radar (SAR) Image Formation” on page 1-305.

Introduction

The model shows a SAR system setup to simulate IQ returns and perform image formation from the IQ data. The aircraft/airborne platform in this example exploits SFM waveform and its corresponding processing. The SFM waveform is an alternative technique to obtain larger bandwidth and it has some advantages over conventional LFM waveform.

As the SFM constructs a wide-band chirp from a burst of narrow-band chirps it provides improved noise figure and receiver sensitivity because of the narrow instantaneous bandwidth.

```
helperslexSARSystemSim('openModel');
```

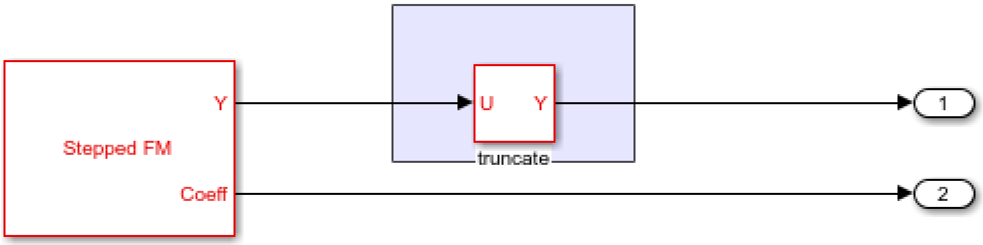


The model has various sections Tx, Channel, Target, Platform and Rx. Tx section simulates the generation and transmission of SFM waveform. The channel section models the two-way propagation of the signal to and from the target. The target section models three point targets located at 800 m, 1000 m and 1300 m each with a mean RCS of 1. The platform subsystem models the flight path. The Rx section simulates the reception, range, and azimuth processing of SAR raw data.

Waveform Generation

The Waveform Generation subsystem includes a SFM waveform generating module. The transmitted SFM burst consists of 4 sub pulses with 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.

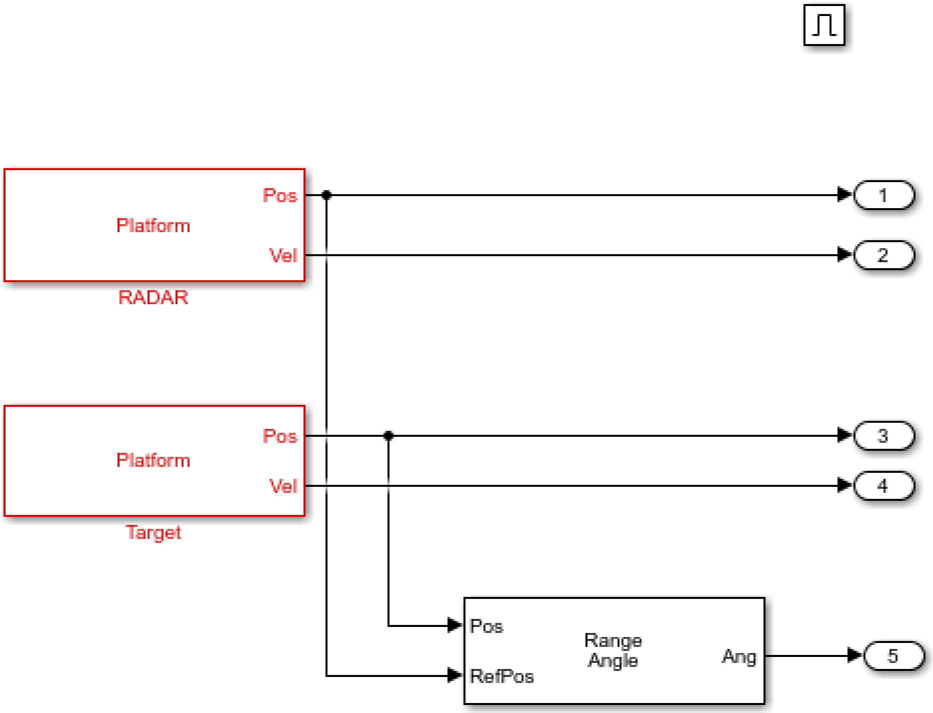
```
helperslexSARSystemSim('showWaveform');
```



Platform

The platform subsystem simulates the motion of flight on which the synthetic aperture radar is mounted. It is a triggered subsystem that mimics a stop and go assumption as the platform moves from a 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 sub pulses in a burst are transmitted and received. The sub pulses here implies the 4 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.

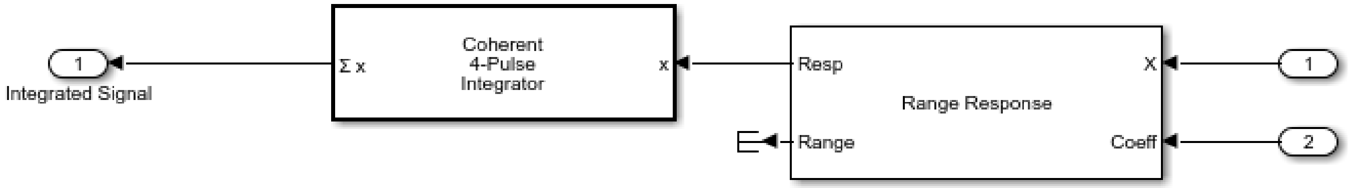
```
helperslexSARSystemSim('showPlatform');
```



Range Processing

This is the first step in obtaining the SAR image. The Range Processing subsystem depicts how we perform 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 sub pulse in the burst is matched filter using the matched coefficients generated from the waveform generation subsystem. Secondly, after matched filtering of the sub pulses they are integrated together to complete the matched filtering of the entire burst. The pulse compressed sub pulses are coherently integrated to obtain the range compressed data.

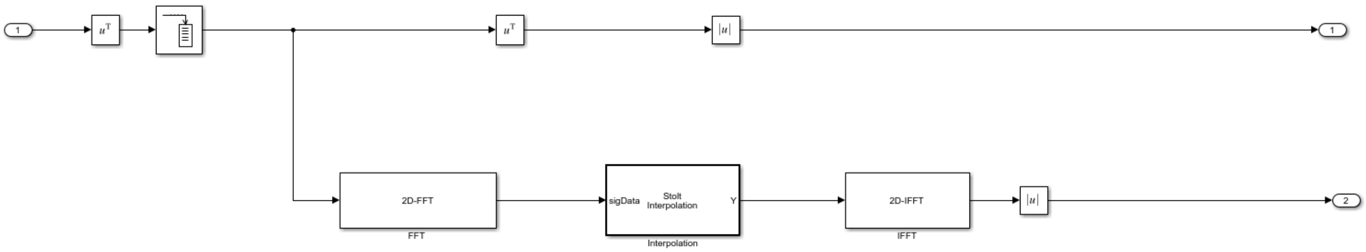
```
helperslexSARSystemSim('showRangeProcessing');
```



Azimuth Processing

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 cross-range/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 on `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 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.

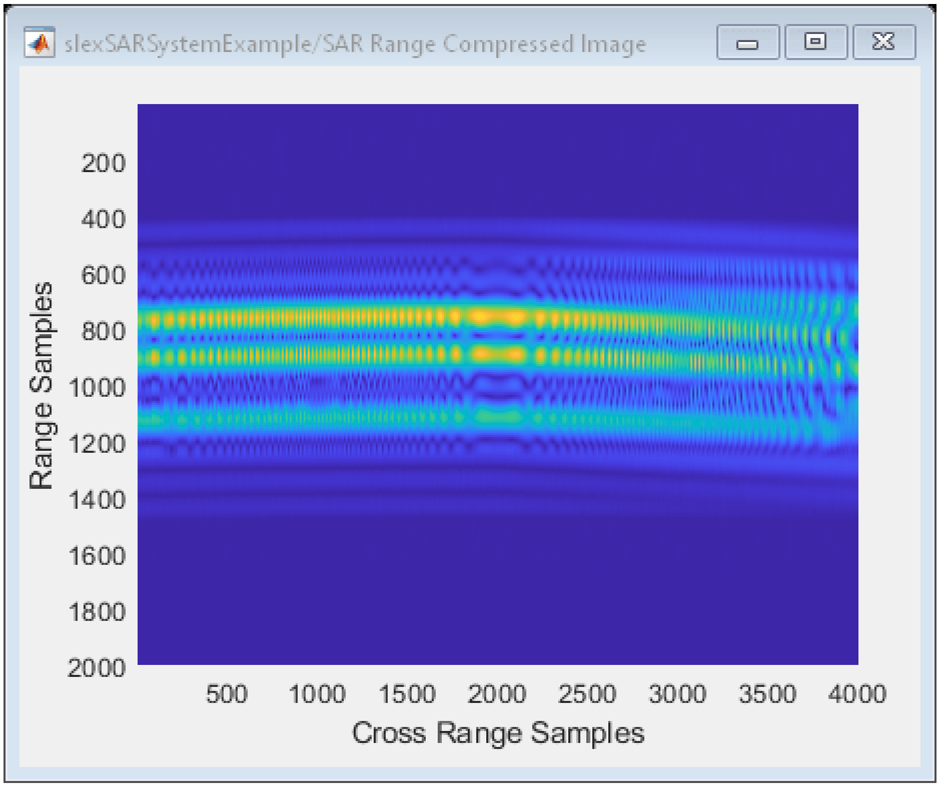
Results and Display

The results below show the processed data at different stages of the simulation.

```
helperslexSARSystemSim('runModel');
```

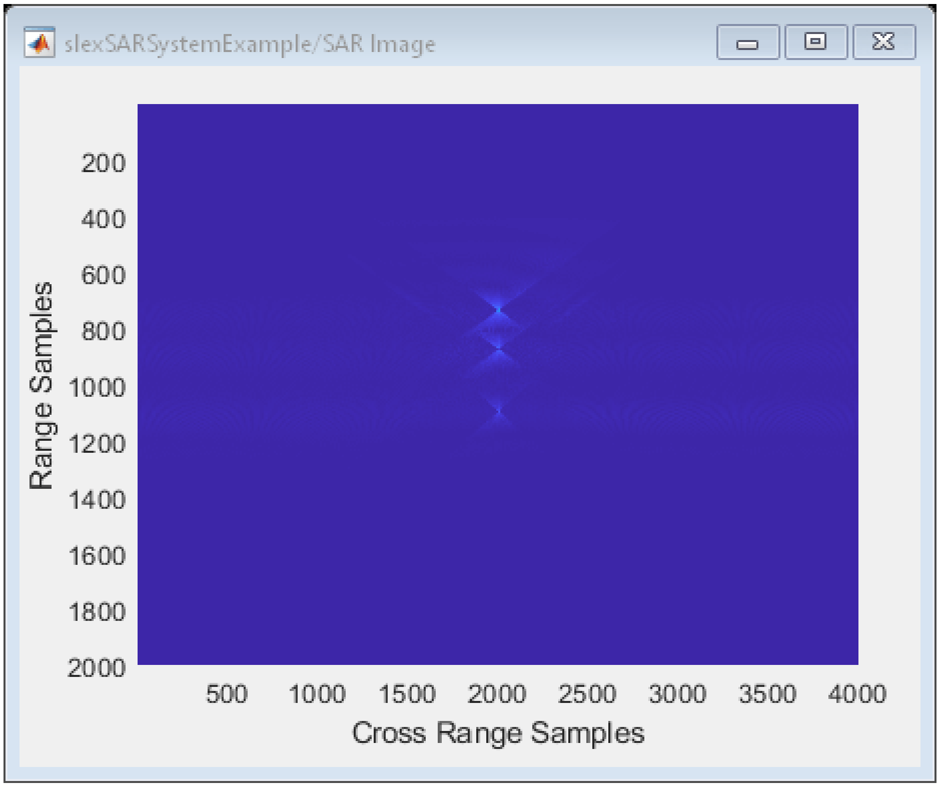
The below image shows data after range processing of the burst waveform.

```
helperslexSARSystemSim('showRangeCompressed');
```



The below image shows SAR image after range and azimuth focusing.

```
helperslexSARSystemSim('showSARImage');
```



Summary

This example exhibits how to use a 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 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.

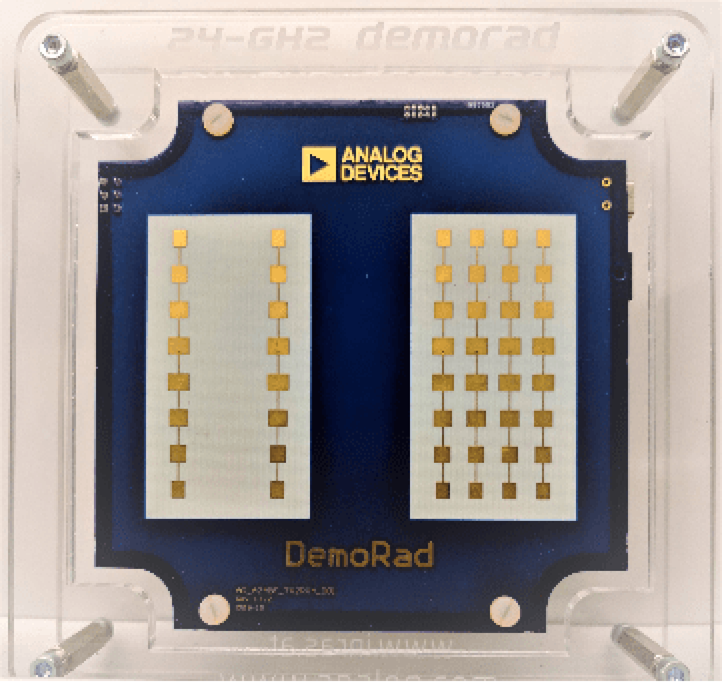
Required Hardware and Software

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 is comprised of 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** 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.

```
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:

```
c0 = physconst('LightSpeed');
wfMetadata = radarSource.Metadata;              % Struct of waveform metadata
bandwidth = wfMetadata.StopFrequency ...
  - wfMetadata.StartFrequency;                  % Chirp bandwidth
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 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
rxElementSpacing = lambda/2;
beamwidth = rad2deg(lambda/ ...
  (radarSource.NumChannels*rxElementSpacing))        % Effective beamwidth
```

```
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

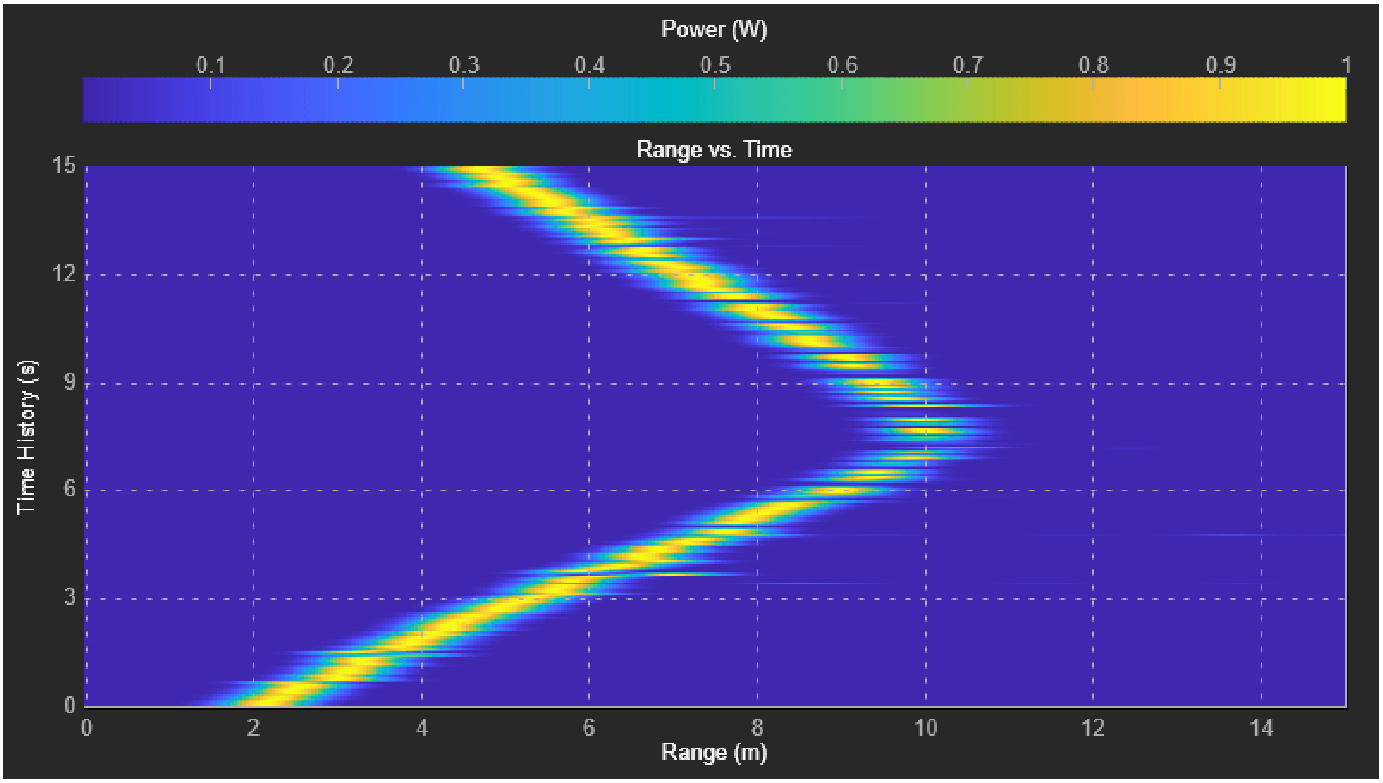
Setup the scopes to view the processed FMCW reflections. We set the viewing window of the range-time intensity scope to 15 seconds.

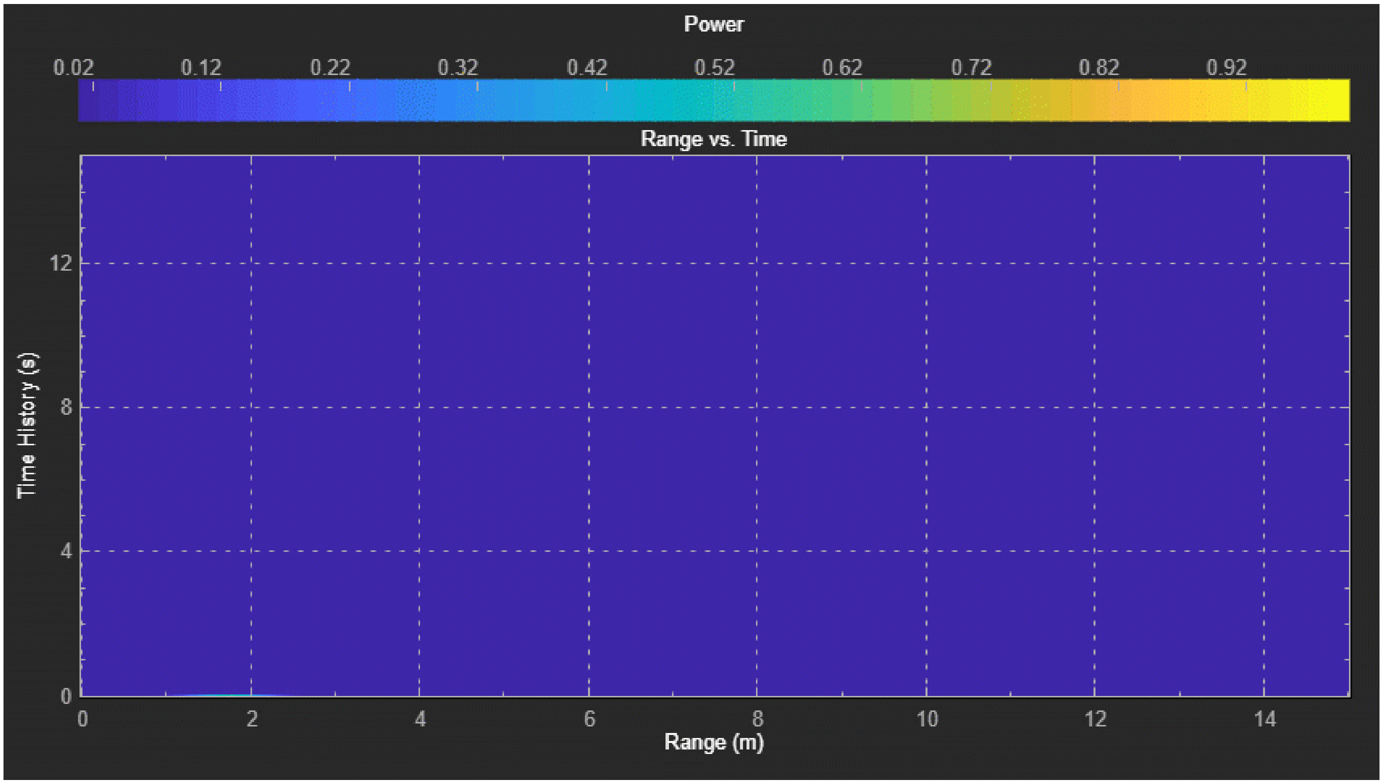
```
timespan = 15;

% 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 to 1 W
  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
```


Power (W)
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Range vs. Time
Time History (s)
15
12
9
6
3
0
Range (m)
0
2
4
6
8
10
12
14


Power
0.02
0.12
0.22
0.32
0.42
0.52
0.62
0.72
0.82
0.92
Range vs. Time
Time History (s)
12
8
4
0
Range (m)
0
2
4
6
8
10
12
14

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. It assumes 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) vs. the probability of false alarm (Pfa) for a given signal-to-noise ratio (SNR).

Introduction

The probability of detection (Pd) is the probability of saying that "1" is true given that event "1" occurred. The probability of false alarm (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.

A detector's performance is measured by its ability to achieve a certain probability of detection and probability of false alarm for a given SNR. Examining a detector's ROC curves provides insight into its performance. We can use the rocsnr function to calculate and plot 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. Assuming we have an SNR value of 8 dB and our requirements dictate a Pfa value of at most 1%, what value of Pd can the detector achieve? We 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
```

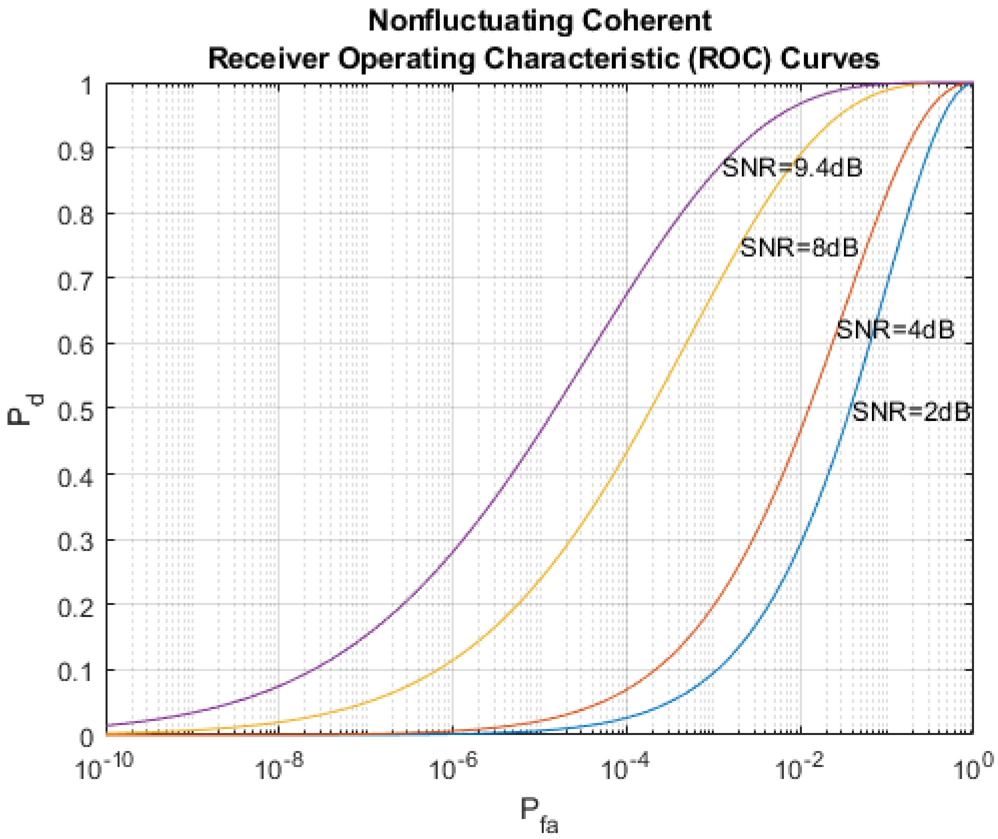
Using the index determined above we can find the Pd value that corresponds to Pfa = 0.01.

```
Pd(idx)
```

```
ans = 0.8899
```

One feature of the rocsnr function is that you can specify a vector of SNR values and rocsnr calculates the ROC curve for each of these SNR values. Instead of individually calculating Pd and Pfa values for a given SNR, we 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);
```

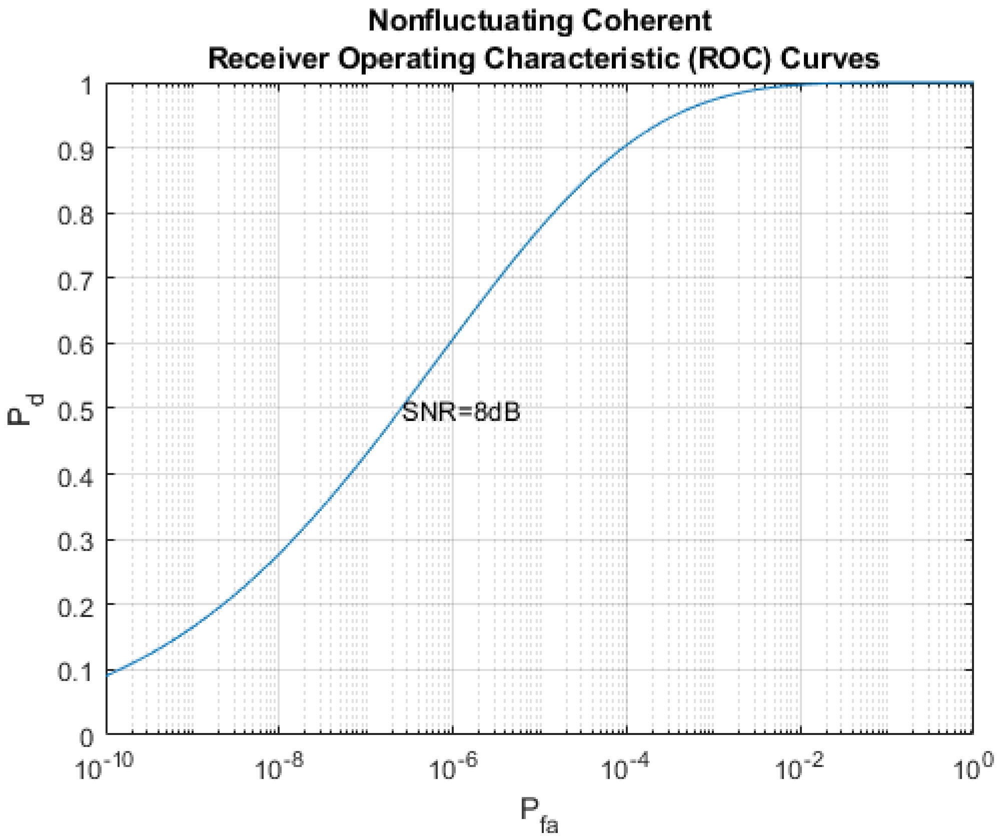


In the plot we can select the data cursor button in the toolbar (or in the Tools menu) and then select the SNR = 8 dB curve at the point where Pd = 0.9 to verify that Pfa is approximately 0.01.

Multiple Pulse Detection

One way to improve a detector's 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. Let's continue our example by assuming an SNR of 8 dB and averaging over two pulses.

```
rocsnr(8,'NumPulses',2);
```

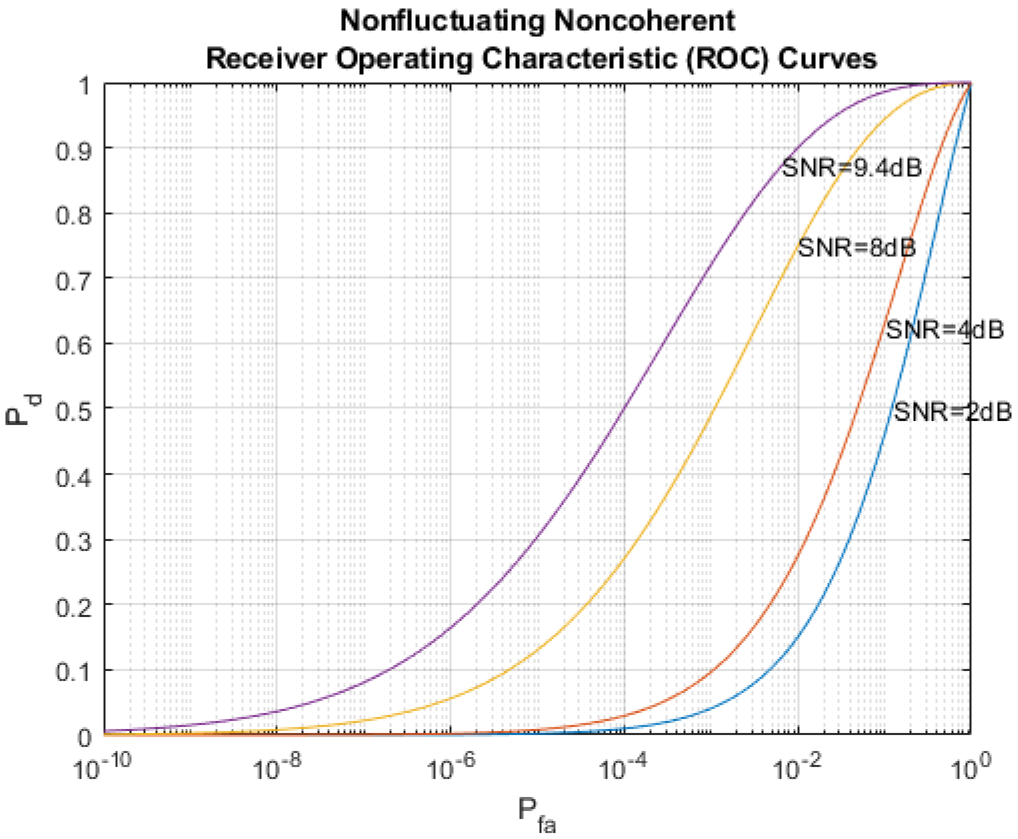


By inspecting the plot we can see that averaging over two pulses resulted in a higher probability of detection for a given false alarm rate. With an SNR of 8 dB and averaging over two pulses, you can constrain the probability of false alarm to be at most 0.0001 and achieve a probability of detection of 0.9. Recall that for a single pulse, we 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, we have assumed we were dealing 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, let's analyze the performance of a noncoherent detector.

```
rocsnr(SNRvals,'SignalType','NonfluctuatingNoncoherent');
```



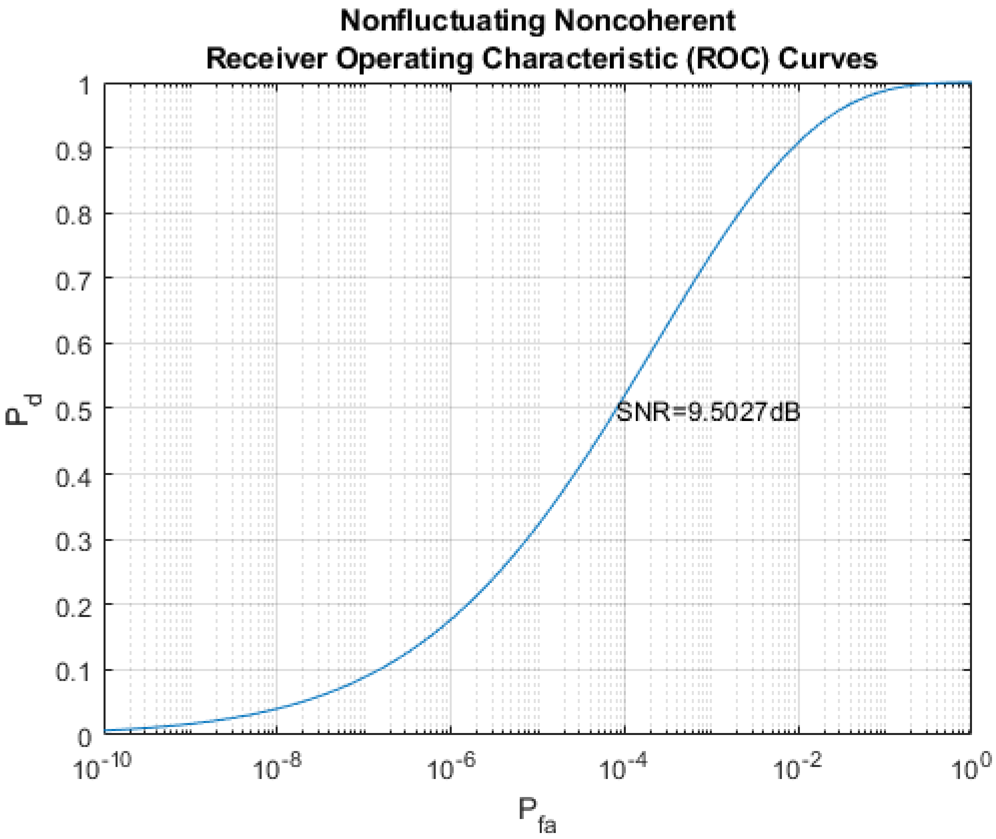
Focus on the ROC curve corresponding to an SNR of 8dB. By inspecting the graph with the data cursor, you can see that to achieve a probability of detection of 0.9, you must tolerate a false-alarm probability of up to 0.05. Without using phase information, we need a higher SNR to achieve the same Pd for a given Pfa. For noncoherent linear detectors, we can use Albersheim's equation to determine what value of SNR will achieve our 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, we 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 target's statistical characteristics do not change over time. However, in real scenarios, targets can accelerate and decelerate as well as roll and pitch. These factors cause the target's radar cross section (RCS) to vary over time. A set of statistical models called Swerling models are often used to describe the random variation in target RCS.

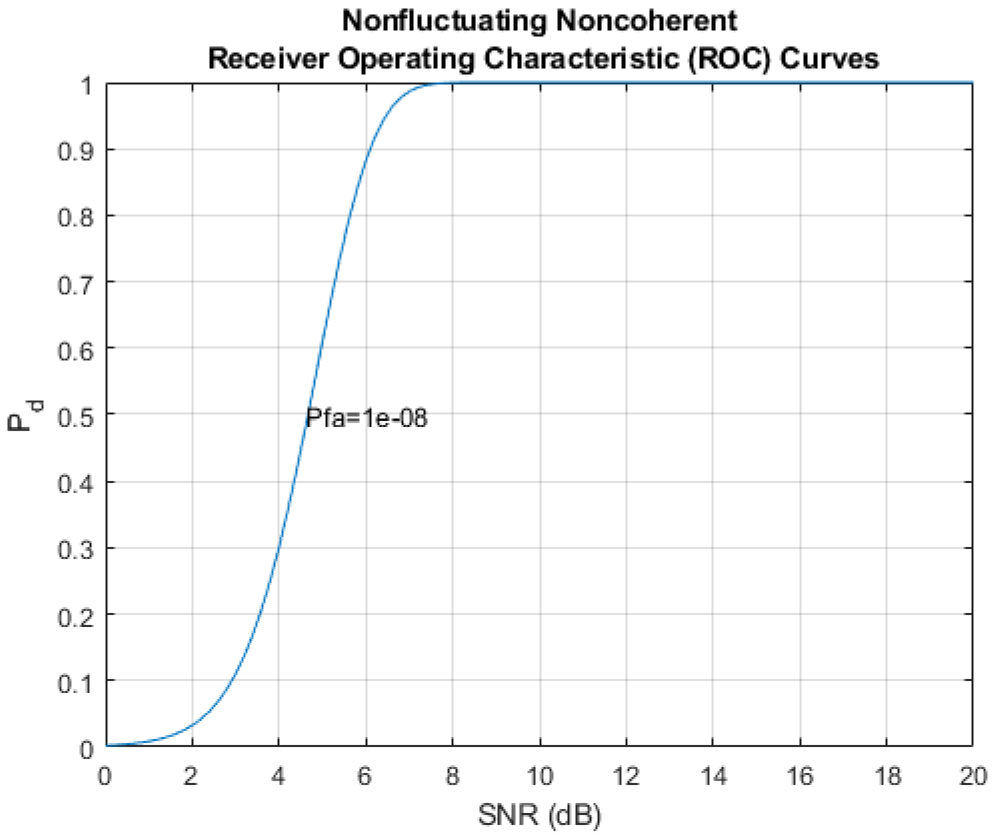
There are four Swerling models, namely Swerling 1-4. The nonfluctuating target is often termed either Swerling 0 or Swerling 5. Each Swerling model describes how a target's RCS varies over time and the probability distribution of the variation.

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.

Let us now compare the ROC curves for a nonfluctuating target and a Swerling 1 target. In particular, we want to explore what the SNR requirements are for both situations if we 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 ROC curve in this form.

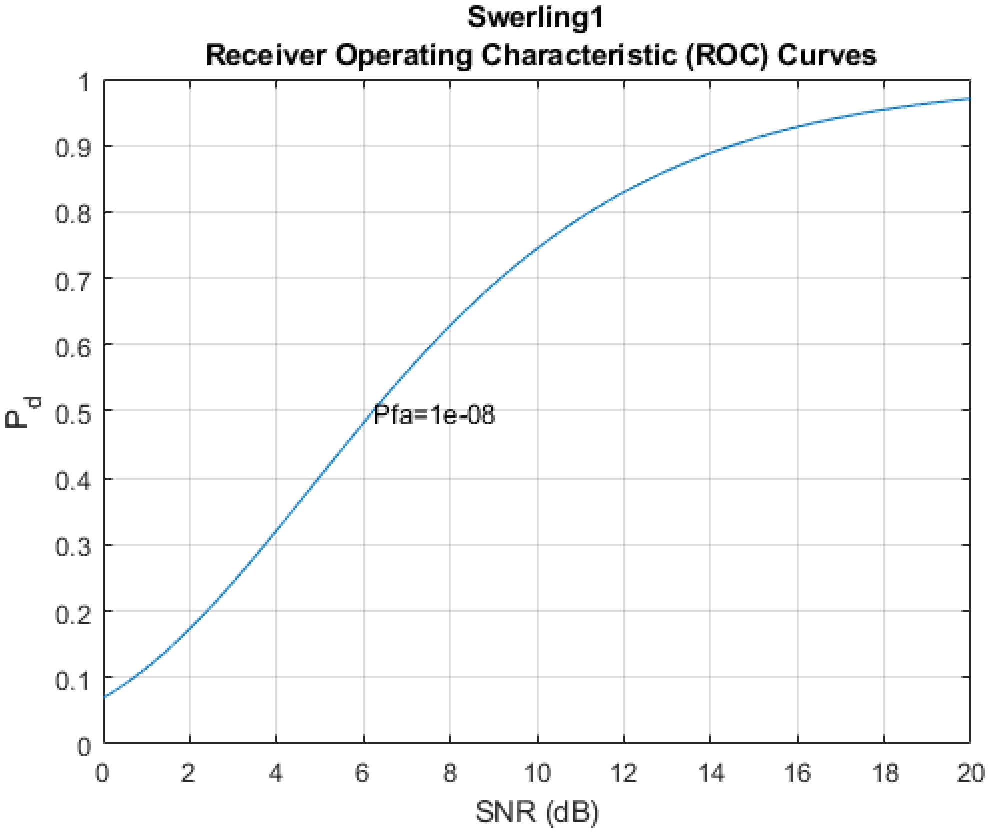
Let us assume that we are doing noncoherent detection with 10 integrated pulses, with the desired Pfa being at most 1e-8. We first plot the ROC curve for a nonfluctuating target.

```
rocpfa(1e-8,'NumPulses',10,'SignalType','NonfluctuatingNoncoherent')
```



We then plot the ROC curve for a Swerling 1 target for comparison.

```
rocpfa(1e-8,'NumPulses',10,'SignalType','Swerling1')
```



From the figures, we can see that for a Pd of 0.9, we require 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 impact the design of the system.

As in the case of nonfluctuating targets, we 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 we 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. We used the rocsnr function to analyze the effectiveness of a linear detector for various SNR values. We also reviewed the improvement in detector performance achieved by averaging multiple samples. Lastly we showed how we 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 = 293.1830
```

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 objec™ 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.

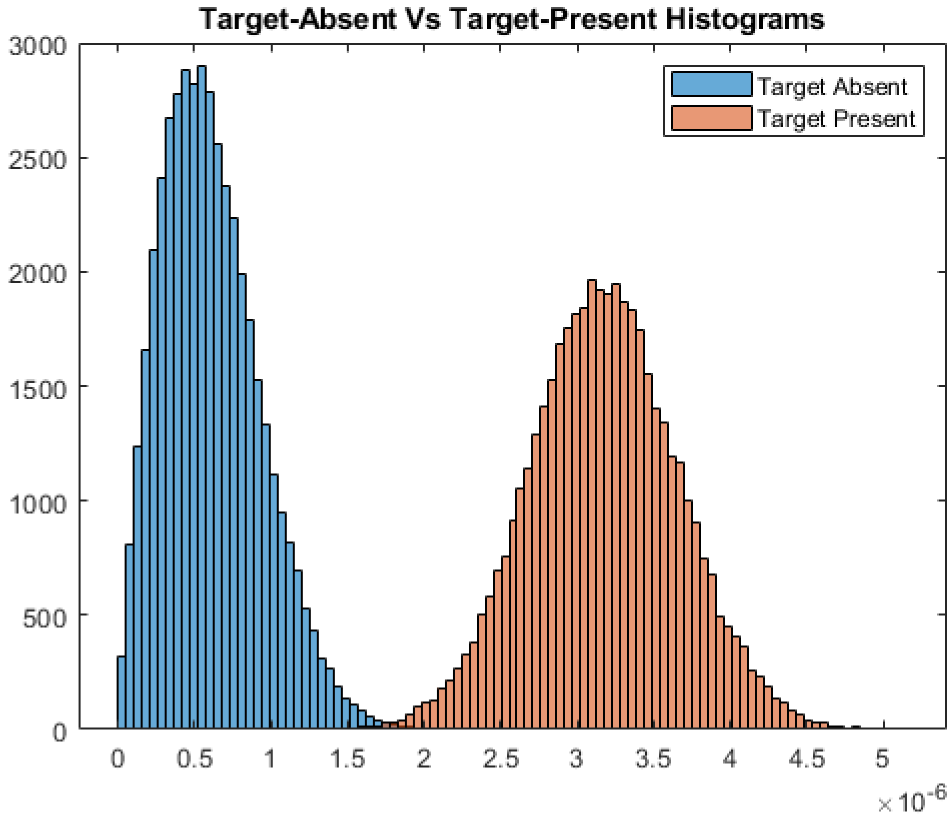
1. Radiate the signal into space using `phased.Radiator`.
2. Propagate the signal to the target and back to the antenna using `phased.FreeSpace`.
3. Reflect the signal from the target using `phased.Target`.
4. Receive the reflected signals at the antenna using `phased.Collector`.
5. Pass the received 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.

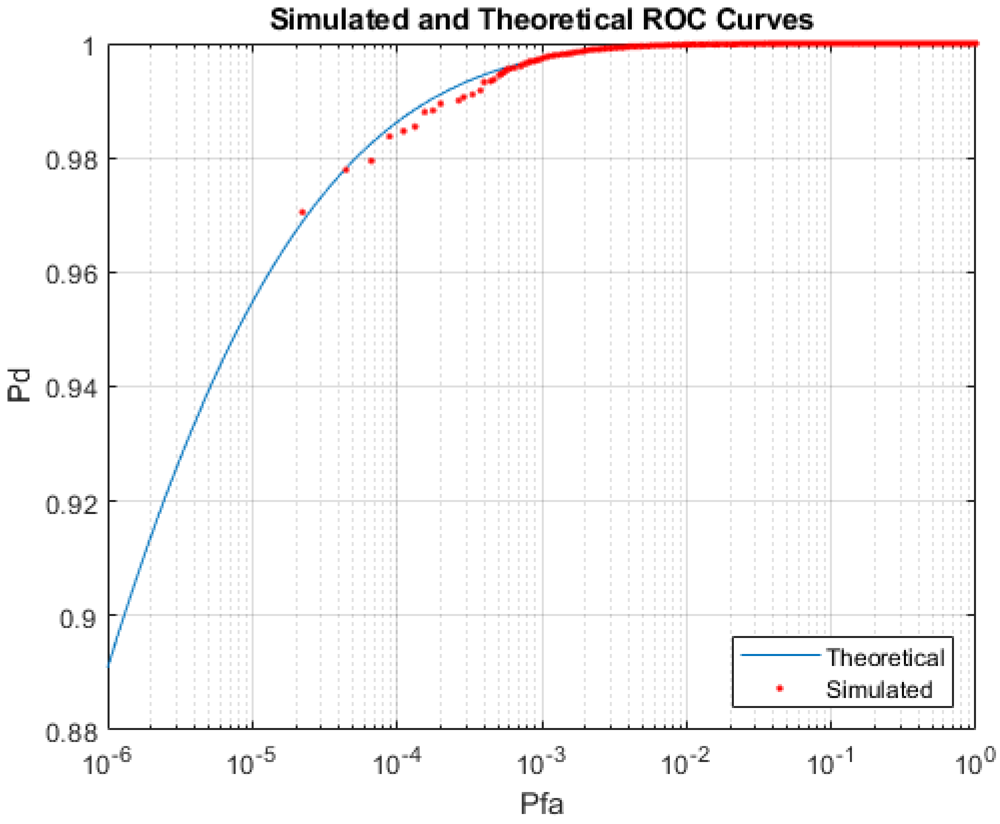
```
pfa_diff = diff(sim_pfa);
idx = (pfa_diff == 0);
sim_pfa(idx) = [];
sim_pd(idx) = [];
```

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)).';
```

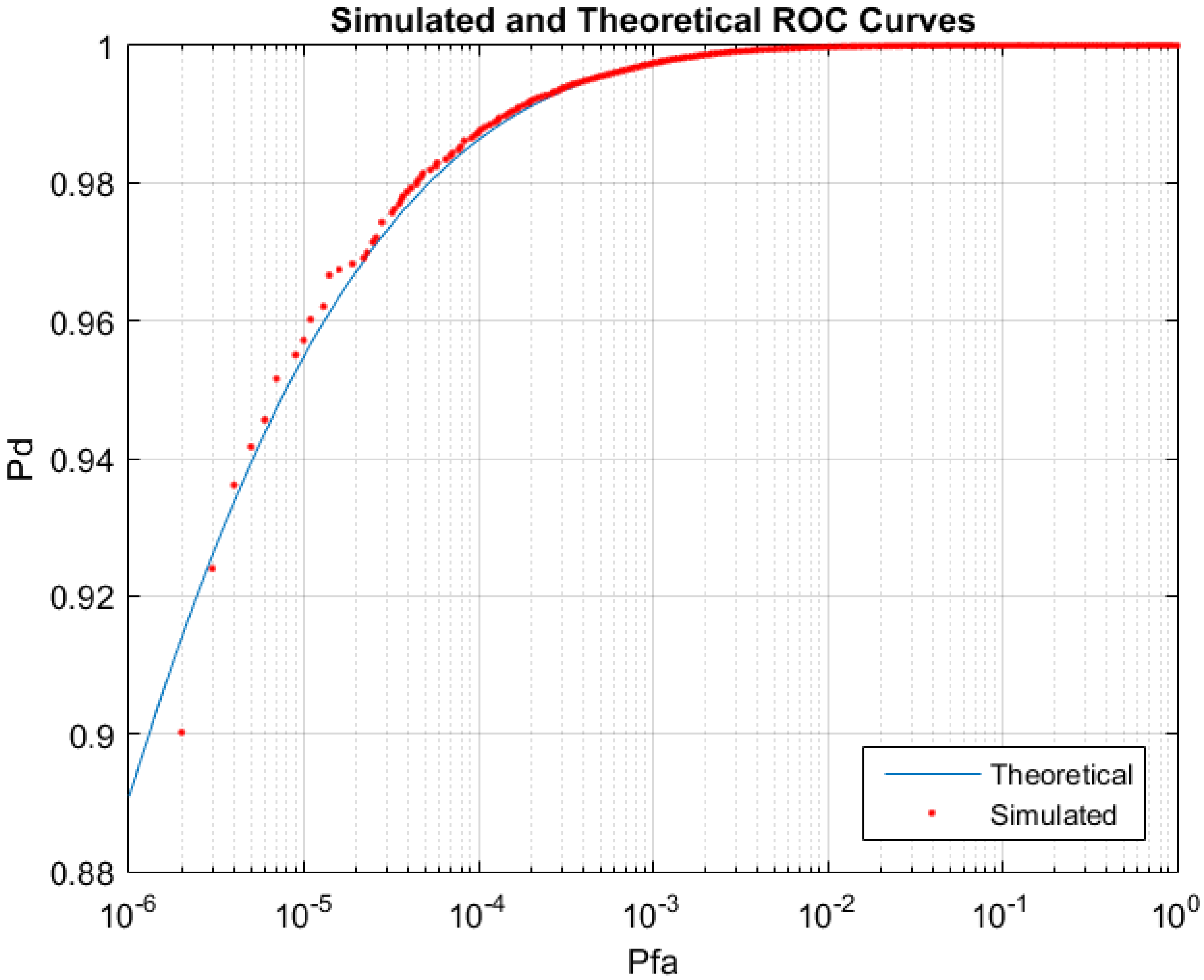
Compute the theoretical *Pfa* and *Pd* values from the smallest *Pfa* to 1. Then plot the theoretical *Pfa* curve.

```
[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 = $P_{\mathrm{fa}} = 0$)
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.

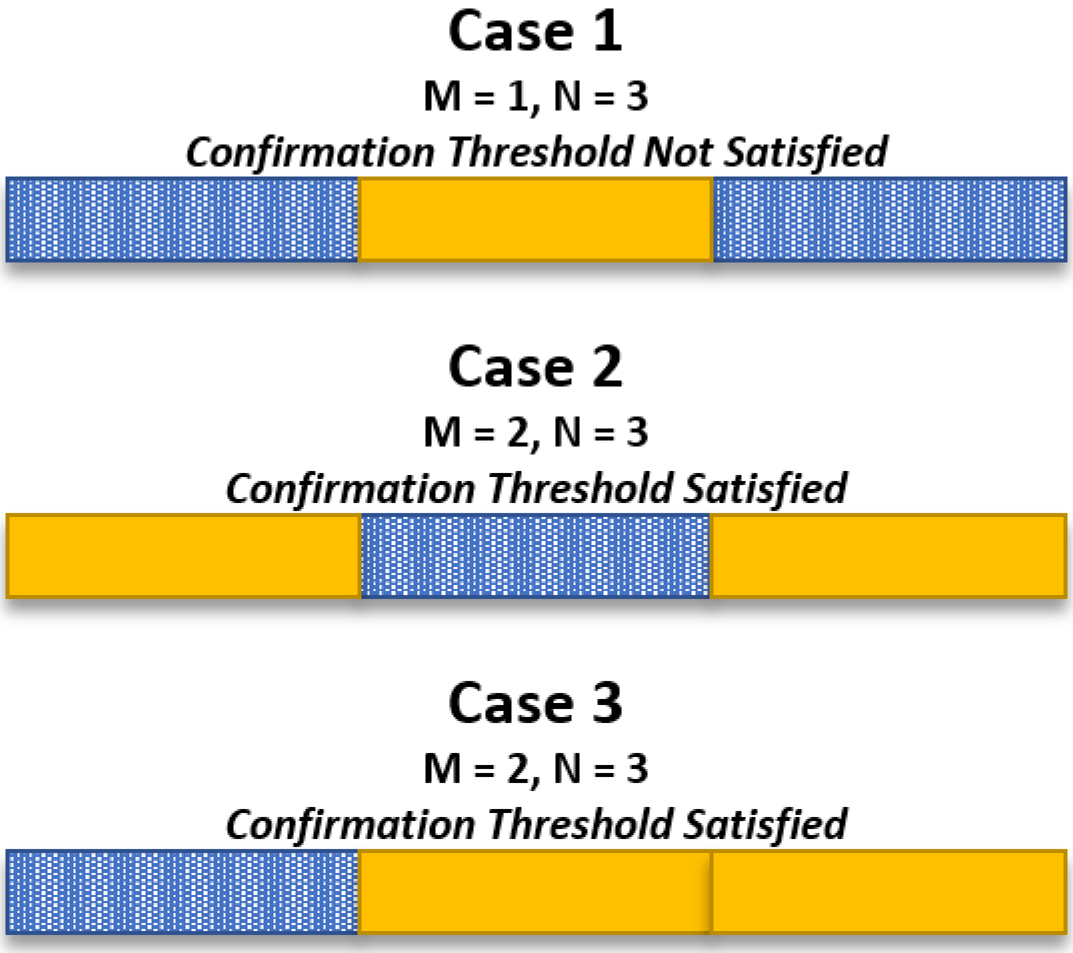
For a confirmation threshold logic of M-of-N, the target track probability P_{dt} is defined as

$$P_{\mathrm{dt}} = \sum_{i = M}^{N} C(N, i) P_{\mathrm{d}}^{i} (1 - P_{\mathrm{d}})^{(N - i)},$$

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

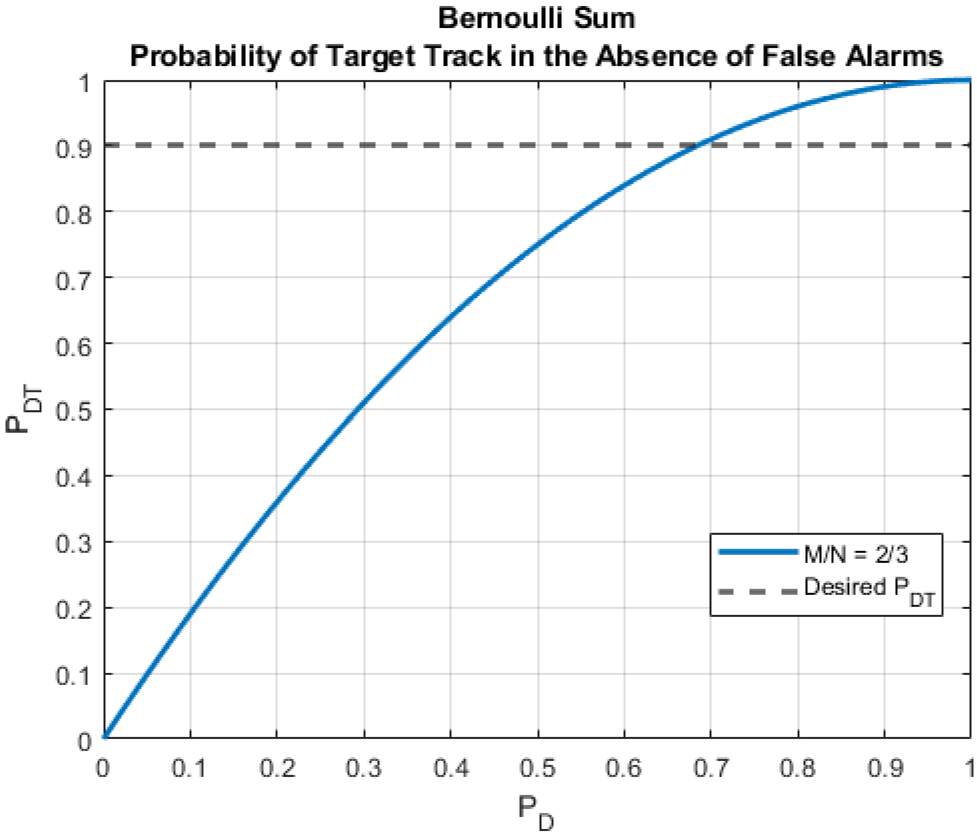
Investigate the probability of target track versus probability of detection for a confirmation threshold of 2/3 using the Bernoulli sum method. Perform the Bernoulli sum calculation, assuming that one hit is required to initialize a track.

```
% 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 0.0374 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

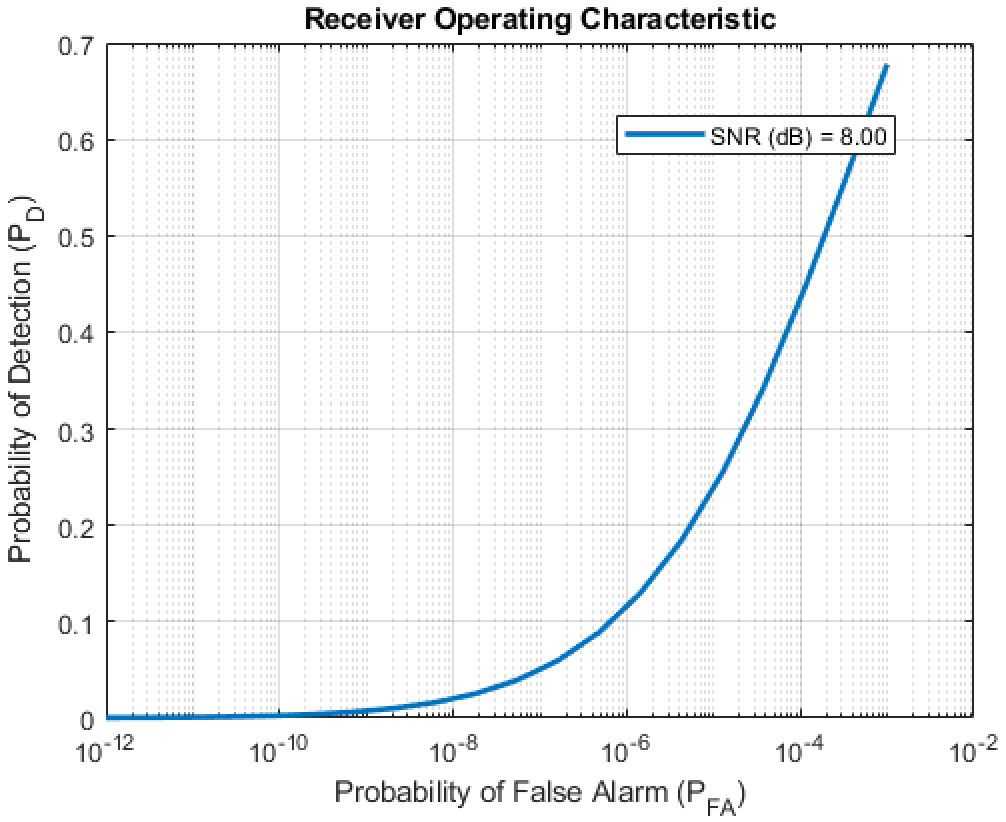
The CGH algorithm can be used to calculate all four probability types:

- Probability of a single target track in the absence of false alarms ($P_{\mathrm{fa}} = 0$)
- Probability of a single false track in the absence of targets ($P_{\mathrm{d}} = 0$)
- 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.

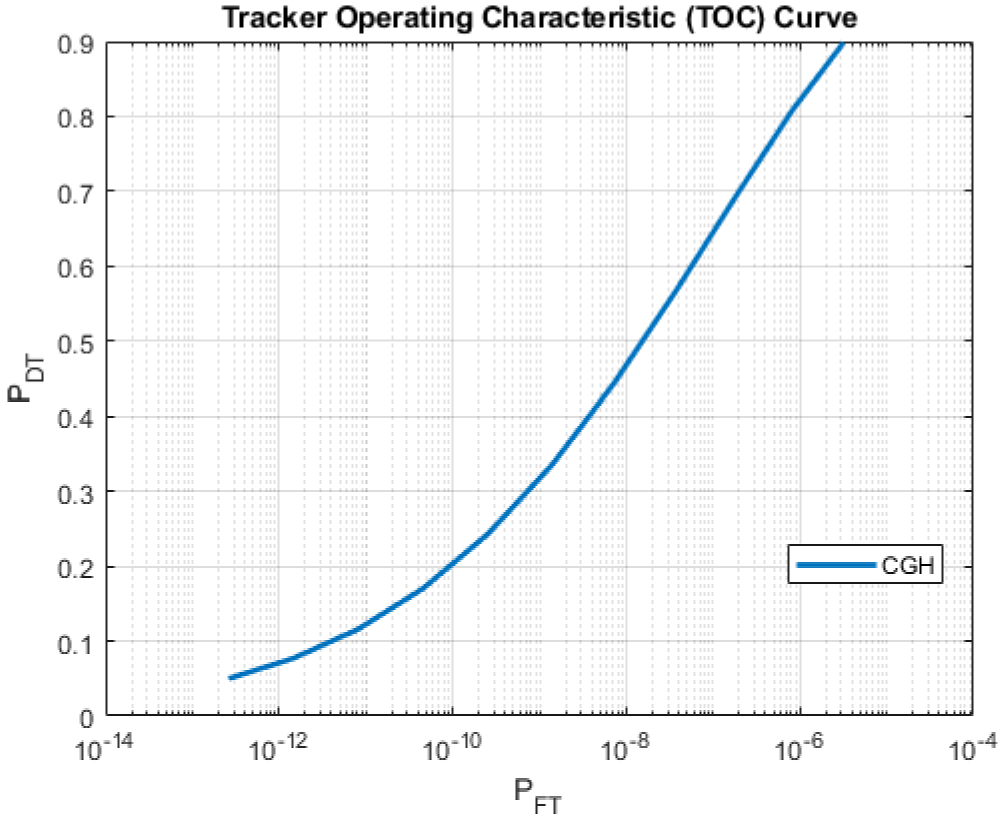
```
% Receiver operating characteristic (ROC)
snrdB = 8; % SNR (dB)
[Pd,Pfa] = rocsnr(snrdB,'MaxPfa',1e-3,'MinPfa',1e-12,'NumPoints',20);

% 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 0.8056 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

$$Q = \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 \end{bmatrix} q,$$

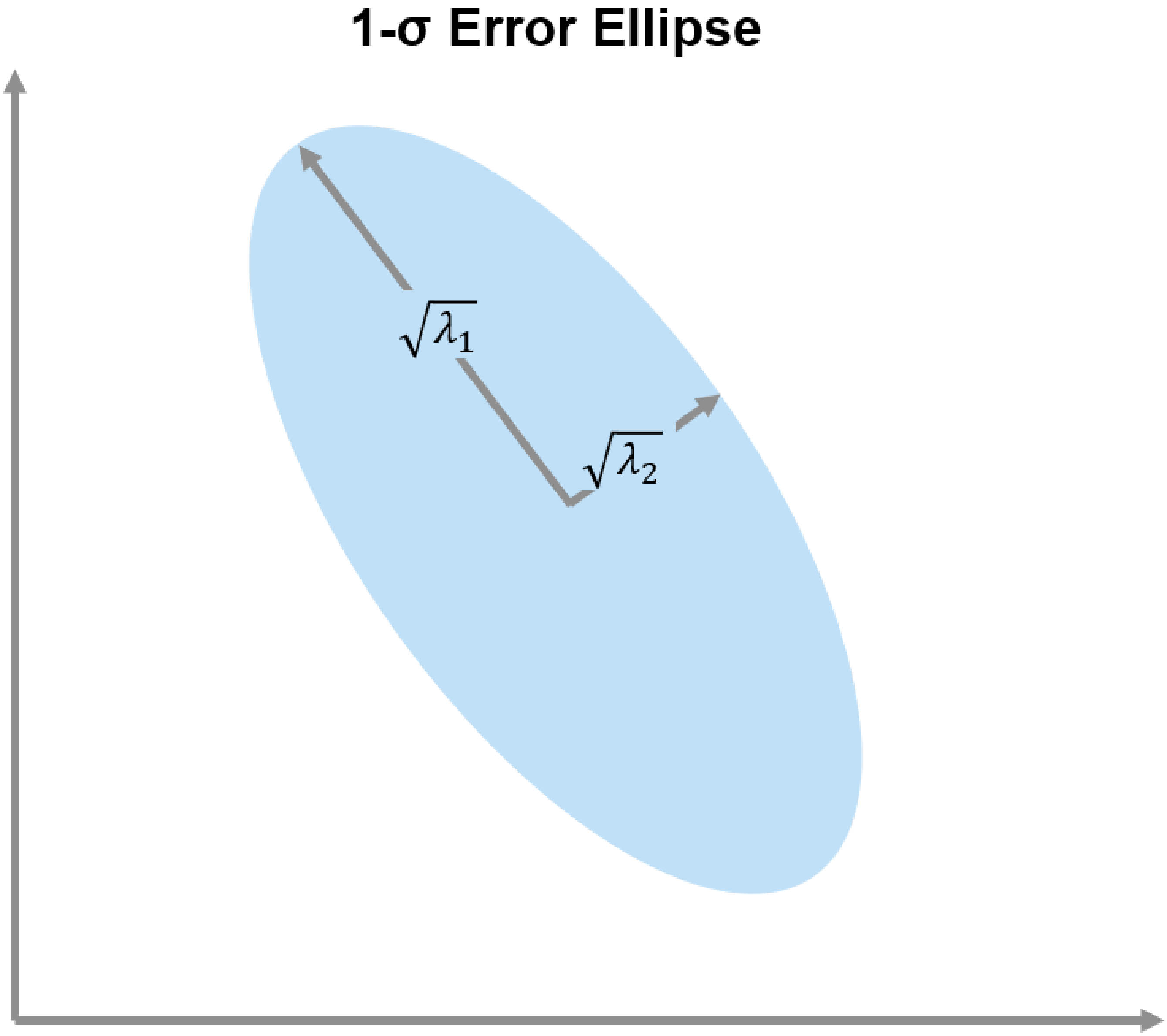
where q is a tuning factor defined as

$$q = a_{\max}^2 \Delta t.$$

The input $a_{\max}^2$is the maximum target acceleration expected. Assume that a maximum acceleration of 4 m/s2 is expected for vehicles.

```
% 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.



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

$\text{Error Ellipse Area} = \pi\sqrt{\lambda_1}\sqrt{\lambda_2}$.

The area of the bins is calculated as

$\text{Bin Area} = (\Delta\text{Range})(\Delta\text{Range Rate})$.

Finally, the gate size in bins is

$$\text{Gate Size} = \frac{\text{Error Ellipse Area}}{\text{Bin Area}}.$$

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.

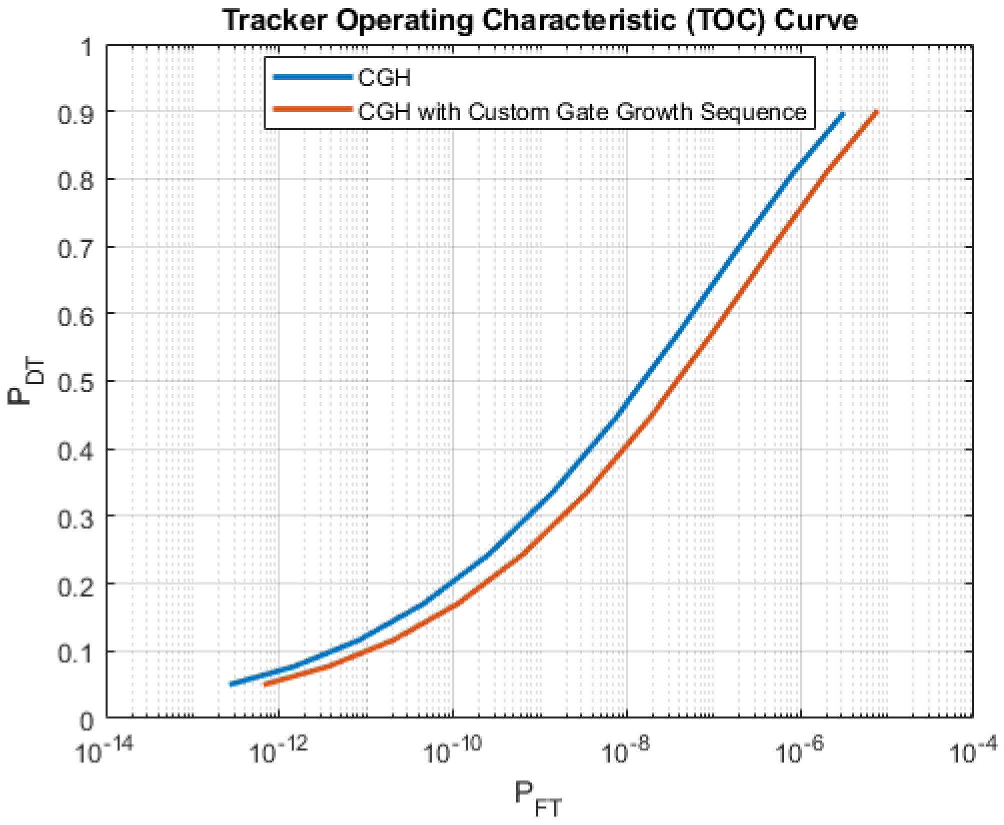
```
% Calculate gate growth sequence
res = [1, 1]; % Bin resolutions [range (m), range-rate (m/s)]
gs = helperCalculateGateSize(N,trkfilt,res)
```

```
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 0.1015 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.

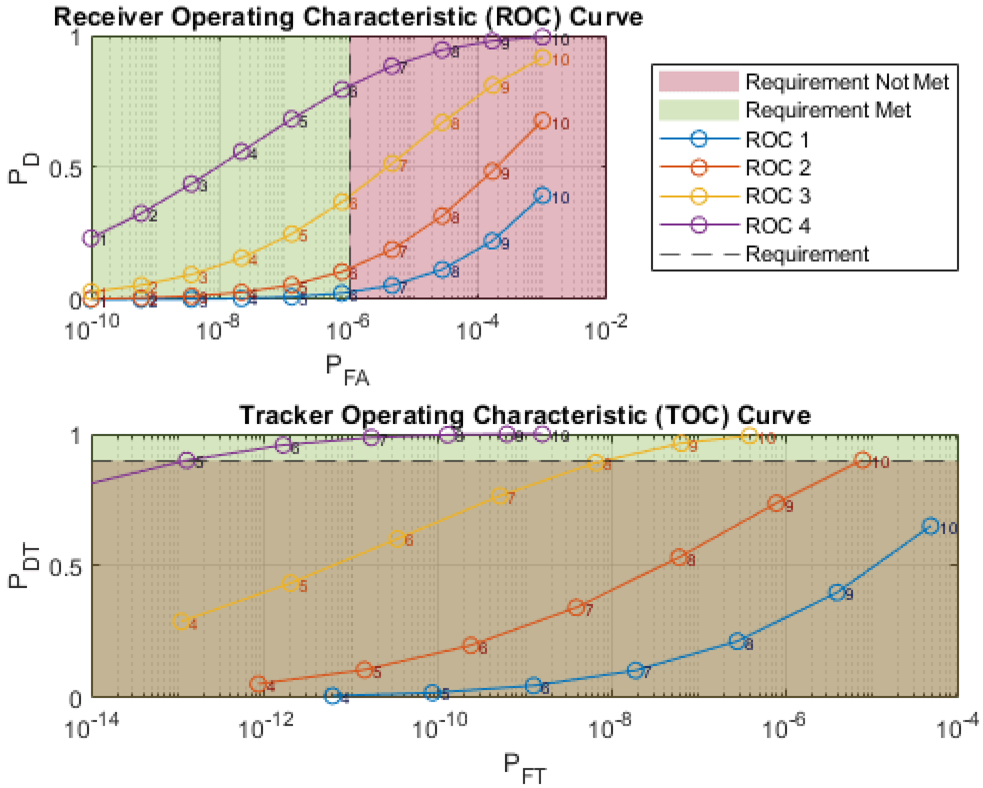
```
% Calculate ROC curves
snrdB = [6 8 10 12]; % SNR (dB)
[Pd,Pfa] = rocsnr(snrdB,'MaxPfa',1e-3,'MinPfa',1e-10,'NumPoints',10);
```

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.

```
% 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 0.8796 seconds.
```

```
% 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)
```



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

$$E_{\mathrm{ft}} = P_{\mathrm{ft,\,nt}}N_{\mathrm{c}} + P_{\mathrm{ft}}N_{\mathrm{t}},$$

where $P_{\mathrm{ft,\,nt}}$ 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

$$\text{Number of Cells} = (\text{Number of Range Cells}) \times (\text{Number of Range Rate Cells}) = 1000 \times 100 = 10^5 .$$

```
% 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.

```
% 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);
```

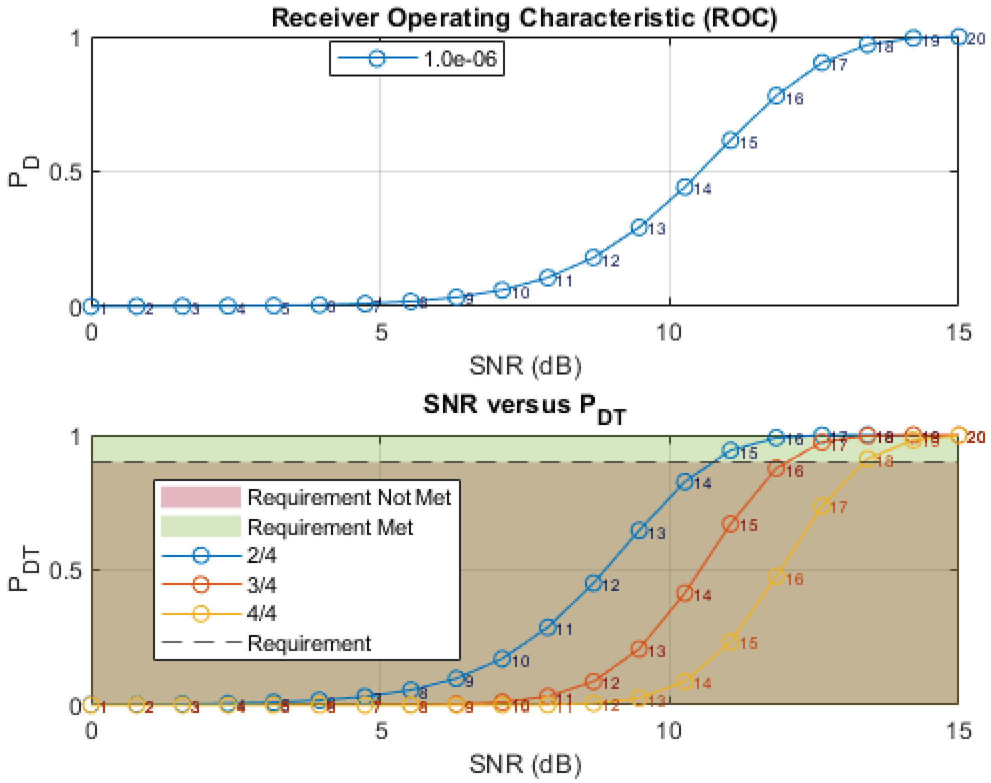
Update the gate growth sequence due to the greater number of observations.

```
% 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.
% 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{:});
    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
```

```
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
```

```
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
```

```
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.

```
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.

```
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);
```

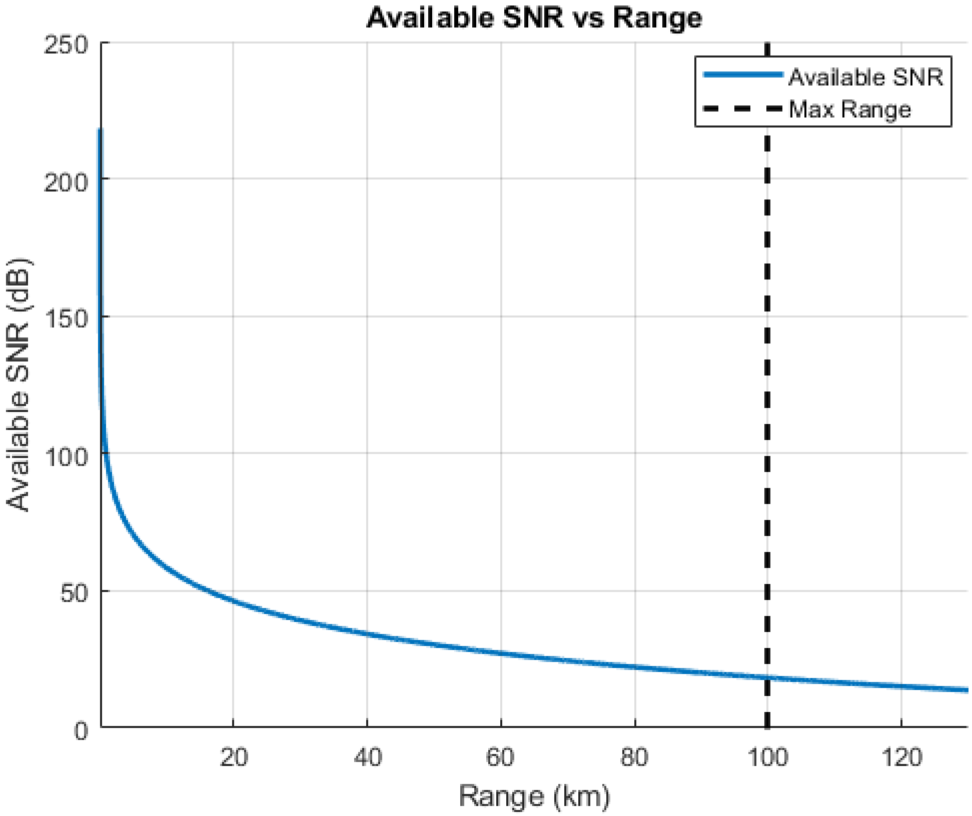
Calculate the available SNR at the required maximum range of 100 km.

```
SNRatRm = SNR(find(R>=Rm,1))
```

```
SNRatRm = 18.3169
```

Plot the maximum range requirement together with the computed available SNR.

```
radarmetricplot(R*1e-3,SNR,'MetricName','Available SNR','MaxRangeRequirement',Rm*1e-3,'RangeUnit
legend('Location','best');
```



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 m^2 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
```

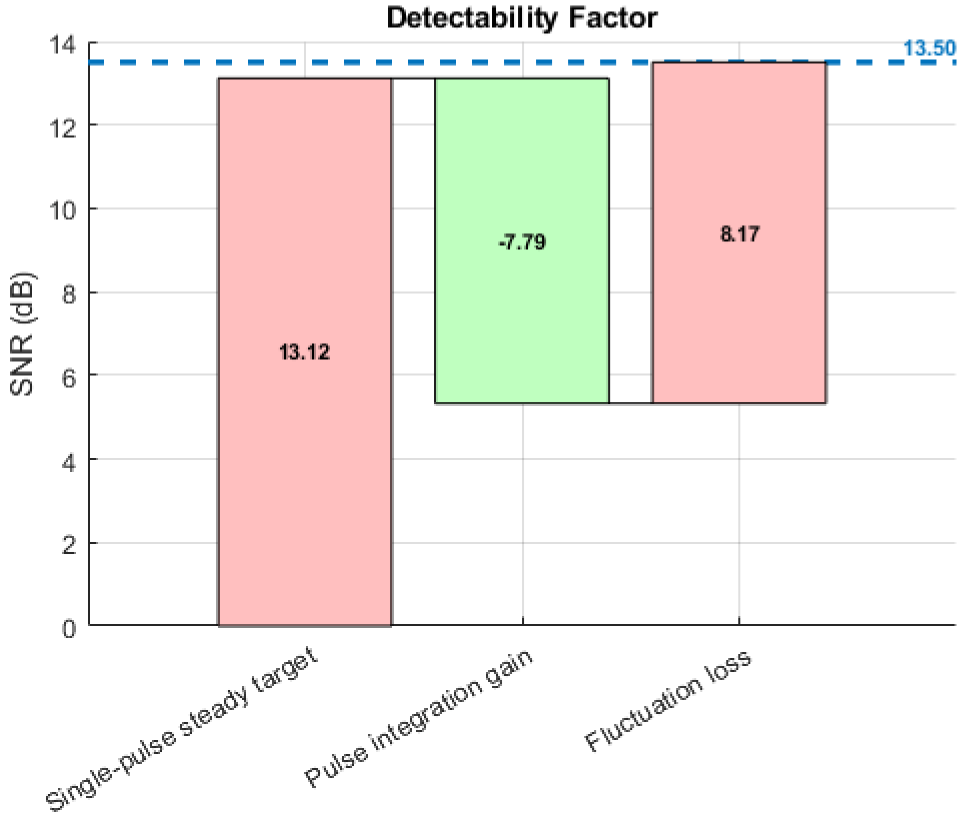
The fluctuation loss is a difference between the SNR required to detect a fluctuating target and the SNR required to detect a steady target.

```
Lf = detectability(Pd,Pfa,N,'Swerling1') - detectability(Pd,Pfa,N,'Swerling0')
```

```
Lf = 8.1696
```

Use a waterfall chart to illustrate the components of the detectability factor.

```
helperDetectabilityWaterfallPlot([D0 -Gi Lf], {'Single-pulse steady target','Pulse integration ga
```



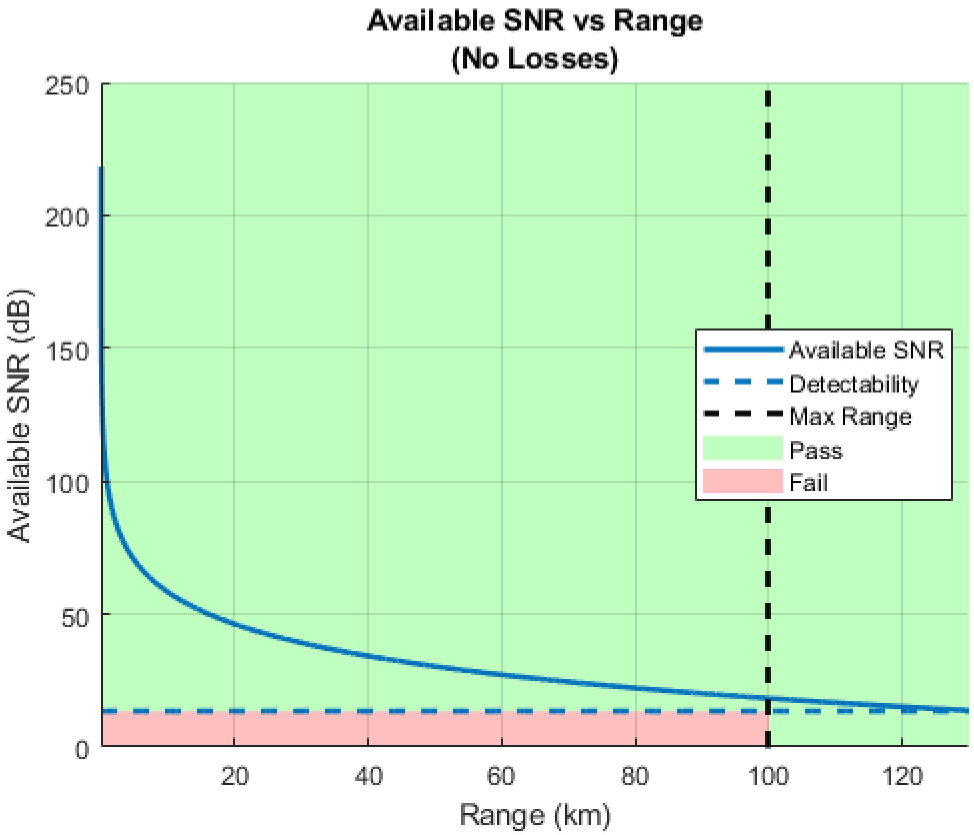
Substitute the detectability factor into the range form of the radar equation as the minimum required SNR to evaluate the actual maximum range of the system.

```
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);
title([{'Available SNR vs Range'}, {'(No Losses)'}]);
legend('Location','best');
```



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)
```

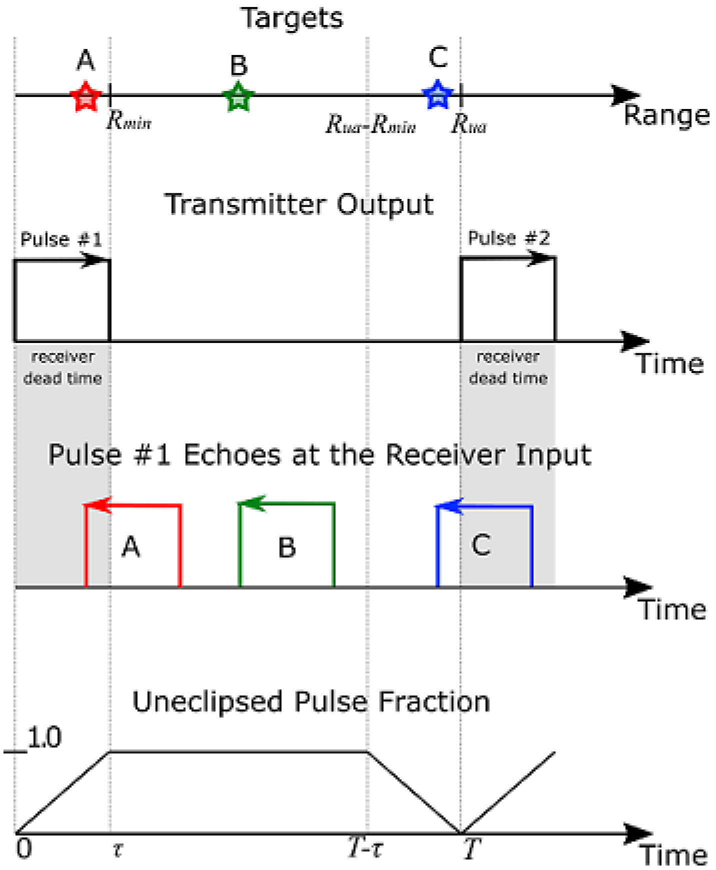
```
Rmin = 1.6489e+03
```

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 ≈0.75 ms), calculate the unambiguous range of the system.

```
prf = 1350;                          % Pulse repetition frequency
Rua = time2range(1/prf)
```

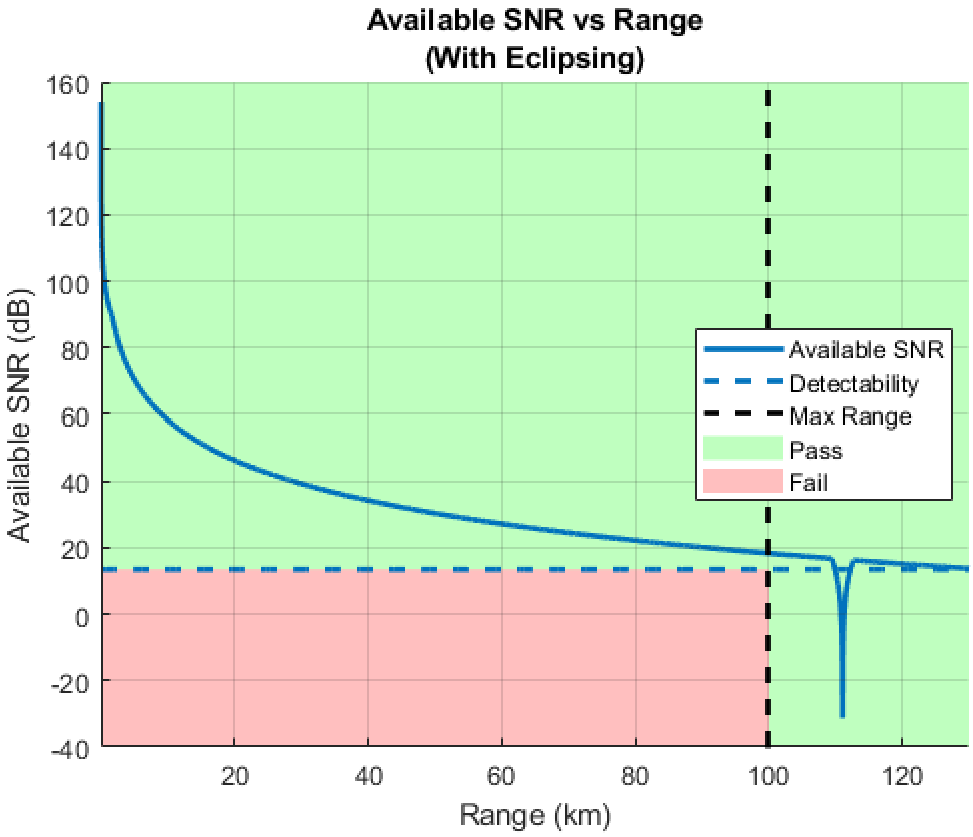
```
Rua = 1.1103e+05
```

The echoes arriving from the ranges $R_{ua} \pm R_{min}$will be eclipsed by the next transmitted pulse. This diagram llustrates 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);
title([{'Available SNR vs Range'}, {'(With Eclipsing)'}]);
legend('Location','best');
```



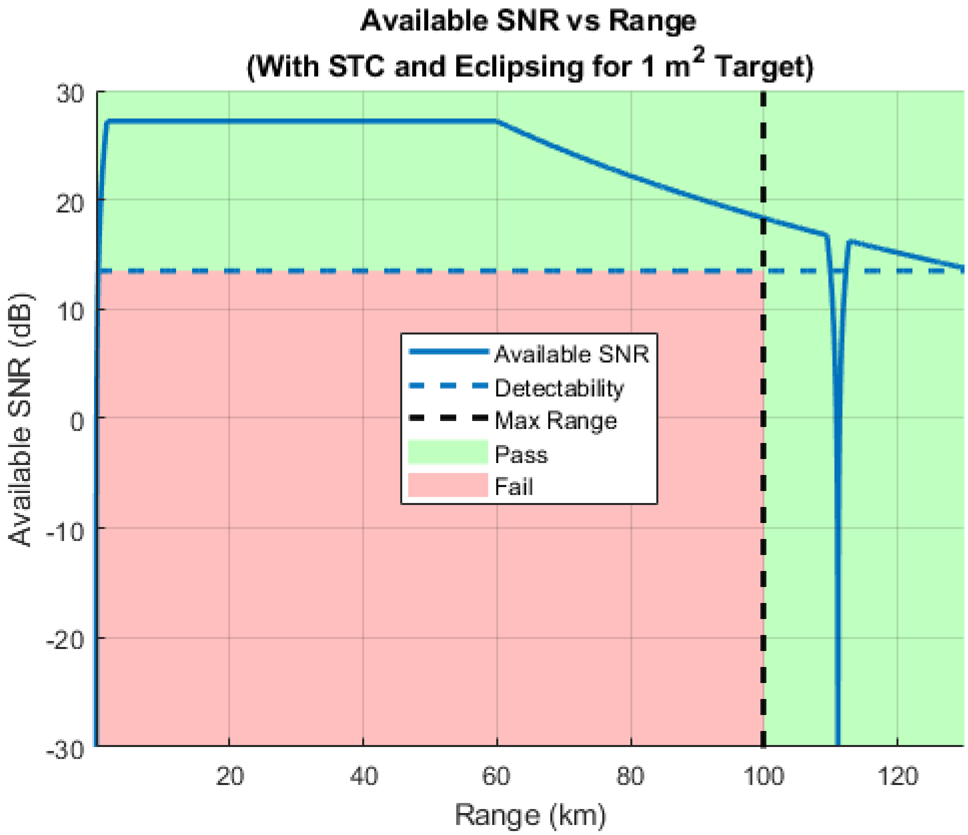
The real-world radar systems utilize PRF diversity in order to prevent the eclipsing loss and to extend the unambiguous range of the system.

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.

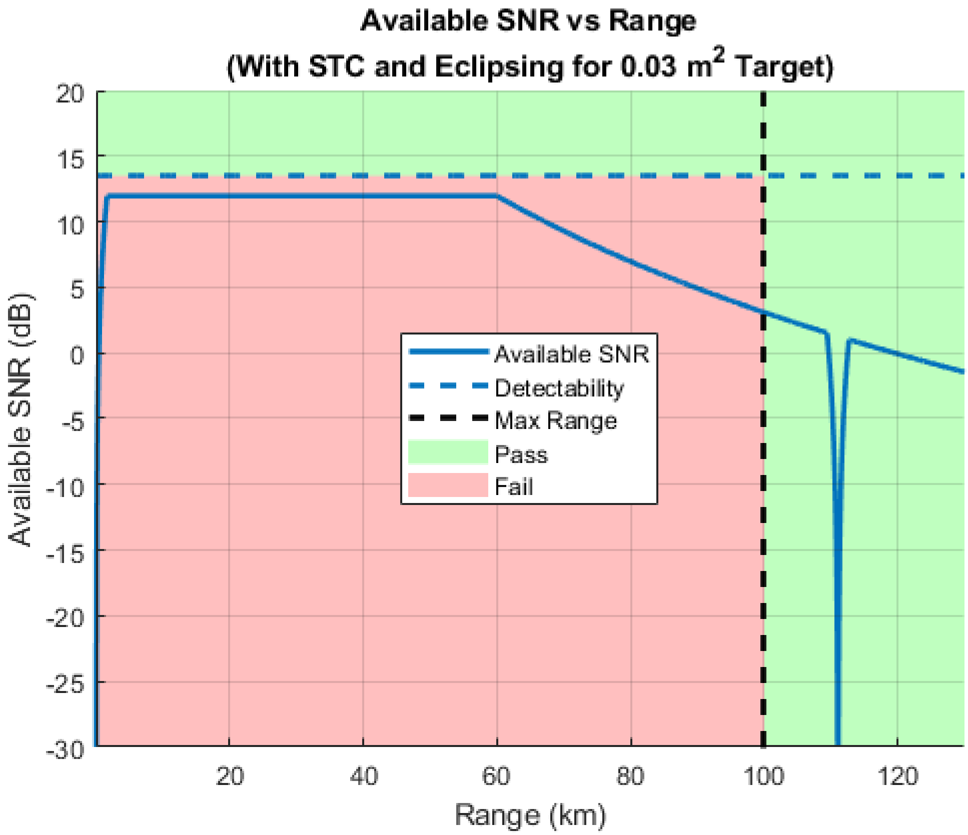
```
Rstc = 60e3;                      % STC cutoff range (m)
Xstc = 4;                         % STC exponent selected to maintain target detectability at
Fstc = stcfactor(R,Rstc,Xstc);    % STC factor

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);
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);
title([{'Available SNR vs Range'}, {'(With STC and Eclipsing for 0.03 m^2 Target)'}]);
legend('Location','best');
ylim([-30 20])
```



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.

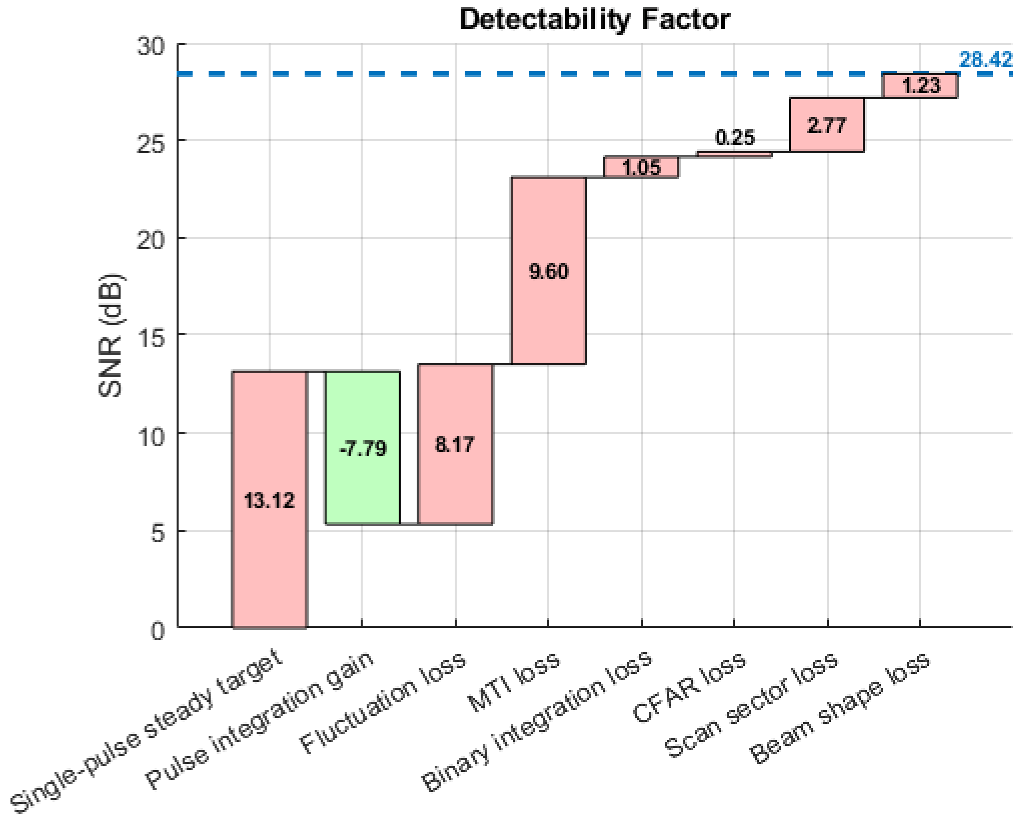
```
Nrc = 120;
Lcfar = cfarloss(Pfa,Nrc)
```

```
Lcfar = 0.2500
```

Effective Detectability Factor

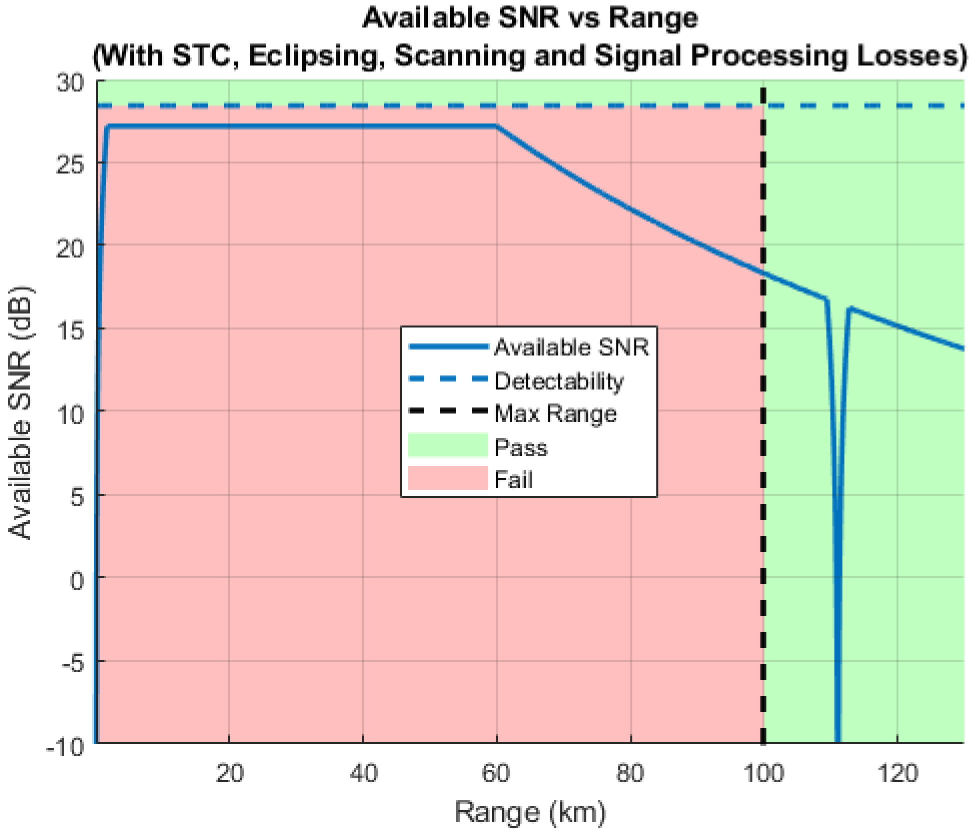
The scanning and the signal processing losses increase the detectability factor, which means that more energy is required to make a detection. The resulting detectability factor that includes effects of all these losses is called the effective detectability factor. The waterfall chart shows the combined 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];
helperDetectabilityWaterfallPlot(D, {'Single-pulse steady target','Pulse integration gain','Fluc
    'MTI loss', 'Binary integration loss', 'CFAR loss', 'Scan sector loss', 'Beam shape loss'});
```



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);
title([{'Available SNR vs Range'}, {'(With STC, Eclipsing, Scanning and Signal Processing Losses
legend('Location','best')
ylim([-10 30]);
```



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.

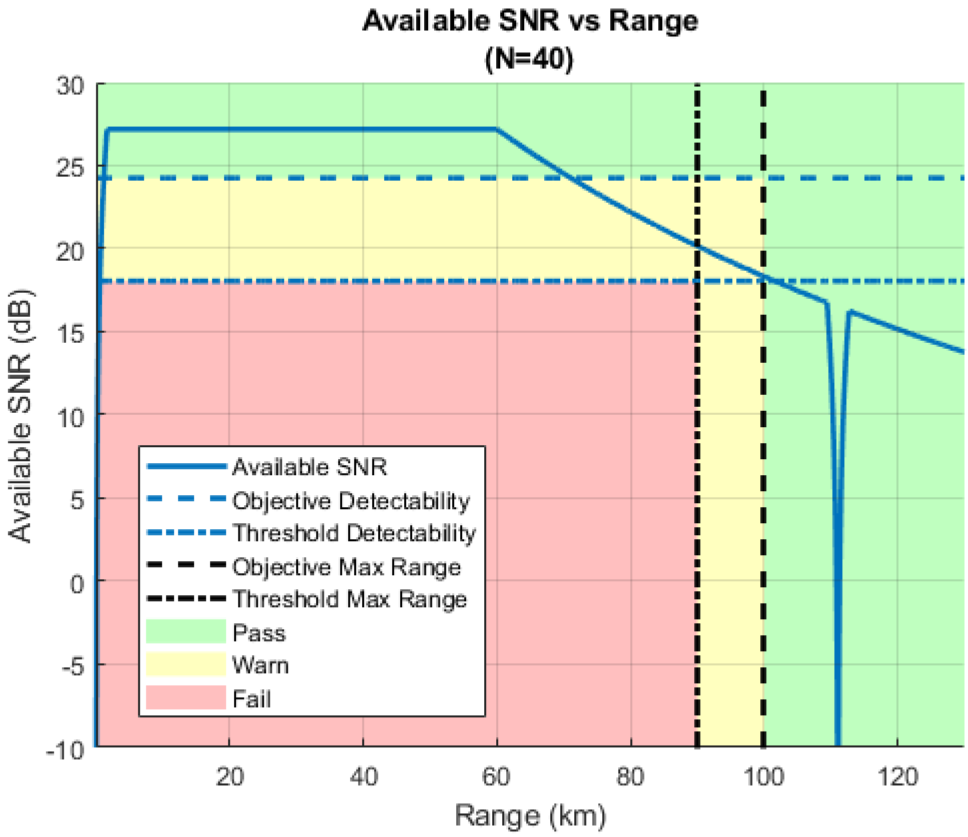
In addition, we can change the requirements on the maximum range and the detection probabilities. Instead of a single number specifying the desired probability of detection or the maximum range, a pair of `Objective` and `Threshold` values can be defined. The `Objective` requirement describes the desired 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);
title([{'Available SNR vs Range'}, {'(N=40)'}])
legend('Location','best')
ylim([-10 30]);
```

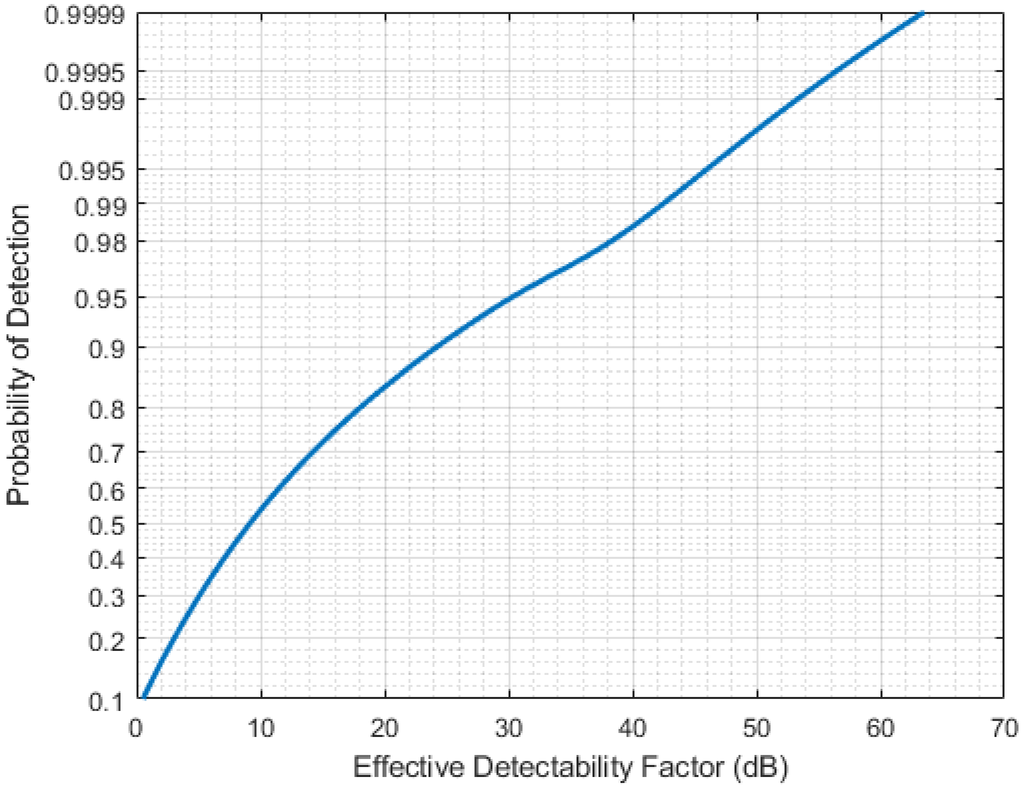


The SNR vs Range plot now has a yellow Warn zone indicating the SNR values and target ranges where the performance of the system is between the `Objective` and the `Threshold` requirements. We can see that up to approximately 70 km the system meets the `Objective` requirement for P_d. From 70 km to 100 km the `Objective` requirement for P_d is violated while the `Threshold` requirement is still satisfied.

