Computer Vision System Toolbox™

Getting Started Guide

MATLAB®

R2015b

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Computer Vision System Toolbox™ Getting Started Guide

Revision History

April 2011	Online only	Revised for Version 4.0 (Release 2011a)
September 2011	Online only	Revised for Version 4.1 (Release 2011b)
March 2012	Online only	Revised for Version 5.0 (Release 2012a)
September 2012	Online only	Revised for Version 5.1 (R2012b)
March 2013	Online only	Revised for Version 5.2 (R2013a)
September 2013	Online only	Revised for Version 5.3 (R2013b)
March 2014	Online only	Revised for Version 6.0 (R2014a)
October 2014	Online only	Revised for Version 6.1 (R2014b)
March 2015	Online only	Revised for Version 6.2 (Release R2015a)
September 2015	Online only	Revised for Version 7.0 (Release R2015b)

Contents

1 Product Overview

Computer Vision System Toolbox Product Description **1-2**
Key Features **1-2**

2 Computer Vision Algorithms and Video Processing

Computer Vision Capabilities **2-2**

Video Processing in MATLAB **2-3**

Computer Vision System Toolbox Preferences **2-4**
Parallel Computing Toolbox Support **2-4**

3 Coordinate Systems

Coordinate Systems **3-2**
Pixel Indices **3-2**
Spatial Coordinates **3-2**
3-D Coordinate Systems **3-3**

4 System Objects

What Is a System Toolbox? **4-2**

What Are System Objects? **4-3**

System Objects vs. MATLAB Functions **4-5**
System Objects vs. MATLAB Functions **4-5**
Process Audio Data Using Only MATLAB Functions Code . . **4-5**
Process Audio Data Using System Objects **4-6**

System Design and Simulation in MATLAB **4-8**

System Design and Simulation in Simulink **4-9**

System Objects in MATLAB Code Generation **4-10**
System Objects in Generated Code **4-10**
System Objects in codegen **4-14**
System Objects in the MATLAB Function Block **4-15**
System Objects in the MATLAB System Block **4-15**
System Objects and MATLAB Compiler Software **4-15**

System Objects Methods That Support Code Generation . . **4-16**
Code Generation Supported System Objects Methods **4-16**
Simulation-Only System Objects Methods **4-16**

System Objects in Simulink **4-18**
System Objects in the MATLAB Function Block **4-18**
System Objects in the MATLAB System Block **4-18**

System Object Methods **4-19**
What Are System Object Methods? **4-19**
The Step Method **4-19**
Common Methods **4-20**

System Design in MATLAB Using System Objects **4-22**
Create Components for Your System **4-22**
Configure Components for Your System **4-23**
Assemble Components to Create Your System **4-25**
Run Your System **4-26**
Reconfigure Your System During Runtime **4-27**

System Design in Simulink Using System Objects **4-29**
Define New Kinds of System Objects for Use in Simulink . . **4-29**
Test New System Objects in MATLAB **4-34**
Add System Objects to Your Simulink Model **4-35**

5 Strategies for Real-Time Video Processing in Simulink

Optimizing Your Implementation **5-2**

Developing Your Models **5-4**

6 Data Type Support

Block Data Type Support **6-2**

Fixed-Point Support for MATLAB System Objects **6-3**
Getting Information About Fixed-Point System Objects **6-3**
Displaying Fixed-Point Properties **6-4**
Setting System Object Fixed-Point Properties **6-5**

Product Overview

Computer Vision System Toolbox Product Description

Design and simulate computer vision and video processing systems

Computer Vision System Toolbox provides algorithms, functions, and apps for designing and simulating computer vision and video processing systems. You can perform feature detection, extraction, and matching; object detection and tracking; motion estimation; and video processing. For 3-D computer vision, the system toolbox supports camera calibration, stereo vision, 3-D reconstruction, and 3-D point cloud processing. With machine learning based frameworks, you can train object detection, object recognition, and image retrieval systems. Algorithms are available as MATLAB® functions, System objects, and Simulink® blocks.

For rapid prototyping and embedded system design, the system toolbox supports fixed-point arithmetic and C-code generation.