Effective Probability of Detection

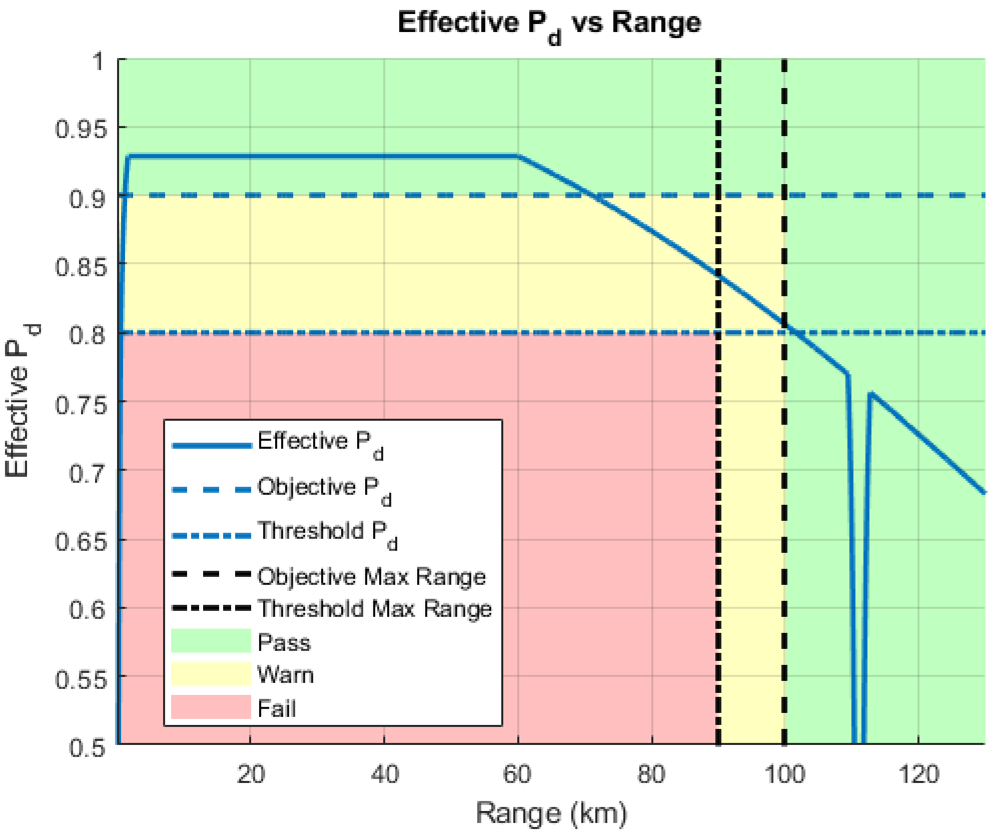
The SNR vs Range plot above shows that the detection performance of the radar system varies with range. A 1 m2 target at ranges below 70 km will be detected with the probability of detection greater or equal to 0.9, while between 70 km and 100 km it will be detected with P_d of at least 0.8. Since some 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);
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 m^2 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

$$I_m = \frac{C_i}{C_o},$$

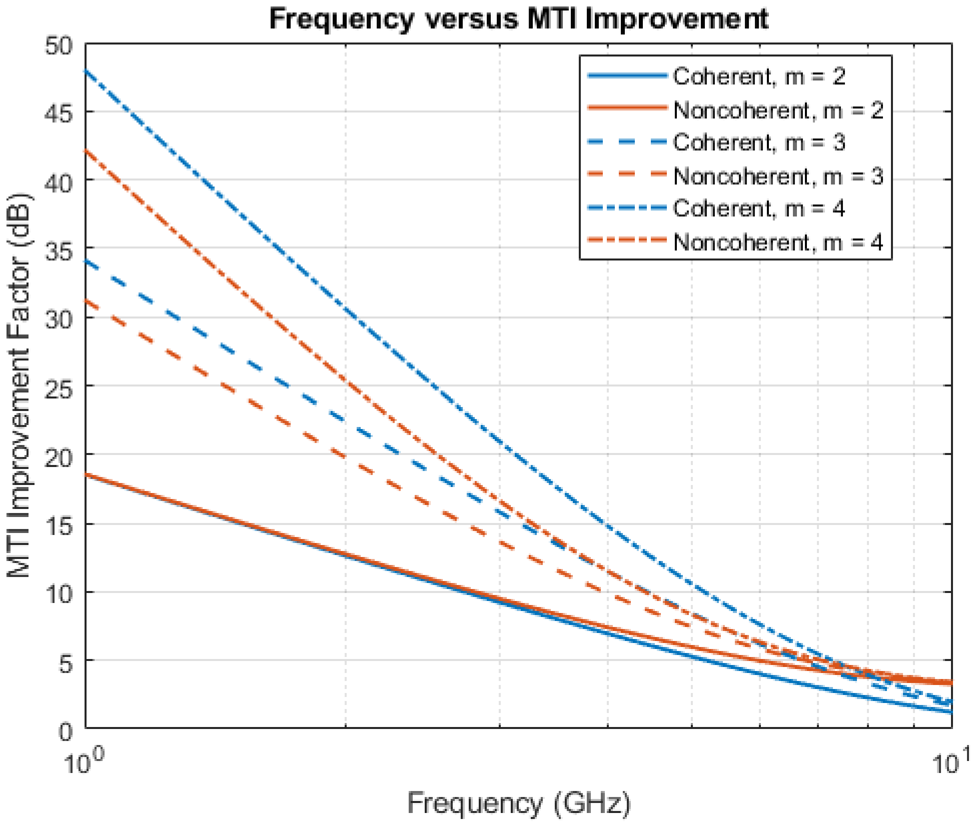
where $C_i$is the clutter power into the receiver and $C_o$is the clutter power after MTI processing.

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);
```



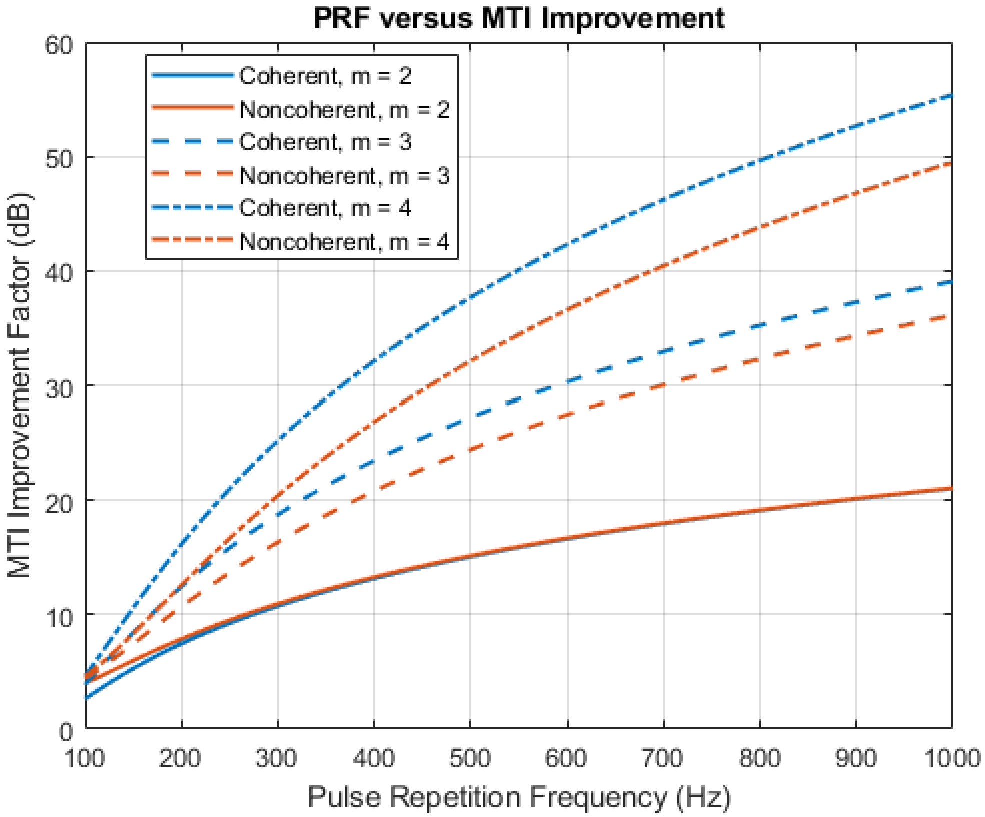
There are several takeaways from the results. First, the difference between the coherent and noncoherent results decreases with increasing frequency for the same m for the m = 3 and 4 cases. The results for the m = 2 case show that the improvement factor is very similar at lower frequencies but performance 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 In
```

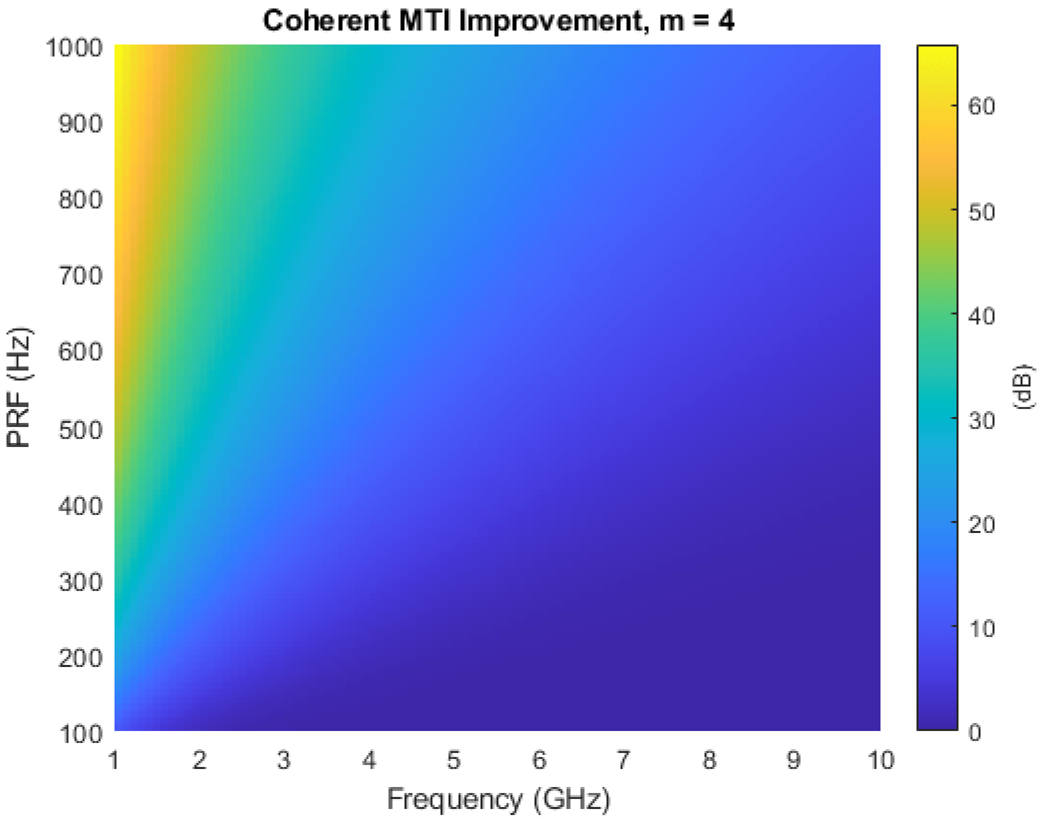


When frequency is held constant, there are several results to note. First, the difference between the coherent and noncoherent improvement factors increase for increasing frequency for the same m for the m = 3 and 4 cases. The results for the m = 2 case show that the improvement factor is very similar 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);
```



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.

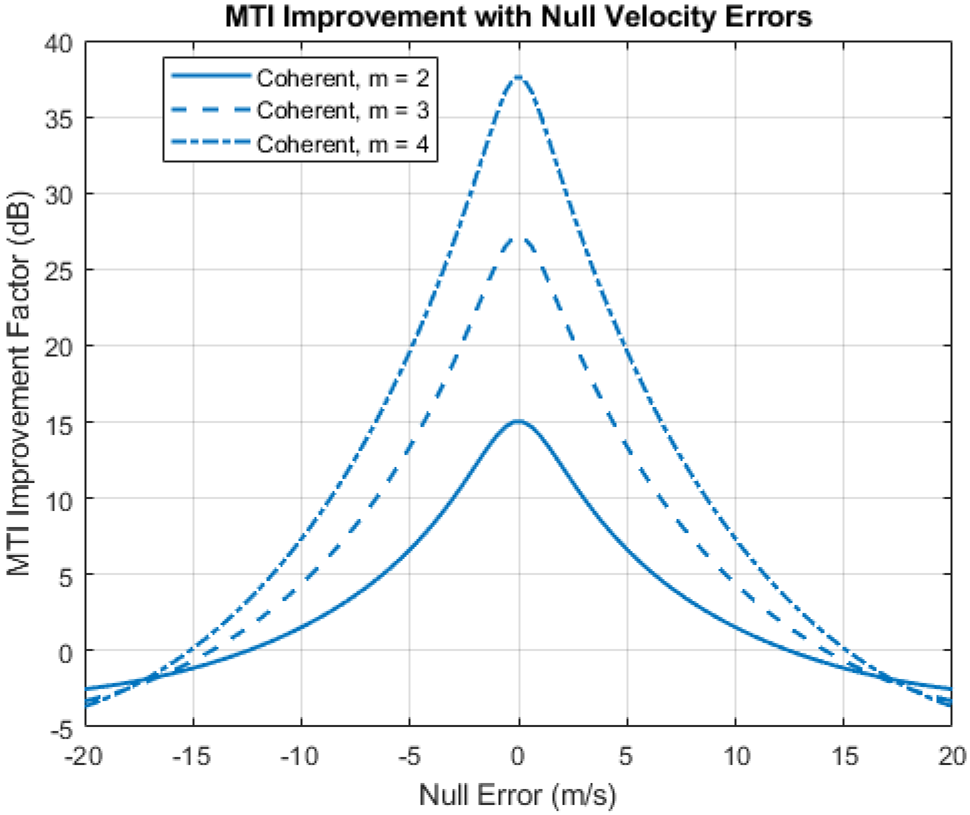
Null Velocity Errors

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.

```
% 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
```



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

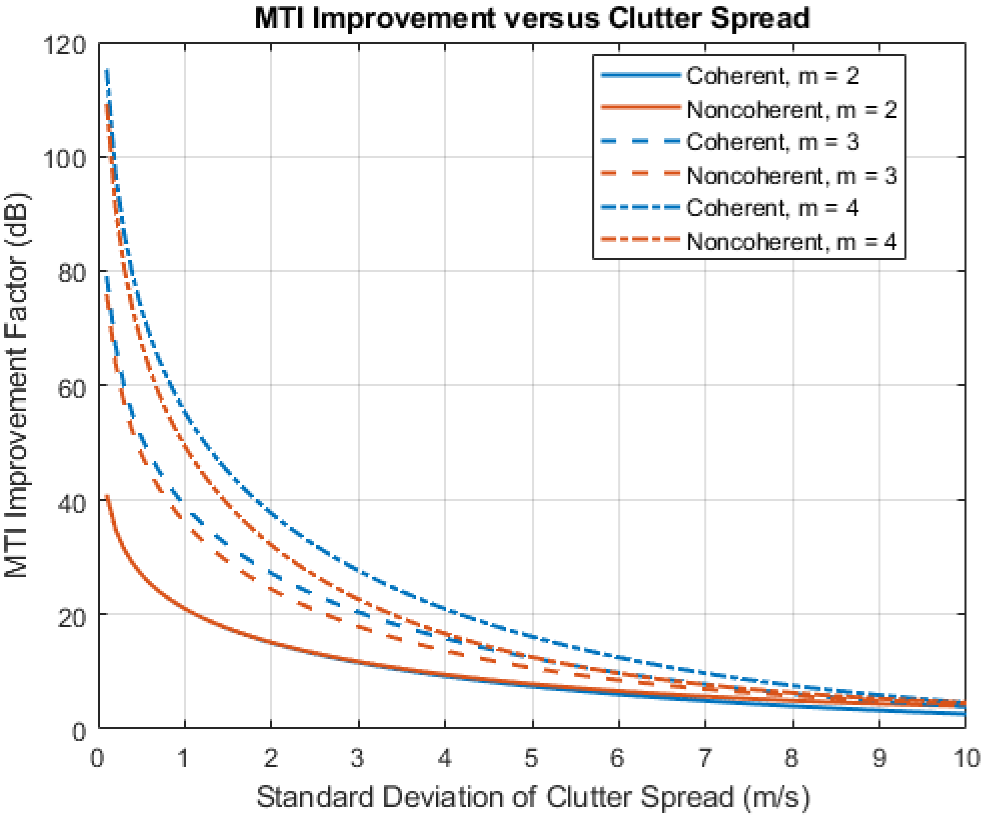
A wider clutter spread results in more clutter energy outside of an MTI filter null and thus results in less clutter cancellation. While clutter spread is due in part to the inherent motion of the clutter scatterers, other sources of clutter spread can be due to:

- 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,'ClutterStandardDevia
    end
end

% Plot results
helperPlotMTI(m,sigmav,ImCoherent,ImNoncoherent,'Standard Deviation of Clutter Spread (m/s)','MT
```



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.

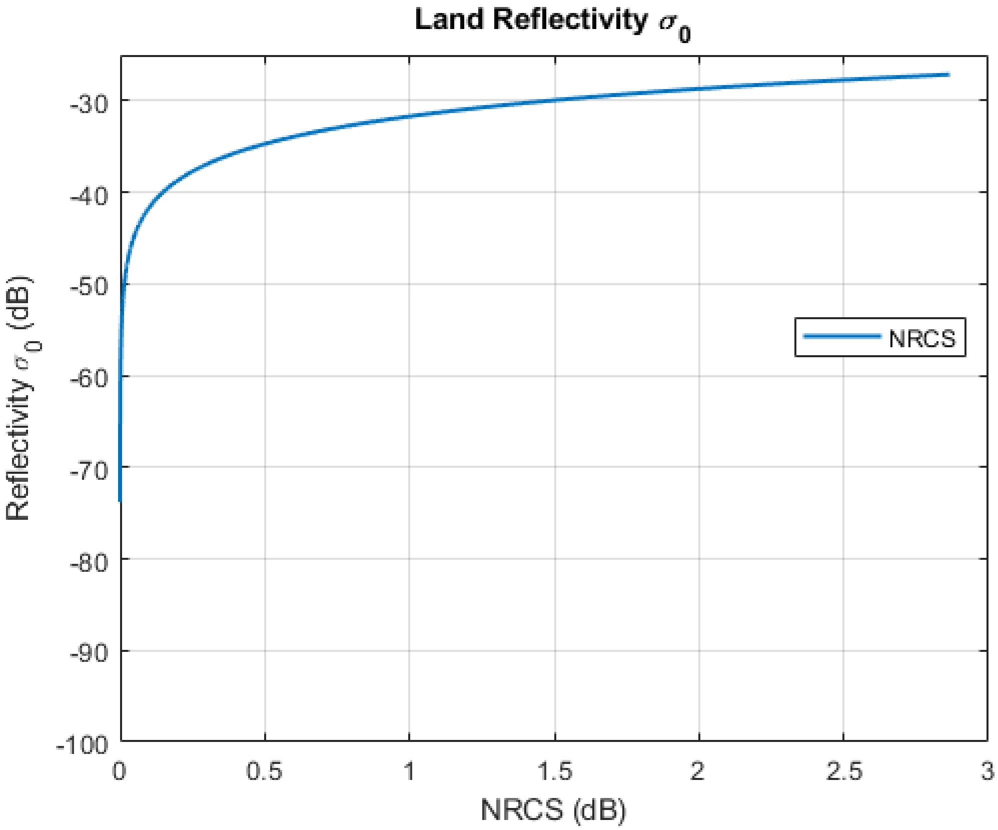
First, setup the radar and MTI processing parameters.

```
% 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
```

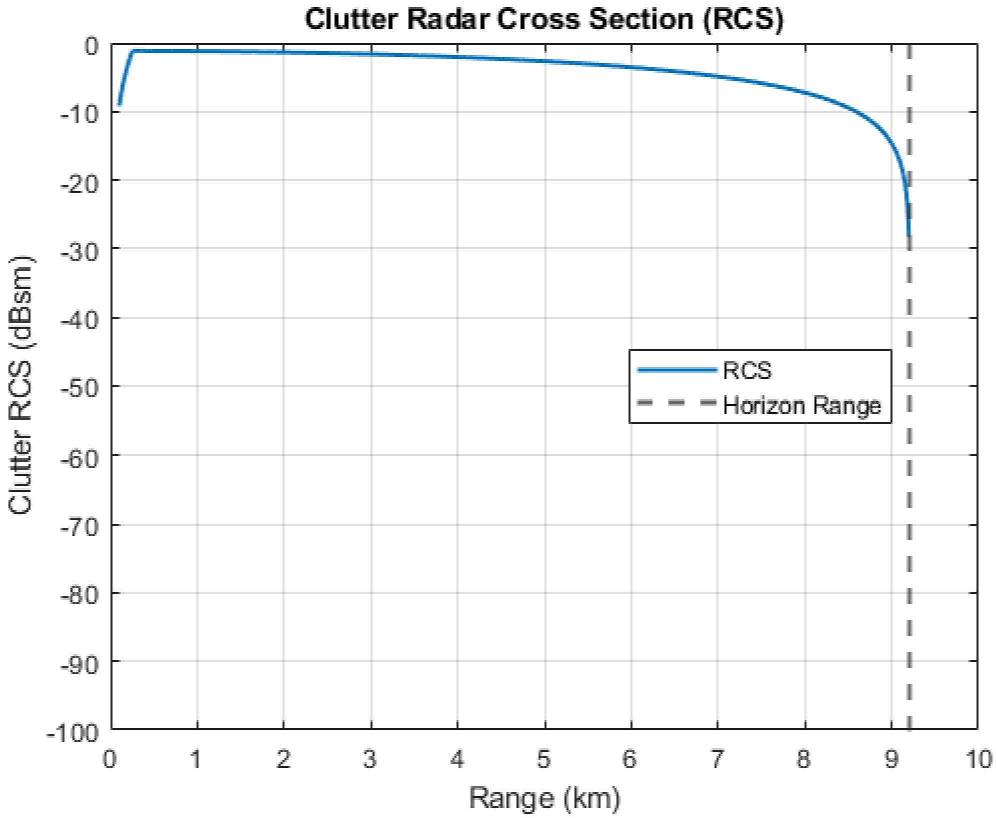
Consider a wooded hills operating environment with a clutter spread of 1 m/s and a mean clutter velocity of 0 m/s. Calculate and plot the land surface reflectivity for the grazing angles of the defined geometry. Use the `landroughness` and `landreflectivity` functions for the physical surface properties and reflectivity calculation, respectively.

```
% Clutter properties
sigmav     = 1;    % Standard deviation of clutter spread (m/s)
clutterVel = 0;    % Clutter velocity (m/s)
landType   = 'Wooded Hills';

% 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
grazAng = grazingang(anht,Rm,'TargetHeight',surfht);
nrcs = landreflectivity(landType,grazAng,freq);
nrcsdB = pow2db(nrcs);
helperPlot(grazAng,nrcsdB,'NRCS','NRCS (dB)','Reflectivity \sigma_0 (dB)','Land Reflectivity \sig
```



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 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 Sectio
helperAddHorizLine(hAxes,RhorizKm);
```



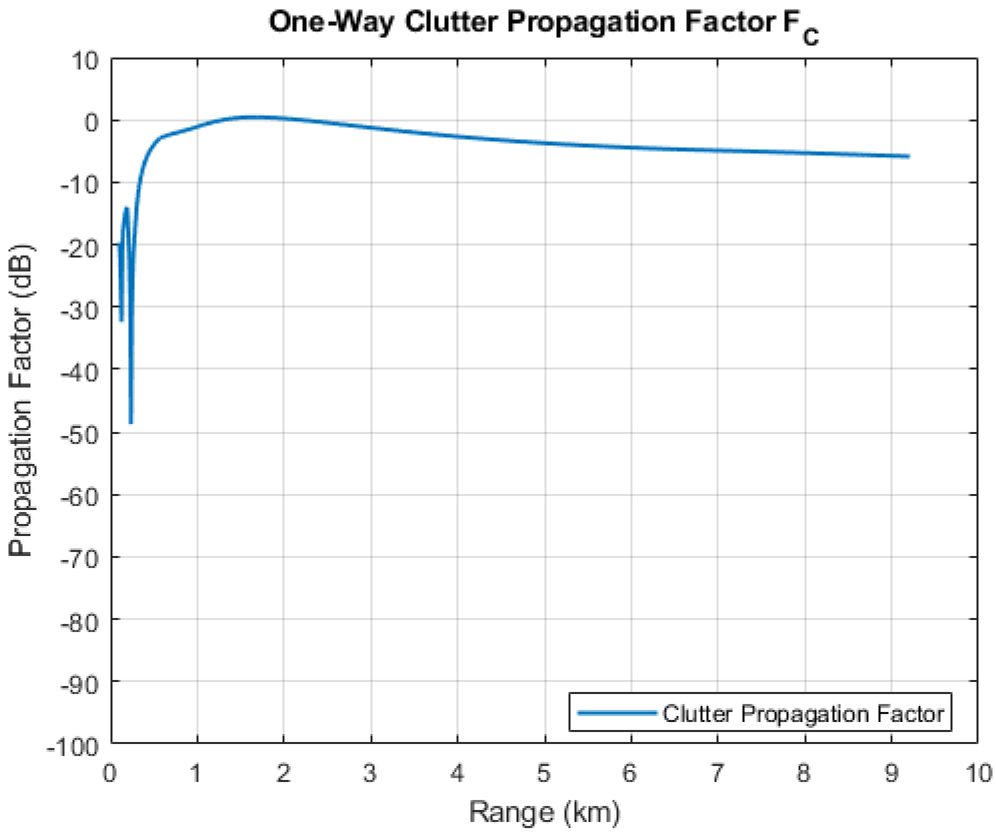
Since the propagation path deviates from free space, include the clutter propagation factor and atmospheric losses in the calculation.

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);
```

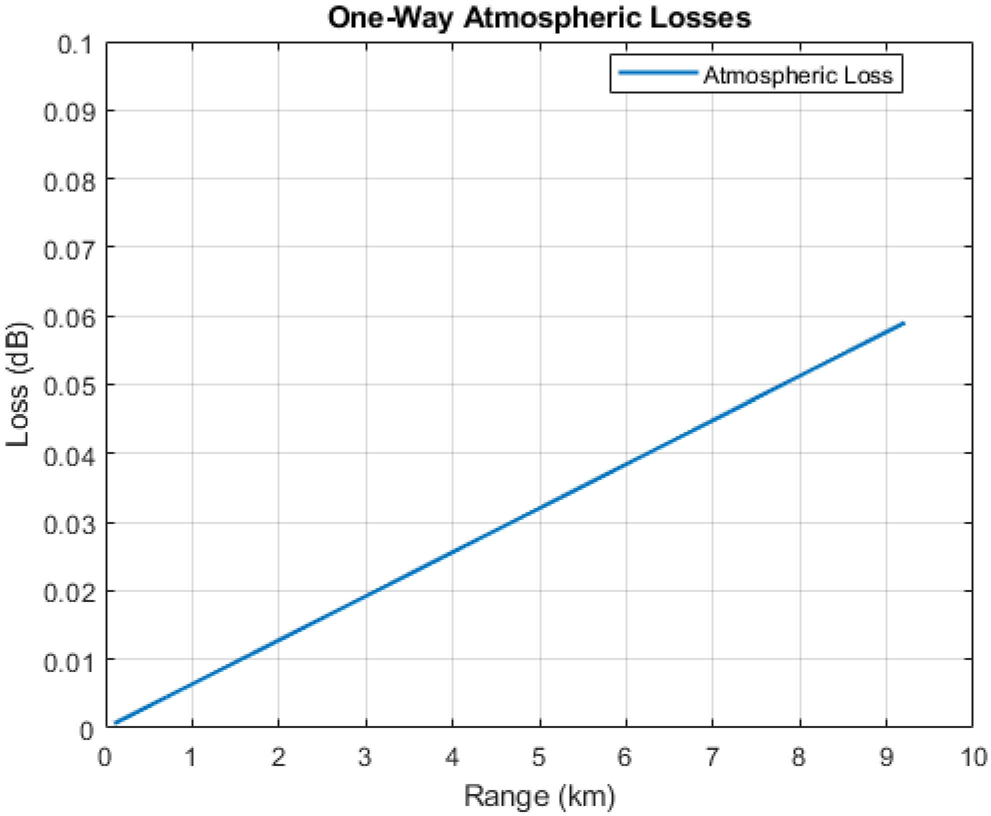
Calculate the clutter propagation factor using the `radarpropfactor` function. Include the vegetation type in the calculation. At higher frequencies, the presence of vegetation can cause additional losses.

```
% 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');
```



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 Lo
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.

```
% 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)
```

```
Im = 55.3986
```

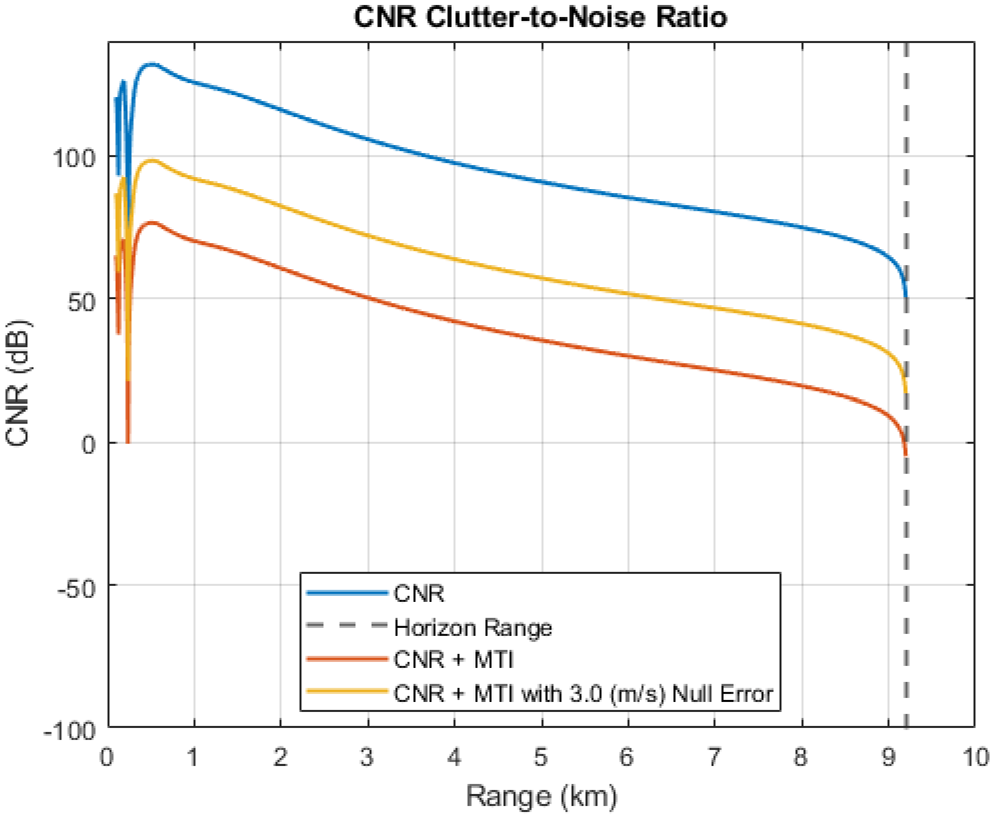
```
cnrMTI = cnr - Im;
helperAddPlot(Rkm,cnrMTI,'CNR + MTI',hAxes);
```

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 21 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.

```
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:

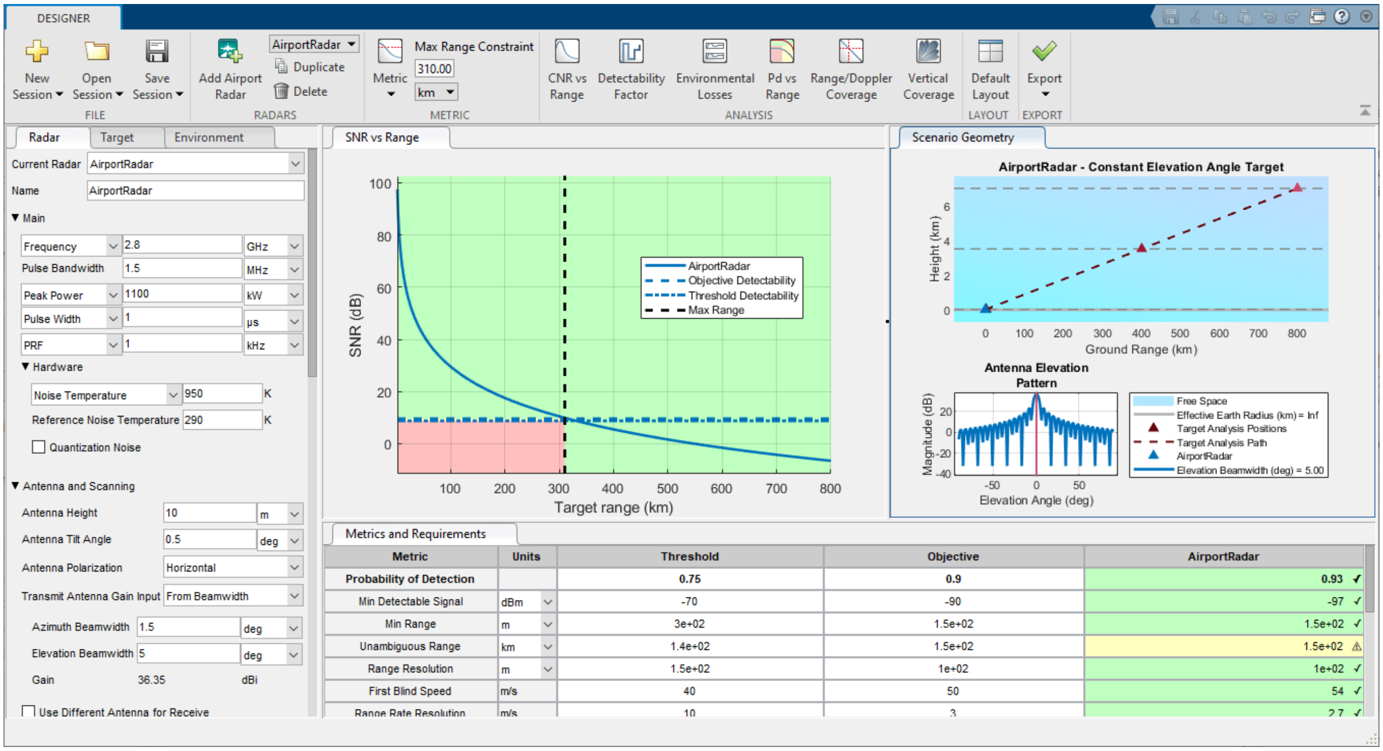
- The peak transmit power should not exceed 2000 W
- The radar should provide 360-degree coverage in azimuth and 60 degrees coverage in elevation
- The radar should detect small manned aircraft with the radar cross section of 1 m^2 at ranges from 300 m to 18 km
- The radar should detect small unmanned 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.



Radar | Target | Environment

Current Radar: AirportRadar

Name: AirportRadar

▼ Main

Frequency: 2.8 GHz

Pulse Bandwidth: 1.5 MHz

Peak Power: 1100 kW

Pulse Width: 1 μs

PRF: 1 kHz

▼ Hardware

Noise Temperature: 950 K

Reference Noise Temperature: 290 K

□ Quantization Noise

▼ Antenna and Scanning

Antenna Height: 10 m

Antenna Tilt Angle: 0.5 deg

Antenna Polarization: Horizontal

Transmit Antenna Gain Input: From Beamwidth

Azimuth Beamwidth: 1.5 deg

Elevation Beamwidth: 5 deg

Gain: 36.35 dBi

□ Use Different Antenna for Receive

Metrics and Requirements

Metric	Units	Threshold	Objective	AirportRadar
Probability of Detection		**0.75**	**0.9**	**0.93 ✓**
Min Detectable Signal	dBm	-70	-90	-97 ✓
Min Range	m	3e+02	1.5e+02	1.5e+02 ✓
Unambiguous Range	km	1.4e+02	1.5e+02	1.5e+02 ⚠
Range Resolution	m	1.5e+02	1e+02	1e+02 ✓
First Blind Speed	m/s	40	50	54 ✓
Range Rate Resolution	m/s	10	3	2.7 ✓

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

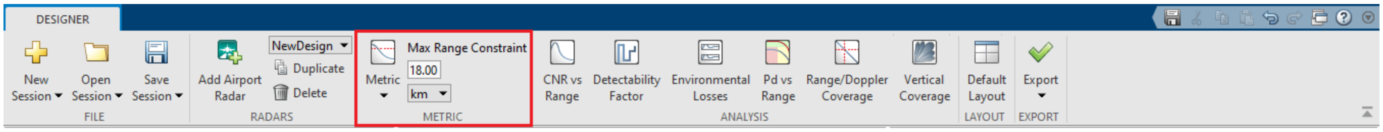
In this example we have a specification detailing the performance of the final system. However, it does not specify all the design parameters needed to achieve the required levels of performance. Most of the design parameters must be derived from the requirements given in the specification. The **Radar 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

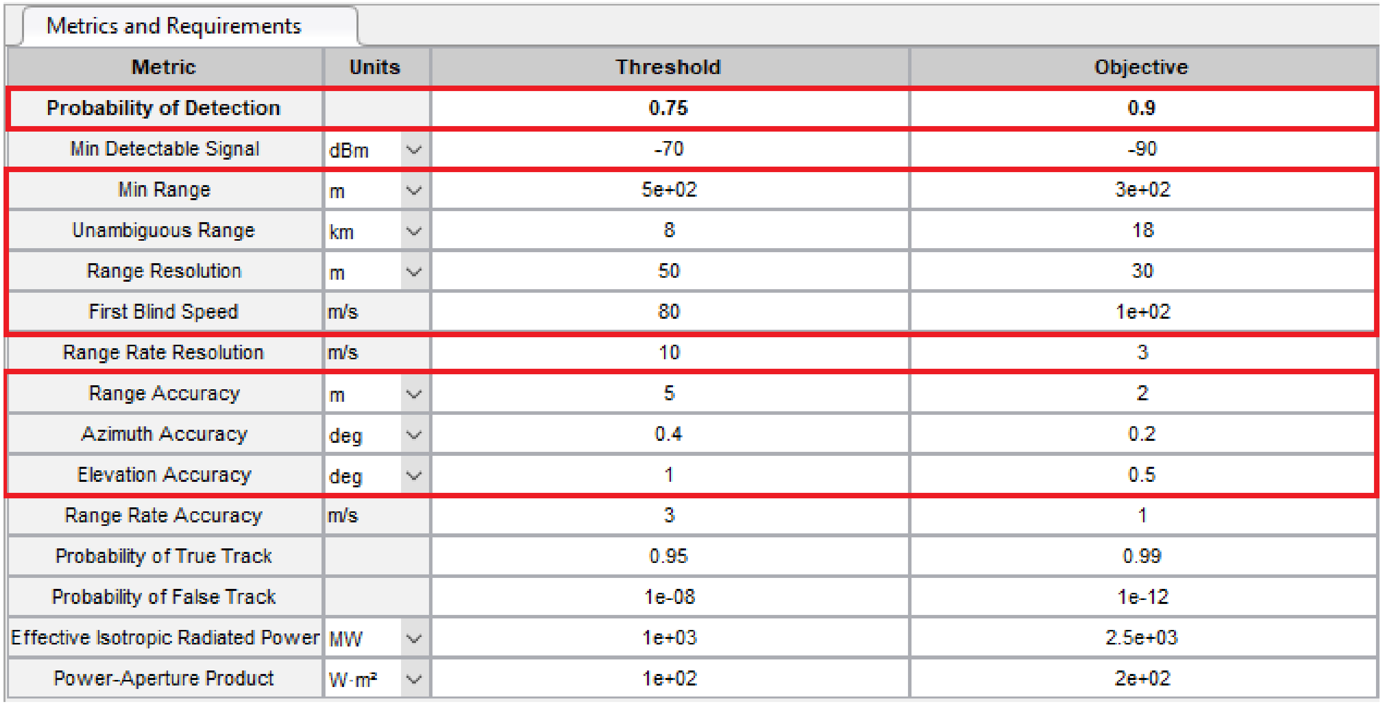
The values between `Threshold` and `Objective` constitute the trade-space that can be used by a radar engineer to balance multiple, sometimes conflicting, performance requirements.

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.



Max Range Constraint 18.00 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.

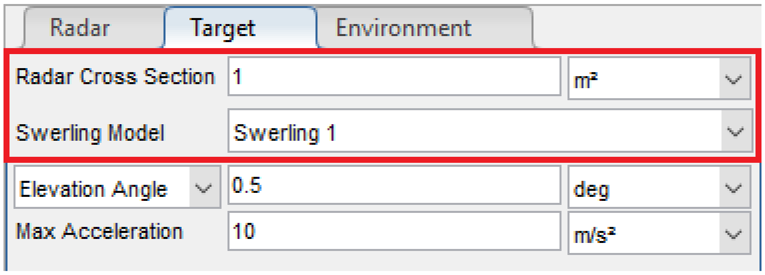


Metrics and Requirements

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 Cross Section 1 m²

Swerling Model Swerling 1

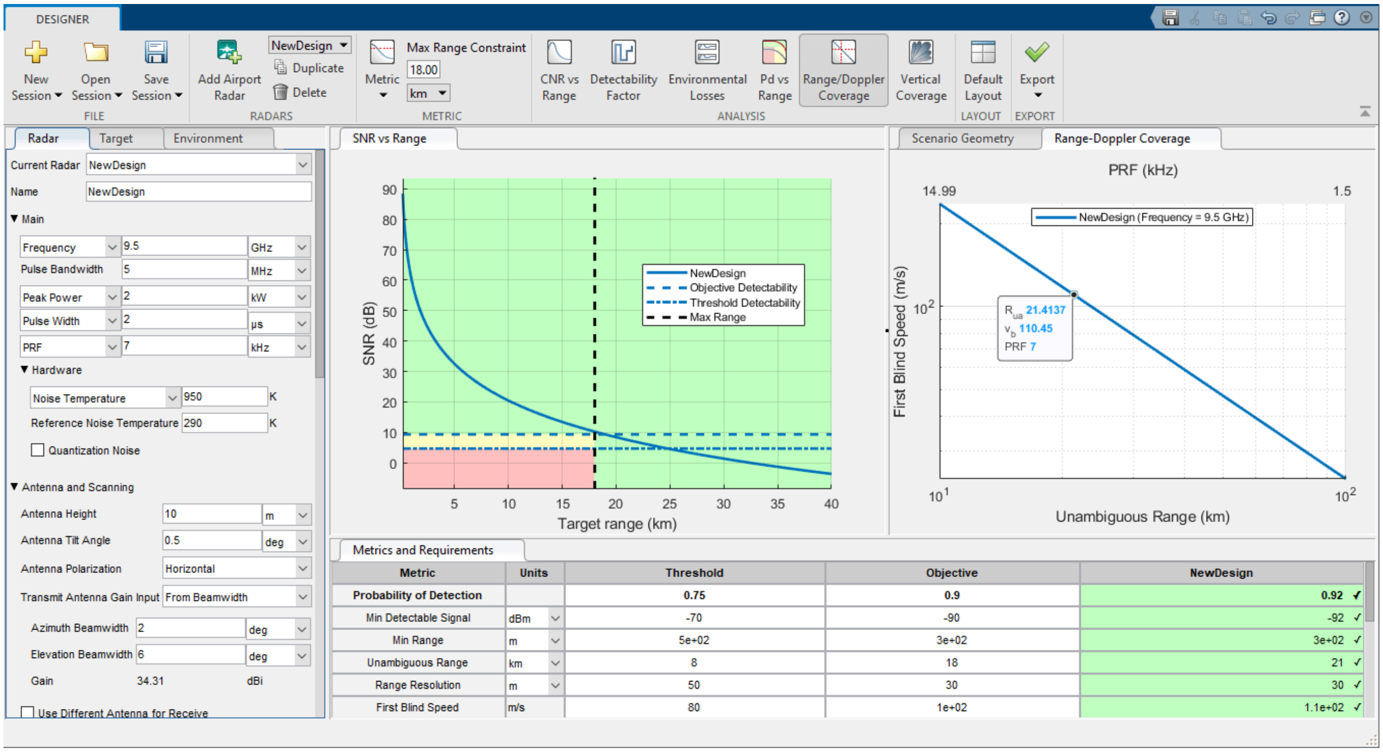
Elevation Angle 0.5 deg

Max Acceleration 10 m/s²

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`.



Current Radar: NewDesign

Name: NewDesign

▼ Main

Frequency 9.5 GHz

Pulse Bandwidth 5 MHz

Peak Power 2 kW

Pulse Width 2 μs

PRF 7 kHz

▼ Hardware

Noise Temperature 950 K

Reference Noise Temperature 290 K

□ Quantization Noise

▼ 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

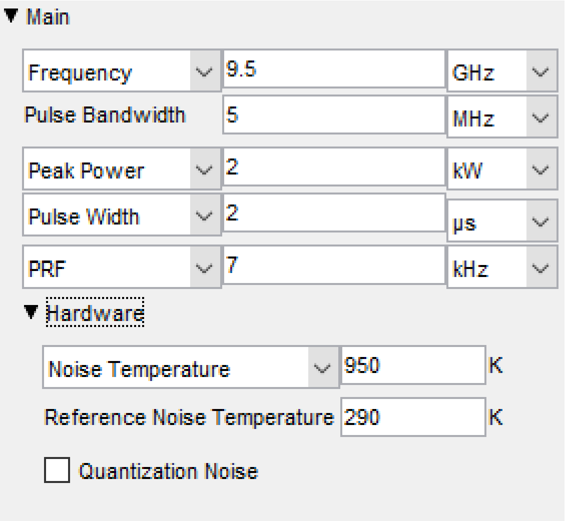
Gain 34.31 dBi

□ Use Different Antenna for Receive

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 ✓

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.



▼ Main

Frequency 9.5 GHz

Pulse Bandwidth 5 MHz

Peak Power 2 kW

Pulse Width 2 μs

PRF 7 kHz

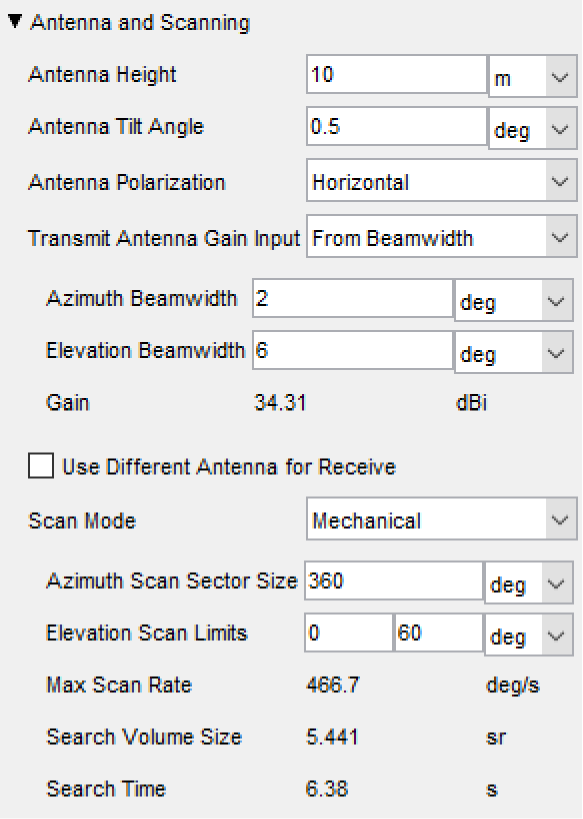
▼ Hardware

Noise Temperature 950 K

Reference Noise Temperature 290 K

□ Quantization Noise

- `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.



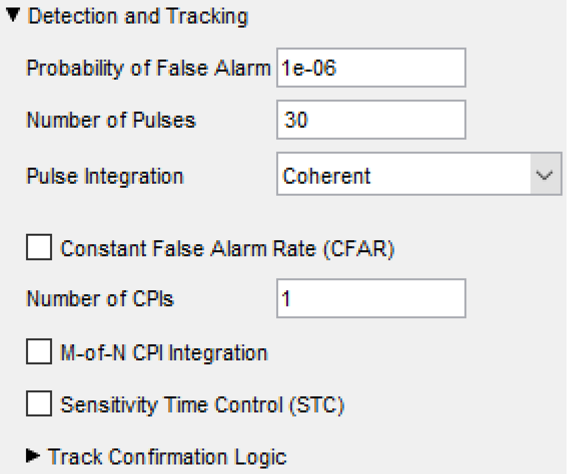
▼ Antenna and Scanning

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.



▼ Detection and Tracking

Probability of False Alarm: 1e-06

Number of Pulses: 30

Pulse Integration: Coherent

□ Constant False Alarm Rate (CFAR)

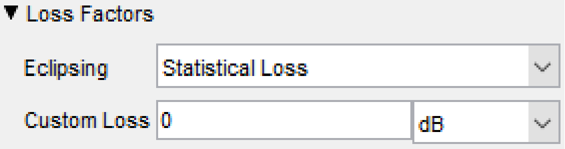
Number of CPIs: 1

□ M-of-N CPI Integration

□ Sensitivity Time Control (STC)

▶ Track Confirmation Logic

- `Loss Factors`. To account for losses due to pulse eclipsing we add the statistical eclipsing loss to the link budget analysis.

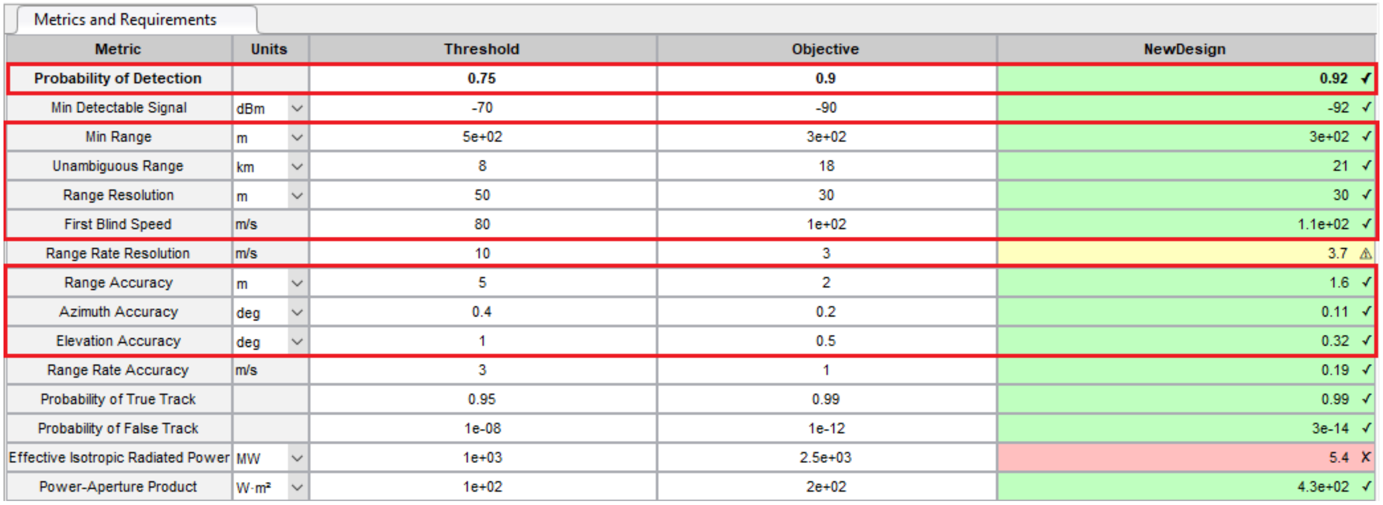


▼ Loss Factors

Eclipsing: Statistical Loss

Custom Loss: 0 dB

After these adjustments, the **Metric and Requirements** table shows that this design satisfies the specification for small manned aircraft with RCS of 1 m^2 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 m^2 target with this probability of detection is –92 dBm.



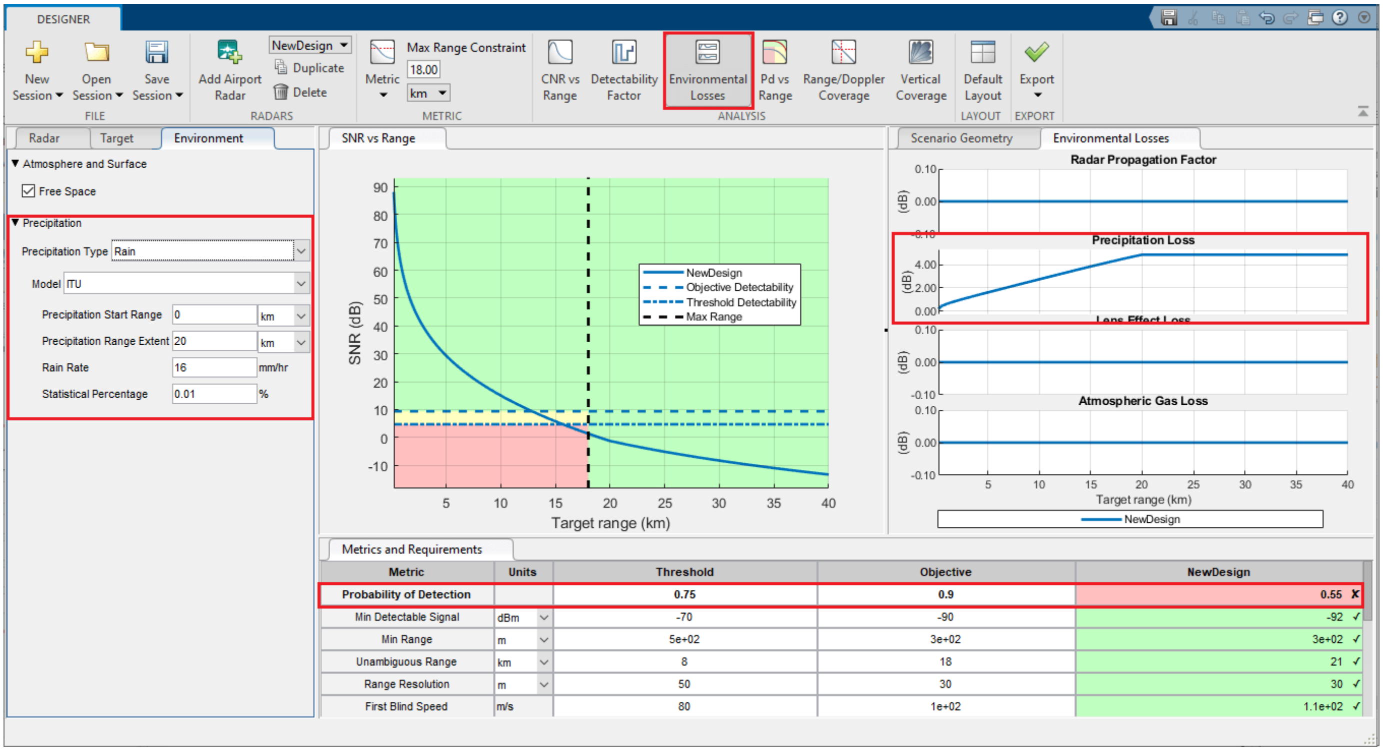
Metrics and Requirements

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.



▼ Atmosphere and Surface

☑ Free Space

▼ Precipitation

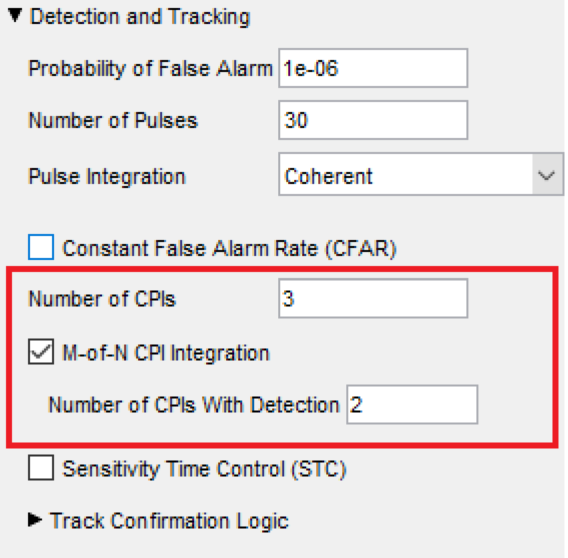
Precipitation Type	Rain	
Model	ITU	
Precipitation Start Range	0	km
Precipitation Range Extent	20	km
Rain Rate	16	mm/hr
Statistical Percentage	0.01	%

Metric	Units	Threshold	Objective	NewDesign
Probability of Detection		0.75	0.9	0.55 ✗
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 ✓

he `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.



▼ Detection and Tracking

Probability of False Alarm: 1e-06

Number of Pulses: 30

Pulse Integration: Coherent

□ Constant False Alarm Rate (CFAR)

Number of CPIs: 3

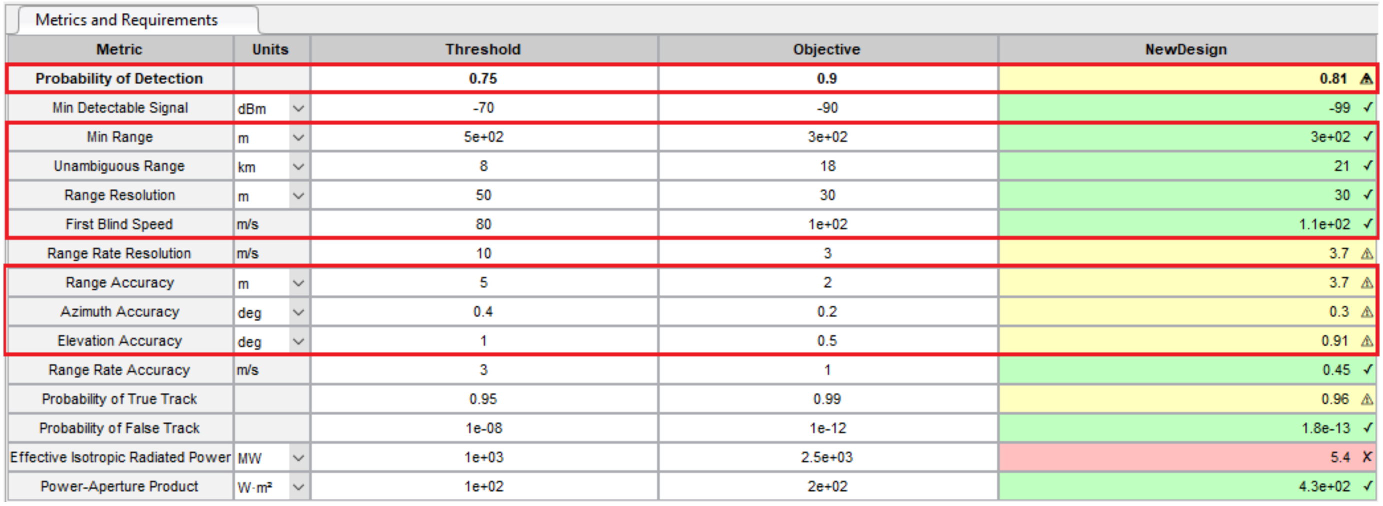
☑ M-of-N CPI Integration

Number of CPIs With Detection: 2

□ Sensitivity Time Control (STC)

▶ Track Confirmation Logic

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.



Metrics and Requirements

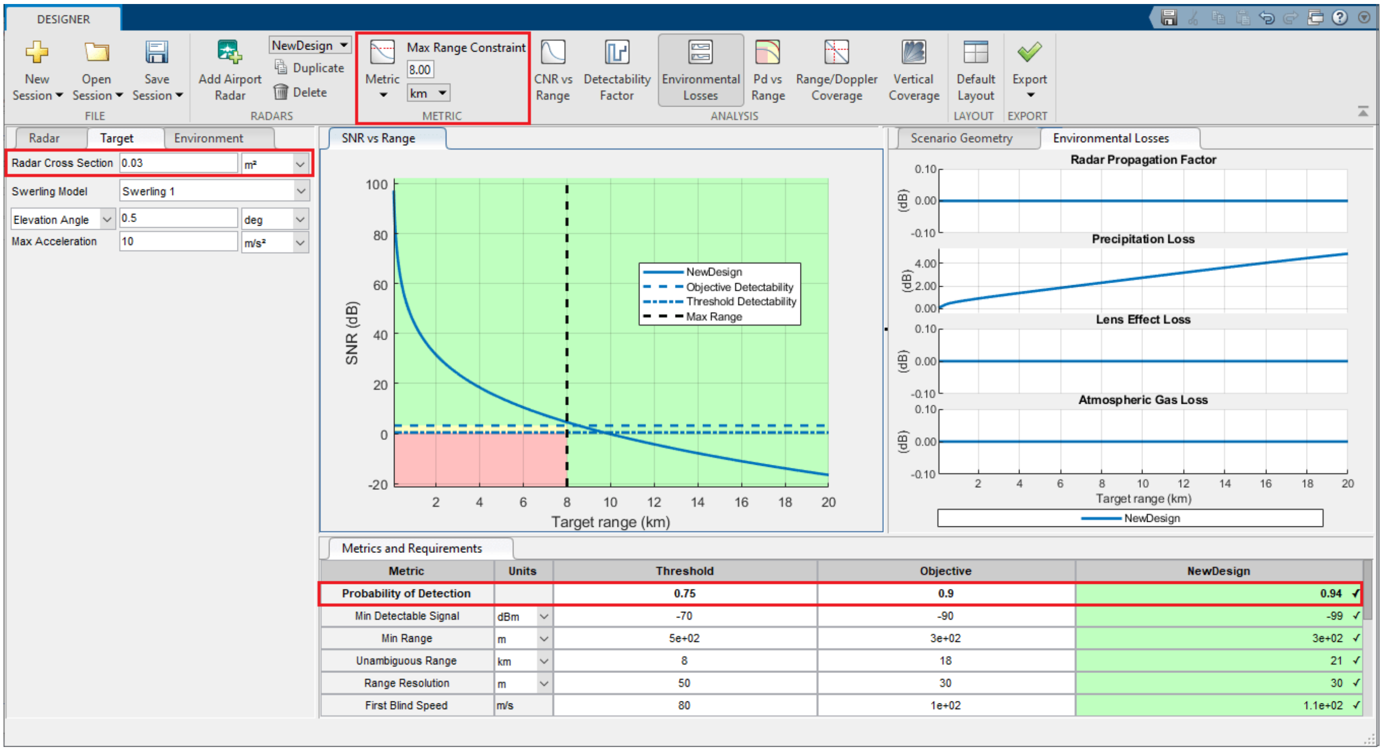
Metric	Units	Threshold	Objective	NewDesign
Probability of Detection		**0.75**	**0.9**	**0.81** ⚠
Min Detectable Signal	dBm	-70	-90	-99 ✓
Min Range	m	5e+02	3e+02	3e+02 ✓
Unambiguous Range	km	8	18	21 ✓
Range Resolution	m	50	30	30 ✓
First Blind Speed	m/s	80	1e+02	1.1e+02 ✓
Range Rate Resolution	m/s	10	3	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.



Max Range Constraint 8.00 km

Radar Cross Section 0.03 m²

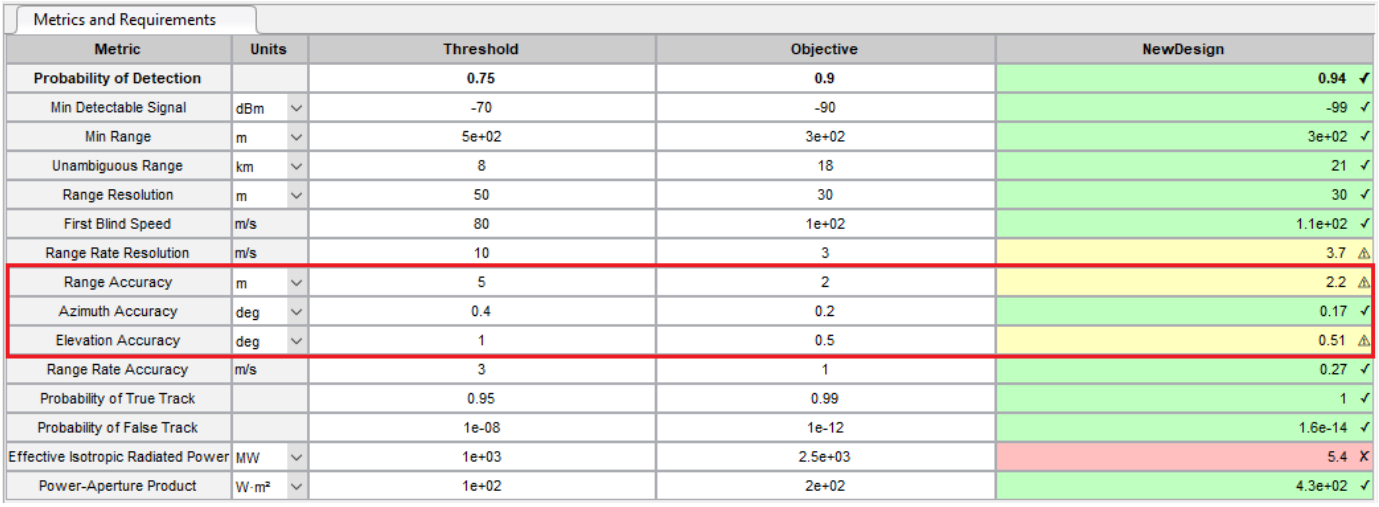
Swerling Model Swerling 1

Elevation Angle 0.5 deg

Max Acceleration 10 m/s²

Metric	Units	Threshold	Objective	NewDesign
Probability of Detection		**0.75**	**0.9**	**0.94** ✓
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 ✓

The resulting range and the elevation accuracies, however, are still below the `Objective` and above the `Threshold` requirements.



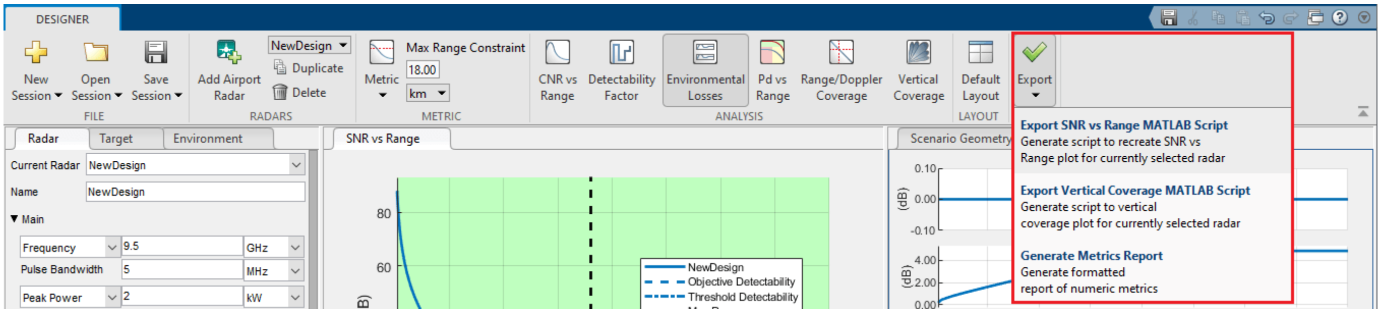
Metrics and Requirements

Metric	Units	Threshold	Objective	NewDesign
Probability of Detection		**0.75**	**0.9**	**0.94** ✓
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	2.2 ⚠
Azimuth Accuracy	deg	0.4	0.2	0.17 ✓
Elevation Accuracy	deg	1	0.5	0.51 ⚠
Range Rate Accuracy	m/s	3	1	0.27 ✓
Probability of True Track		0.95	0.99	1 ✓
Probability of False Track		1e-08	1e-12	1.6e-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 ✓

.

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.



Export SNR vs Range MATLAB Script
Generate script to recreate SNR vs Range plot for currently selected radar

Export Vertical Coverage MATLAB Script
Generate script to vertical coverage plot for currently selected radar

Generate Metrics Report
Generate formatted report of numeric 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.

This example requires Antenna Toolbox™ and Radar Toolbox.

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.

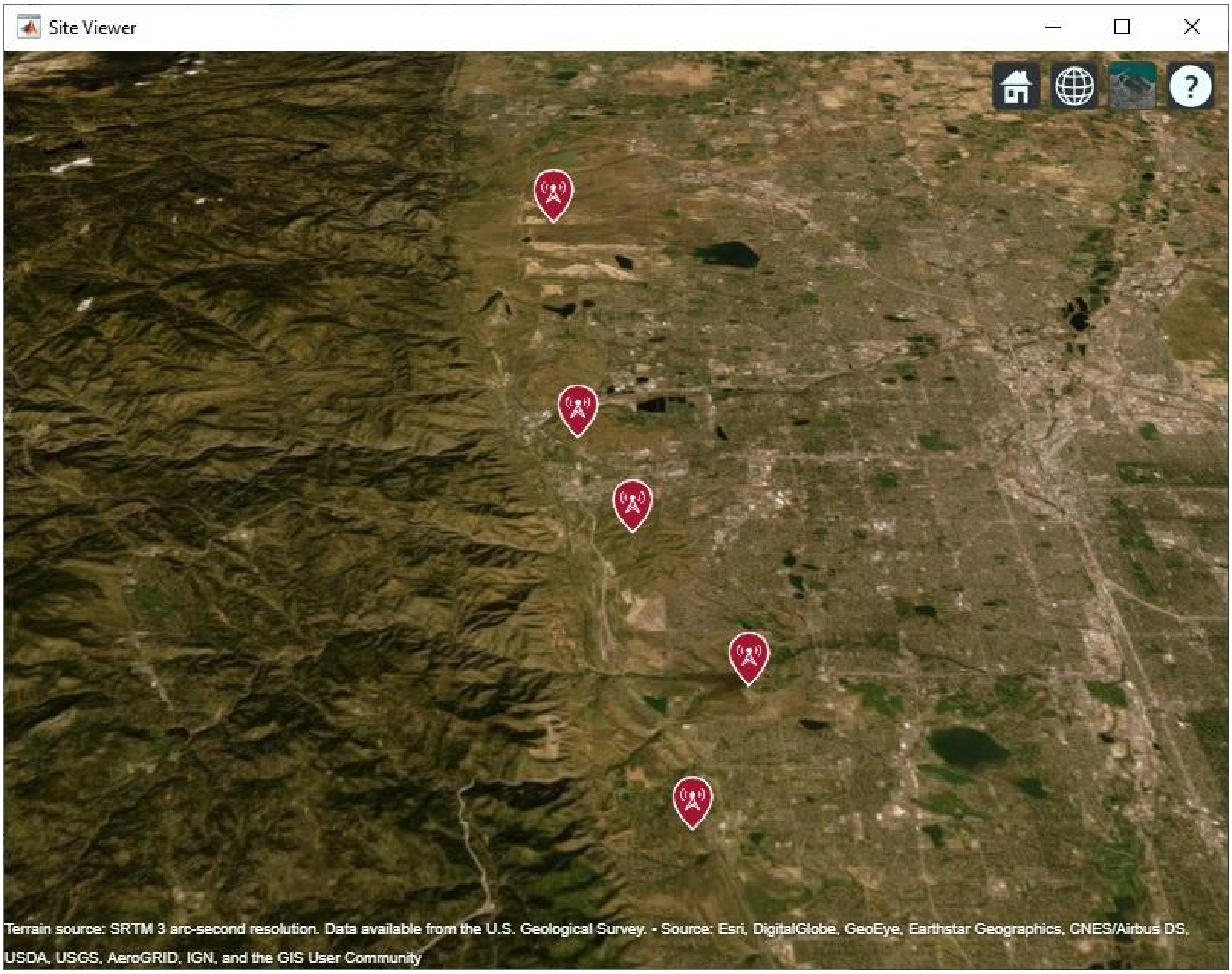
```
viewer = siteviewer("Terrain","southboulder");
```

Show Candidate Radar Sites

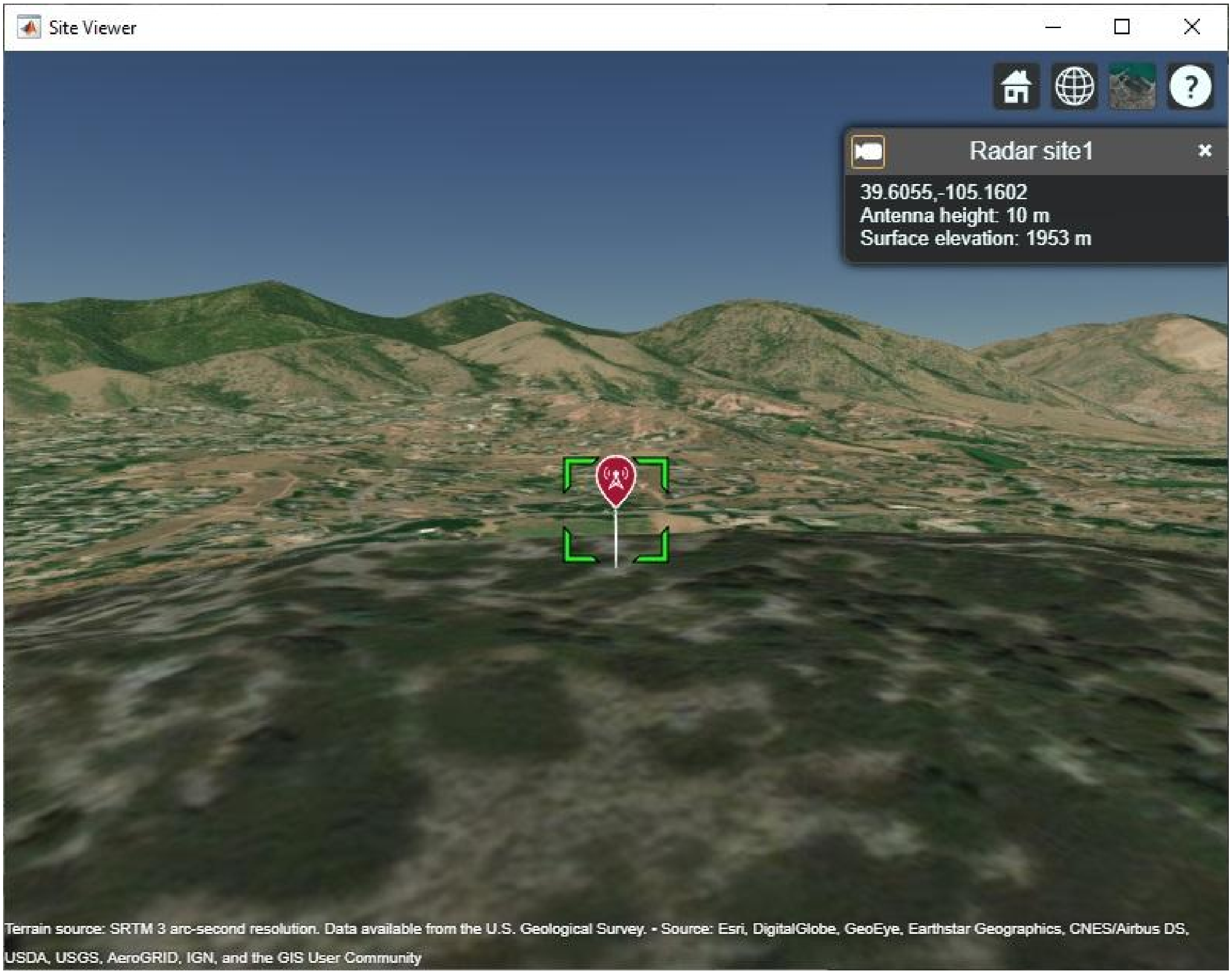
The region contains mountains to the west and flatter areas to the east. Radars will be placed in the flat area to detect targets over the mountainous region. Define five candidate locations for placing the radars and show them on the map. The candidate locations are chosen to correspond to local high 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);
```


Site Viewer
Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

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.


Site Viewer
Radar site1
39.6055,-105.1602
Antenna height: 10 m
Surface elevation: 1953 m
Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Design Monostatic Radar System

Design a basic monostatic pulse radar system to detect non-fluctuating targets with 0.1 square meter radar cross section (RCS) at a distance up to 35000 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))
```

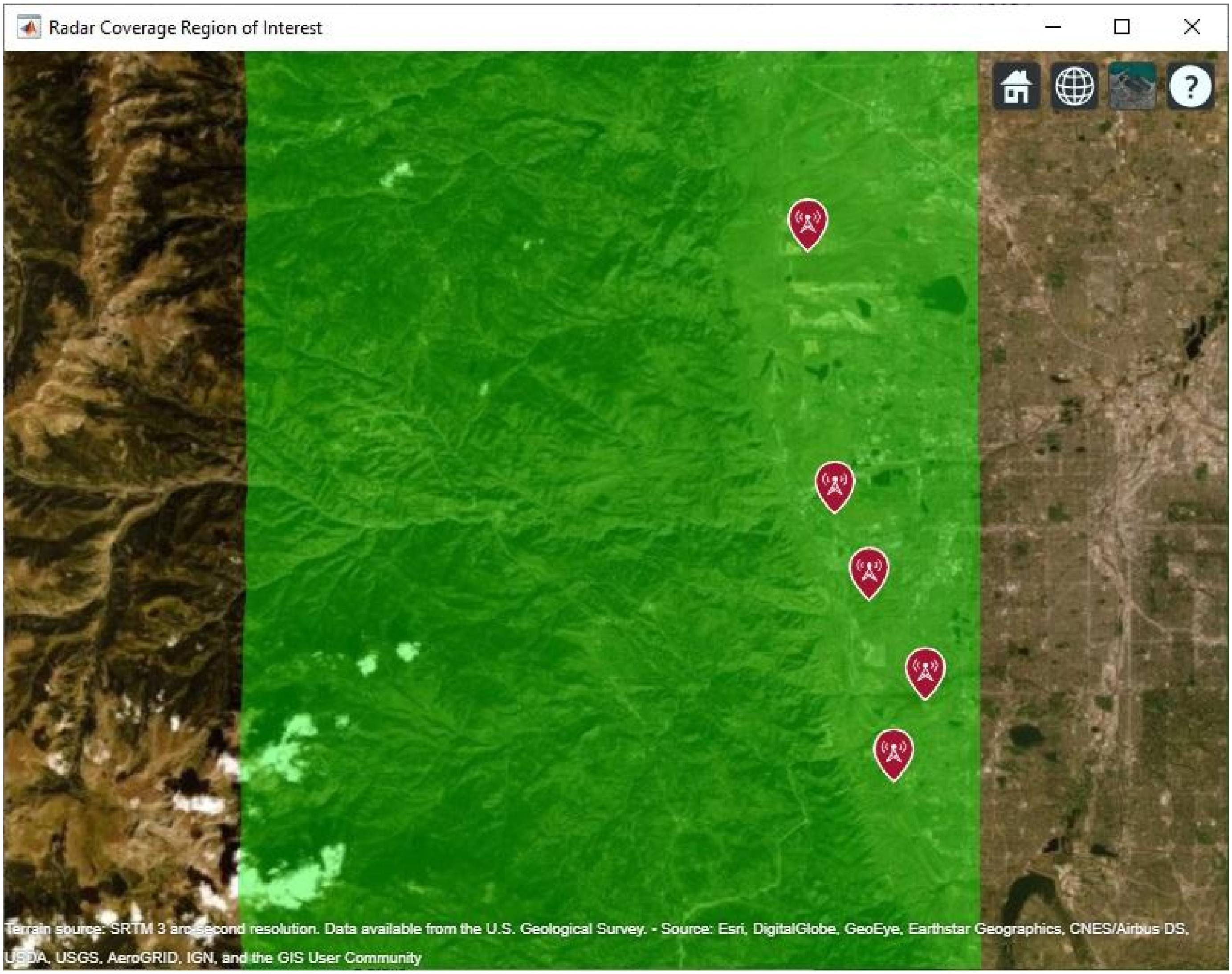
```
Ground elevation (meters): Min    Max    Mean
                           1257   3953   2373
```

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)
```



Radar Coverage Region of Interest
Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

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 Terrain Integrated Rough Earth Model™ (TIREM™) from Alion Science if it is available, or else use the Longley-Rice (aka 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.

```
% 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.

```
% 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.

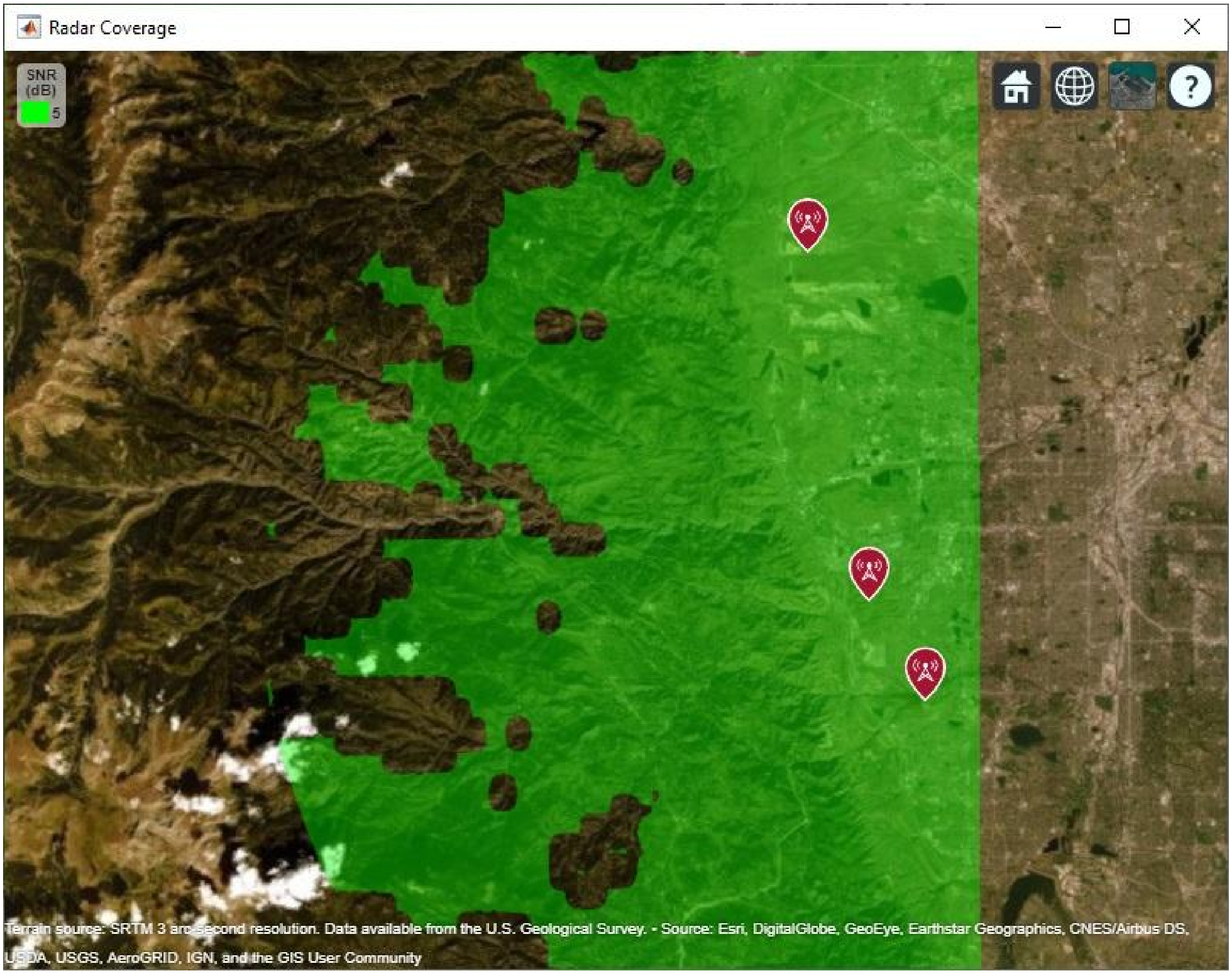
```
bestsitenums = helperOptimizeRadarSites(rawsnr, snrthreshold);
snr = max(rawsnr(:,bestsitenums),[],2);
```

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.

```
% 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)
```

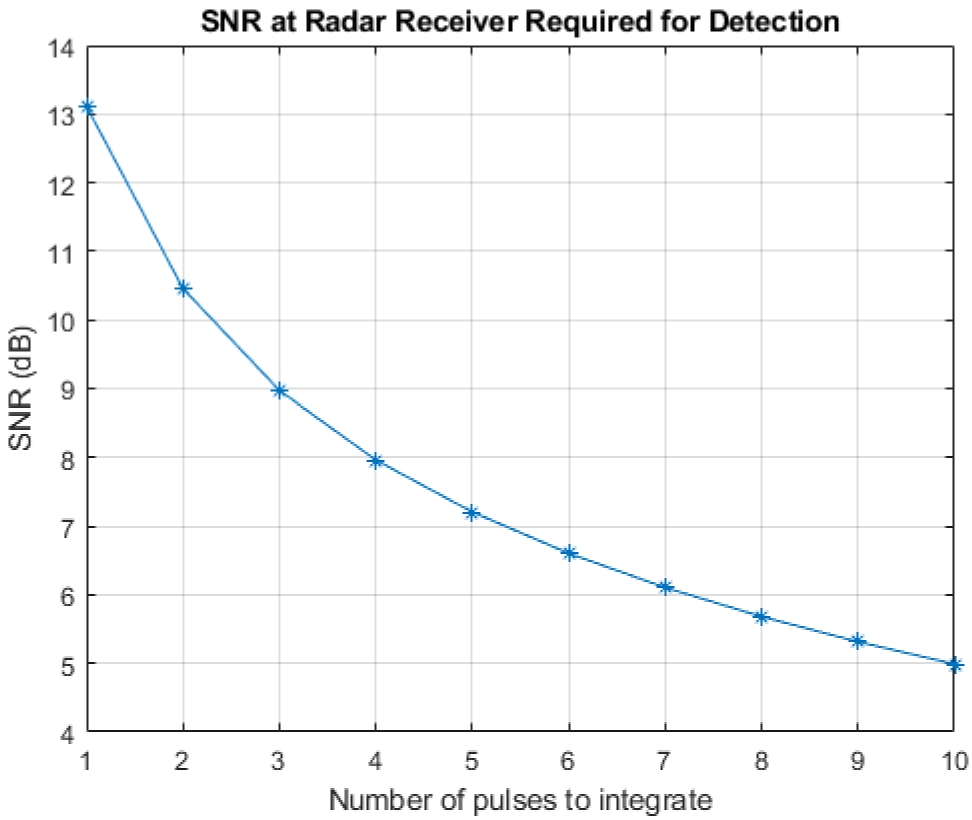

Radar Coverage
SNR (dB)
🟩 5
Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Vary the Number of Pulses to Integrate

The analysis above 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.

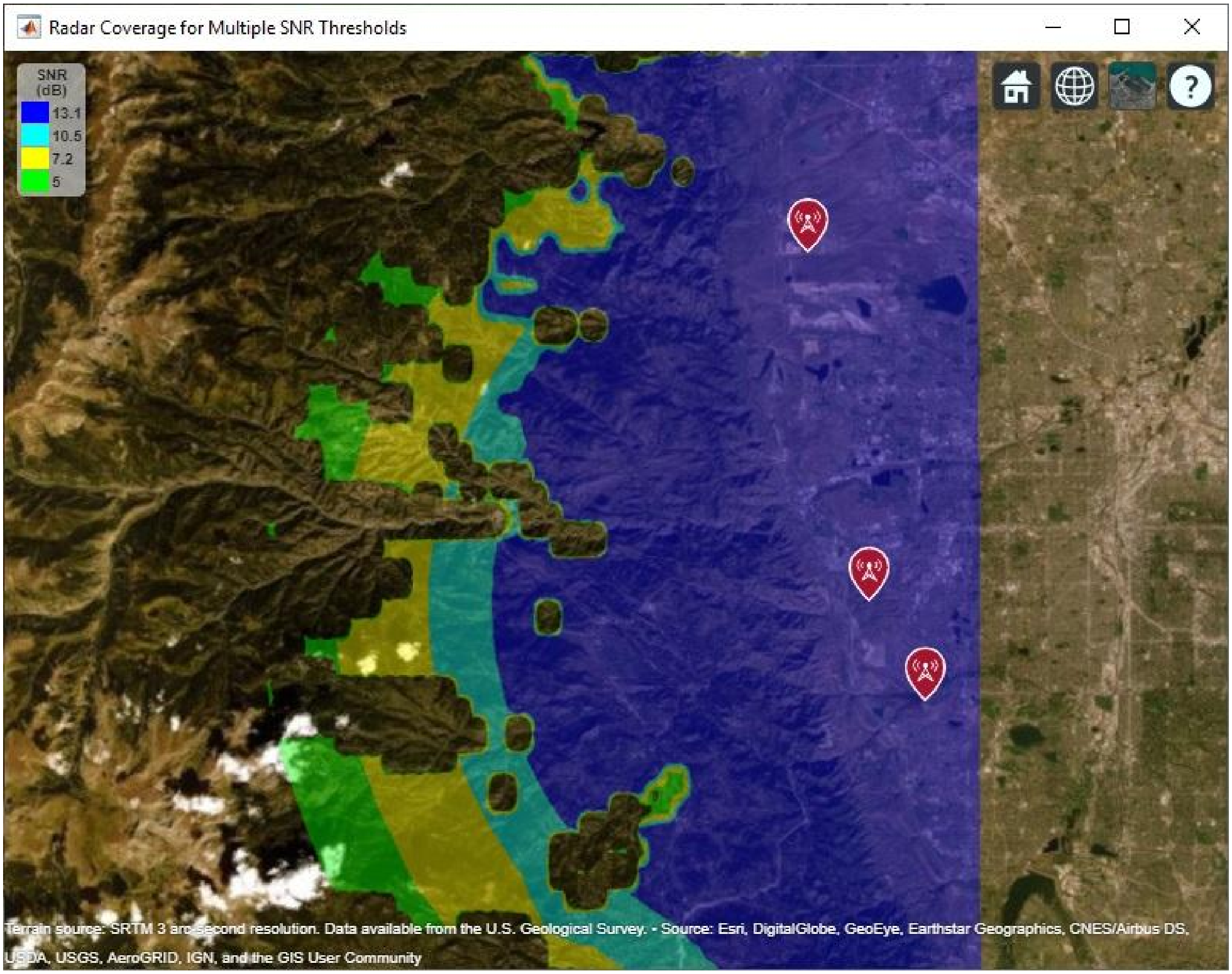
```
% Calculate SNR thresholds corresponding to different number of pulses
numpulses = 1:10;
snrthresholds = zeros(1,numel(numpulses));
for k = 1:numel(numpulses)
    snrthresholds(k) = albersheim(pd, pfa, numpulses(k));
end

% 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;
```



Show 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.

```
% 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)
```



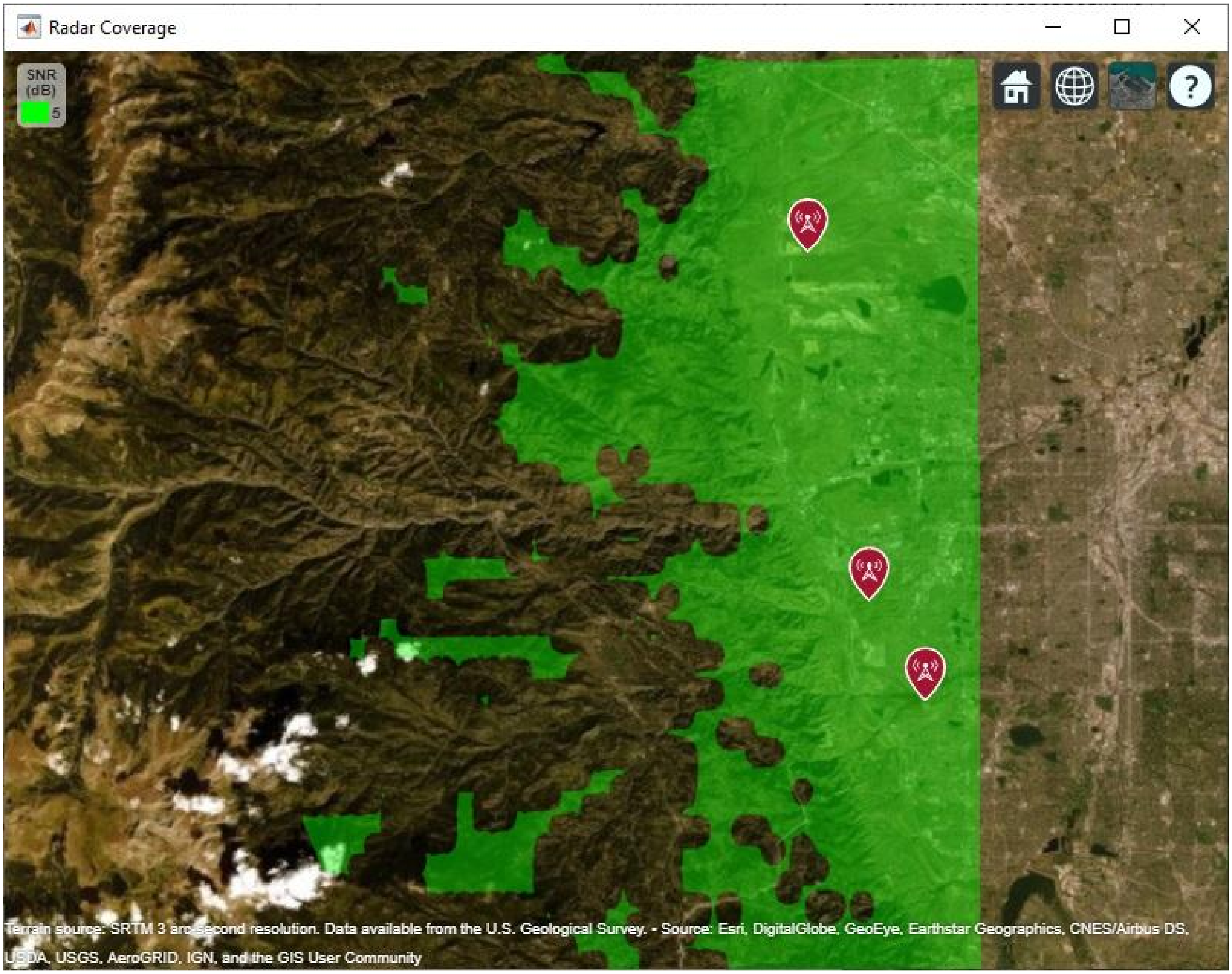
Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

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
```

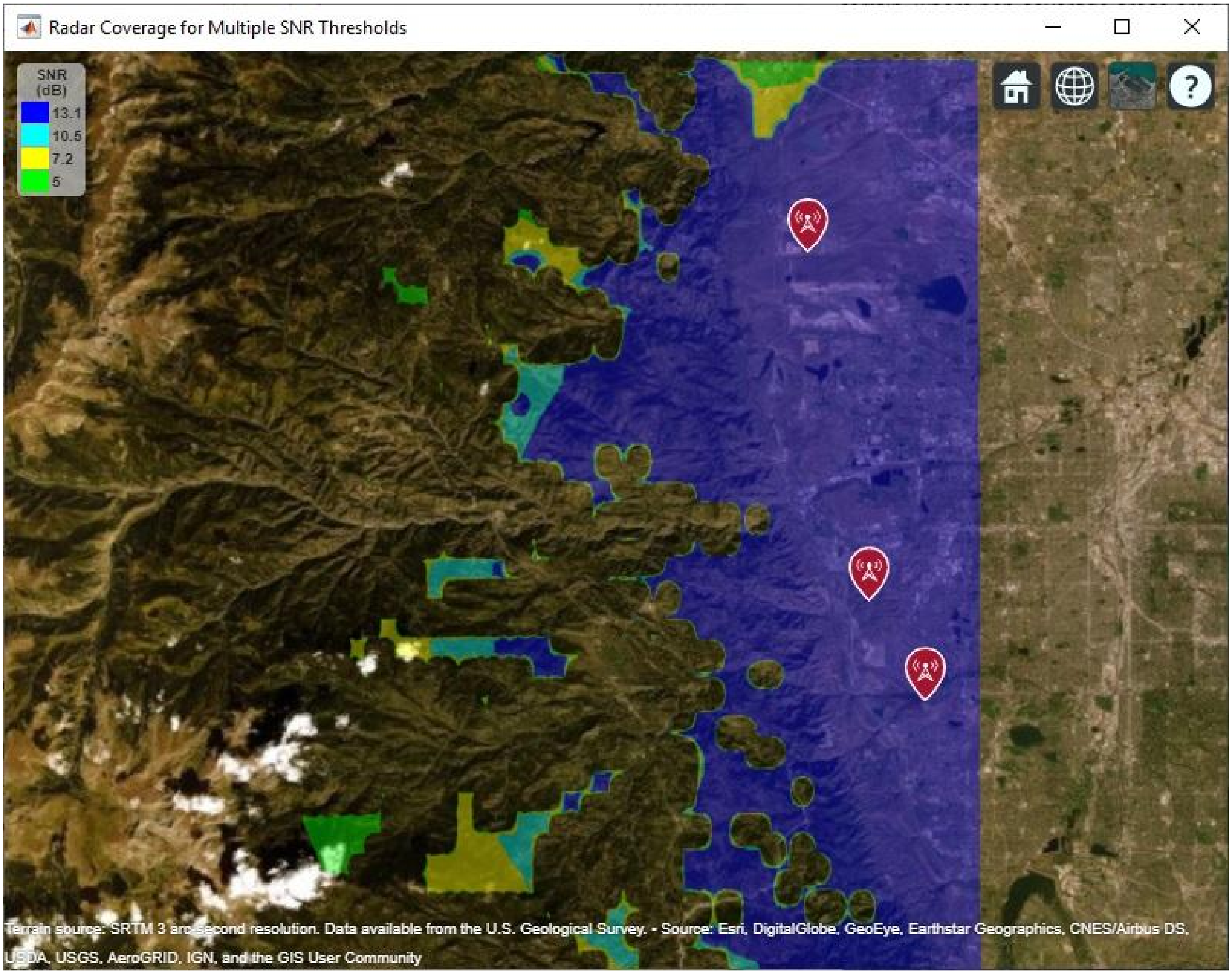
```
end

% 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)
```



Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Show radar coverage map for multiple SNR thresholds.

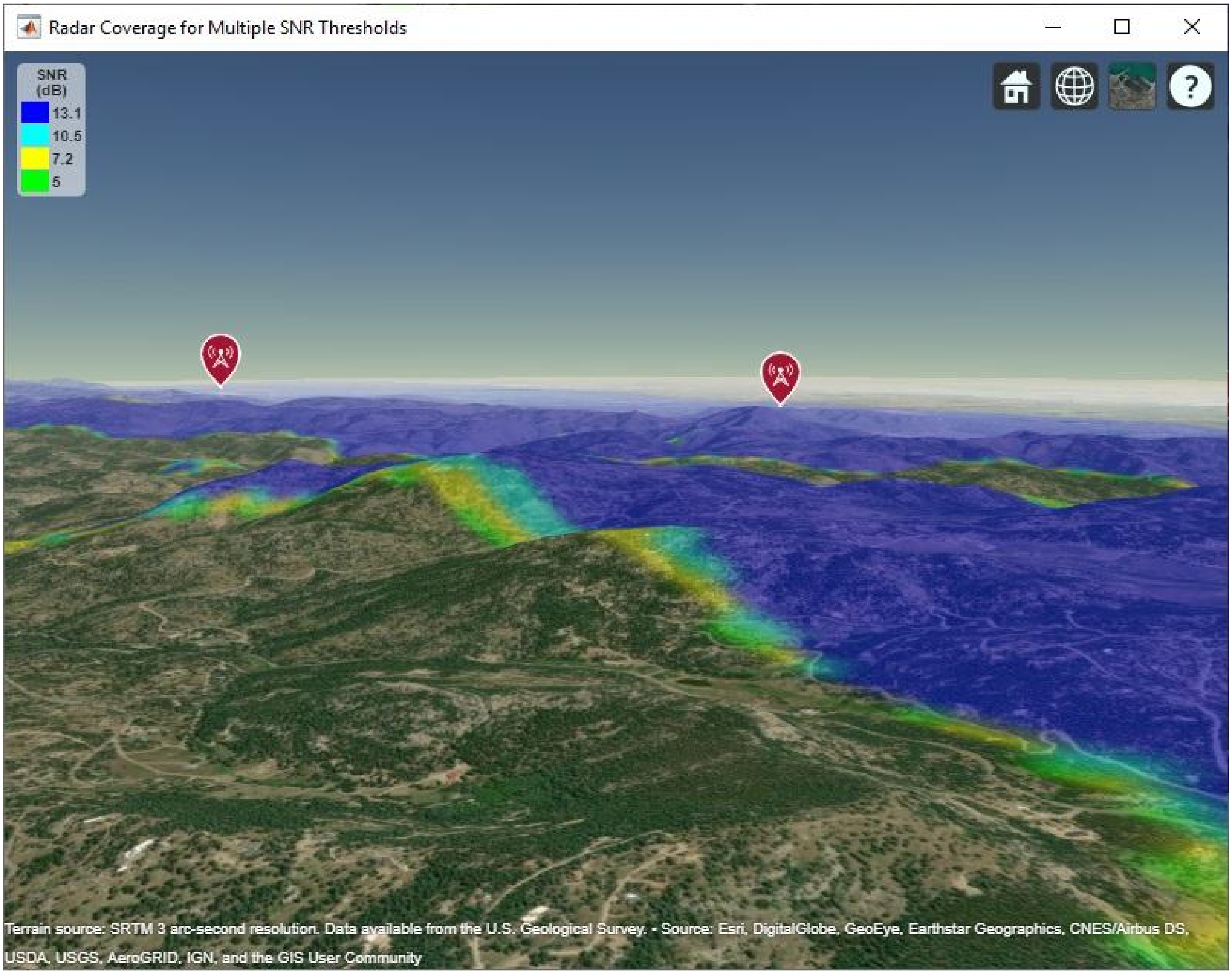
```
% Show best sites
viewer.Name = "Radar Coverage for Multiple SNR Thresholds";
show(rdrtxs(bestsitenums))

contour(rdrData, ...
   "Levels",snrthresholds([1 2 5 10]), ...
   "Colors",colors, ...
   "LegendTitle",legendTitle)
```



Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

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.



Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Clean up by closing Site Viewer and removing the imported terrain data.