Key Features

- Object detection and tracking, including the Viola-Jones, Kanade-Lucas-Tomasi (KLT), and Kalman filtering methods
- Training of object detection, object recognition, and image retrieval systems, including cascade object detection and bag-of-features methods
- Camera calibration for single and stereo cameras, including automatic checkerboard detection and an app for workflow automation
- Stereo vision, including rectification, disparity calculation, and 3-D reconstruction
- 3-D point cloud processing, including I/O, visualization, registration, denoising, and geometric shape fitting
- Feature detection, extraction, and matching
- Support for C-code generation and fixed-point arithmetic with code generation products

Computer Vision Algorithms and Video Processing

- “Computer Vision Capabilities” on page 2-2
- “Video Processing in MATLAB” on page 2-3
- “Computer Vision System Toolbox Preferences” on page 2-4

Computer Vision Capabilities

Computer Vision System Toolbox provides algorithms and tools for the design and simulation of computer vision and video processing systems. The toolbox includes algorithms for feature extraction, motion detection, object detection, object tracking, stereo vision, video processing, and video analysis. Tools include video file I/O, video display, drawing graphics, and compositing. Capabilities are provided as MATLAB functions, MATLAB System objects, and Simulink blocks. For rapid prototyping and embedded system design, the system toolbox supports fixed-point arithmetic and C-code generation.

The link below provides an overview video of Computer Vision System Toolbox capabilities and applications:

Computer Vision System Toolbox capabilities

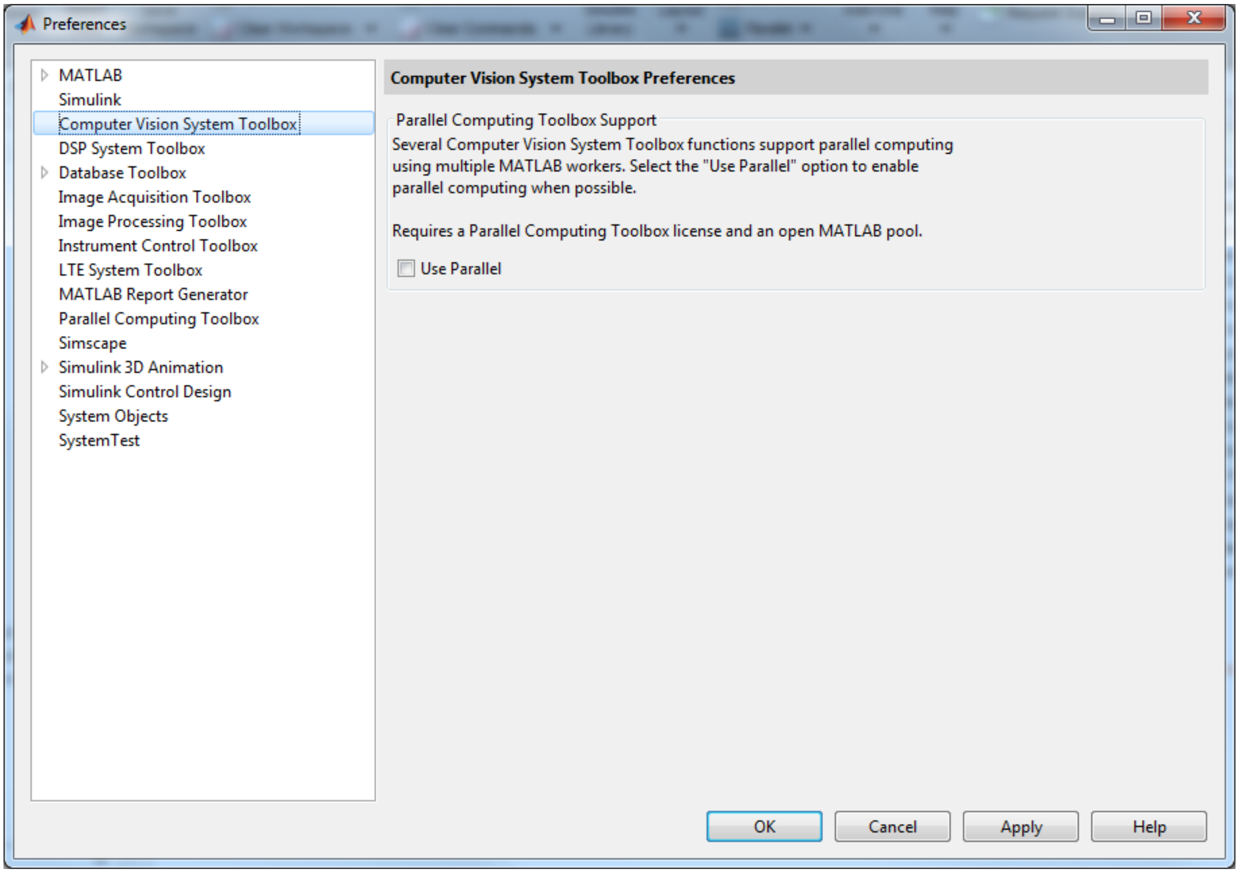
Video Processing in MATLAB

Computer Vision System Toolbox provides algorithms and tools for video processing workflows. You can read and write from common video formats, perform common video processing algorithms such as deinterlacing and chroma-resampling, and display results with text and graphics burnt in to the video. Video processing in MATLAB uses System objects, which avoids excessive memory use by streaming data to and from video files.