```
close(viewer)
removeCustomTerrain("southboulder")
```

Sea Clutter Simulation for Maritime Radar

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

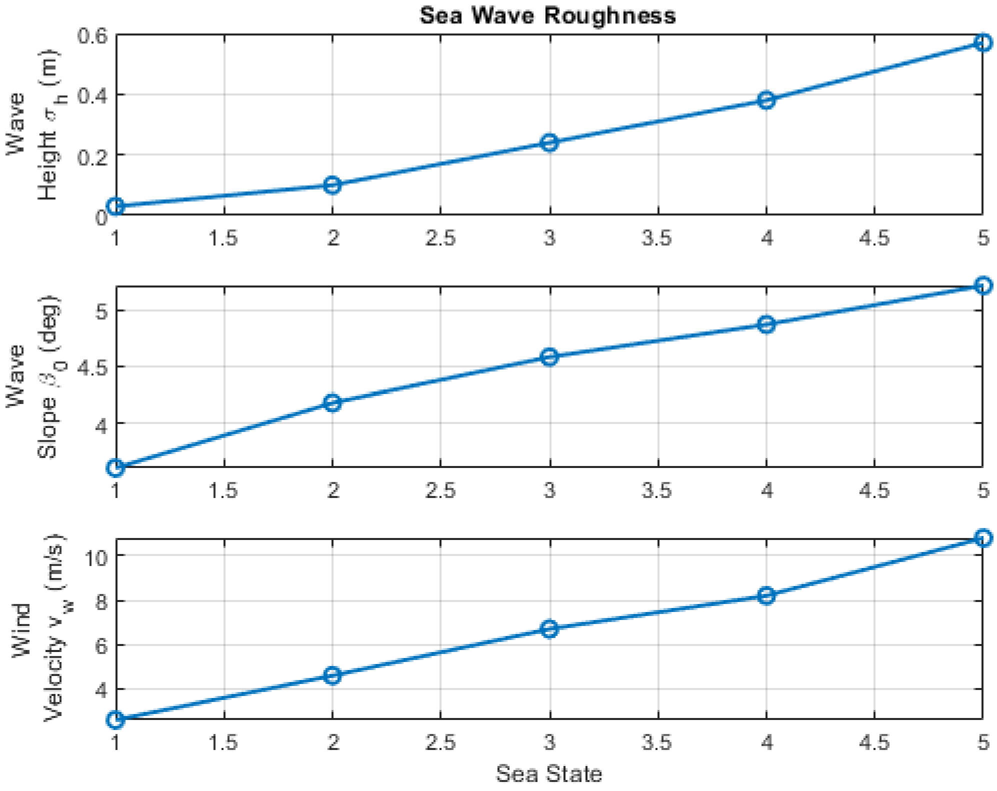
In describing sea clutter, it is important first to establish the physical properties of the sea surface. In modeling sea clutter for radar, there are three important parameters:

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

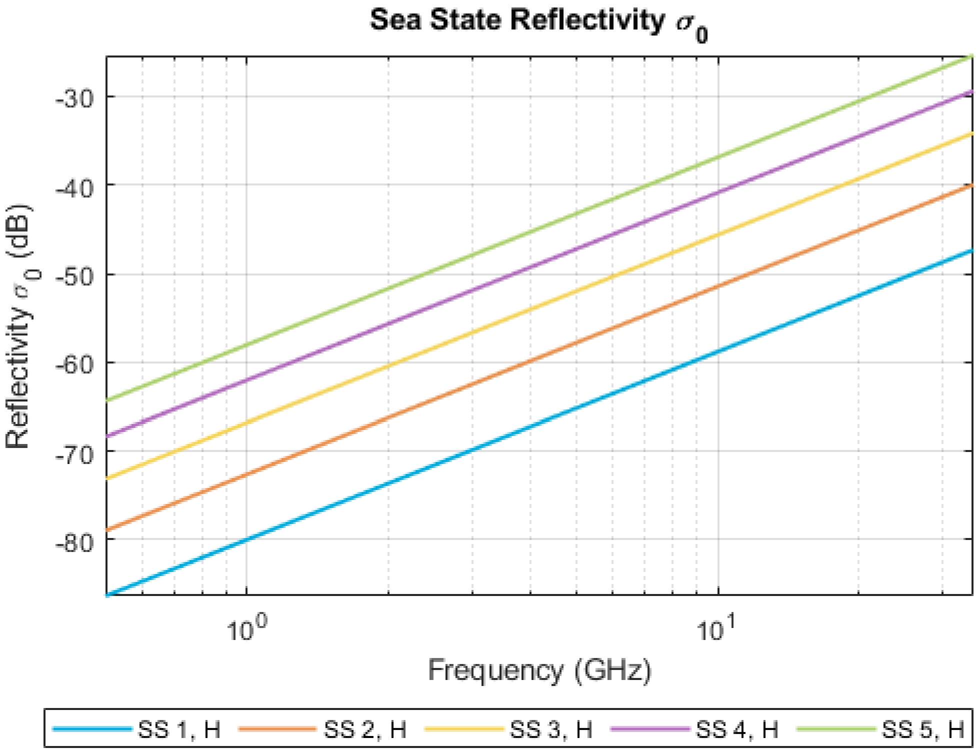
The sea surface is composed of water with an average salinity of about 35 parts per thousand. The reflection coefficient of sea water is close to −1 for microwave frequencies and at low grazing angles.

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);
```

```
% Calculate reflectivity
for is = 1:numSeaStates
    nrcsH(:,is) = seareflectivity(ss(is),grazAng,freq,'Polarization',pol);
end

% Plot reflectivity
helperPlotSeaReflectivity(ss,grazAng,freq,nrcsH,pol);
```



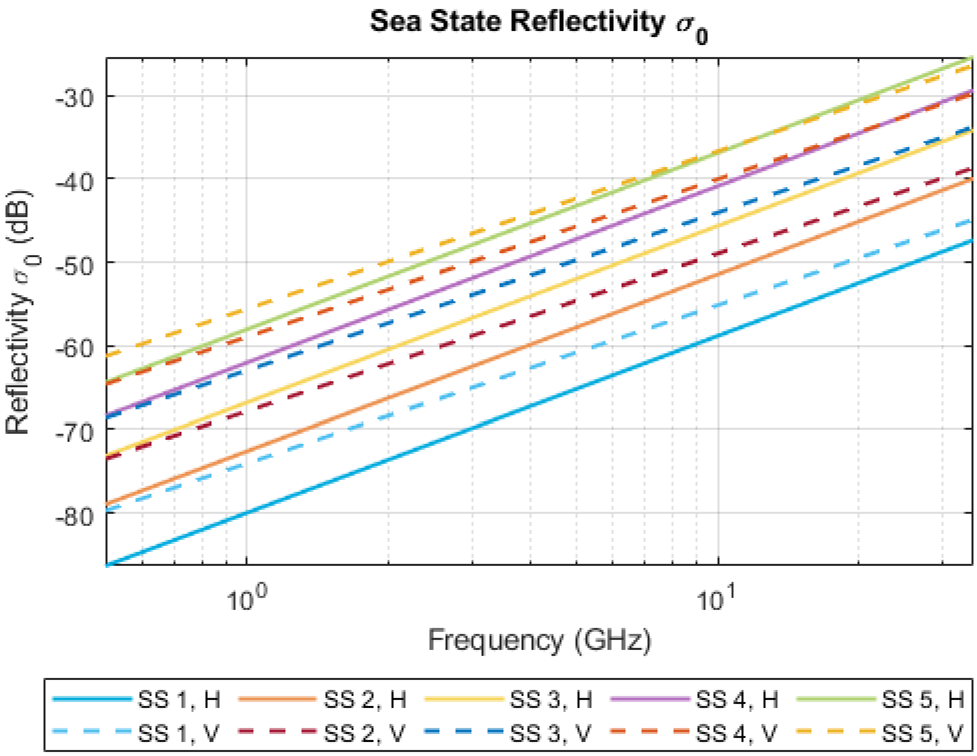
The figure shows that the sea surface reflectivity is proportional to frequency. Additionally, as the sea state number increases, which corresponds to increasing roughness, the reflectivity also increases.

Polarization Effects

Next, consider polarization effects on the sea surface reflectivity. Maintain the same grazing angle and frequency span from the previous section.

```
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);
```



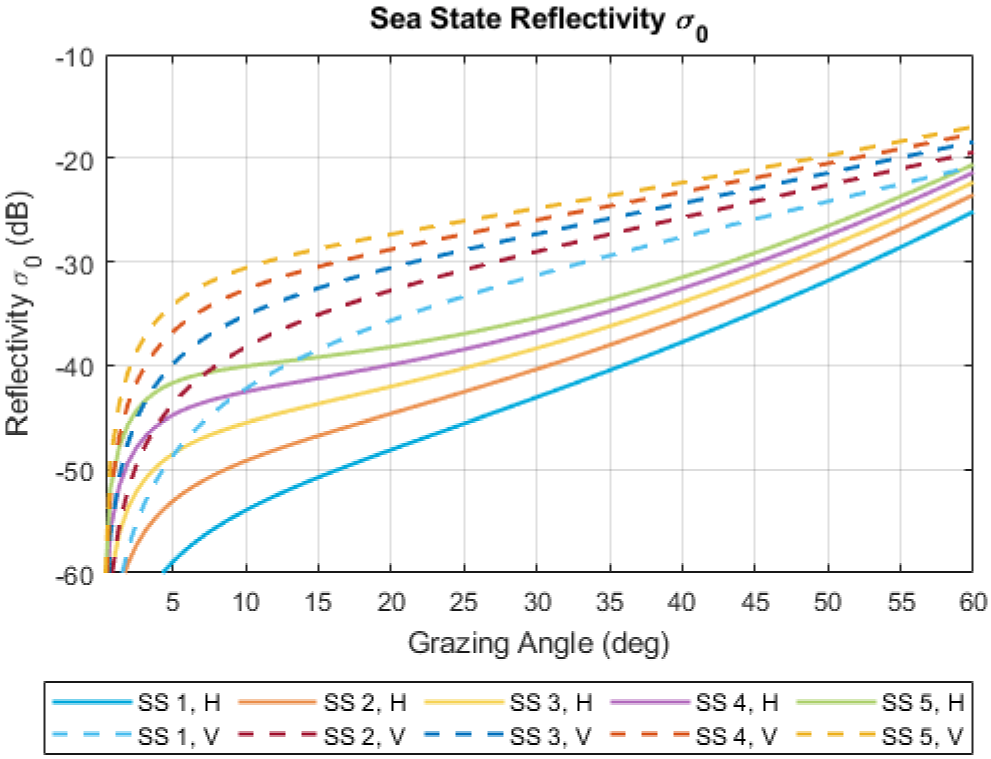
The figure shows that there is a noticeable effect on the reflectivity based on polarization. Notice that the difference between horizontal and vertical polarizations is greater at lower frequencies than at higher frequencies. As the sea state number increases, the difference between horizontal and vertical 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]);
```



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

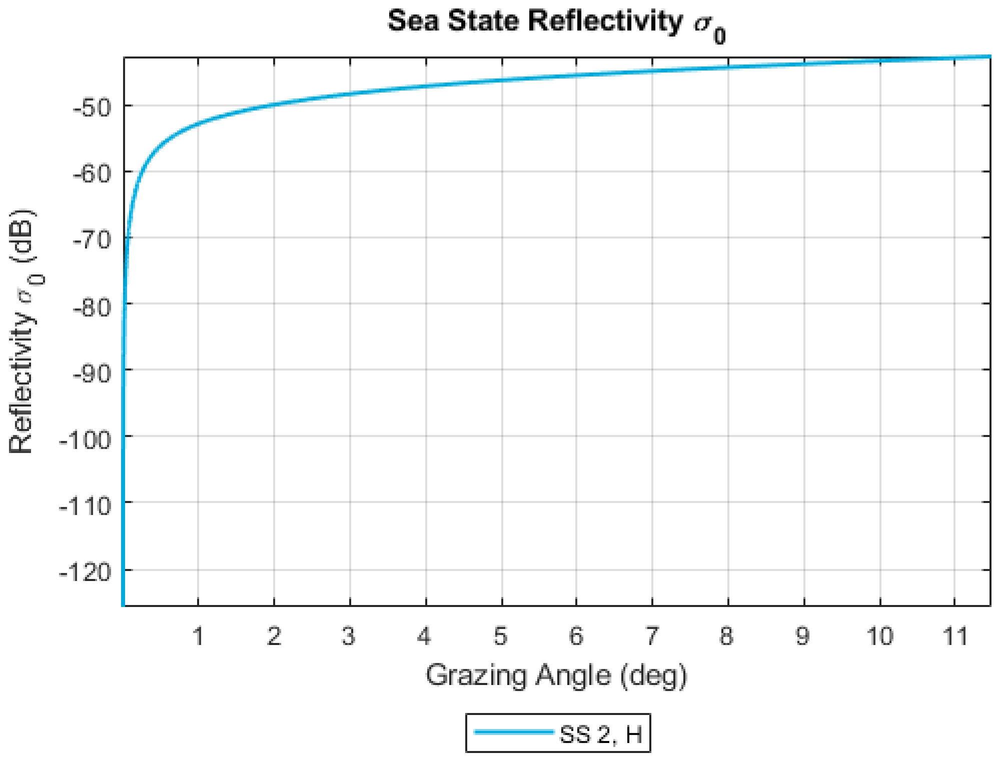
Consider a horizontally polarized maritime surveillance radar system operating at 6 GHz (C-band). Define the radar system.

```
% 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)
```

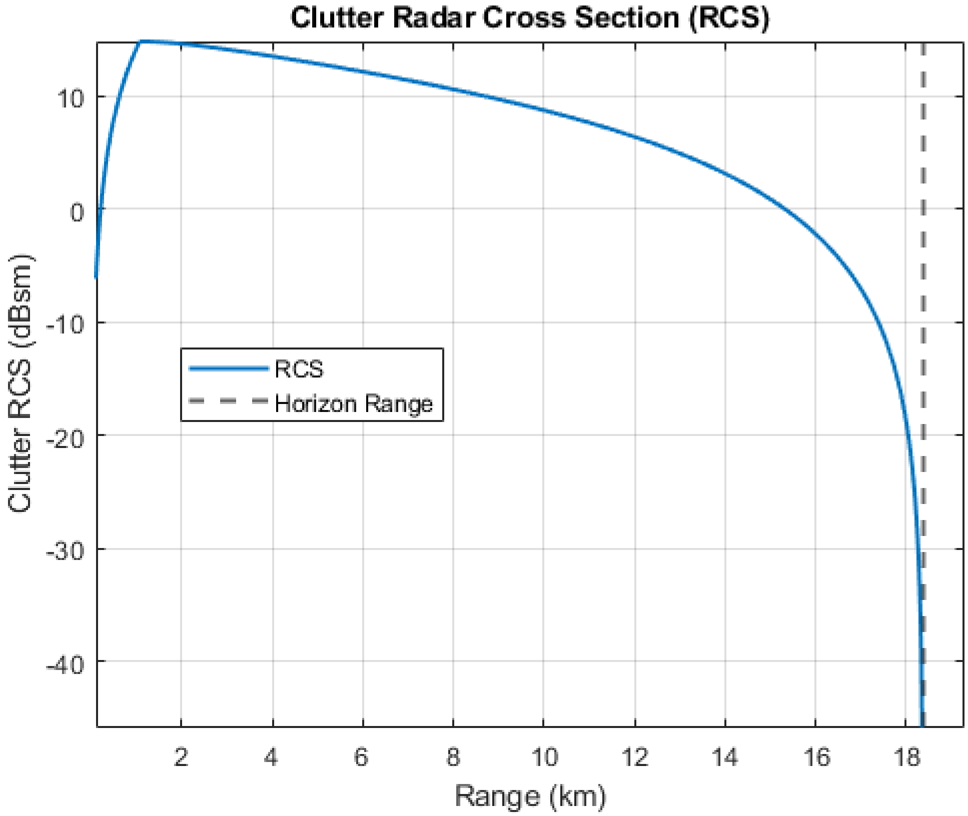
Next, simulate an operational environment where the sea state is 2. Calculate and plot the sea surface reflectivity for the grazing angles of the defined geometry.

```
% 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
grazAng = grazingang(anht,Rm,'TargetHeight',surfht);
nrcs = seareflectivity(ss,grazAng,freq);
helperPlotSeaReflectivity(ss,grazAng,freq,nrcs,'H');
```



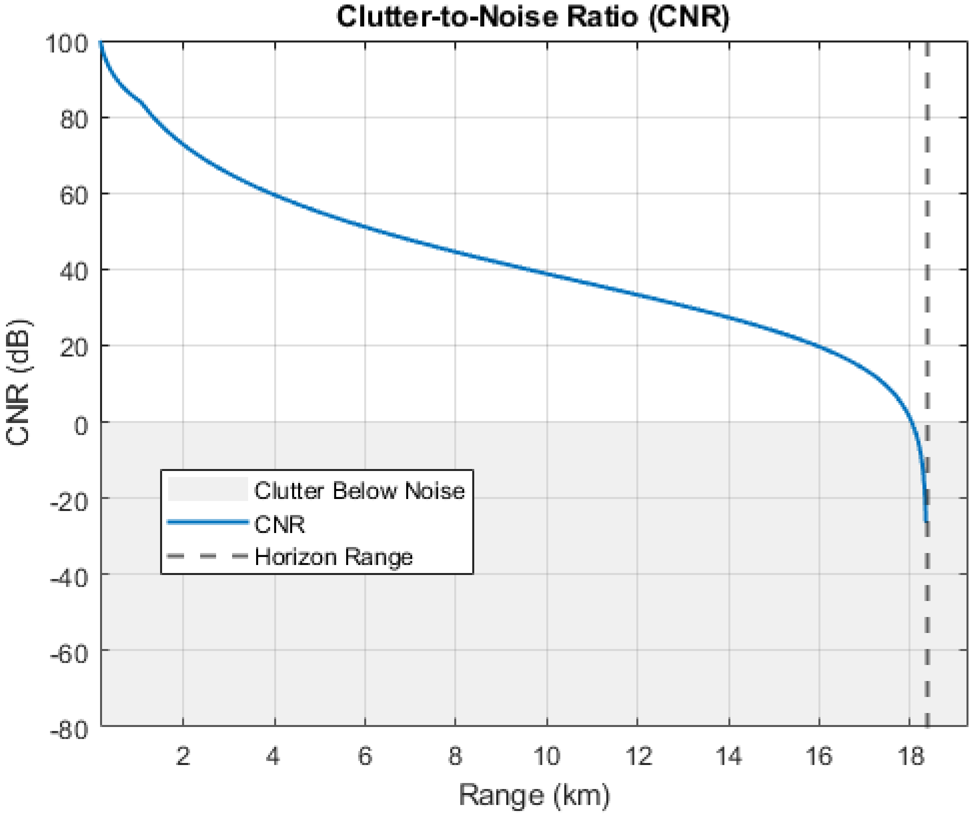
Next, calculate the radar cross section (RCS) of the clutter using the `clutterSurfaceRCS` function. Note the drop in the clutter RCS as the radar horizon range is reached.

```
% 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);
```



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);
```



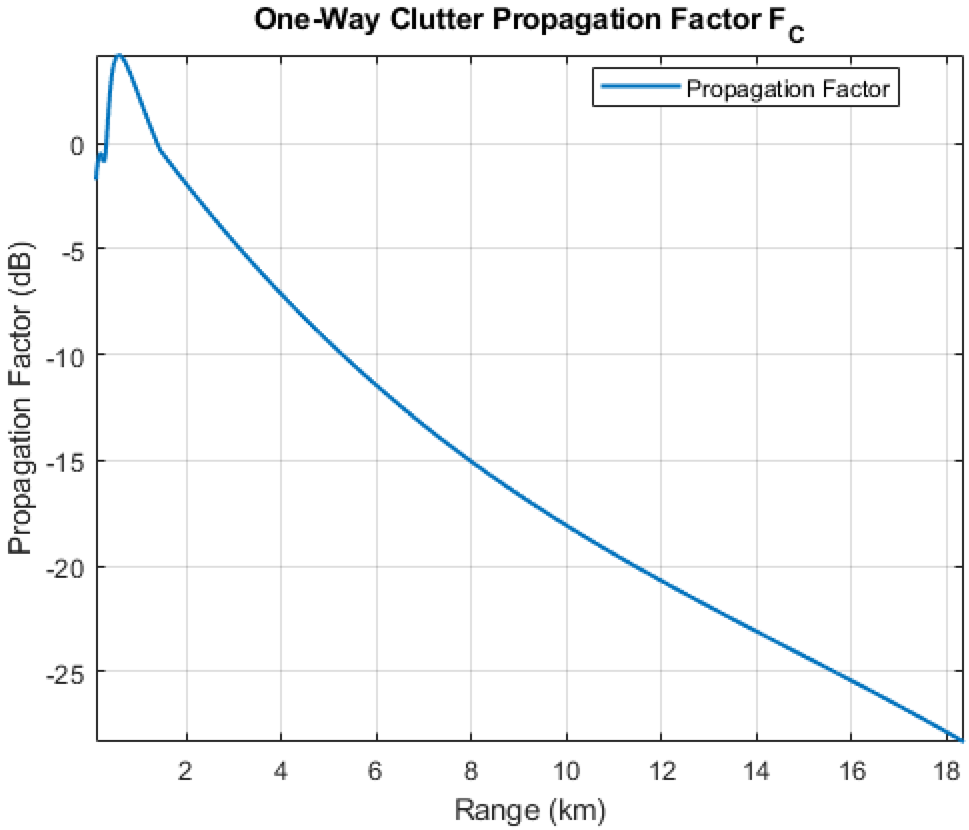
```
% Range when clutter falls below noise
helperFindClutterBelowNoise(Rkm,cnr);
```

```
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');
```

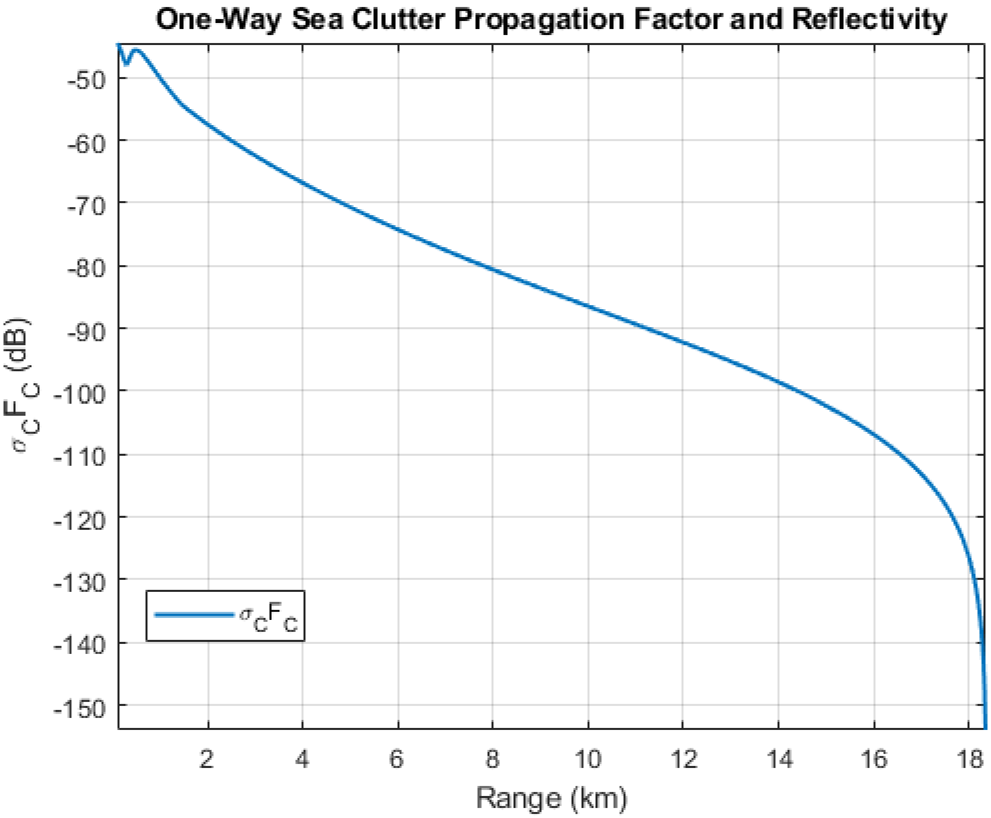


Within the above plot, two propagation regions are visible:

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.

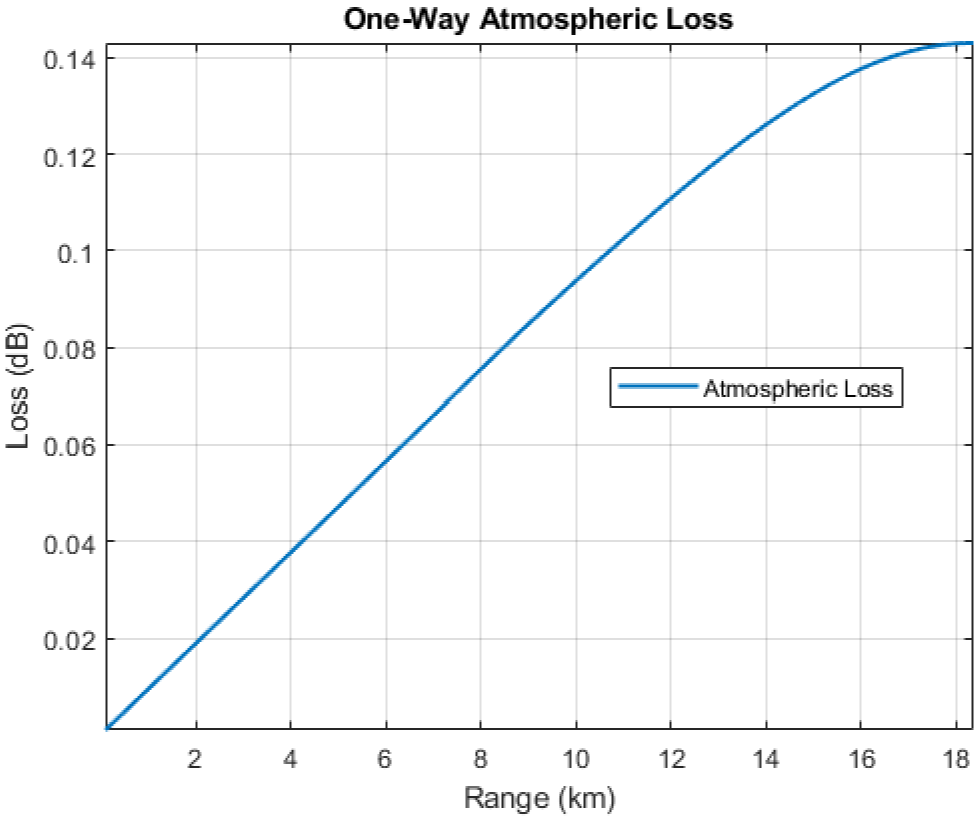
Typically, the clutter propagation factor and the sea reflectivity are combined as the product $\sigma_C F_C^4$, because measurements of surface reflectivity are generally measurements of the product rather than just the reflectivity σ_C. Calculate this product and plot the results.

```
% 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');
```



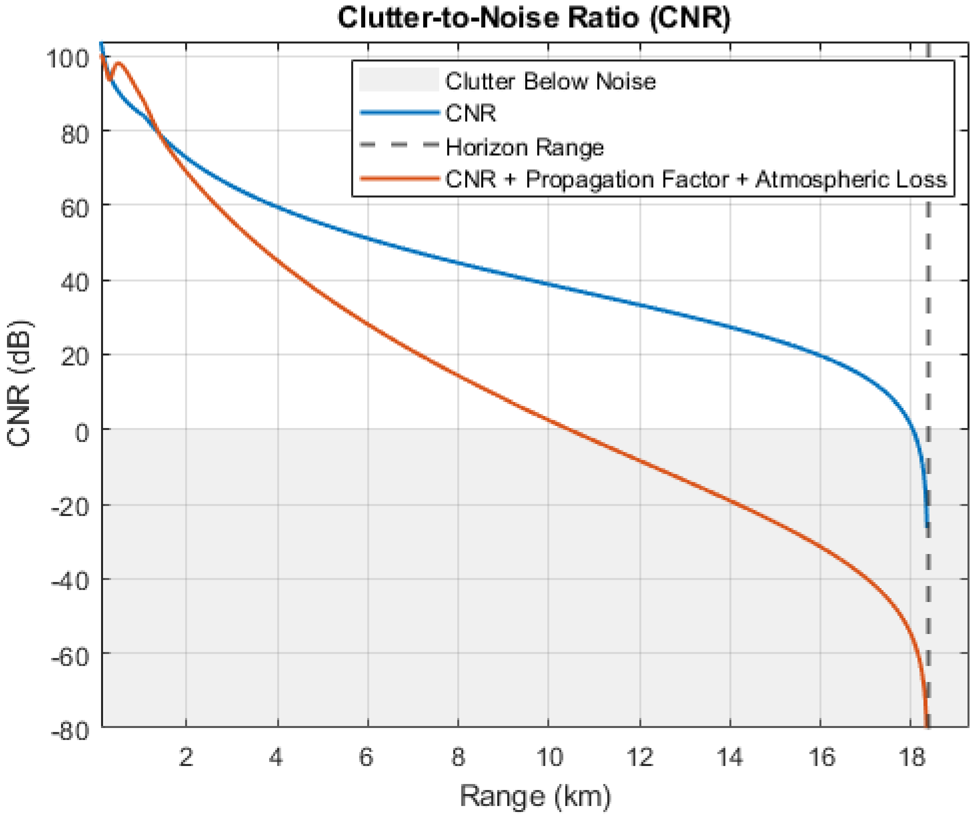
Next, calculate the atmospheric loss on the path using the slant-path `tropopl` function. Use the default standard atmospheric model for the calculation.

```
% 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');
```



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);
```



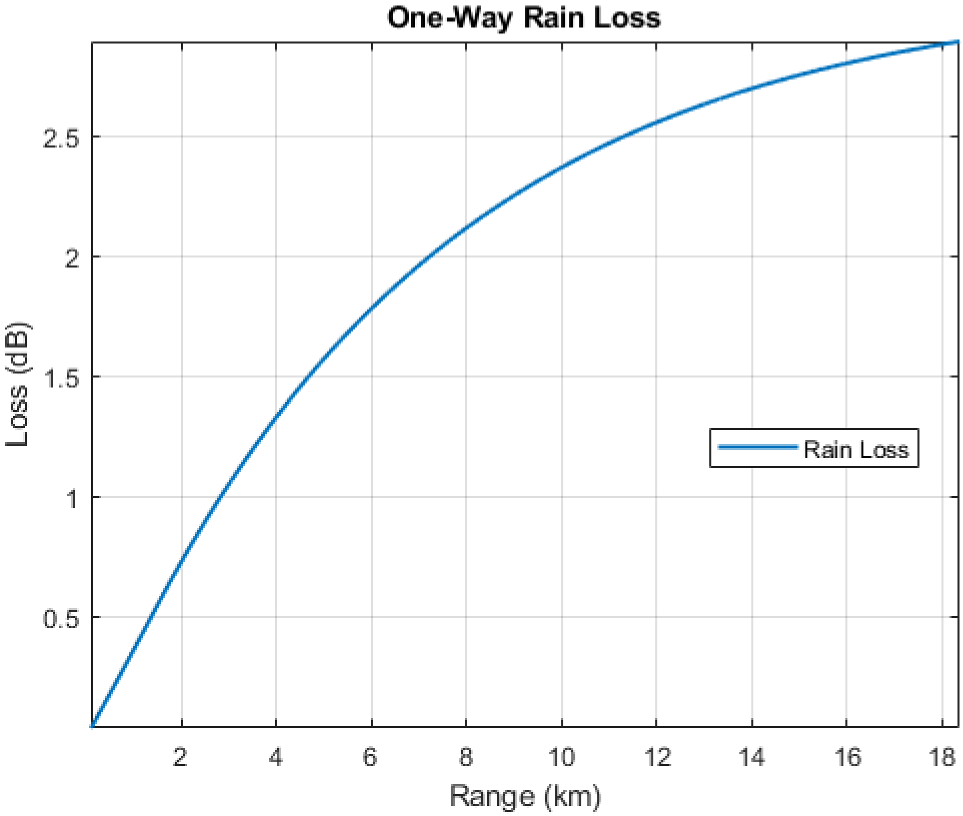
```
% Range when clutter falls below noise
helperFindClutterBelowNoise(Rkm,cnr);
```

```
Range at which clutter falls below noise (km) = 10.44
```

Understanding Weather Effects

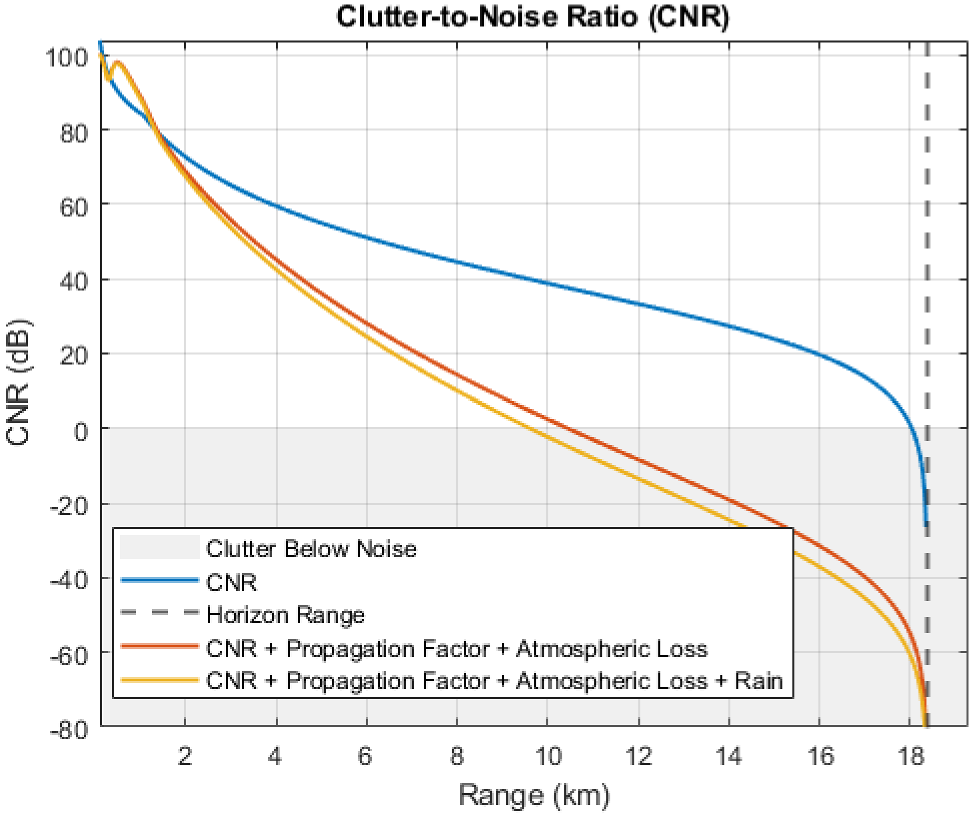
Just as the atmosphere affects the detection of a target, weather also affects the detection of clutter. Consider the effect of rain over the simulated ranges. First calculate the rain attenuation.

```
% 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.

```
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);
    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
```

```
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: {}
```

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:

      Pattern: [2x2 double]
      Azimuth: [-180 180]
    Elevation: [2x1 double]
    Frequency: [0 1.0000e+20]
```

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

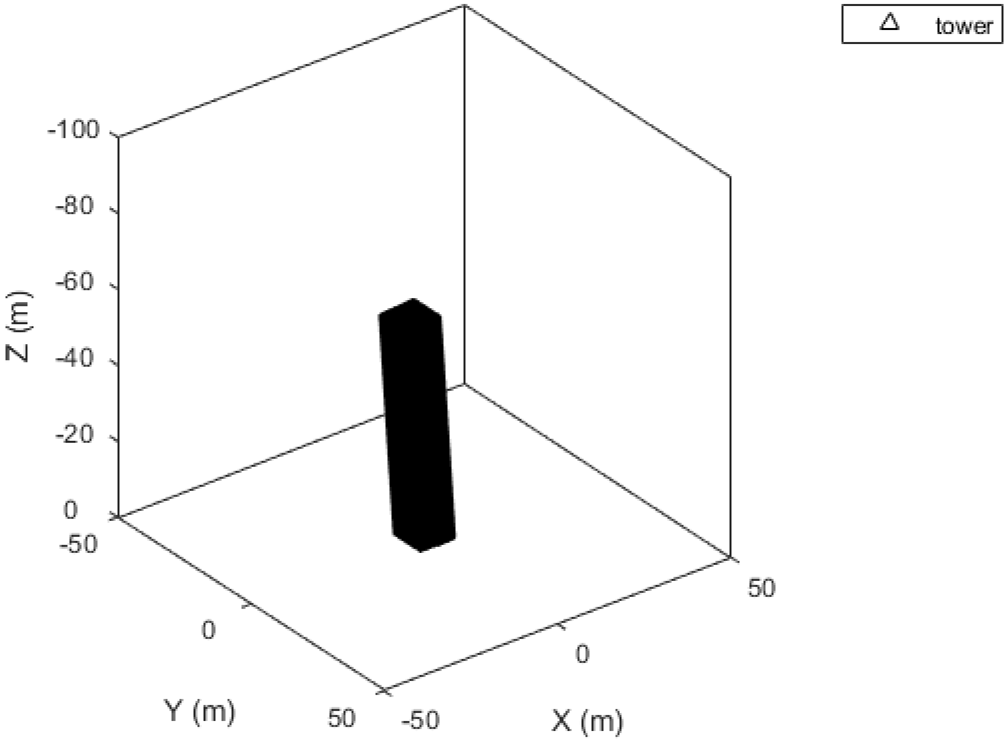
The `theaterPlot` object provides an interface to plot objects dynamically in a 3-D scene. You may use standard MATLAB axes plotting methods to add or remove annotations to the theater plot's axes, which you can obtain via its `Parent` property.

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.

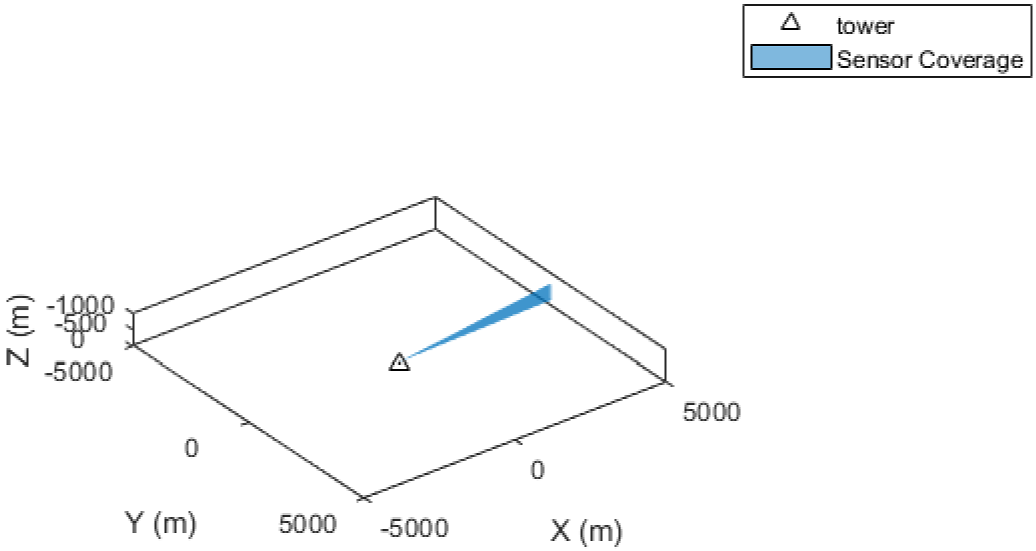
The radarDataGenerator has the option to report detections in scenario coordinates. Reporting detections in scenario coordinates makes it easier to compare the generated detections with the positions of the objects in the scenario.

```
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

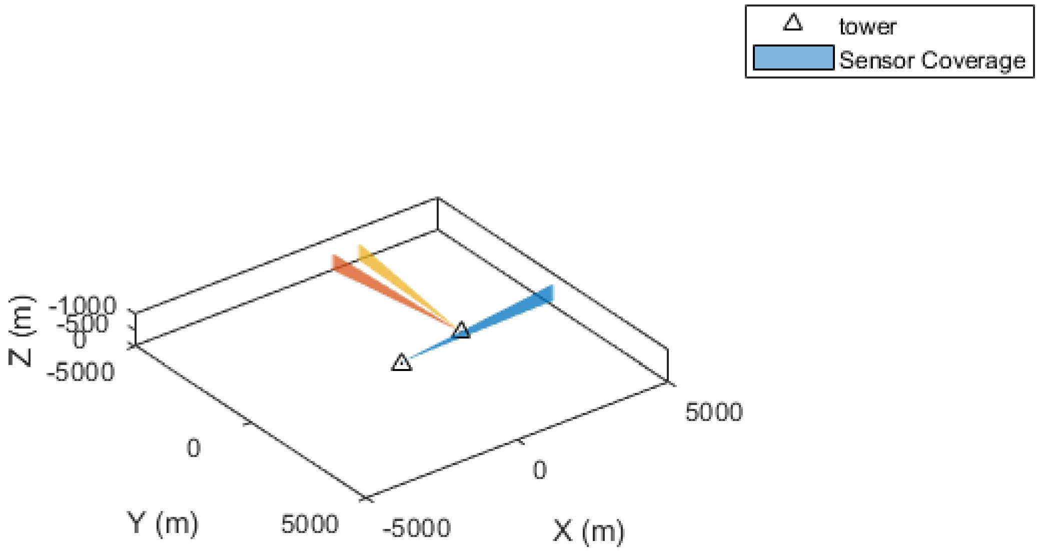
You can arrange for the helicopter to cross the path of the radar beam. This shows how to make the helicopter follow a straight 100-meter path at a constant velocity with an elevation of 250 meters for seven seconds:

```
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

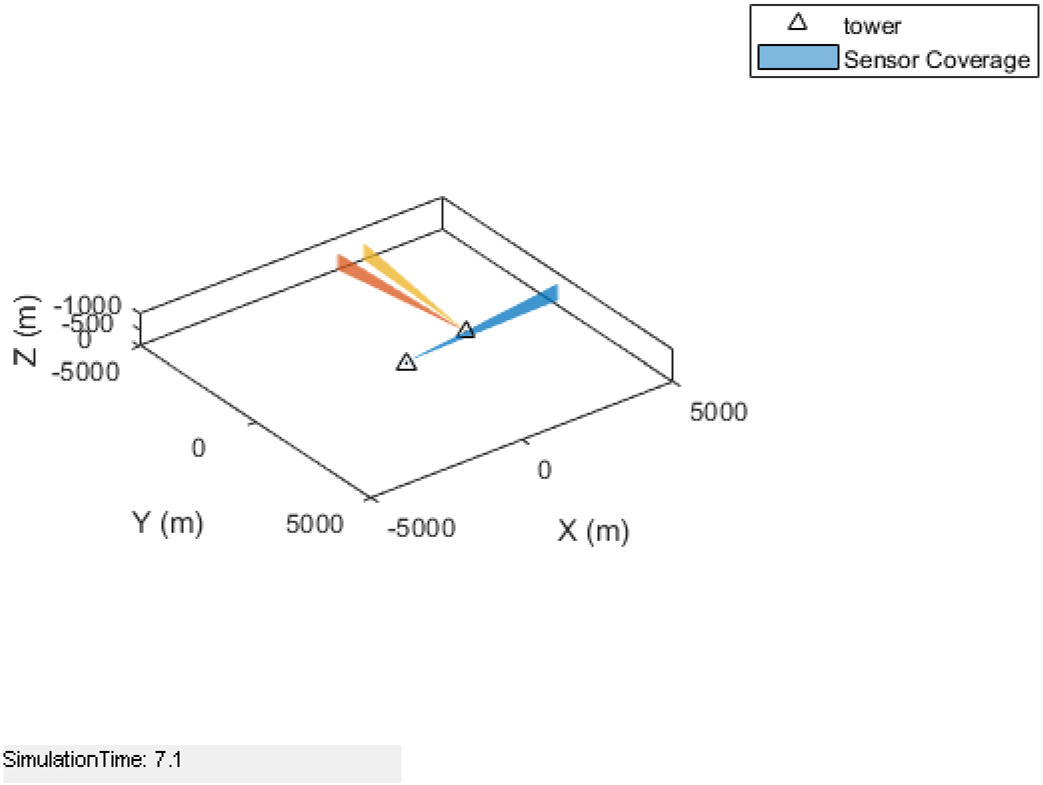
In the example above, you added three radars with different update rates: the tower has an update rate of 10 Hz, and the helicopter had two radars with update rates of 20 Hz and 30 Hz, respectively.

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;
```

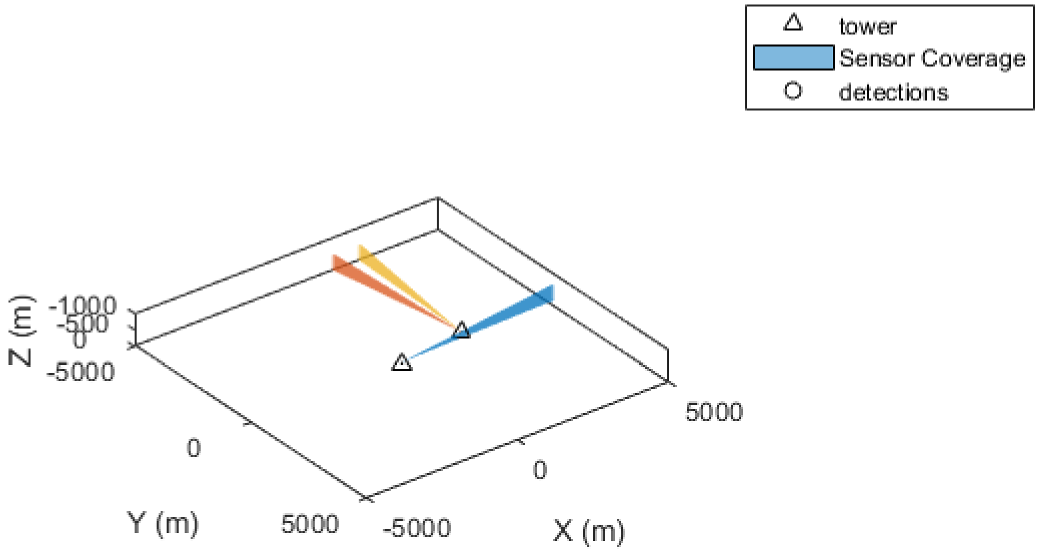
To show the simulation time, add a UI control to the figure.

```
fig = ancestor(tPlot.Parent,'Figure');
timeReadout = uicontrol(fig,'Style','text','HorizontalAlignment','left','Position',[0 0 200 20])
timeReadout.String = "SimulationTime: " + scenario.SimulationTime;
```



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');
```

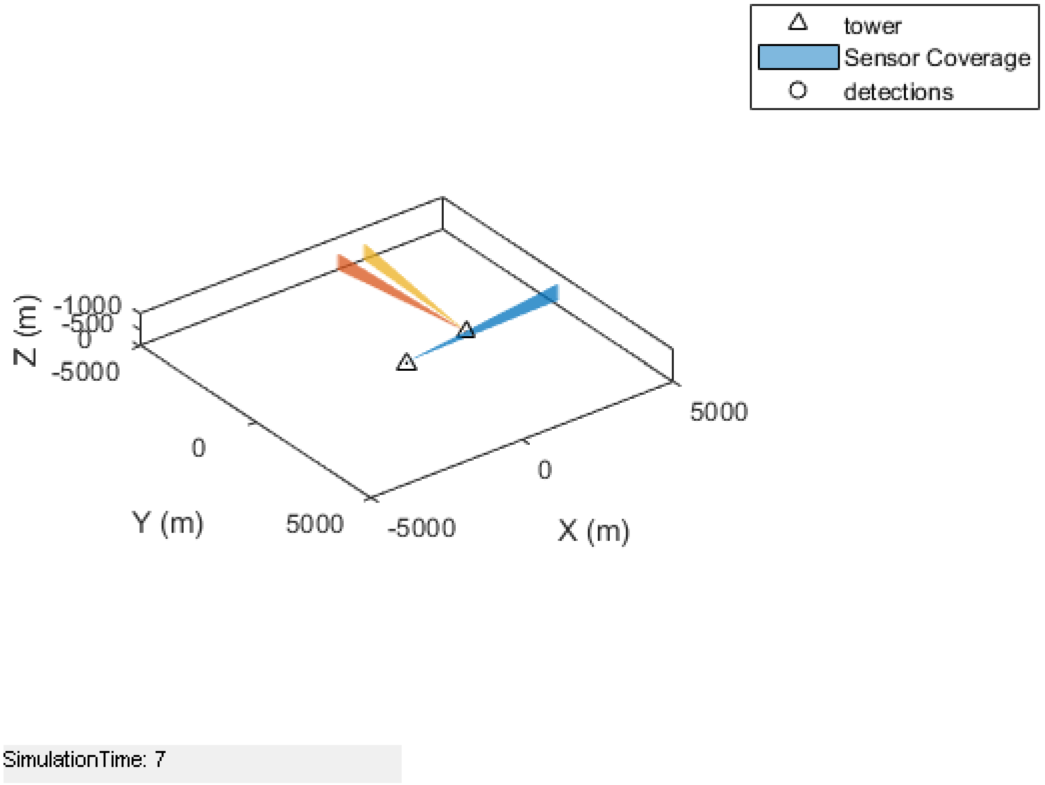


SimulationTime: 7.1

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.

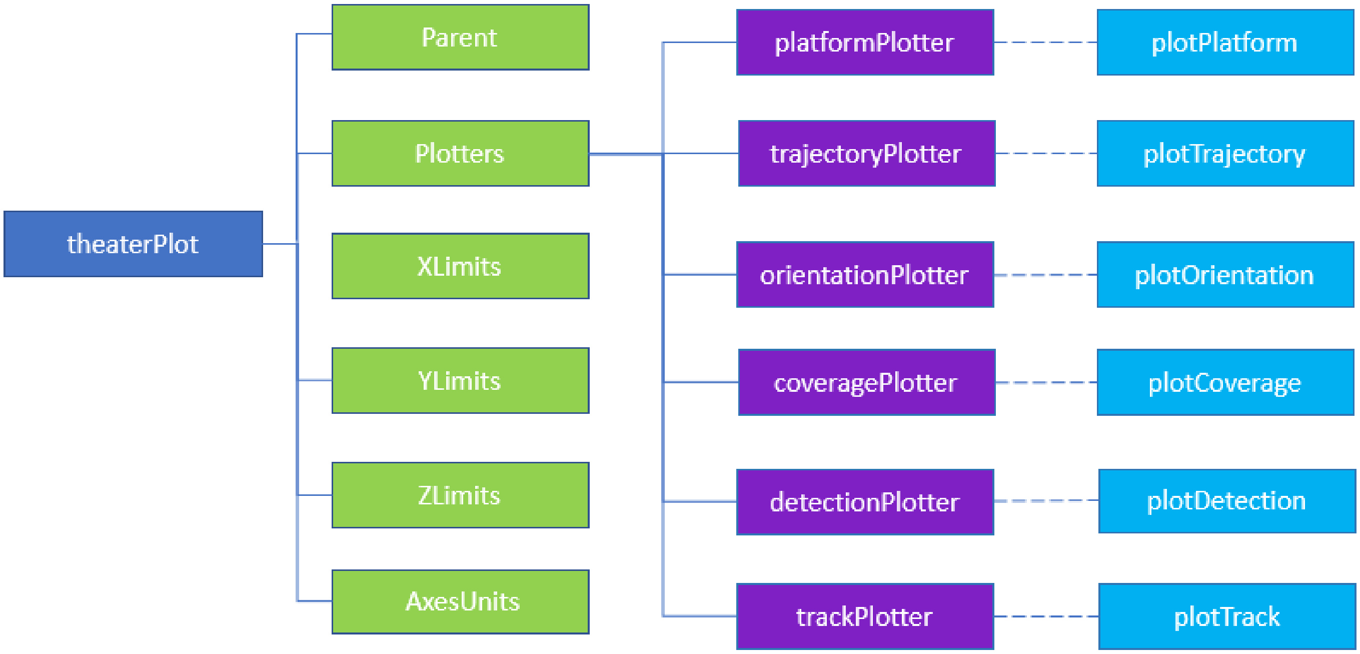
Use theaterPlot to Visualize Radar Scenario

This example shows how to use `theaterPlot` 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 pair 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

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 even though you do not necessarily need a radarScenario 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 a Trajectory Plotter and a Platform Plotter for a 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 a Platform with a 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.

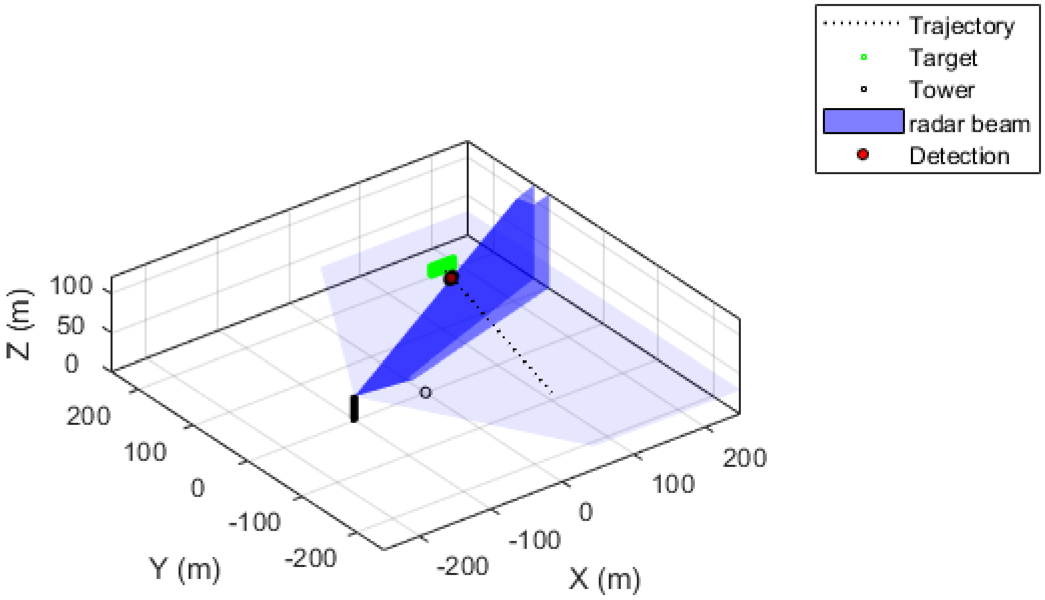
```
detPlotter = detectionPlotter(tp,'DisplayName','Detection','MarkerFaceColor','r','MarkerSize',4)
```

Run the scenario and update the 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 approriate 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.

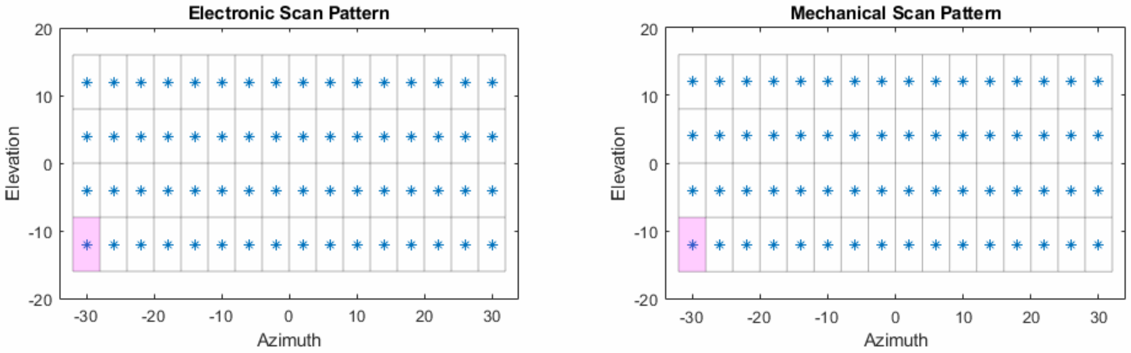
Scenario Management

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 measureables (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 measureable. 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).

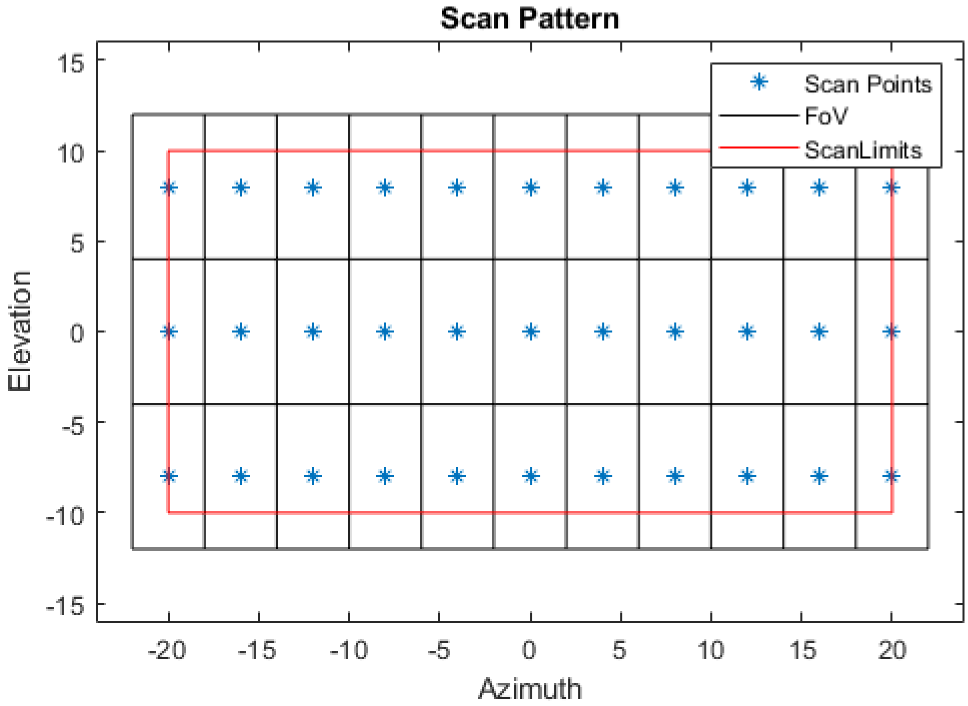
```
pitch = elLimits(1); % mount the antenna rotated upwards
```

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','Detec
    '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],'Velocit
```

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

    8.0854    0.0051
```

```
rangeMargin = (Rd - mean(tgtRangeTruth,1))/1e3 % km
```

```
rangeMargin = 1×2

   23.5360    0.0074
```

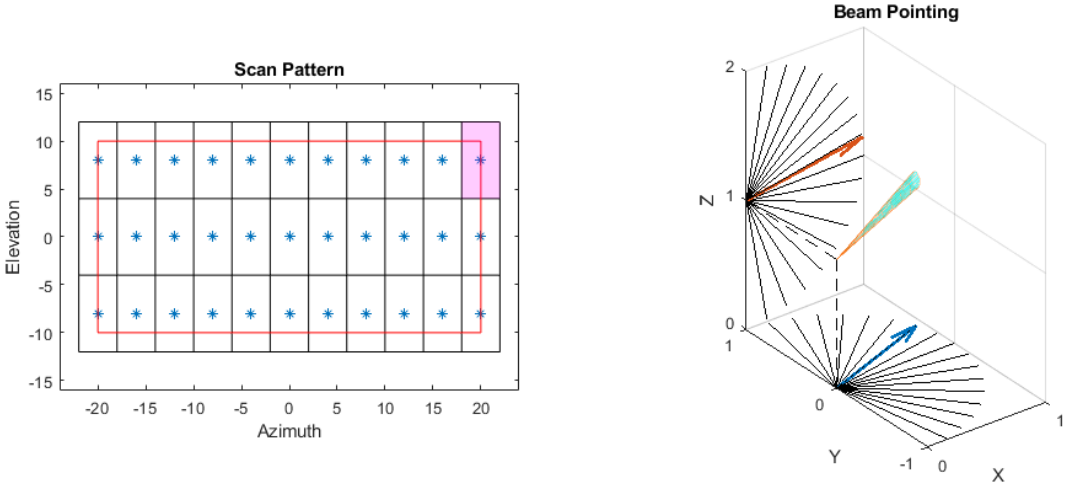
The first target is about 8 dB brighter than needed for detection with the given CFAR parameters, while the second target is considered barely detectable.

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.

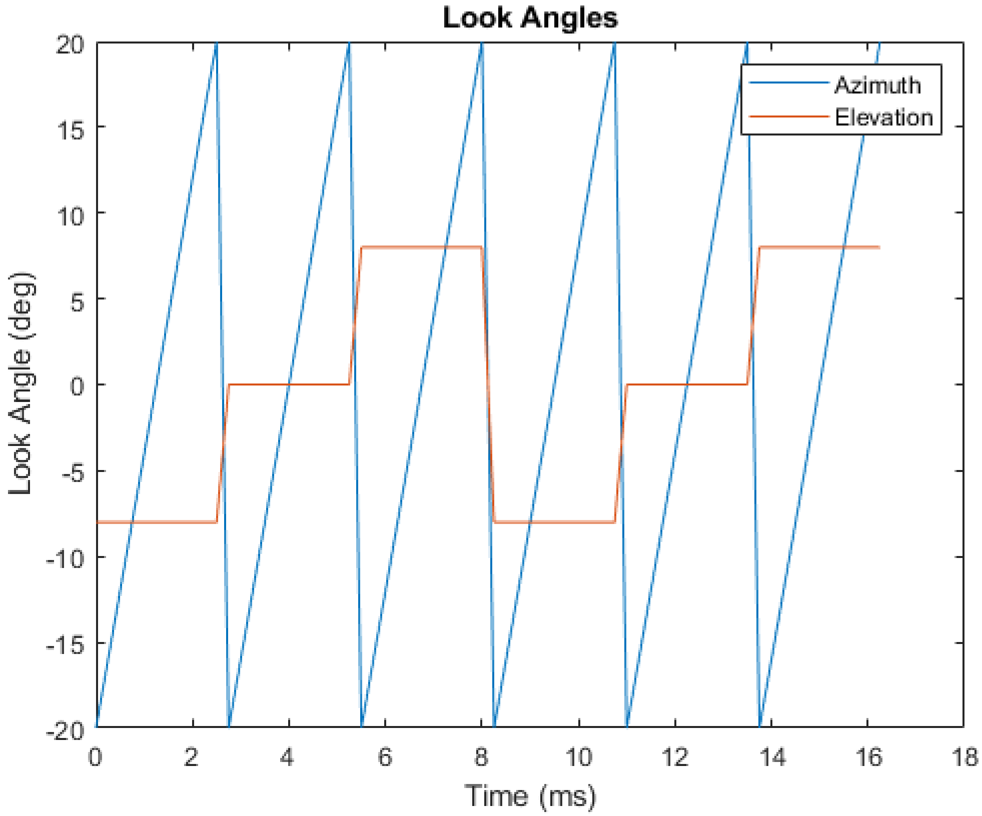
```
allDets = []; % initialize list of all detections generated
lookAngs = zeros(2,numFrames);
simTime = zeros(1,numFrames);
frame = 0; % current frame
while advance(scene) % advance until StopTime is reached

    % 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.

```
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: 7.5000e-04
              Measurement: [3x1 double]
         MeasurementNoise: [3x3 double]
              SensorIndex: 1
            ObjectClassID: 0
    MeasurementParameters: [1x1 struct]
         ObjectAttributes: {[1x1 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: 12.5894
```

`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.

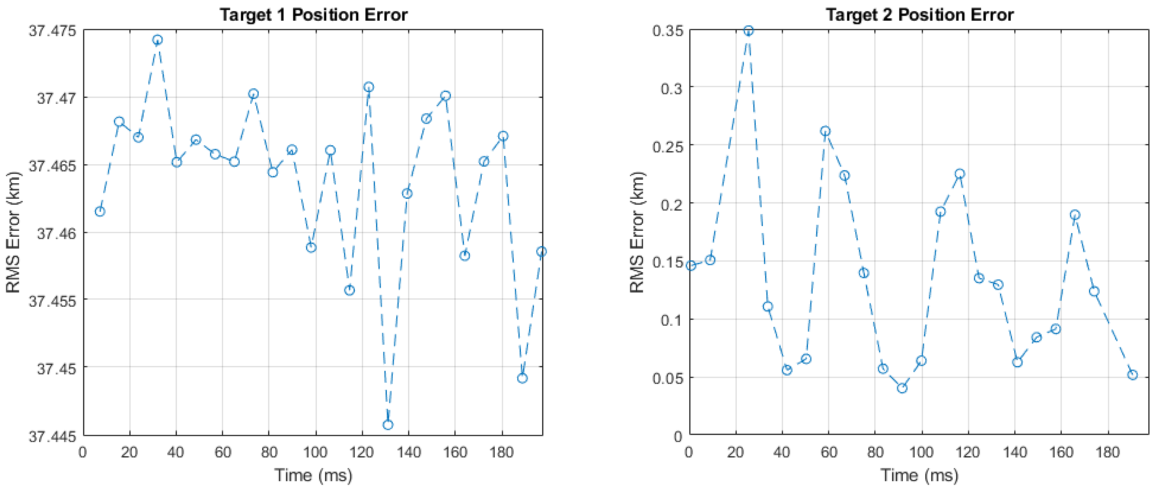
```
numFAs = numel(find(detTgtIdx==-1))
```

```
numFAs = 7
```

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.

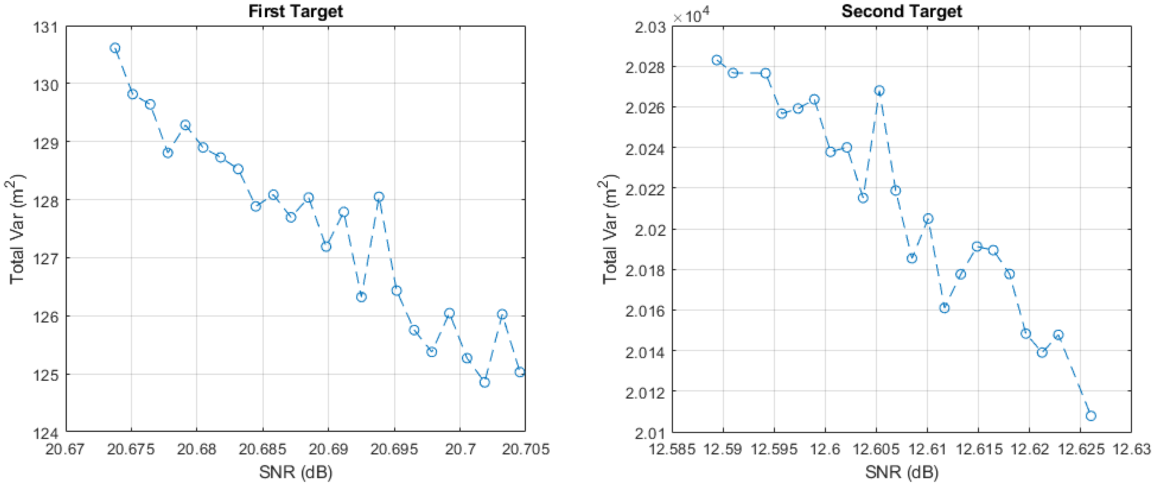
```
tgtPosObs = cell2mat(cellfun(@(t) t.Measurement,allDets,'UniformOutput',0).');
subplot(1,2,1);
helperPlotPositionError( tgtPosTruth(:,:,1),time,tgtPosObs(:,firstTgtDet),detTime(firstTgtDet),sc
title('Target 1 Position Error');
subplot(1,2,2);
helperPlotPositionError( tgtPosTruth(:,:,2),time,tgtPosObs(:,secondTgtDet),detTime(secondTgtDet)
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: [6x1 double]
     StateCovariance: [6x6 double]
     StateParameters: [1x1 struct]
       ObjectClassID: 0
          TrackLogic: 'History'
     TrackLogicState: [1 1 0 0 0]
         IsConfirmed: 1
           IsCoasted: 0
      IsSelfReported: 1
    ObjectAttributes: [1x1 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: 18.5910
```

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
```

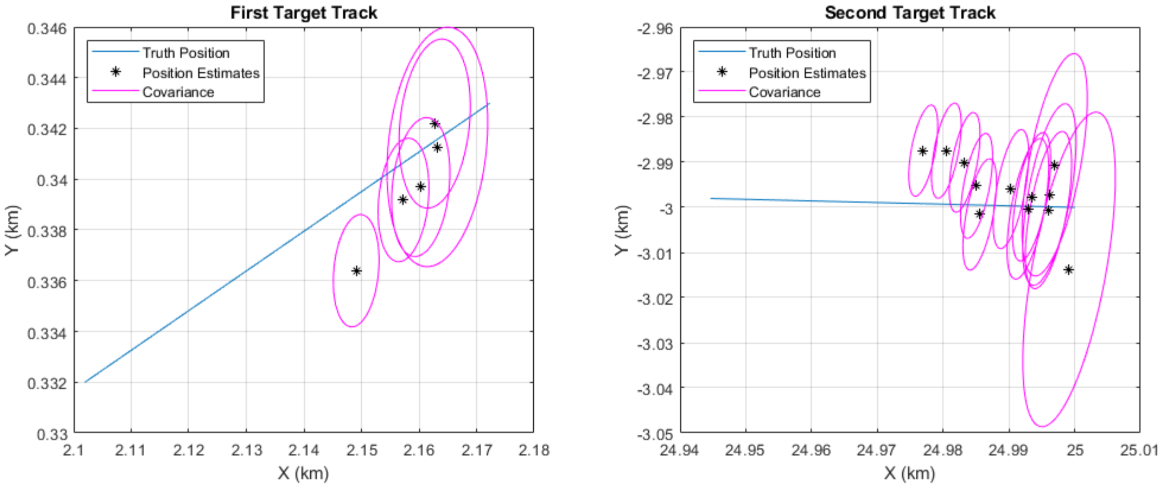
This is roughly the number of non-FA detections generated in the first part.

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.

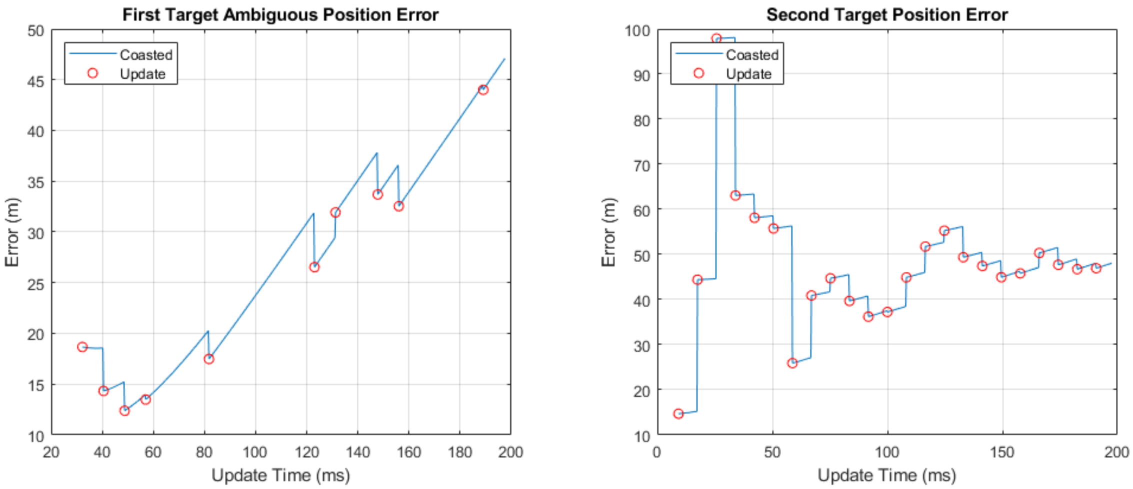
```
figure;
subplot(1,2,1);
helperPlotPositionTrack( tgtPosTruthAmb,allTracks(firstTgtTrack) );
title('First Target Track');
subplot(1,2,2);
helperPlotPositionTrack( tgtPosTruth(:,:,2),allTracks(secondTgtTrack) );
title('Second Target Track');
set(gcf,'Position',get(gcf,'Position')+[0 0 600 0]);
```



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(secon
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');
```

```
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:

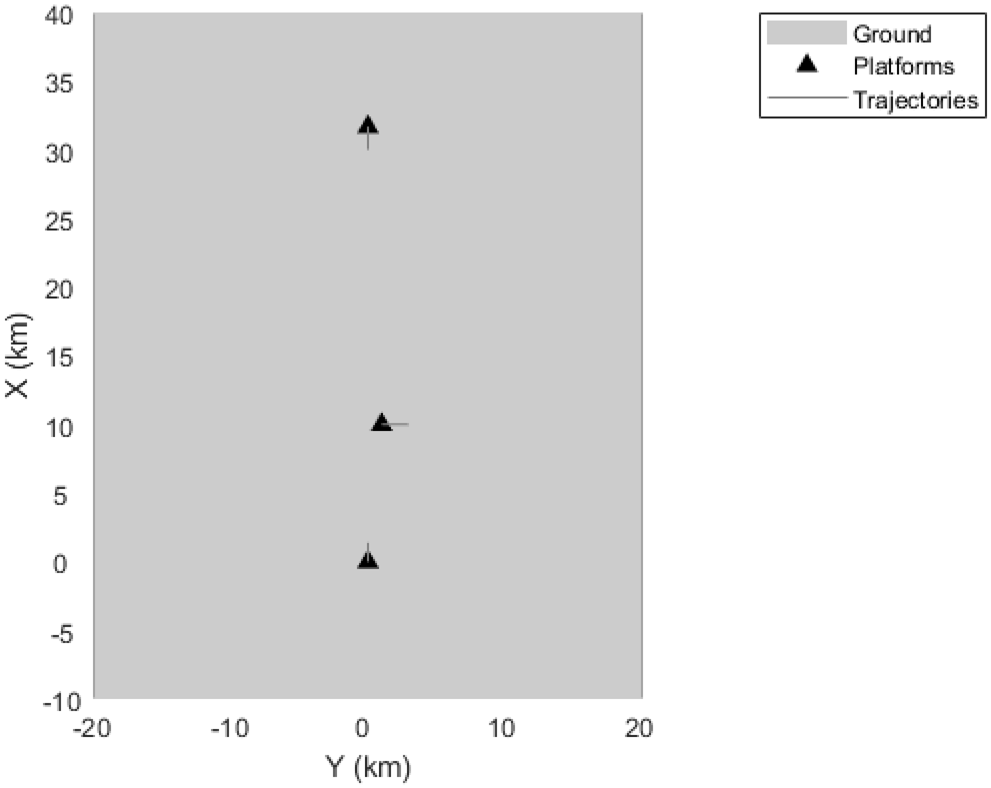
1. Airborne platform flying north at 500 km/h from the bottom of the scenario
2. Airborne platform flying south at 600 km/h from the top of the scenario
3. Airborne platform flying east at 700 km/h in the middle of the scenario

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] , 'Y
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'
```

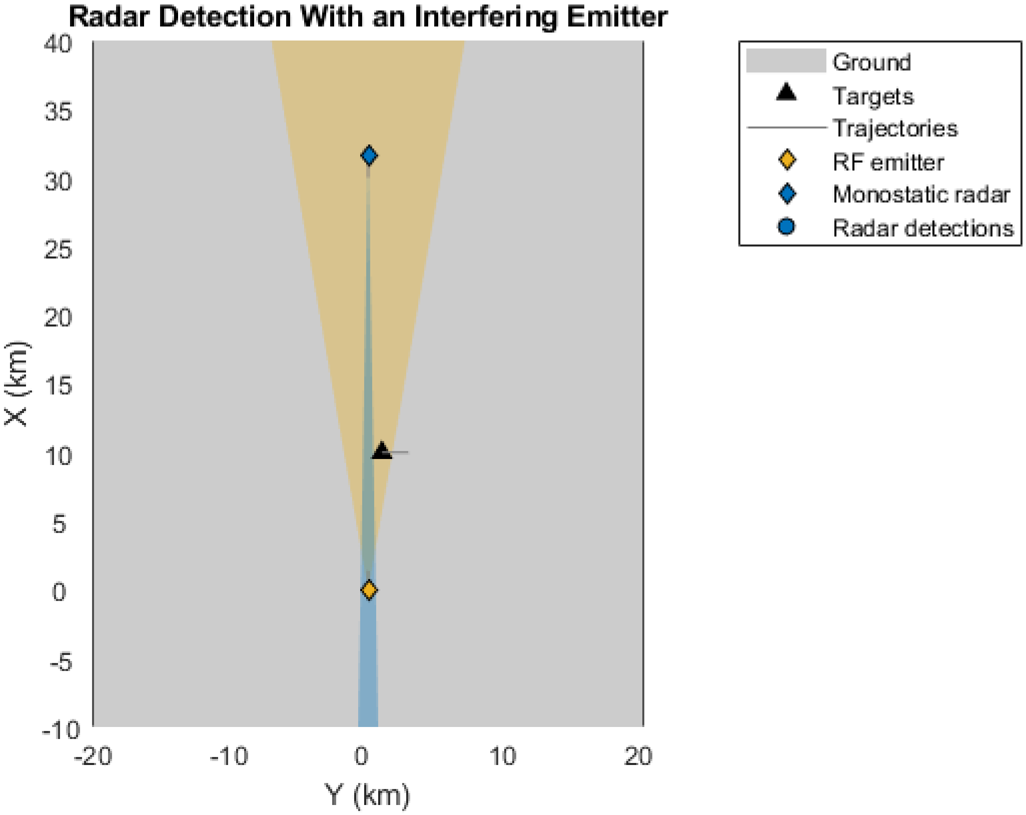
```
radar = 
  radarDataGenerator with properties:

              SensorIndex: 2
               UpdateRate: 12.5000
            DetectionMode: 'Monostatic'
                 ScanMode: 'Mechanical'
    InterferenceInputPort: 1
       EmissionsInputPort: 0

         MountingLocation: [0 0 0]
           MountingAngles: [0 0 0]

              FieldOfView: [2 10]
              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','Marke
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});
```



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`.

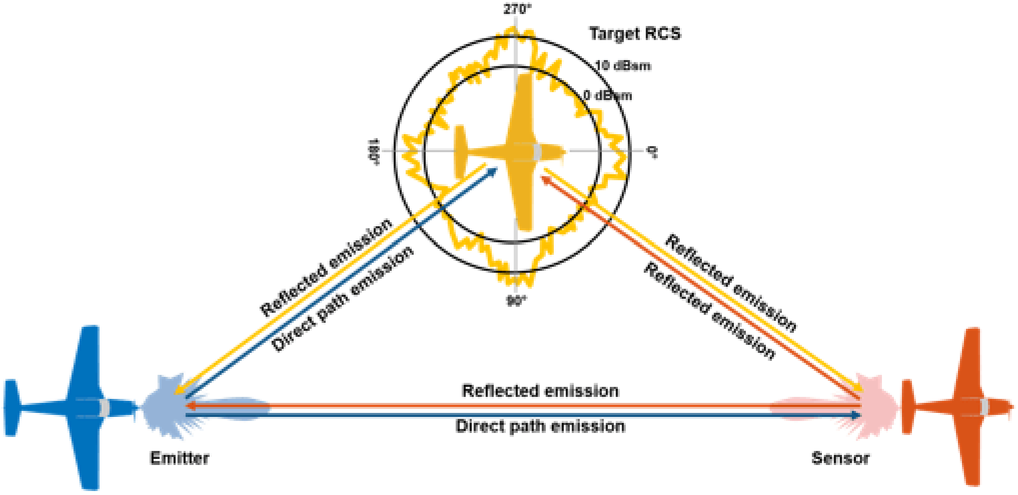
```
scene.UpdateRate = 0
```

```
scene = 
  radarScenario with properties:

     IsEarthCentered: 0
          UpdateRate: 0
      SimulationTime: 0
            StopTime: 10
    SimulationStatus: NotStarted
           Platforms: {1x3 cell}
```

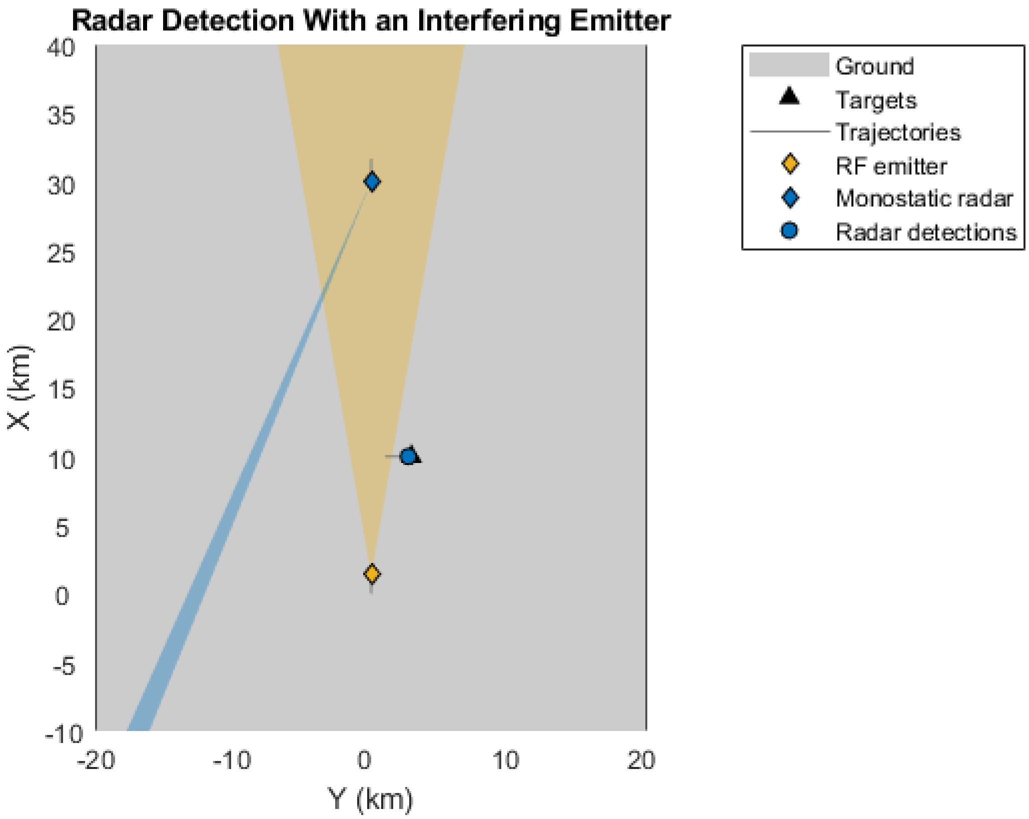
For each step in the following loop, use:

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.



```
% 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 of 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
```



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)
```

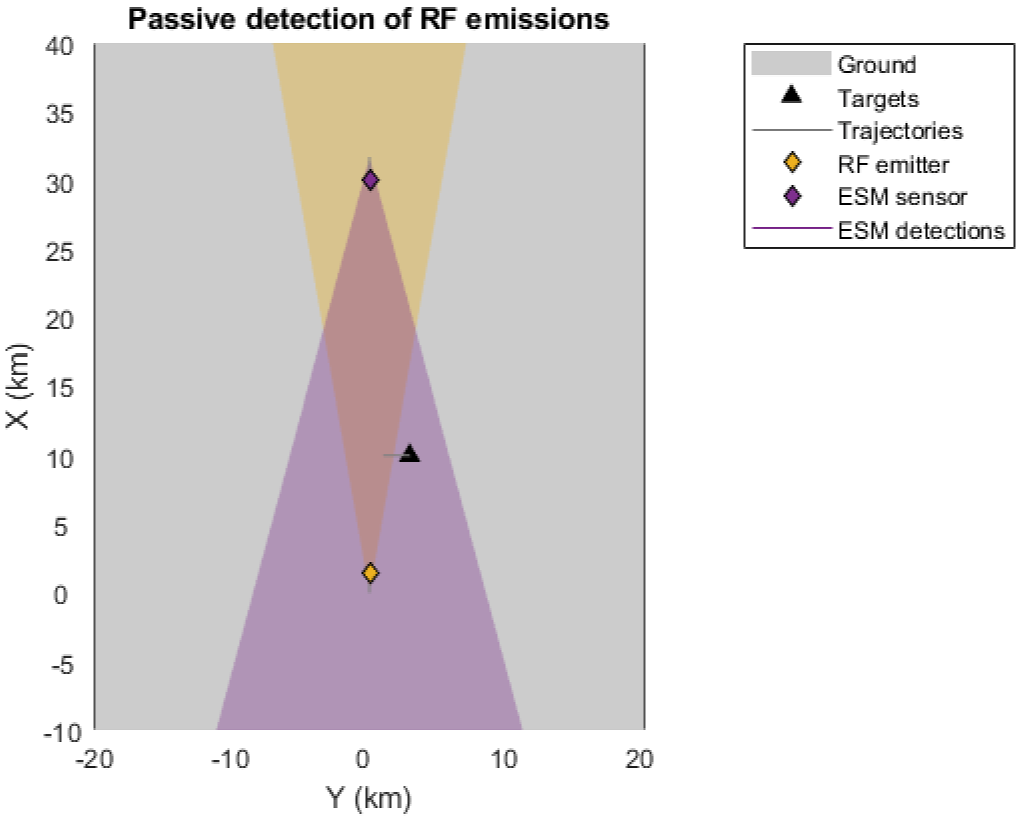
```
esm = 
  radarDataGenerator with properties:

           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');
```

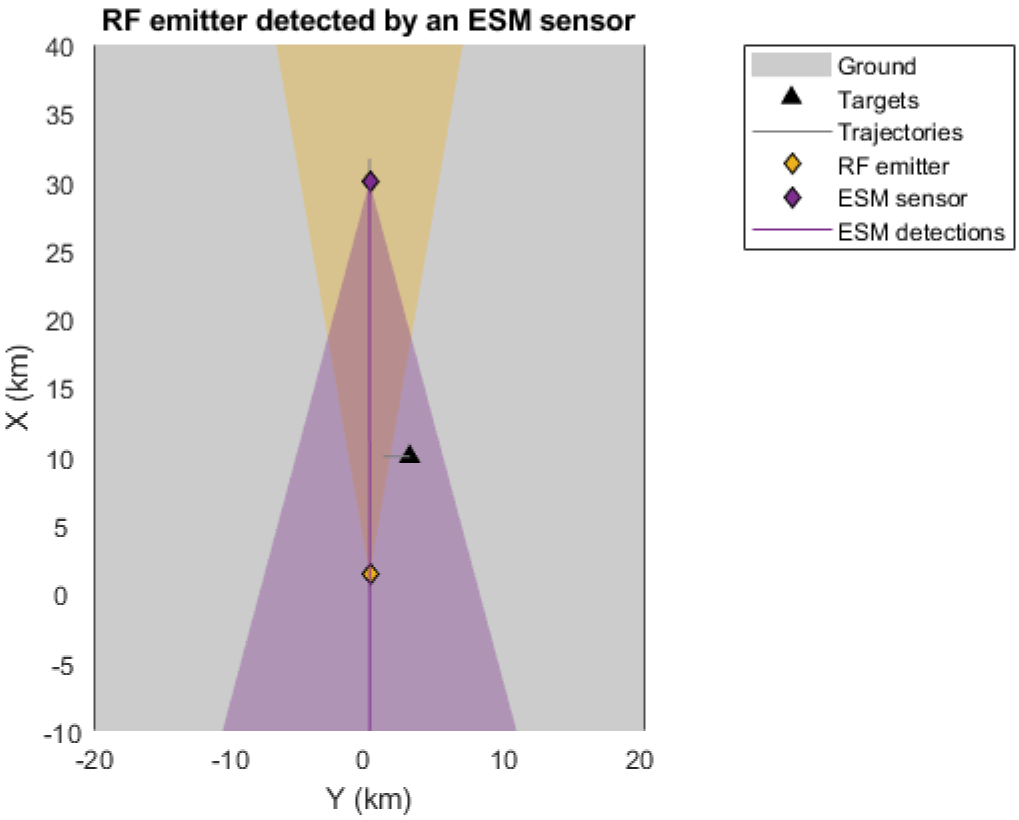


In the preceding figure, the radar is replaced by an ESM sensor mounted on the second platform. The ESM sensor's field of view is shown in magenta and includes both the emitting and target platforms.

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 of 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.

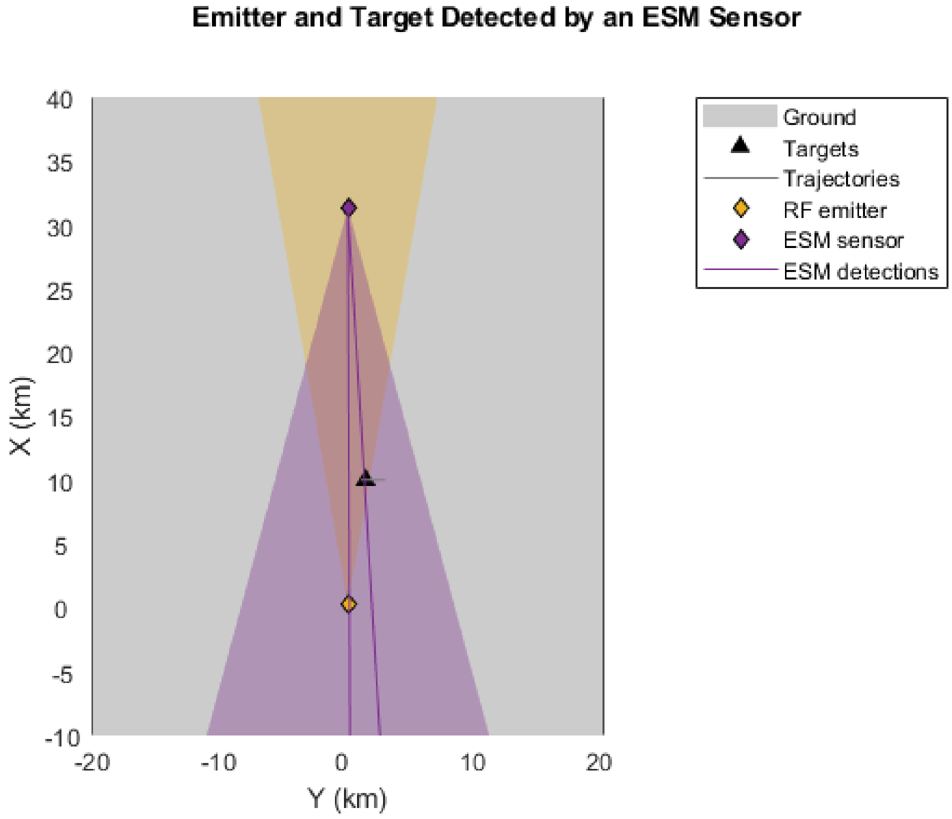
The ESM sensor classifies the waveform types in its reported detections. For this emitter, it reports the noise waveform type used by the emitter: 0.

```
dets{1}.ObjectAttributes{1}
```

```
ans = struct with fields:
     TargetIndex: 1
    EmitterIndex: 1
    WaveformType: 0
             SNR: 184.8224
```

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 of 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;
```

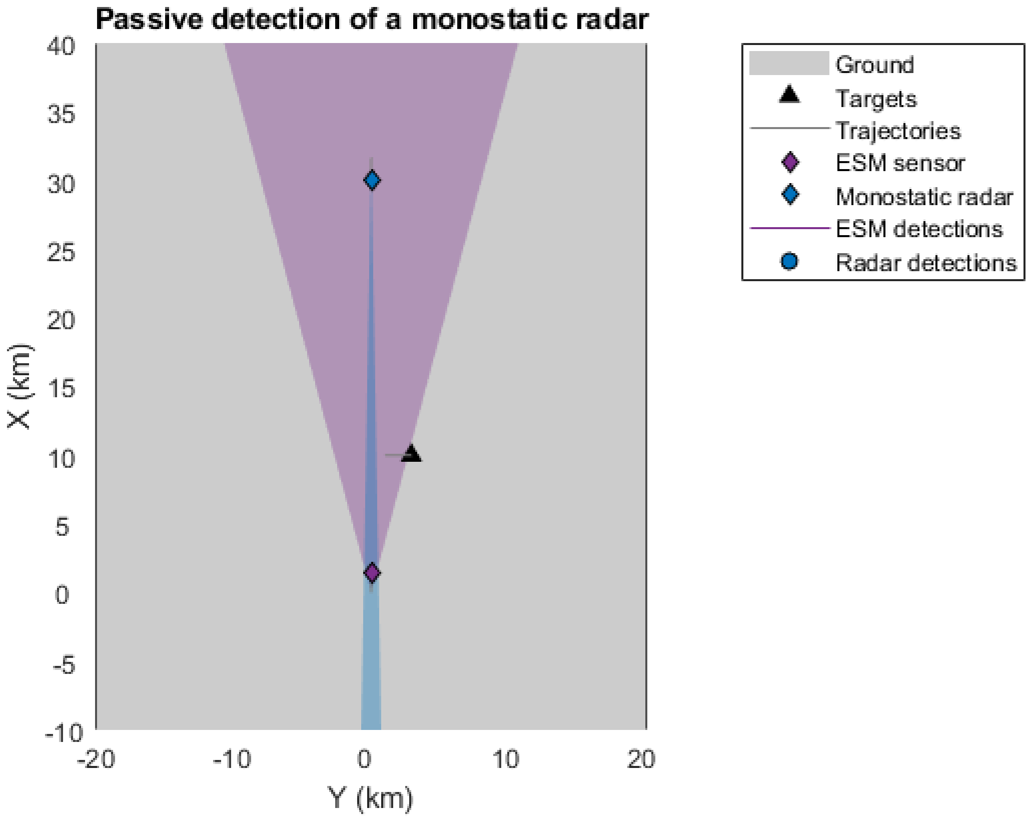
Reuse the ESM sensor from before, but set the list of known waveform types for the ESM sensor to include the waveform emitted by the radar. If the radar's waveform type is not known to the ESM sensor, it will not be detected.

```
% 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.

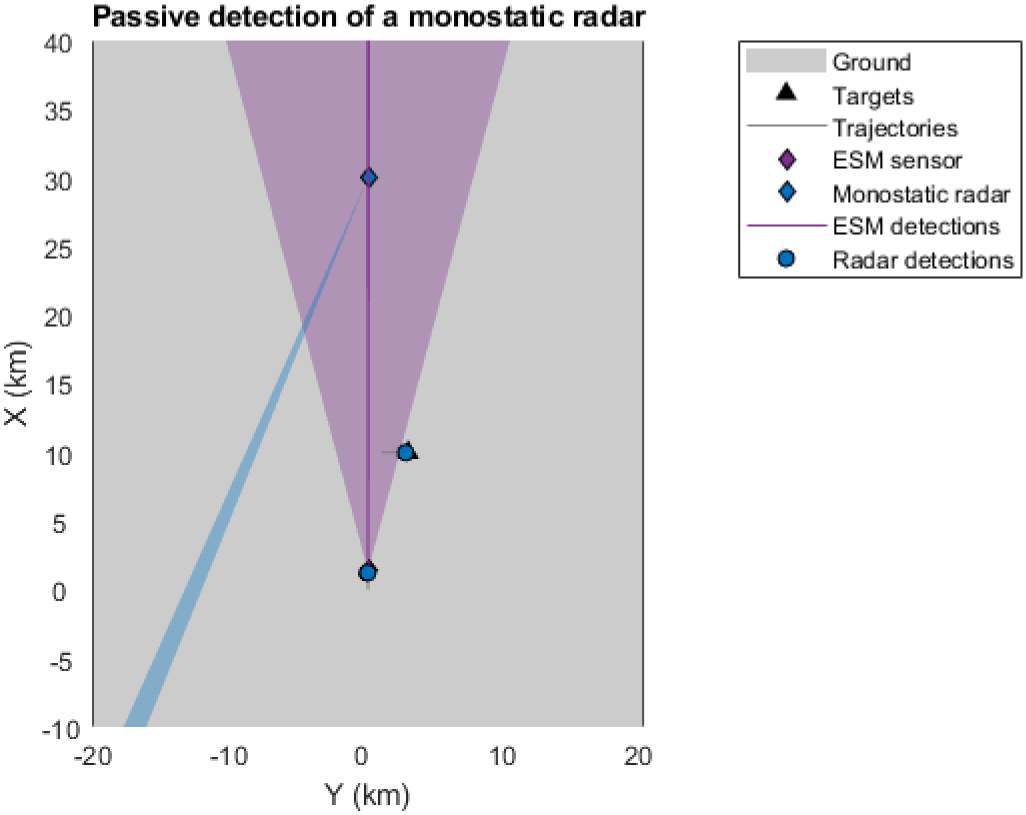
```
detPlotter = detectionPlotter(theaterDisplay,'DisplayName','Radar detections','MarkerFaceColor',
platRadarPlotter.DisplayName = 'Monostatic radar';
platRadarPlotter.MarkerFaceColor = radarColor;
platEmitPlotter.DisplayName = 'ESM sensor';
platEmitPlotter.MarkerFaceColor = esmColor;
```

```
clearData(esmDetPlotter);
clearData(covPlotter);
covcon = coverageConfig(scene);
plotCoverage(covPlotter, covcon([1 3]) , [1 -2], {esmColor, radarColor});
title(ax,'Passive detection of a monostatic radar');
```



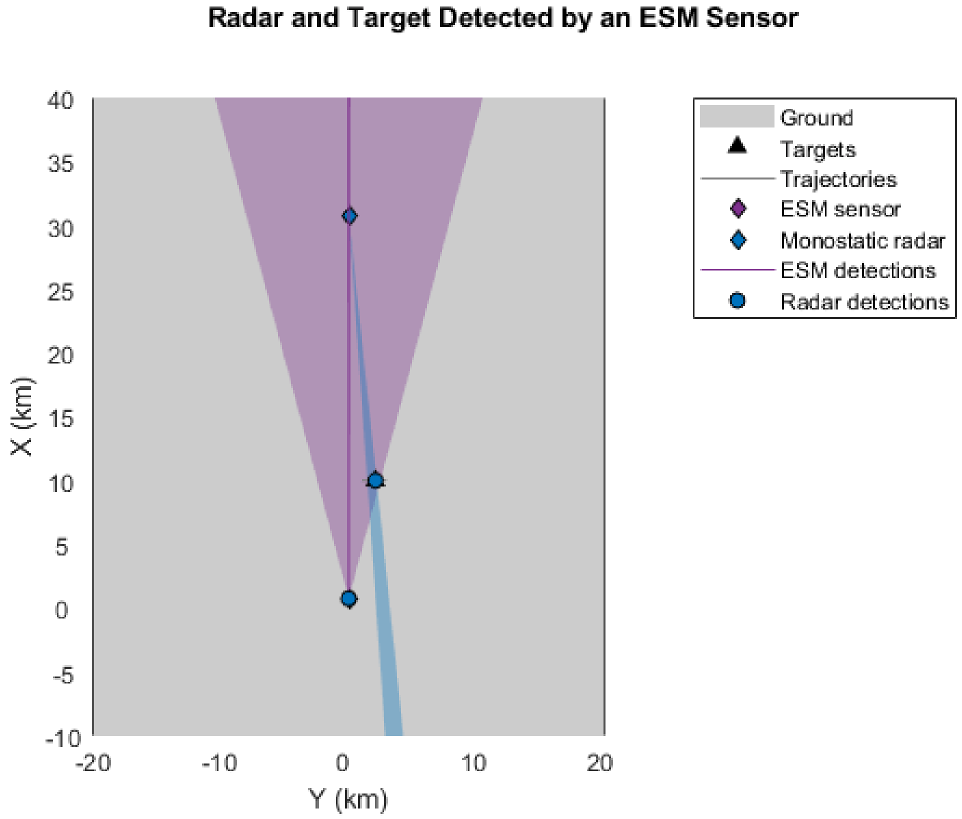
The preceding figure shows the radar scanning an azimuth sector in front of its platform, which includes both the target platform as well as the platform carrying the ESM sensor. The radar generates detections for both of these platforms when its field of view (shown in blue) sweeps over their location. 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);
    % Reflect the emitted signal off of 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');
```



Because of the high power (EIRP) of the radar's emissions, the emitted energy is reflected off of 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

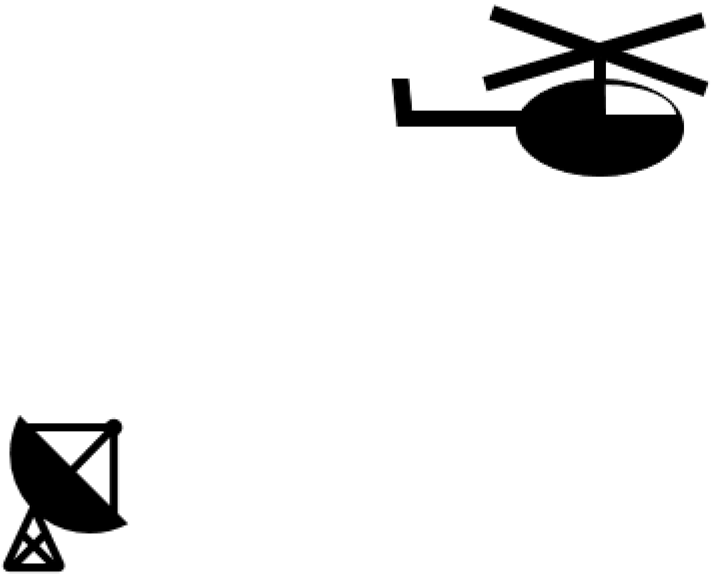
This example introduces the basic concept of a micro-Doppler effect in the radar return of a target due to the rotation of that target. You can use the micro-Doppler signature to help identify the target.

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.

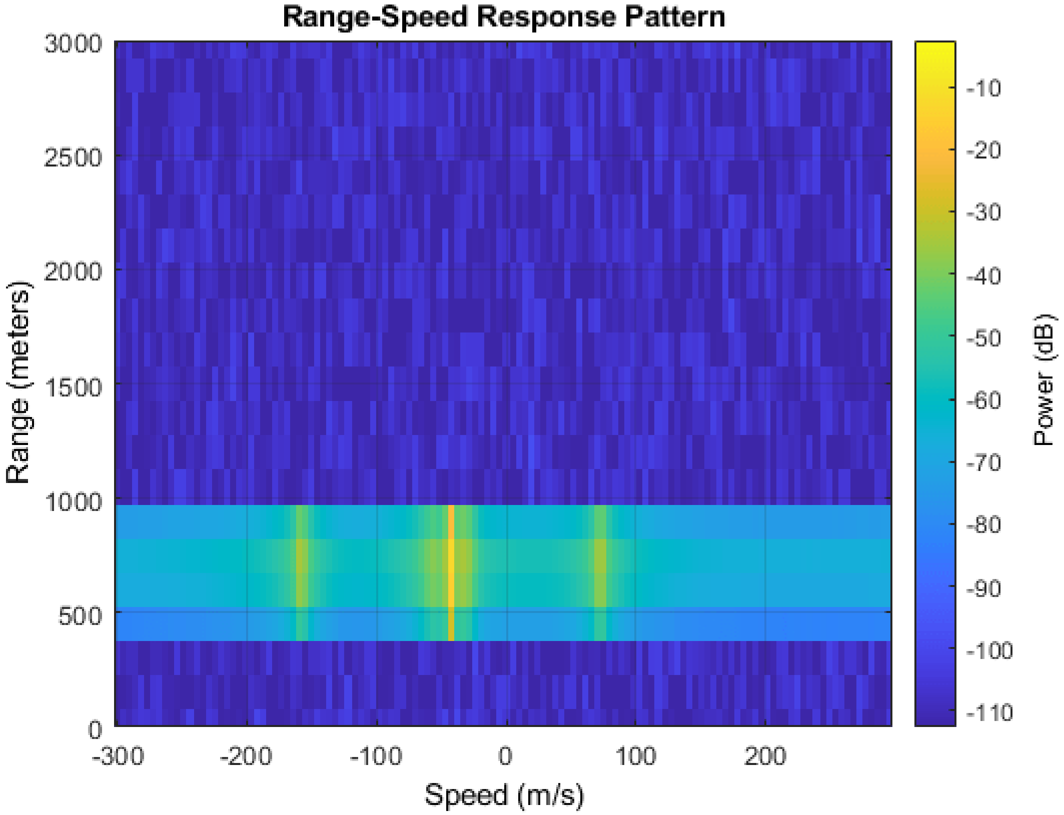
```
fs     = 1e6;
prf    = 2e4;
lambda = c/fc;

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)
```

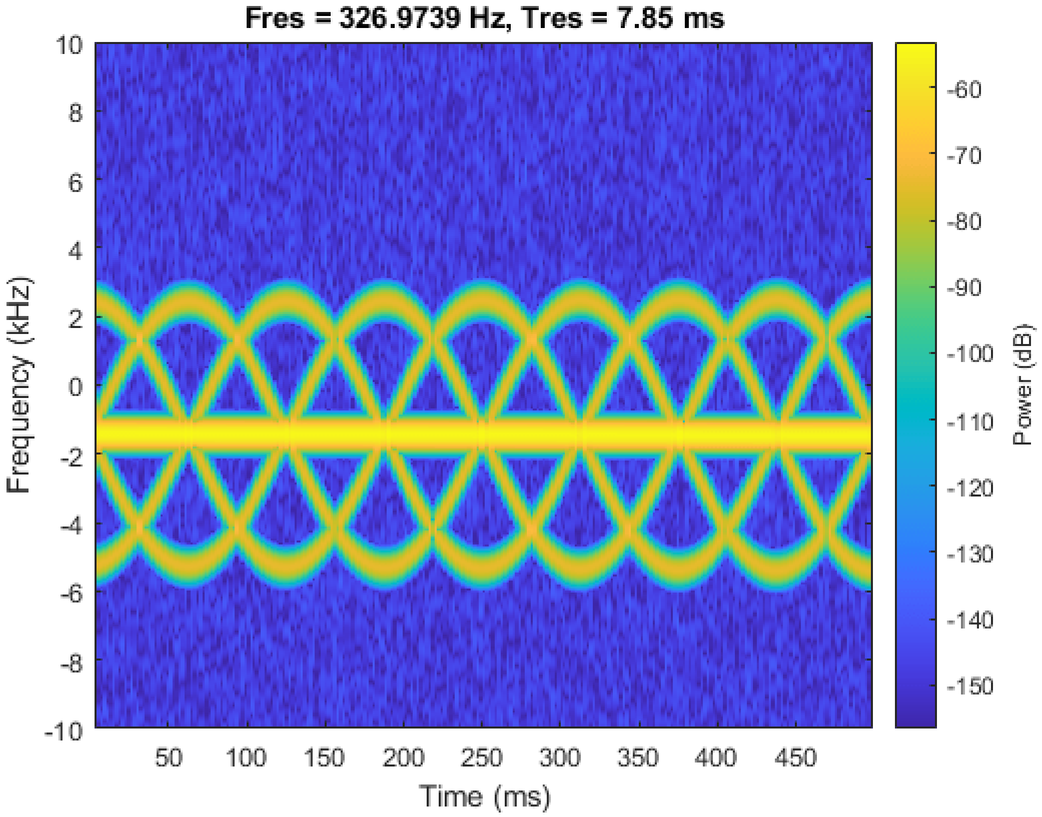
```
vtp =

   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

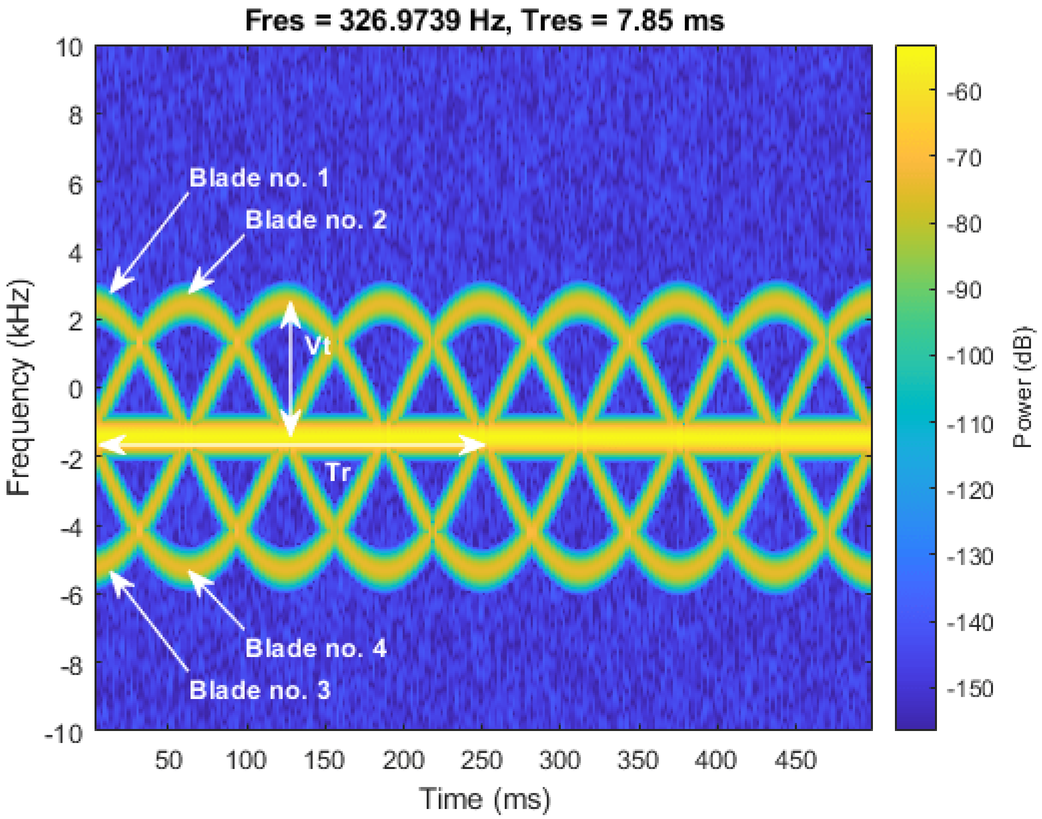
The time-frequency representation of micro-Doppler effects can reveal more information. This code constructs a time-frequency representation in the detected target range bin.

```
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
```

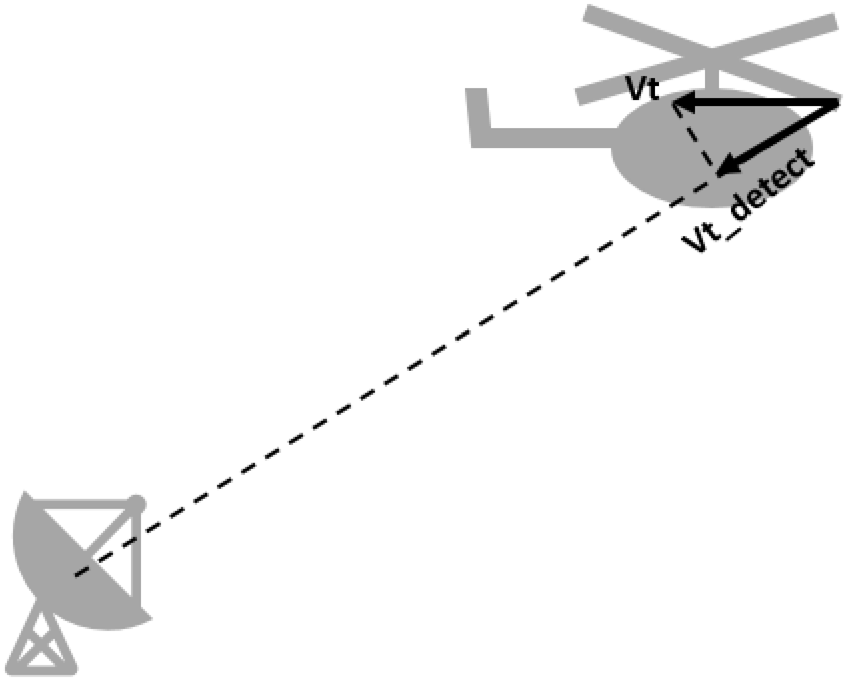
```
bladerate_est =

     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
```

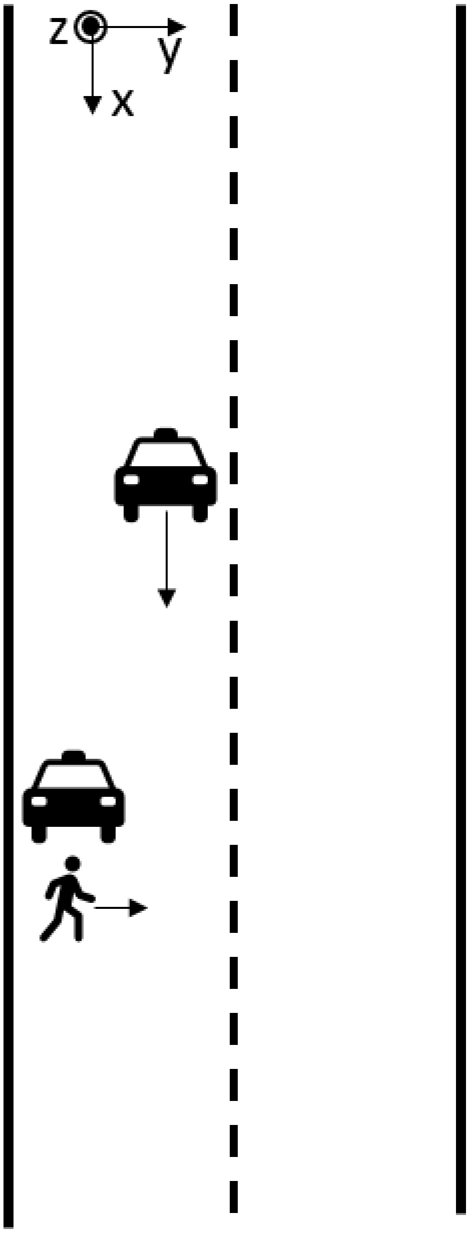
Note that the result matches the simulation parameter of 6.5 meters. Information such as number of blades, blade length, and blade rotation rate can help identify the model of the helicopter.

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



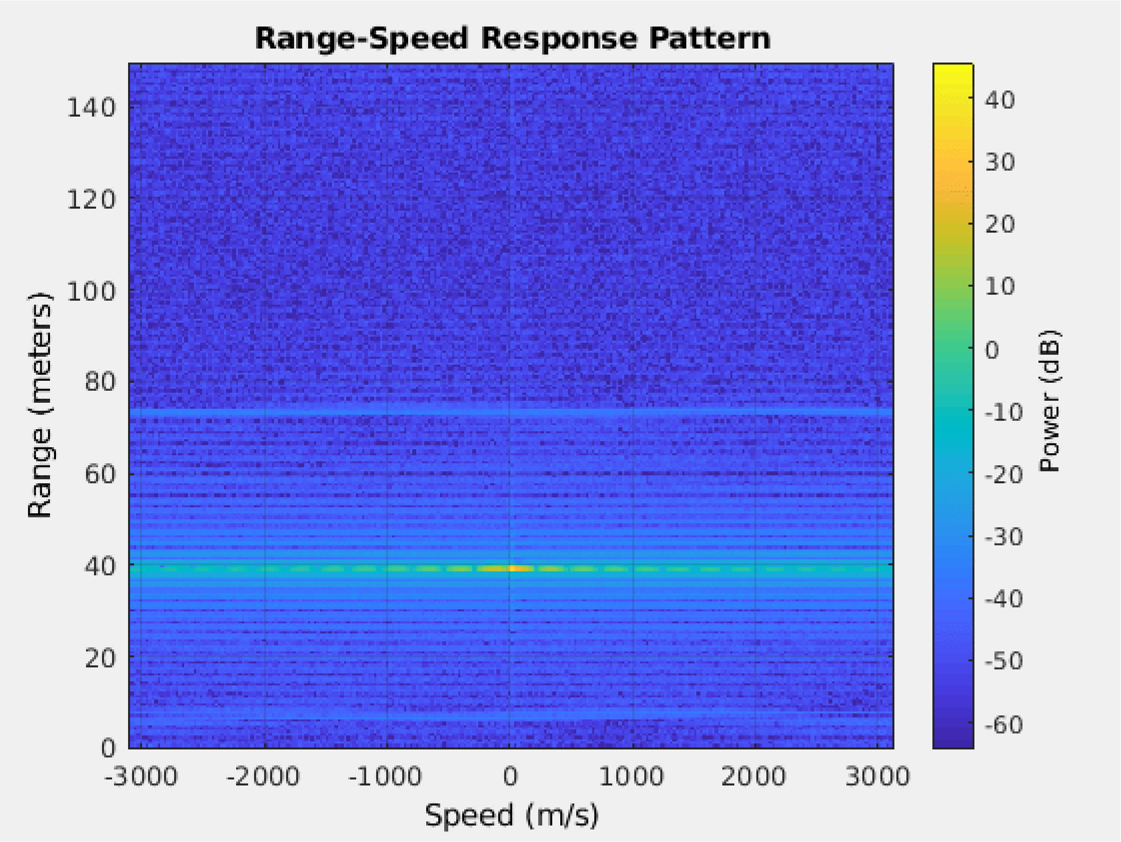
Based on this setup, if the ego car cannot identify that a pedestrian is present, an accident may occur.

```
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.



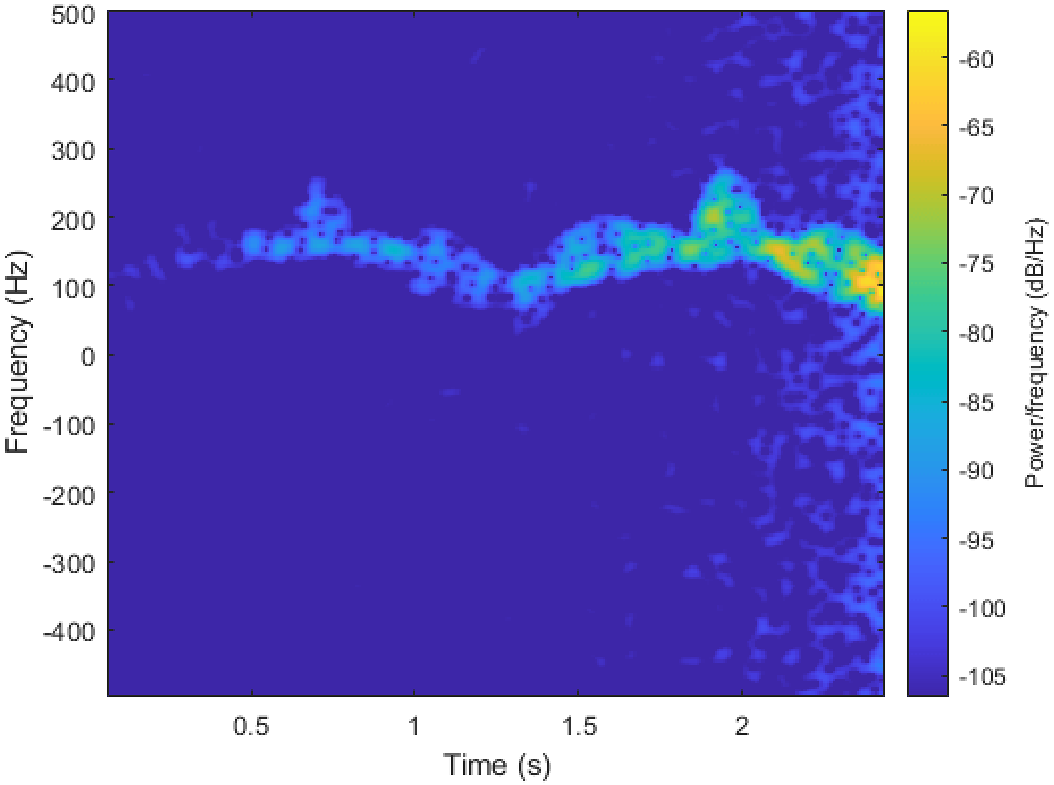
This means that conventional processing cannot satisfy our needs under this situation.

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.

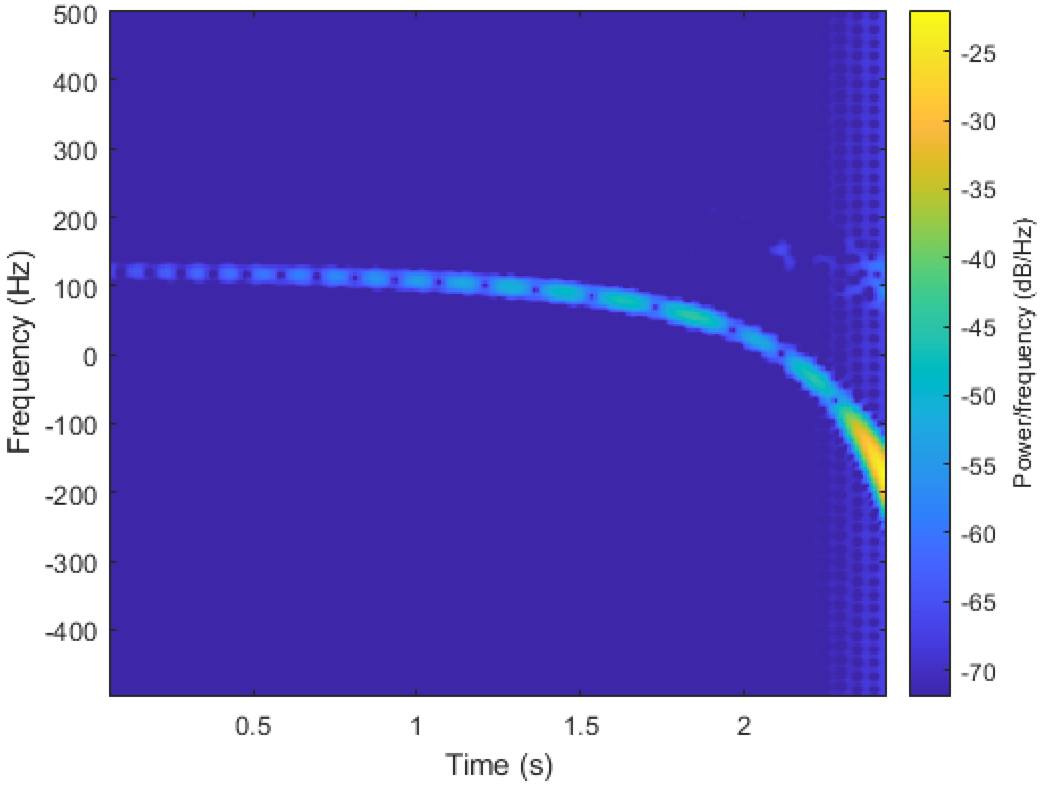
```
clf;
spectrogram(sum(xd_ped),kaiser(128,10),120,256,1/Tsamp,'centered','yaxis');
clim = get(gca,'CLim');
set(gca,'CLim',clim(2)+[-50 0])
```



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.

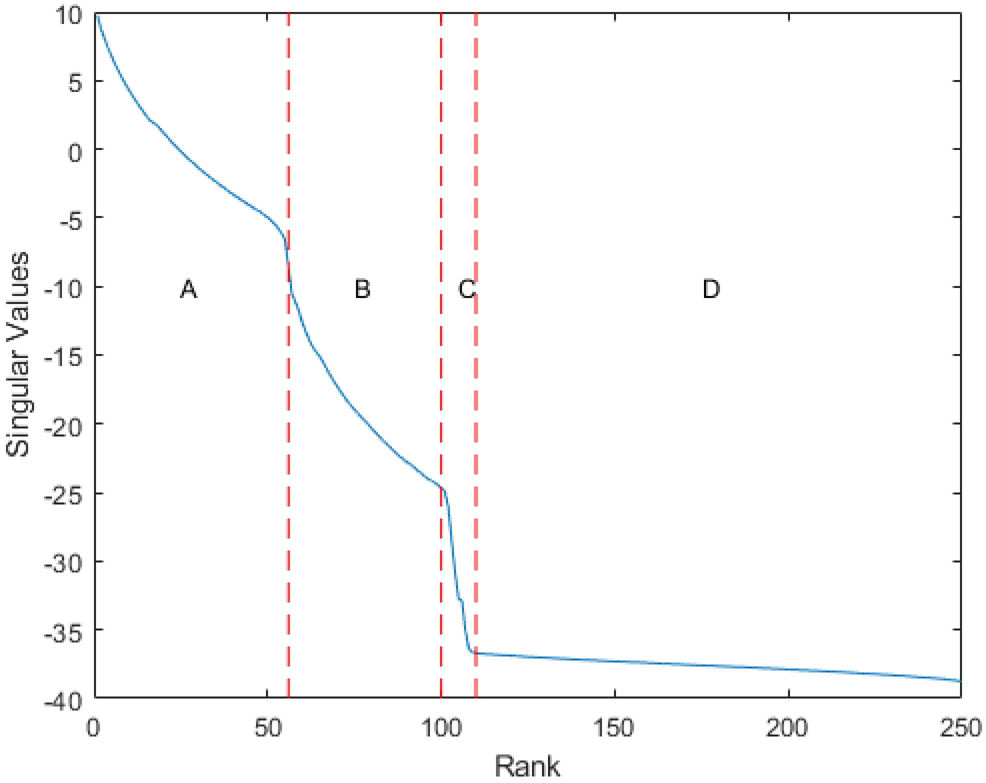
```
spectrogram(sum(xd),kaiser(128,10),120,256,1/Tsamp,'centered','yaxis');
clim = get(gca,'CLim');
set(gca,'CLim',clim(2)+[-50 0])
```



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.

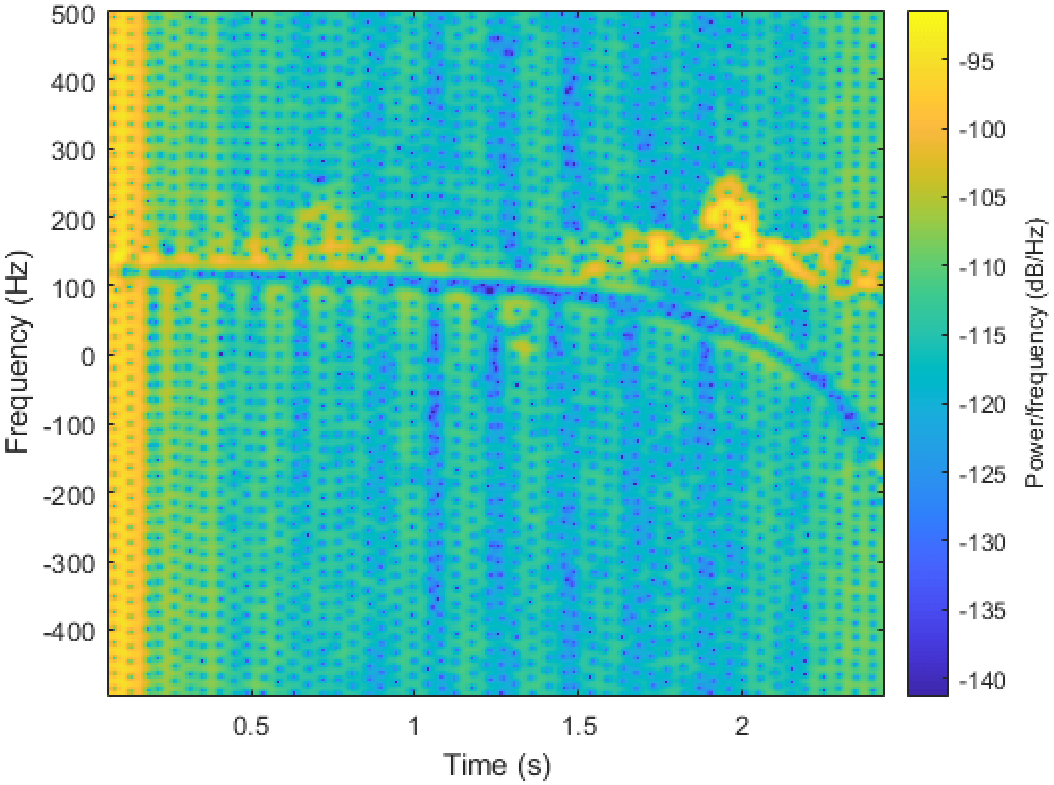
To see if there is a return hidden behind the strong return, we can use the singular value decomposition. The following plot shows the distribution of the singular value of the dechirped pulses.

```
[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 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 we'll use the radar system built in the example “Simulating Test Signals for a Radar Receiver”. Readers are encouraged to explore the details of the radar system design through that example. Change the antenna's 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 transmitter's power source 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 received samples received within the corresponding pulse interval. We can simulate a coherent on receive system by setting the transmitter and receiver as following:

```
transmitter.CoherentOnTransmit = false;
transmitter.PhaseNoiseOutputPort = true;
receiver.PhaseNoiseInputPort = true;
```

Define Targets

Next, define two moving targets.

The first target is located at 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 target's Doppler signature is aliased to the pulse repetition frequency. This setting prevents the MTI radar from detecting the target. We use the dop2speed() function to calculate a blind speed which has a corresponding Doppler frequency equal to the pulse repetition frequency.

```
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

Now we 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);
```

We 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));
```

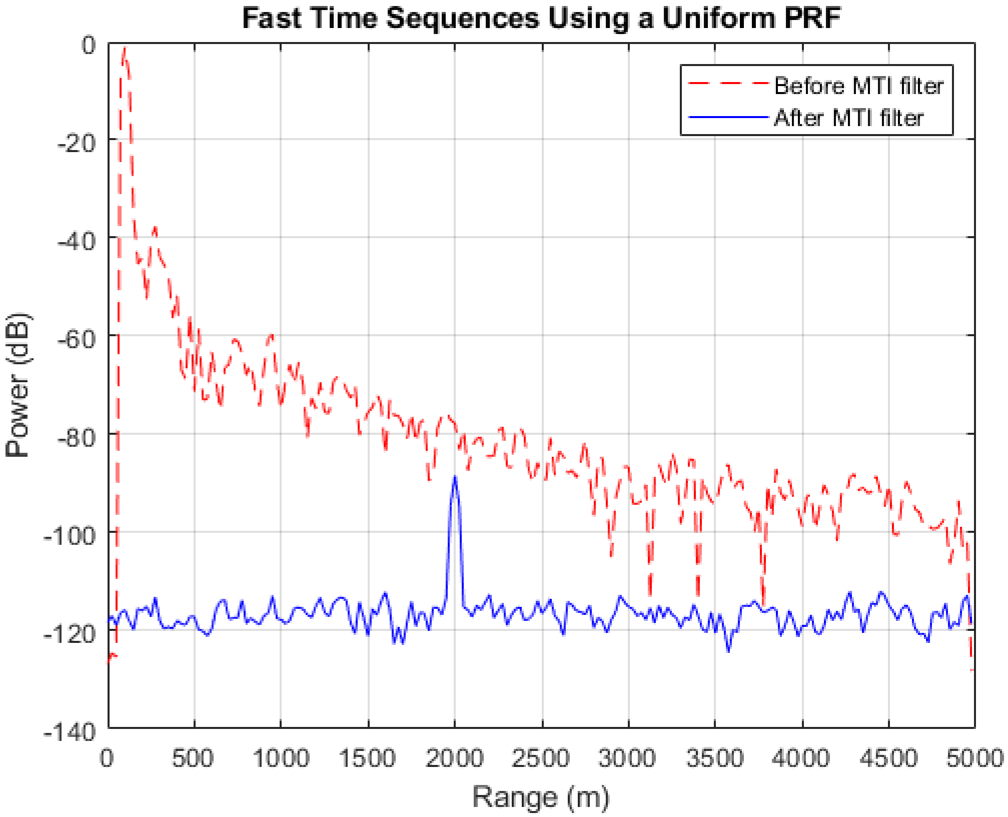
Perform MTI Processing Using a Three-Pulse Canceller

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 canceller is a popular and simple MTI filter. The canceller 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');
```

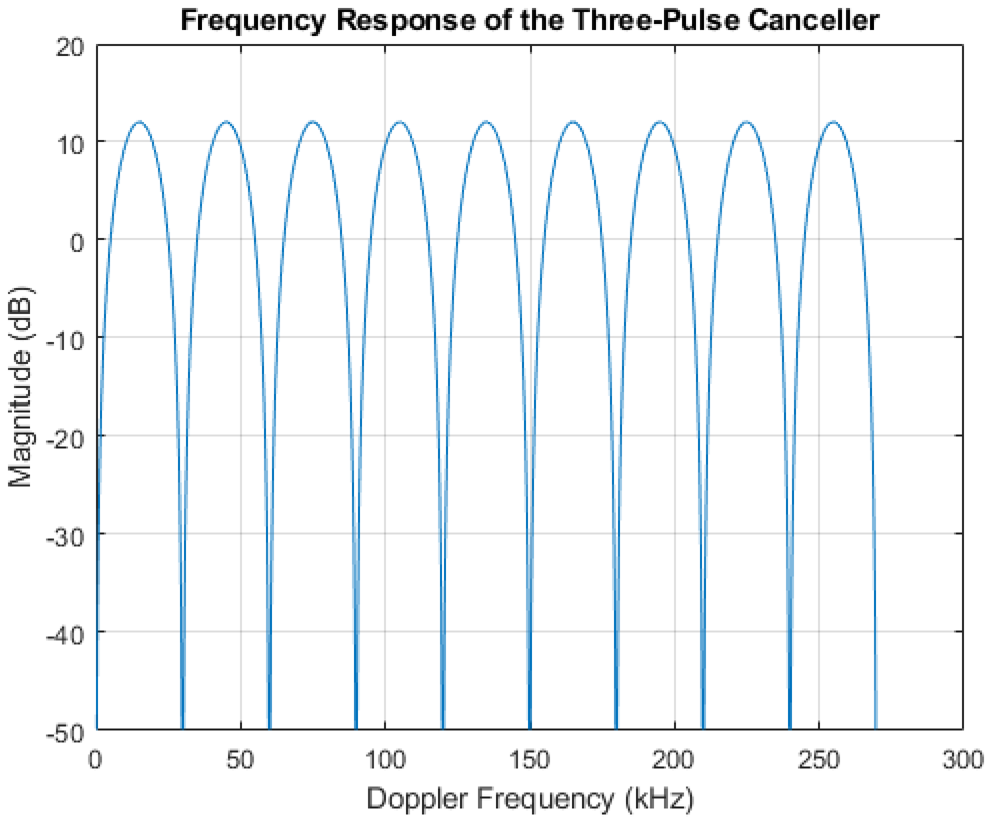


Recall that there are two targets (at 2 km and 3 km). In the case before MTI filtering, both targets are buried in clutter returns. The peak at 100 m is the direct path return from the ground right below the radar. Notice that the power is decreasing as the range increases, which is due to the signal 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 canceller. At the 2 km range, we 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 canceller suppresses the second target which travels at the canceller's blind speed.

To better understand the blind speed problem, let us look at the frequency response of the three-pulse canceller.

```
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 Canceller');
```



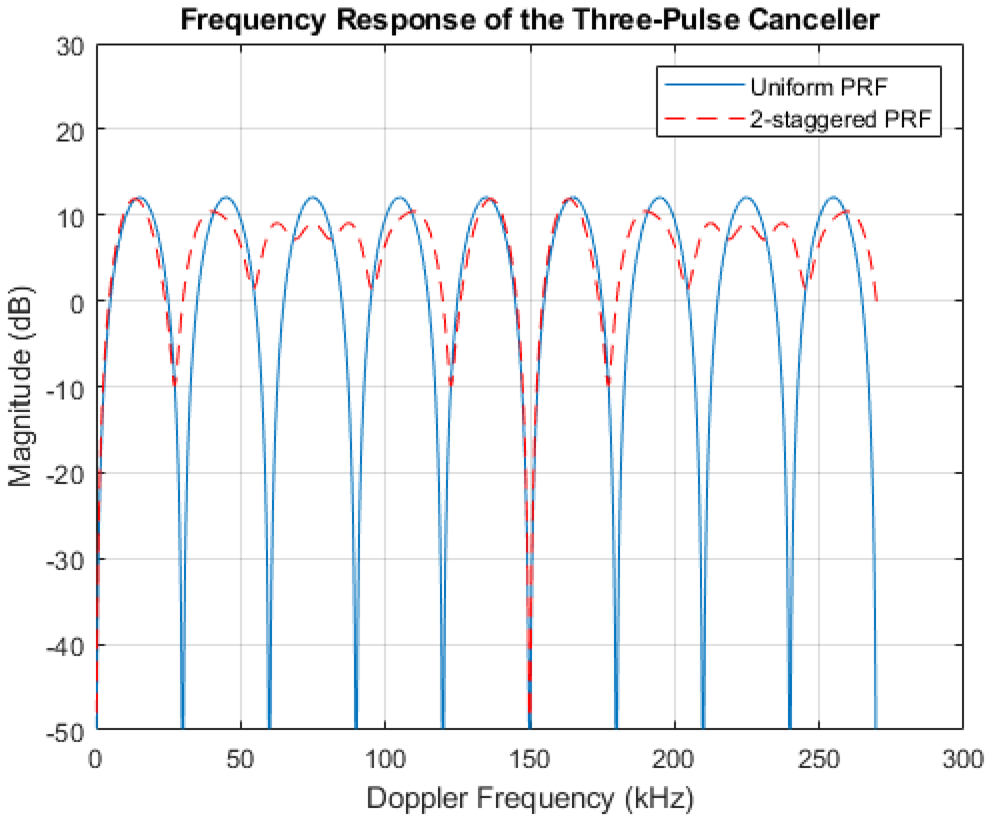
Notice the recurring nulls in the frequency response. The nulls correspond to the Doppler frequencies of the blind speeds. Targets with these Doppler frequencies are cancelled by the three-pulse canceller. The plot shows that the nulls occur at integer multiples of the PRF (approximately 30kHz, 60kHz,...). If we can remove these nulls or push them away from the Doppler frequency region of the radar specifications, we can avoid the blind speed problem.

Simulate the Received Pulses Using Staggered PRFs

One solution to the blind speed problem is to use a nonuniform PRF, or staggered PRFs. Adjacent pulses are transmitted at different pulse repetition frequencies. Such configuration pushes the lower bound of blind speeds to a much higher value. To illustrate this idea, we will use a two-staggered PRF, and plot the frequency response of the three-pulse canceller.

Let us 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 canceller
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','2-staggered PRF');
```



From the plot of the staggered PRFs we can see 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.

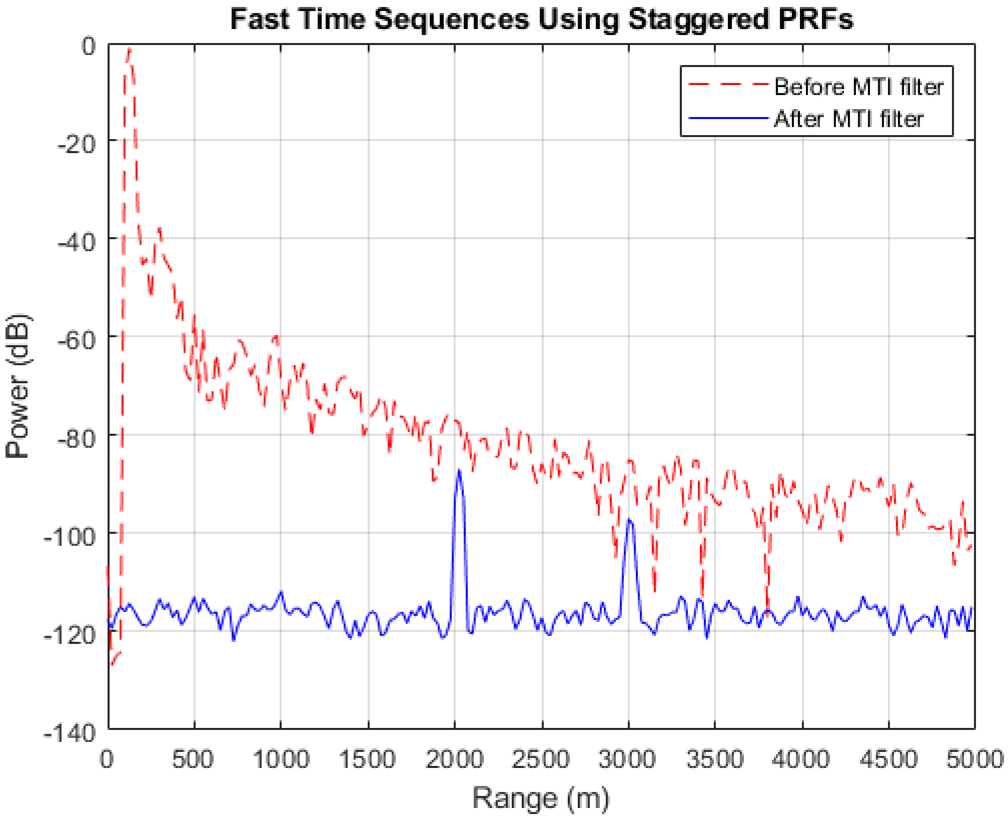
Now, simulate the reflected signals from the targets using the staggered PRFs.

```
% 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

We 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 canceller 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 clutters having a wide spectrum, the MTI processing could be poor. That type of clutter can be suppressed using space-time adaptive processing. See the example “Introduction to Space-Time Adaptive Processing” for details.

Appendix

Reference: Mark A. Richards, *Fundamentals of Radar Signal Processing*, 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.

```
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.

```
load NEXRAD_Parameters.mat
```

Antenna Pattern

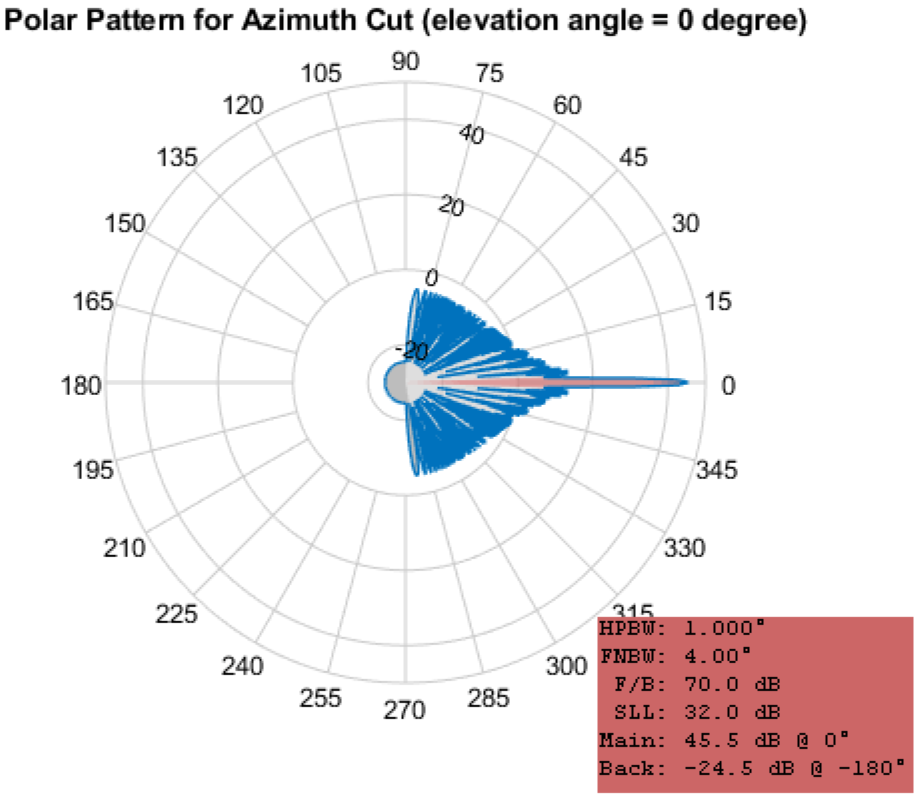
As NEXRAD is polarimetric, modeling the polarimetric characteristics of the antenna and weather targets is important. According to NEXRAD specifications, the antenna pattern has a beamwidth of about 1 degree and first sidelobe below -30 dB.

```
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)

```
HPBW:  1.000°
FNBW:  4.00°
 F/B:  70.0 dB
 SLL:  32.0 dB
Main:  45.5 dB @ 0°
Back:  -24.5 dB @ -180°
```

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 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).

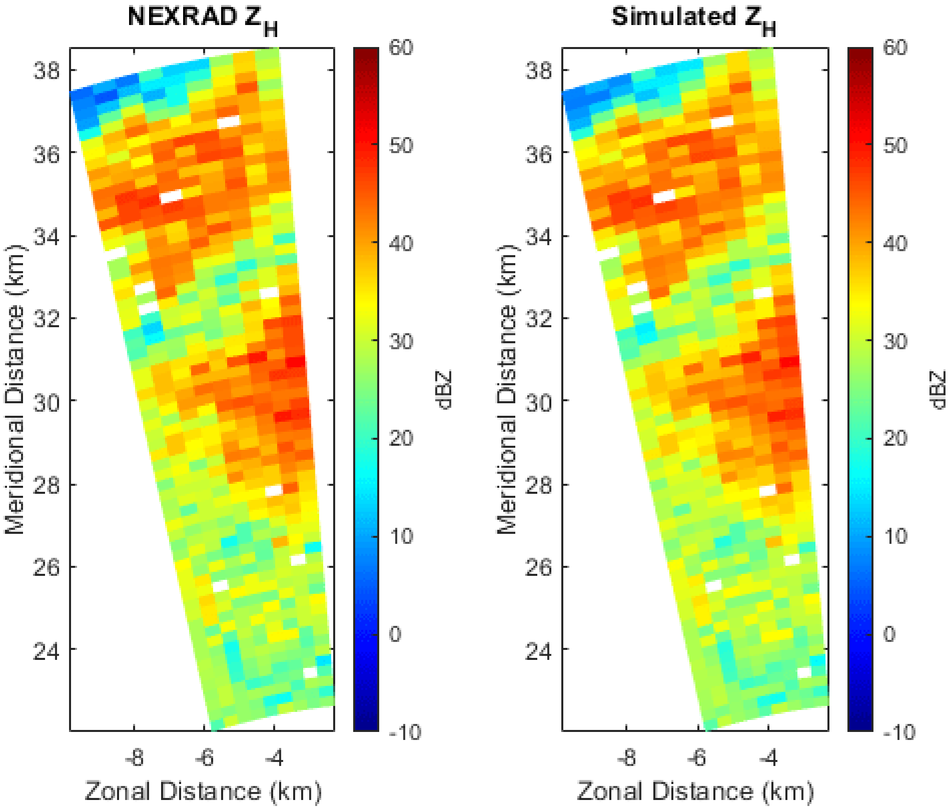
Define the azimuth and range for plotting.

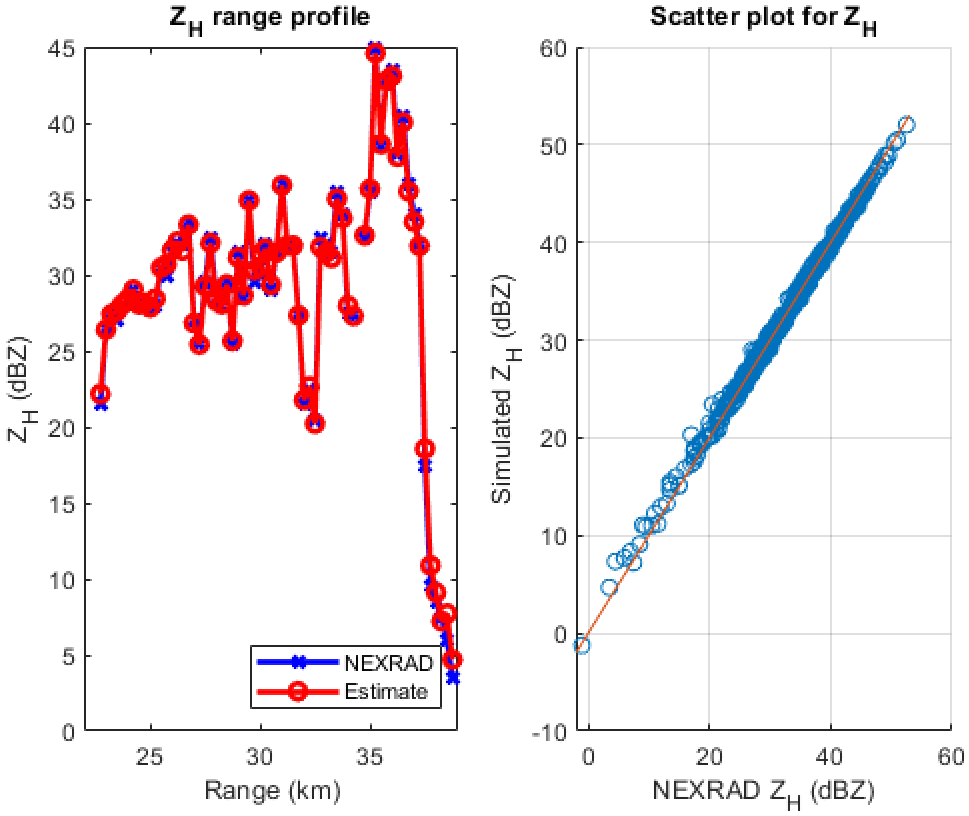
```
azimuth = nexrad.azimuth(nexrad_aoi.r1:nexrad_aoi.r2);
range = (nexrad_aoi.b1-1:nexrad_aoi.b2-1)*250 + 2000;
```

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');
```

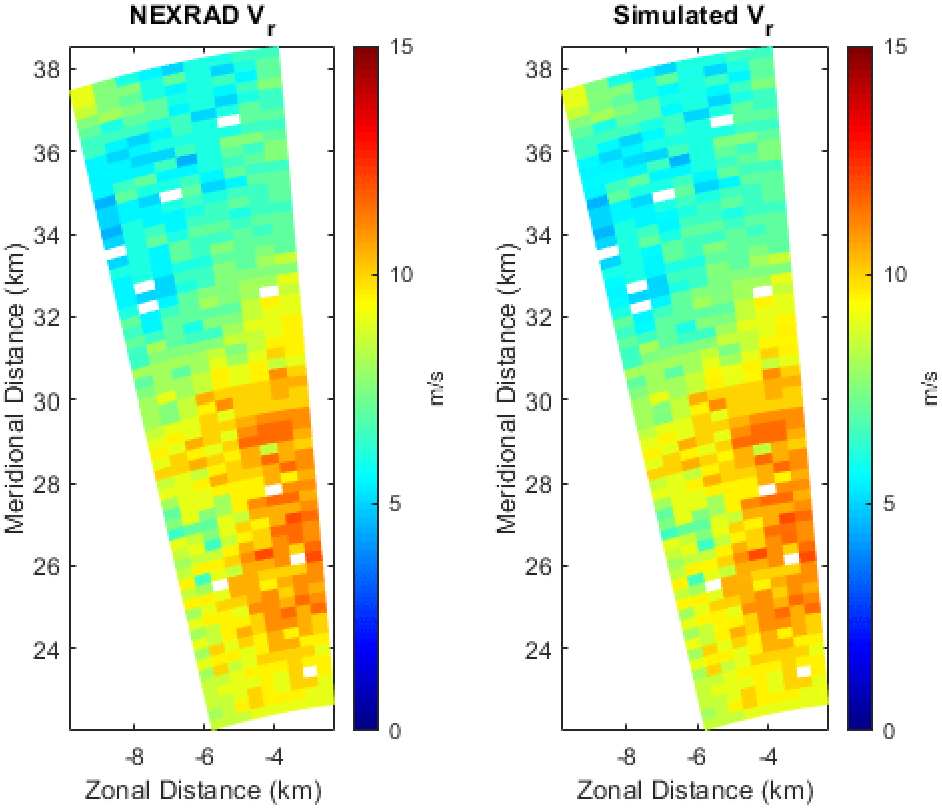

NEXRAD Z_H
Simulated Z_H
Meridional Distance (km)
Zonal Distance (km)
38
36
34
32
30
28
26
24
-8
-6
-4
dBZ
60
50
40
30
20
10
0
-10

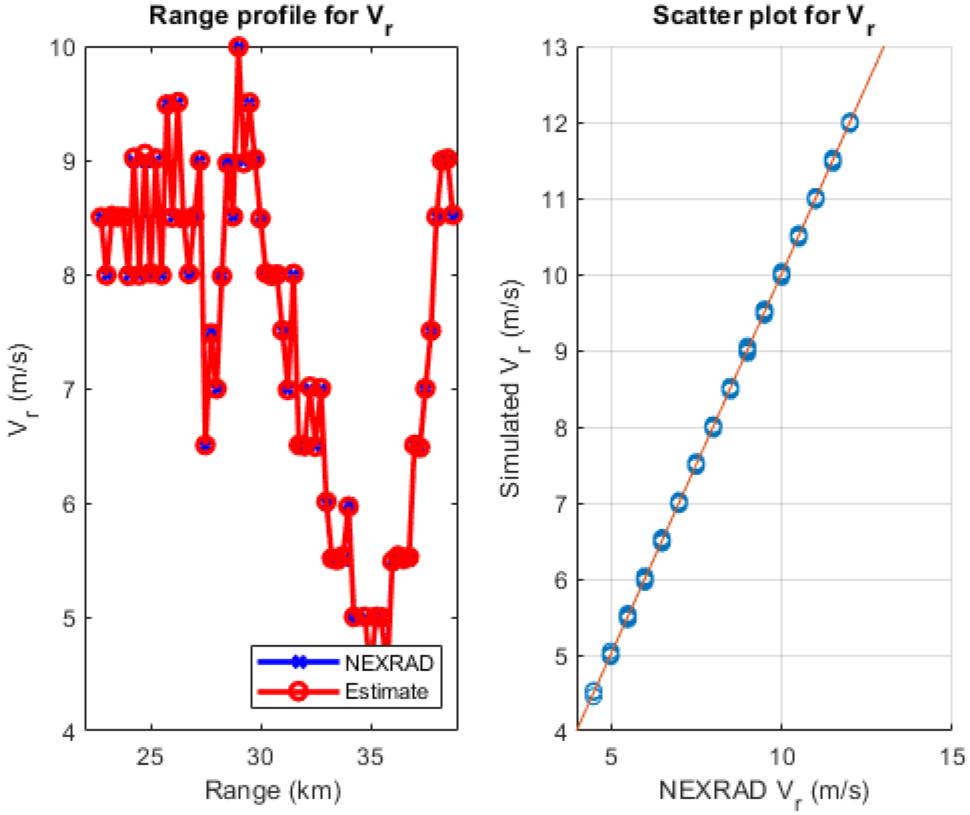


Radial Velocity

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');
```

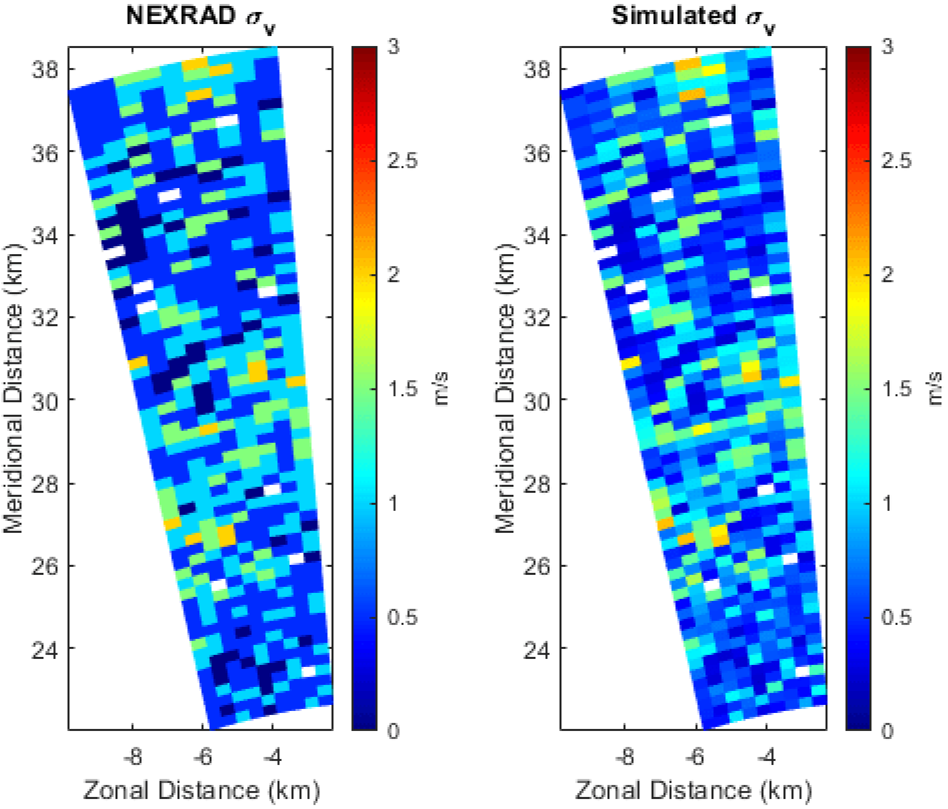

NEXRAD V_r
Simulated V_r
Meridional Distance (km)
Zonal Distance (km)
38
36
34
32
30
28
26
24
-8
-6
-4
m/s
15
10
5
0

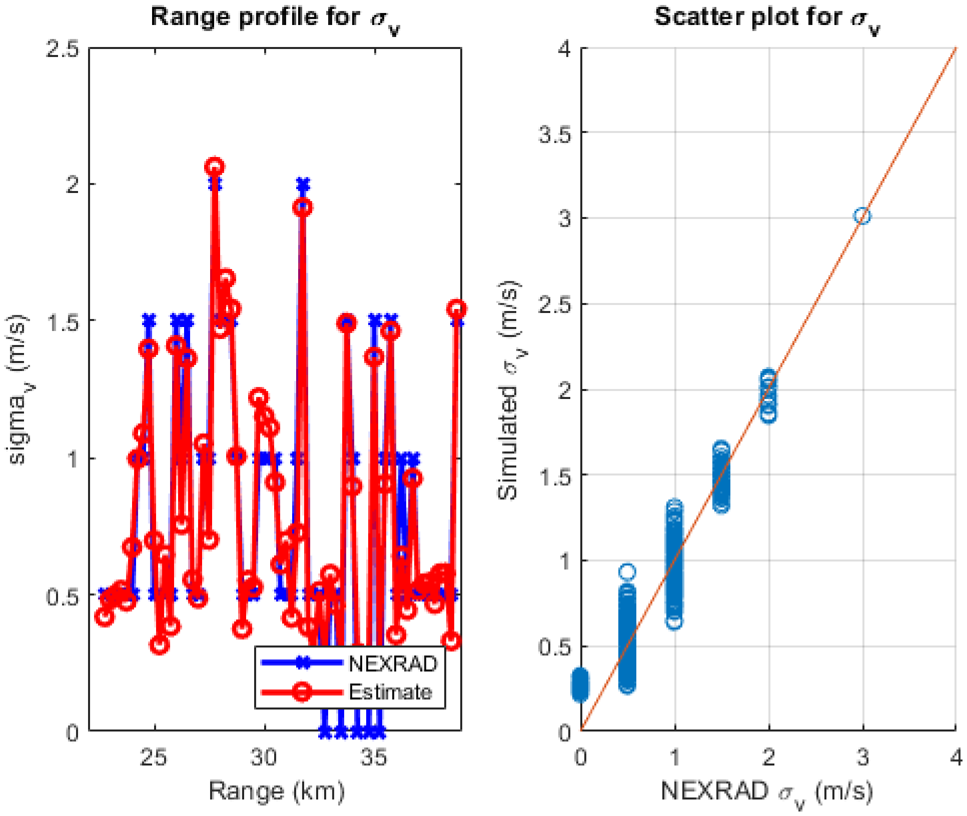


Spectrum Width

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');
```

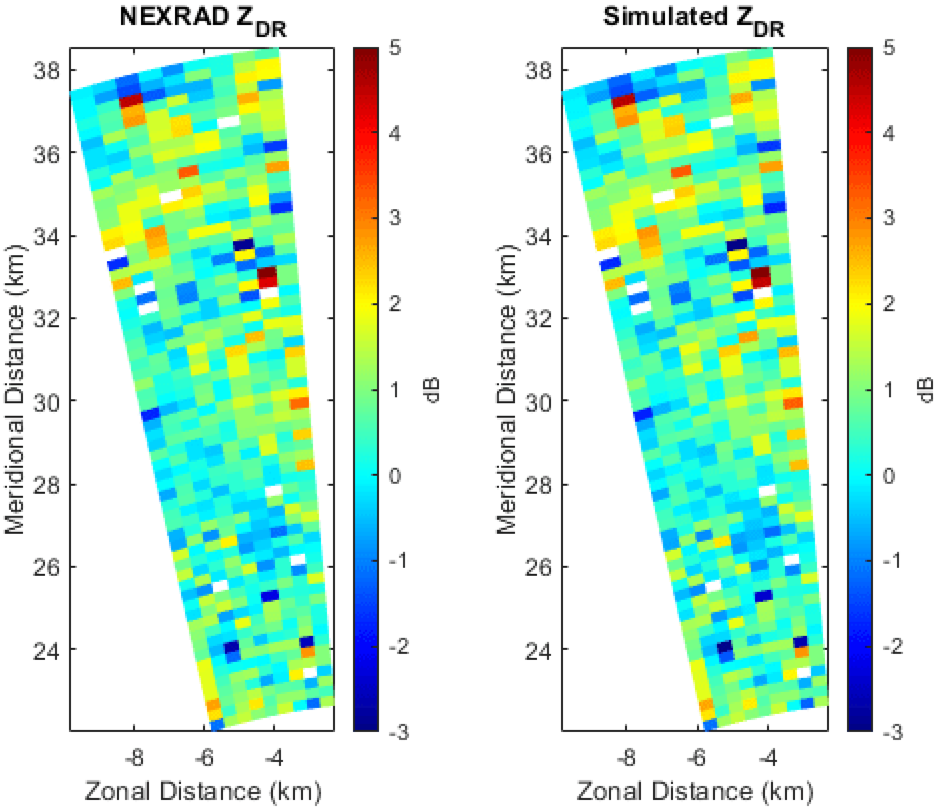

NEXRAD σ_v
Meridional Distance (km)
38
36
34
32
30
28
26
24
Zonal Distance (km)
-8
-6
-4
m/s
3
2.5
2
1.5
1
0.5
0
Simulated σ_v
Meridional Distance (km)
38
36
34
32
30
28
26
24
Zonal Distance (km)
-8
-6
-4
m/s
3
2.5
2
1.5
1
0.5
0

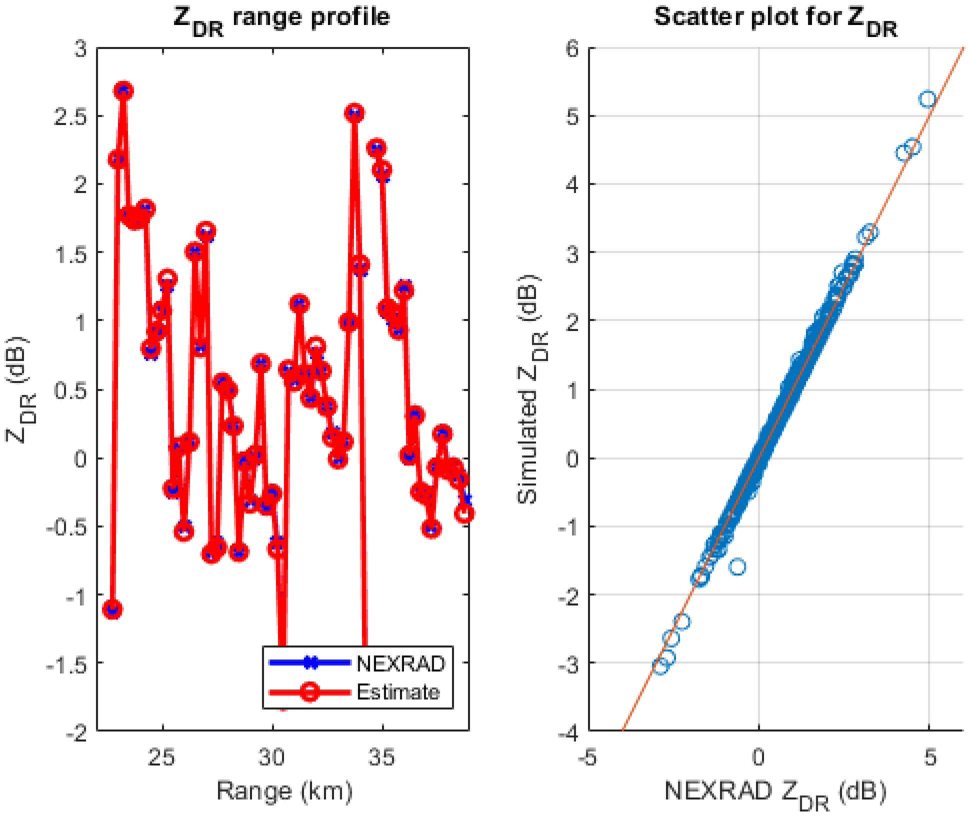


Differential Reflectivity

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');
```

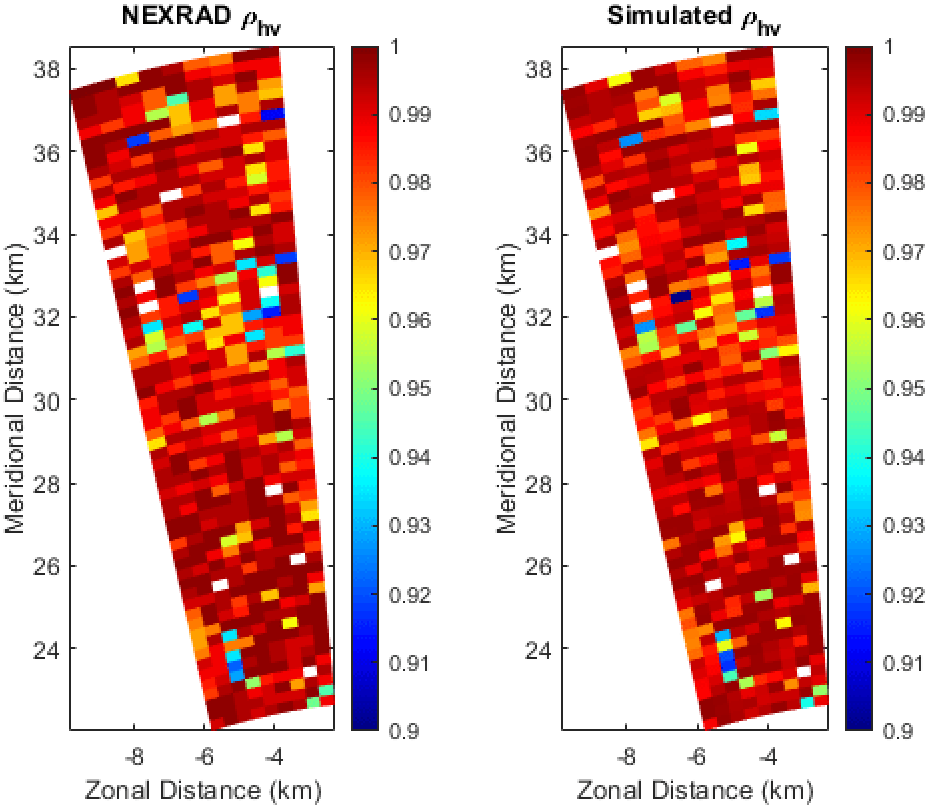

NEXRAD Z_{DR}
Simulated Z_{DR}
Meridional Distance (km)
Zonal Distance (km)
38
36
34
32
30
28
26
24
-8
-6
-4
dB
5
4
3
2
1
0
-1
-2
-3

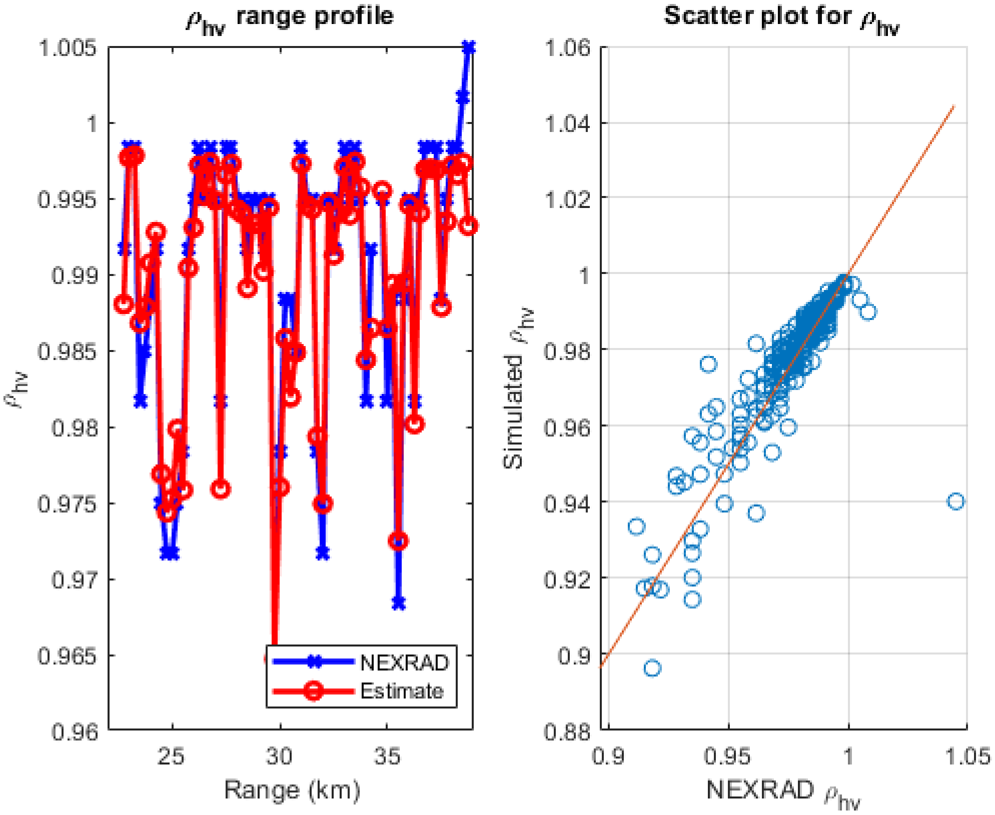


Correlation Coefficient

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');
```

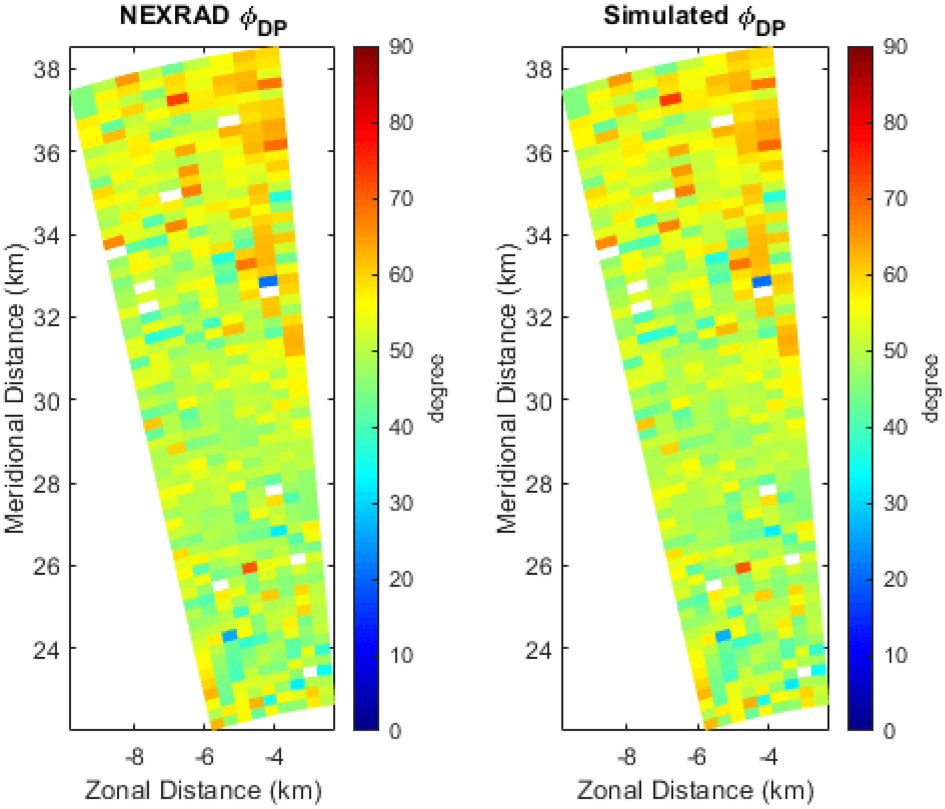

NEXRAD ρ_{hv}
Simulated ρ_{hv}
Meridional Distance (km)
Zonal Distance (km)
38
36
34
32
30
28
26
24
-8
-6
-4
1
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.91
0.9

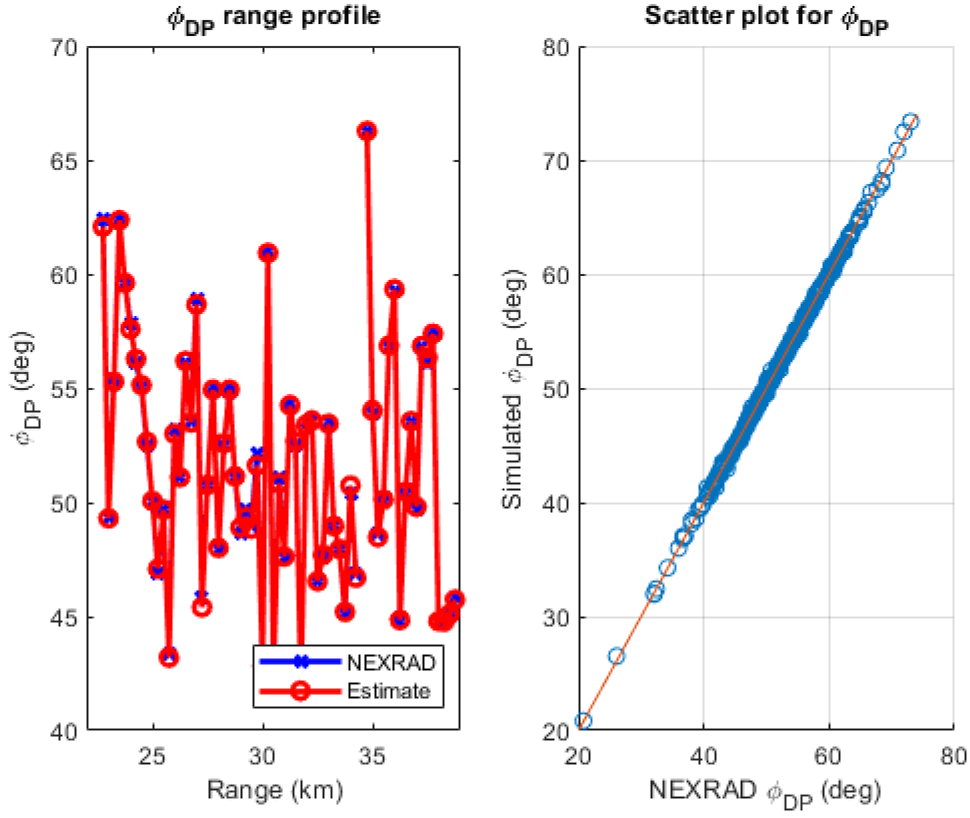


Differential Phase

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');
```


NEXRAD ϕ_{DP}
Simulated ϕ_{DP}
Meridional Distance (km)
38
36
34
32
30
28
26
24
Zonal Distance (km)
-8
-6
-4
degree
90
80
70
60
50
40
30
20
10
0



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.

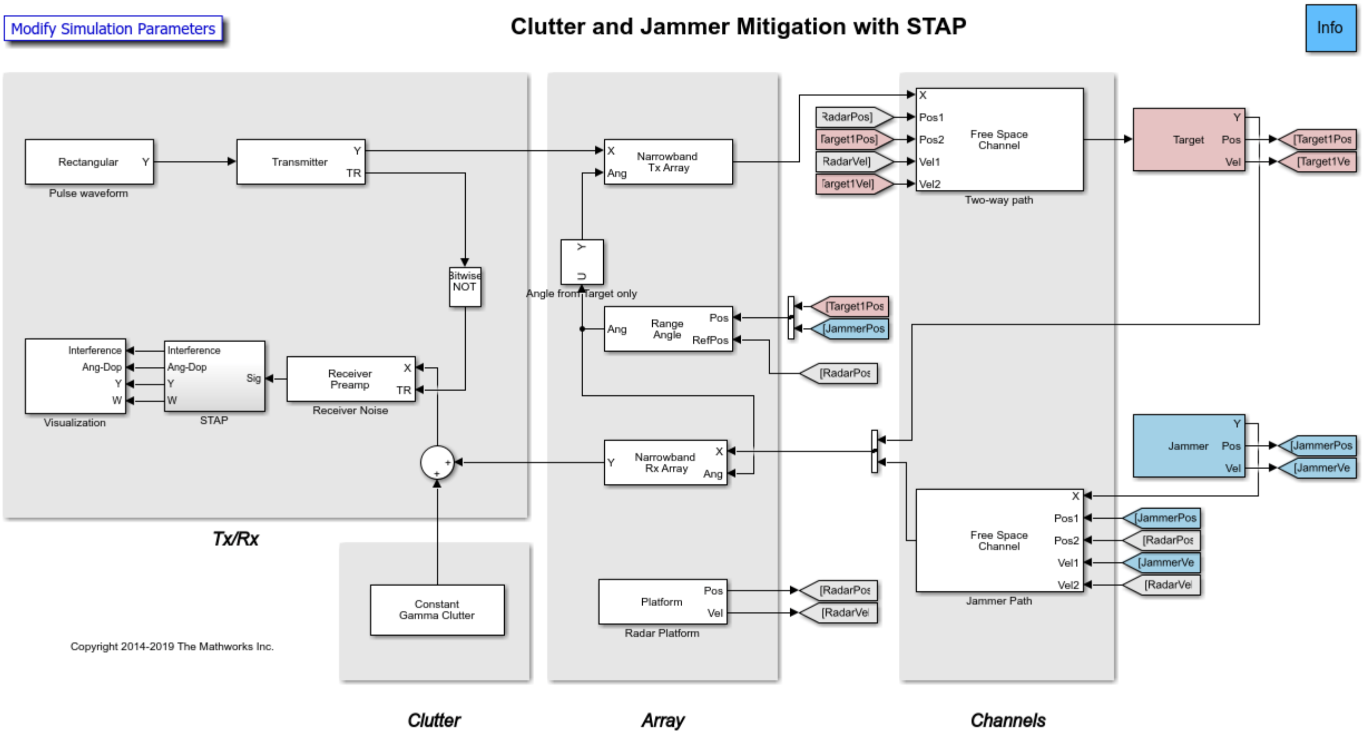
Clutter and Jammer Mitigation with STAP

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 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® please refer to the “Simulating Test Signals for a Radar Receiver in Simulink” example. For an introduction to STAP please refer to 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 6-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 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.

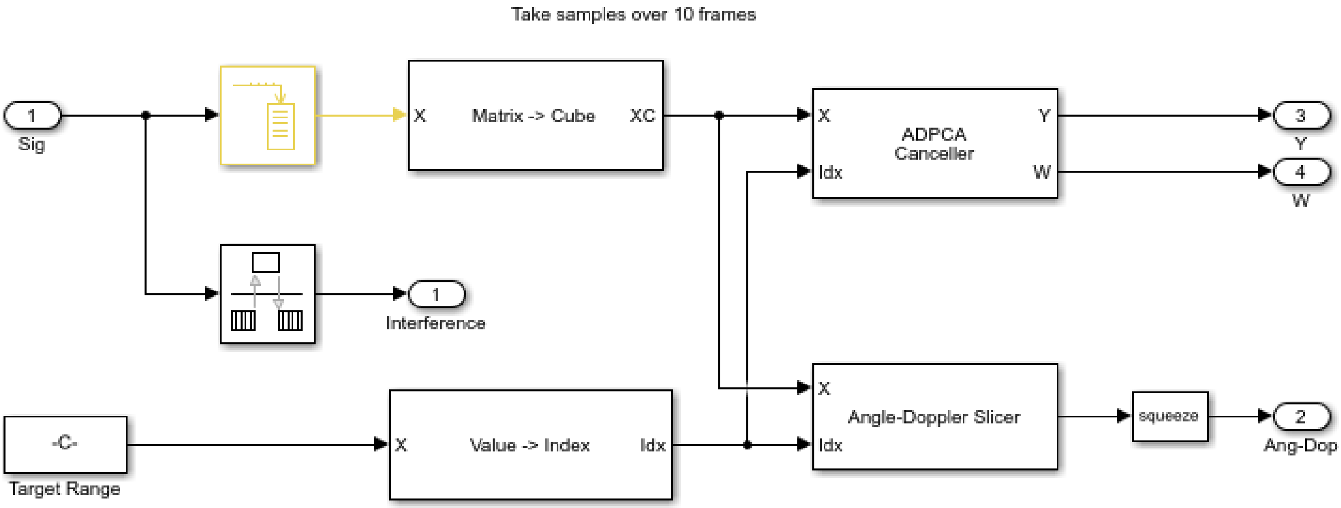
Several blocks in this example need to share the same sensor array configuration. This is done by assigning a sensor array configuration object to a MATLAB variable and sharing this variable in the `Sensor Array` tab of the block's dialog as will be shown later.



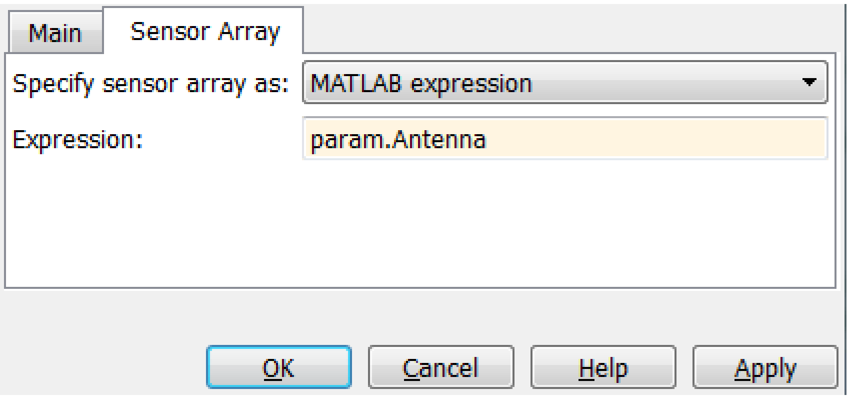
In addition to the blocks listed in the “Simulating Test Signals for a Radar Receiver in Simulink” example, there are:

- FreeSpace - Performs two-way propagation of the signal when two-way propagation is selected on the block's dialog panel. 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 MxQxN 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. In this example the cube has a 200X6X10 dimensions.
- Value to Index - Calculates the index of the estimated target's range bin from the range value.
- ADPCA Canceller - Perform adaptive displaced phase center array (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's dialog. 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.



Specify sensor array as: MATLAB expression

Expression: param.Antenna

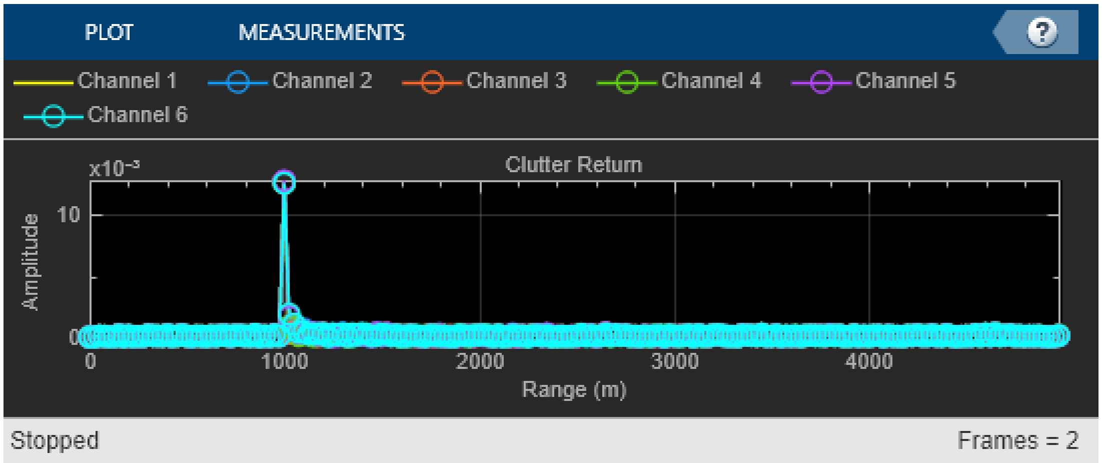
- `Angle Doppler Slicer` - Slices the data cube along the dimension specified by the dialog parameter. This example examines the angle-doppler slice of the cube at the estimated range.
- `Visualization` - This subsystems displays the clutter interference in the time domain, the angle Doppler response of the received data, the output of the ADPCA Canceller as well as the weights.

Exploring the Example

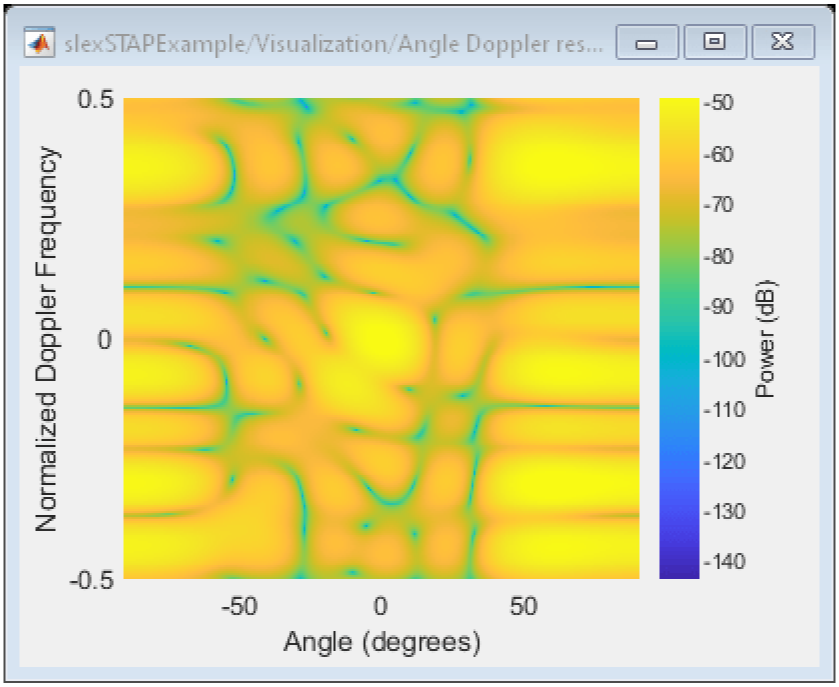
Several dialog parameters of the model are calculated by the helper function helperslexSTAPParam. To open the function from the model, click on `Modify Simulation Parameters` block. This function is executed once when the model is loaded. It exports to the workspace a structure whose fields are referenced 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

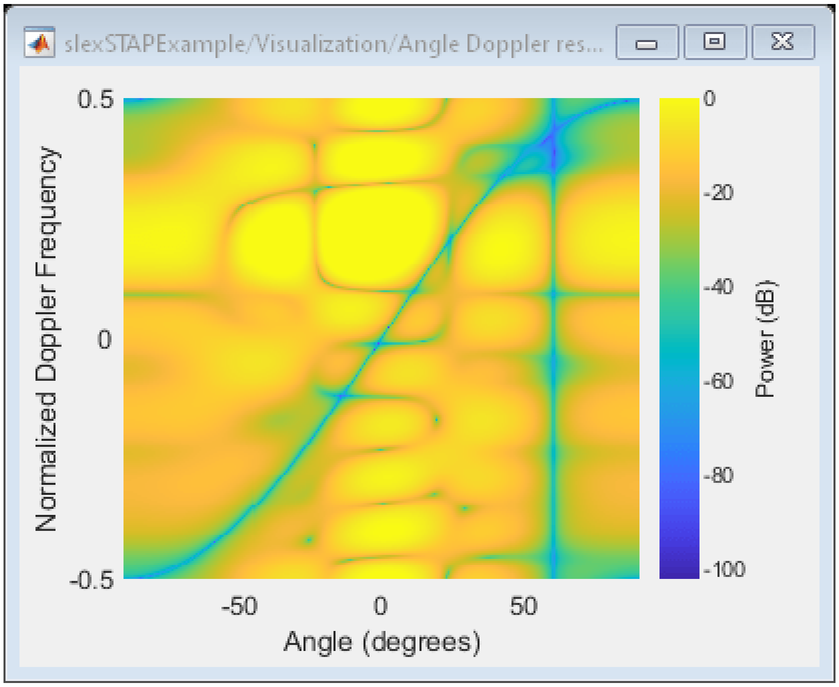
Displays from different stages of the simulation are shown below. The first figure below shows how the signal received at the antenna array is dominated by the clutter return. Because the radar is located 1000 meters above the surface, the clutter returns from the ground start at 1000 meters.



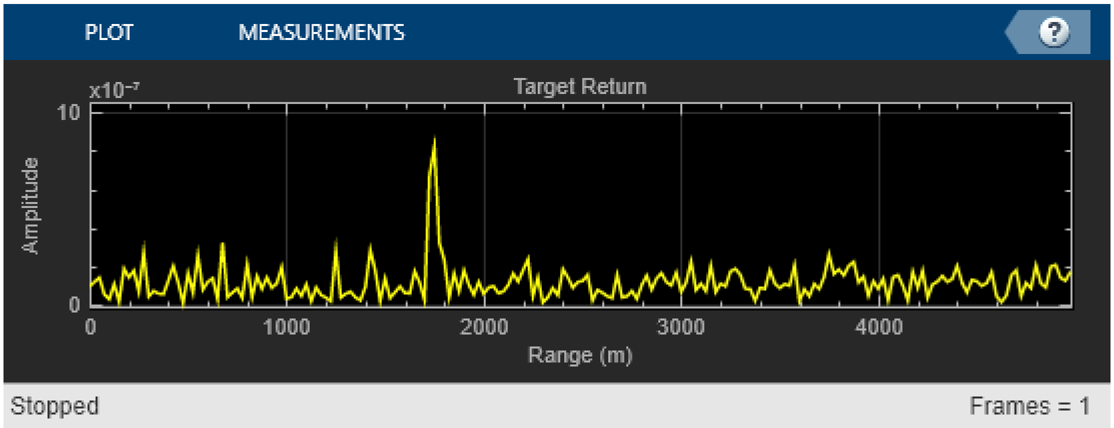
The figure below shows the angle-Doppler response of the return for the estimated range bin. It presents the clutter as a function of angle and Doppler. The clutter return looks like a diagonal line in angle-Doppler space. Such a line is often referred to as *clutter ridge*. The received jammer signal is white noise, spread over the entire Doppler spectrum at approximately 60 degrees.



As you can see in the next figure, the weights of the ADPCA Canceller produce a deep null along the clutter ridge and also in the direction of the jammer.



The figure below 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.


PLOT
MEASUREMENTS
Target Return
x10⁻⁷
10
0
Amplitude
0
1000
2000
3000
4000
Range (m)
Stopped
Frames = 1

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 1-510 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, we simulate returns from target, clutter, and jammer and illustrate how STAP techniques filter the interference from the received signal.

System Setup

We 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
```

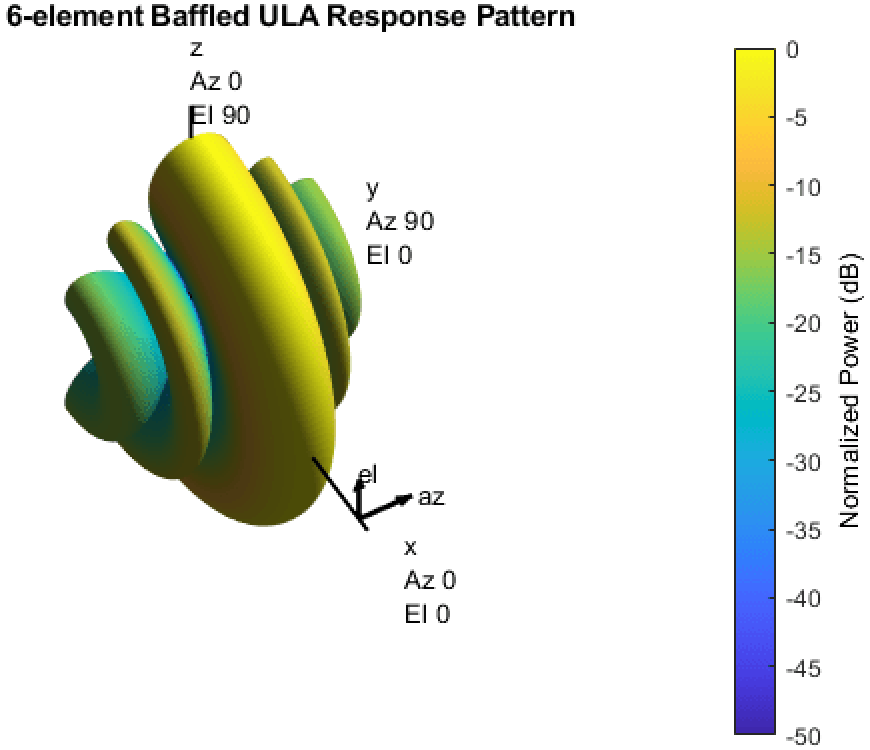
Antenna Definition

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 ULA with a custom element pattern. The element spacing is assumed 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('6-element Baffled ULA Response Pattern')
view(60,50)
```



Radar Setup

Next, mount the antenna array on the radiator/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 Radar Toolbox is available, please set the varaible hasRadarToolbox to true to define a simple barrage jammer with an effective radiated power of 100 watts. Otherwise, the simulation will use 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

In this example we simulate the 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 Radar Toolbox is not available, the simulation will use 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 we are using 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

We are now ready to simulate the returns. Collect 10 pulses before processing. 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
```

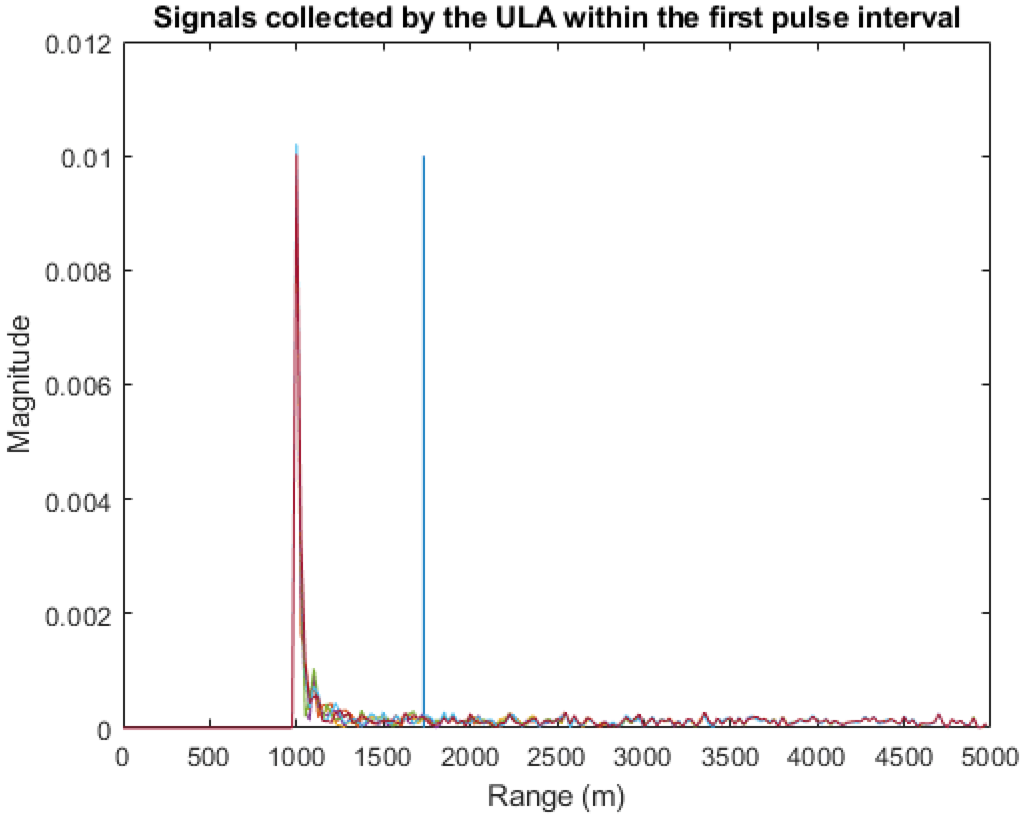
The target Doppler normalized frequency is about 0.21.

```
sp = radialspeed(tgtpos, tgtmotion.Velocity, ...
                sensorpos, sensormotion.Velocity);
tgtDp = 2*speed2dop(sp,lambda);  % Round trip Doppler
tgtDp/prf
```

```
ans = 0.2116
```

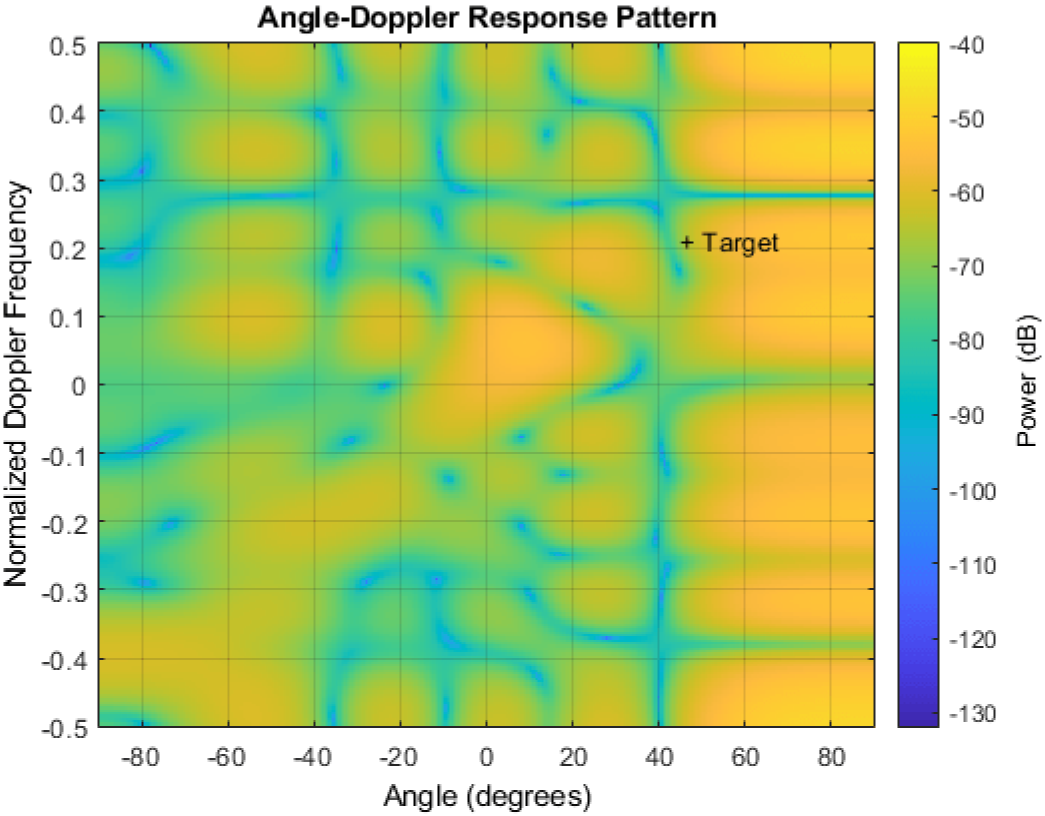
The total received signal contains returns from the target, clutter and jammer combined. The signal is a data cube with three dimensions (range bins x number of elements x number of pulses). Notice that the clutter return dominates the total return and masks the target return. We cannot detect the target (blue vertical line) without further processing at this stage.

```
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')
```



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 we know exactly where the target is, we 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')
```



If we look at the angle Doppler response which is dominated by the clutter return, we 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 antenna's array axis 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 may need to search through all combinations of angle and Doppler to locate a target, but for the example, because we know exactly where the target is, we can direct our processor to that point.

```
rxmainlobedir = [0; 0];
stapdpca = phased.DPCACanceller('SensorArray',ula,'PRF',prf,...
    'PropagationSpeed',c,'OperatingFrequency',fc,...
    'Direction',rxmainlobedir,'Doppler',tgtDp,...
    'WeightsOutputPort',true)
```

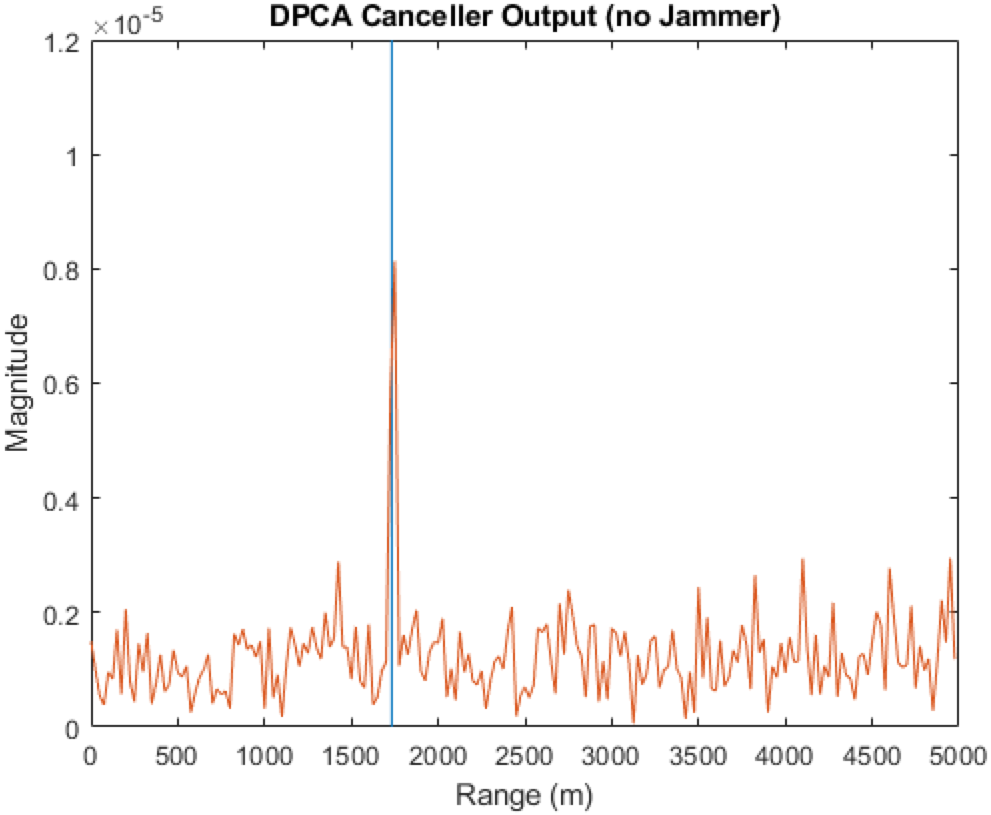
```
stapdpca = 
  phased.DPCACanceller with properties:

            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);
```

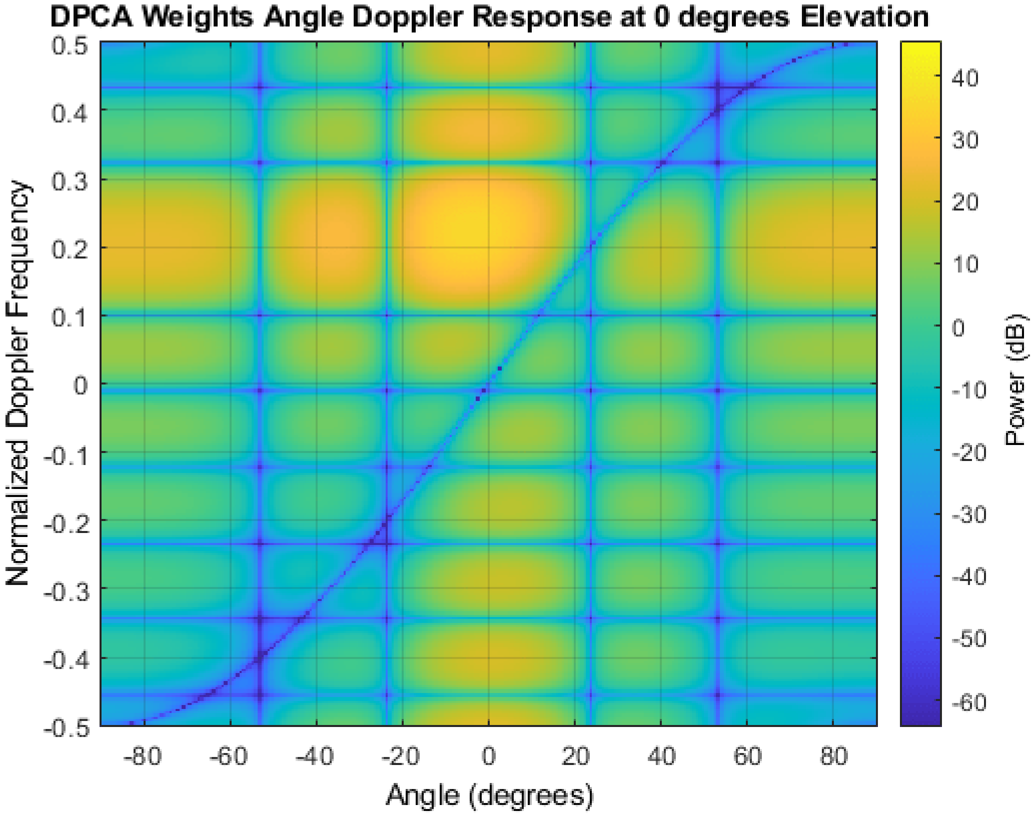
The processed data combines all information in space and across the pulses to become a single pulse. Next, examine the processed signal in the time domain.

```
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, we can also see that the weights produce a deep null along the clutter ridge.