The link below provides an introduction video to a typical workflow for motion estimation:
Video processing in MATLAB

Computer Vision System Toolbox Preferences

To open Computer Vision System Toolbox preferences, on the **Home** tab, in the **Environment** section, click **Preferences**. Select **Computer Vision System Toolbox**.



Computer Vision System Toolbox Preferences

Parallel Computing Toolbox Support

Several Computer Vision System Toolbox functions support parallel computing using multiple MATLAB workers. Select the "Use Parallel" option to enable parallel computing when possible.

Requires a Parallel Computing Toolbox license and an open MATLAB pool.

☐ Use Parallel

Parallel Computing Toolbox Support

Several Computer Vision System Toolbox functions support parallel computing using multiple MATLAB workers. Select the **Use Parallel** check box to enable parallel computing when possible.

Parallel computing functionality requires a Parallel Computing Toolbox™ license and an open MATLAB pool.

The functions and methods listed below take an optional logical input parameter, `'UseParallel'` to control whether the individual function can use `parfor`. Set this logical to `'true'` to enable parallel processing for the function or method.

- bagOfFeatures
- `encode (bagOfFeatures)`
- `trainImageCategoryClassifier`
- imageCategoryClassifier
- `predict (imageCategoryClassifier)`

See `parpool` for details on how to create a special job on a pool of workers, and connect the MATLAB client to the parallel pool.

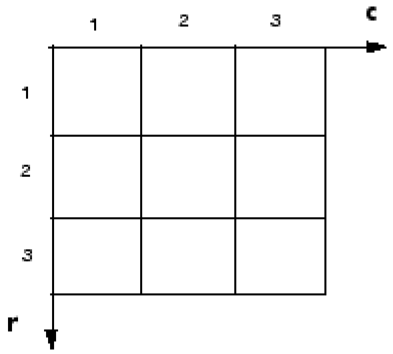
Coordinate Systems

Coordinate Systems

You can specify locations in images using various coordinate systems. Coordinate systems are used to place elements in relation to each other. Coordinates in pixel and spatial coordinate systems relate to locations in an image. Coordinates in 3-D coordinate systems describe the 3-D positioning and origin of the system.

Pixel Indices

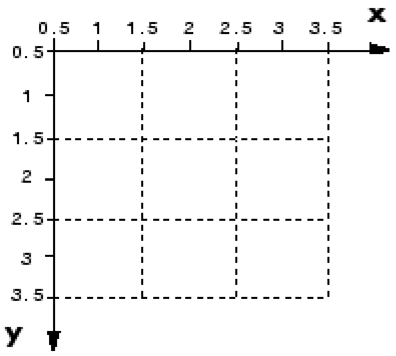
Pixel coordinates enable you to specify locations in images. In the pixel coordinate system, the image is treated as a grid of discrete elements, ordered from top to bottom and left to right.



For pixel coordinates, the number of rows, *r*, downward, while the number of columns, *c*, increase to the right. Pixel coordinates are integer values and range from 1 to the length of the row or column. The pixel coordinates used in Computer Vision System Toolbox software are one-based, consistent with the pixel coordinates used by Image Processing Toolbox™ and MATLAB. For more information on the pixel coordinate system, see “Pixel Indices” in the Image Processing Toolbox documentation.

Spatial Coordinates

Spatial coordinates enable you to specify a location in an image with greater granularity than pixel coordinates. Such as, in the pixel coordinate system, a pixel is treated as a discrete unit, uniquely identified by an integer row and column pair, such as (3,4). In the spatial coordinate system, locations in an image are represented in terms of partial pixels, such as (3.3, 4.7).



For more information on the spatial coordinate system, see “Spatial Coordinates” in the Image Processing Toolbox documentation.

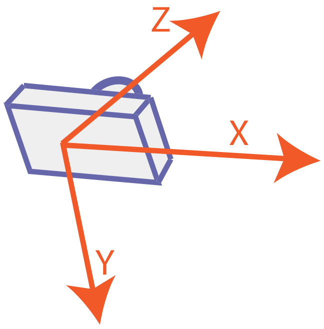
3-D Coordinate Systems

When you reconstruct a 3-D scene, you can define the resulting 3-D points in one of two coordinate systems. In a camera-based coordinate system, the points are defined relative to the center of the camera. In a calibration pattern-based coordinate system, the points are defined relative to a point in the scene.