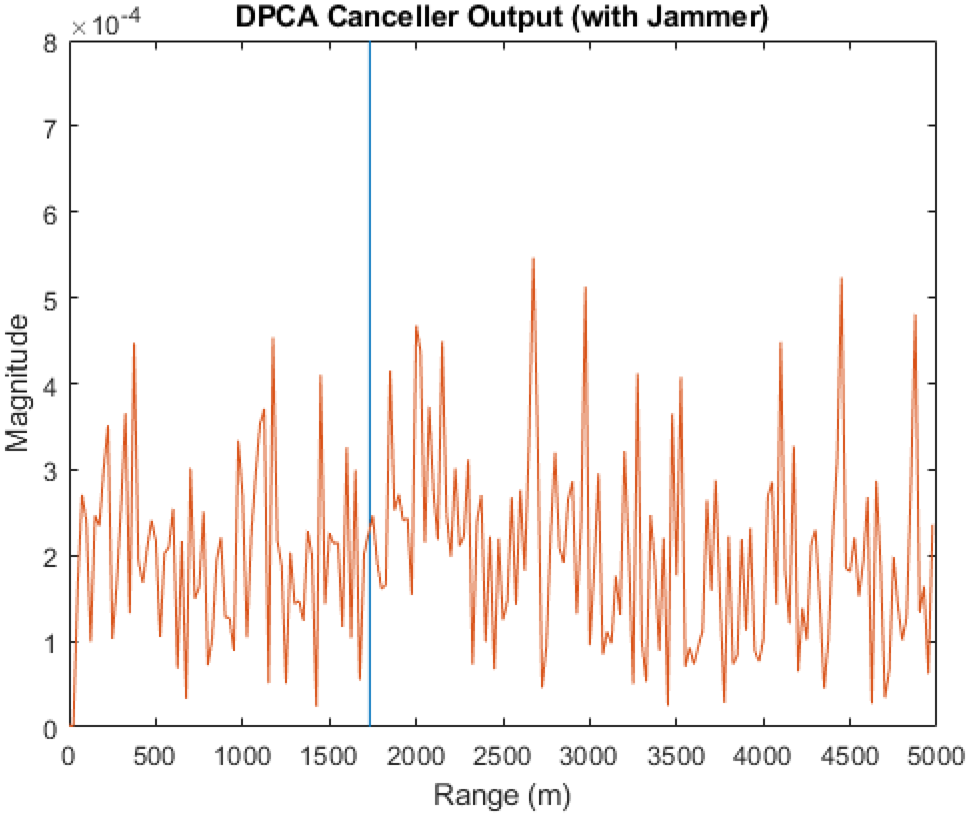
```
angdopresp.ElevationAngle = 0;
plotResponse(angdopresp,w,'NormalizeDoppler',true);
title('DPCA Weights Angle Doppler Response at 0 degrees Elevation')
```



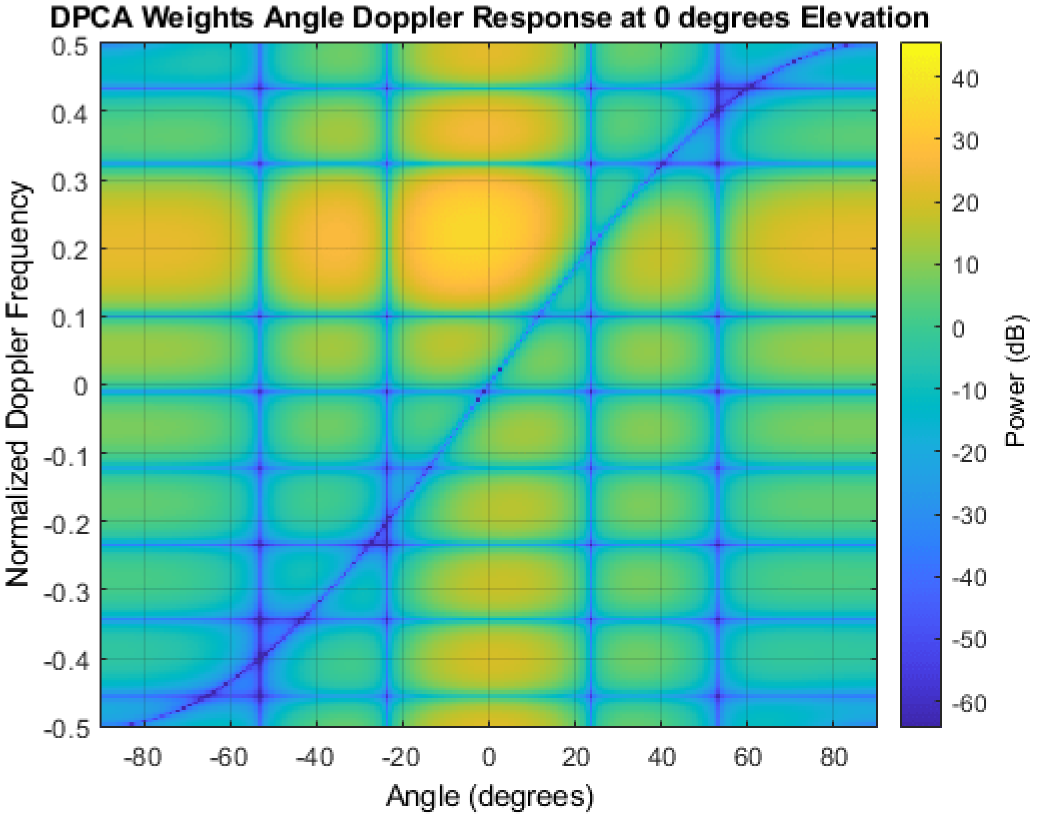
Although the results obtained by DPCA are very good, the radar platform has to satisfy very strict requirements in its movement to use this technique. Also, the DPCA technique cannot suppress the jammer interference.

Applying DPCA processing to the total signal produces the result shown in the following figure. We 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')
```



Clutter and Jammer Suppression with an SMI Beamformer

To suppress the clutter and jammer simultaneously, we need a more sophisticated algorithm. The optimum receiver weights, when the interference is Gaussian-distributed, are given by [1]

$$\mathbf{w} = k\mathbf{R}^{-1}\mathbf{s}$$

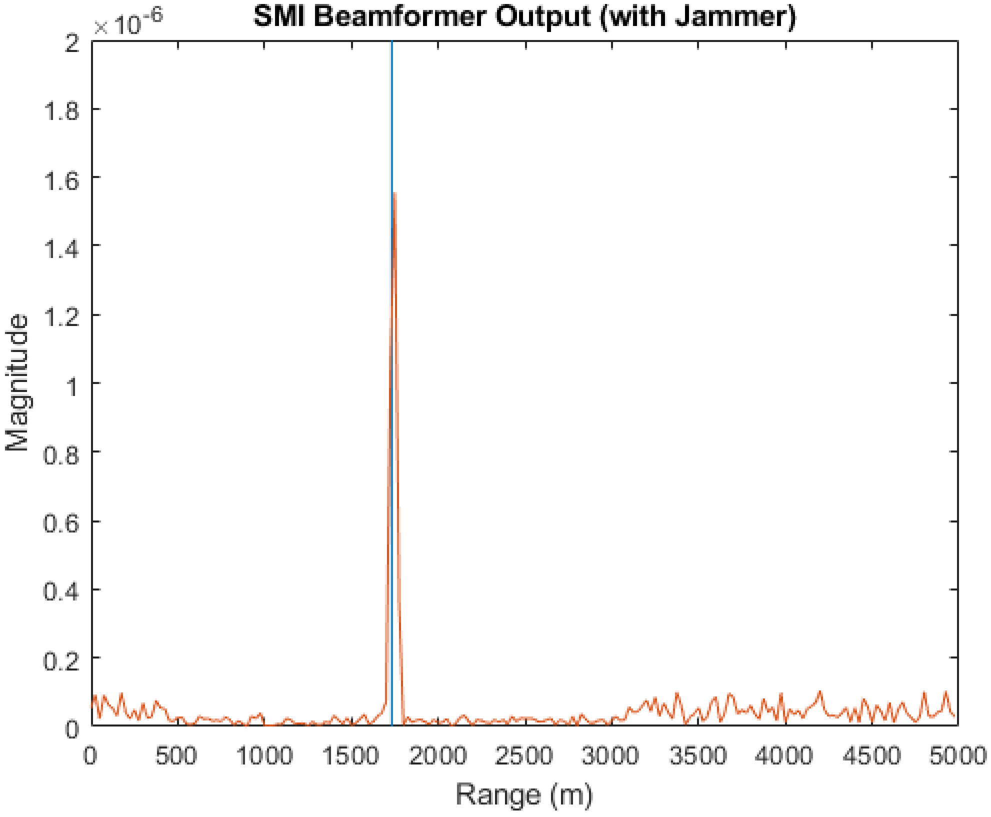
where k is a scalar factor, $\mathbf{R}$ is the space-time covariance matrix of the interference signal, and $\mathbf{s}$ is the desired space-time steering vector. The exact information of $\mathbf{R}$ is often unavailable, so we will use the sample matrix inversion (SMI) algorithm. The algorithm estimates $\mathbf{R}$ 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. Thus, we should not use the cells that are close to the target cell for the estimates because they may contain some target information, i.e., we should 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.

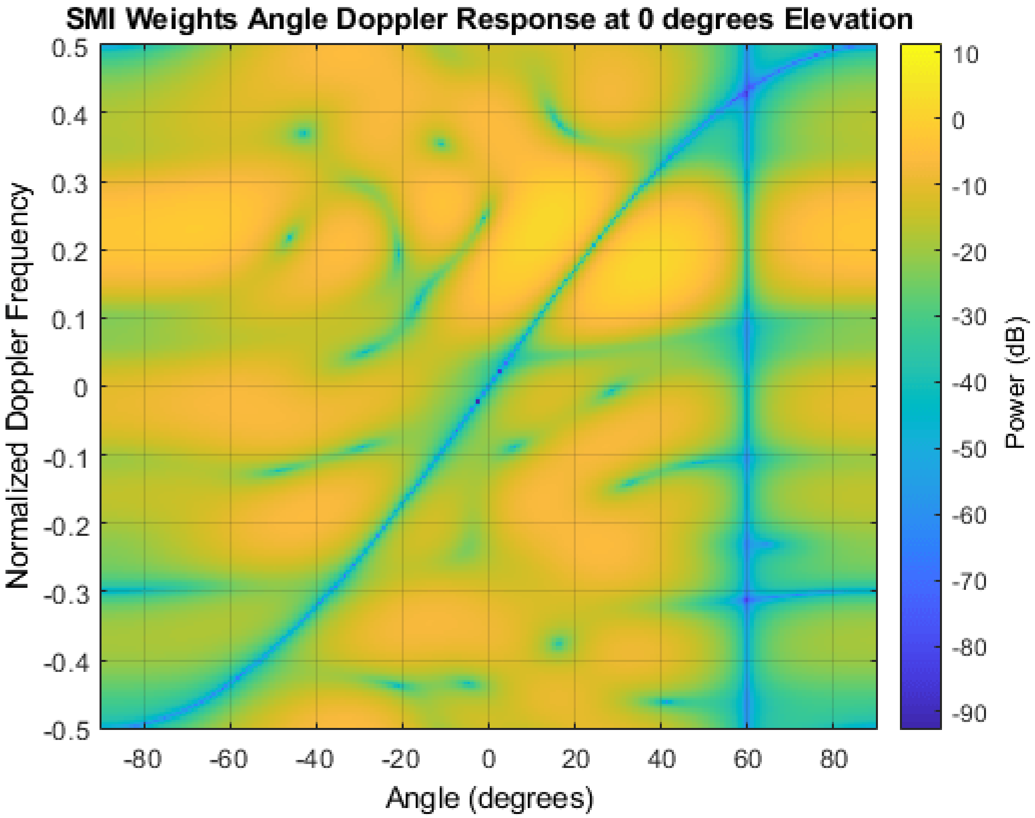
```
tgtAngle = [tgtAzAngle; tgtElAngle];
stapsmi = phased.STAPSMIBeamformer('SensorArray', ula, 'PRF', prf, ...
    'PropagationSpeed', c, 'OperatingFrequency', fc, ...
    'Direction', tgtAngle, 'Doppler', tgtDp, ...
    'WeightsOutputPort', true,...
    'NumGuardCells', 4, 'NumTrainingCells', 100)
```

```
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(angdopresp,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.

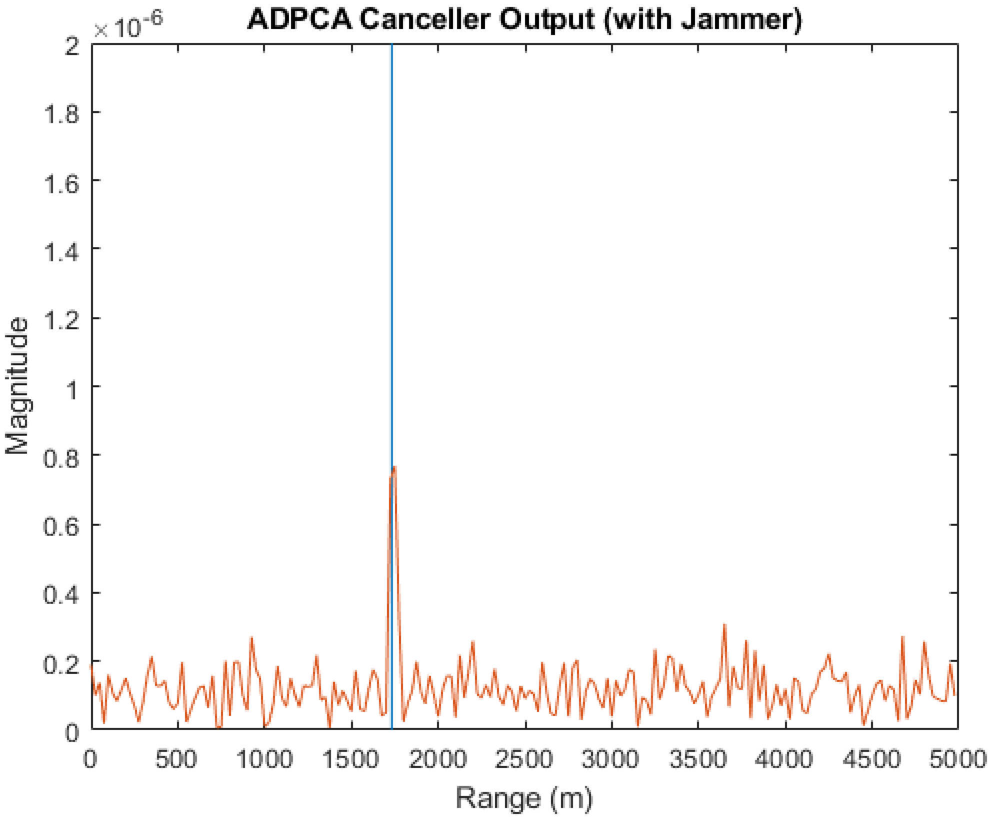
An adaptive DPCA (ADPCA) canceller filters out the clutter in the same manner as DPCA, but it also has the capability to suppress the jammer as it estimates the interference covariance matrix using two consecutive pulses. Because there are only two pulses involved, the computation is greatly reduced. 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.

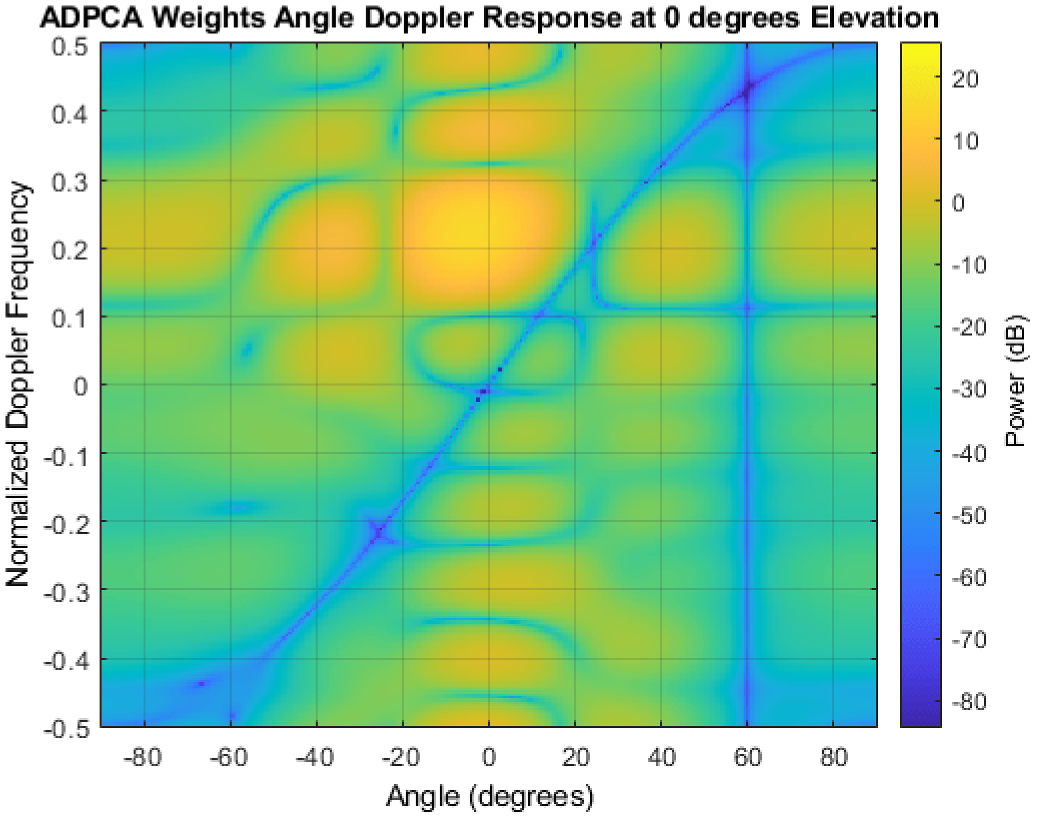
```
stapadpca = phased.ADPCACanceller('SensorArray', ula, 'PRF', prf, ...
    'PropagationSpeed', c, 'OperatingFrequency', fc, ...
    'Direction', rxmainlobedir, 'Doppler', tgtDp, ...
    'WeightsOutputPort', true,...
    'NumGuardCells', 4, 'NumTrainingCells', 100)
```

```
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
```

```
[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)')
```



```
plotResponse(angdopresp,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 space-time adaptive processing 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] J. R. Guerci, *Space-Time Adaptive Processing for Radar*, 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 6-element uniform linear array (ULA). The example focuses on comparing the performance of clutter simulation between GPU, code generation and the MATLAB interpreter. Interested readers 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.

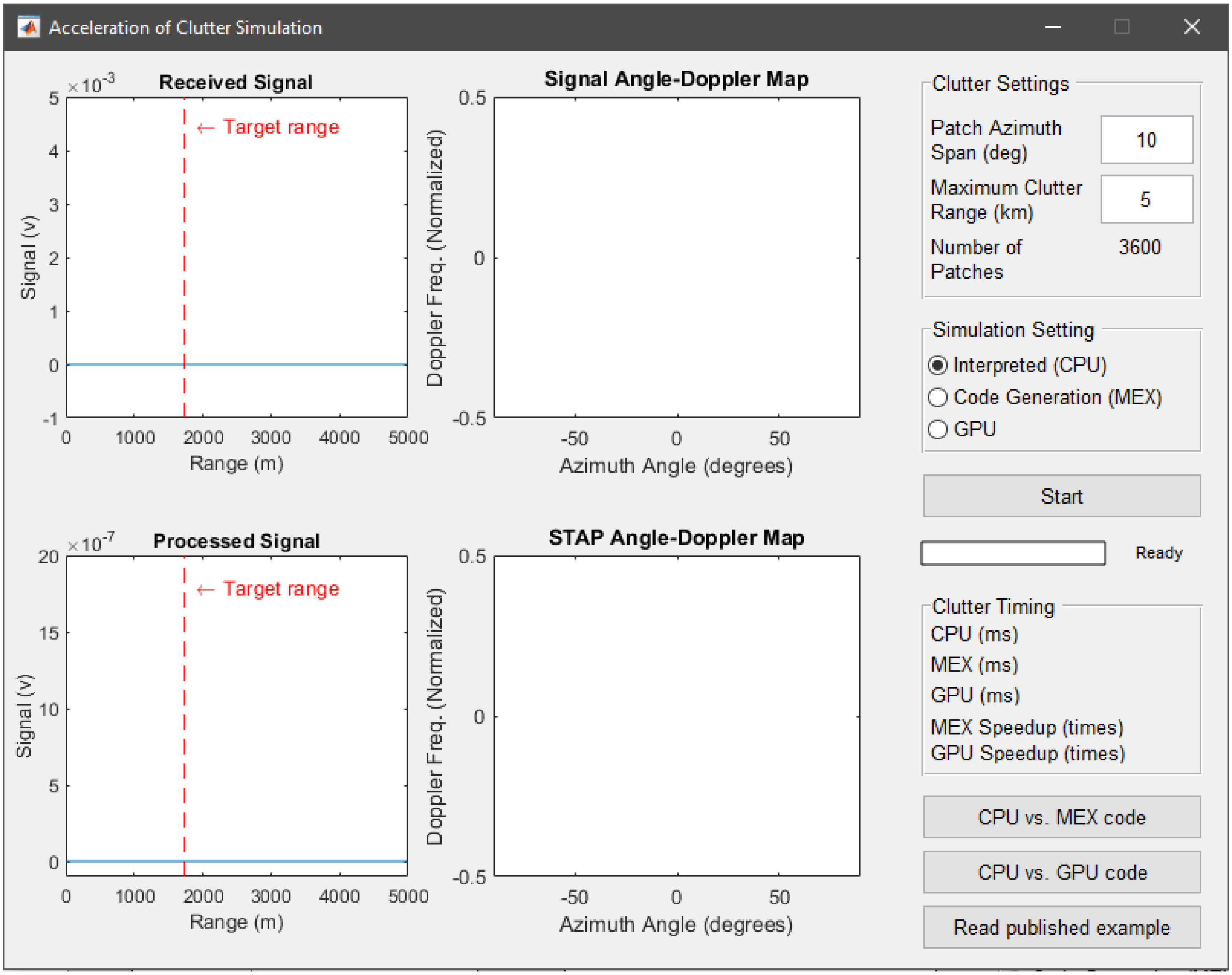
If you have access to MATLAB Coder, you can also speed up clutter simulation by generating C code for `constantGammaClutter`, compiling it and running the compiled version. When running in code generation mode, this example compiles stapclutter using the codegen command:

```
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 the 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:



Clutter Settings

Patch Azimuth Span (deg) 10

Maximum Clutter Range (km) 5

Number of Patches 3600

Simulation Setting

● Interpreted (CPU)
○ Code Generation (MEX)
○ GPU

Clutter Timing

CPU (ms)
MEX (ms)
GPU (ms)
MEX Speedup (times)
GPU Speedup (times)

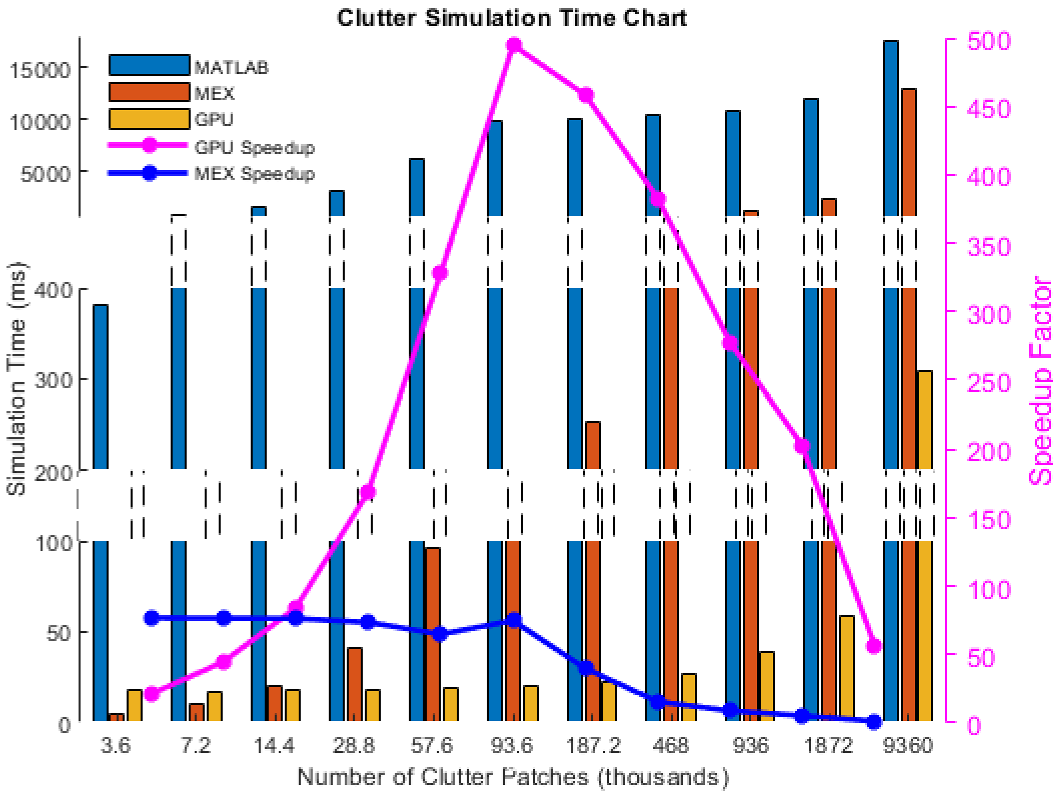
The left side of the GUI contains four plots, showing the raw received signal, the angle-Doppler response of the received signal, the processed signal, and the angle-Doppler response of the STAP processing weights. Again, the details can be found in the example “Introduction to Space-Time Adaptive Processing”. 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.

Next section shows 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` pre-calculates the collected clutter as an array of constant values. For 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 200x6x10, 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 5000x50x50 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

$$\sigma = \lim_{R->\infty} 4\pi R^2 \frac{|E_s|^2}{|E_i|^2}$$

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

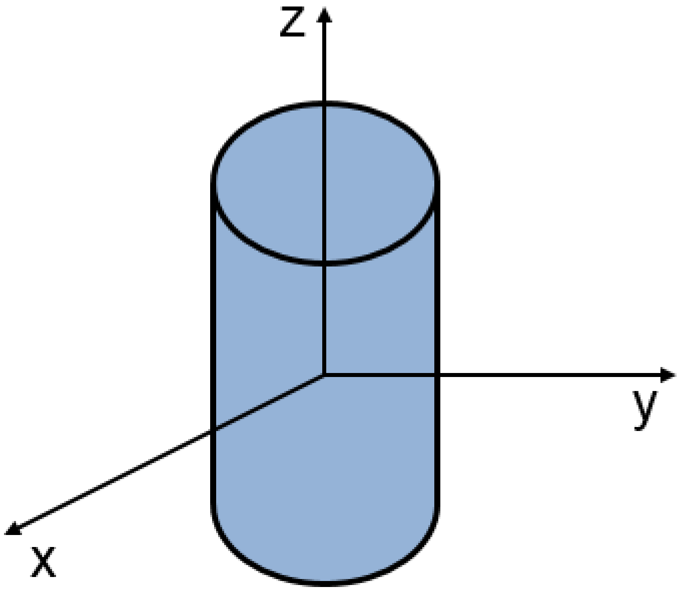
$$G = \frac{4\pi\sigma}{\lambda^2}$$

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.

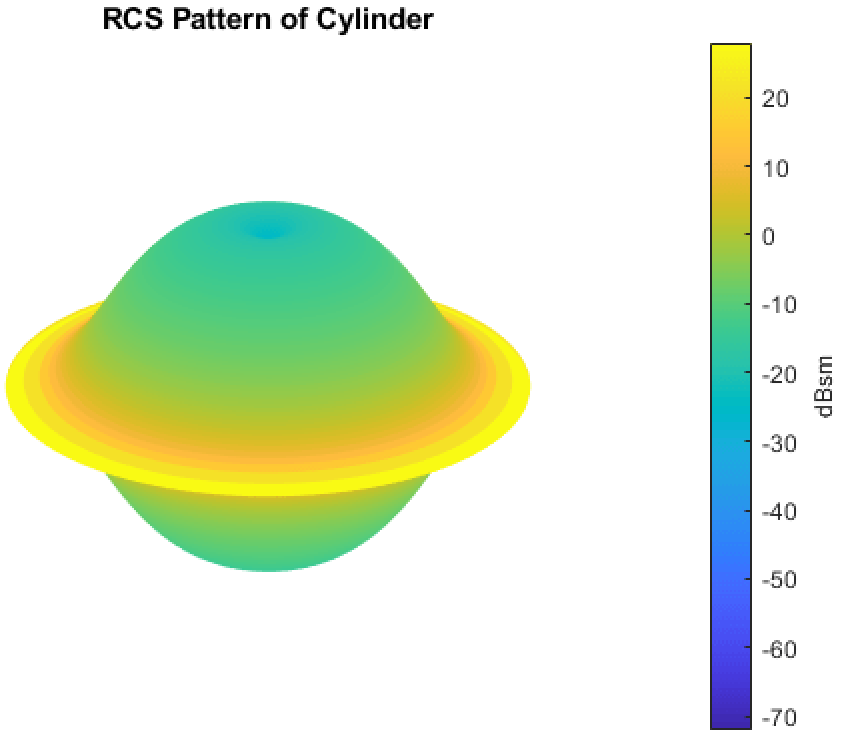
The example below first computes the RCS pattern of a cylindrical target, with a radius of 1 meter and a height of 10 meters, as a function of azimuth and elevation angles.



```
[cylrcs,az,el] = rcscylinder(1,1,10,c,fc);
```

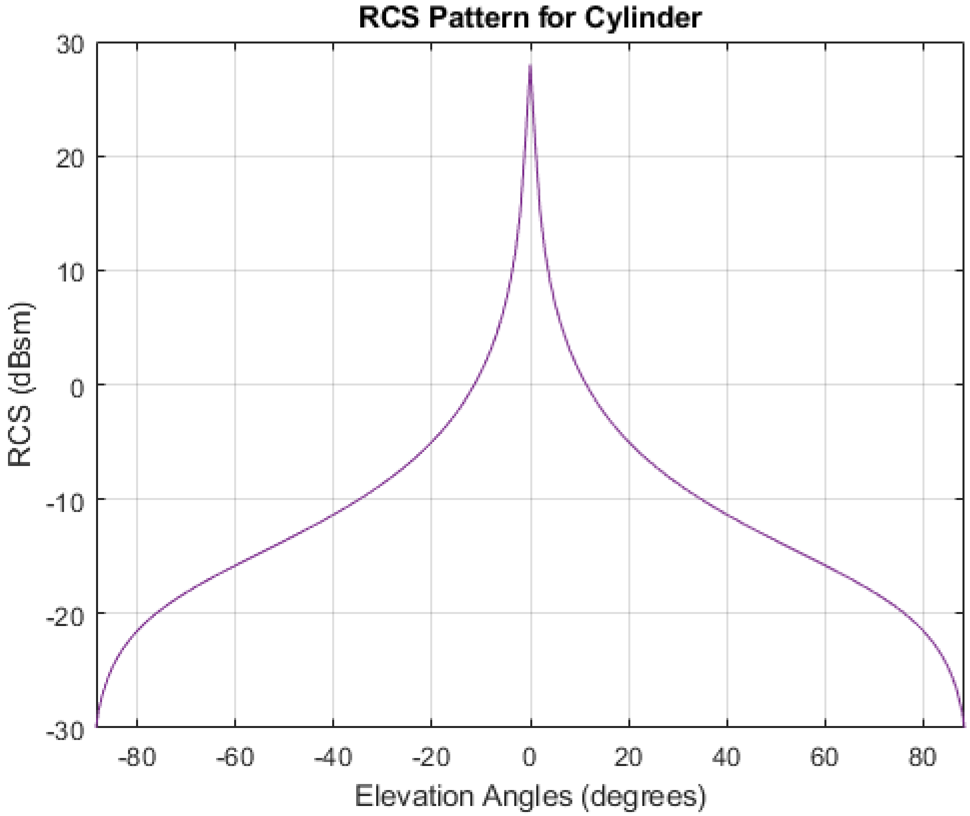
Because the cylinder is symmetric around the z axis, there is no azimuth-angle dependency. RCS values vary only with elevation angle.

```
helperTargetRCSPatternPlot(az,el,cylrcs);
title('RCS Pattern of Cylinder');
```



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');
```



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.

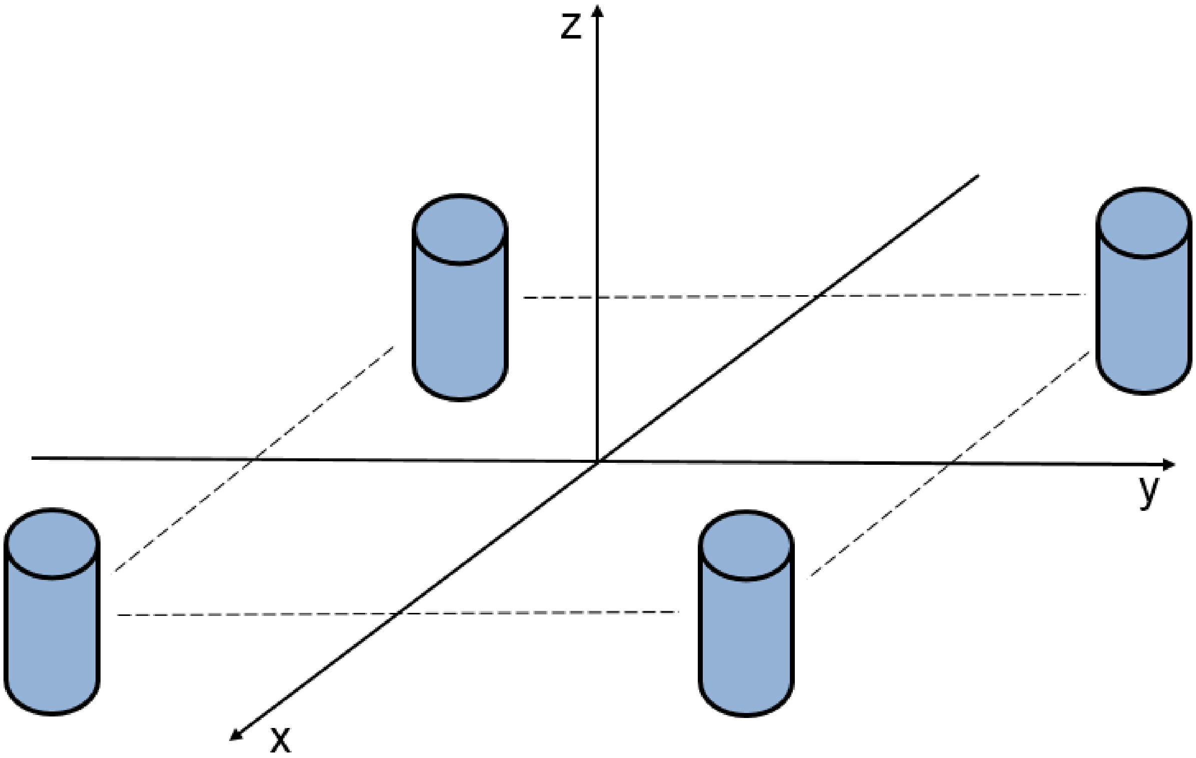
RCS of Extended Targets with Multiple Scatterers

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.

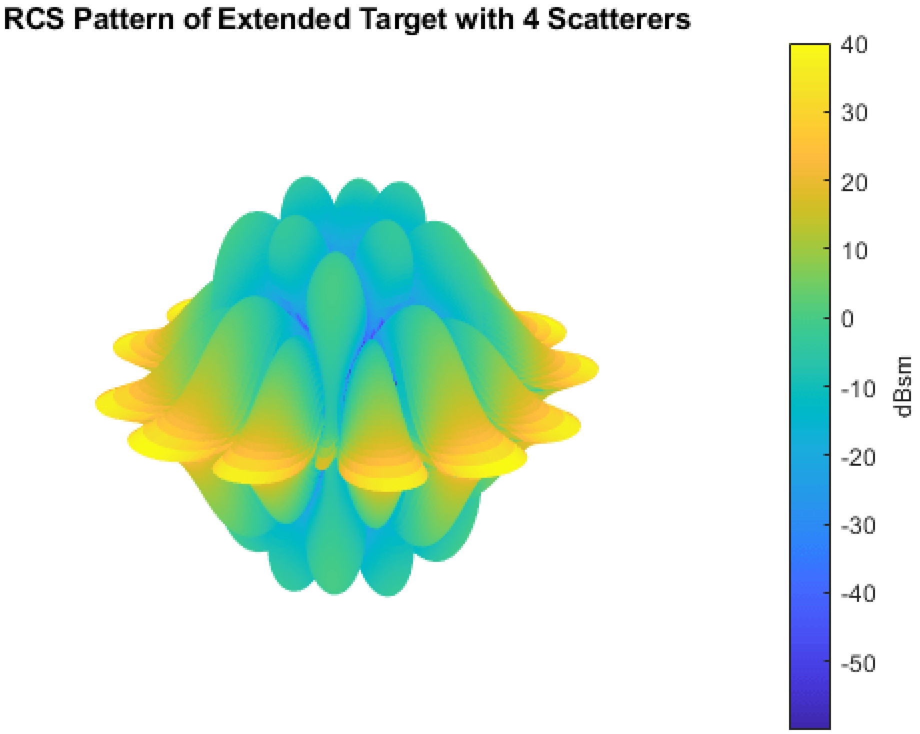
```
scatpos = [-0.5 -0.5 0.5 0.5;0.5 -0.5 0.5 -0.5;0 0 0 0];
```

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');
```



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)
```

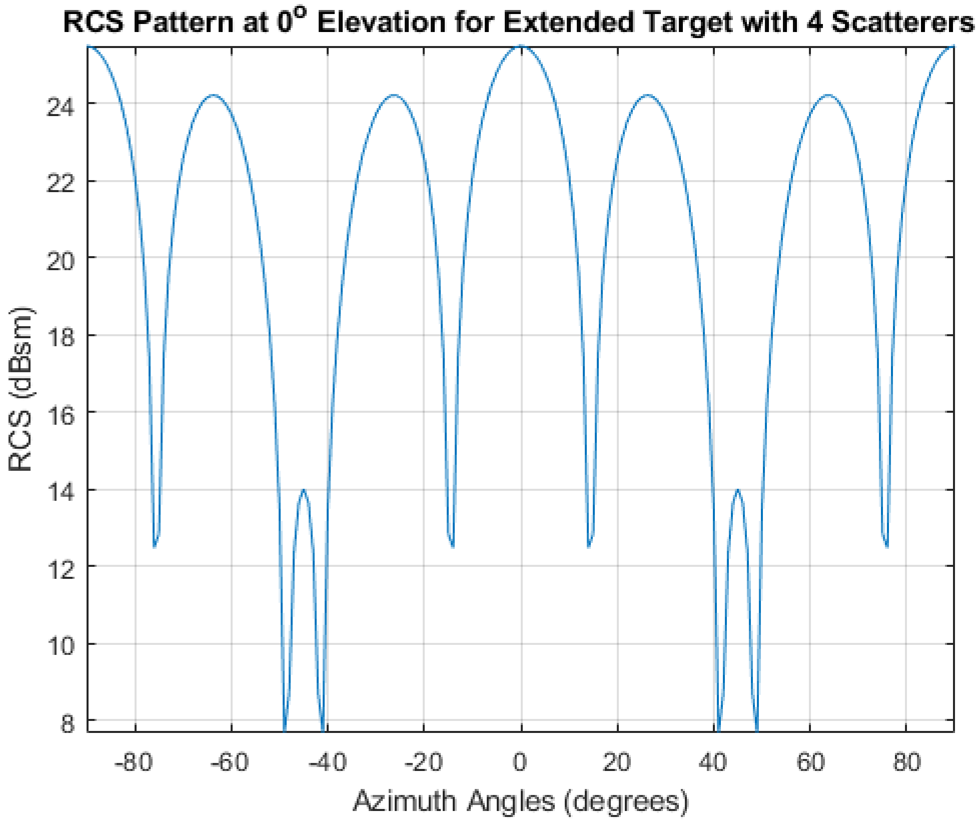
```
ext_echo =

  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;
```

```
[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

```
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.

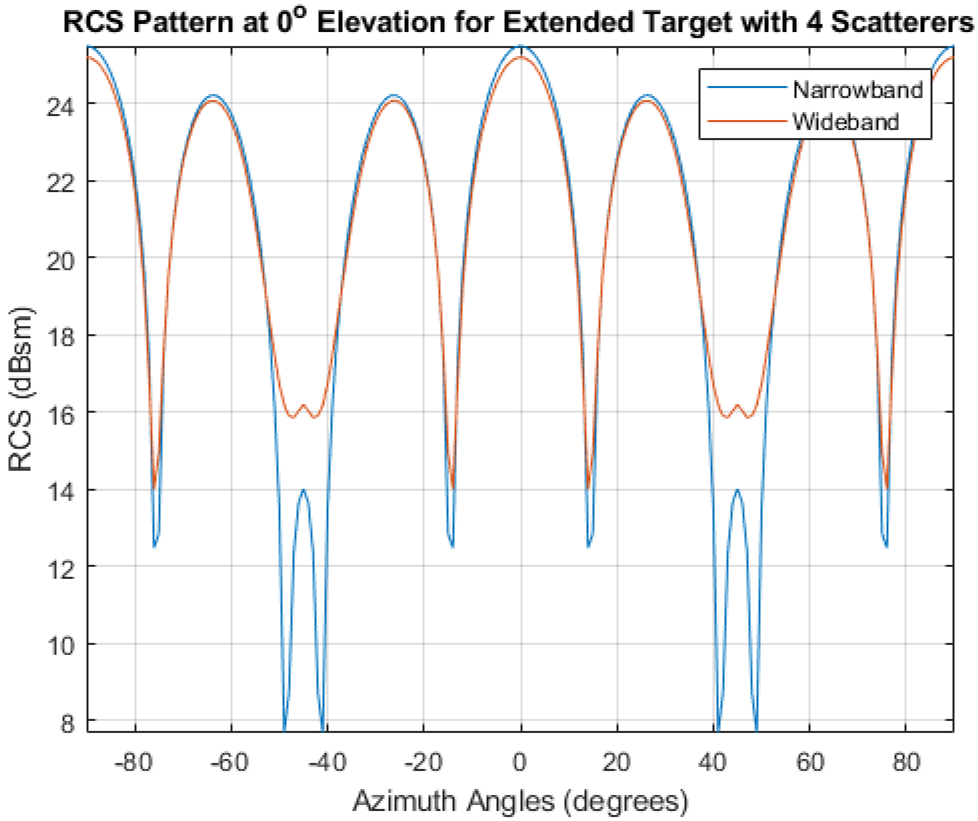
```
nf = numel(modelFreq);
naz = numel(modelAz);
nel = numel(modelEl);
modelExtRCS = zeros(nel,naz,nf);
for k = 1:nf
    for m = 1:nel
        pos = scatpos*modelFreq(k)/fc;
        sv = steervec(pos,[modelAz;modelEl(m)*ones(1,naz)]);
        % sv is squared due to round trip in a monostatic scenario
        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'
```

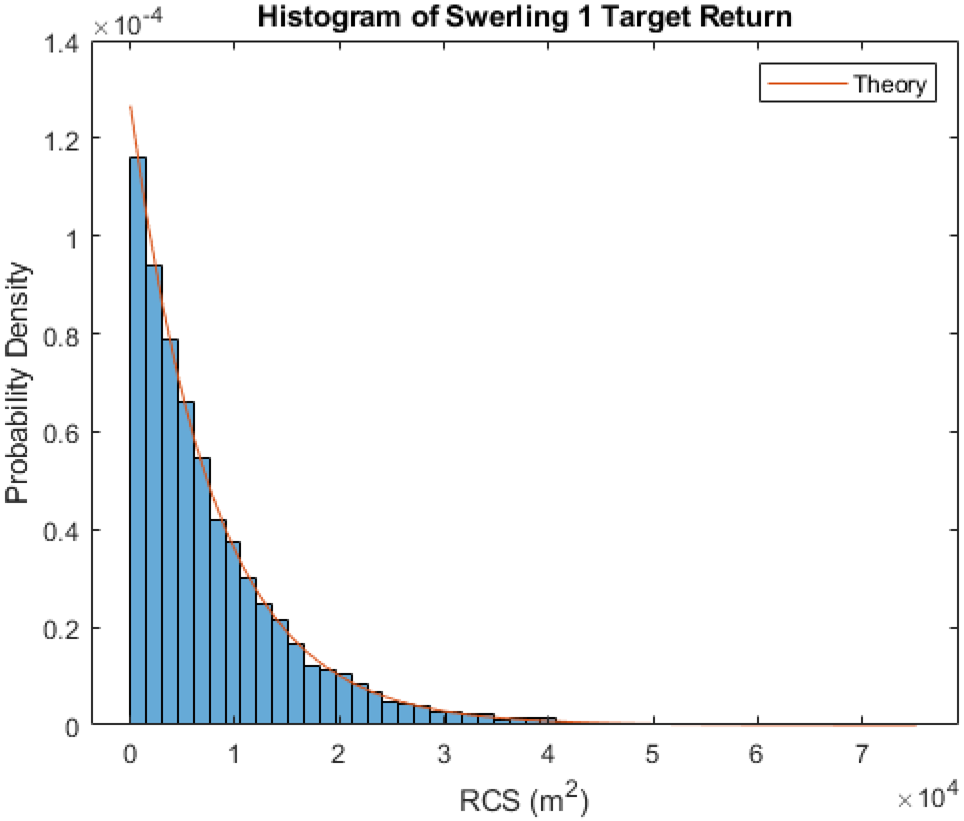
In the Swerling 1 case, the reflection is no longer constant. The RCS value varies from scan to scan. Assuming that the target is illuminated by the signal only once per dwell, the following code simulates the reflected signal power for 10,000 scans for a unit incident signal.

```
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 a 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 waveform's sweep bandwidth.

The transmit antenna array is a stationary 4-element ULA located at 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 4-element ULA; it is located at [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 at [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/vertical polarization components of the input signal become the vertical/horizontal polarization components of the output signal.

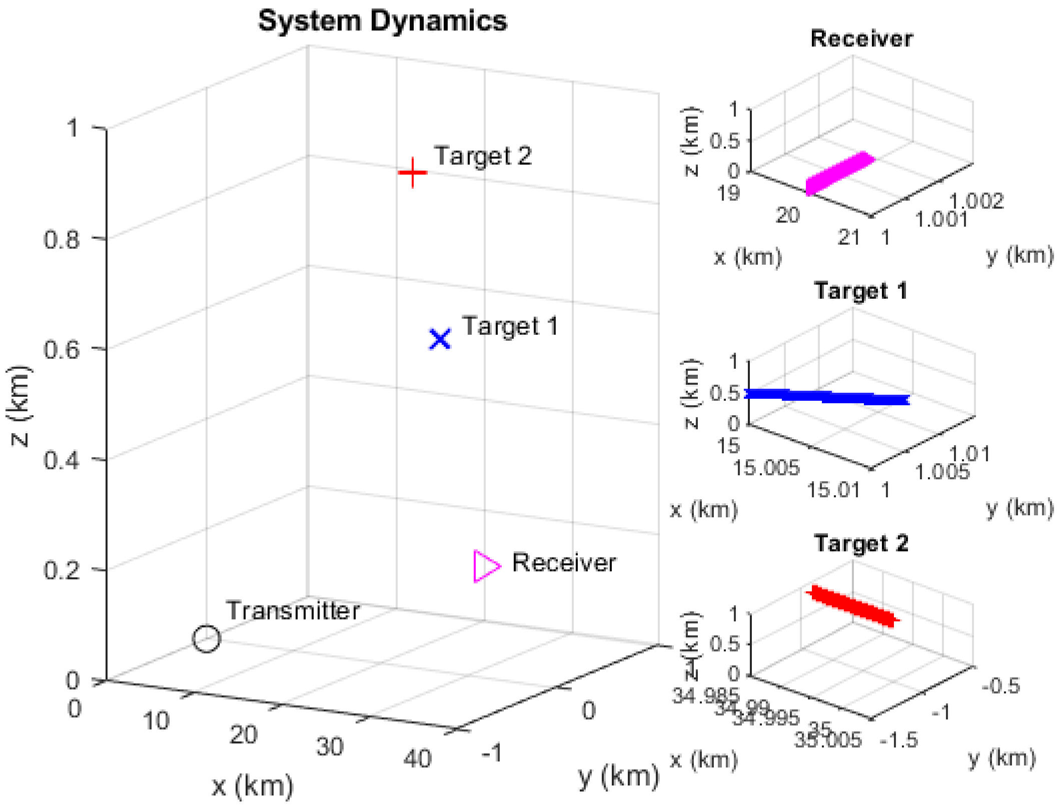
```
[target,tgtmotion,txchannel,rxchannel] = ...
    helperBistatTargetSetup(waveform.SampleRate);
```

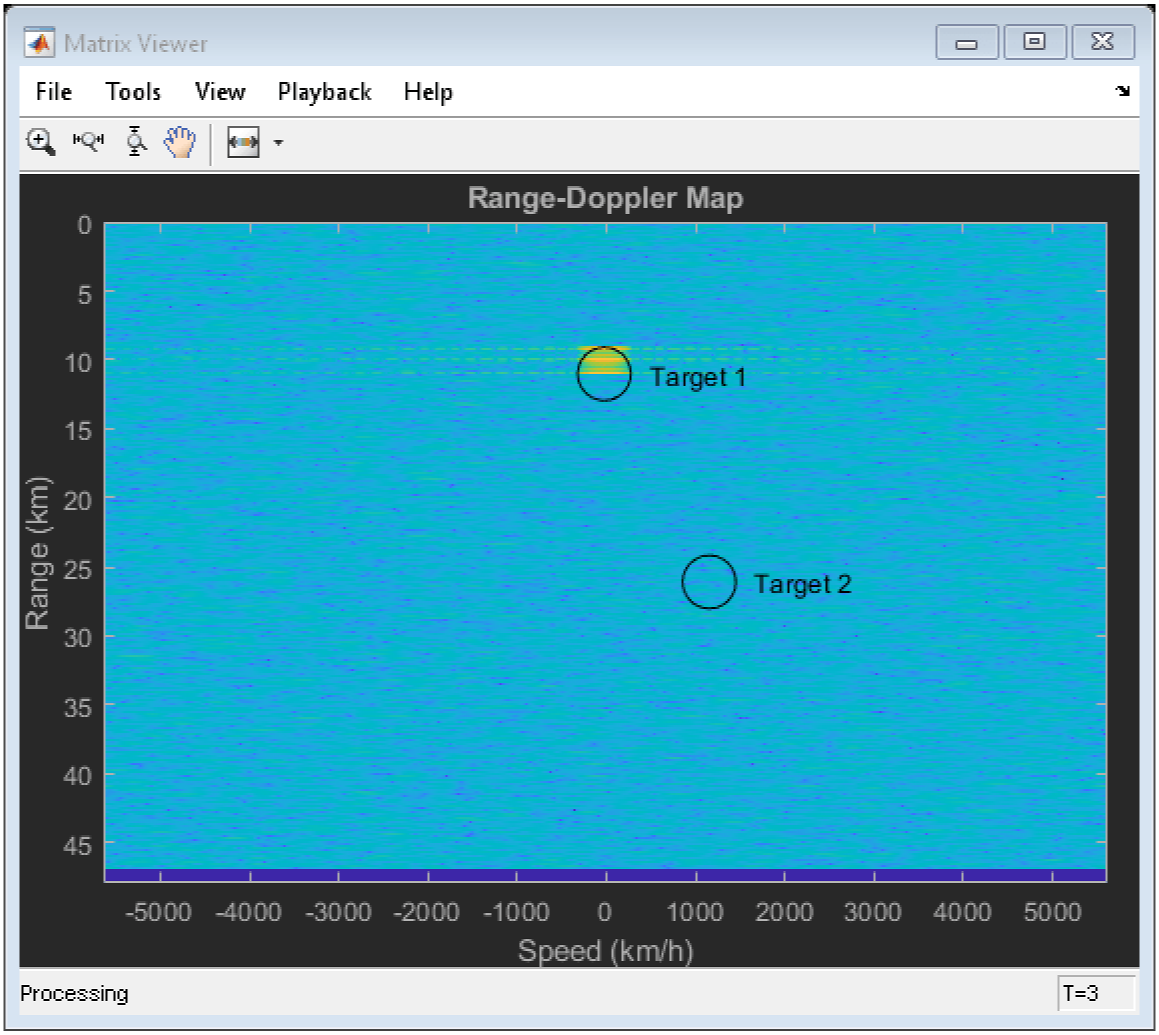
A single scattering matrix is a fairly simple polarimetric model for a target. It assumes that no matter what the incident and reflecting directions are, the power distribution between the H and V components is fixed. However, even such a simple model can reveal complicated target behavior in the simulation because (1) the H and V directions vary for different incident and reflecting directions; and (2) the orientation, defined by the local coordinate system, of the targets 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.

```
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
```


System Dynamics
Target 2
Target 1
Receiver
Transmitter
z (km)
1
0.8
0.6
0.4
0.2
0
0
10
20
30
40
x (km)
0
-1
y (km)
Receiver
z (km)
1
0.5
0
19
20
21
x (km)
1
1.001
1.002
y (km)
Target 1
z (km)
1
0.5
0
15
15.005
15.01
x (km)
1
1.005
1.01
y (km)
Target 2
z (km)
1
0.5
0
34.985
34.99
34.995
35
35.005
x (km)
-1.5
-1
-0.5
y (km)



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 receive array's polarization, resulting in significant polarization loss.

One may 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 those are beyond the scope of this example.

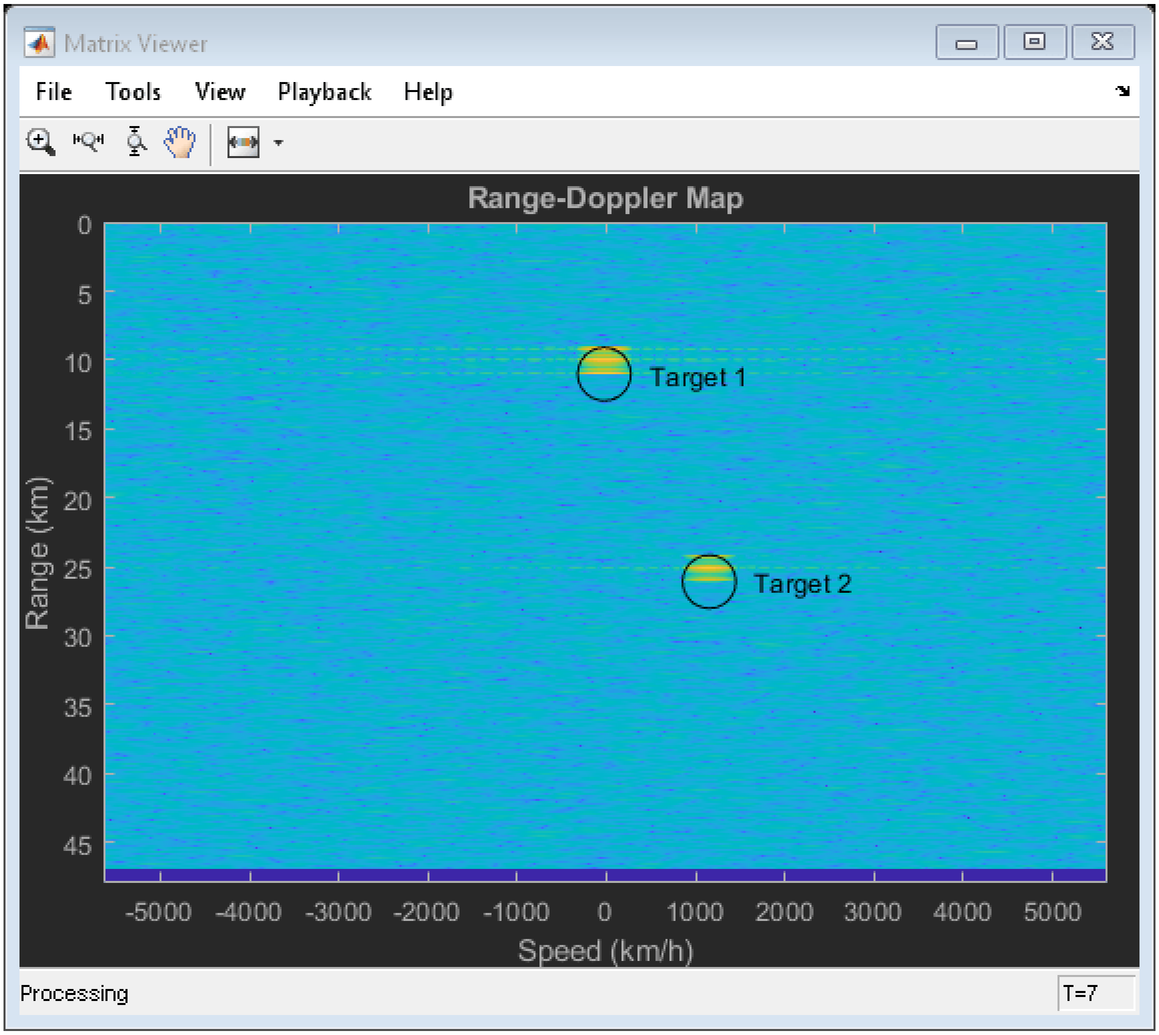
Using Circularly Polarized Receive Array

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 huge.

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

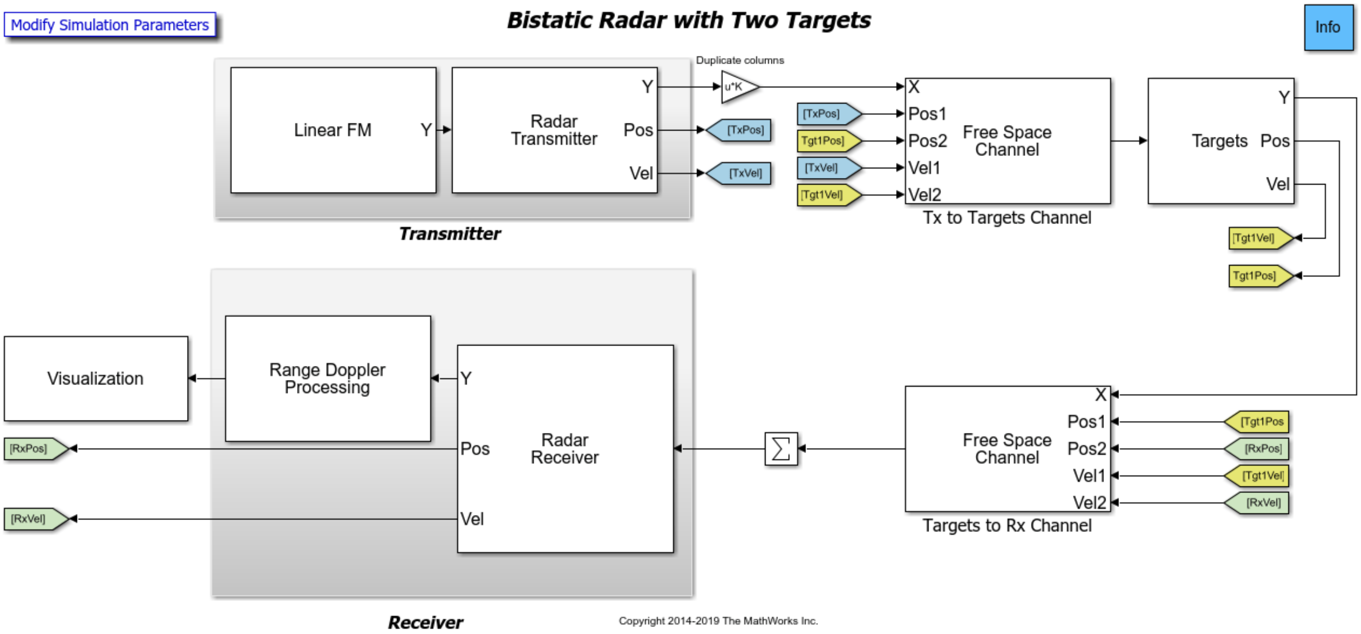
This example shows a system level simulation of a bistatic polarimetric radar. The example generates range-Doppler maps of the received signal for different transmit/receive array polarization configurations and shows how a circularly polarized antenna can be used to avoid losing linear polarized signals due to a target's polarization scattering property.

Simulating a Bistatic Radar with Two Targets

This example shows how to simulates 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.

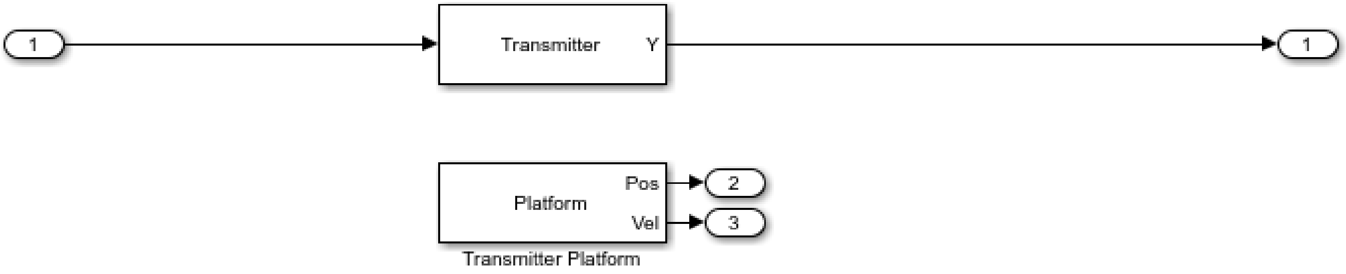
Exploring the Example

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

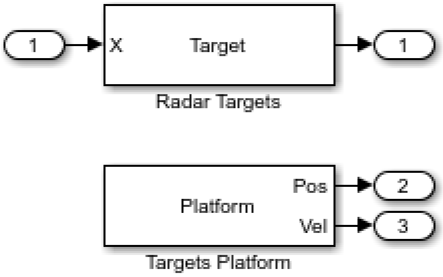
- `Linear FM` - Creates linear FM pulse as the transmitter waveform. The signal sweeps a 3 MHz bandwidth, corresponding to a 50-meter range resolution.
- `Radar Transmitter` - Amplifies the pulse and simulates the transmitter motion. In this case, the transmitter is mounted on a stationary platform located at the origin. The operating frequency of the transmitter is 300 MHz.



Targets

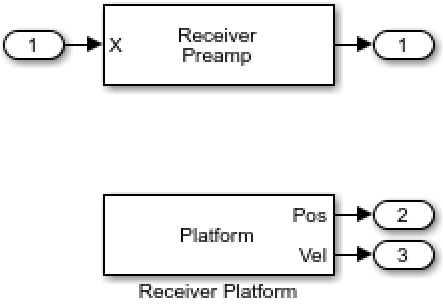
This example includes two targets with similar configurations. The targets are mounted on the moving platforms.

- `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.

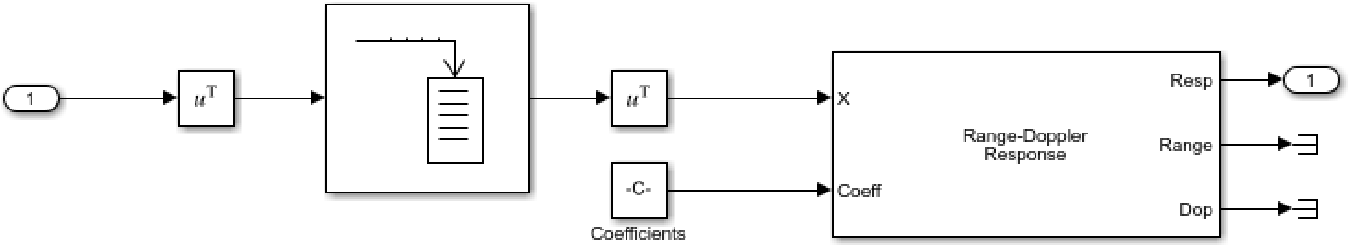


Receiver

- `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.



- `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.

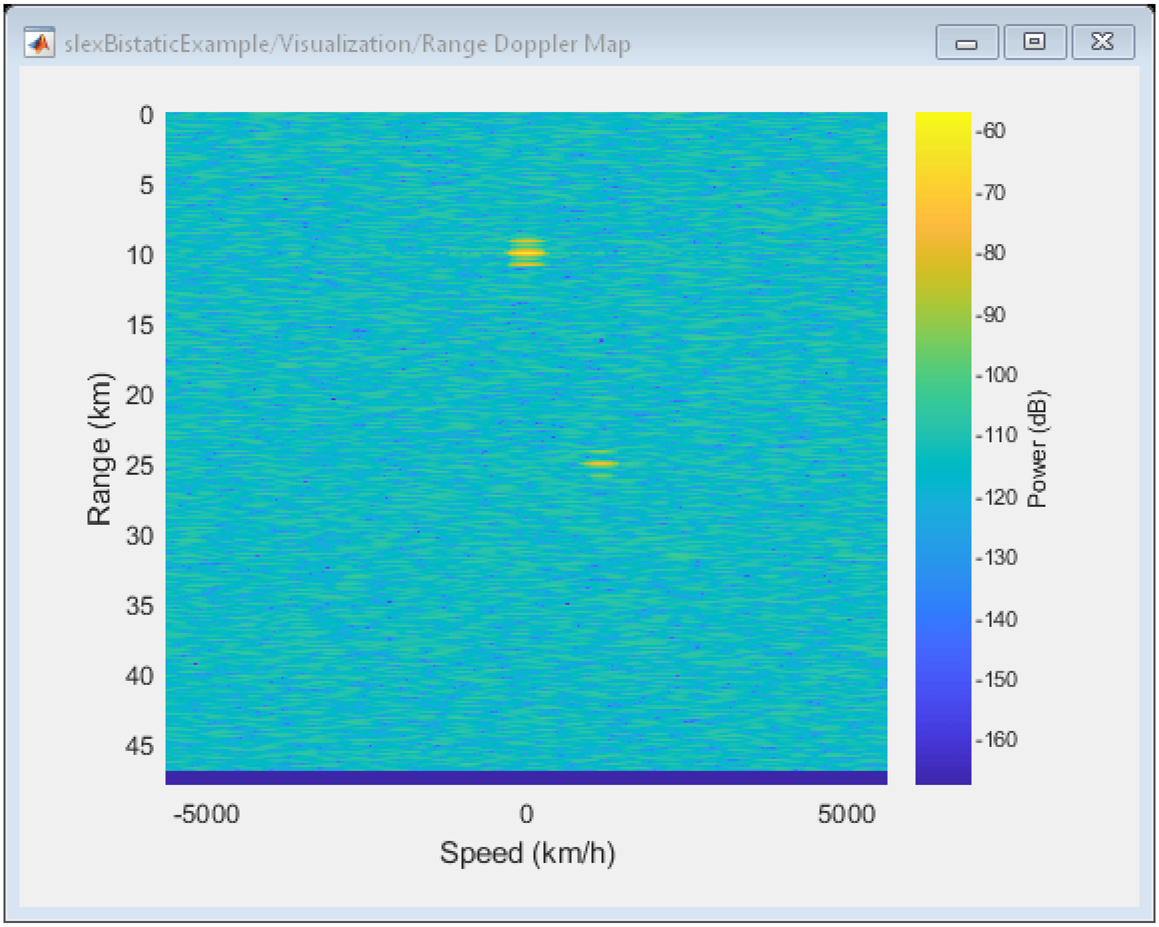


Exploring the Model

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.



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 whos 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.

Summary

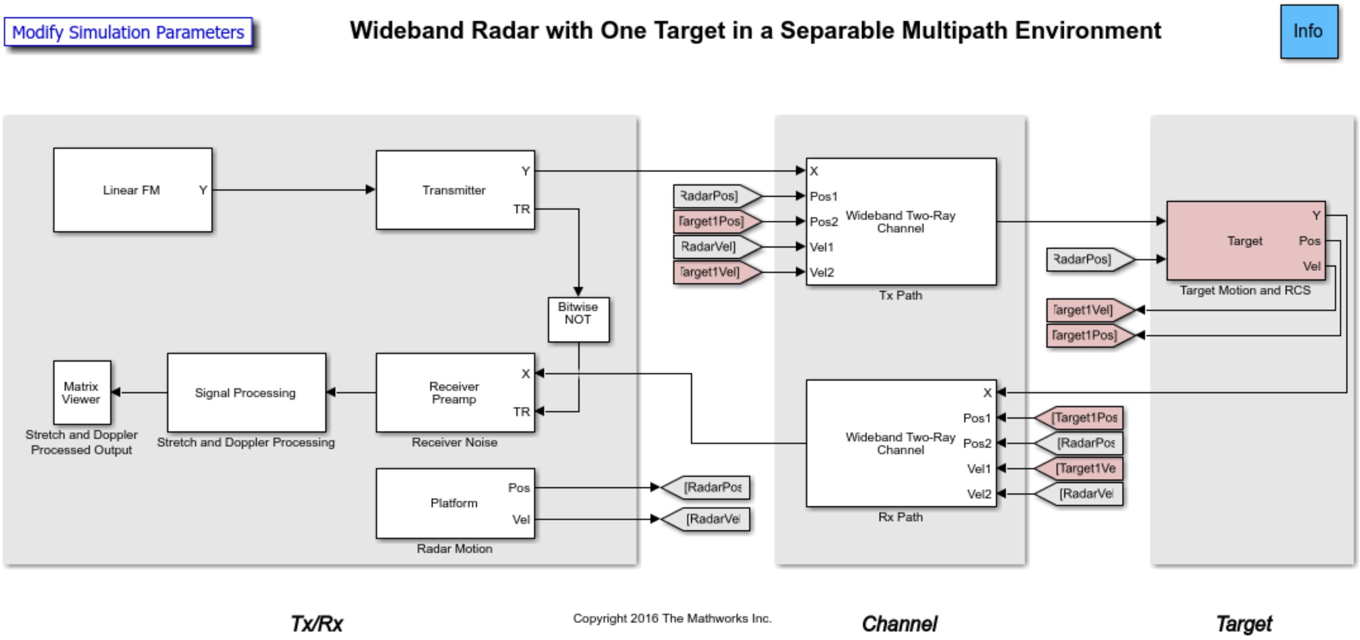
This example shows an end-to-end bistatic radar system simulation with two targets. It explains how to analyze the target return by plotting a range-Doppler map.

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.



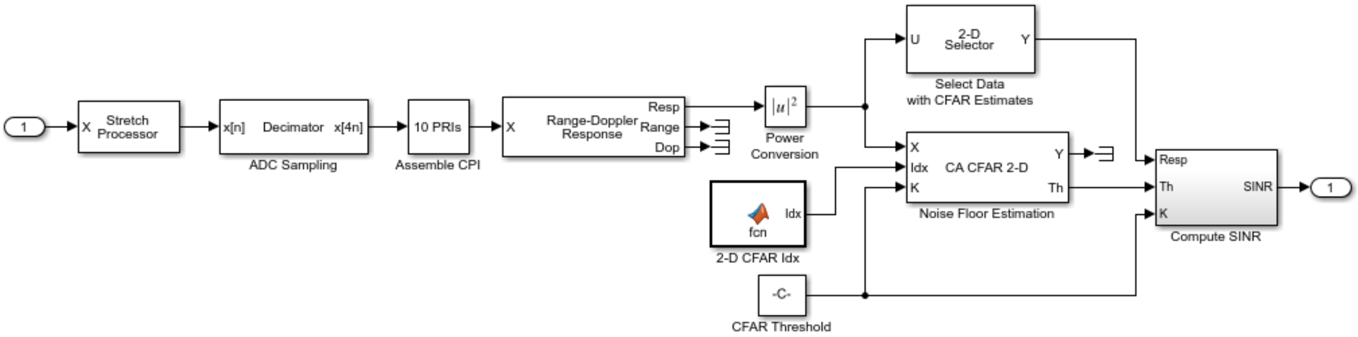
The model consists of a transceiver, a channel, and a target. The blocks that correspond to each section of the model are:

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.
- `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



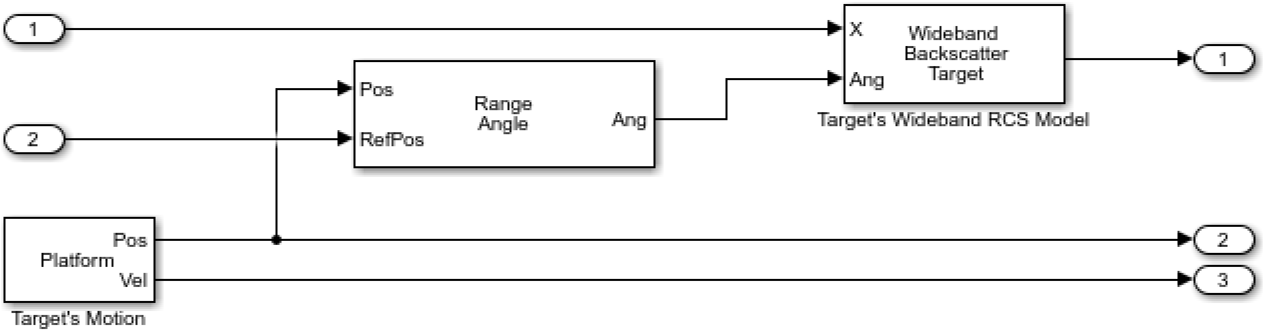
- `Stretch Processor` - Dechirps the received signal by mixing it in the analog domain with the transmitted linear FM waveform delayed to a selected reference range. A more detailed discussion on stretch processing is available in the “FMCW Range Estimation” example.
- `Decimator` - Subsystem models 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.



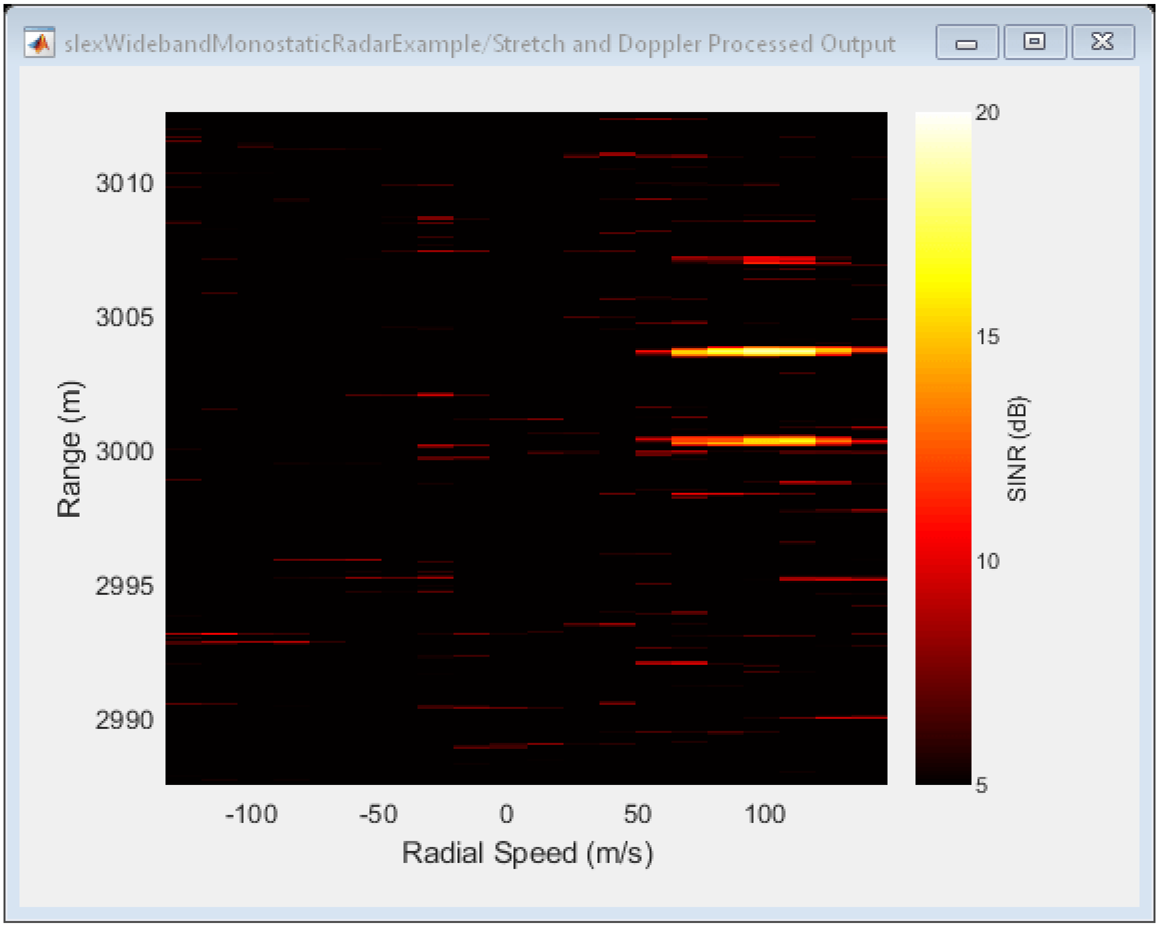
- `Platform` - Used to model the target's motion. The target's position and velocity values are used by the `Wideband Two-Ray Channel` blocks to model propagation and by the `Range Angle` block to compute the signal's incident angles at the target's location.
- `Range Angle` - Computes the propagated 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)
```

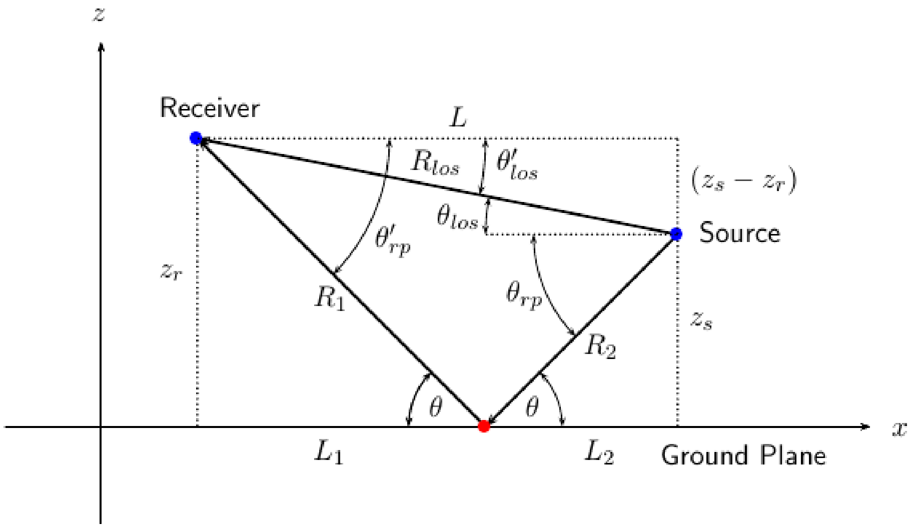
```
tgtRange =

        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. $\text{Radar} \xrightarrow{R_{rp}} \text{Target} \xrightarrow{R_{los}} \text{Radar}$
2. $\text{Radar} \xrightarrow{R_{los}} \text{Target} \xrightarrow{R_{rp}} \text{Radar}$

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 1-61 MATLAB® example.

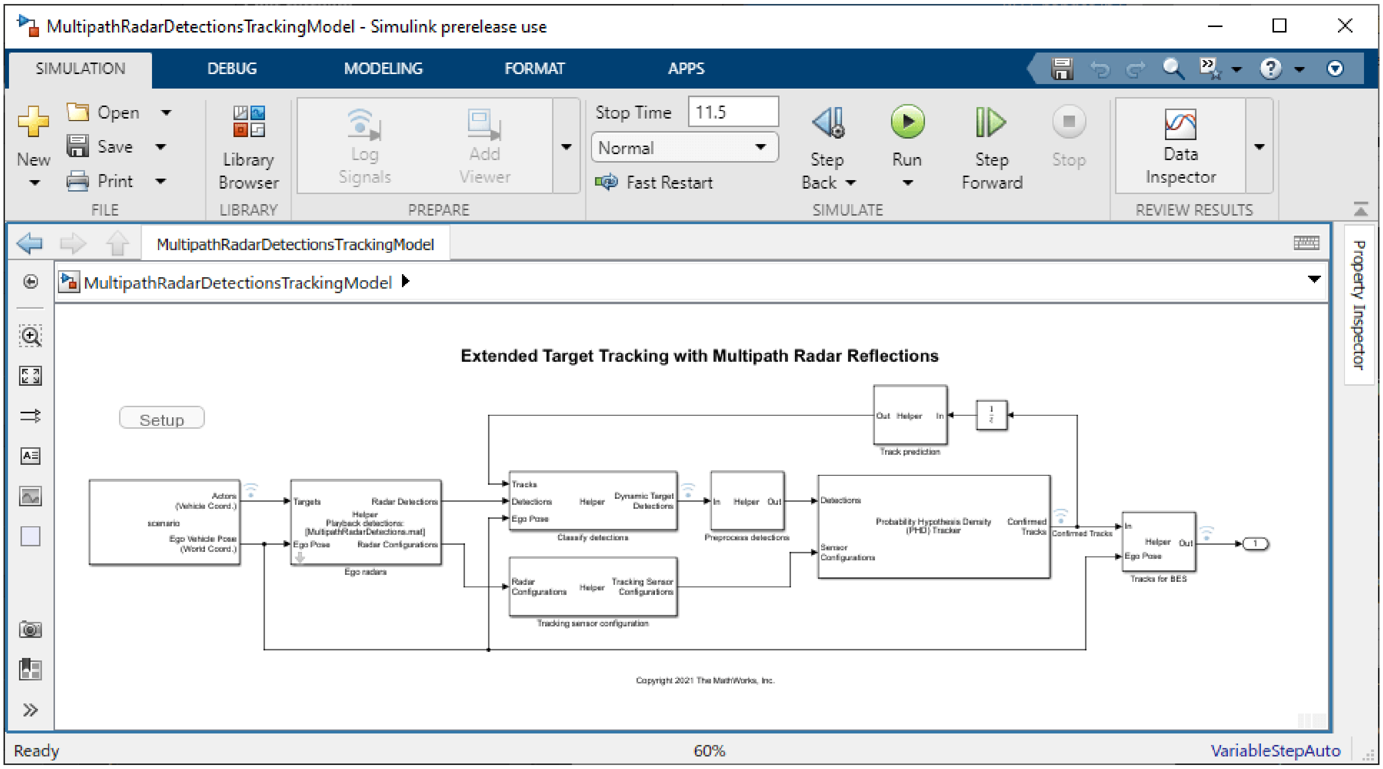
Introduction

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 multipath phenomenon and simulation of ghost detections, refer to the “Simulate Radar Ghosts Due to Multipath Return” on page 1-44 example.

Load Scenario and Radars

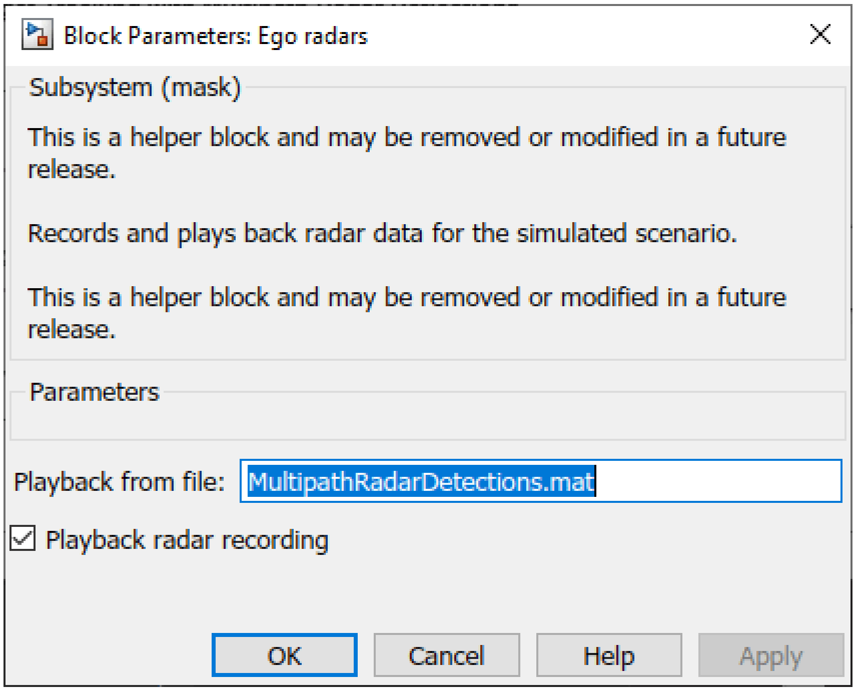
This example uses the same scenario and radars defined by the `helperCreateMultipathDrivingScenario` function used in the “Highway Vehicle Tracking with Multipath Radar Reflections” on page 1-61 example. Opening the model loads this scenario into the workspace for use by the Scenario Reader (Automated Driving Toolbox) block.

```
open_system('MultipathRadarDetectionsTrackingModel')
```



Use the "Ego radars" helper block to playback detections recorded from four radars providing full 360 degree coverage around the ego vehicle. To record a new set of detections, uncheck "Playback radar recording".

```
open_system('MultipathRadarDetectionsTrackingModel/Ego radars')
```


Block Parameters: Ego radars

Subsystem (mask)

This is a helper block and may be removed or modified in a future release.

Records and plays back radar data for the simulated scenario.

This is a helper block and may be removed or modified in a future release.

Parameters

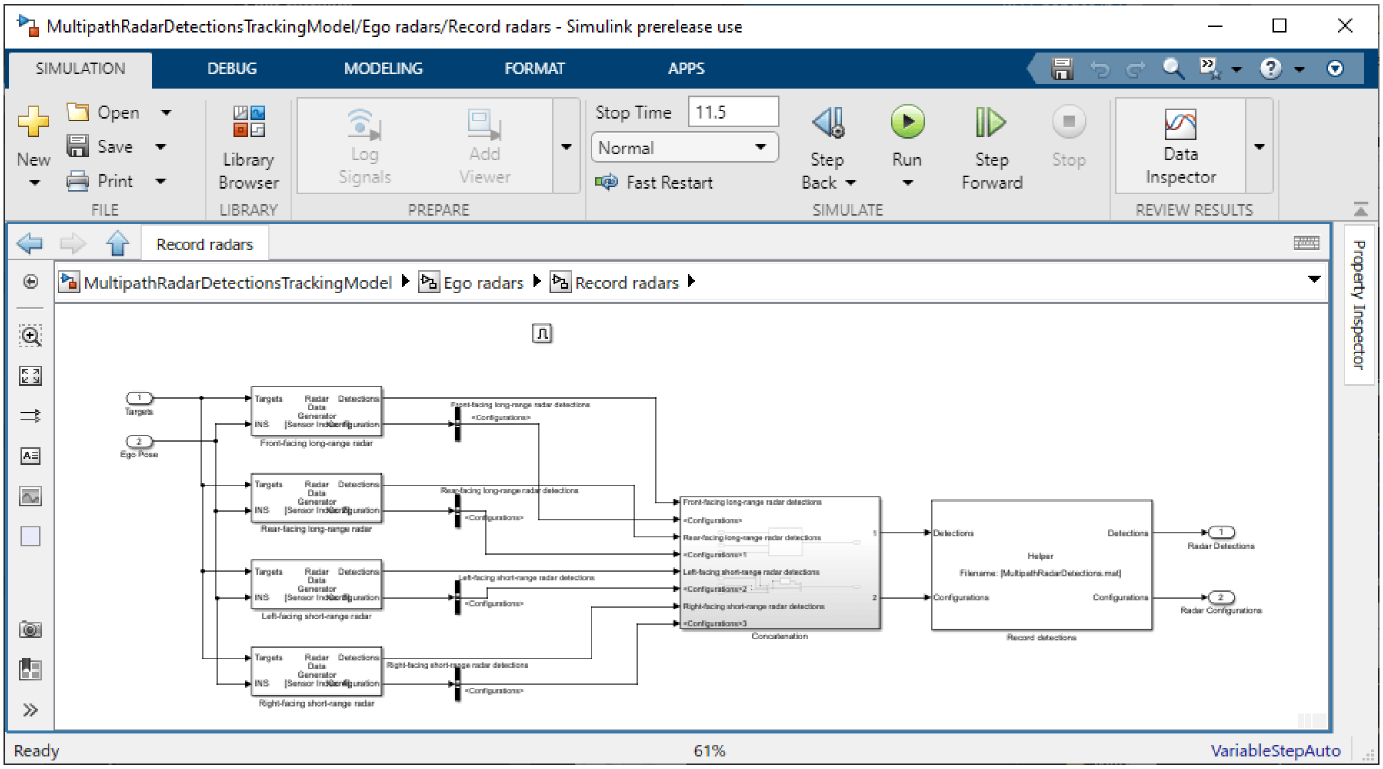
Playback from file: MultipathRadarDetections.mat

☑ Playback radar recording

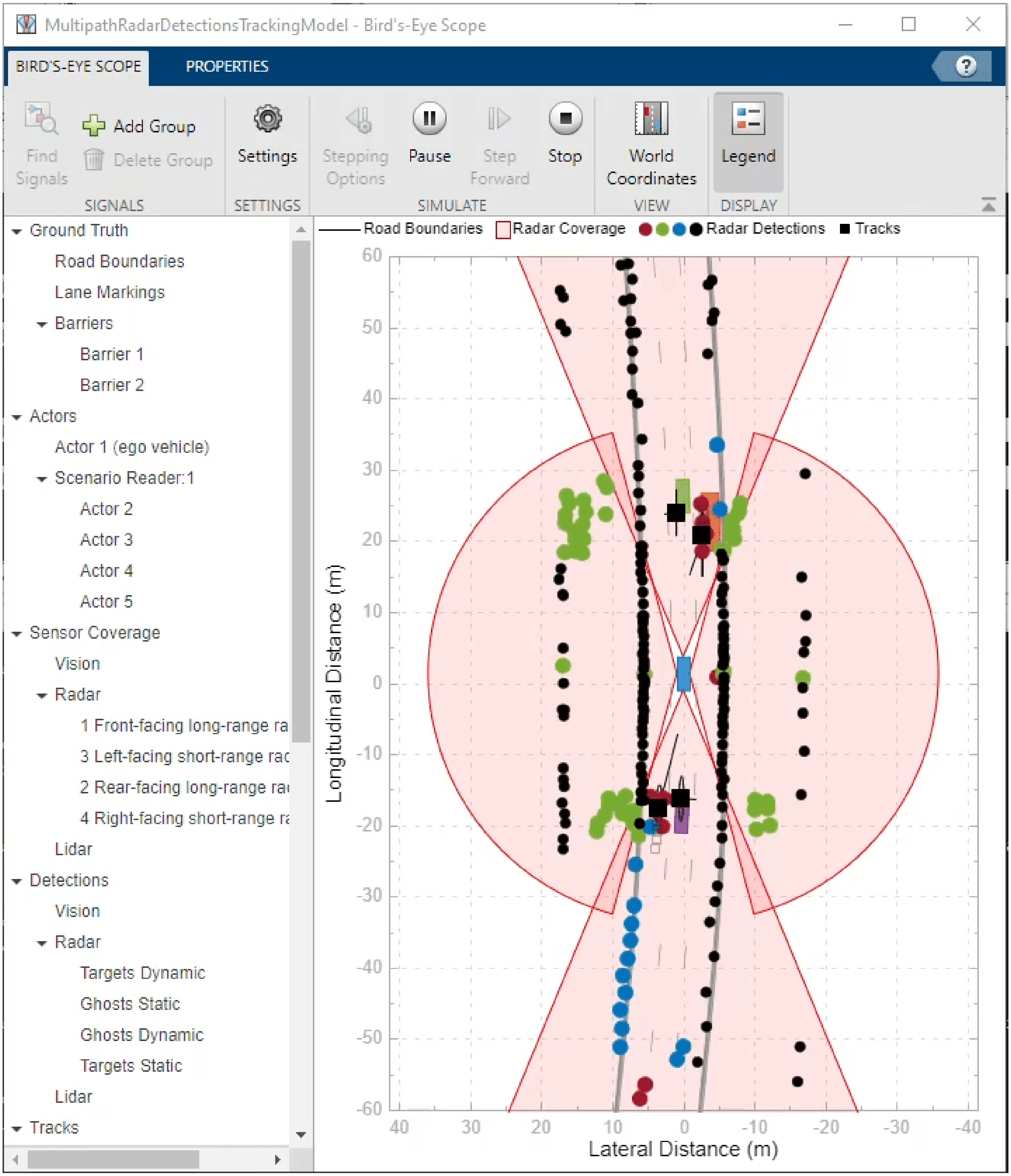
```
close_system('MultipathRadarDetectionsTrackingModel/Ego radars')
```

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.


MultipathRadarDetectionsTrackingModel - Bird's-Eye Scope
BIRD'S-EYE SCOPE
PROPERTIES
Find Signals
Add Group
Delete Group
SIGNALS
Settings
SETTINGS
Stepping Options
Pause
Step Forward
Stop
SIMULATE
World Coordinates
VIEW
Legend
DISPLAY
Ground Truth
Road Boundaries
Lane Markings
Barriers
Barrier 1
Barrier 2
Actors
Actor 1 (ego vehicle)
Scenario Reader:1
Actor 2
Actor 3
Actor 4
Actor 5
Sensor Coverage
Vision
Radar
1 Front-facing long-range ra
3 Left-facing short-range rac
2 Rear-facing long-range rac
4 Right-facing short-range ra
Lidar
Detections
Vision
Radar
Targets Dynamic
Ghosts Static
Ghosts Dynamic
Targets Static
Lidar
Tracks
Road Boundaries
Radar Coverage
Radar Detections
Tracks
60
50
40
30
20
10
0
-10
-20
-30
-40
-50
-60
Longitudinal Distance (m)
40
30
20
10
0
-10
-20
-30
-40
Lateral Distance (m)

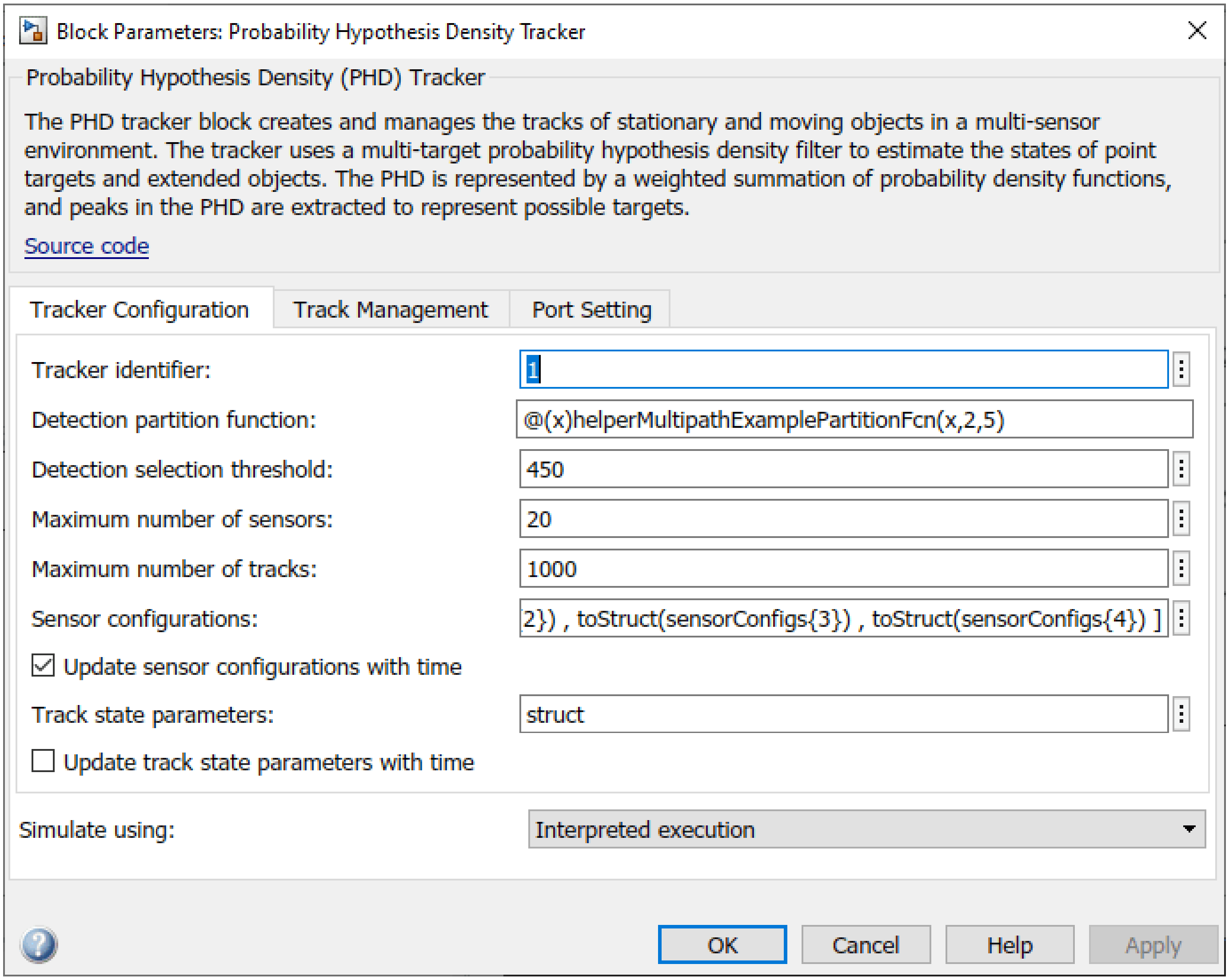
The "Classify detections" helper block classifies the detections generated by the four radars by comparing their measurements to the confirmed tracks from the Probability Hypothesis Density (PHD) Tracker (Sensor Fusion and Tracking Toolbox) block. The detection classification utilizes the measured radial 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 1-61 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 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

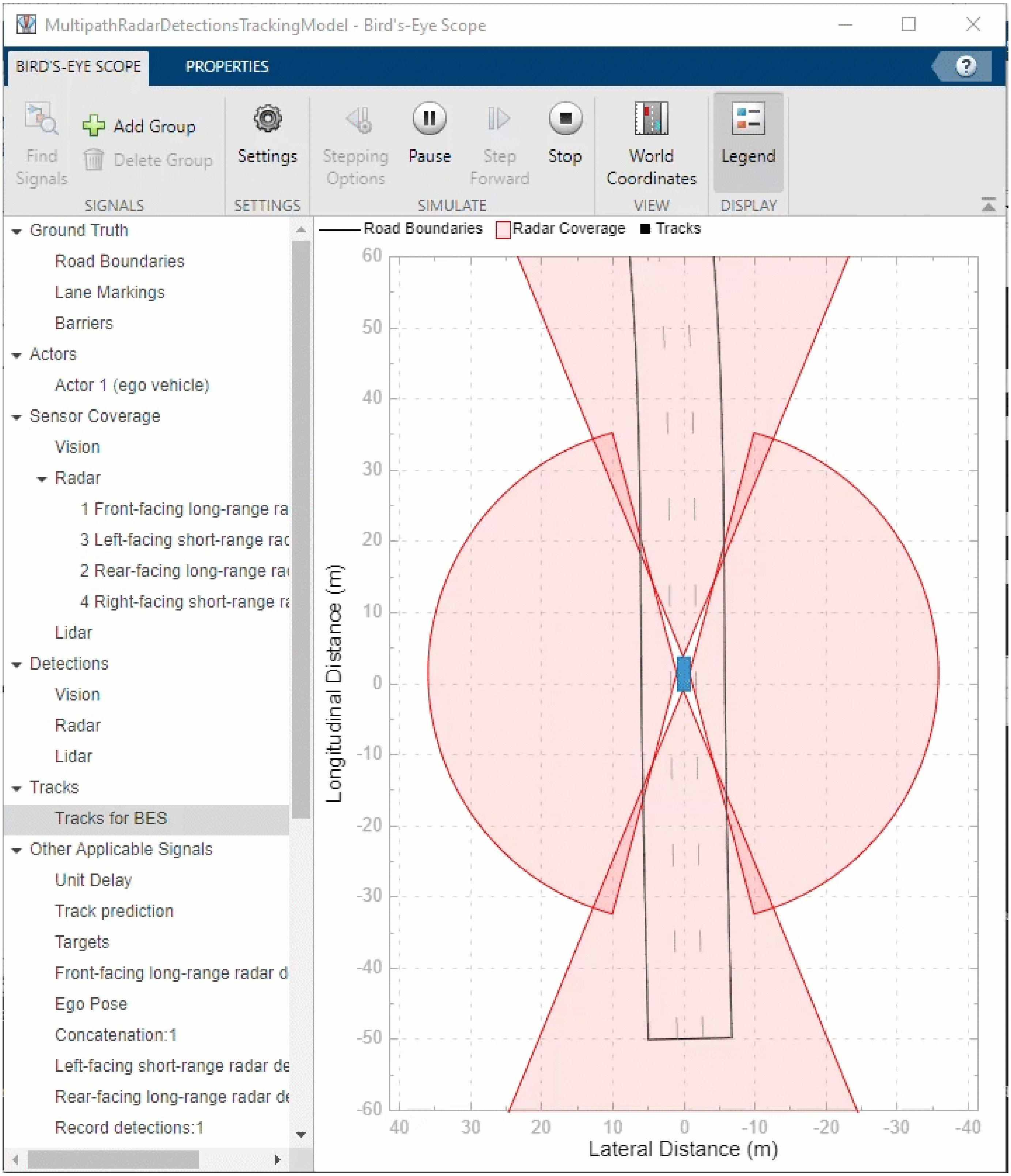
Simulate using: Interpreted execution

```
close_system('MultipathRadarDetectionsTrackingModel/Probability Hypothesis Density Tracker',0)
```

Run Simulation

Use the following command to playback the recorded detections and generate tracks.

```
simout = sim('MultipathRadarDetectionsTrackingModel')
```


MultipathRadarDetectionsTrackingModel - Bird's-Eye Scope
BIRD'S-EYE SCOPE
PROPERTIES
Find Signals
Add Group
Delete Group
SIGNALS
Settings
SETTINGS
Stepping Options
Pause
Step Forward
Stop
SIMULATE
World Coordinates
VIEW
Legend
DISPLAY
Ground Truth
Road Boundaries
Lane Markings
Barriers
Actors
Actor 1 (ego vehicle)
Sensor Coverage
Vision
Radar
1 Front-facing long-range ra
3 Left-facing short-range rac
2 Rear-facing long-range ra
4 Right-facing short-range ra
Lidar
Detections
Vision
Radar
Lidar
Tracks
Tracks for BES
Other Applicable Signals
Unit Delay
Track prediction
Targets
Front-facing long-range radar d
Ego Pose
Concatenation:1
Left-facing short-range radar de
Rear-facing long-range radar de
Record detections:1
Road Boundaries
Radar Coverage
Tracks
Longitudinal Distance (m)
60
50
40
30
20
10
0
-10
-20
-30
-40
-50
-60
40
30
20
10
0
-10
-20
-30
-40
Lateral Distance (m)

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.

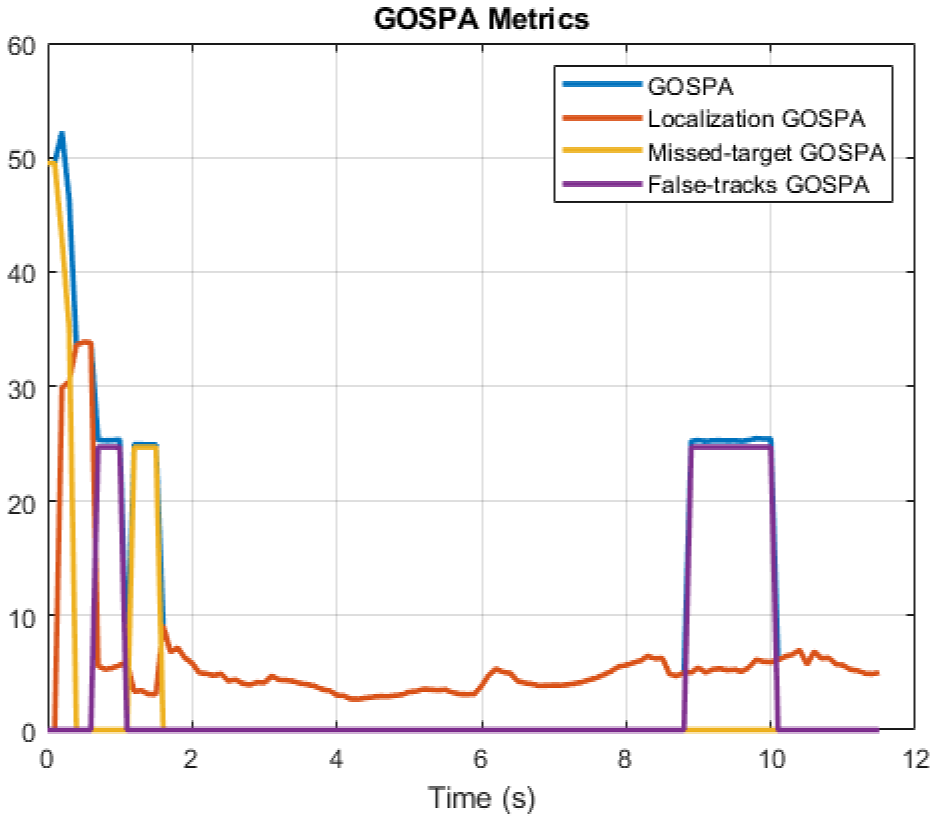
```
[confirmedTracks,confusionMatrix] = helperLoadSimulationLogs('MultipathRadarDetectionsTrackingMod
```

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,

    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 figure below, 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 3% of ghosts from static object reflections and 23% 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                1986        
    Ghost (S)              3244        
    Ghost (D)               836        
    Environment           27441        
    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         90.99        0.76         7.85           0.4          0   
    Ghost (S)        3.18       85.39        11.13          0.31          0   
    Ghost (D)       22.97        0.36        76.67             0          0   
    Environment      1.56        2.92         3.41          92.1          0   
    Clutter         19.42       65.47        14.39          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

A radar system's propagation path is modified by the presence of 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 may be increased by as much as 12 dB, whereas in a minimum the return can theoretically go to 0.

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 takes into account the interference between the direct and ground-reflected rays, as well as refraction. This example will define an L-band radar system in the presence of heavy clutter and will show you how to visualize its 3-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 (dBi)
```

Define the 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 below.

```
% 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);
```

Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey • Sources: Esri, HERE, Garmin, USGS, Intermap, INCREMENT P, NRCan, Esri Japan, METI, Esri China (Hong Kong), Esri Korea, Esri (Thailand), NGCC, (c) OpenStreetMap contributors, and the GIS User Community

Investigate the Free Space Range

For this example, the free space range `rfs` is the range at which a target would have a 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, we can use a pulse integration technique to reduce the required SNR. For this system, we 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 one 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 the Vertical Coverage

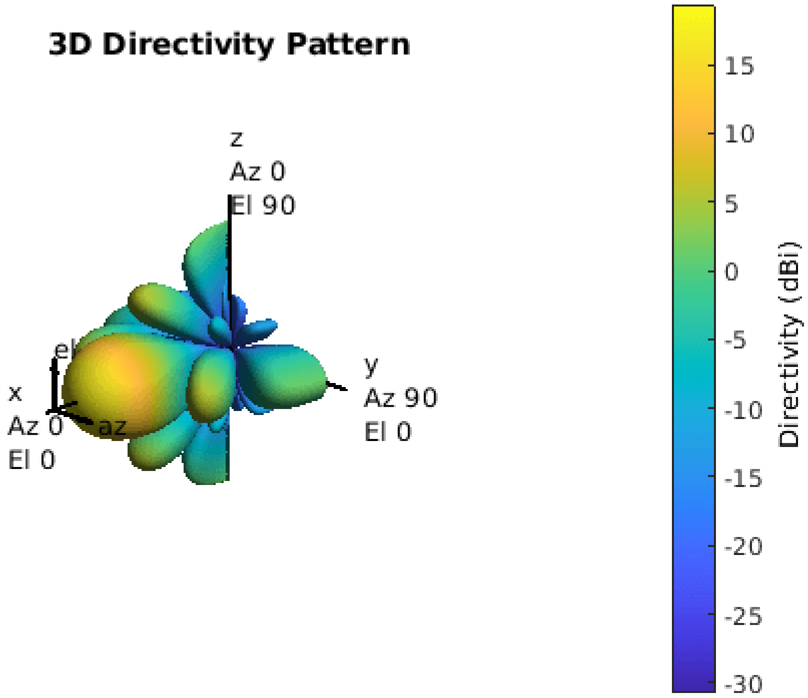
The vertical coverage pattern is visualized using a Blake chart, also known as a range-height-angle chart. The range along the *x*-axis is the propagated range and the height along the *y*-axis is relative to the origin of the ray.

The vertical coverage contour is calculated using the `radarvcd` function. The default permittivity model in `radarvcd` is based on a sea permittivity model in Blake's "Machine Plotting of Radar Vertical-Plane Coverage Diagrams." 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);
```



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);
```

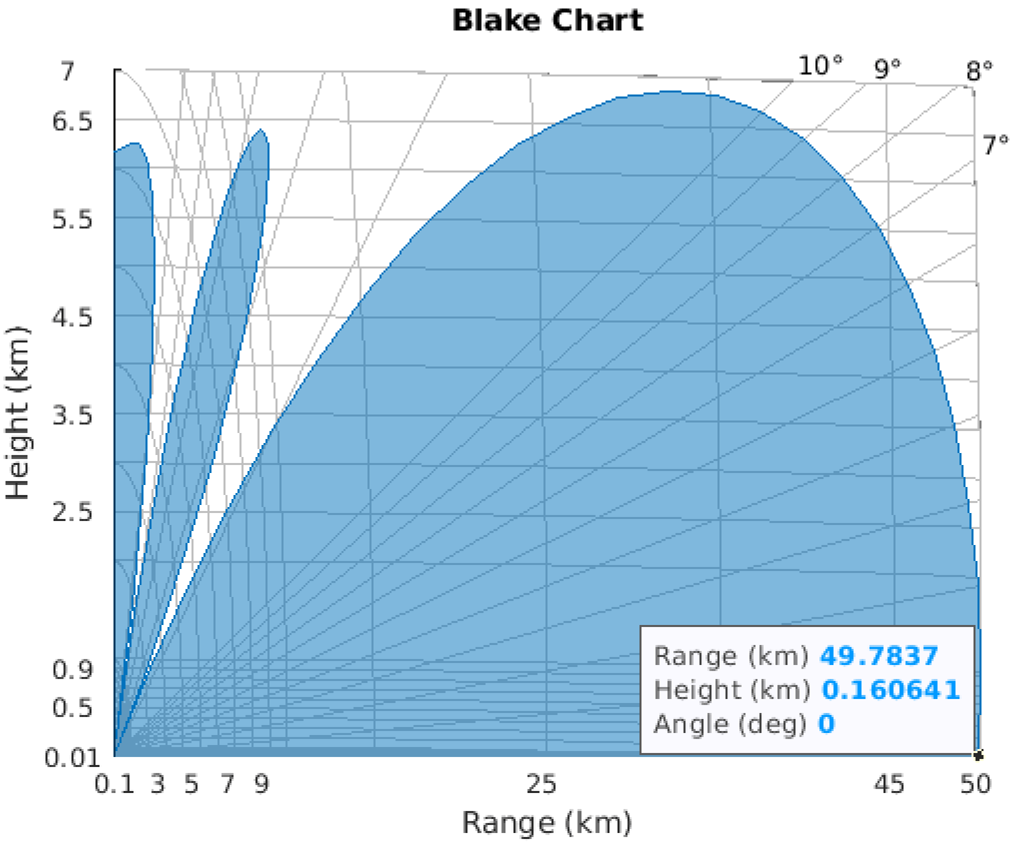
Use the surface height at the site to obtain the surface refractivity from `refractiveidx`.

```
% 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 approximates the refractivity profile as an exponential decay versus height, which has been shown to be an excellent approximation for a normal atmosphere (i.e., 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)
```



Visualize the 3-D Vertical Coverage over Terrain

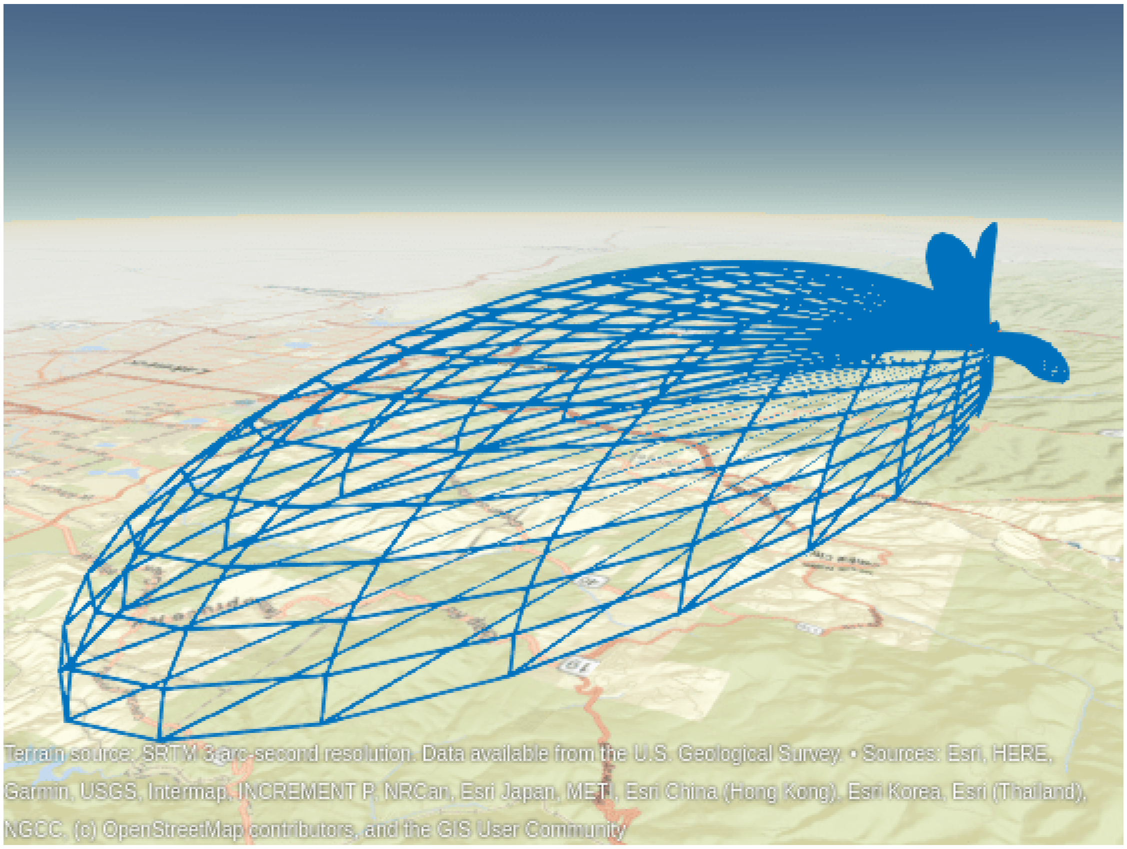
The next section calculates the vertical coverage at the specified azimuth intervals. The vertical coverage range and angle is converted to height using the `range2height` function using the `'CRPL'` method. The range-height-angle values are then converted to Cartesian.

```
% 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 will appear 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]);
```



Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. • Sources: Esri, HERE, Garmin, USGS, Intermap, INCREMENT P, NRCan, Esri Japan, METI, Esri China (Hong Kong), Esri Korea, Esri (Thailand), NGCC, (c) OpenStreetMap contributors, and the GIS User Community

Summary

This example discussed a method to calculate a 3-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, taking into account the interference from the surface reflection and refraction.

References

1. Barton, David K. *Radar Equations for Modern Radar*. Norwood, MA: Artech House, 2013.
2. Blake, Lamont V. "Radio Ray (Radar) Range-Height-Angle Charts." Naval Research Laboratory, NRL Report 6650, Jan. 22, 1968.
3. Blake, Lamont 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 Government Printing Office, 1959.

```
% 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
```

See Also

`blakechart` | `detectability` | `effearthradius` | `height2range` | `radareqsnr` | `radarvcd` | `range2height` | `refractiveidx` | `toccgh`

Related Examples

- “Modeling Target Position Estimation Errors” on page 1-654

Introduction to SAR Target Classification Using Deep Learning

This example shows creation and training of a simple Convolution Neural Network (CNN) to classify SAR targets using deep learning.

Deep learning is a powerful technique that can be used to train a robust classifier. It has shown its effectiveness in diverse areas ranging from image analysis to natural language processing. In general, these developments have huge potential for SAR data analysis and processing. A major objective for SAR imaging has been object detection and classification, which is called Automatic Target Recognition (ATR). Here, a simple CNN is used 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 dataset.
- Load and analyze image data.
- Splitting and augmentation of the data.
- Defining the network architecture.
- Training the network.
- Predicting the labels of new data and calculate the classification accuracy.

To illustrate this workflow, Moving and Stationary Target Acquisition and Recognition (MSTAR) Mixed Targets dataset published by the Air Force Research Laboratory [1 on page 1-0] is used. The dataset is available for download here. Our goal is to develop a model to classify ground targets based on SAR imagery.

Download Dataset

This example uses MSTAR target dataset containing 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 used are BMP2 (Infantry Fighting Vehicle), BTR70 (armored car), and T72 (tank). The images were captured at two different depression angles of 15 degrees and 17 degrees with 190 ~ 300 different aspect versions. These are full aspect coverage over 360 degrees.

Download the dataset from the given URL using the `helperDownloadMSTARTargetData` helper function. The size of data set is 28 MB.

```
outputFolder = pwd;
dataURL = ['https://ssd.mathworks.com/supportfiles/radar/data/' ...
    'MSTAR_TargetData.tar.gz'];
helperDownloadMSTARTargetData(outputFolder,dataURL);
```

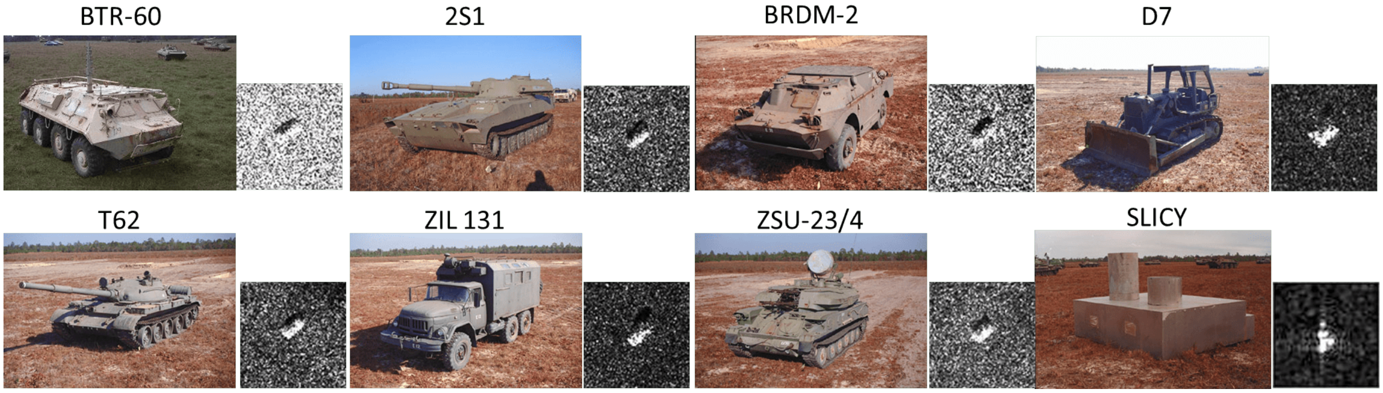
Depending on your Internet connection, the download process can take some time. The code suspends MATLAB® execution until the download process is complete. Alternatively, you can download the data set to your local disk using your web browser and extract the file. If you do so, change the outputFolder 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 batch of images during training of a CNN.

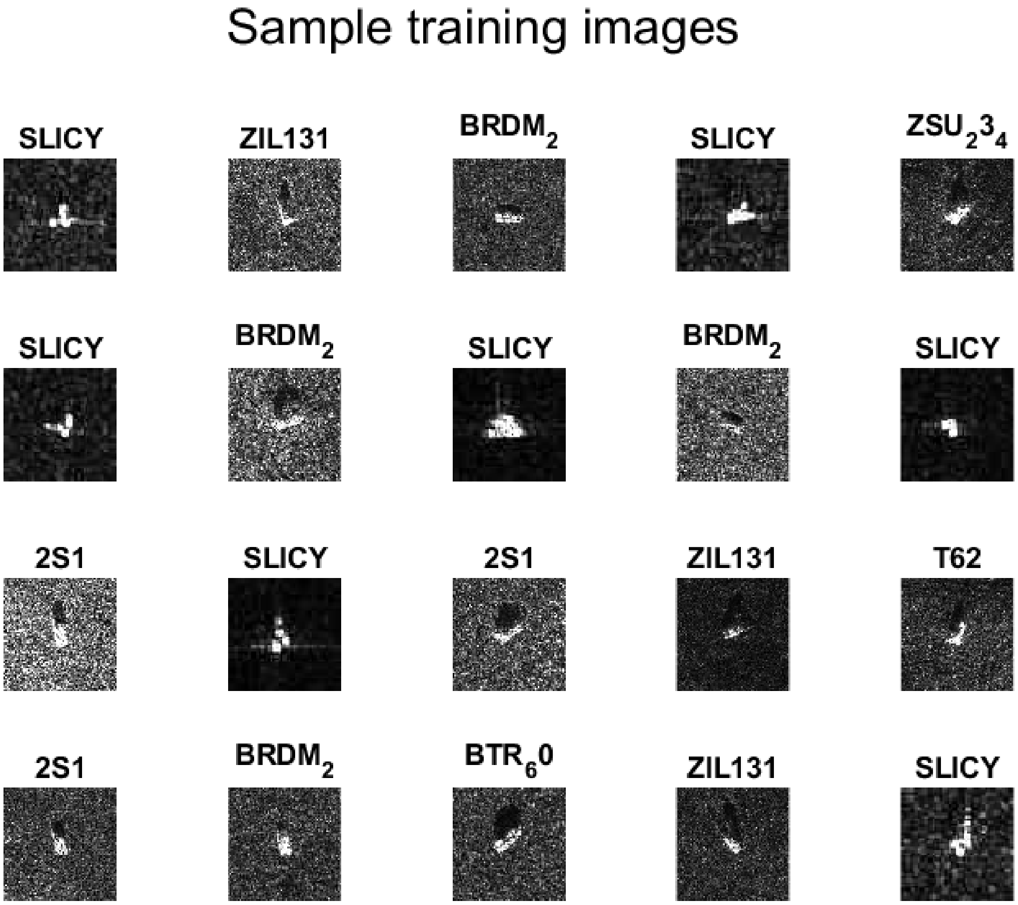
```
sarDatasetPath = fullfile(pwd,'Data');
imds = imageDatastore(sarDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');
```

The MSTAR dataset contains sensor returns from 7 ground vehicles and a calibration target. Optical images and SAR images for these 8 targets are shown below



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
```



The `imds` variable 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 incured in training.

```
imgSize = [128,128,1];
```

Create Datastore Object for Training, Validation and Testing

Divide the data into training, validation and test sets. Here, 80% of dataset for training, 10% for model validation during training is used apart from 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 the network, the image size must match the size of the network's input layer. Instead of resizing the images manually, 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 the data.

```
auimdsTrain = augmentedImageDatastore(imgSize, imdsTrain);
auimdsValidation = augmentedImageDatastore(imgSize, imdsValidation);
auimdsTest = augmentedImageDatastore(imgSize, imdsTest);
```

Define Network Architecture

Define the CNN architecture using the `createNetwork` helper function.

```
layers = createNetwork(imgSize);
```

Train Network

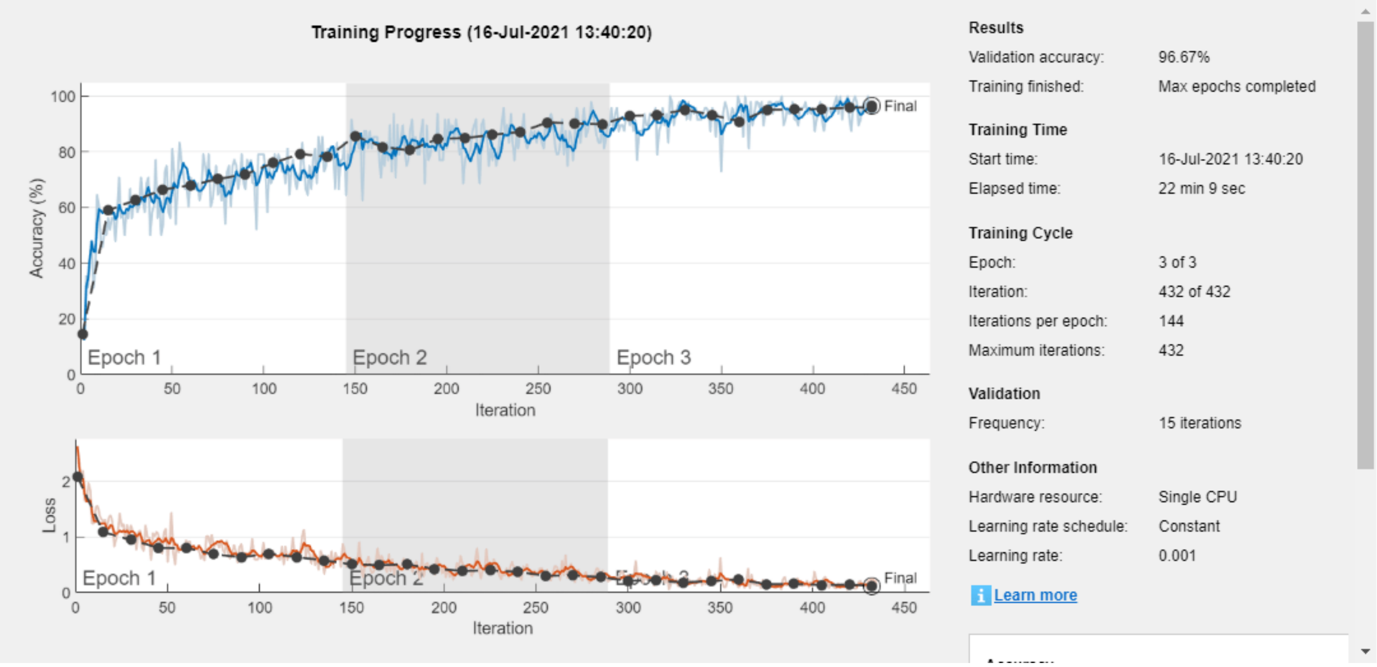
After defining the network architecture, use `trainingOptions` to specify the training options. Train the network using stochastic gradient descent with momentum (SGDM) with an initial learning rate of 0.001. Set the maximum number of epochs to 3. An epoch is a full training cycle on the entire training data set. Monitor the network accuracy during training by specifying validation data and validation frequency. Shuffle the data every epoch. The network is trained on the training data and calculates the accuracy at regular intervals during training. The validation data is not used to update the network weights. Set 'CheckpointPath' to a temporary location.

```
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, accuracy and the validation loss with accuracy. For more information on the training progress plot, see Monitor Deep Learning Training Progress. The accuracy is the percentage of images that the network classifies correctly.

```
net = trainNetwork(auimdsTrain,layers,options);
```



Results

Validation accuracy: 96.67%

Training finished: Max epochs completed

Training Time

Start time: 16-Jul-2021 13:40:20

Elapsed time: 22 min 9 sec

Training Cycle

Epoch: 3 of 3

Iteration: 432 of 432

Iterations per epoch: 144

Maximum iterations: 432

Validation

Frequency: 15 iterations

Other Information

Hardware resource: Single CPU

Learning rate schedule: Constant

Learning rate: 0.001

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% for an eight-class classifier. Furthermore, note that the validation accuracy and training accuracy are similar. When the training accuracy is much higher than the validation accuracy, the model is overfitting (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.

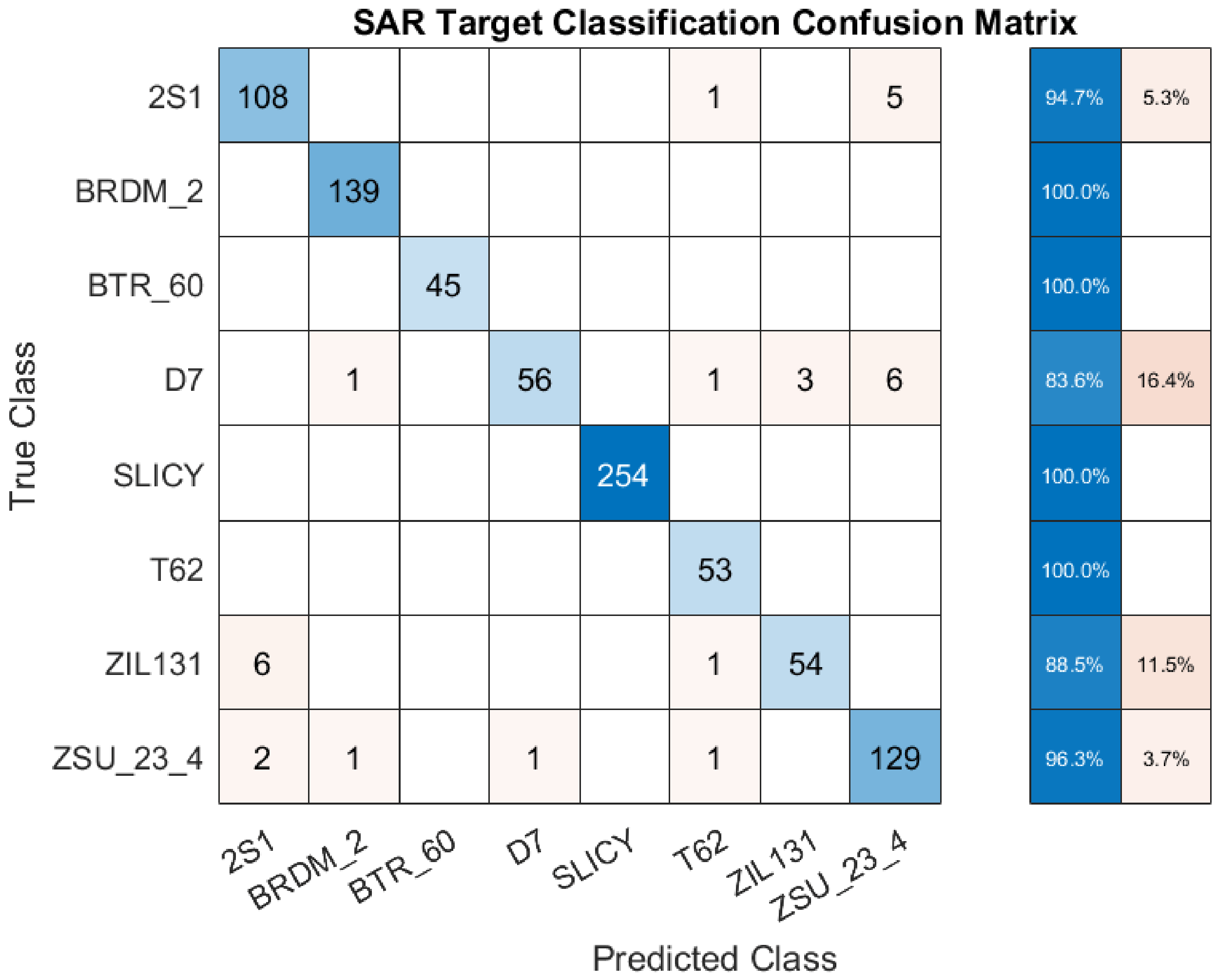
```
YPred = classify(net,auimdsTest);
YTest = imdsTest.Labels;

accuracy = sum(YPred == YTest)/numel(YTest)
```

```
accuracy = 0.9666
```

The test accuracy is very close to the validation accuracy, giving confidence in the model's predictive ability.

Use a confusion matrix to study the model's classification behavior in greater detail. A strong center diagonal represents accurate predictions. Ideally, small randomly located values off the diagonal is expected. 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

	2S1	BRDM_2	BTR_60	D7	SLICY	T62	ZIL131	ZSU_23_4		
2S1	108					1		5	94.7%	5.3%
BRDM_2		139							100.0%	
BTR_60			45						100.0%	
D7		1		56		1	3	6	83.6%	16.4%
SLICY					254				100.0%	
T62						53			100.0%	
ZIL131	6					1	54		88.5%	11.5%
ZSU_23_4	2	1		1		1		129	96.3%	3.7%

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 some misclassification by the trained model is observed. However, it is still able to achieve greater than 90% accuracy for the class.

Summary

This example demonstrates how to create and train a CNN to classify SAR targets obtained from the MSTAR database. The trained network attained an accuracy of 96.7% overall and 90% for ZSU-23/4 target 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 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.

Batch Normalization Layer Batch normalization layer normalizes the activation and gradient 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 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 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. 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 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 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.

```
function layers = createNetwork(imgSize)
    layers = [
        imageInputLayer([imgSize(1) imgSize(2) 1])      % Input Layer
        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(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 Networks (R-CNN) for target recognition in large scene Synthetic Aperture Radar (SAR) images using the Deep Learning Toolbox™ and Parallel Computing Toolbox™.

The Deep Learning Toolbox provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and apps.

The 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. It has 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 provide effective and efficient performance solution that scales to large scene SAR images as well.

This example demonstrates how to:

- Download dataset and pretrained model.
- Load and analyze image data.
- Define the network architecture.
- Specify training options.
- Train the network.
- Evaluation of network.

To illustrate this workflow, Moving and Stationary Target Acquisition and Recognition (MSTAR) clutter dataset published by the Air Force Research Laboratory is utilised. The dataset is available for download here. Alternatively, a subset of the data used to showcase the workflow is provided. The goal is to develop a model that can detect and recognize the targets.

Download Dataset

This example uses a subset of the MSTAR clutter dataset that contains 300 training and 50 testing clutter images with 5 different targets. The data was collected using an X-band sensor in spotlight mode, with a 1-foot resolution. The data contains rural and urban types of clutters. The type of target 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 PNG image format and the corresponding ground truth data is stored in `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 respectively. The size of the dataset is 1.6 GB.

Download the dataset from the given URL 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 local disk using web browser and extract the file. When using the alternative approach, change the outputFolder variable in the example to the location of the downloaded file.

Download Pretrained Network

Download the pretrained network from the given URL 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 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 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 location 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");
```

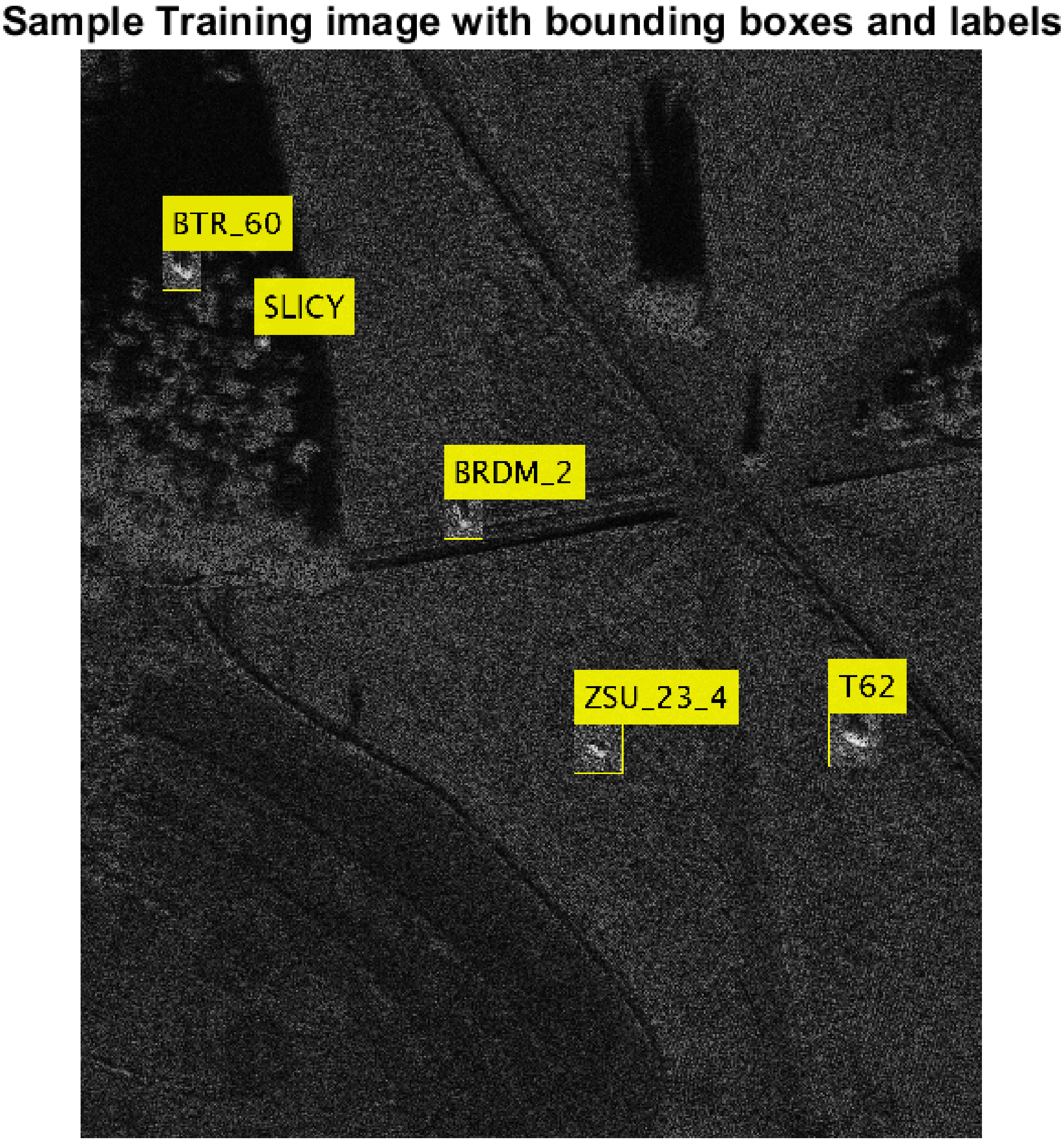
The ground truth data is stored in a six-column table, where the first column contains the image file paths and the second to the sixth column contains the different target bounding boxes.

```
% 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')
```



Define 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 5 targets specified above and a background class in order to be trained using `trainRCNNObjectDetector` 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`.

```
% 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 5-class object detector.

Train Faster R-CNN

Use `trainingOptions` 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 pair argument of `trainingOptions`. To automatically detect 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`'.

```
% 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,

```
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
```

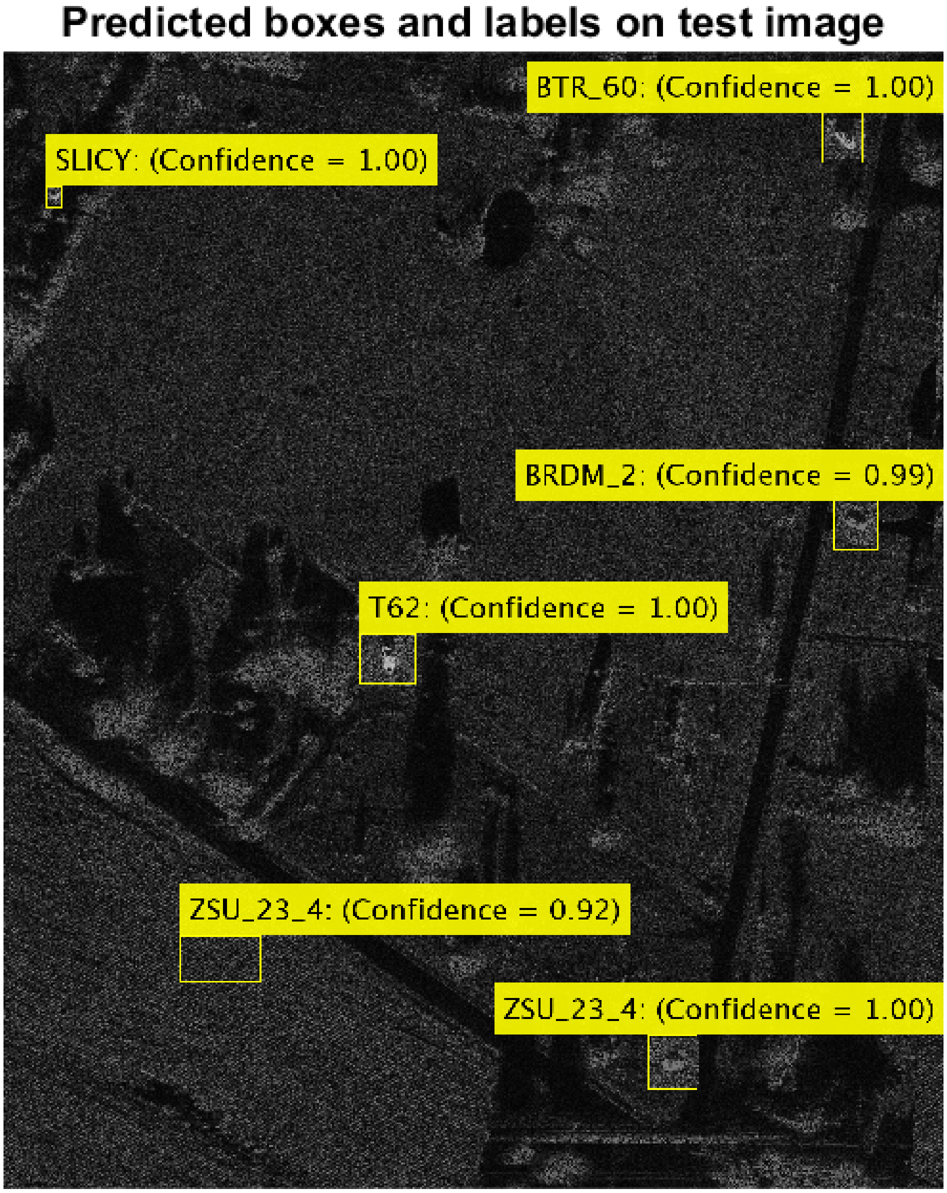
Evaluate Detector on a Test Image

To get a qualitative idea of the functioning of 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 detector's 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 will also result 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
BTR_60: (Confidence = 1.00)
SLICY: (Confidence = 1.00)
BRDM_2: (Confidence = 0.99)
T62: (Confidence = 1.00)
ZSU_23_4: (Confidence = 0.92)
ZSU_23_4: (Confidence = 1.00)

Evaluate Model

By looking at the images sequentially, the detector performance can be understood. 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)

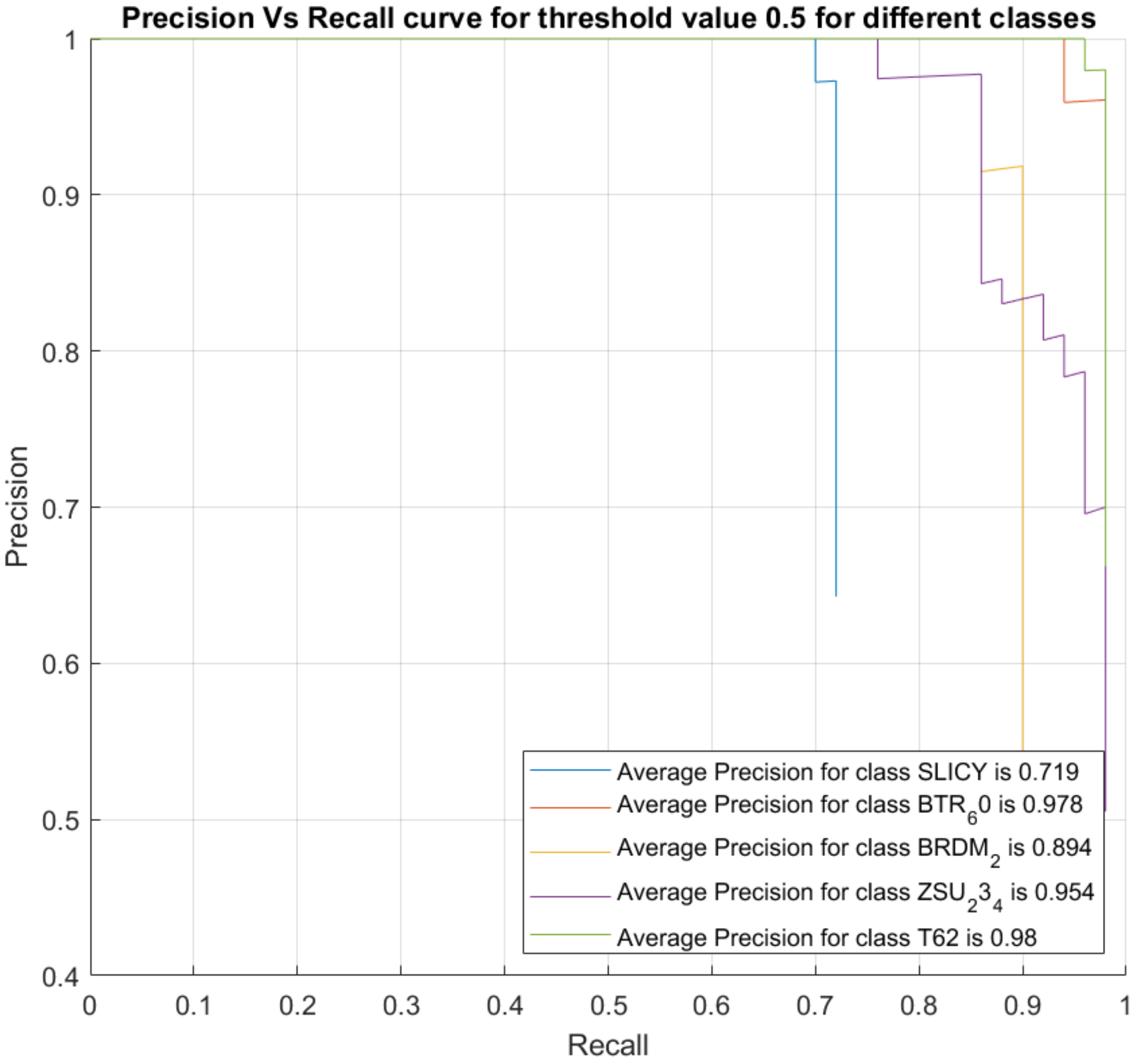
A detector with a precision of 1 is considered good at detecting targets that are present while a detector with a recall of 1 is good at avoiding false detections. Precision and recall have an inverse relationship.

Plot the relationship between precision and recall for each class. The average value of each curve is the AP. Curves for 0.5 detection thresholds are plotted.

For more details, see the documentation for `evaluateDetectionPrecision`.

```
% 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.Pr
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 'SLICY' targets. However, it is still able to achieve AP of 0.7 for the class.

Summary

This example demonstrates how to train a R-CNN for target recognition in SAR images. The pretrained network attained an accuracy of AP of more than 0.9.

Helper Function

The function `createNetwork` takes as input the image size `inputSize` and number of classes `numClassesPlusBackground`. The function returns a convolution neural network architecture.

```
function layers = createNetwork(inputSize,numClassesPlusBackground)
    layers = [
        imageInputLayer(inputSize)                      % Input Layer
        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
```

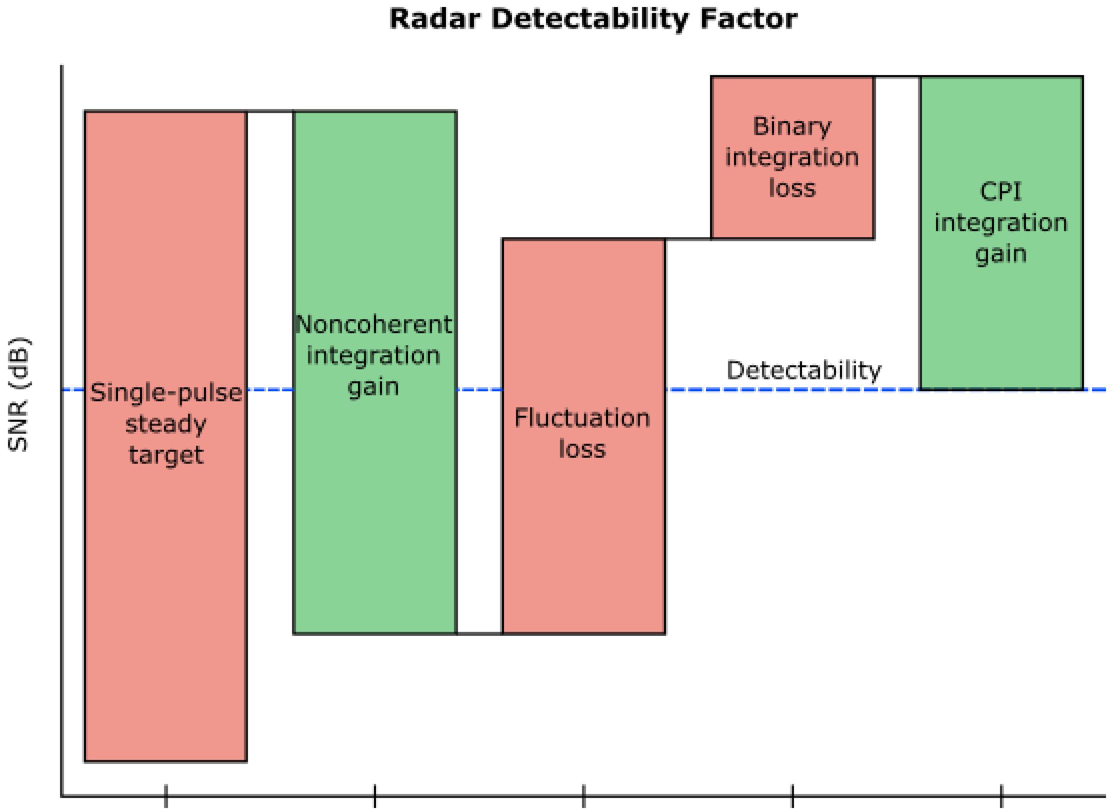
References

[1] MSTAR Dataset. 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 1-372 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.


Radar Detectability Factor
SNR (dB)
Single-pulse steady target
Noncoherent integration gain
Fluctuation loss
Binary integration loss
CPI integration gain
Detectability

This example illustrates how to compute the pulse integration loss for several pulse integration techniques. It also demonstrates computation of the losses due to the target's RCS fluctuation.

Pulse Integration

Typically, in 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 the coherent 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}. Let's examine how sensitive is the noncoherent integration gain to these parameters and compare it to the coherent integration gain.

We will 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].

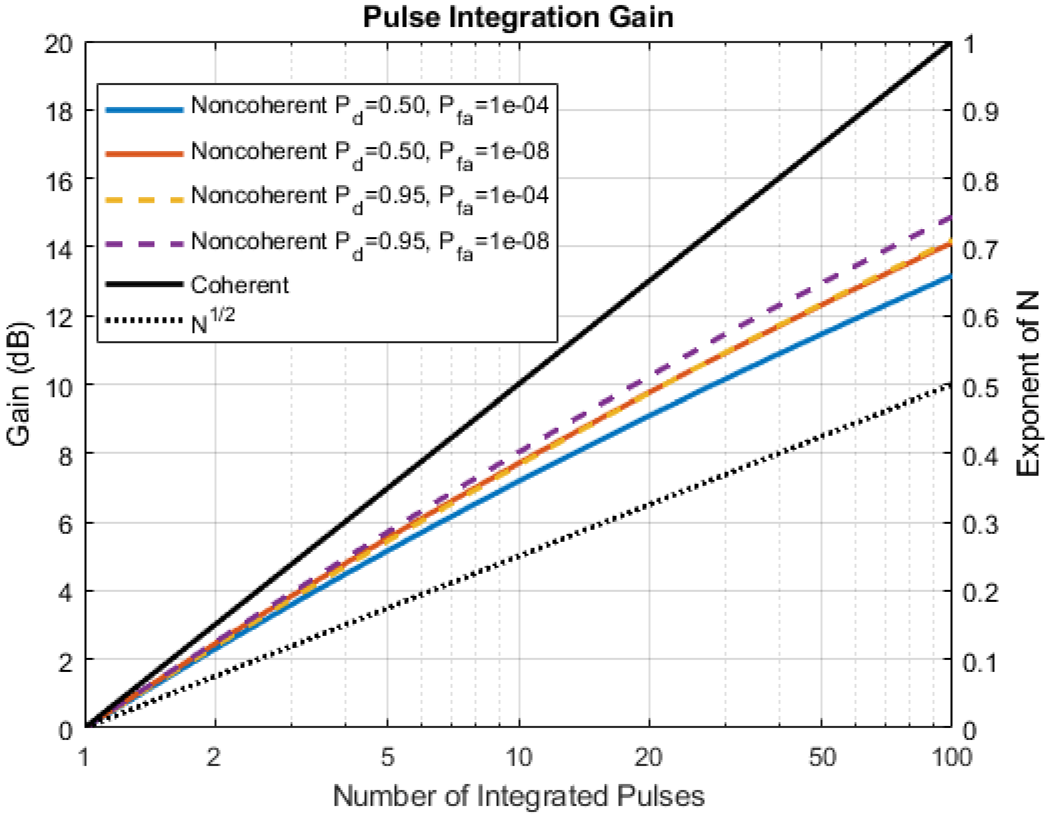
```
% Single-pulse detectability factor
D0 = detectability(Pd,Pfa,1);

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 we 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 example, if $P_d = 0.95, P_{fa} = 10^{-8}$, $N = 4$, and $M = 2$, the probabilities of detection and false alarm for each individual detection must be set to

```
[~,pd,pfa] = binaryintloss(0.95,1e-8,4,2)
```

```
pd = 0.7514
```

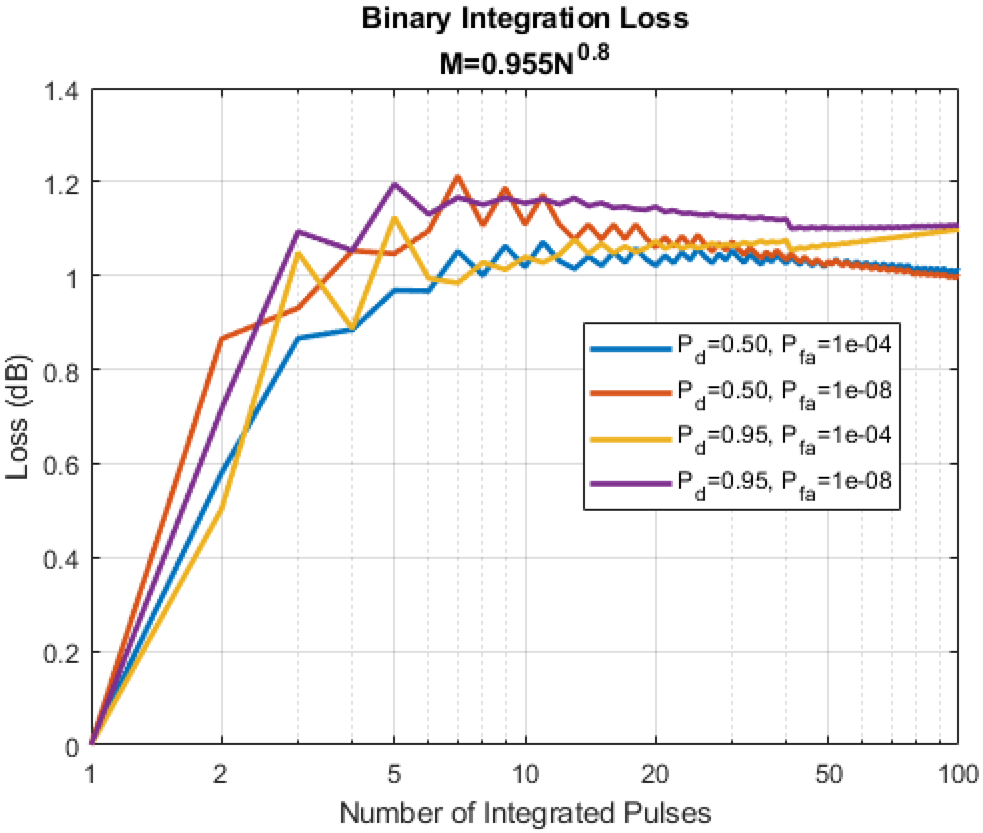
```
pfa = 4.0826e-05
```

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 non-fluctuating target a good choice of M was shown to be $M = 0.955N^{0.8}$ [3].

The binary integration loss as a function of N for several P_d and P_{fa} is shown below.

```
% 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 non-coherently 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 below as a function of the total number of pulses, N, for several choices of the M-of-N threshold.

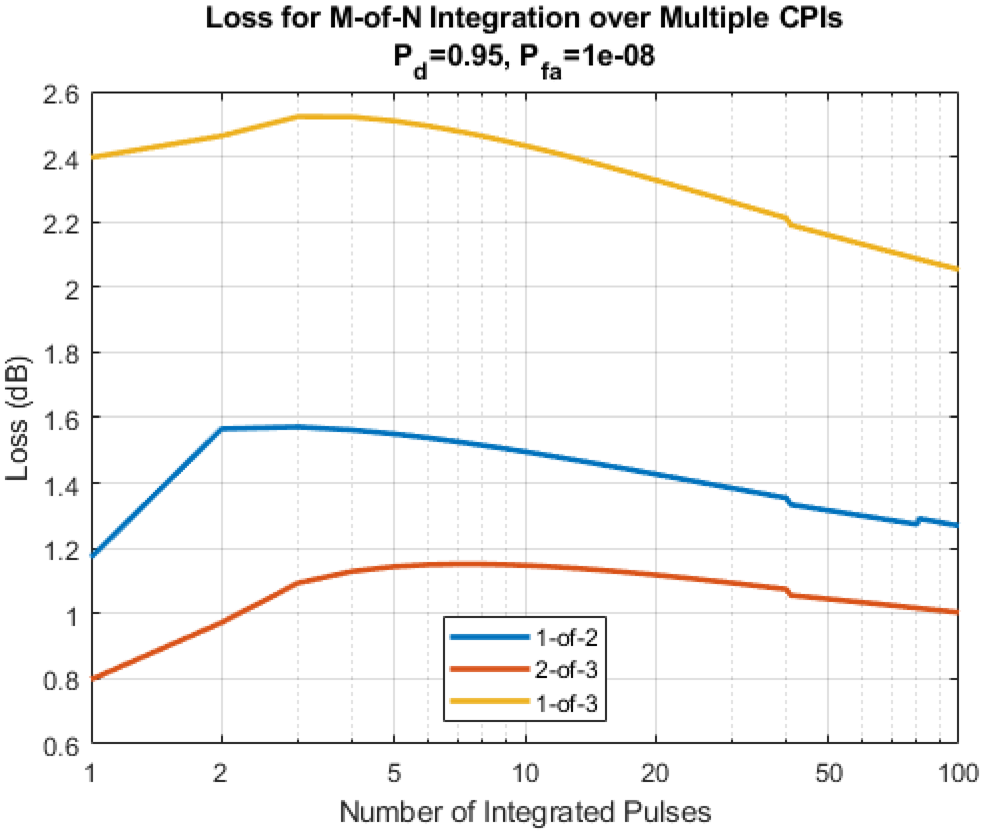
```
% M-of-N detection thresholds: 1-of-2, 2-of-3, and 1-of-3
n = [2 3 3];
m = [1 2 1];

% 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. Below we 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 $P_{fa} = 10^{-8}$.

The detectability factor for the coherent case is computed assuming a single pulse is detected in the envelope detector after integrating *N* pulses coherently.

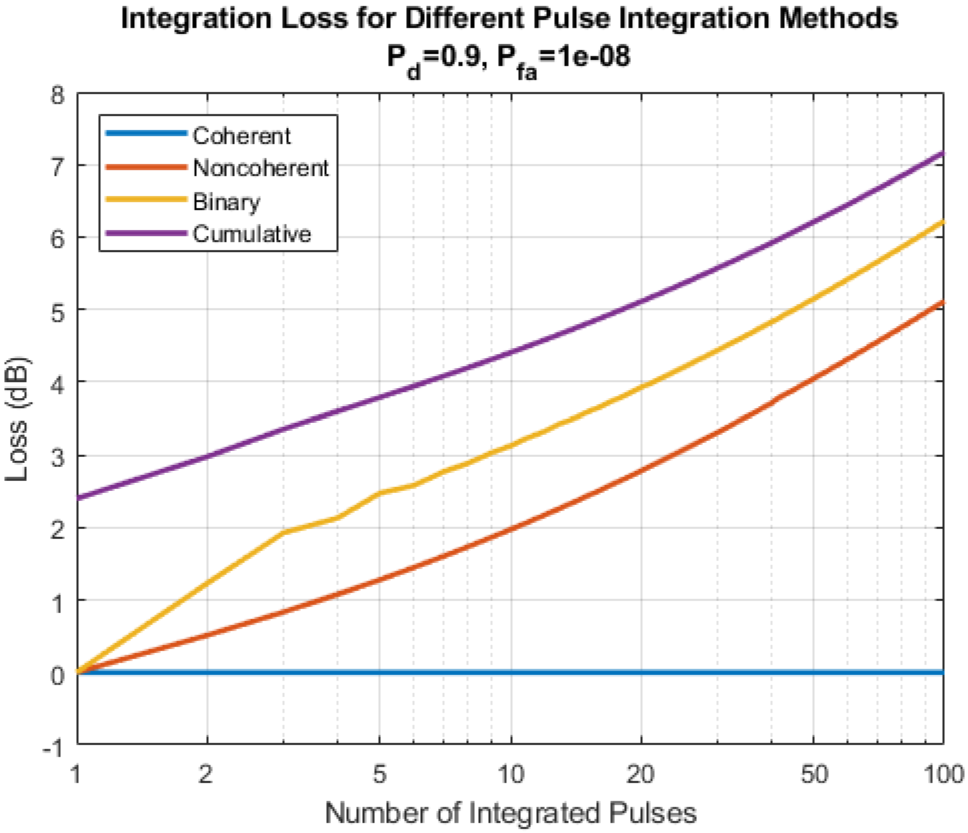
```
% Detectability factor for coherent integration
Dc = detectability(Pd(2),Pfa(2),1) - 10*log10(N);
```

Then the losses for different integration types with respect to coherent integration are

```
% 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,:);
```

The plot below compares 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. We will 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, $D_0(P_d, P_{fa}, N)$

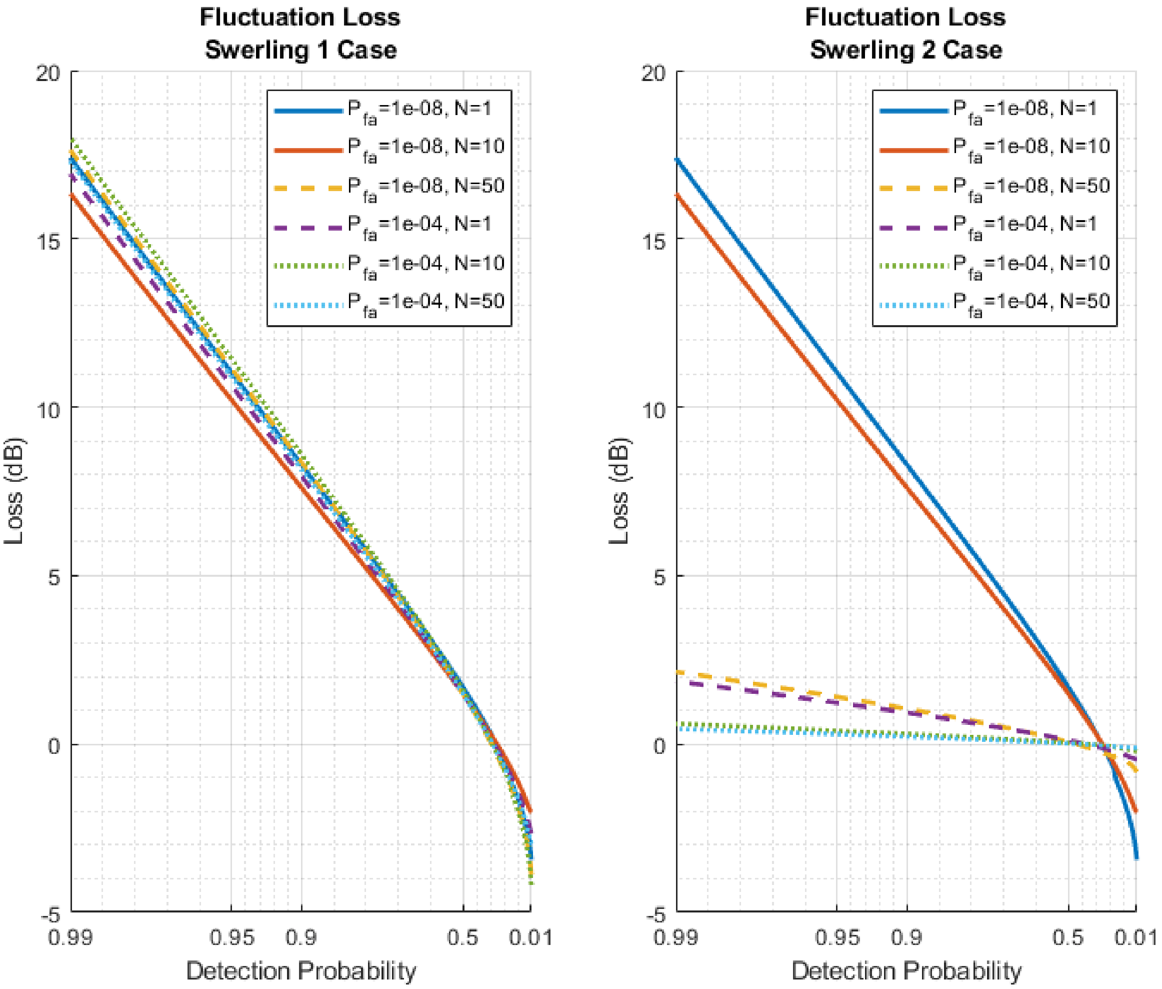
$$L_{kf} = D_k(P_d, P_{fa}, N) - D_0(P_d, P_{fa}, N)$$

where the subscript $k = 1, 2, 3, 4$ indicates the Swerling case [1]. In this example we perform computation for Swerling 1 and Swerling 2 cases that model slowly 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 below 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 (e.g., frequency, space, polarization, etc.). As was just shown, 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 that the required P_d and P_{fa} are

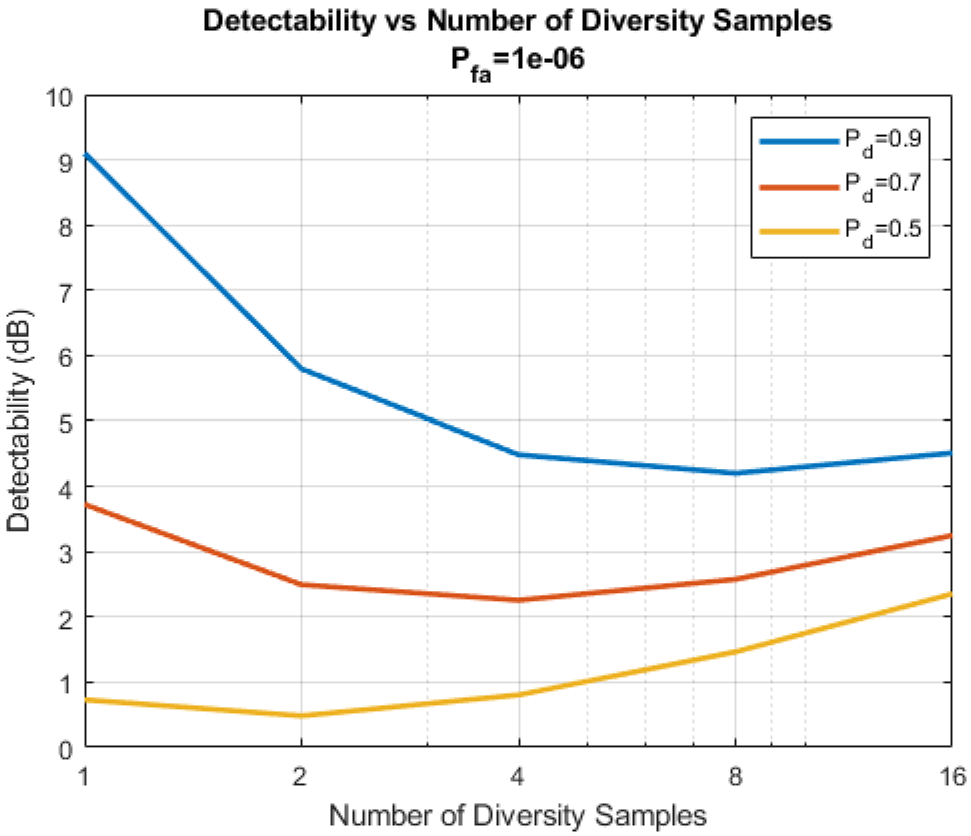
```
% Detection probability
Pd = [0.9 0.7 0.5];

% 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 plot below shows the detectability factor as a function of the number of diversity samples. From this result we 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
```



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, i.e., 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

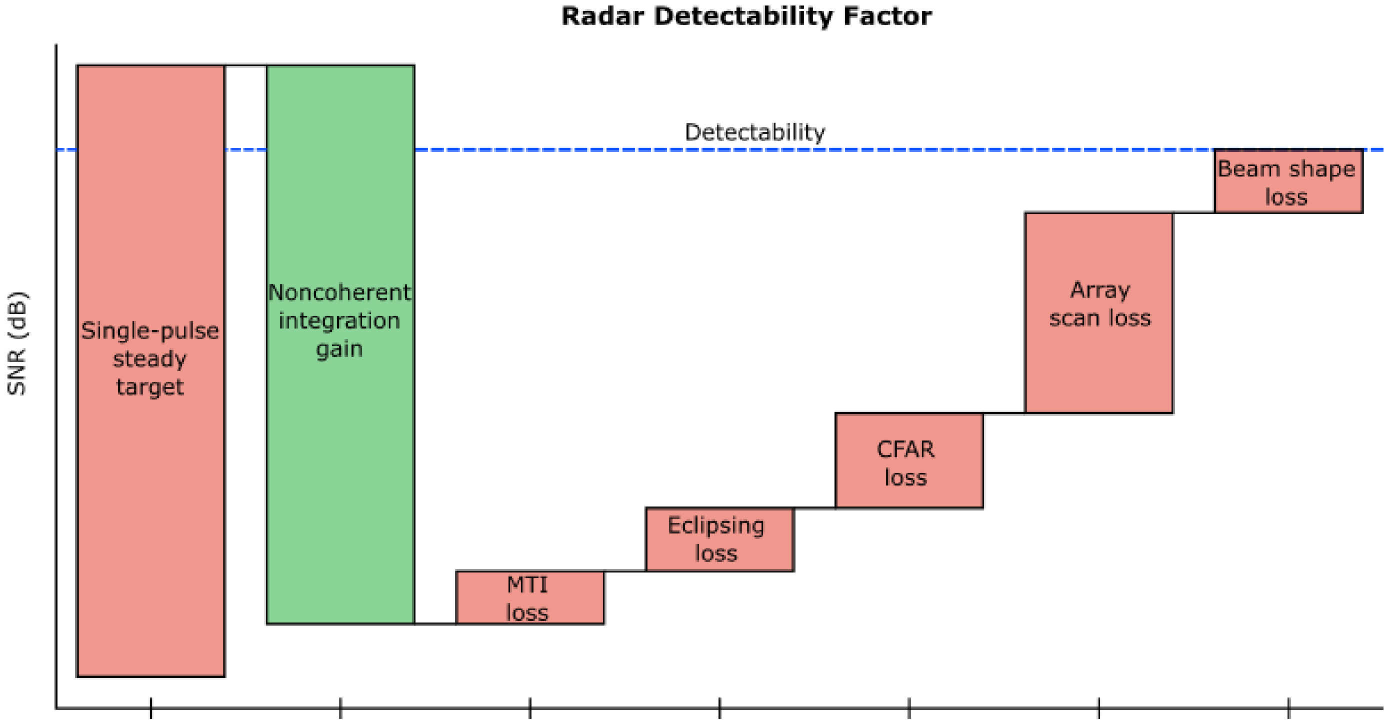
1 Barton, D. K. *Radar equations for modern radar*. Artech House, 2013.

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*. McGraw-Hill Education, 2014.

Introduction to Scanning and Processing Losses in Pulse Radar

In an idealized radar system with no losses and a non-fluctuating 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 demonstrates 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 MTI processing. It also addresses CFAR loss and filter matching loss.



The “Modeling Radar Detectability Factors” on page 1-372 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, $\overline{\chi}$

$p = p(\overline{\chi})$.

For a specific target position, the available SNR, $\overline{\chi}_0$, 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

$$p = p(\overline{\chi}_0 H(x))$$

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

$$\overline{P} = \frac{1}{x_2 - x_1}\int_{x1}^{x2} p(\overline{\chi}_0 H(x))dx$$

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 $\overline{\chi}_s$ needed to achieve the required P_d on average can be found by solving the following equation

$$P_d - \frac{1}{x_2 - x_1}\int_{x1}^{x2} p(\overline{\chi} H(x))dx = 0$$

The increase in the SNR, $\overline{\chi}_s$, compared to the idealized case when there are no probability of detection variations with x, is called a statistical loss [1]

$$L_s = \frac{\overline{\chi}_s}{\overline{\chi}_0}$$

This example considers three statistical losses:

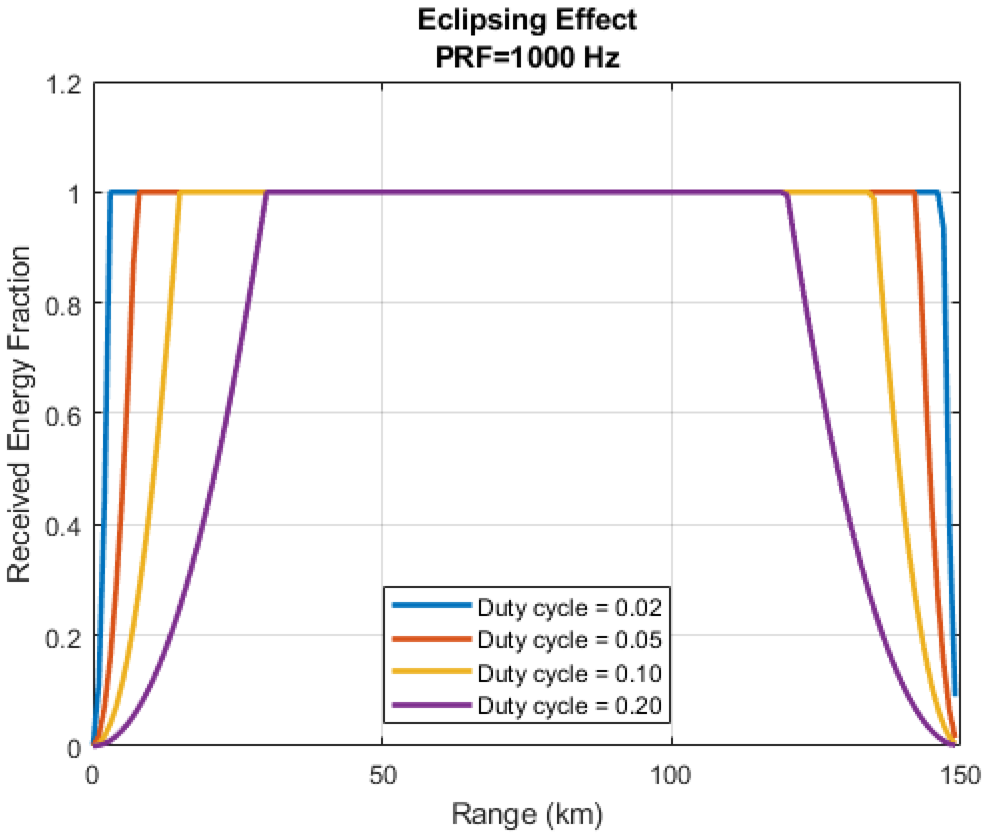
- Eclipsing loss - accounts for P_d variations with range due to the pulse eclipsing effect.
- Array scan sector loss - accounts for P_d variations with a scan angle in electronically scanned arrays due the reduced projected array area in the beam direction and a reduction of the effective aperture area 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 below as a function of the target range assuming the 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
```

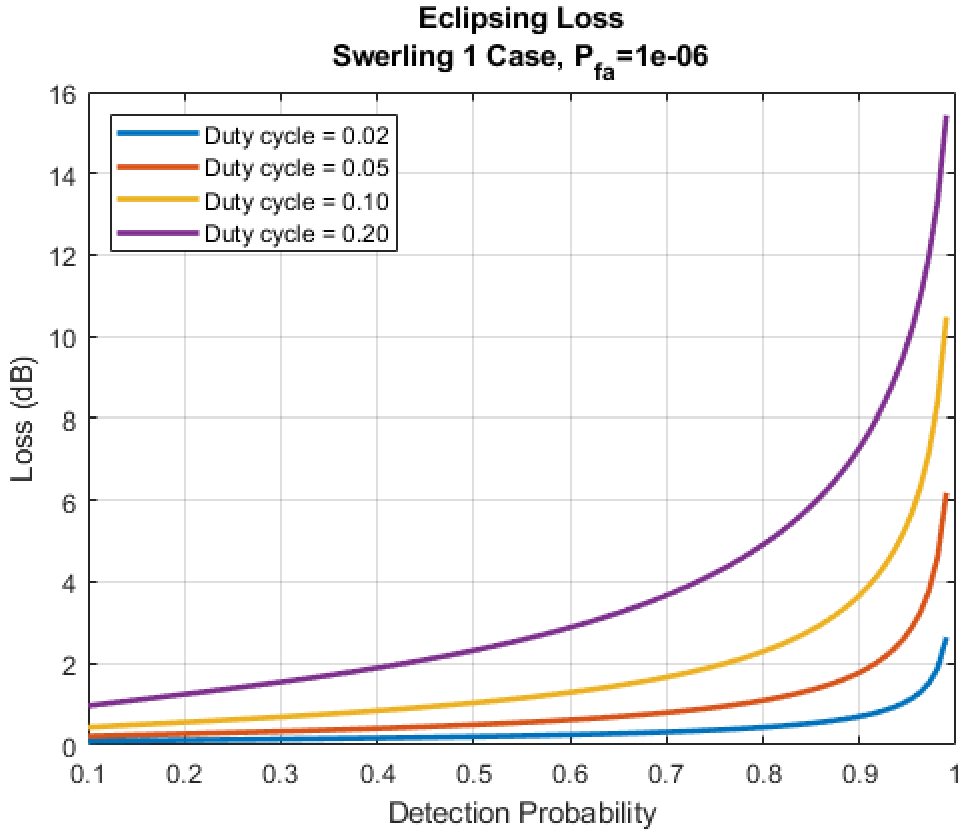
```
% Plot the eclipsing factor in linear units
figure
plot(R*1e-3,db2pow(Fecl),'LineWidth',2)

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])
```



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. We 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);

% 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
```

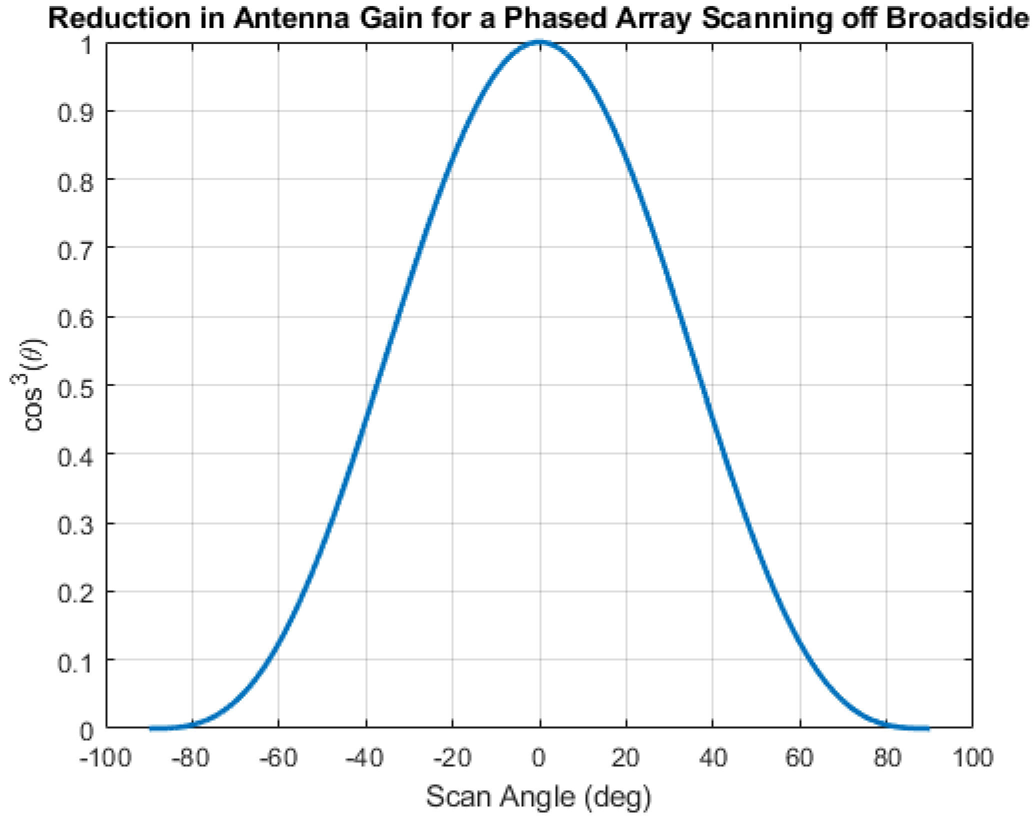


The statistical eclipsing loss increases with the duty cycle and can become very large for high values of P_d. This loss should be included in the radar equation when analyzing high-PRF radar systems in which a target can pass rapidly through the eclipsing regions. In such systems the PRF diversity is 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
```

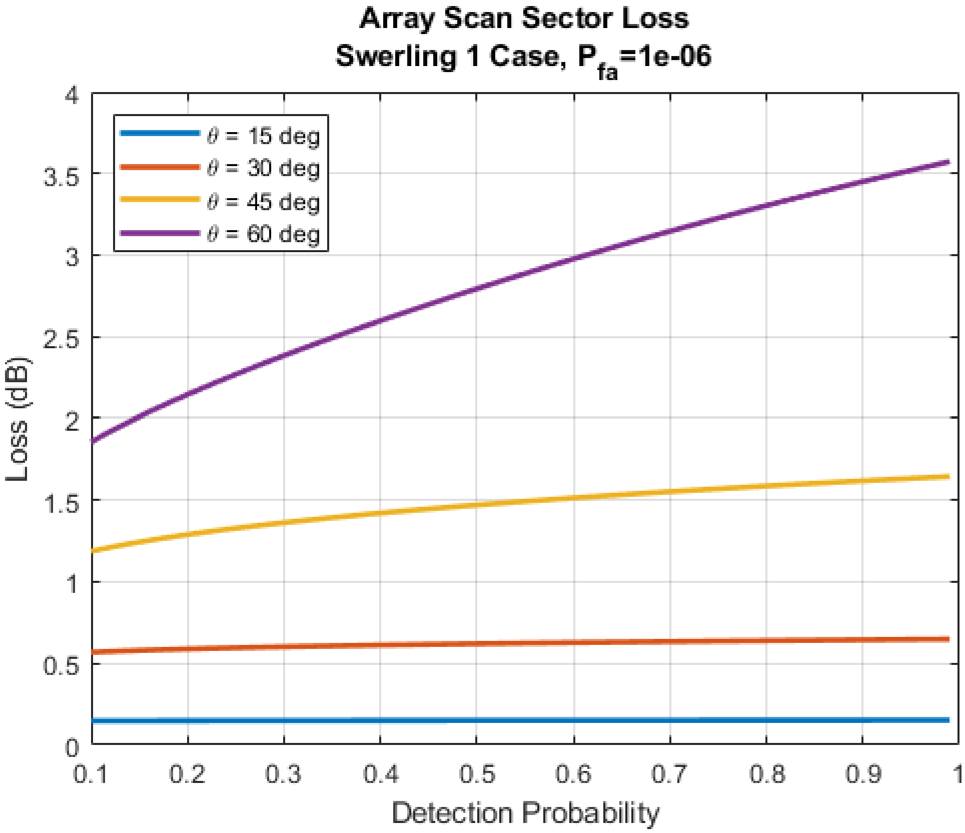


The figure above shows that the received energy is reduced by half when the beam is pointed to about 37 degrees off broadsided.

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.

We 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, i.e., 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
```

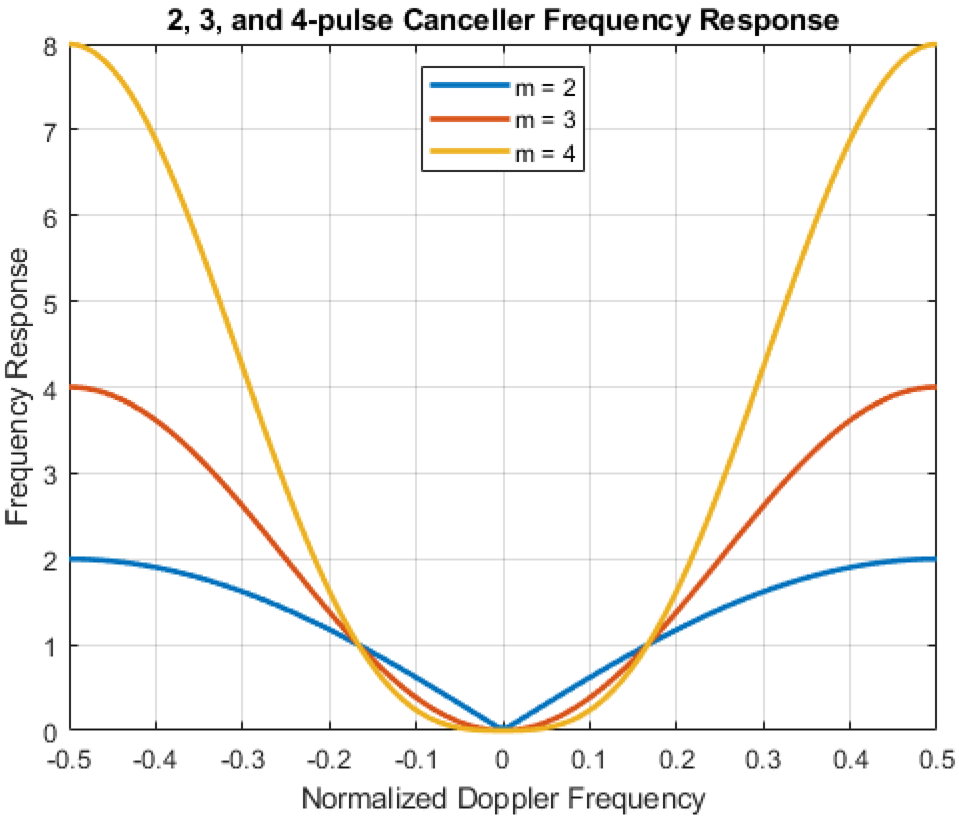


For small scan sectors the statistical scan sector loss is relatively small and does not significantly vary with the probability of detection. As the size of the scan sector increases, the reduction in antenna gain becomes larger resulting in more rapid increase in the scan sector loss for larger values of the detection probability.

MTI Velocity Response Loss

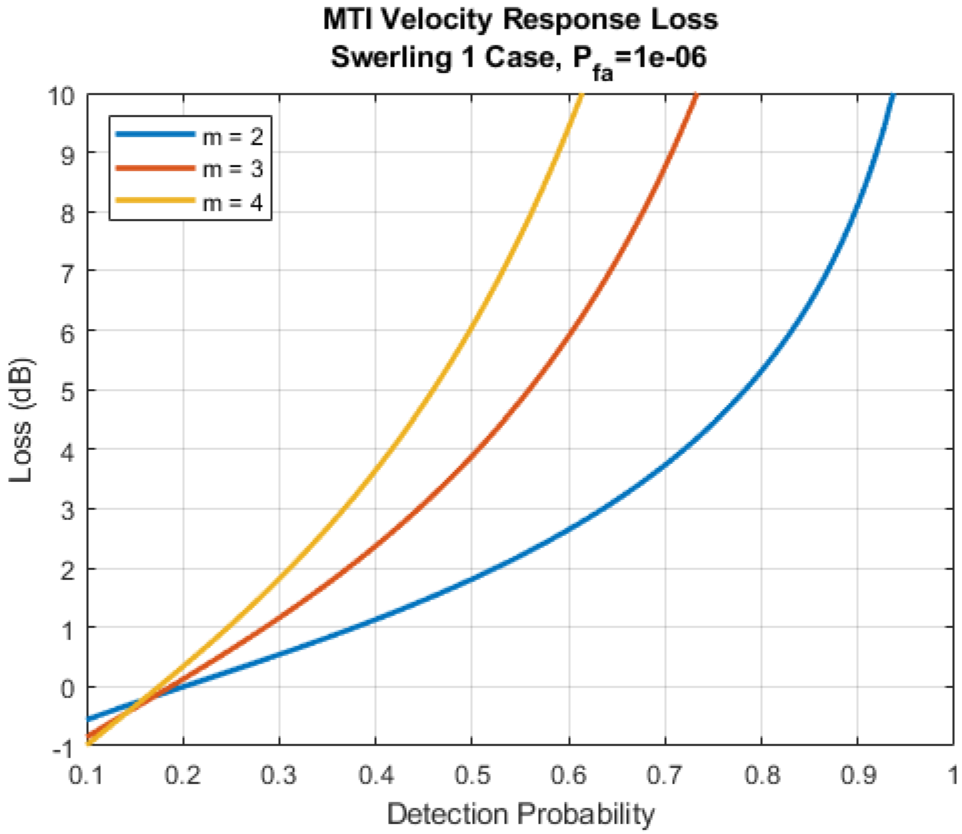
In moving target indicator (MTI) radar systems, the MTI filter is used to reject clutter components at or near zero Doppler frequency or near the center frequency of the clutter spectrum while passing the target signal spectrum as much as possible. Because the MTI filter is not ideal, it can significantly suppress or even cancel the targets that are close to the null of its frequency response. To illustrate this, the frequency responses of the 2, 3, and 4-pulse MTI cancellers are shown below

```
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. We 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]

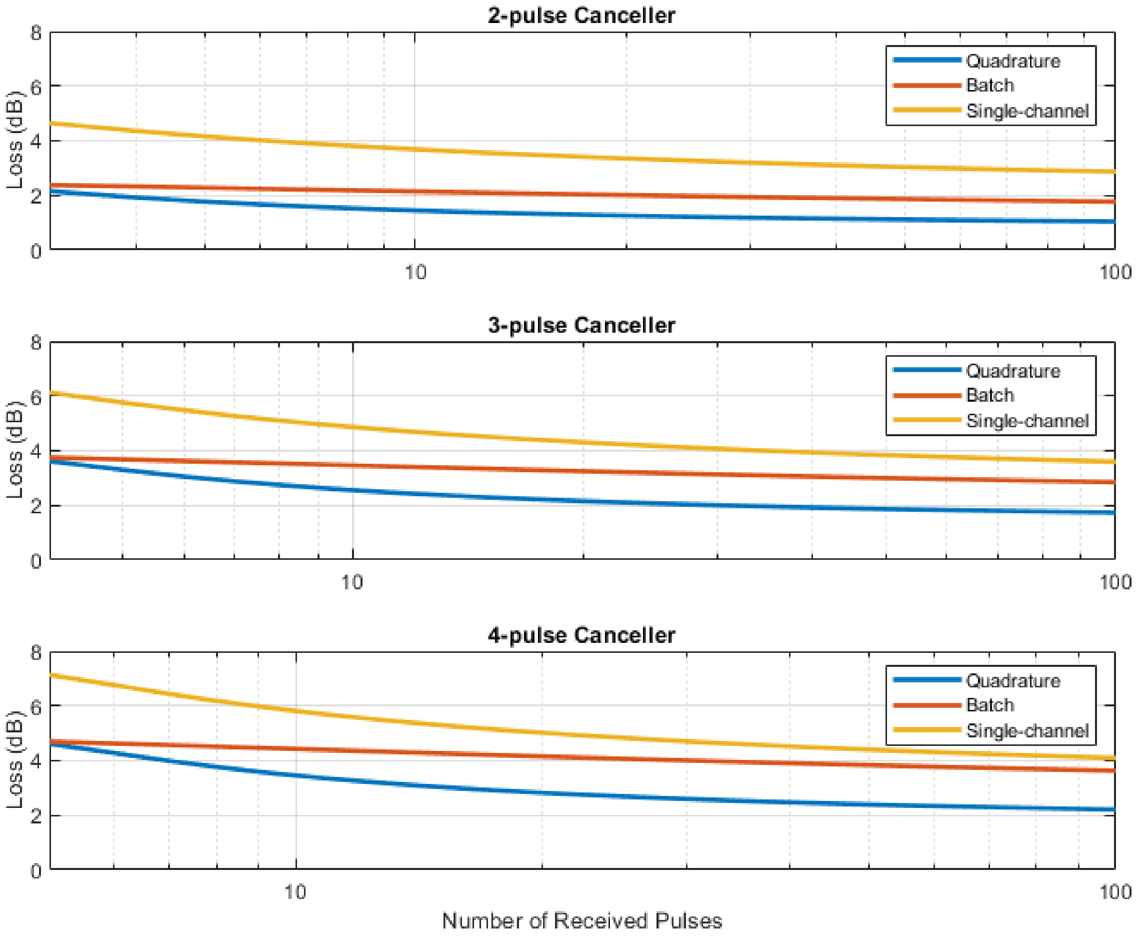
- MTI noise correlation loss. This loss is a result of the partial correlation introduced by the MTI processing to the received pulses that pass through an m-pulse MTI canceller.
- MTI batch processing loss. The result of using a batch processing MTI instead of a sequential MTI. A batch MTI processes 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 we consider 2, 3, and 4-pulse MTI cancellers. The probability of detection is set to $P_d = 0.9$ and the probability of false alarm to $P_{fa} = 10^{-6}$.

```
% 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. We consider a quadrature MTI with sequential processing, a quadrature MTI with batch processing, and a single-channel MTI with sequential processing.

```
% 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;
        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])
```



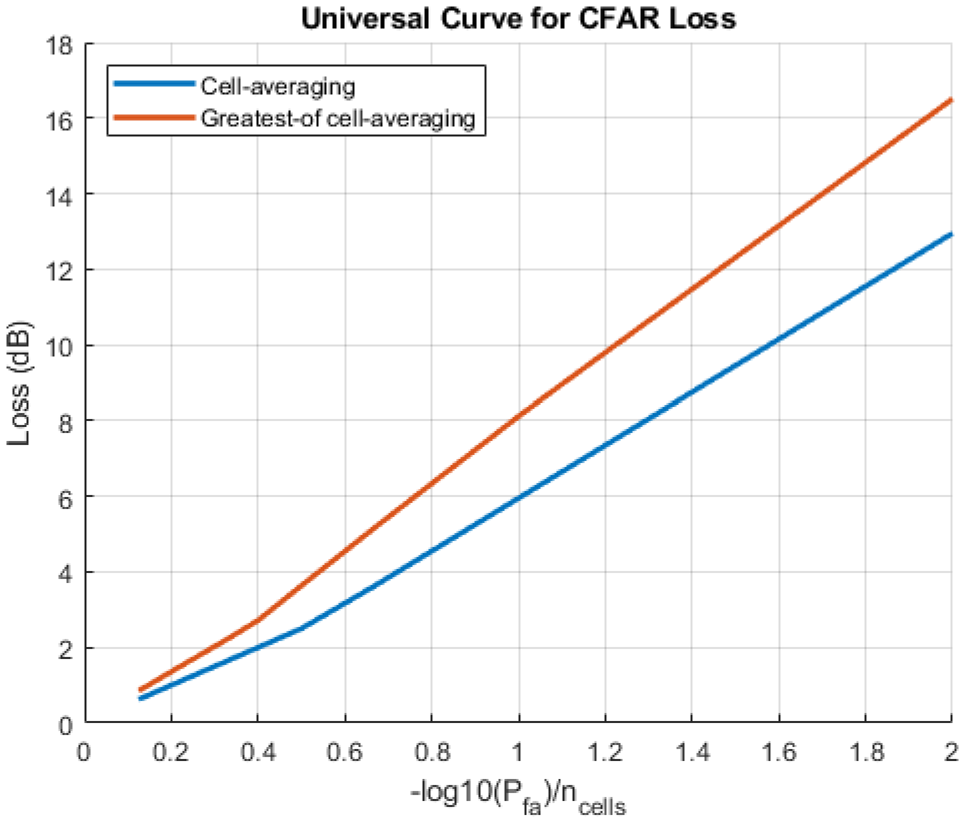
From these results we can see that the quadrature MTI with sequential processing is the most efficient. The noise correlation loss decreases as more pulses become available for integration. For the batch processing MTI, the loss is close to the sequential case when the number of received pulses is small. 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
```



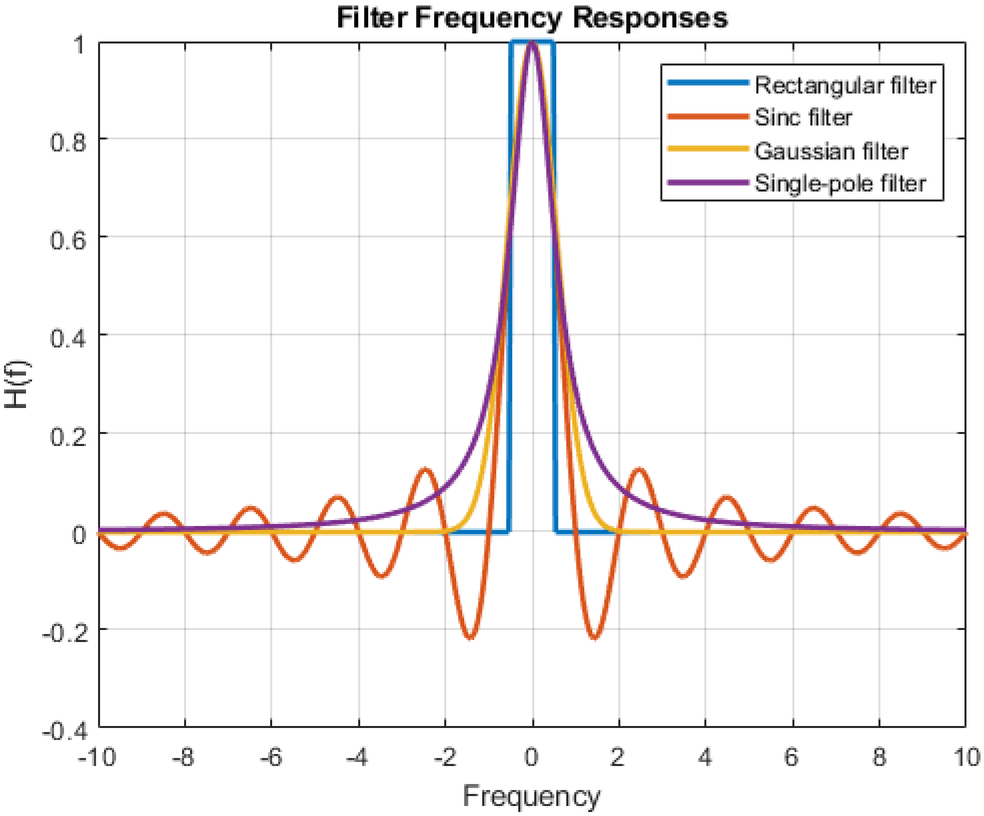
For a given CFAR method, the approximated loss depends only on the ratio $-log_{10}(P_{fa})/n_{cells}$, called a CFAR ratio. This allows for finding a trade-off between the required probability of false alarm and the number of reference cells to achieve a desired level of CFAR loss. The approximation 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 non-fluctuating 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

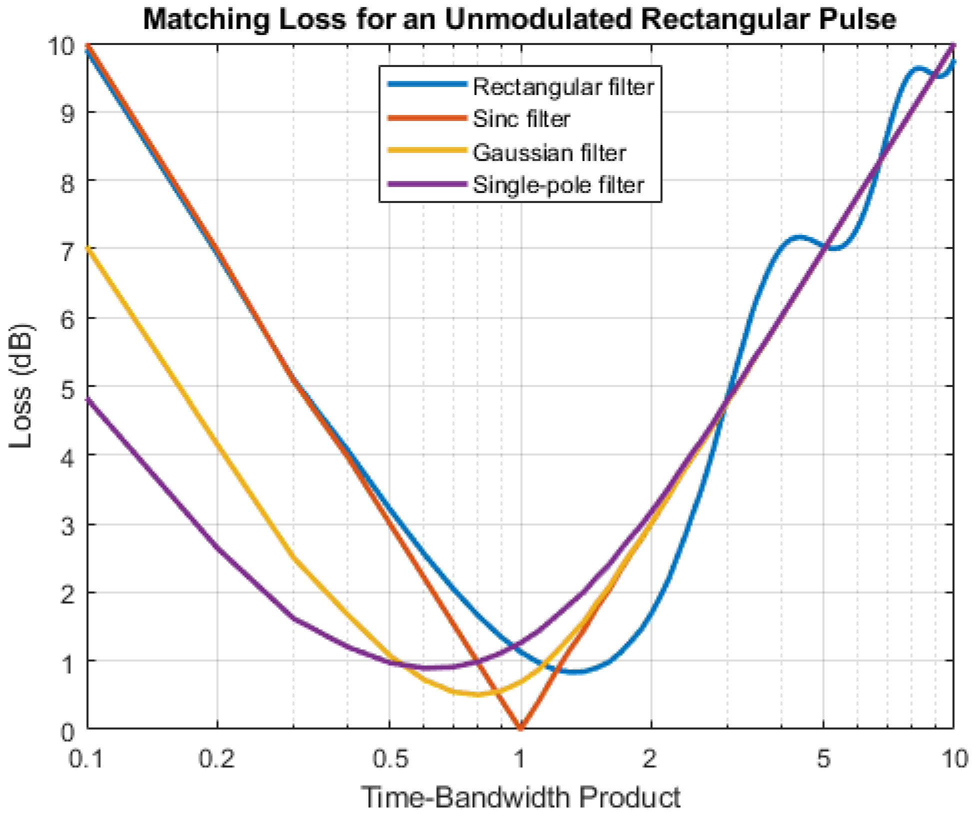
When the spectrum of the received signal is different from the spectrum of the matched filter, the system will incur a matching loss. This loss can be defined as a ratio of the received output SNR to the SNR available from a filter perfectly matched to the received signal. In this example we consider 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 figure below.

```
% 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])
```



The received signal in this example is an unmodulated ideal rectangular pulse. The filter matching loss for the four filter types is shown below 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
```



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 non-ideal MTI filter is demonstrated in this example. It is 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 demonstrates the computation of the filter matching loss.

References

1 Barton, D. K. *Radar equations for modern radar*. Artech House, 2013.

2 Richards, M. A. *Fundamentals of radar signal processing*. 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 Estimation Errors

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.

First, we will discuss several atmospheric models. Next, we will discuss an approximation of the standard atmospheric model using a simple exponential decay versus height. We will then discuss these models within the context of a maximum range assessment and visualize the refracted path. Lastly, we will end with a discussion on how the atmosphere causes errors in determining target height, slant range, and angle.

Atmospheric Models

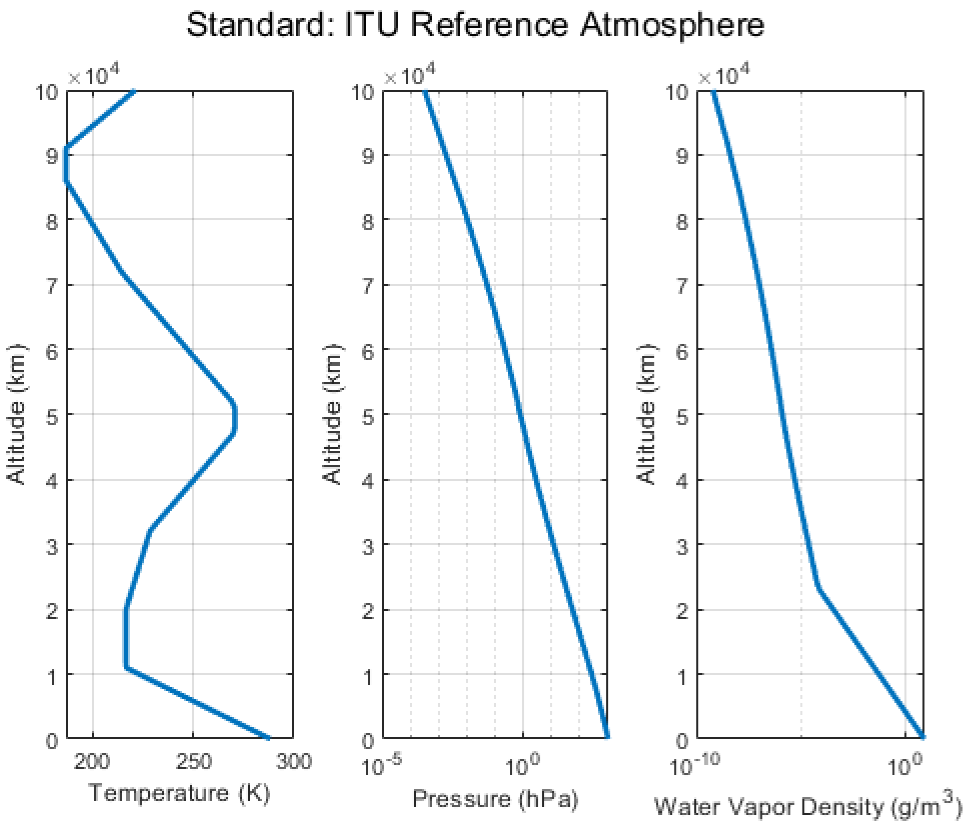
Calculating tropospheric losses and the refraction phenomenon requires models of the atmospheric temperature, pressure, and water vapor density, which are dependent on height. The function `atmositu` offers 6 ITU reference atmospheric models, namely:

- 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, hence there is only a single model. 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,h,'LineWidth',2)
grid on
xlabel('Temperature (K)')
ylabel('Altitude (km)')

% Plot pressure profile
subplot(1,3,2)
semilogx(P,h,'LineWidth',2)
grid on
xlabel('Pressure (hPa)')
ylabel('Altitude (km)')

% Plot water vapor density profile
subplot(1,3,3)
semilogx(wvden,h,'LineWidth',2)
```

```
grid on
xlabel('Water Vapor Density (g/m^3)')
ylabel('Altitude (km)')

% Add title
sgtitle('Standard: ITU Reference Atmosphere')
```



Once temperature T (K), pressure P (hPa), and water vapor density ρ (g/m3) profiles are obtained, these profiles can be translated to refractivity values N as

$$N = 77.6\frac{P}{T} - 5.6\frac{e}{T} + 3.75 \times 10^5\frac{e}{T^2},$$

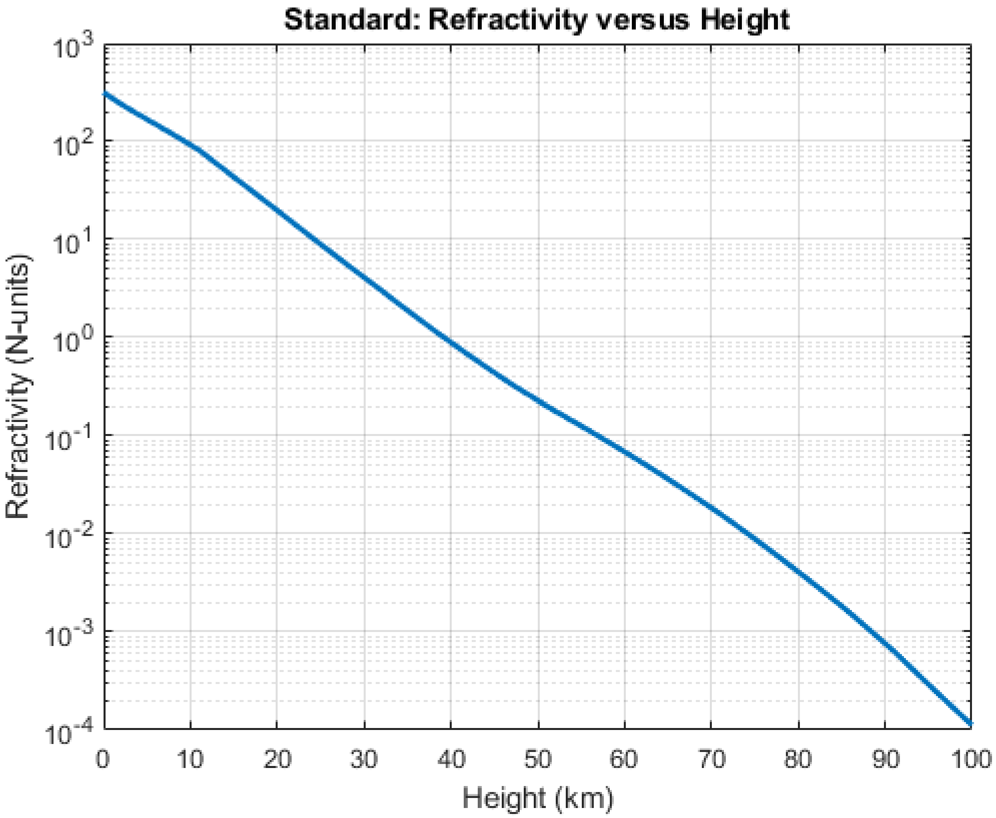
where e is the water vapor pressure (hPa) calculated as

$$e = \frac{\rho T}{216.7}.$$

The refractivity profile for the standard atmosphere is shown below and is obtained from the function `refractiveidx`. `refractiveidx` allows users to specify one of the ITU reference models listed above 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

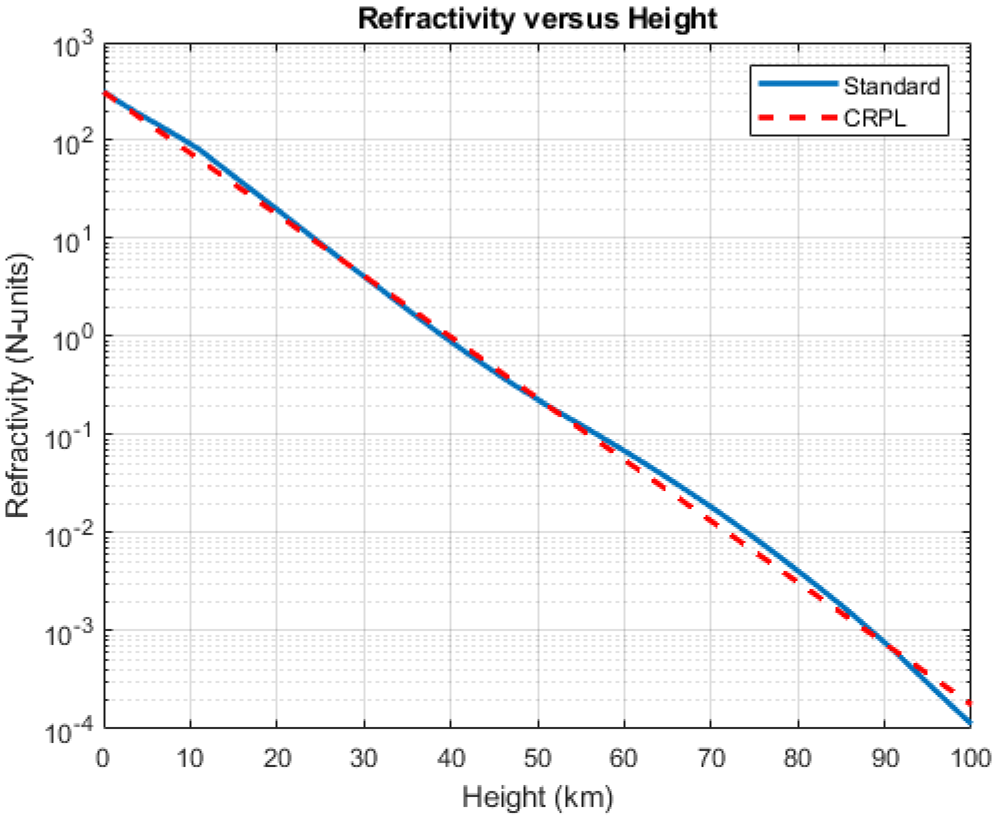
$$c_e = \ln\frac{N_S}{N_S - \Delta N}.$$

In the above equation, $\Delta N$is the difference between the surface refractivity N_S and the refractivity at a height of 1 km. ΔNcan 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.

```
% 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 taken into account through the use of 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, we need to compute an appropriate free space range. Consider the case of an X-band radar system operating in an urban environment.

```
% Radar paramaters
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] = buildingMaterialPermittivity('concrete',freq);
```

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);
```

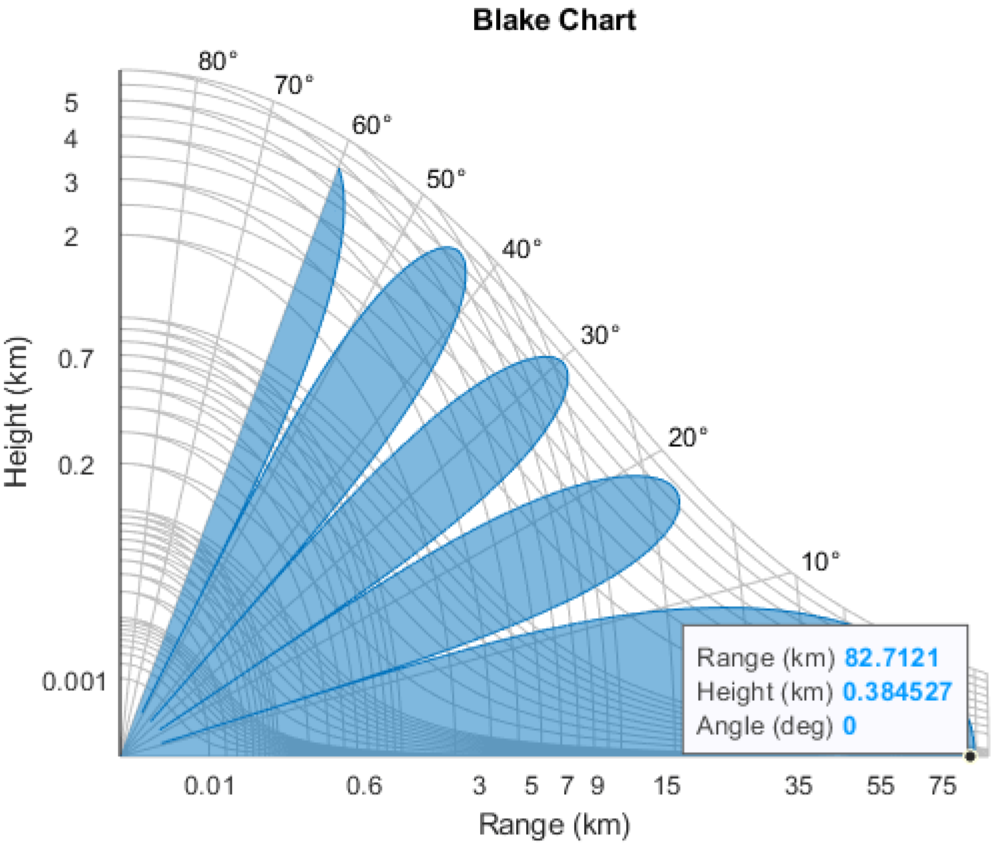
```
Minimum detectable SNR = 3.5 dB
```

```
% 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 figure below 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;
tgthtNoRef = range2height(Rm,anht,el0,'EffectiveEarthRadius',physconst('EarthRadius'));
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 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, one 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 would be about 11/4 and the radius would be 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 takes into account 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. For this example, we will 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
[~,srTrue,elTrue] = 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
```

```
[~,trueSR,trueEl] = height2range(tgthtCRPL(end),anht,el0,'Method','CRPL');
fprintf('Target slant range error = %.4f m\n',Rm(end) - trueSR);
```

```
Target slant range error = 25.3316 m
```

```
fprintf('Target angle error = %.4f deg\n',el0 - trueEl);
```

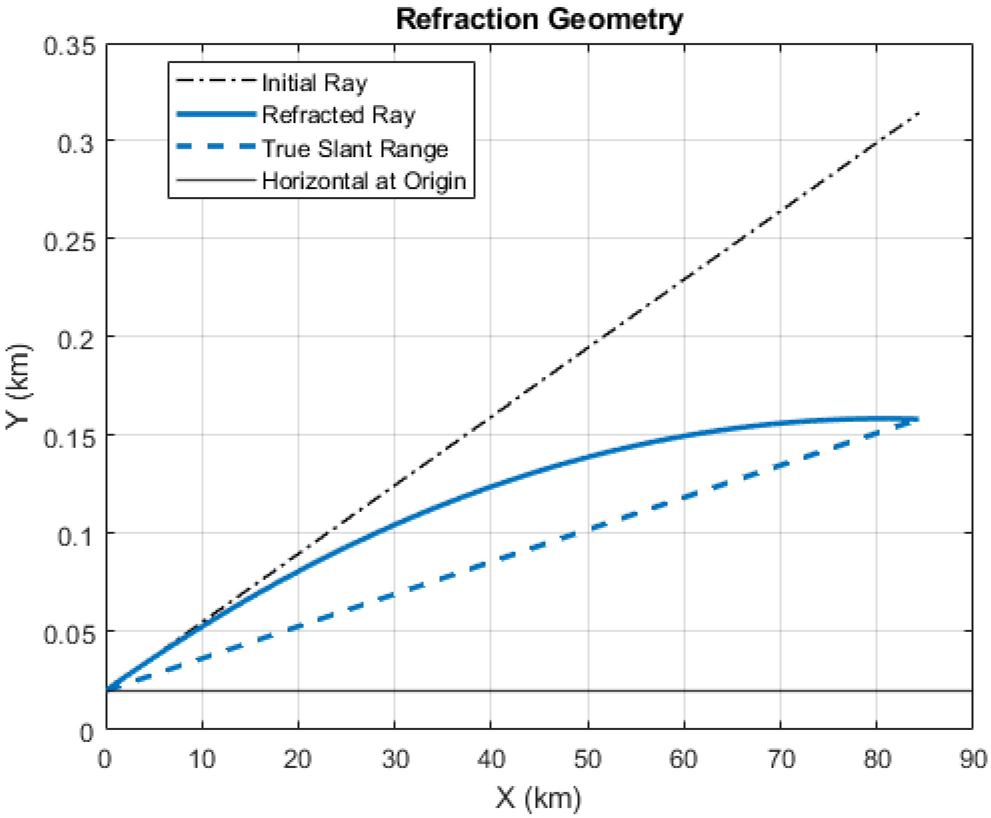
```
Target angle error = 0.1059 deg
```

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 (i.e., 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(elTrue),srTrue*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.

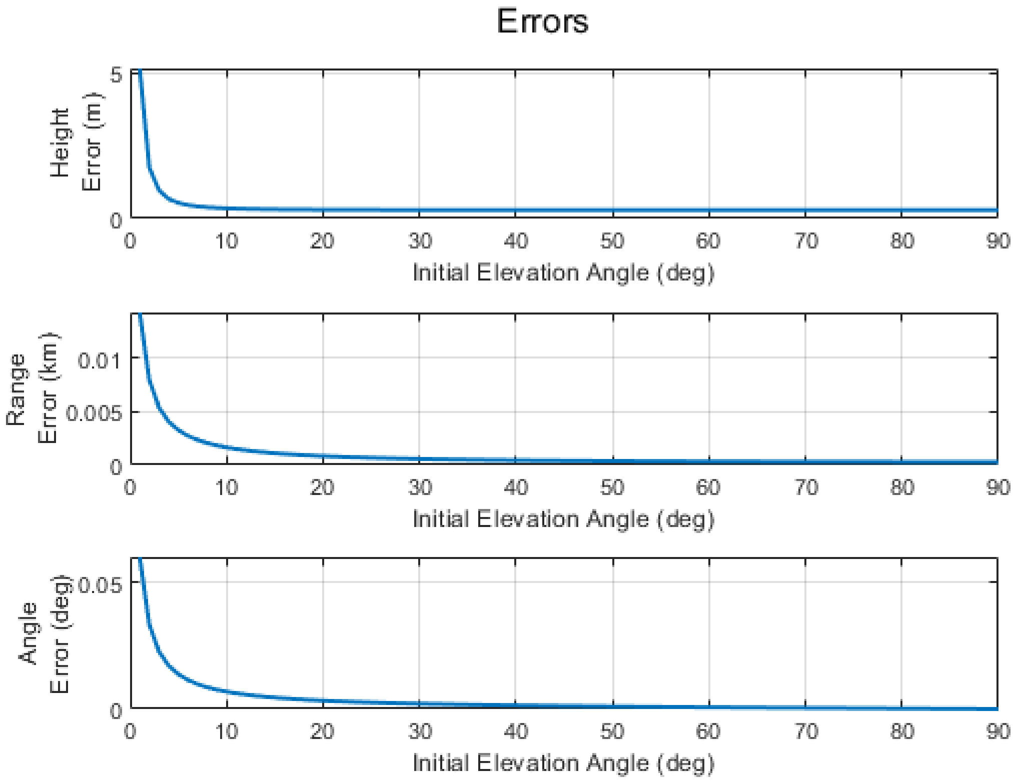
```
% Analysis parameters
tgthtTrue = 1e3;     % True target height (m)
el        = 90:-1:1; % Initial elevation angle (degrees)
anht      = 10;      % Antenna height (m)
```

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).*1e-3,'LineWidth',1.5)
grid on
xlabel('Initial Elevation Angle (deg)')
ylabel(sprintf('Range\nError (km)'))

% 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.

Summary

This example discussed atmospheric models and investigated their effect on target range, height, and angle estimation.

References

1. International Telecommunication Union (ITU). "Attenuation by Atmospheric Gases." Recommendation ITU-R P.676-12, P Series,
2. 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.
3. Barton, David K. Radar Equations for Modern Radar. Norwood, MA: Artech House, 2013.
4. Blake, L.V. "Radio Ray (Radar) Range-Height-Angle Charts." Naval Research Laboratory, NRL Report 6650, Jan. 22, 1968.
5. 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.
6. Bean, B. R., and G. D. Thayer. CRPL Exponential Reference Atmosphere. Washington, DC: US Gov. Print. Off., 1959.

See Also

`atmositu` | `blakechart` | `detectability` | `height2range` | `landroughness` | `radareqrng` | `radarvcd` | `range2height` | `refractionexp` | `refractiveidx`

Related Examples

- “Radar Vertical Coverage over Terrain” on page 1-599

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]:

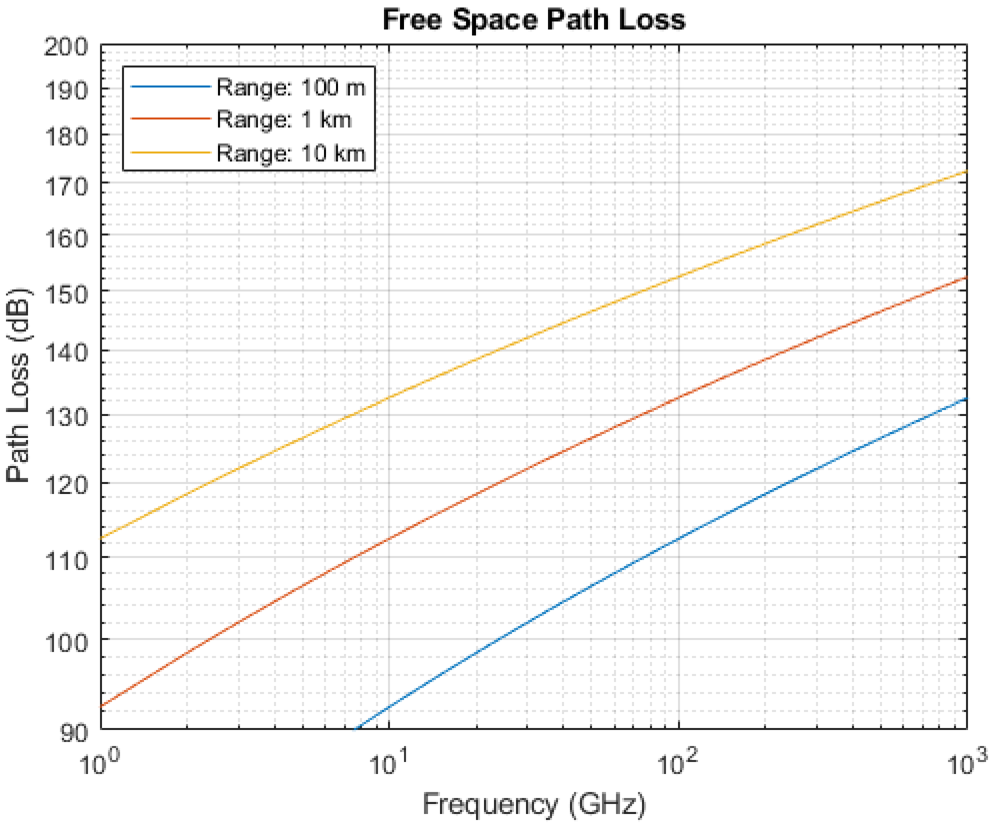
$$L_{fs} = 20 * \log_{10}(\frac{4\pi R}{\lambda}) \quad dB$$

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

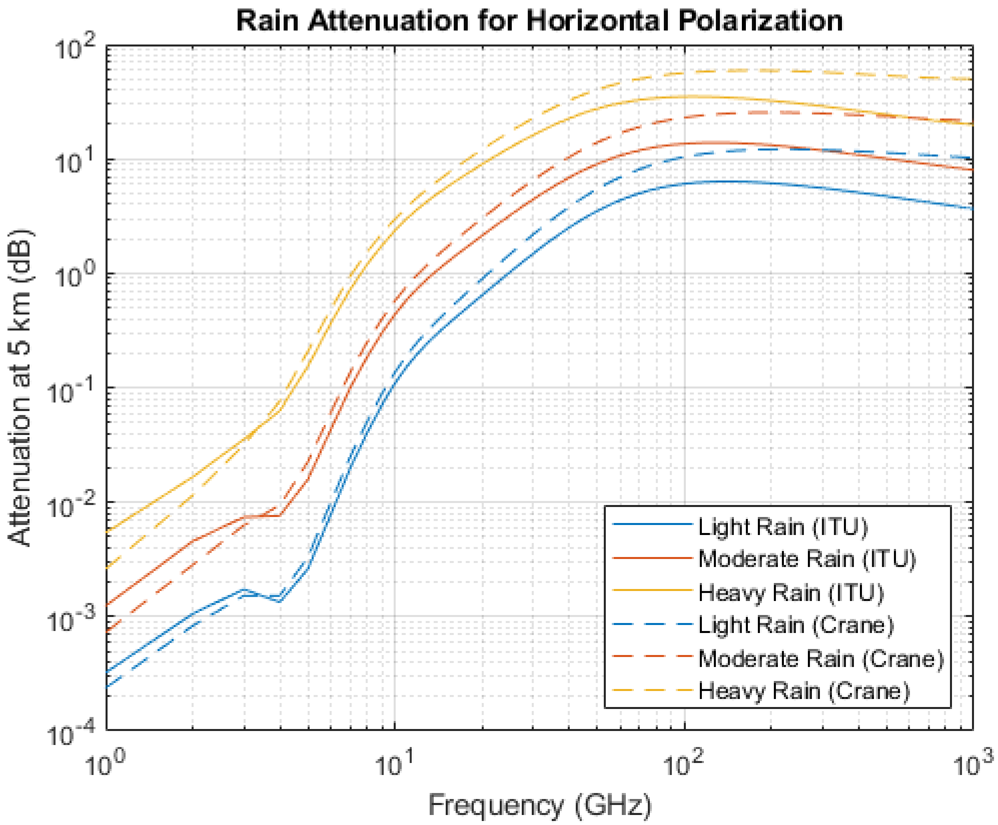
In reality, signals do not always travel in a vacuum, so free space path loss describes only part of the signal attenuation. Signals interact with particles in the air and lose energy along the propagation path. The loss varies with different factors such as pressure, temperature, and water density.

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.

```
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');
```

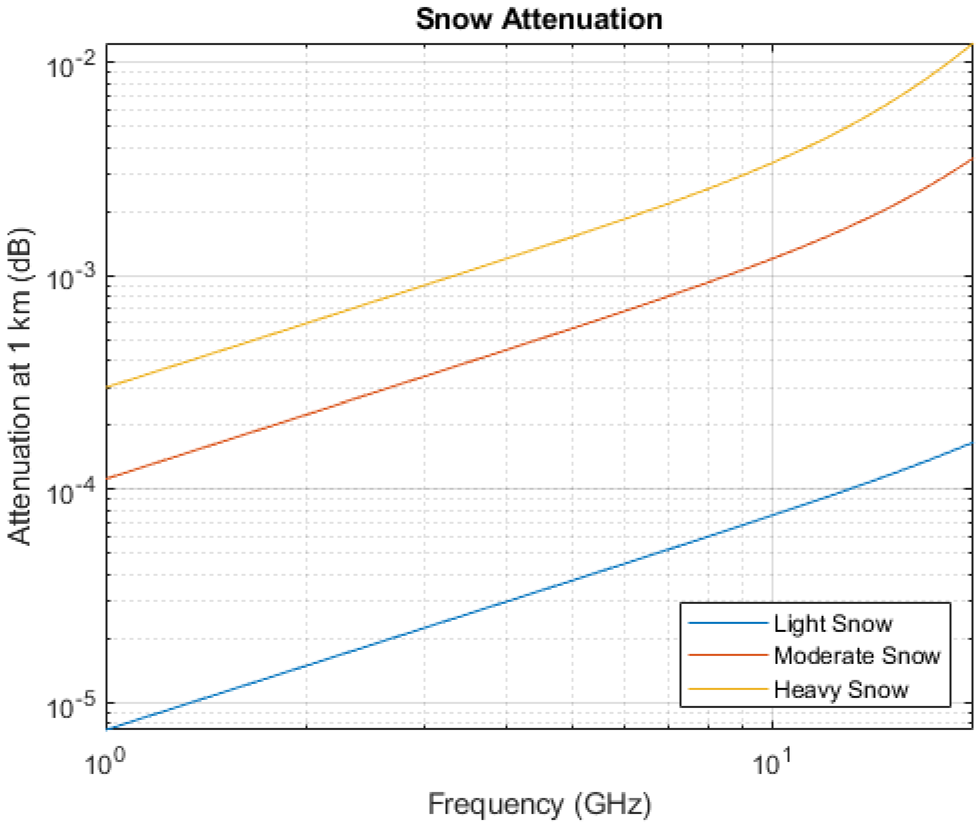


The losses computed with the Crane model are mostly larger than the losses computed with the ITU model at this propagation range. At smaller propagation ranges and lower frequencies, the ITU model may output a smaller attenuation value than Crane. Note that the models differ greatly enough that at higher 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');
```

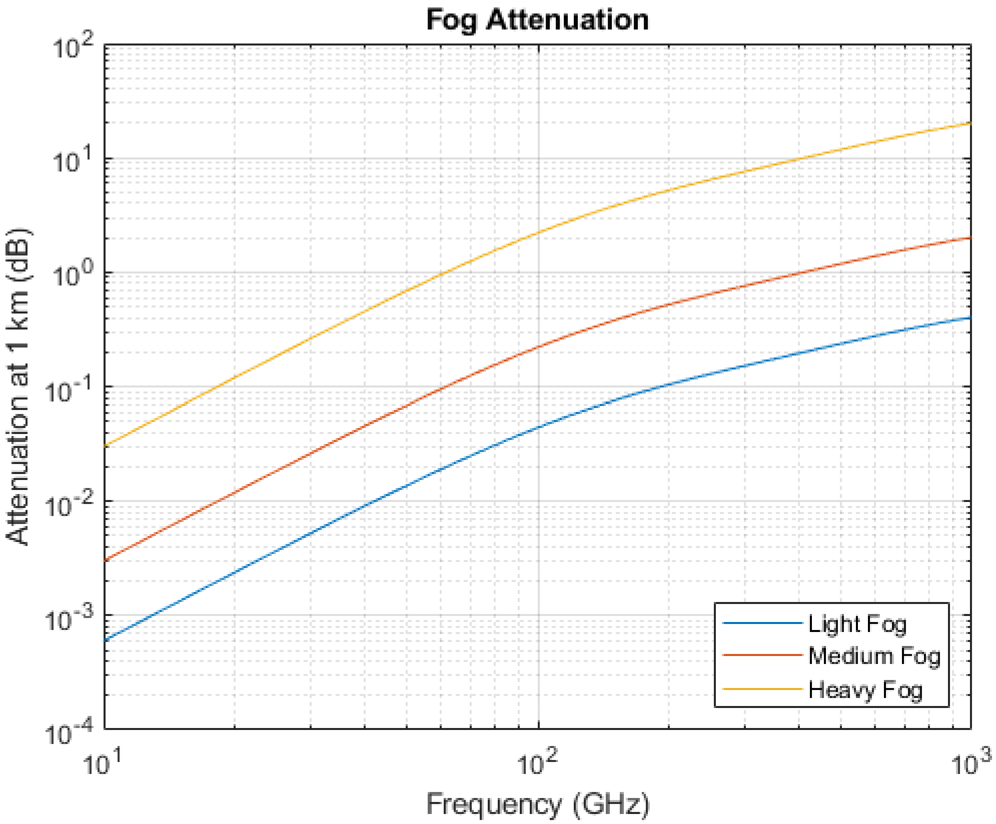


Loss Due to Fog and Cloud

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');
```



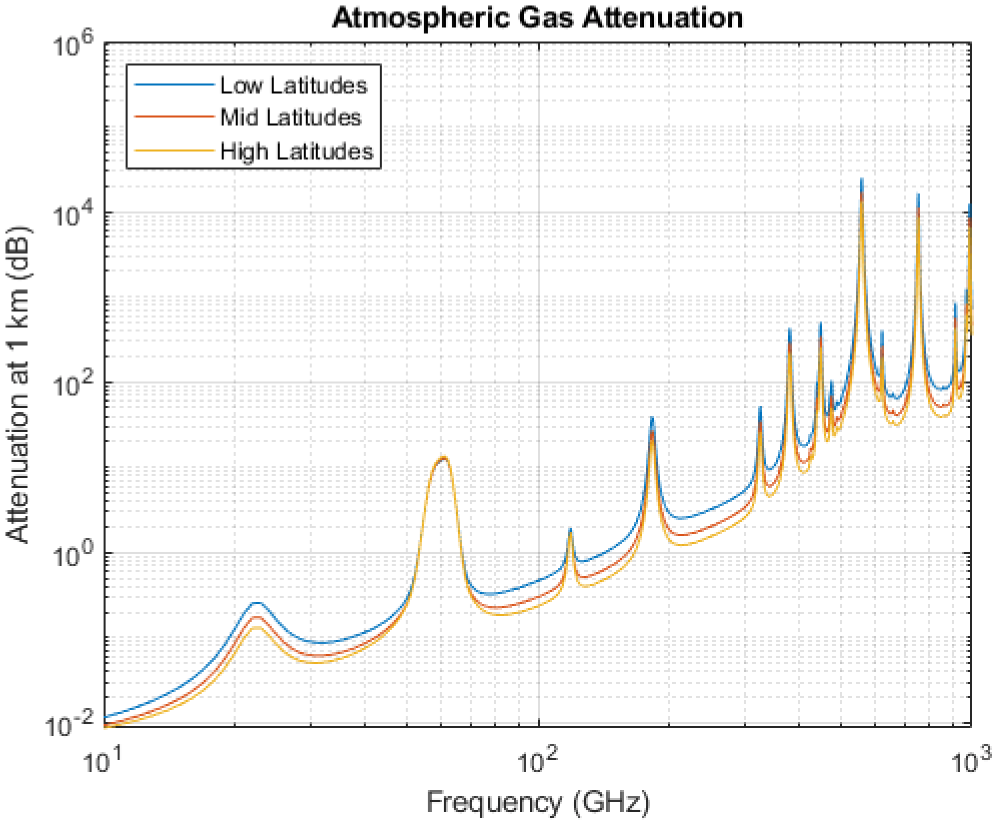
Note that in general fog is not present when it is raining.

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.

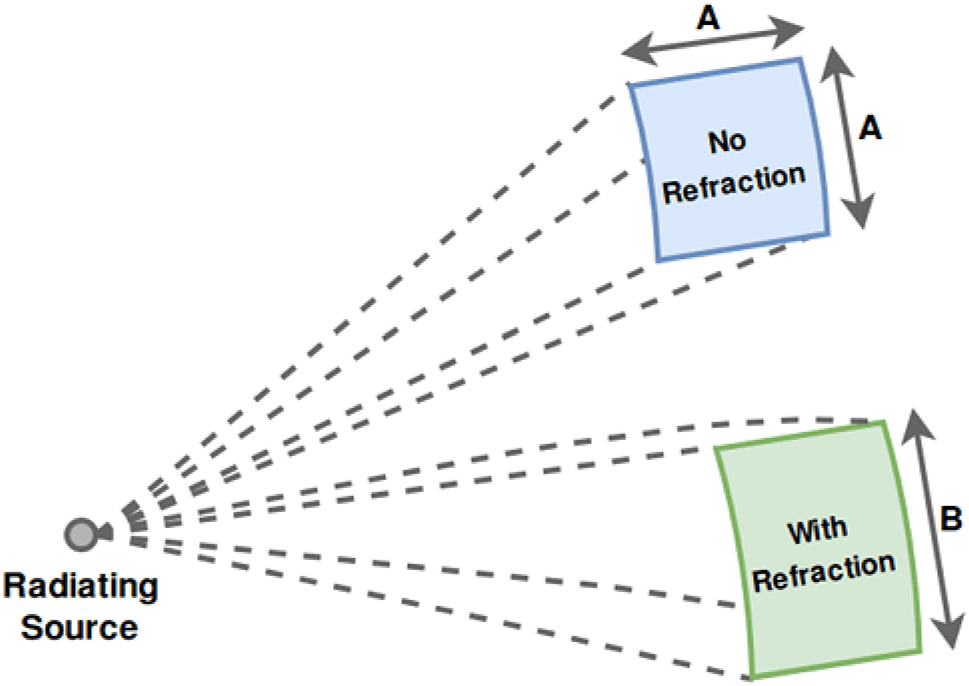
```
height = 2e3;
el = -5; % elevation angle
atmloss_low = tropopl(R0,freq,height,el,'LatitudeModel','Low');
atmloss_mid = tropopl(R0,freq,height,el,'LatitudeModel','Mid');
atmloss_high = tropopl(R0,freq,height,el,'LatitudeModel','High');
loglog(freq/1e9,atmloss_low);
hold on;
loglog(freq/1e9,atmloss_mid);
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.

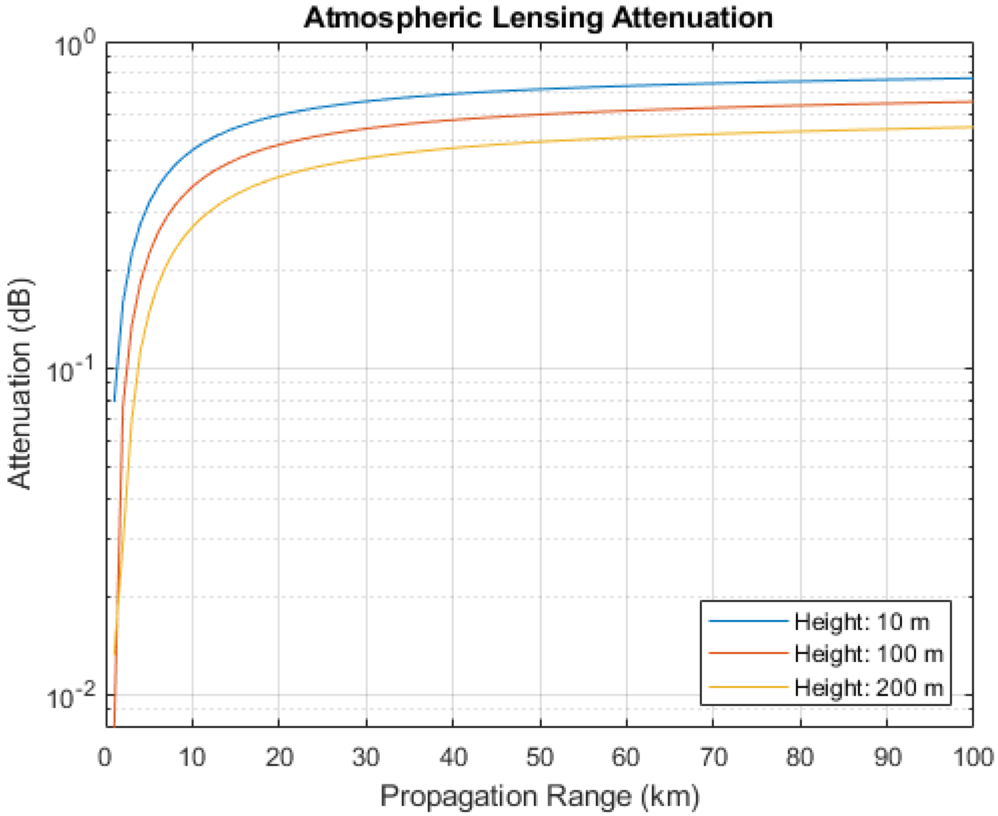
Another source of losses due to atmosphere is from atmospheric lensing. This is a phenomenon whereby the angular extent of a transmission is increased with range due to a refractivity gradient. This spreading of energy decreases the energy density along the nominal (straight) propagation path, independent 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`.

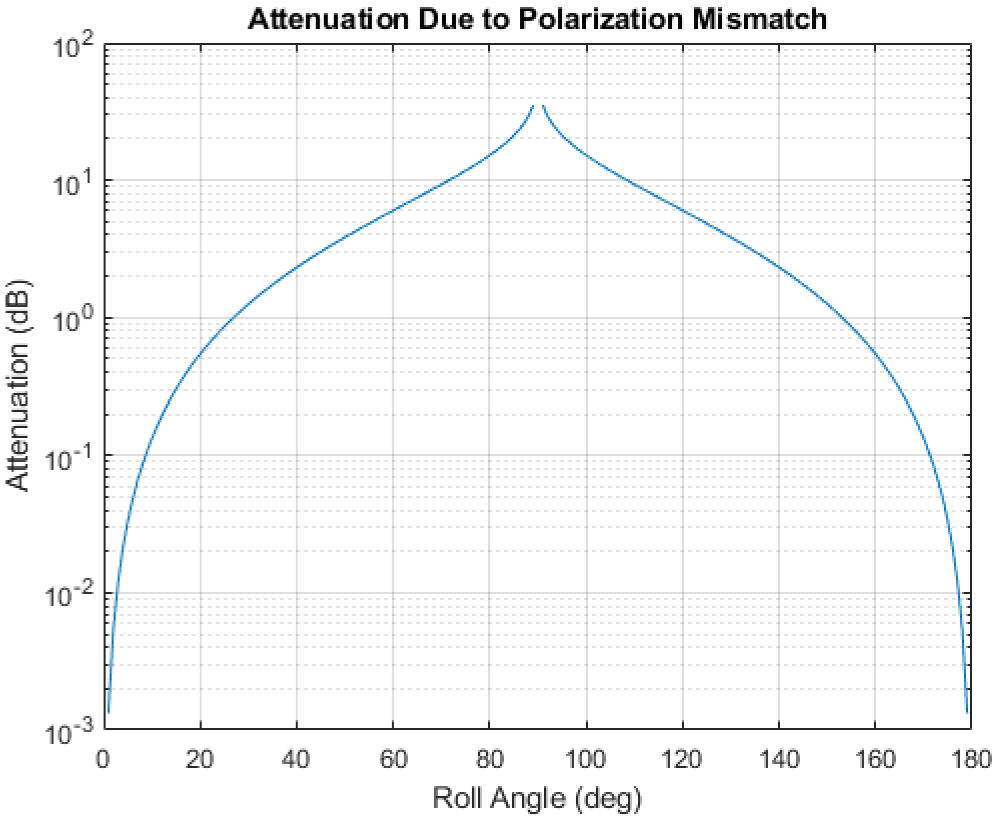
Loss Due to Polarization Mismatch

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.

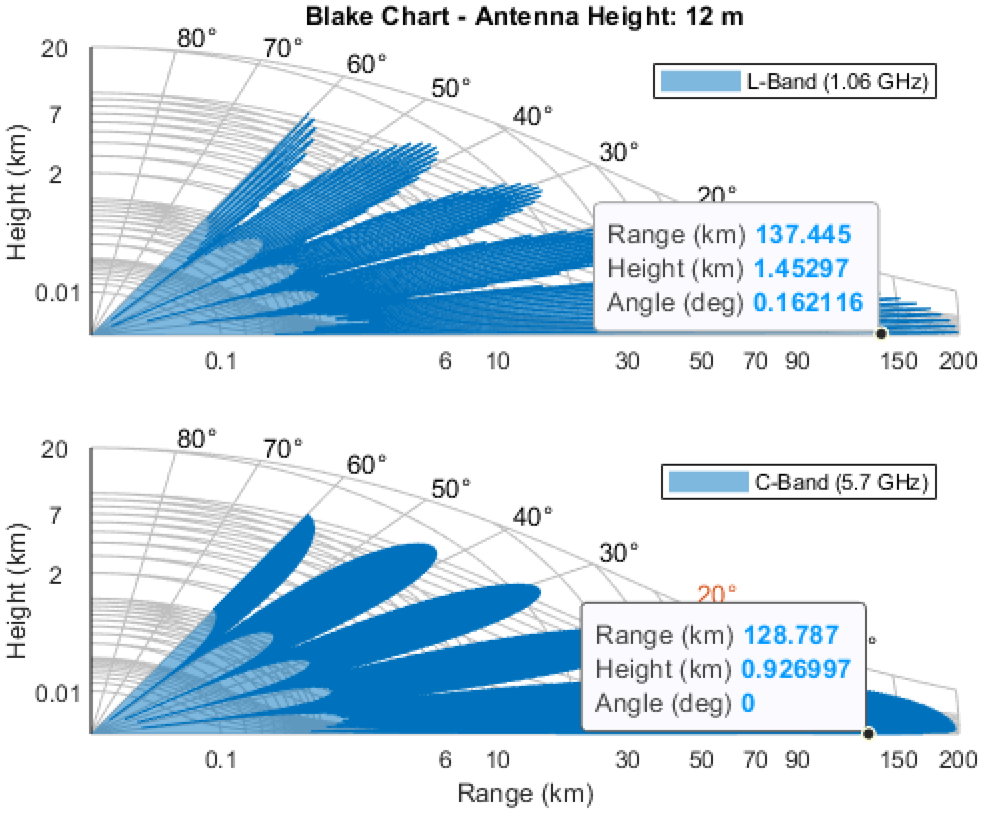
The function radarvcd takes a reference range as input and returns the range at which the received power in the multipath environment equals what it would be in free space. This effective range is plotted on a range-height-angle chart. This can quickly give, for example, the actual detection range given 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.

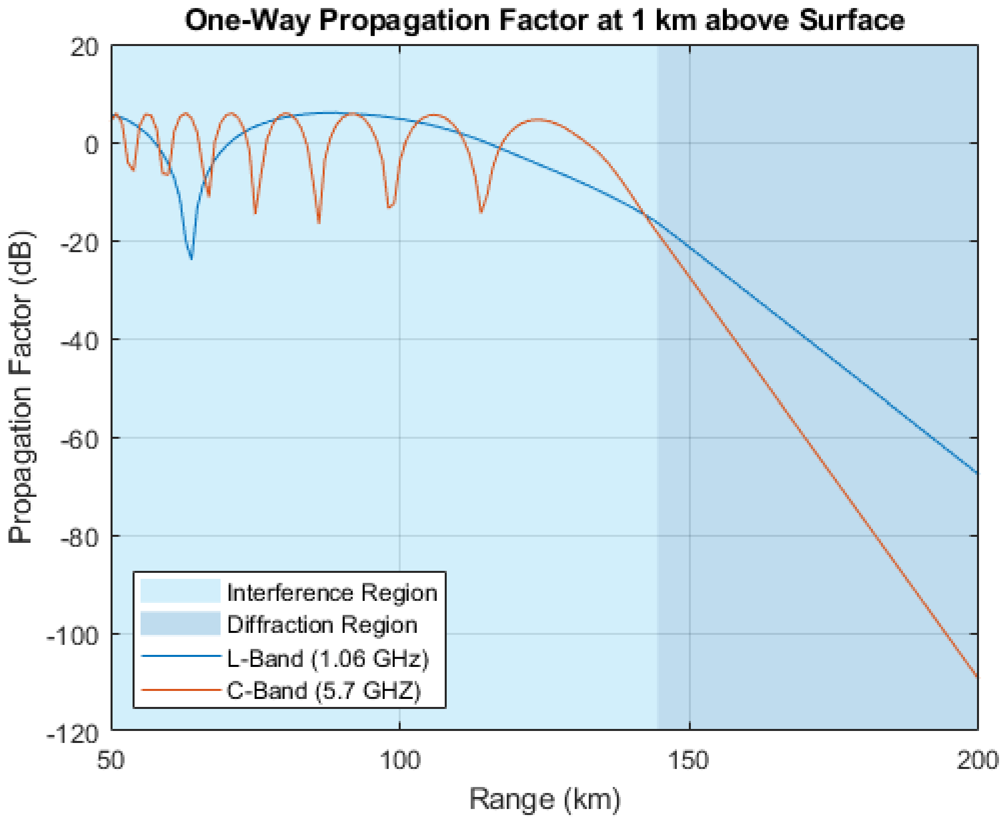
```
[~,N] = refractiveidx(0); % atmospheric refractivity at the surface
helperPlotBlakeChart(vcp,vcpang,N)
```



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,'SurfaceSlo
end
helperPlotPropagationFactor(R,F,Rd)
```

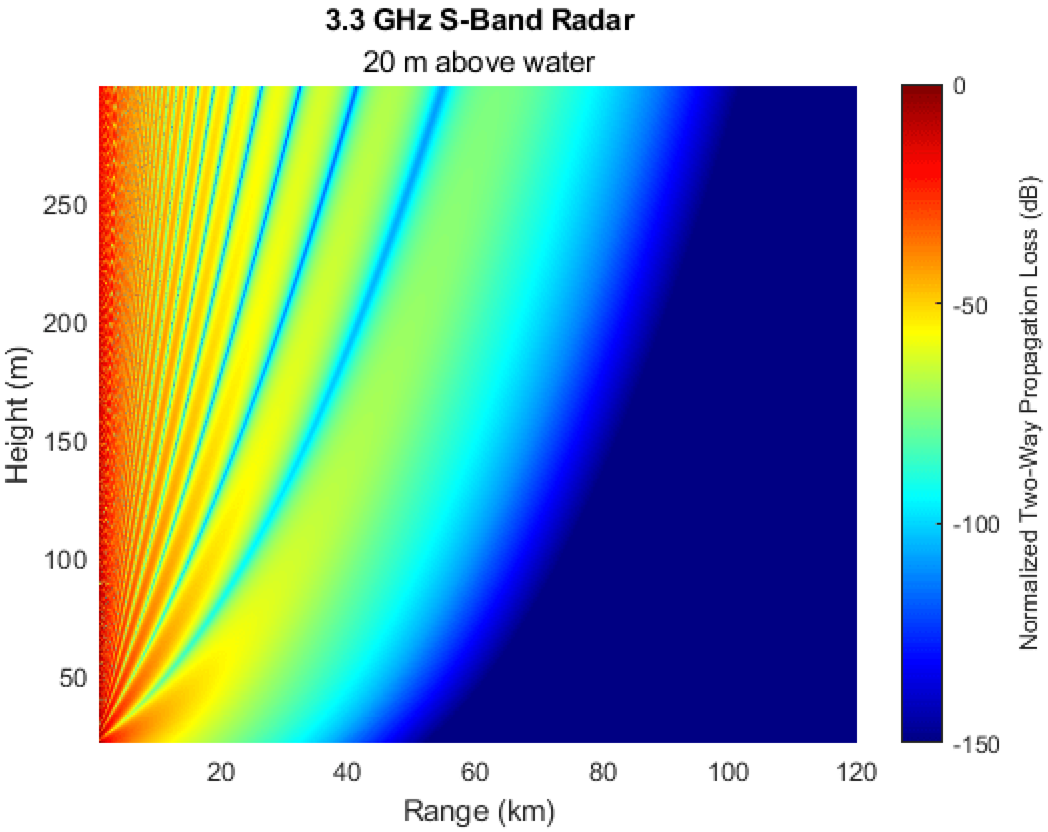


The propagation factor oscillates in the interference region, then decreases quickly in the diffraction region.

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.

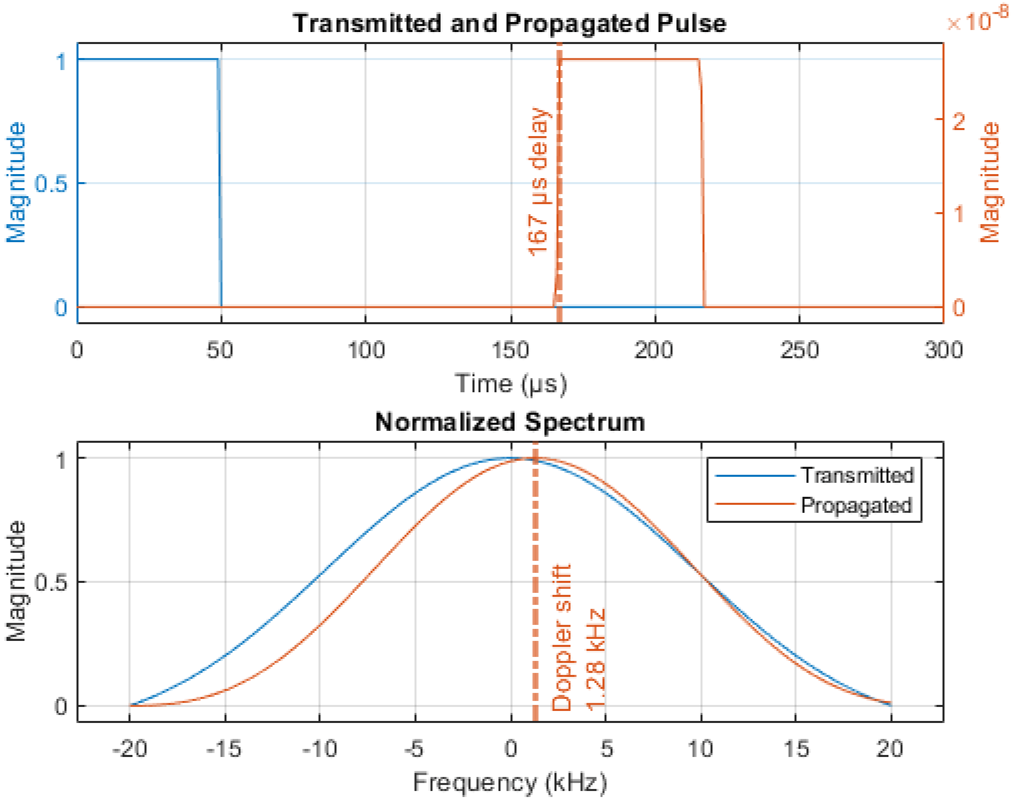
```
fc = 1.9e9;
channel = twoRayChannel('PropagationSpeed',c,'OperatingFrequency',fc);
```

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.

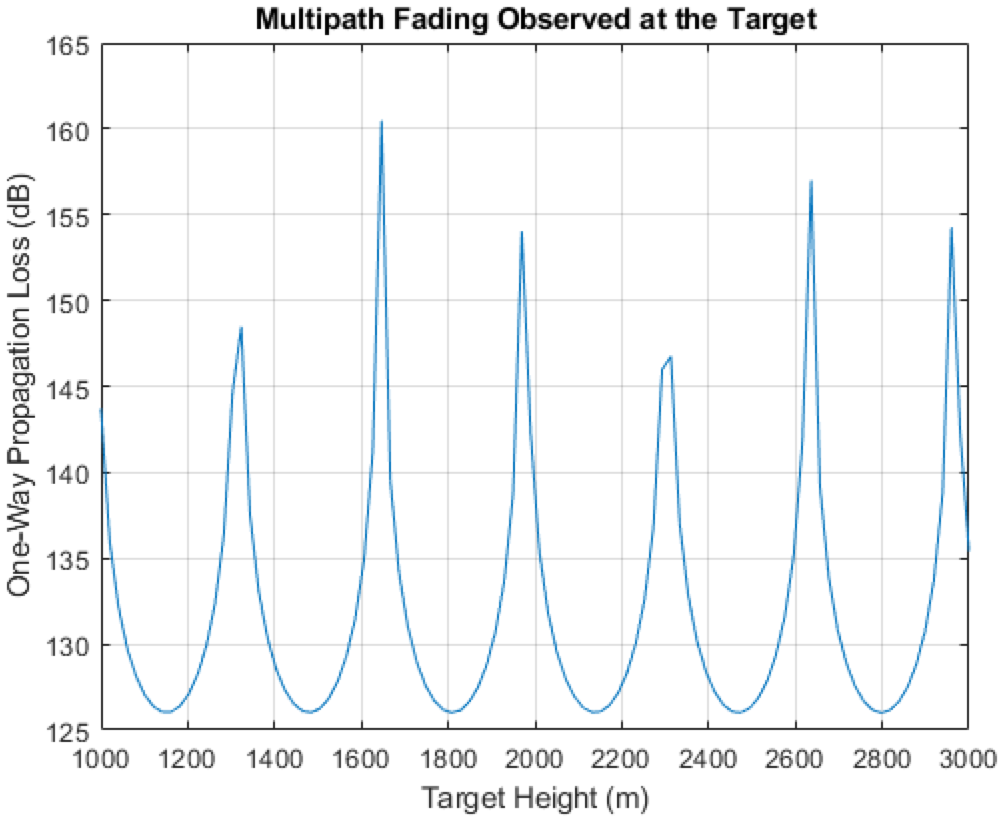
The narrowband channel in the preceding section was shown to be very sensitive to multipath fading. Slight changes in the target's height resulted in considerable signal losses.

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;
```



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.

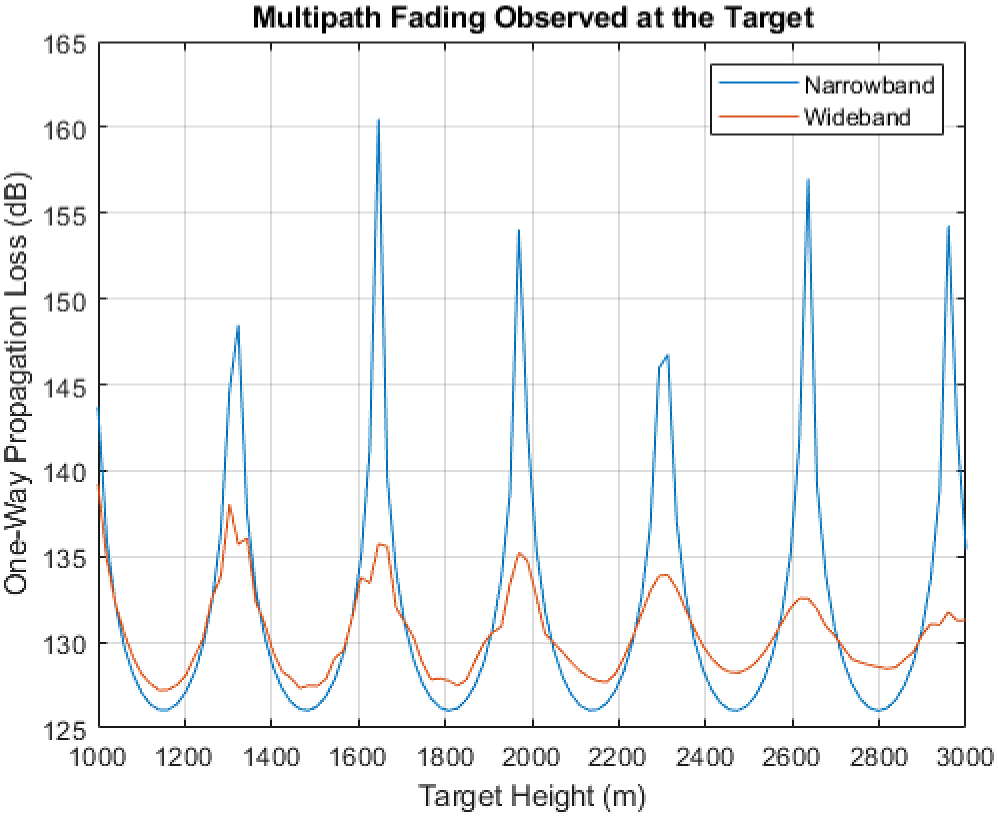
```
bw = 0.08*fc;
pulse_width = 1/bw;
fs = 2*bw;

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

```
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

```
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

```
function [delay, dop] = helperPlotDelayAndDopplerShift(wav,y2ray,Fs)
% Plot transmitted and propagated pulse
t = 1e6*(0:numel(wav)-1)'/Fs;
subplot(211)
```

```
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

```
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,
% 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
```

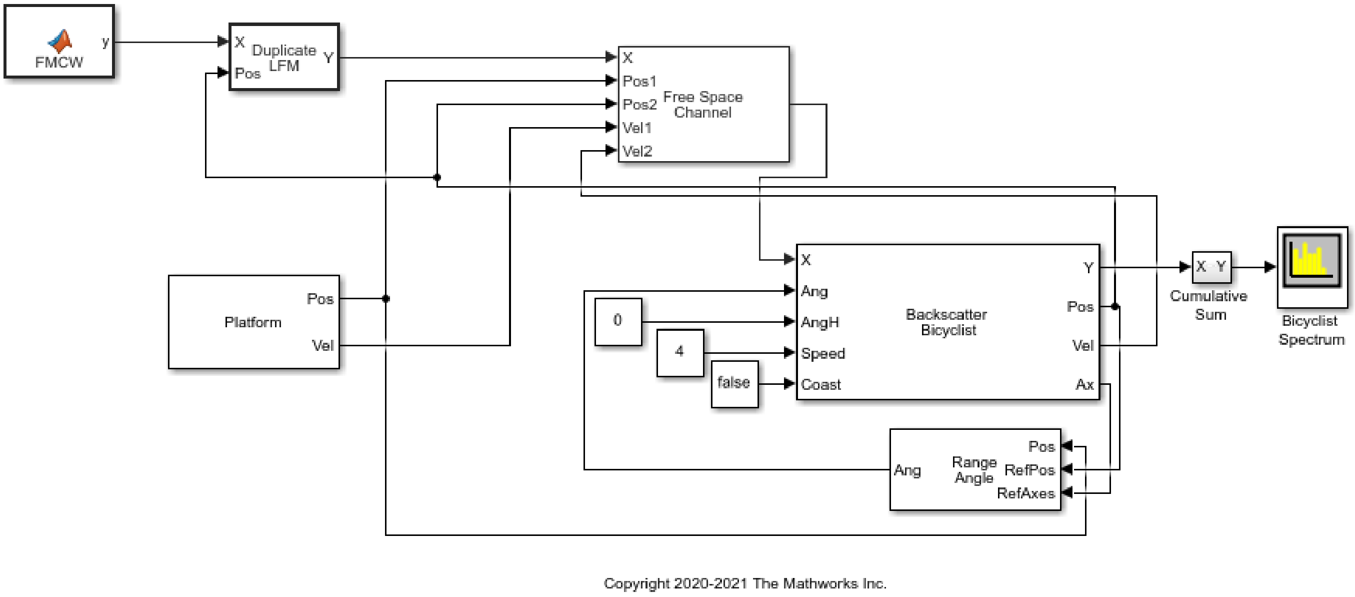
helperPlotCombinedEnvLosses

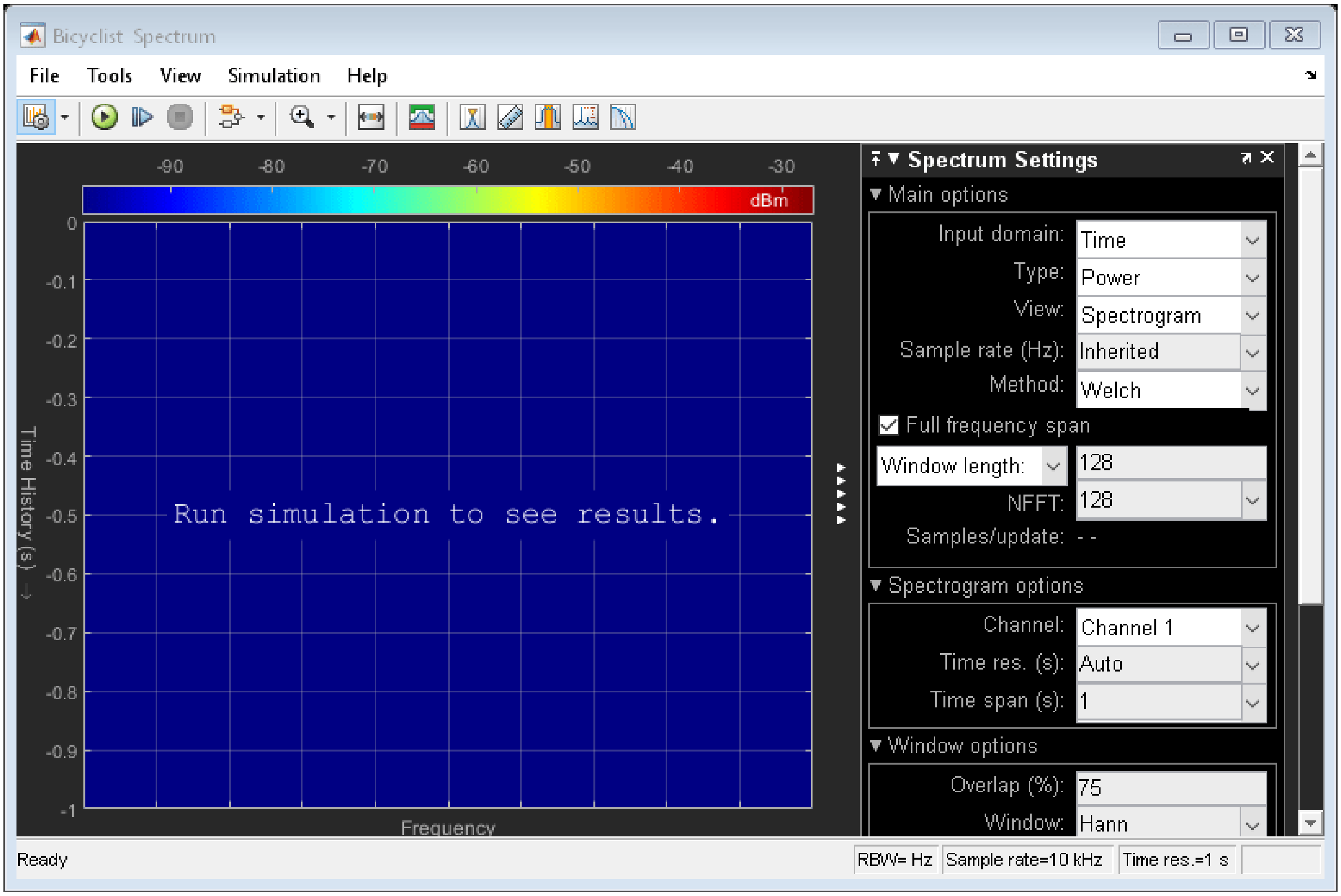
```
function helperPlotCombinedEnvLosses(Rkm,freq,anht,tgtht,PLdBNorm)
% Plot combined losses for different heights and ranges
hP = pcolor(Rkm,tgtht,PLdBNorm);
set(hP, 'EdgeColor', 'none');
title([num2str(freq/1e9) ' GHz S-Band Radar'])
subtitle([num2str(round(anht)) ' m above water'])
xlabel('Range (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');
```





Run simulation to see results.

Input domain: Time
Type: Power
View: Spectrogram
Sample rate (Hz): Inherited
Method: Welch
☑ Full frequency span
Window length: 128
NFFT: 128
Samples/update: - -

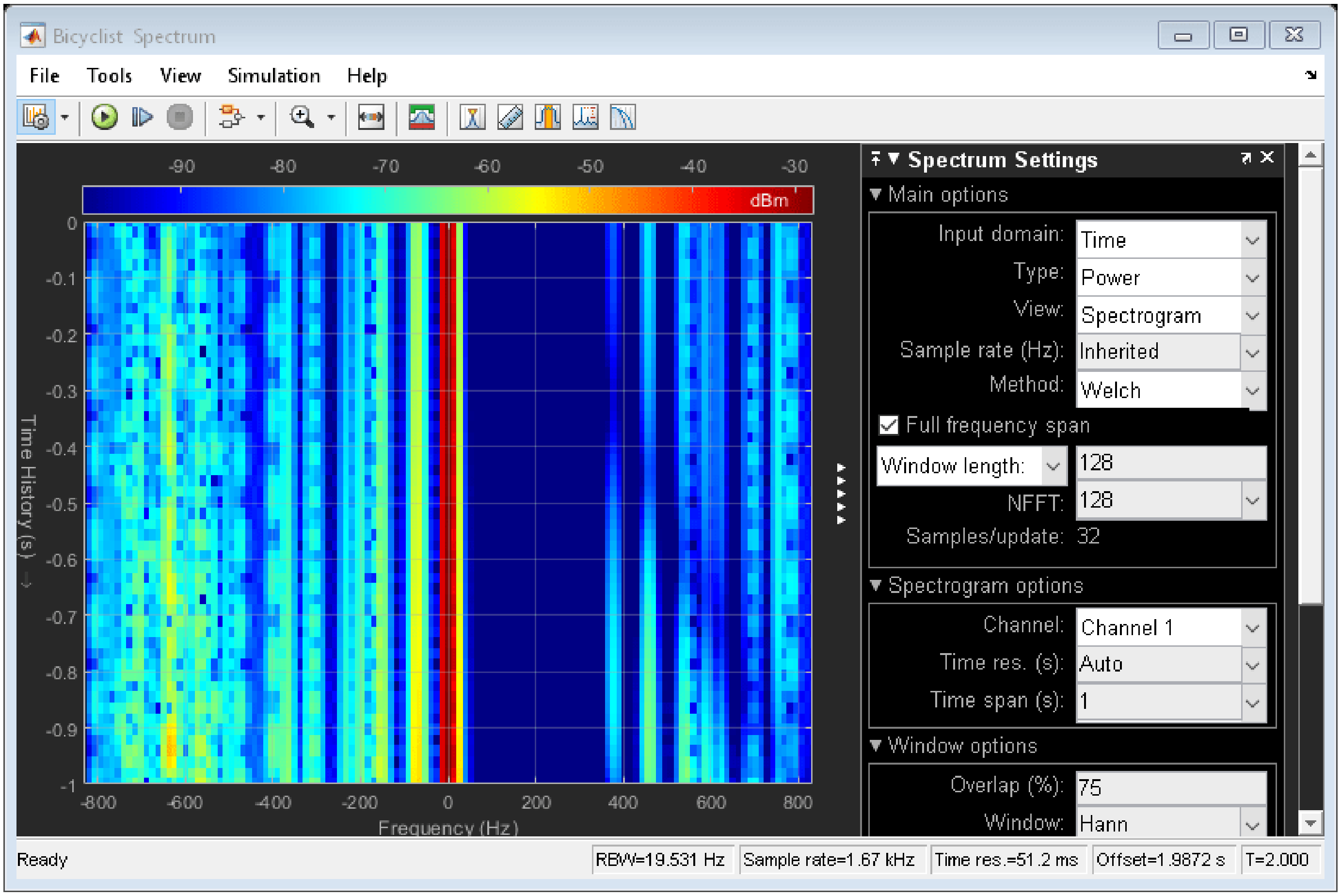
Channel: Channel 1
Time res. (s): Auto
Time span (s): 1

Overlap (%): 75
Window: Hann

Ready | RBW= Hz | Sample rate=10 kHz | Time res.=1 s

Display the micro-Doppler spectrum.

```
sim('BicyclistMicrospectrumExample');
```



Spectrum Settings

Main options

Input domain: Time
Type: Power
View: Spectrogram
Sample rate (Hz): Inherited
Method: Welch
☑ Full frequency span
Window length: 128
NFFT: 128
Samples/update: 32

Spectrogram options

Channel: Channel 1
Time res. (s): Auto
Time span (s): 1

Window options

Overlap (%): 75
Window: Hann

Ready | RBW=19.531 Hz | Sample rate=1.67 kHz | Time res.=51.2 ms | Offset=1.9872 s | T=2.000

```
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)
```

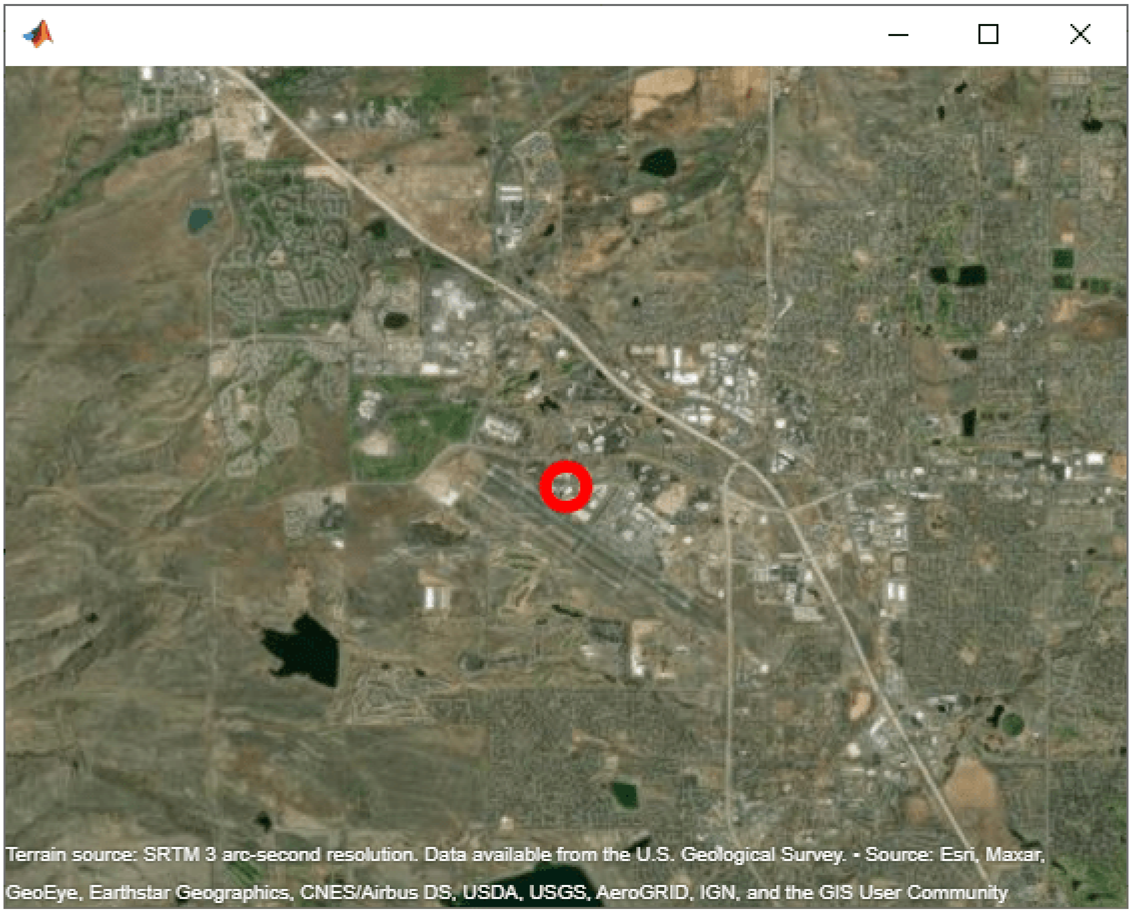
Define the radar ground location as the Rocky Mountain Metropolitan Airport in Broomfield, Colorado, USA. The radar is mounted on a tower 10 meters above the ground. The radar altitude is the sum of the ground elevation and the radar tower height referenced to the mean sea level (MSL).

```
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.";
[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);
```


Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, Maxar, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

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.

```
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)
```

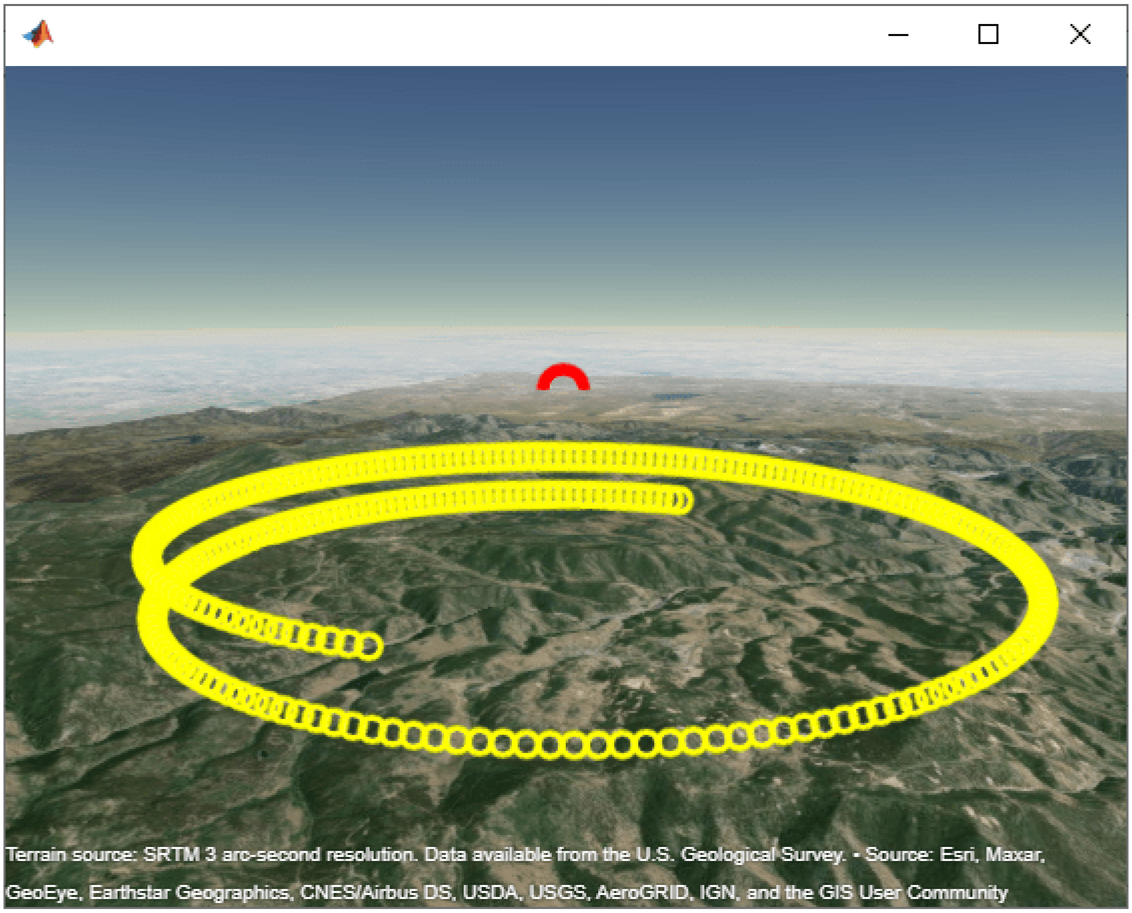
```
% Convert from polar coordinates to Cartesian East, North, Up (ENU).
[X,Y] = pol2cart(deg2rad(azs),r);

% 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)
```



Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. • Source: Esri, Maxar, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

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

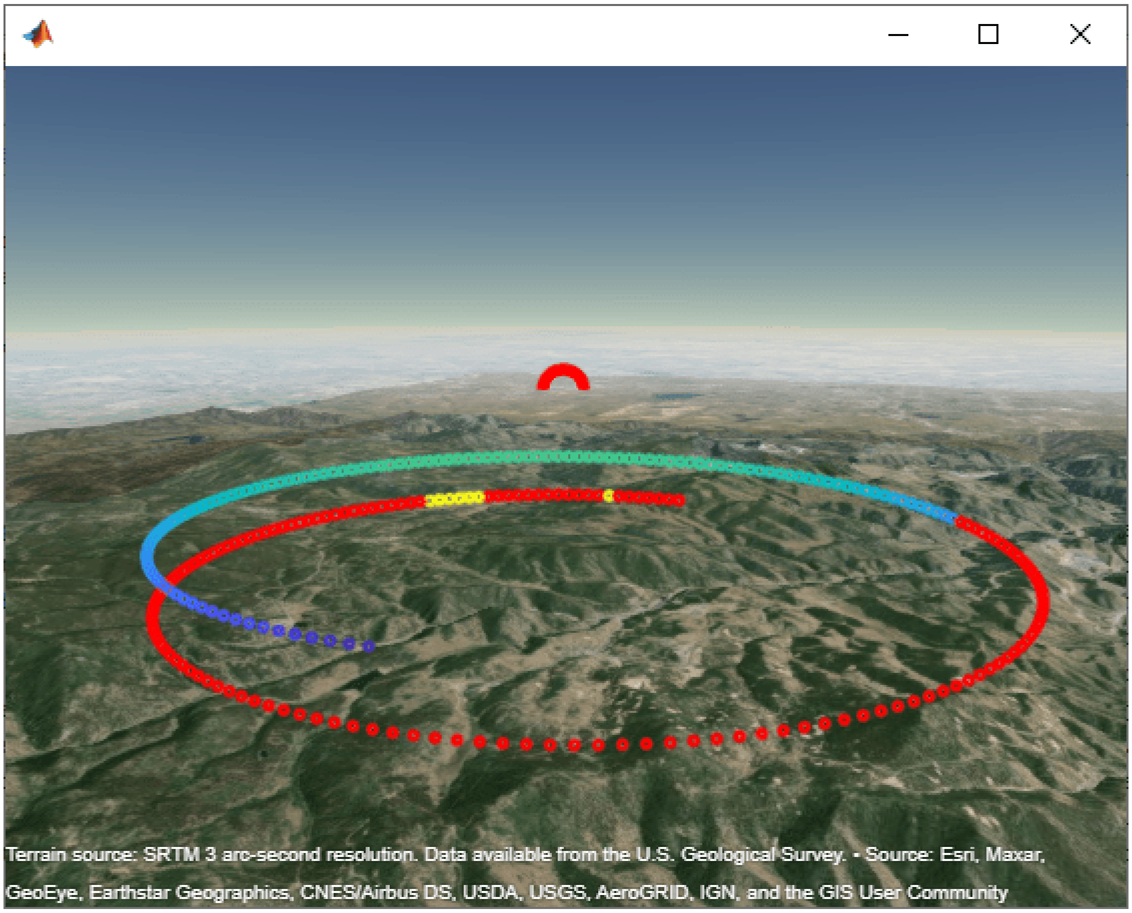
```
tsnr = -inf(size(t));
F = zeros(size(t));
trange = zeros(size(t));
isVisible = false(size(t));
idx = 1;

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) = norm(tpos);
        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, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

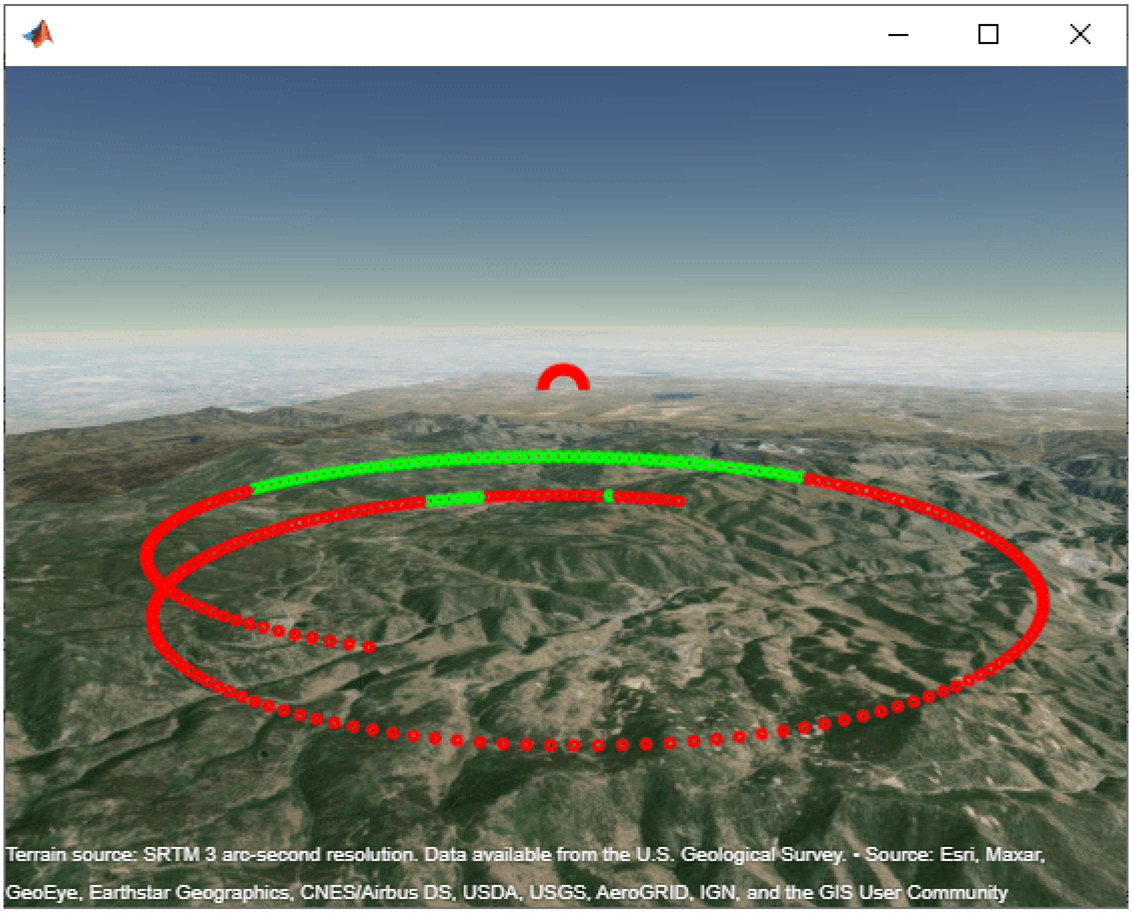
As the aircraft performs the corkscrew maneuver, the SNR of the received signal might vary, as shown in the figure. The radar has an unobstructed view of the aircraft if there is a line-of-sight path. The red portion of the trajectory indicates that there is no line-of-sight path between the aircraft 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
```


Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. • Source: Esri, Maxar, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

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.

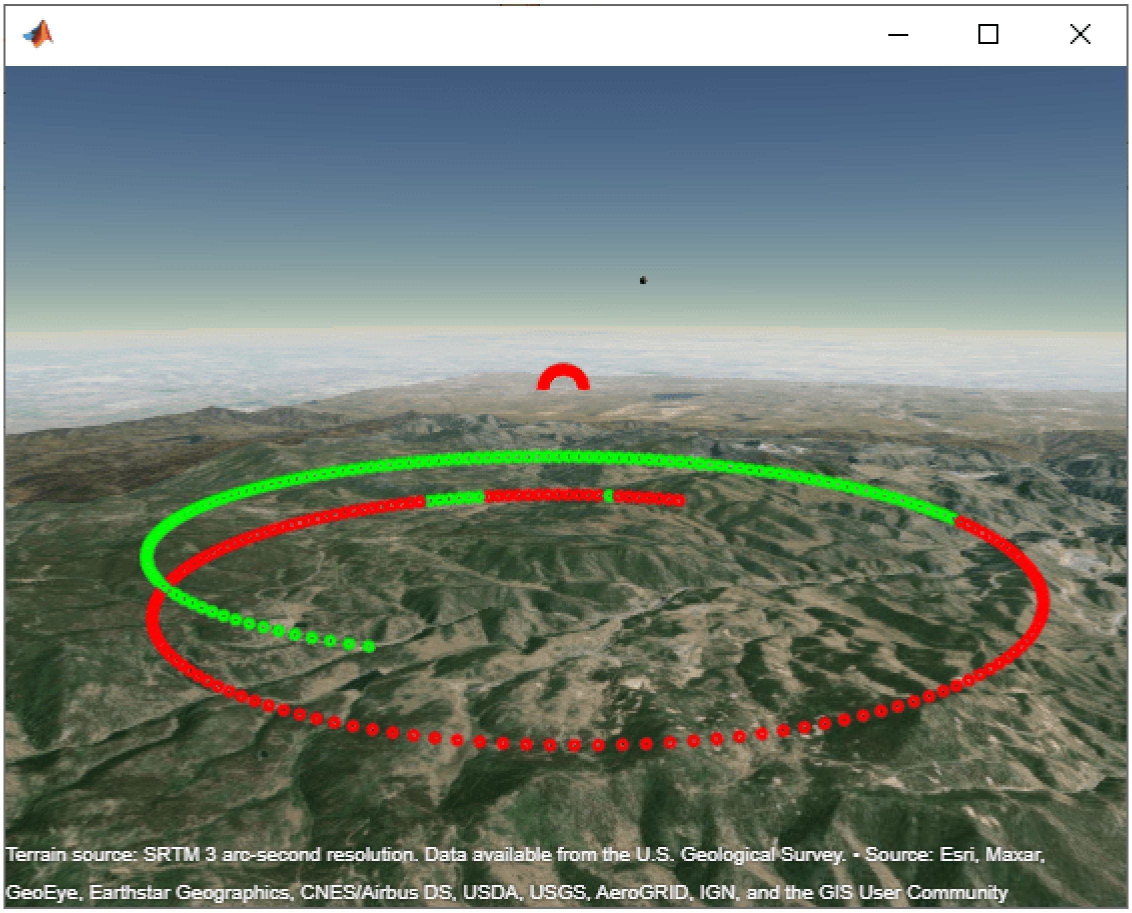
The above list, while in no way exhaustive, shows some of the tradeoffs in the design of an terminal airport surveillance system. For this example, increase the peak power. Since this is a ground-based system, increasing the power is not expected to be too burdensome. Additionally, other airport radars 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.

```
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.

```
for m = 1:numel(tsnr)
    if isdetectable_32p(m)
        set(hsnr(m),'Color','g');
    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, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

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.

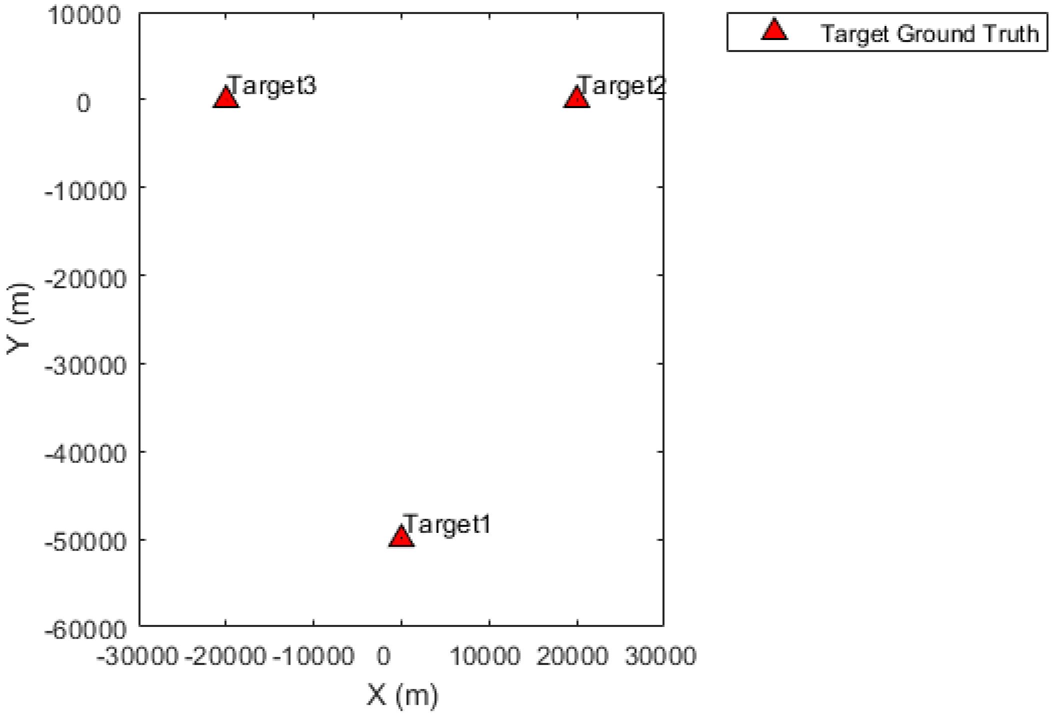
At early stages of a project, as design tradeoffs are being explored, modeling at the radar equation level may be adequate. As a project progresses, it will be necessary to increase the level of model fidelity, moving from a statistical level to signal level simulation. In addition, the length of a scenario 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'});
```



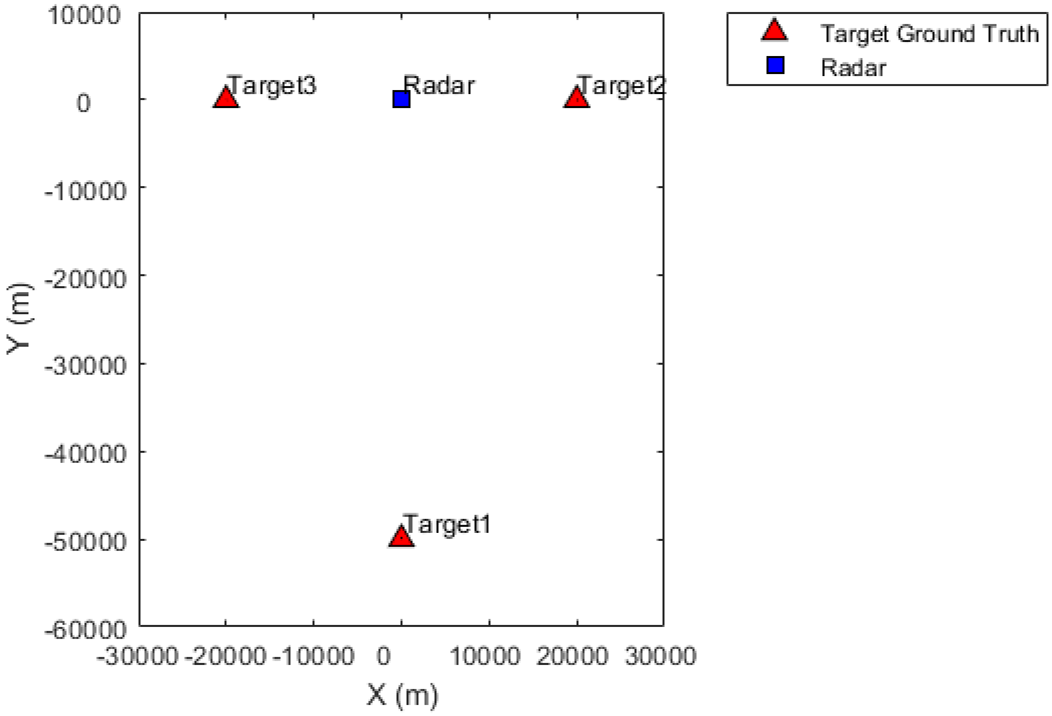
Define Radar for Detection Generation

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
```

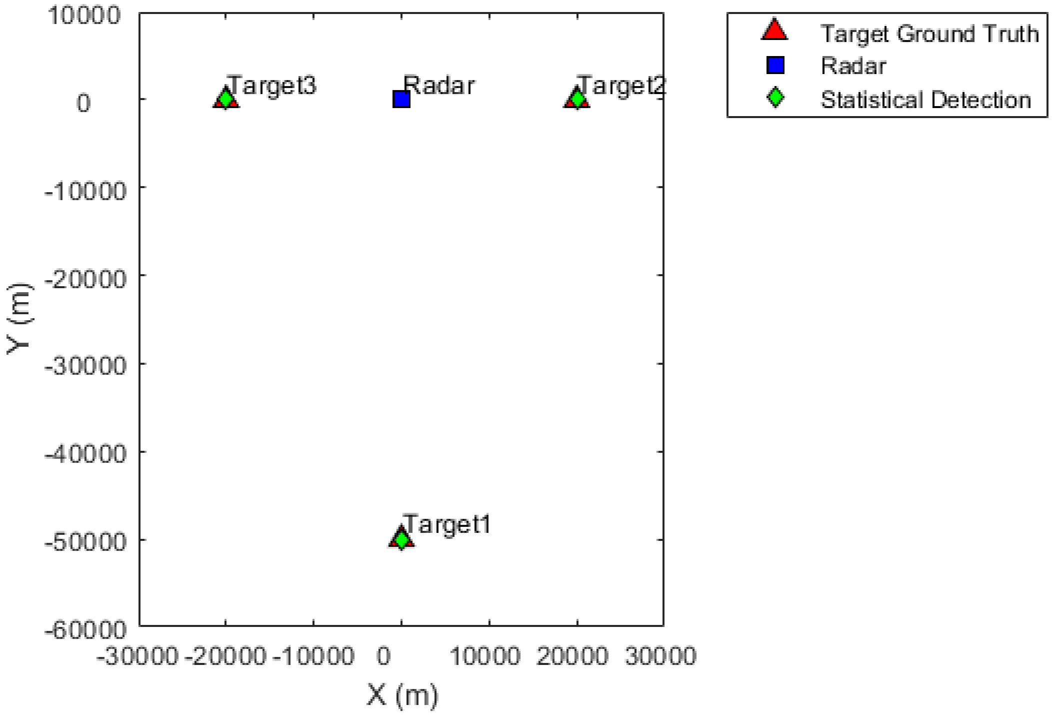
```
sensor = radarDataGenerator(1, 'Rotator', ...
    'DetectionProbability', 0.99, ...
    'UpdateRate', updaterate, ...
    'MountingLocation', [0 0 -15], ...
    'MaxAzimuthScanRate', scanrate, ...
    'FieldOfView', fov, ...
    'AzimuthResolution', fov(1));

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
```

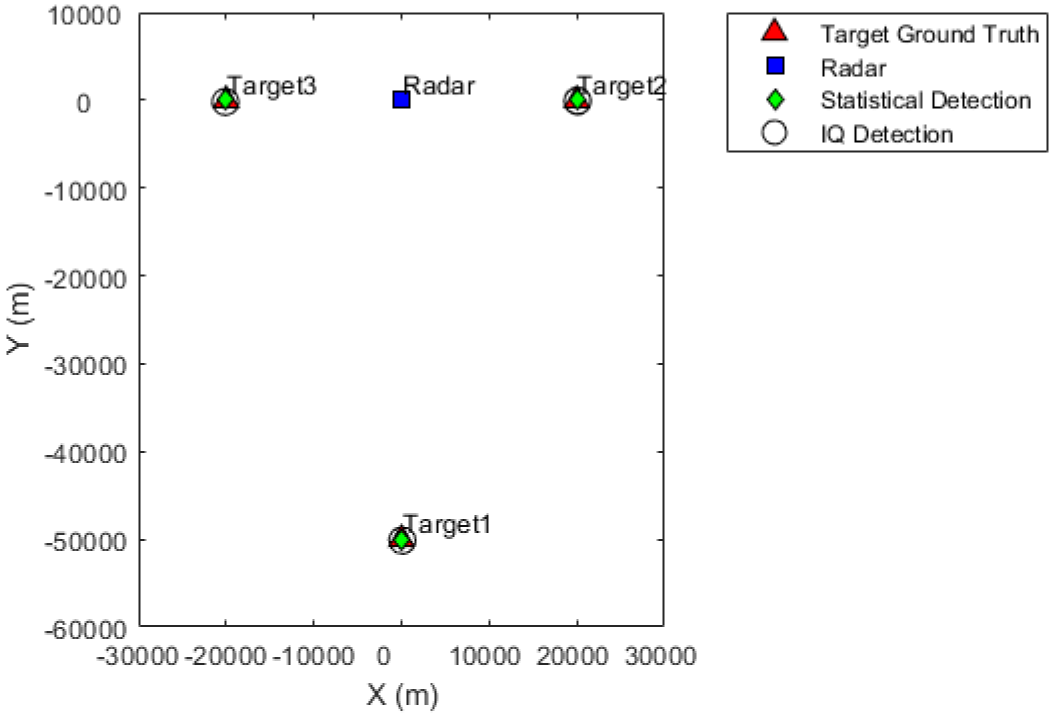
Notice that the configuration of the `sensor_iq` variable is closer to a physical system. `sensor_iq` produces IQ signals that you can then process. For this example, implement a simple threshold detector to generate detections.

```
% 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.

```
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.

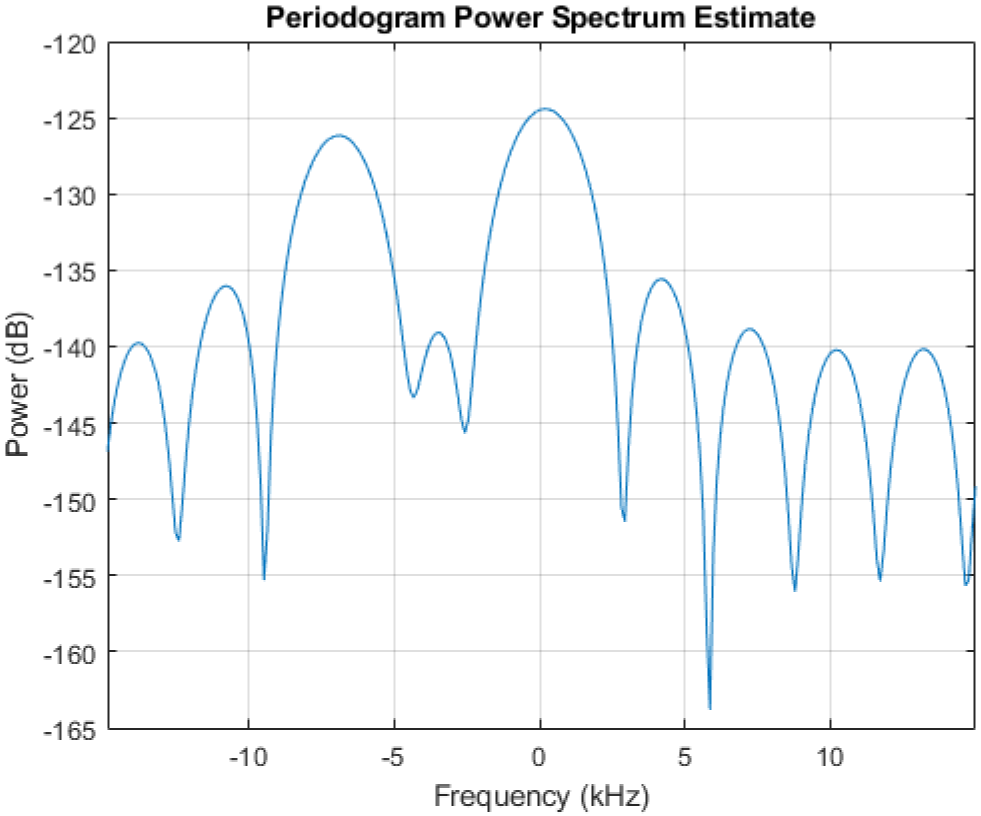
```
speed_vec = dop2speed(f1,lambda)/2;
```

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');
```



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)
```

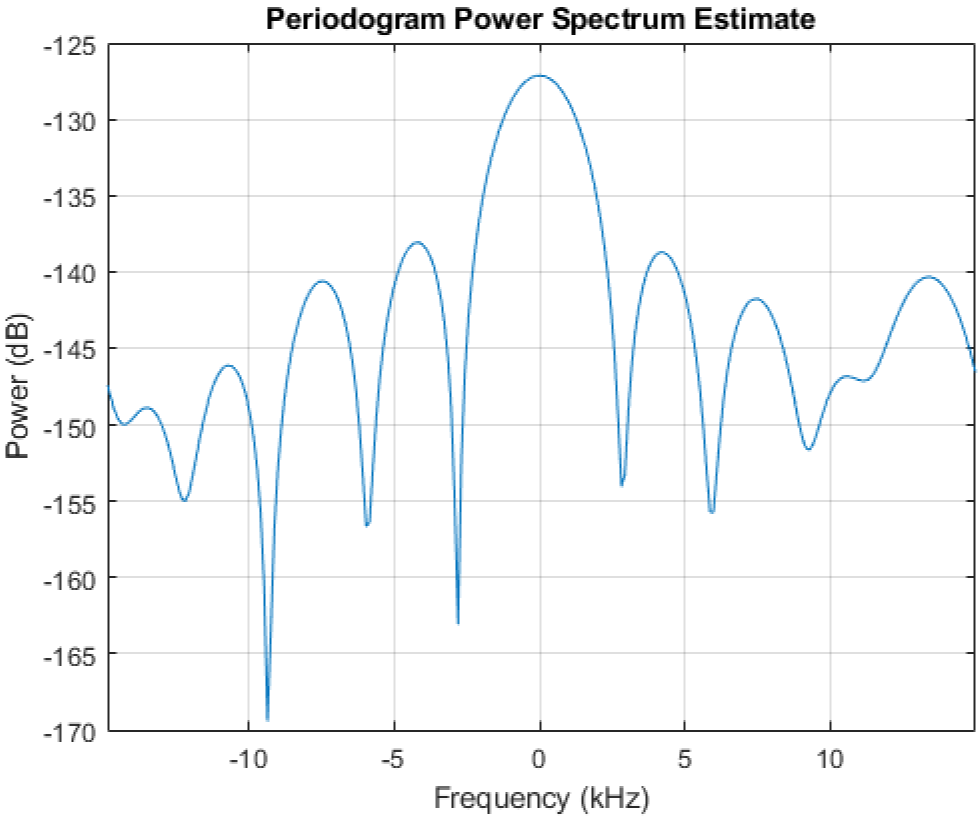
```
sp1 = 2×1

 -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');
```



```
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.

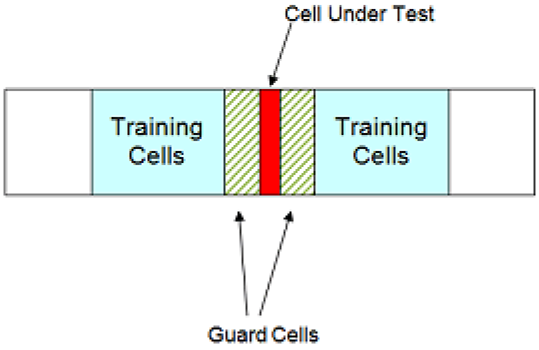
Cell Averaging CFAR Detection

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.

```
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.

```
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.

```
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.

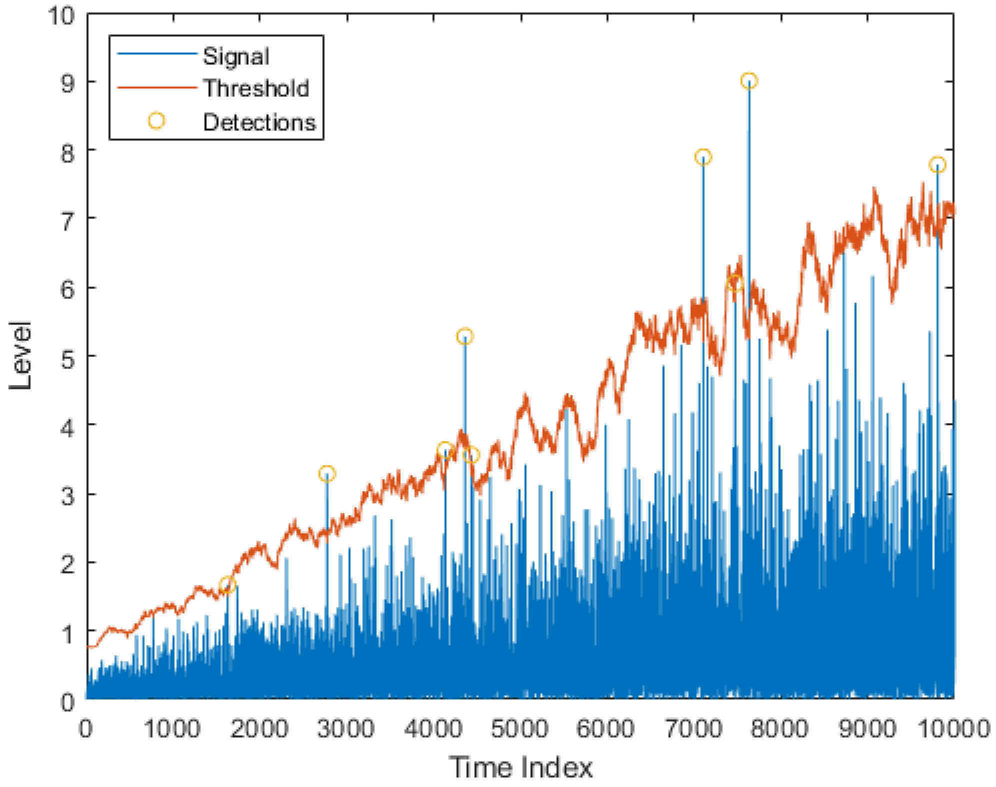
```
rs = RandStream('mt19937ar','Seed',2010);
Npoints = 1e4;
rsamp = randn(rs,Npoints,1)+1i*randn(rs,Npoints,1);
ramp = linspace(1,10,Npoints)';
xRamp = abs(sqrt(npower*ramp./2).*rsamp).^2;
```

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
```

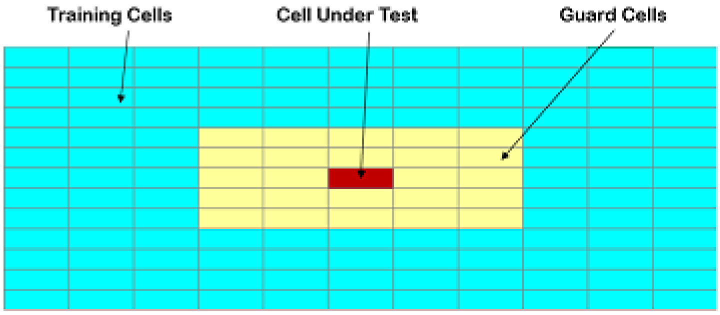
Note that the resulting false alarm rate cannot meet the requirement. However, using the CFAR detector with a custom threshold factor, we can obtain the desired false alarm rate.

```
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.



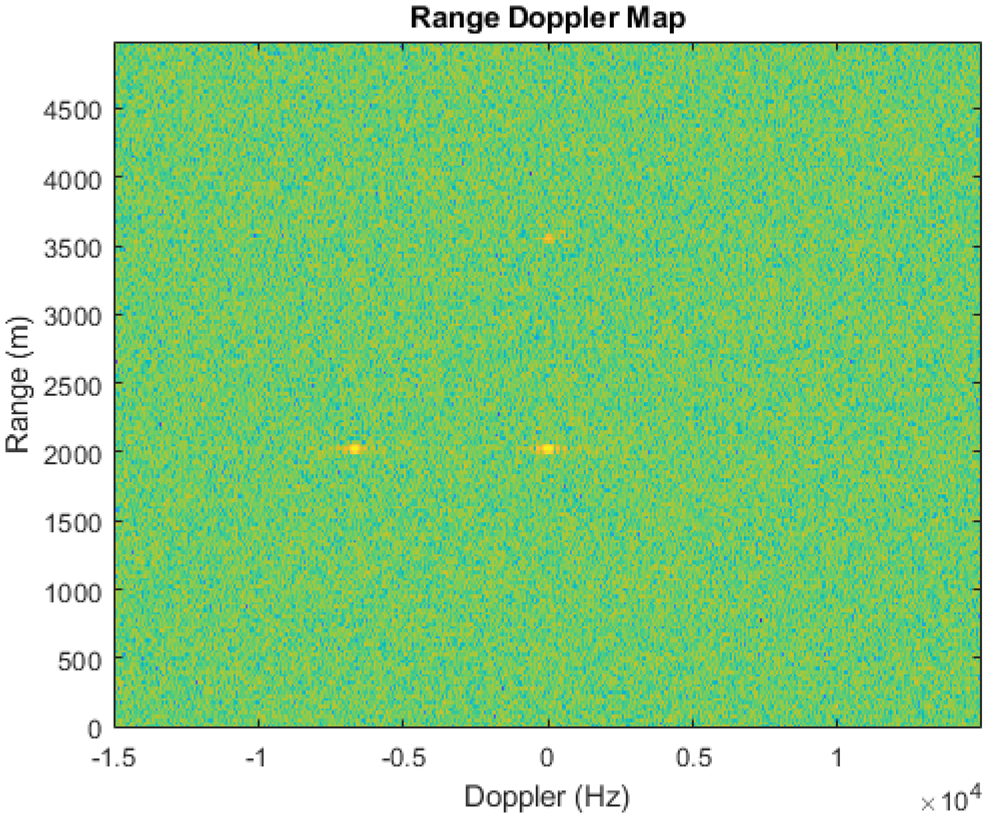
In the figure above, the guard band size is [2 2] and the training band size is [4 3]. The size indices refer to the number of cells on each side of the CUT in the row and columns dimensions, respectively. The guard band size can also be defined as 2, since the size is the same along row and column dimensions.

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;
```

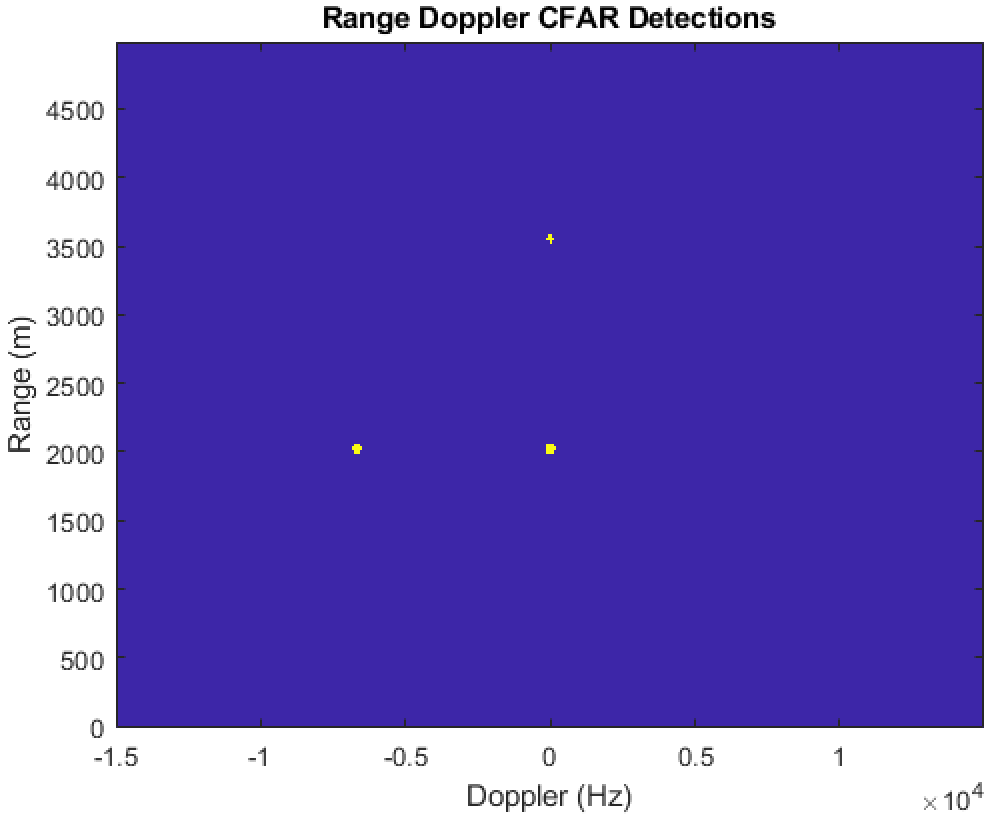


Use CFAR to search the range-Doppler space for objects, and plot a map of the detections. Search from -10 to 10 kHz and from 1000 to 4000 m. First, define the cells under test for this region.

```
[~,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)
```



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

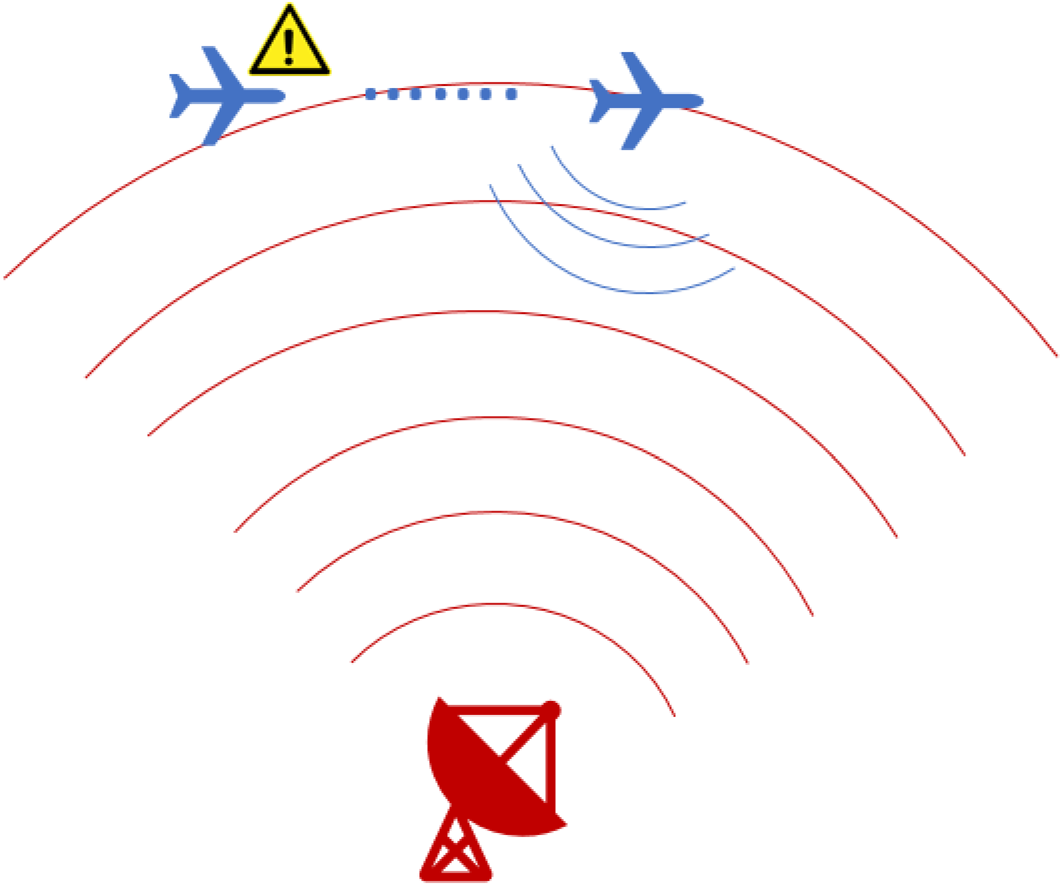
Waveform Parameter Extraction from Received Pulse

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 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.



In this example, we simulate a scenario where a ground surveillance radar and an airplane with an RWR present. 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 sub-banded signal is fed to the signal processor. Beam steering is applied towards the direction of arrival of this sub-banded 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);
```

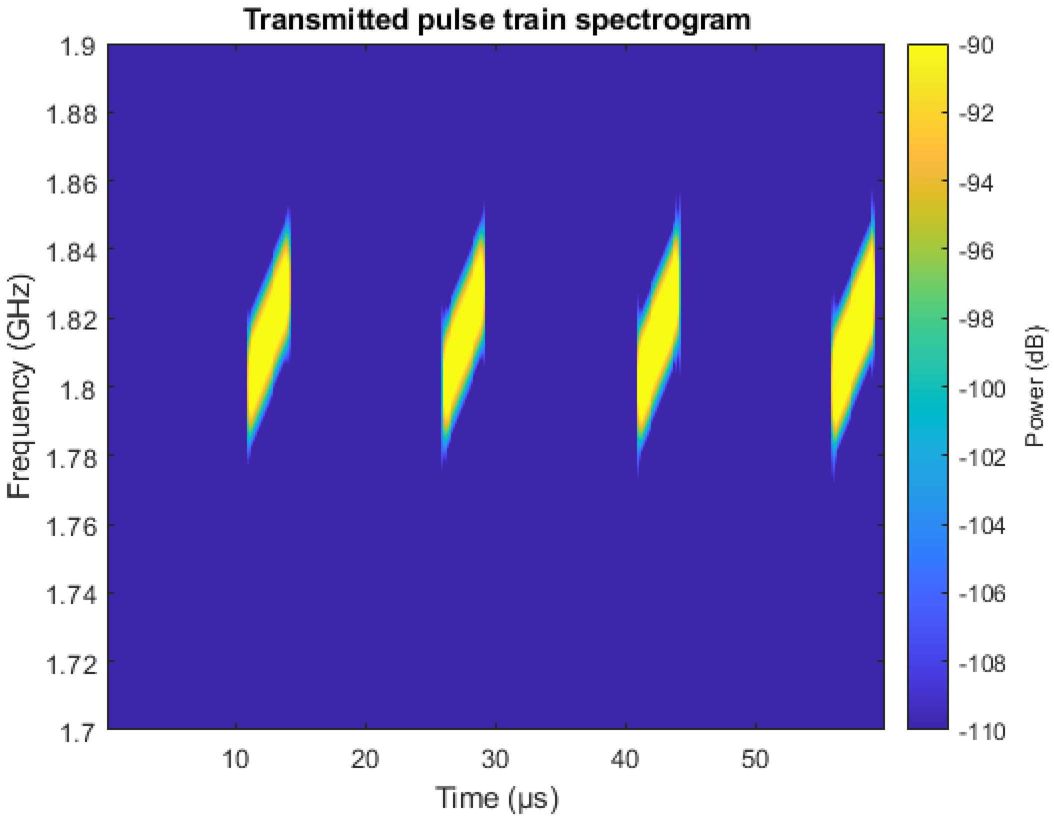
The transmit antenna of the radar is a 8x8 uniform rectangular phased array, having a spacing of $\lambda/2$ between its elements. The signal propagates from the radar to the aircraft and is intercepted and analyzed by the RWR. For simplicity, the waveform is chosen as a linear FM waveform with a peak 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 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 a 4-pulse train arrived 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 10x10 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.

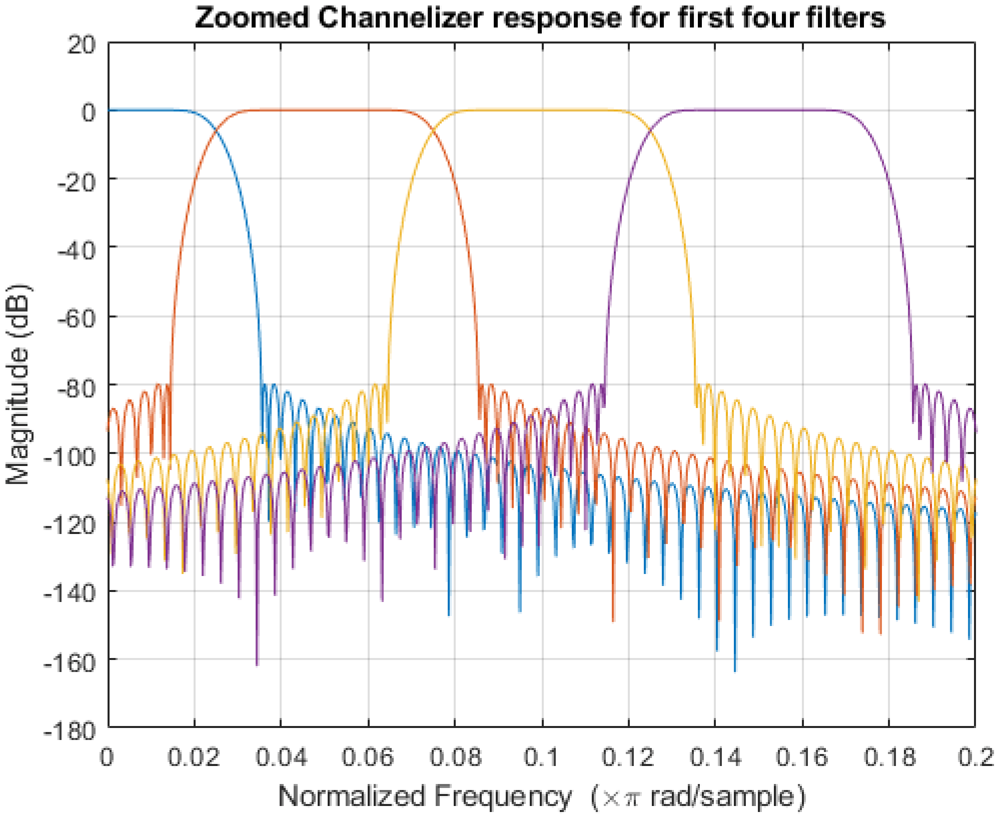
```
% Truncate the received data
truncTime = 50e-6;
truncInd = round(truncTime*fs);
yr = yr(1:truncInd,:);
```

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 presents. In this example, the signal is divided into sub-bands of 100 MHz bandwidth. An added benefit for such operation is that instead of sampling the entire bandwidth covered by the RWR, the signal in each subband can be down-sampled to a sampling frequency of 100 MHz.

```
% Define the bandwidth of each frequency sub-band
stepFreq = 100e6;

% Calculate number of sub-bands and configure dsp.Channelizer
numChan = fs/stepFreq;
channelizer = dsp.Channelizer('NumFrequencyBands', numChan, 'StopbandAttenuation', 80);
```

The plot below shows the first four band created by the filter bank.

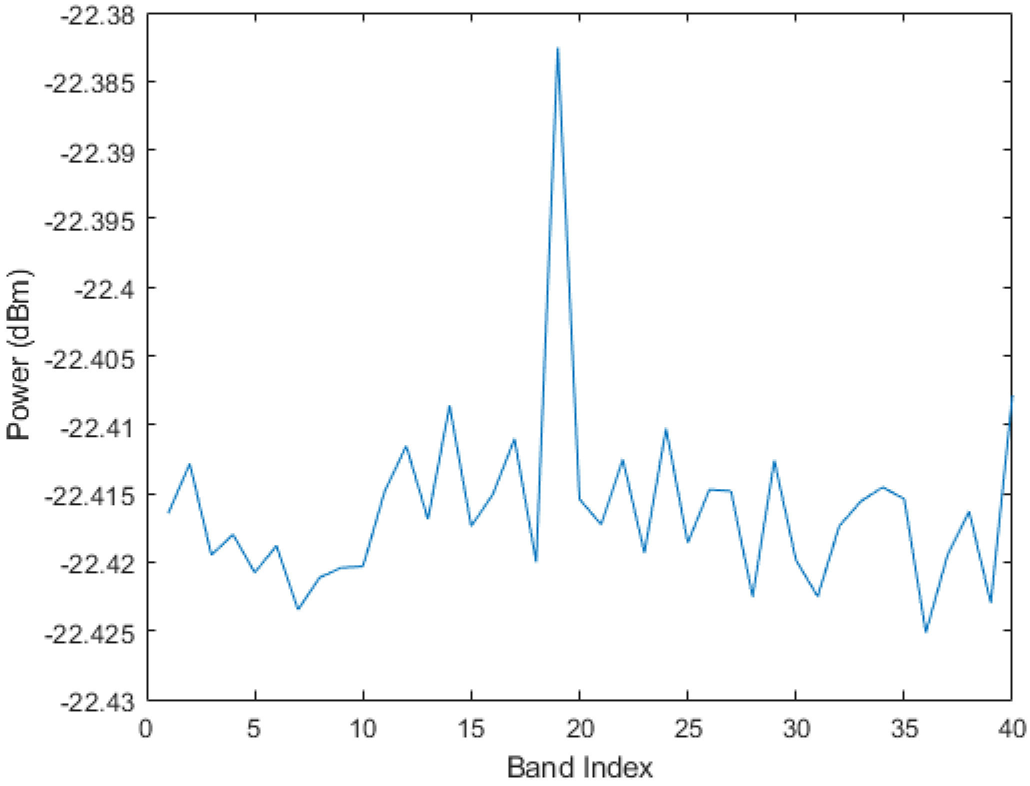
```
% Visualize the first four filters created in the filter bank of the
% channelizer
freqz(channelizer, 1:4)
title('Zoomed Channelizer response for first four filters')
xlim([0 0.2])
```



```
% Pass the received data through the channelizer
subData = channelizer(yr);
```

The received data, `subData`, has 3 dimensions. The first dimension represents the fast-time, the second dimension represents the sub-bands, and the third dimension corresponds to the receiving elements of the receiving array. For the RWR's 10x10 antenna configuration used in this example, we 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 power are summed across these elements to enhance the signal-to-nose ratio (SNR) and get a better estimates 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)');
```

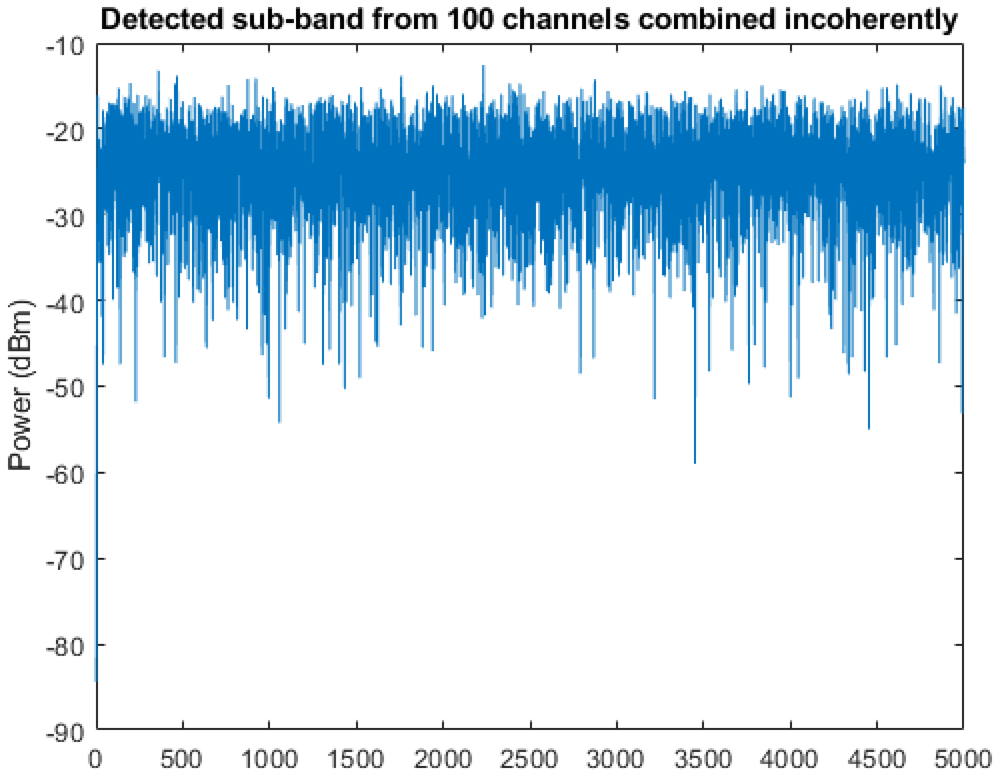


```
% Find the sub-band with maximum power
[~,detInd] = max(subbandPow);
```

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 sub-band 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;
```

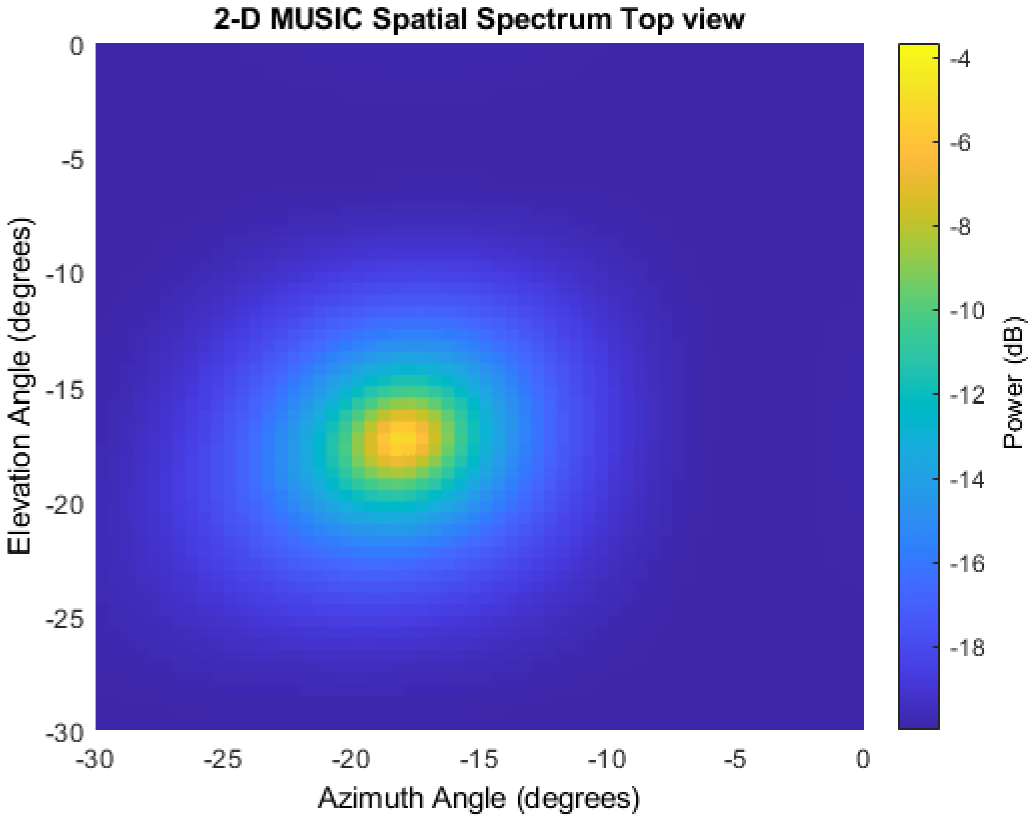
The subData is now a two-dimensional matrix. The first dimension represents fast-time samples and the second dimension is the data across 100 receiving antenna channels. The detected sub-band starting frequency is calculated to find the carrier frequency of the detected signal.

The next step for the RWR is to find the direction from which the radio waves are arriving. This angle of arrival information would be used to steer the receive antenna beam in the direction of the emitter, and locate the emitter on the ground using single base-line 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, thus help the waveform parameter extraction.

Assume that ground plane is flat and parallel to the xy-plane of the coordinate system. such, 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]);
```



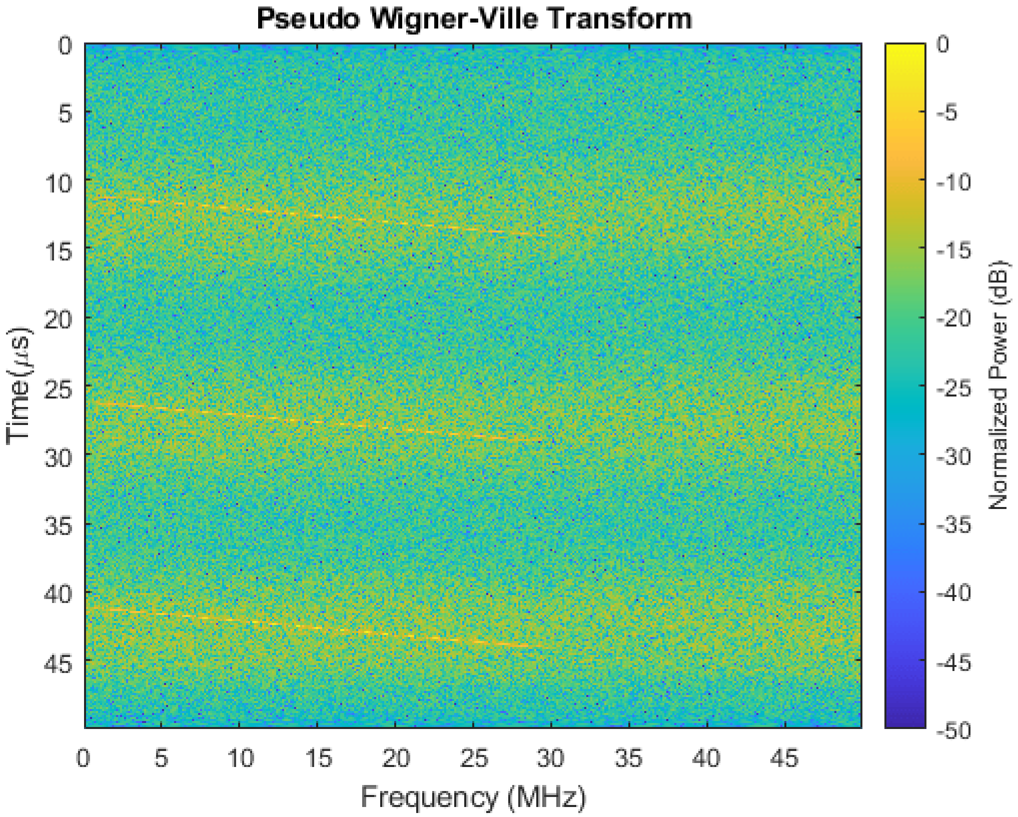
The figure clearly shows the location of the emitter.

```
% 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 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 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')
```



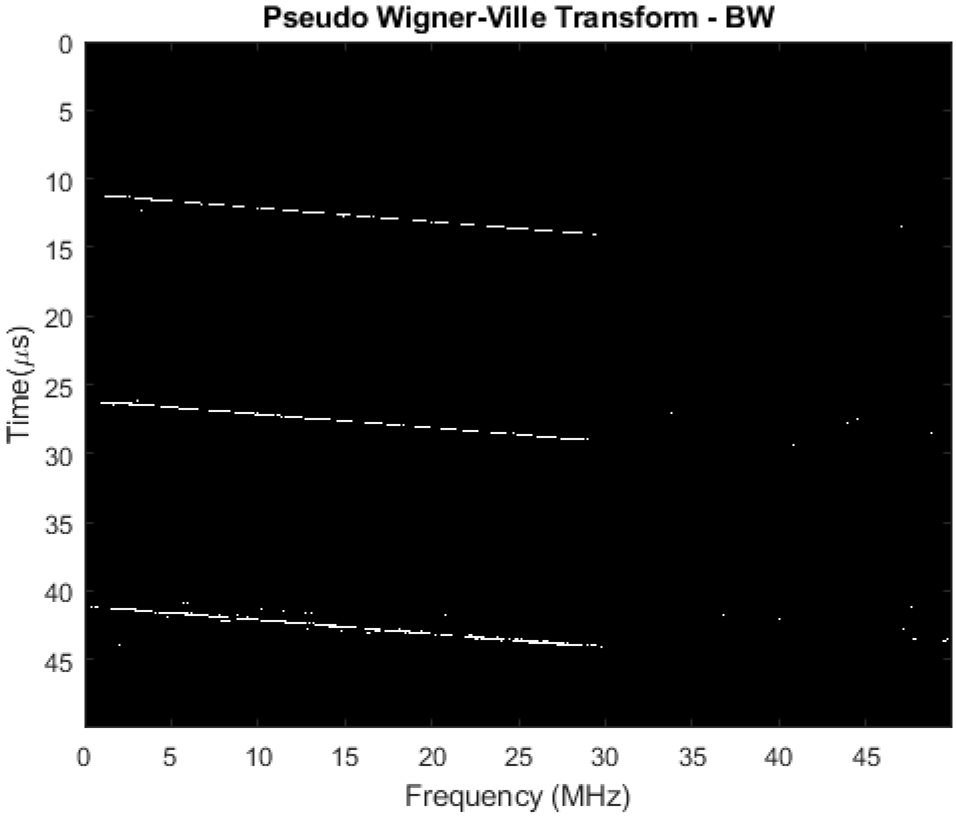
Using human eyes, even though the resulting time frequency representation is noisy, it is not too hard to separate the signal from the background. Each pulse appears as a line in the time frequency plane. Thus, using beginning and end of the time-frequency lines, we can derive the pulse width and the bandwidth of the pulse. Similarly, the time between lines from different pulses gives us the pulse repetition interval.

To do this automatically without relying on human eyes, we use Hough transform to identify those lines from the image. The Hough transform can perform well in the presence of noise, and is an enhancement to the time-frequency signal analysis method.

To use Hough transform, it is necessary to convert the time frequency image into a binary image. Next code snippet performs some data smoothing on the image and then use `imbinarize` to do the conversion. The conversion threshold can be modified based on the signal-noise characteristics of the receiver and the operating environment.

```
% 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')
```

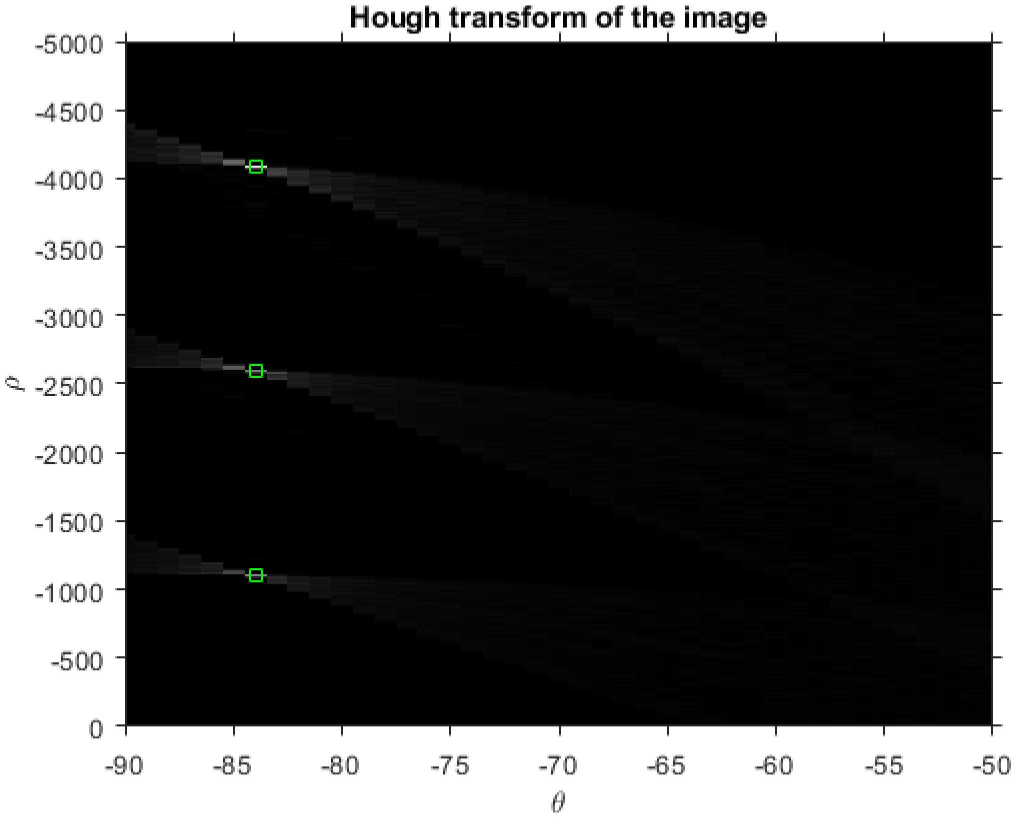


Using Hough transform, the binary pseudo Wigner-Ville image are first transformed to peaks. This way, instead of detecting the line in an image, we just need to detect a peak in an image.

```
% 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')
```

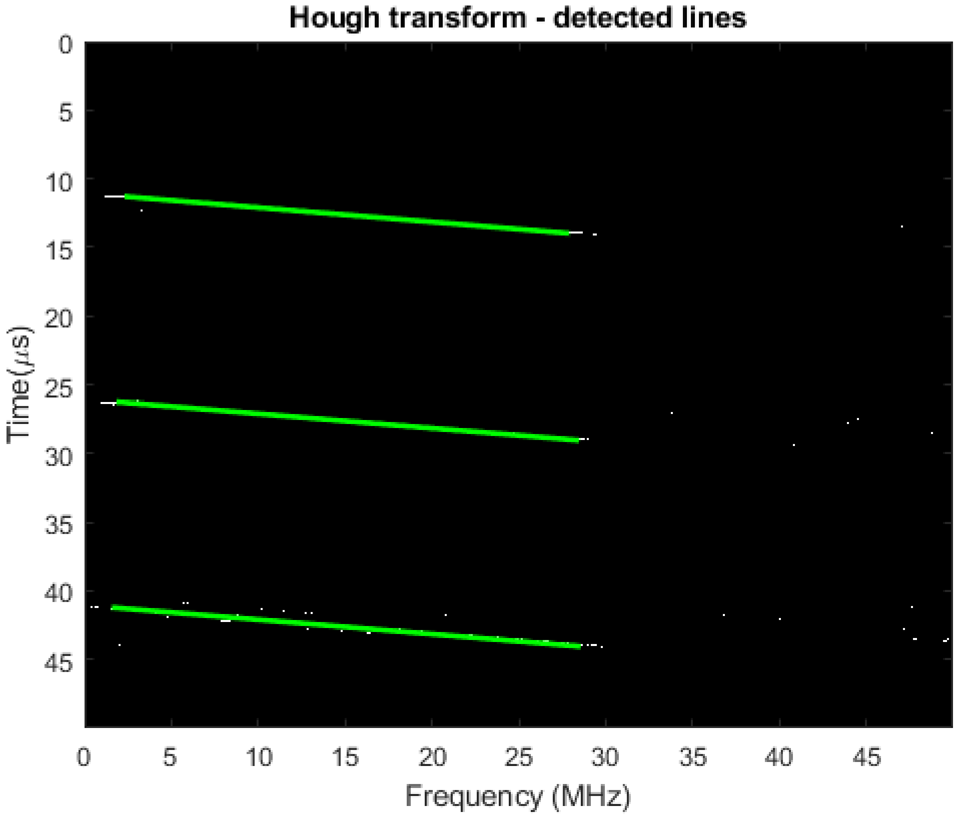
The peak positions are extracted using houghpeaks.

```
% 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, `houghlines` can reconstruct the lines in the original binary image. Then as discussed earlier, the beginning and the end of these lines help us 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
```



```
% 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 below. 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 demo 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] Daniel L. Stevens, Stephanie A. Schuckers, *Detection and Parameter Extraction of Low Probability of Intercept Radar Signals using the Hough Transform* . Global Journal of Research In Engineering Vol 15 Issue 6, Jan. 2016

Lidar and Radar Fusion in an Urban Air Mobility Scenario

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

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

Create an urban air mobility scenario

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

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

Next, you add a few UAVs to the scenario.

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

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

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

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

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

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

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

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

Define UAV sensor suite

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

- Range resolution: 3 cm
- Maximum range: 100 m
- 360 degrees azimuth span with 0.2° resolution
- 30 degrees elevation span with 2° resolution
- Update rate: 10 Hz
- Mount with a 90° tilt to look down

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

Next you add a radar using the `radarDataGenerator` System object from the Radar Toolbox. To add this sensor to the UAV platform, you need to define a custom adaptor class. The details are shown in the “Simulate Radar Sensor Mounted On UAV” (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. Inspired from the Echodyne EchoFlight UAV radar [2], the radar configuration is selected as:

- Frequency: 24.45 - 24.65 GHz
- Field of view: 120° azimuth 80° elevation
- Resolution: 2 deg in azimuth, 6° in elevation
- Full scan rate: 1 Hz
- Sensitivity: 0 dBsm at 200 m

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

```
% Mount a radar on the quadrotor.
radarSensor = radarDataGenerator("no scanning","SensorIndex",1,...
    "FieldOfView",[120 80],...
    "UpdateRate", 1,...
    'MountingAngles',[0 30 0],...
    "HasElevation", true,...
    "ElevationResolution", 6,...
    "AzimuthResolution", 2, ...
    "RangeResolution", 4, ...
    "RangeLimits", [0 200],...
    'ReferenceRange',200,...
    'CenterFrequency',24.55e9,...
    'Bandwidth',200e6,...
    "TargetReportFormat","Detections",...
    "DetectionCoordinates","Sensor rectangular",...
    "HasFalseAlarms",false,...
    "FalseAlarmRate", 1e-7);

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

Define tracking system

Lidar point cloud processing

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

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

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

```
lidarDetector = helperLidarDetector(scene)
```

```
lidarDetector = 
  helperLidarDetector with properties:

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

Lidar tracker

You use a point target tracker, `trackerJPDA`, to track the lidar bounding box detections. A point tracker assumes that each UAV can generate at most one detection per sensor scan. This assumption is valid because you have clustered the point cloud into cuboids. To setup a tracker, you need to define the motion model and the measurement model. In this example, you model the dynamics of UAVs using an augmented constant velocity model. The constant velocity model is sufficient to track trajectories consisting of straight flight legs or slowly varying segments. Moreover, assume the orientation of the UAV is constant and assume the dimensions of the UAVs are constant. As a result, the track state and state transition equations are $X = \left[x, v_x\ , \ y, \ v_y\ , \ z, \ v_z\ , L, \ W, \ H, q_0, q_1, q_2, q_3\right]$ and

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

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

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

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

```
lidarJPDA = 
  trackerJPDA with properties:

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

                OOSMHandling: 'Terminate'

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

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

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

                     NumTracks: 0
            NumConfirmedTracks: 0
```

Radar tracker

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

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

```
radarPHD = createRadarTracker(radarSensor, egoUAV)
```

```
radarPHD = 
  trackerPHD with properties:

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

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

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

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

Track fusion

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

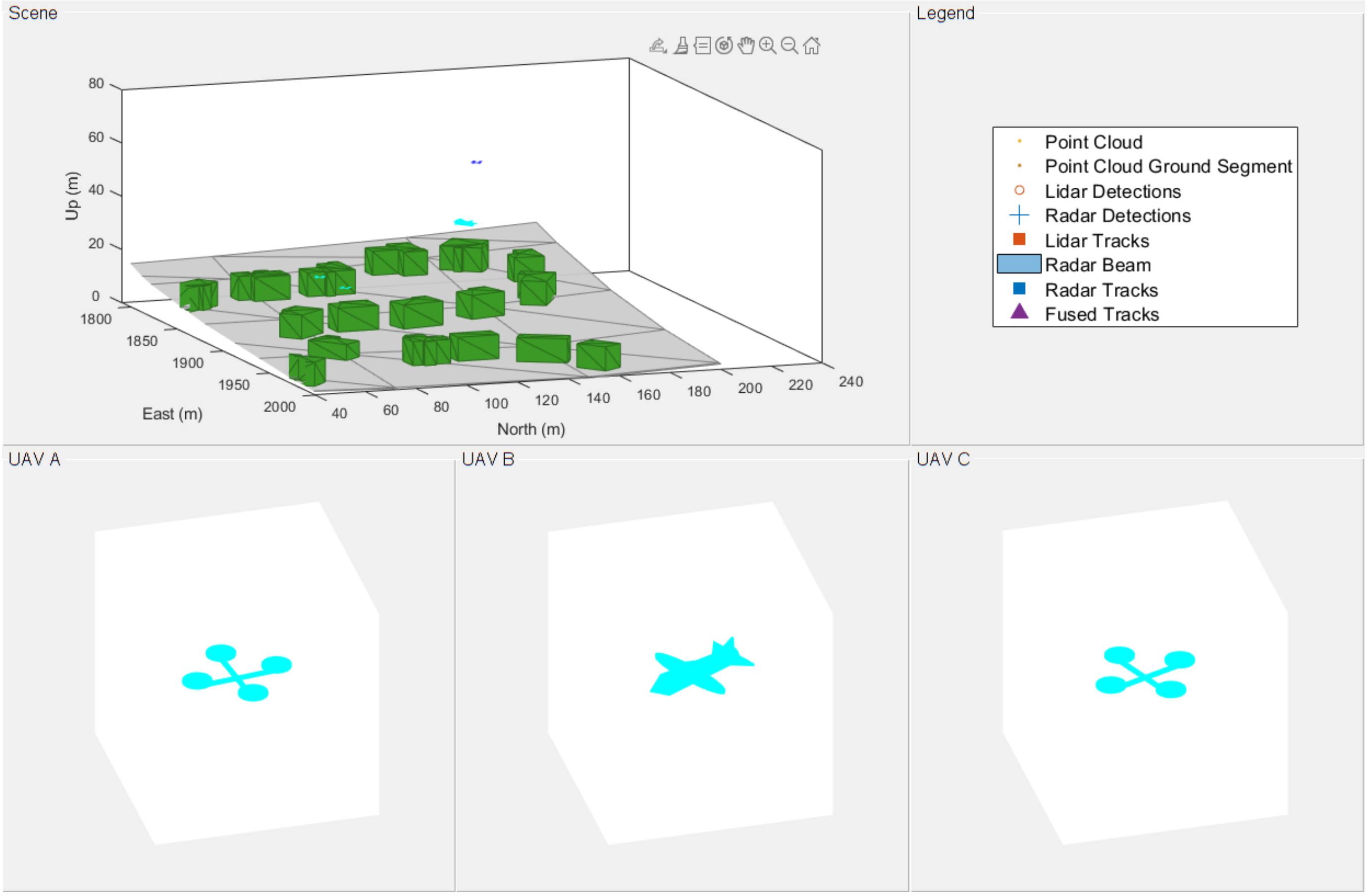
```
radarConfig = fuserSourceConfiguration('SourceIndex',1,...
    'IsInitializingCentralTracks',true);

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

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

Visualization

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

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

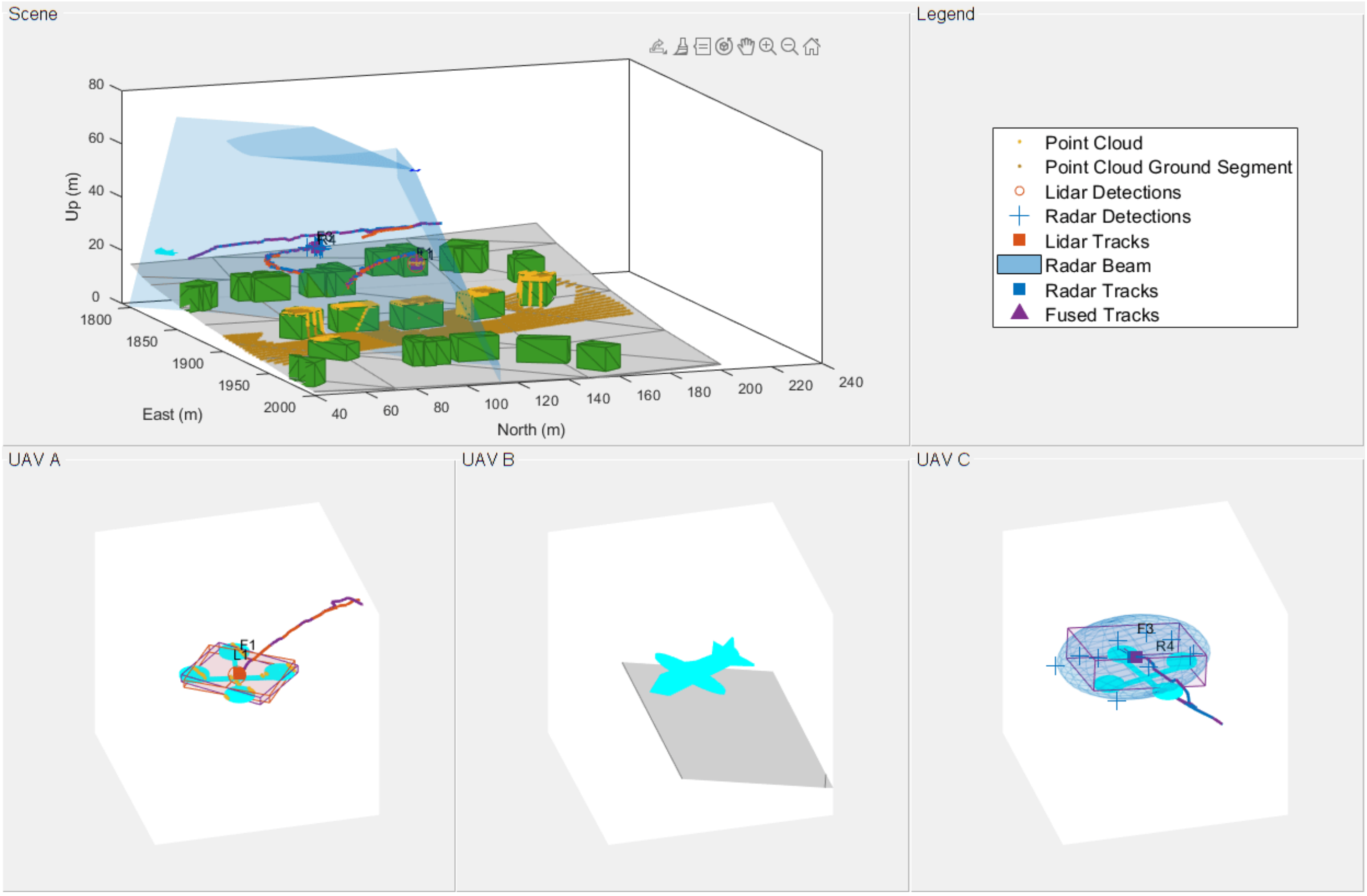
```
viewer = helperUAVDisplay(scene);
```



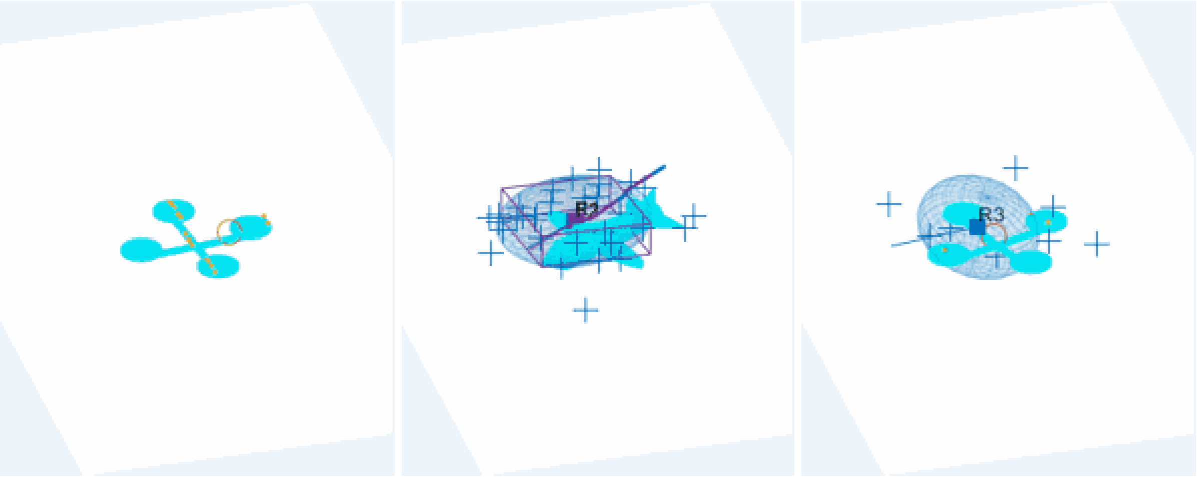
Simulate the scenario

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

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

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

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

    % Track with radar
    [radardets, radarTracks, inforadar] = updateRadarTracker(radar,radarPHD, egoPose, time);
    % Track with lidar
    [lidardets, lidarTracks, nonGroundCloud, groundCloud] = updateLidarTracker(lidar,lidarDetecto

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

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

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



Based on the visualization results, you perform an initial qualitative assessment of the tracking performance. The display at the end of the scenario shows that all three UAVs were well tracked by the ego. With the current sensor suite configuration, lidar tracks were only established partially due to the limited coverage of the lidar sensor. The wider field of view of the radar allowed establishing radar tracks more consistently in this scenario.



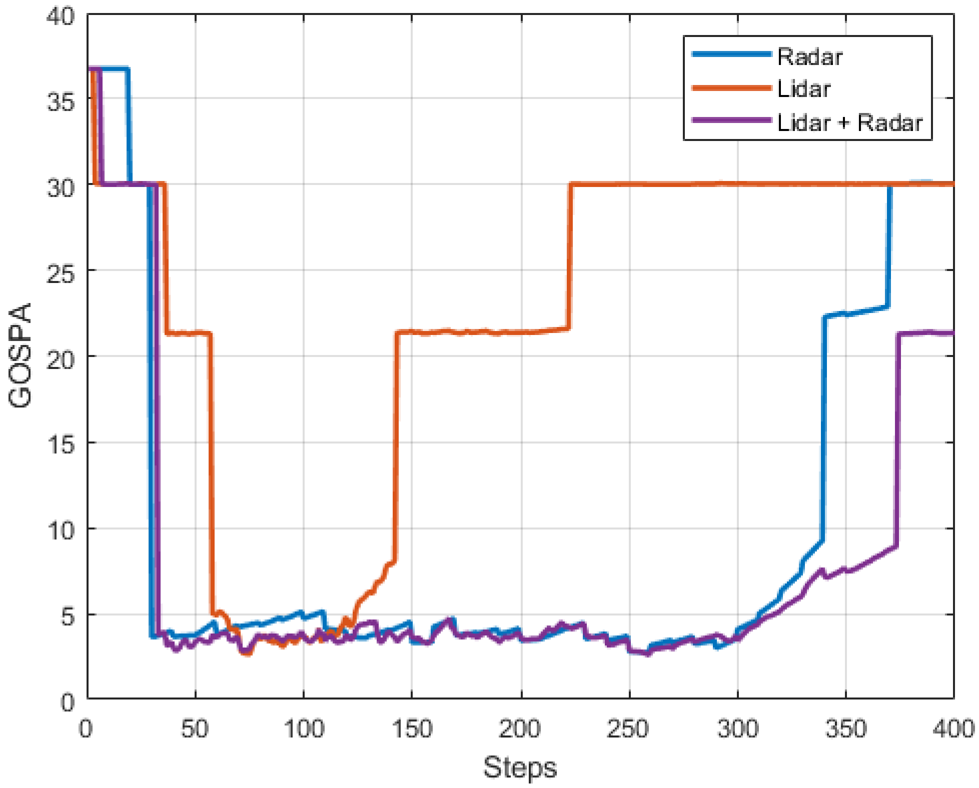
The three animated GIFs above show parts of the chase views. You can see that the quality of lidar tracks (orange box) is affected by the geometry of the scenario. UAV A (left) is illuminated by the lidar (shown in yellow) almost directly from above. This enables the tracker to capture the full extent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

Tracking metrics

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

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

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

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

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

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



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

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

Summary

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

Supporting Functions

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

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

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

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

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

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

end
```

`createRadarTracker` creates the trackerPHD tracker to fuse radar detections.

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

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

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

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

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

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

radarPHDconfig.FilterInitializationFcn = @initRadarFilter;

radarPHDconfig.MinDetectionProbability = 0.4;

% Threshold for partitioning
threshold = [3 16];
tracker = trackerPHD('TrackerIndex',1,...
    'HasSensorConfigurationsInput',true,...
    'SensorConfigurations',{radarPHDconfig},...
    'BirthRate',1e-3,...
    'AssignmentThreshold',50,...% Minimum negative log-likelihood of a detection cell to add bir
    'ExtractionThreshold',0.80,...% Weight threshold of a filter component to be declared a trac
    'ConfirmationThreshold',0.99,...% Weight threshold of a filter component to be declared a co
    'MergingThreshold',50,...% Threshold to merge components
    'DeletionThreshold',0.1,...% Threshold to delete components
    'LabelingThresholds',[1.01 0.01 0],...% This translates to no track-splitting. Read 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 1) initialize new birth components in the density and 2) initialize new component from detection partitions.

```
function phd = initRadarFilter (detectionPartition)

if nargin == 0

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

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

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

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

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

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

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

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

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

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

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

        Bk = H(:,1:2:end);
        Bk2 = eye(3)/Bk;
        V = (dof-4)*Bk2*Z*Bk2';

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

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

    end
```

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

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

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

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

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

end
```

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

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

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

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

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

```
function ekf = initLidarFilter(detection)

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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/

2

Clutter

Clutter Modeling

In this section...

"Surface Clutter Overview" on page 2-2

"Approaches for Clutter Simulation or Analysis" on page 2-2

"Considerations for Setting Up a Constant Gamma Clutter Simulation" on page 2-2

"Related Examples" on page 2-4

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 1-510
- “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”

3

Interference

Barrage Jammer

In this section...

“Support for Modeling Barrage Jammer” on page 3-2

“Model Barrage Jammer Output” on page 3-2

“Model Effect of Barrage Jammer on Target Echo” on page 3-3

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:

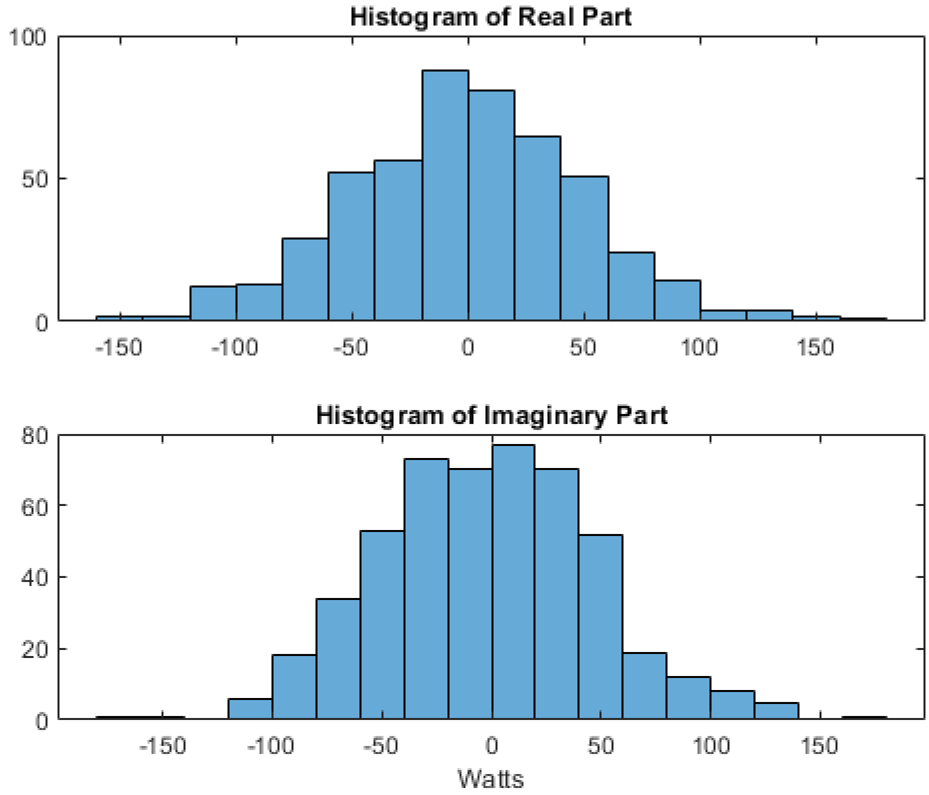
$$w[n] = \sqrt{\frac{P}{2}}x[n] + j\sqrt{\frac{P}{2}}y[n]$$

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);
```

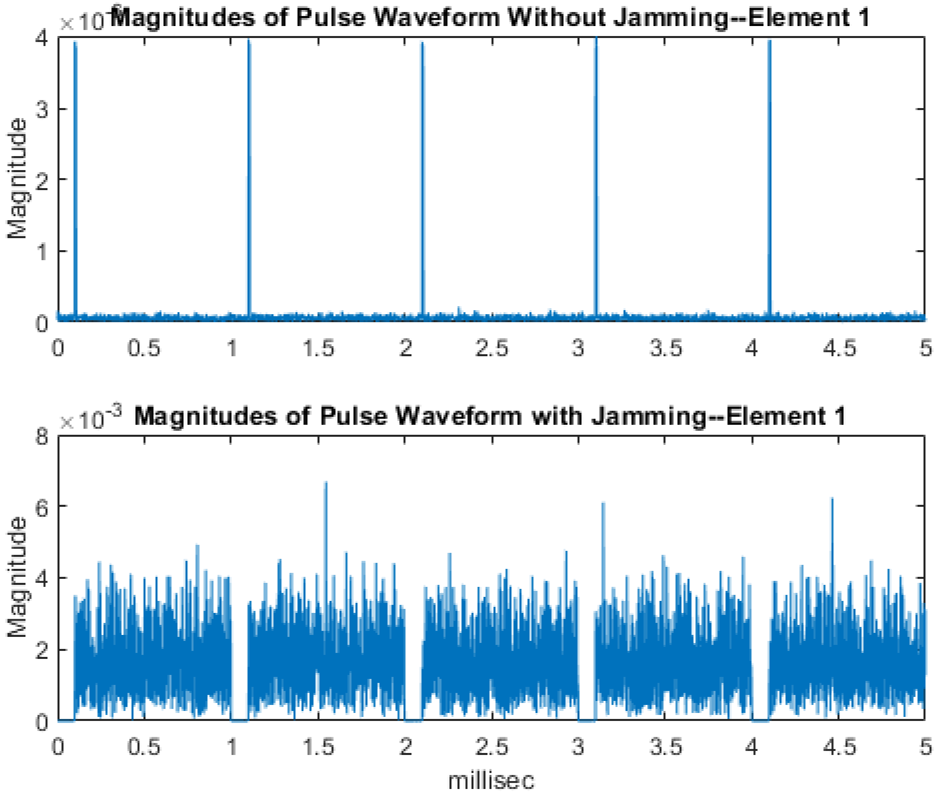
Generate the jamming signal

```
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')
```



4

Radar Equation

Radar Equation

In this section...

"Radar Equation Theory" on page 4-2

"Plot Vertical Coverage Pattern Using Default Parameters" on page 4-3

"Compute Peak Power Using Radar Equation Calculator App" on page 4-4

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 (SNR) ratio. 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

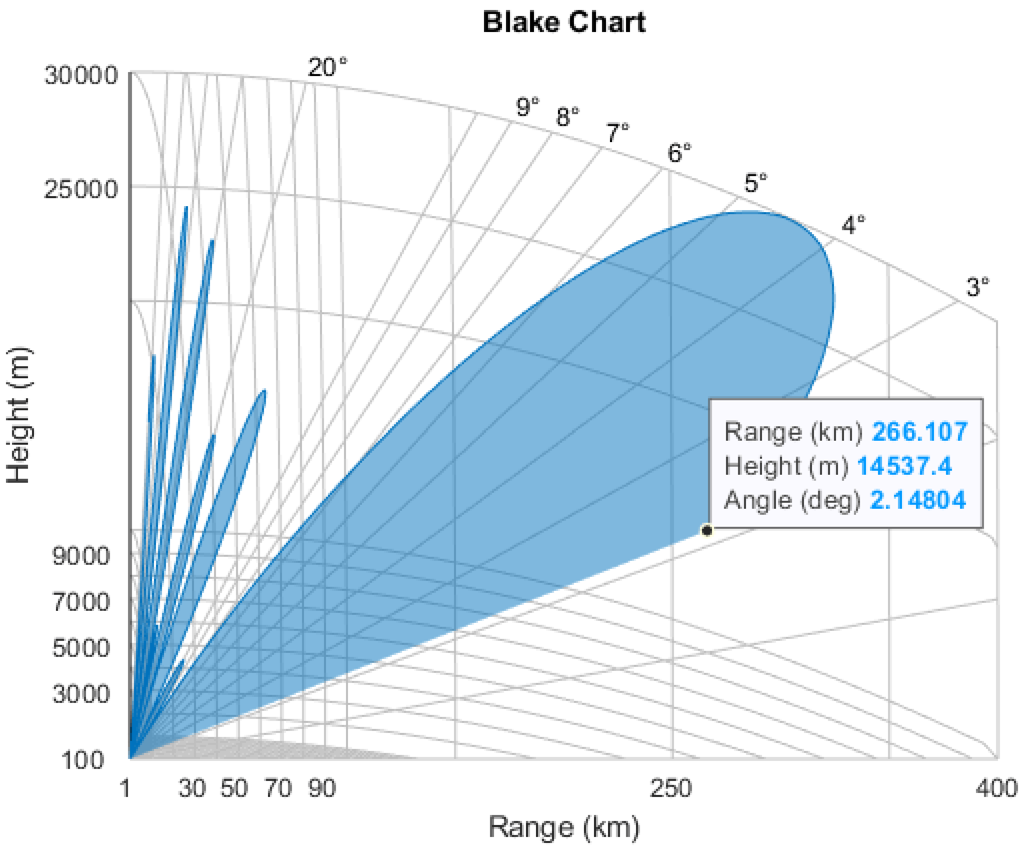
Set the frequency to 100 MHz, the antenna height to 10 m, and the free-space range to 200 km. The antenna pattern, surface roughness, antenna tilt angle, and field polarization assume their default values as specified in the `AntennaPattern`, `SurfaceRoughness`, `TiltAngle`, and `Polarization` properties.

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);
```



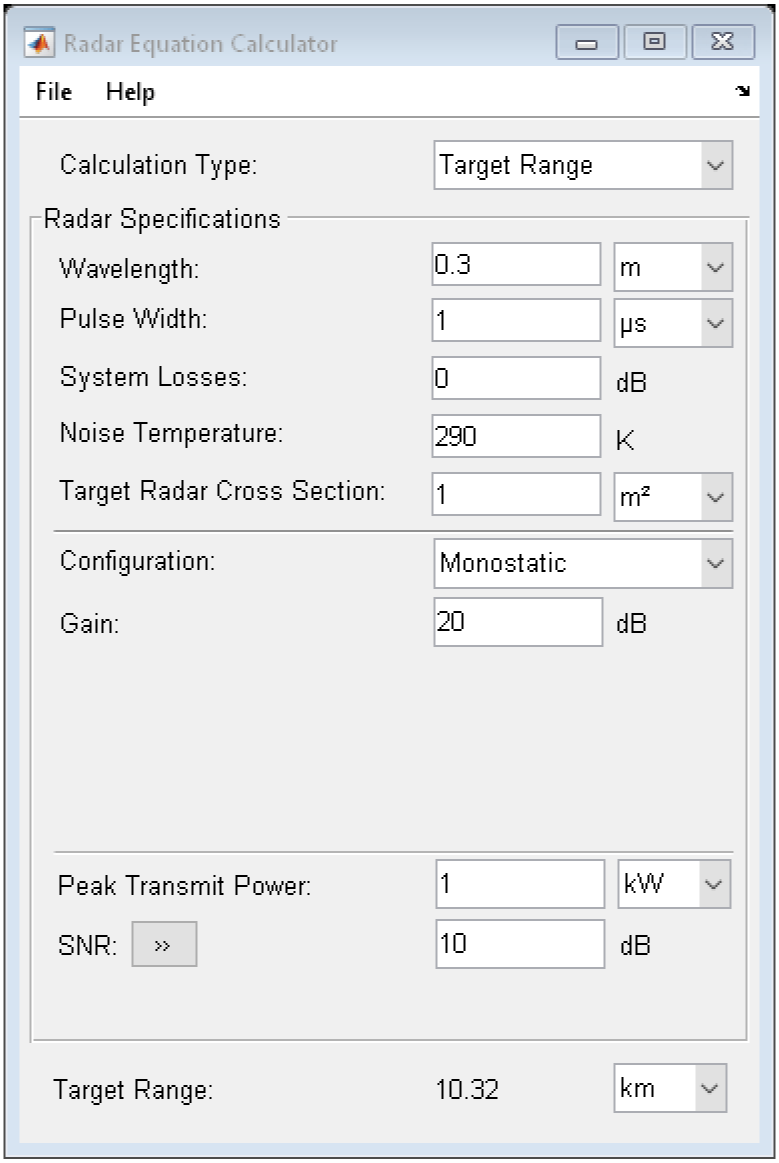
Compute Peak Power Using Radar Equation Calculator App

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
```



Calculation Type: Target Range

Wavelength: 0.3 m

Pulse Width: 1 μs

System Losses: 0 dB

Noise Temperature: 290 K

Target Radar Cross Section: 1 m²

Configuration: Monostatic

Gain: 20 dB

Peak Transmit Power: 1 kW

SNR: 10 dB

Target Range: 10.32 km

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 m2
6. Choose **Configuration** as `Monostatic`
7. Set the **Gain** to be `40` dB

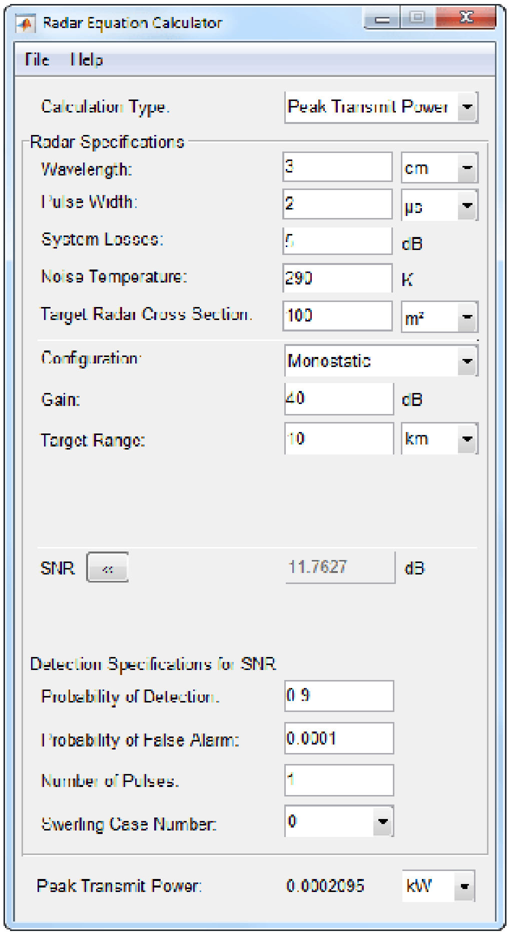
8. Open the **SNR** box
9. Specify the **Probability of Detections** as `0.9`
10. Specify the **Probability of False Alarm** as `0.0001`

Close the app window. Normally, you close the app using the close button.

```
hg = findall(0,'Name','Radar Equation Calculator');
close(hg)
```

You can see from this previously prepared screen shot that the required peak transmit power is .2095 W.

```
im = imread('radarEquationExample_03.png');
figure('Position',[344 206 849 644])
image(im)
axis off
set(gca,'Position',[0.083 0.083 0.834 0.888])
```



Calculation Type: Peak Transmit Power

Radar Specifications

Wavelength: 3 cm

Pulse Width: 2 μs

System Losses: 5 dB

Noise Temperature: 290 K

Target Radar Cross Section: 100 m²

Configuration: Monostatic

Gain: 40 dB

Target Range: 10 km

SNR 11.7627 dB

Detection Specifications for SNR

Probability of Detection: 0.9

Probability of False Alarm: 0.0001

Number of Pulses: 1

Swerling Case Number: 0

Peak Transmit Power: 0.0002095 kW

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 bette
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.

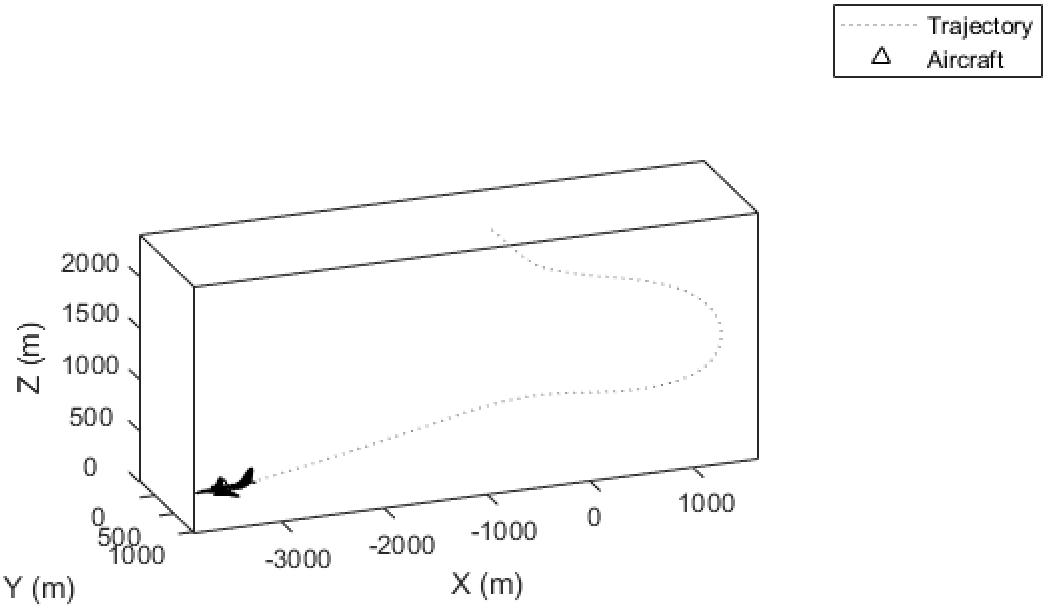
Create a second aircraft trajectory with the same waypoints as the first aircraft trajectory, but set its `AutoPitch` and `AutoBank` properties to `true`. This generates a trajectory more representative of the possible aircraft maneuvers.

```
aircraftTraj2 = waypointTrajectory(waypoints,timeOfArrival, ...
    'AutoPitch',true, ...
    'AutoBank',true);
```

Plot the second trajectory and observe the change in aircraft orientation.

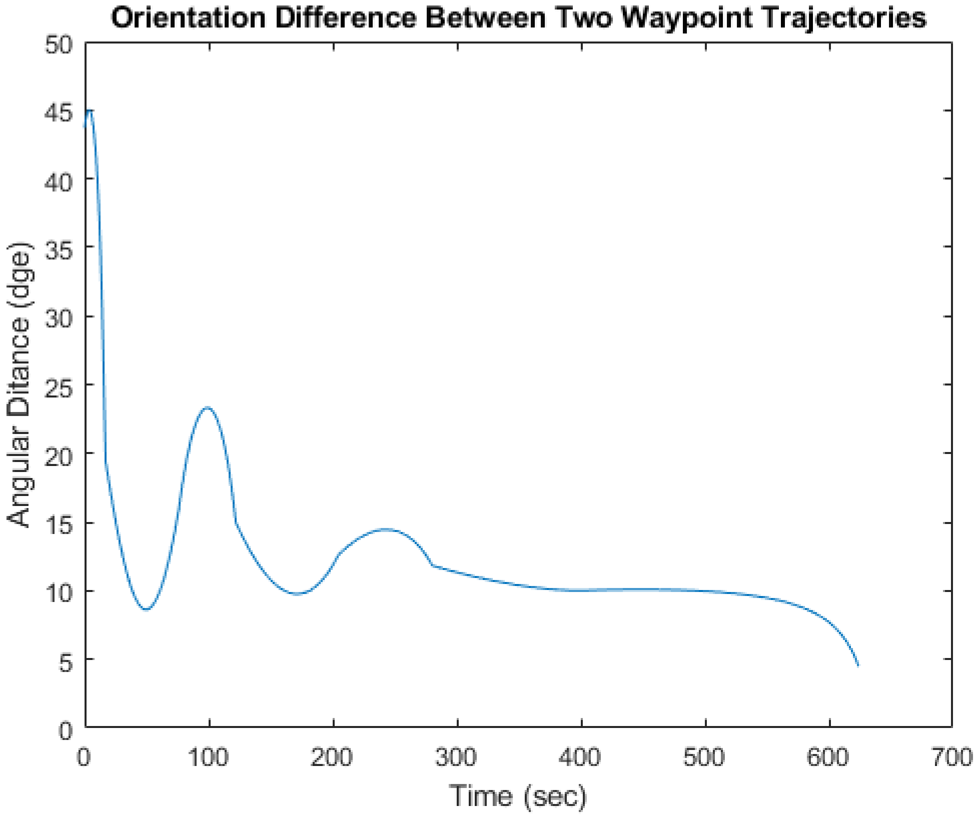
```
[positions2,orientations2] = lookupPose(aircraftTraj2,sampleTimes);
for i = 1:numSteps
```

```
    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.

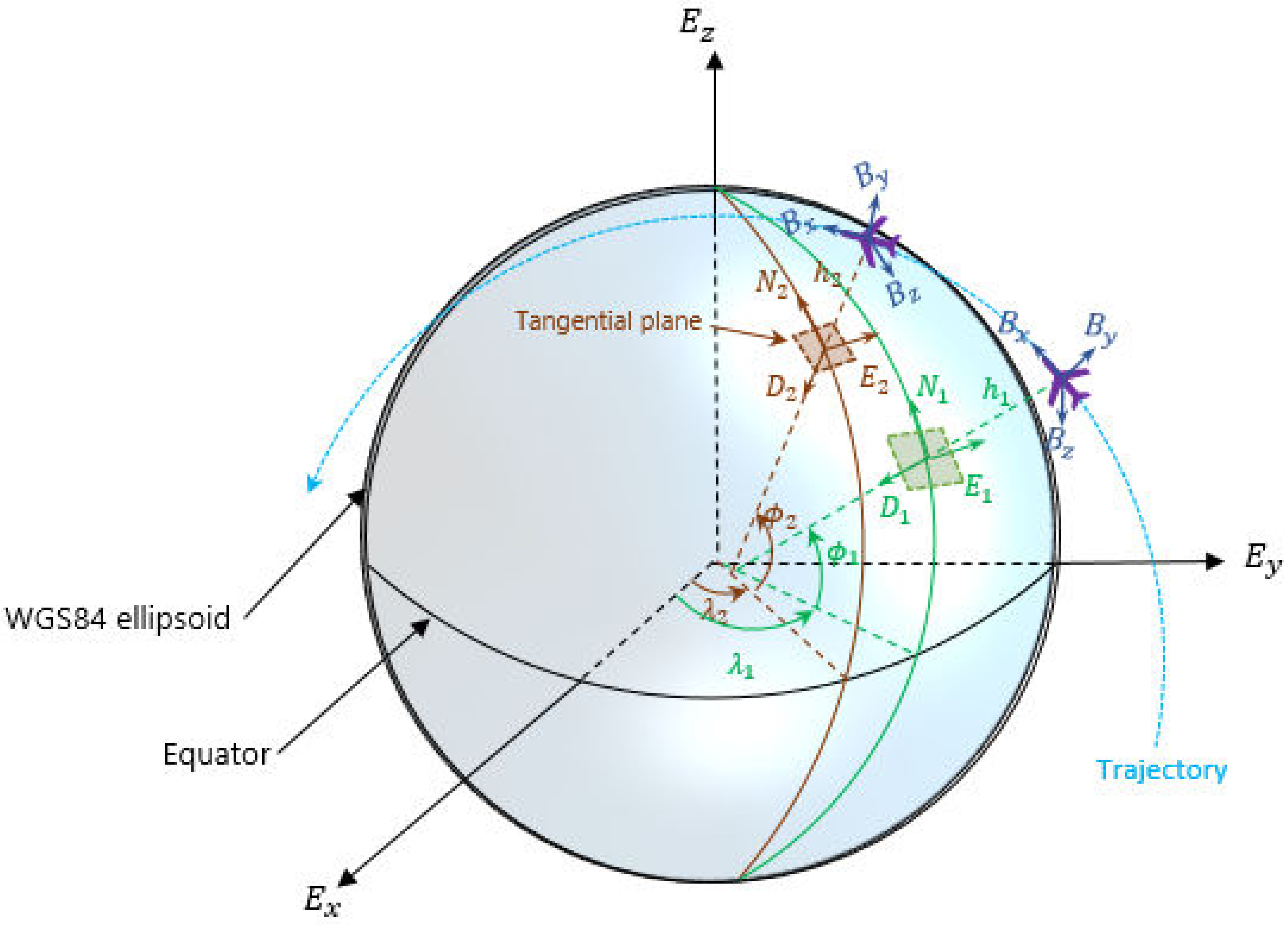
```
distance = dist(orientations2,orientations);
figure
plot(sampleTimes,distance*180/pi)
xlabel('Time (sec)')
ylabel('Angular Ditance (dge)')
title('Orientation Difference Between Two Waypoint Trajectories')
```



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.

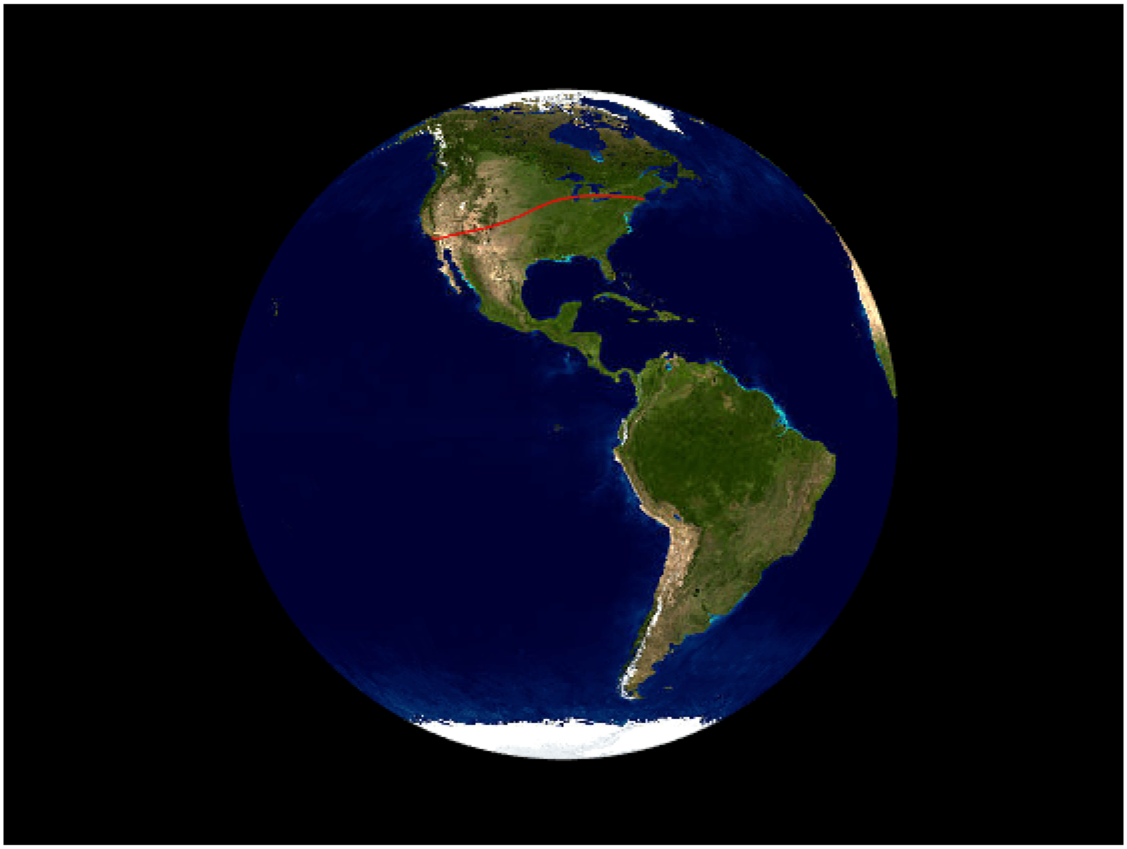
```
planeTraj = geoTrajectory([latitudes longitudes heights],timeOfArrival, ...
    'Course',courses,'GroundSpeed',speeds);
```

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.

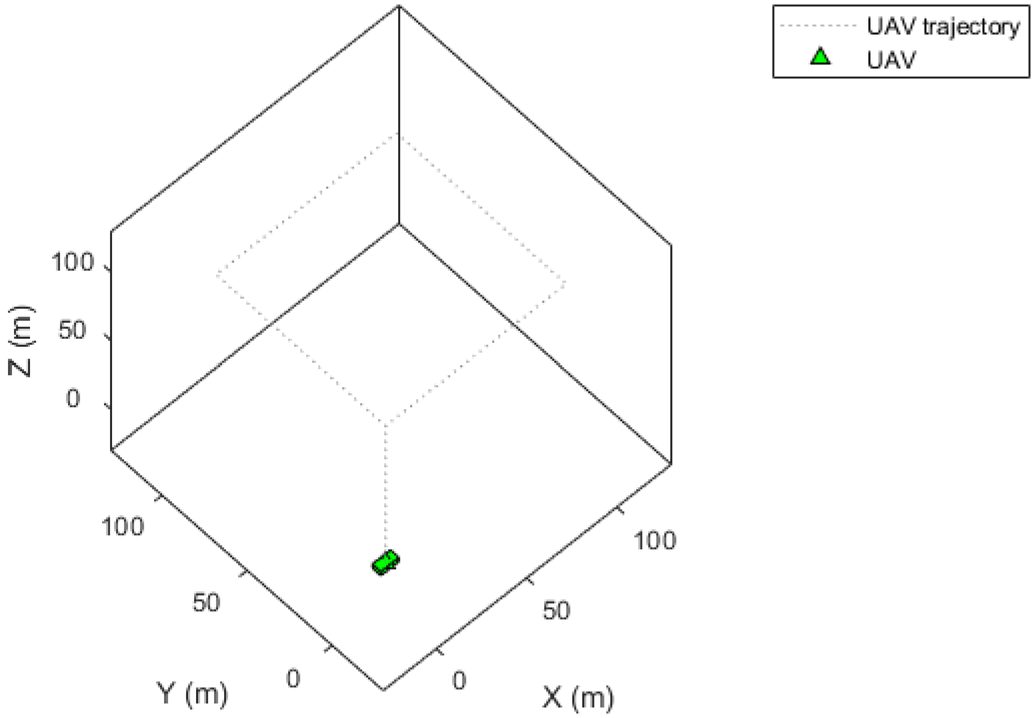
```
segSteps = floor(numSteps/12);
accelerations = zeros(3,numSteps);
acc = 1;
% Acceleration for taking off and ascending in the z-direction
accelerations(3,1:segSteps) = acc;
accelerations(3,segSteps+1:2*segSteps) = -acc;

% 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
```

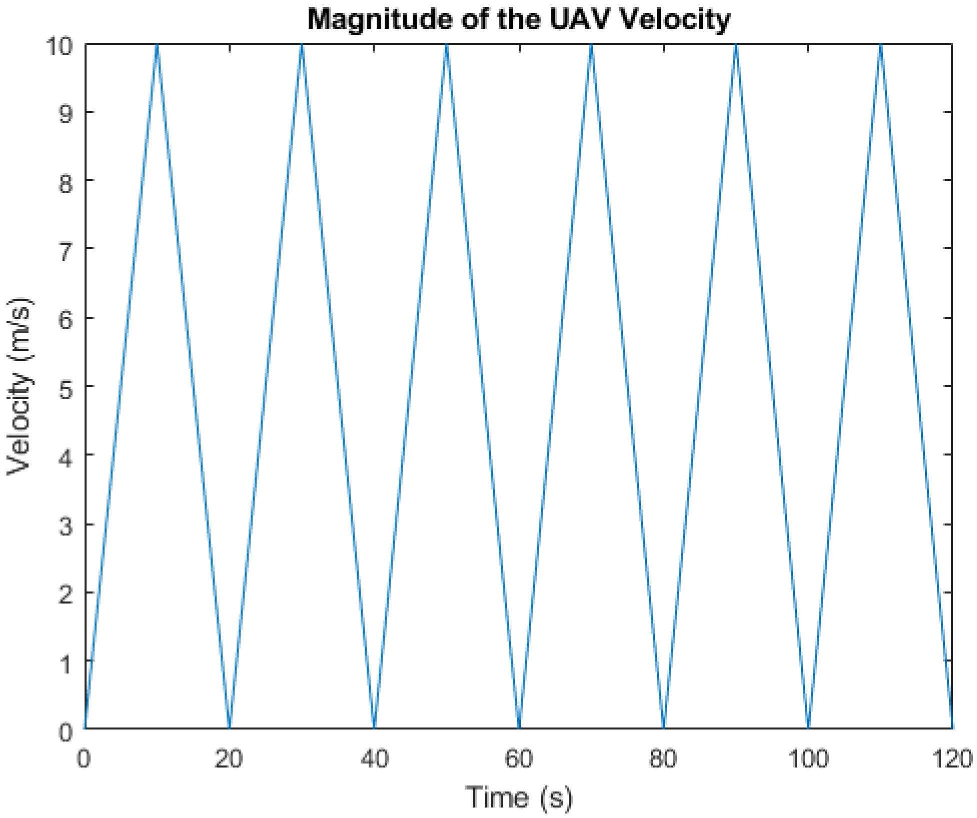
Visualize the trajectory using `theaterPlot`.

```
% 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')
```

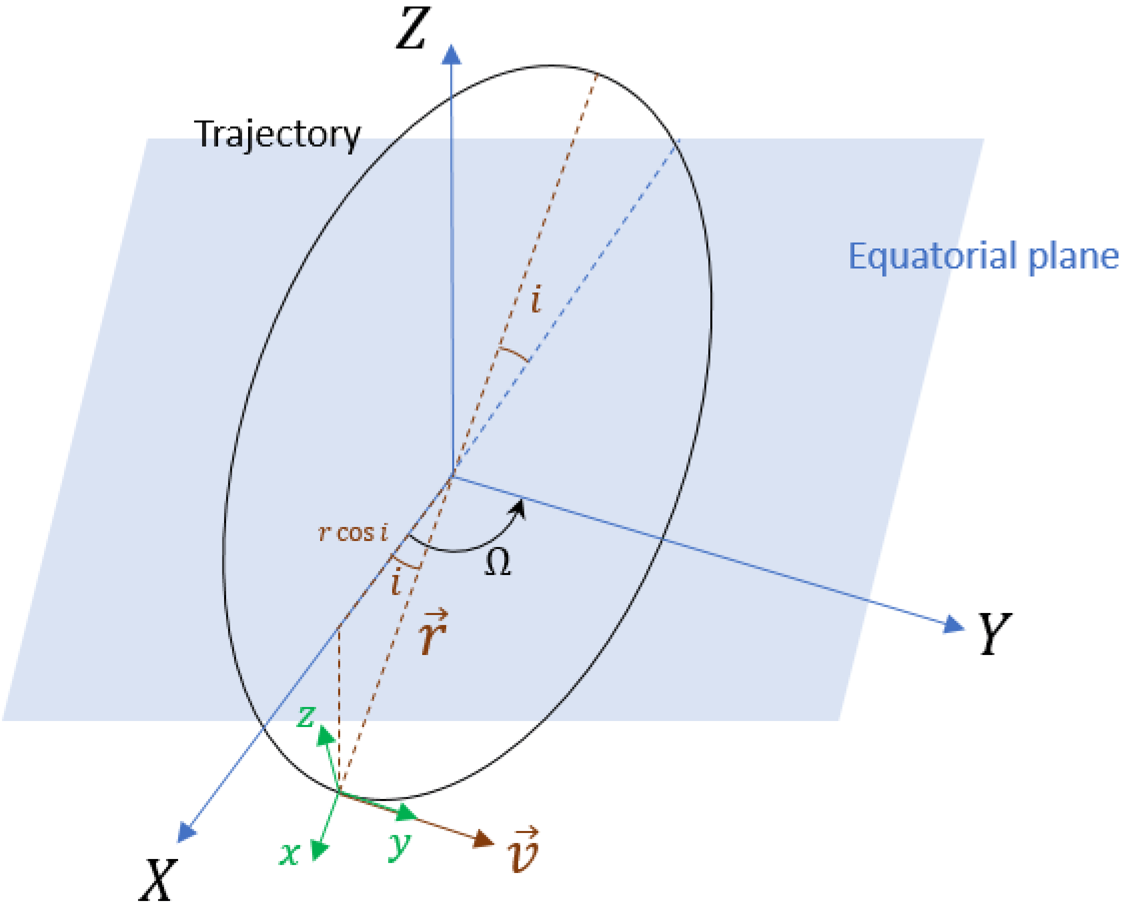


kinematicTrajectory Example For Spacecraft Trajectory

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 ($v = -90^\circ$), 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]';
```

```
a = v^2/radius;
acceleration = [-a 0 0]';
```

Set a simulation time of one orbital period. Specify a simulation step as 2 seconds.

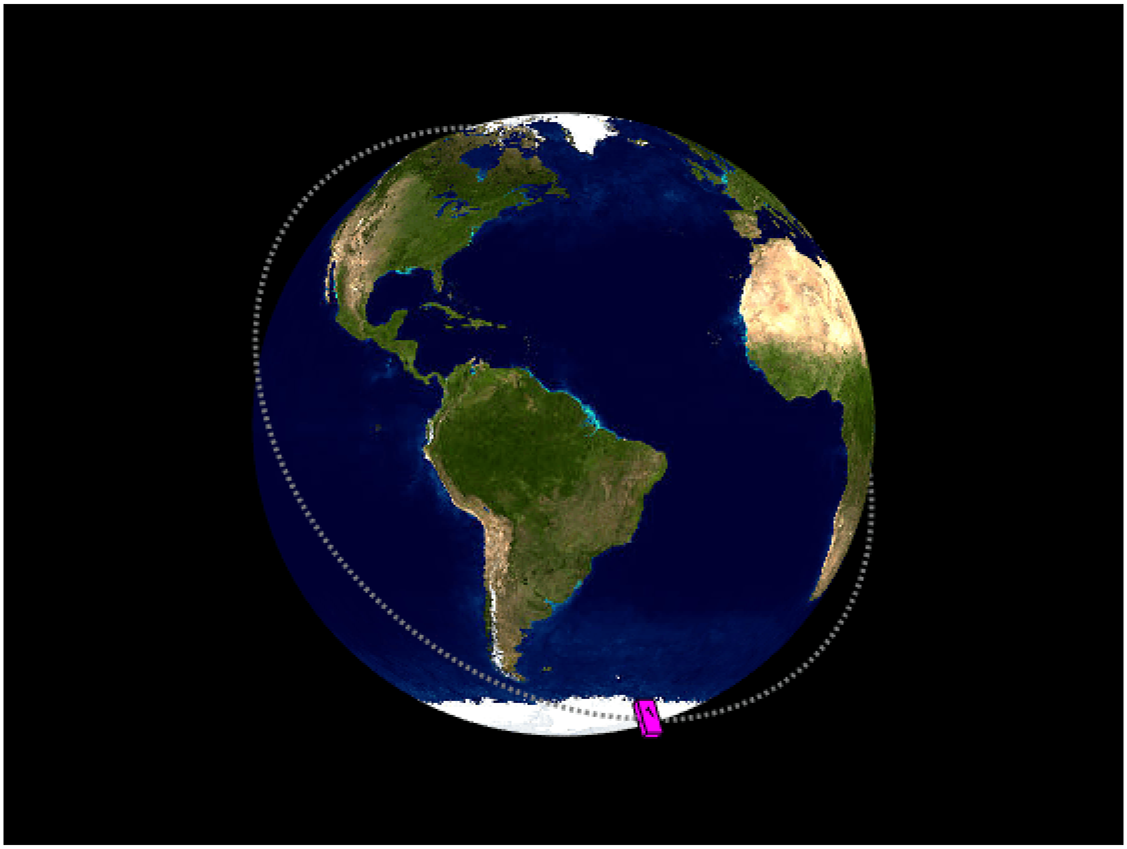
```
tFinal = 2*pi/omega;
dt = 2;
sampleRate = 1/dt;
tspan = 0:dt:tFinal;
numSteps = length(tspan);
```

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.

```
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.

```
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

6

Kalman Filters

- “Linear Kalman Filters” on page 6-2
- “Extended Kalman Filters” on page 6-7

Linear Kalman Filters

In this section...

“State Equations” on page 6-2

“Measurement Models” on page 6-3

“Linear Kalman Filter Equations” on page 6-4

“Filter Loop” on page 6-4

“Constant Velocity Model” on page 6-5

“Constant Acceleration Model” on page 6-6

When you use a Kalman filter to track objects, you use a sequence of detections or measurements to construct a model of the object motion. Object motion is defined by the evolution of the state of the object. The Kalman filter is an optimal, recursive algorithm for estimating the track of an object. The filter is recursive because it updates the current state using the previous state, using measurements that may have been made in the interval. A Kalman filter incorporates these new measurements to keep the state estimate as accurate as possible. The filter is optimal because it minimizes the mean-square error of the state. You can use the filter to predict future states or estimate the current state or past state.

State Equations

For most types of objects tracked in the toolbox, the state vector consists of one-, two- or three-dimensional positions and velocities.

Start with Newton equations for an object moving in the *x*-direction at constant acceleration and convert these equations to space-state form.

$$m\ddot{x} = f$$

$$\ddot{x} = \frac{f}{m} = a$$

If you define the state as

$$x_1 = x$$

$$x_2 = \dot{x},$$

you can write Newton’s law in state-space form.

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}a$$

You use a linear dynamic model when you have confidence that the object follows this type of motion. Sometimes the model includes process noise to reflect uncertainty in the motion model. In this case, Newton’s equations have an additional term.

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}a + \begin{bmatrix} 0 \\ 1 \end{bmatrix}v_k$$

v_k is the unknown noise perturbations of the acceleration. Only the statistics of the noise are known. It is assumed to be zero-mean Gaussian white 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 on the right side is the state transition model matrix. For independent *x*- and *y*-motions, this matrix is block diagonal.

When you transition to discrete time, you integrate the equations of motion over the length of the time interval. In discrete form, for a sample interval of *T*, the state-representation becomes

$$\begin{bmatrix} x_{1,\,k+1} \\ x_{2,\,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x_{1,\,k} \\ x_{2,\,k} \end{bmatrix} + \begin{bmatrix} 0 \\ T \end{bmatrix}a + \begin{bmatrix} 0 \\ 1 \end{bmatrix}\tilde{v}$$

The quantity 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.

The state equation can be generalized to

$$x_{k+1} = F_k x_k + G_k u_k + v_k$$

F_k is the state transition matrix and G_k is the control matrix. The control matrix takes into account any known forces acting on the object. Both of these matrices are given. The last term represents noise-like random perturbations of the dynamic model. The noise is assumed to be zero-mean Gaussian white noise.

Continuous-time systems with input noise are described by linear stochastic differential equations. Discrete-time systems with input noise are described by linear stochastic differential equations. A state-space representation is a mathematical model of a physical system where the inputs, outputs, and state variables are related by first-order coupled equations.

Measurement Models

Measurements are what you observe about your system. Measurements depend on the state vector but are not always 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. For the linear Kalman filter, the measurements are always linear functions of the state vector, ruling out spherical coordinates. To use spherical coordinates, use the extended Kalman filter.

The measurement model assumes that the actual measurement at any time is related to the current state by

$$z_k = H_k x_k + w_k$$

w_k represents measurement noise at the current time step. The measurement noise is also zero-mean white Gaussian noise with covariance matrix Q described by $Q_k = E[n_k n_k^T]$.

Linear Kalman Filter Equations

Without noise, the dynamic equations are

$$x_{k+1} = F_k x_k + G_k u_k .$$

Likewise, the measurement model has no measurement noise contribution. At each instance, the process and measurement noises are not known. Only the noise statistics are known. The

$$z_k = H_k x_k$$

You can put these equations into a recursive loop to estimate how the state evolves and also how the uncertainties in the state components evolve.

Filter Loop

Start with a best estimate of the state, $x_{0/0}$, and the state covariance, $P_{0/0}$. The filter performs these steps in a continual loop.

1. Propagate the state to the next step using the motion equations.

   $$x_{k+1|k} = F_k x_{k|k} + G_k u_k .$$

   Propagate the covariance matrix as well.

   $$P_{k+1|k} = F_k P_{k|k} F_k^T + Q_k .$$

   The subscript notation *k+1|k* indicates that the quantity is the optimum estimate at the *k+1* step propagated from step *k*. This estimate is often called the *a priori* estimate.

   Then predict the measurement at the updated time.

   $$z_{k+1|k} = H_{k+1} x_{k+1|k}$$

2. Use the difference between the actual measurement and predicted measurement to correct the state at the updated time. The correction requires computing the Kalman gain. To do this, first compute the measurement prediction covariance (innovation)

   $$S_{k+1} = H_{k+1} P_{k+1|k} H_{k+1}^T + R_{k+1}$$

   Then the Kalman gain is

   $$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$$

   and is derived from using an optimality condition.

3. Correct the predicted estimate with the measurement. Assume that the estimate is a linear combination of the predicted state and the measurement. The estimate after correction uses the subscript notation, *k+1|k+1*. is computed from

   $$x_{k+1|k+1} = x_{k+1|k} + K_{k+1}(z_{k+1} - z_{k+1|k})$$

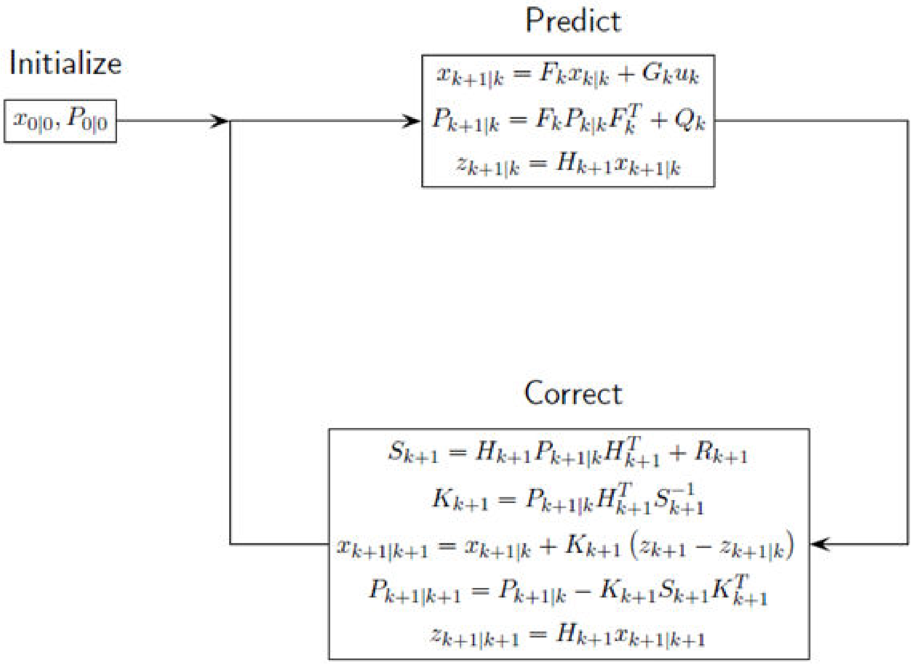
   where K_{k+1} is the Kalman gain. The corrected state is often called the *a posteriori* estimate of the state because it is derived after the measurement is included.

   Correct the state covariance matrix

$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1}S_{k+1}K'_{k+1}$$

Finally, you can compute a measurement based upon the corrected state. This is not a correction to the measurement but is a best estimate of what the measurement would be based upon the best estimate of the state. Comparing this to the actual measurement gives you an indication of the performance of the filter.

This figure summarizes the Kalman loop operations.


Initialize
$x_{0|0}, P_{0|0}$
Predict
$x_{k+1|k} = F_k x_{k|k} + G_k u_k$
$P_{k+1|k} = F_k P_{k|k} F_k^T + Q_k$
$z_{k+1|k} = H_{k+1} x_{k+1|k}$
Correct
$S_{k+1} = H_{k+1} P_{k+1|k} H_{k+1}^T + R_{k+1}$
$K_{k+1} = P_{k+1|k} H_{k+1}^T S_{k+1}^{-1}$
$x_{k+1|k+1} = x_{k+1|k} + K_{k+1}\left(z_{k+1} - z_{k+1|k}\right)$
$P_{k+1|k+1} = P_{k+1|k} - K_{k+1} S_{k+1} K_{k+1}^T$
$z_{k+1|k+1} = H_{k+1} x_{k+1|k+1}$

Constant Velocity Model

The linear Kalman filter contains a built-in linear constant-velocity motion model. Alternatively, you can specify the transition matrix for linear motion. The state update at the next time step is a linear function of the state at the present time. In this filter, the measurements are also linear functions of the state described by a measurement matrix. For an object moving in 3-D space, the state is described by position and velocity in the *x*-, *y*-, and *z*-coordinates. The state transition model for the constant-velocity motion is

$$\begin{bmatrix} x_{k+1} \\ v_{x,k+1} \\ y_{k+1} \\ v_{y,k+1} \\ z_{k+1} \\ v_{z,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & T & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ y_k \\ v_{y,k} \\ z_k \\ v_{z,k} \end{bmatrix}$$

The measurement model is a linear function of the state vector. The simplest case is one where the measurements are the position components of the state.

$$\begin{bmatrix} m_{x,k} \\ m_{y,k} \\ m_{z,k} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ y_k \\ v_{y,k} \\ z_k \\ v_{z,k} \end{bmatrix}$$

Constant Acceleration Model

The linear Kalman filter contains a built-in linear constant-acceleration motion model. Alternatively, you can specify the transition matrix for constant-acceleration linear motion. The transition model for linear acceleration is

$$\begin{bmatrix} x_{k+1} \\ v_{x,k+1} \\ a_{x,k+1} \\ y_{k+1} \\ v_{y,k+1} \\ a_{y,k+1} \\ z_{k+1} \\ v_{z,k+1} \\ a_{z,k+1} \end{bmatrix} = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & T & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ a_{x,k} \\ y_k \\ v_{y,k} \\ a_{y,k} \\ z_k \\ v_{z,k} \\ a_{z,k} \end{bmatrix}$$

The simplest case is one where the measurements are the position components of the state.

$$\begin{bmatrix} m_{x,k} \\ m_{y,k} \\ m_{z,k} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_k \\ v_{x,k} \\ a_{x,k} \\ y_k \\ v_{y,k} \\ a_{y,k} \\ z_k \\ v_{z,k} \\ a_{y,k} \end{bmatrix}$$

See Also

Objects
`trackingKF`

Extended Kalman Filters

In this section...

"State Update Model" on page 6-7

"Measurement Model" on page 6-7

"Extended Kalman Filter Loop" on page 6-8

"Predefined Extended Kalman Filter Functions" on page 6-9

Use an extended Kalman filter when object motion follows a nonlinear state equation or when the measurements are nonlinear functions of the state. A simple example is when the state or measurements of the object are calculated in spherical coordinates, such as azimuth, elevation, and range.

State Update Model

The extended Kalman filter formulation linearizes the state equations. The updated state and covariance matrix remain linear functions of the previous state and covariance matrix. However, the state transition matrix in the linear Kalman filter is replaced by the Jacobian of the state equations. The Jacobian matrix is not constant but can depend on the state itself and time. To use the extended Kalman filter, you must specify both a state transition function and the Jacobian of the state transition function.

Assume there is a closed-form expression for the predicted state as a function of the previous state, controls, noise, and time.

$$x_{k+1} = f(x_k, u_k, w_k, t)$$

The Jacobian of the predicted state with respect to the previous state is

$$F^{(x)} = \frac{\partial f}{\partial x}.$$

The Jacobian of the predicted state with respect to the noise is

$$F^{(w)} = \frac{\partial f}{\partial w_i}.$$

These functions take simpler forms when the noise enters linearly into the state update equation:

$$x_{k+1} = f(x_k, u_k, t) + w_k$$

In this case, $F^{(w)} = 1_M$.

Measurement Model

In the extended Kalman filter, the measurement can 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 enters linearly into the measurement equation:

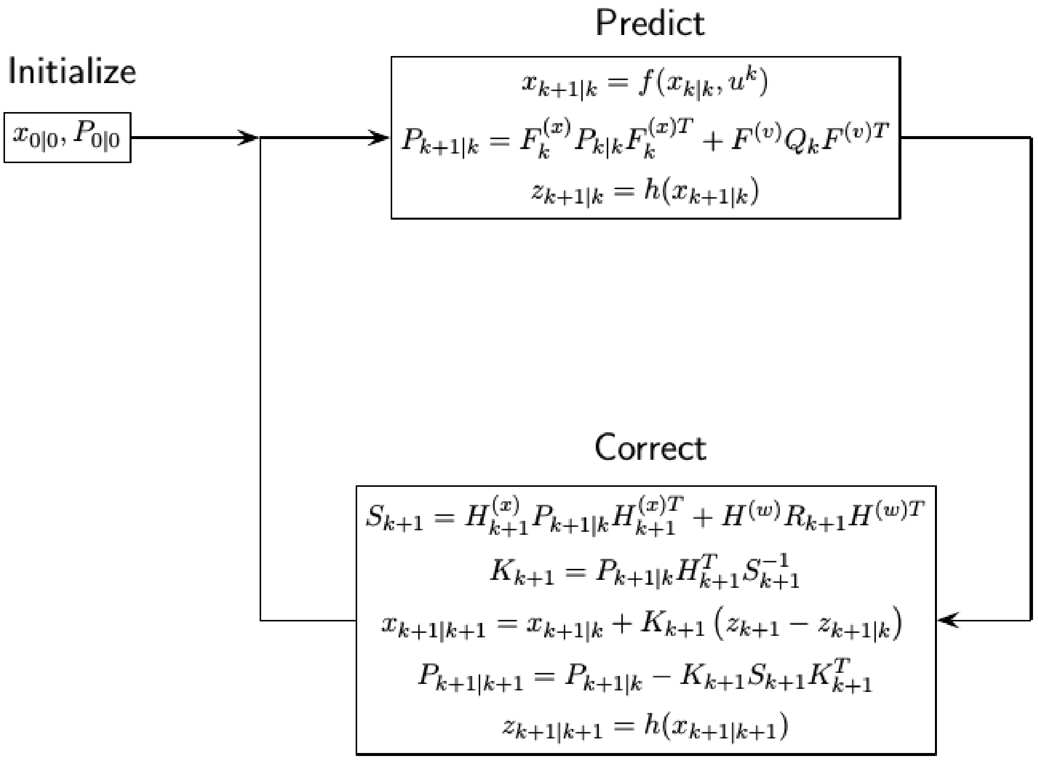
$$z_k = h(x_k, t) + v_k$$

In this case, $H^{(v)} = 1_N$.

Extended Kalman Filter Loop

This extended Kalman filter loop is almost identical to the linear Kalman filter loop except that:

- The exact nonlinear state update and measurement functions are used whenever possible and the state transition matrix is replaced by the state Jacobian
- The measurement matrices are replaced by the appropriate Jacobians.



$$x_{k+1|k} = f(x_{k|k}, u^k)$$
$$P_{k+1|k} = F_k^{(x)} P_{k|k} F_k^{(x)T} + F^{(v)} Q_k F^{(v)T}$$
$$z_{k+1|k} = h(x_{k+1|k})$$

$$S_{k+1} = H_{k+1}^{(x)} P_{k+1|k} H_{k+1}^{(x)T} + H^{(w)} R_{k+1} H^{(w)T}$$
$$K_{k+1} = P_{k+1|k} H_{k+1}^{T} S_{k+1}^{-1}$$
$$x_{k+1|k+1} = x_{k+1|k} + K_{k+1} \left(z_{k+1} - z_{k+1|k} \right)$$
$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1} S_{k+1} K_{k+1}^{T}$$
$$z_{k+1|k+1} = h(x_{k+1|k+1})$$

Predefined Extended Kalman Filter Functions

The toolbox provides predefined state update and measurement functions to use in the extended Kalman filter.

Motion Model	Function Name	Function Purpose
Constant velocity	`constvel`	Constant-velocity state update model
	`constveljac`	Constant-velocity state update Jacobian
	`cvmeas`	Constant-velocity measurement model
	`cvmeasjac`	Constant-velocity measurement Jacobian
Constant acceleration	`constacc`	Constant-acceleration state update model
	`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
	`constturnjac`	Constant turn-rate state update Jacobian
	`ctmeas`	Constant turn-rate measurement model
	`ctmeasjac`	Constant-turnrate measurement Jacobian

See Also

Objects
`trackingEKF`