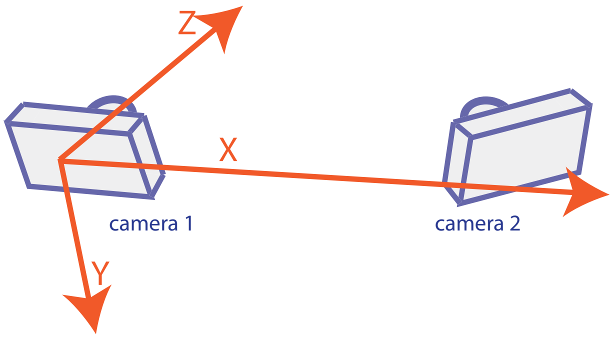
The Computer Vision System Toolbox functions use the right-handed world coordinate system. In this system, the *x*-axis points to the right, the *y*-axis points down, and the *z*-axis points away from the camera. To display 3-D points, use `pcshow`.

Camera-Based Coordinate System

Points represented in a camera-based coordinate system are described with the origin located at the optical center of the camera.



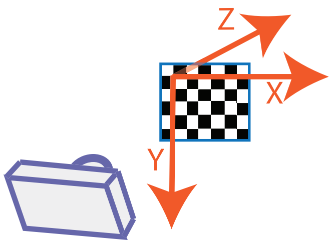
In a stereo system, the origin is located at the optical center of Camera 1.



When you reconstruct a 3-D scene using a calibrated stereo camera, the `reconstructScene` and `triangulate` functions return 3-D points with the origin at the optical center of Camera 1. When you use Kinect® images, the `depthToPointCloud` function returns 3-D points with the origin at the center of the RGB camera.

Calibration Pattern-Based Coordinate System

Points represented in a calibration pattern-based coordinate system are described with the origin located at the (0,0) location of the calibration pattern.



When you reconstruct a 3-D scene from multiple views containing a calibration pattern, the resulting 3-D points are defined in the pattern-based coordinate system. The "Stereo Calibration and Scene Reconstruction" example shows how to reconstruct a 3-D scene from a pair of 2-D images containing a checkerboard pattern.

Related Examples

- "Measuring Planar Objects with a Calibrated Camera"

- “Stereo Calibration and Scene Reconstruction”
- “Depth Estimation From Stereo Video”
- “Structure From Motion From Two Views”

System Objects

- “What Is a System Toolbox?” on page 4-2
- “What Are System Objects?” on page 4-3
- “System Objects vs. MATLAB Functions” on page 4-5
- “System Design and Simulation in MATLAB” on page 4-8
- “System Design and Simulation in Simulink” on page 4-9
- “System Objects in MATLAB Code Generation” on page 4-10
- “System Objects Methods That Support Code Generation” on page 4-16
- “System Objects in Simulink” on page 4-18
- “System Object Methods” on page 4-19
- “System Design in MATLAB Using System Objects” on page 4-22
- “System Design in Simulink Using System Objects” on page 4-29

What Is a System Toolbox?

System Toolbox products provide algorithms and tools for designing, simulating, and deploying dynamic systems in MATLAB and Simulink. These toolboxes contain MATLAB functions, System objects, and Simulink blocks that deliver the same design and verification capabilities across MATLAB and Simulink, enabling more effective collaboration among system designers. Available System Toolbox products include:

- DSP System Toolbox™
- Communications System Toolbox™
- Computer Vision System Toolbox
- Phased Array System Toolbox™

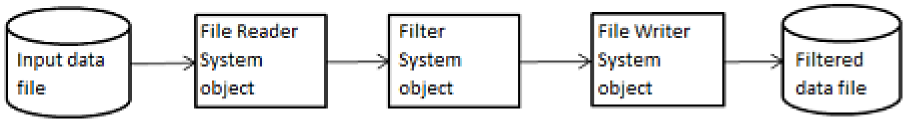
System Toolboxes support floating-point and fixed-point streaming data simulation for both sample- and frame-based data. They provide a programming environment for defining and executing code for various aspects of a system, such as initialization and reset. System Toolboxes also support code generation for a range of system development tasks and workflows, such as:

- Rapid development of reusable IP and test benches
- Sharing of component libraries and systems models across teams
- Large system simulation
- C-code generation for embedded processors
- Finite wordlength effects modeling and optimization
- Ability to prototype and test on real-time hardware

What Are System Objects?

A System object™ is a specialized kind of MATLAB object. System Toolboxes include System objects and most System Toolboxes also have MATLAB functions and Simulink blocks. System objects are designed specifically for implementing and simulating dynamic systems with inputs that change over time. Many signal processing, communications, and controls systems are dynamic. In a dynamic system, the values of the output signals depend on both the instantaneous values of the input signals and on the past behavior of the system. System objects use internal states to store that past behavior, which is used in the next computational step. As a result, System objects are optimized for iterative computations that process large streams of data, such as video and audio processing systems.

For example, you could use System objects in a system that reads data from a file, filters that data and then writes the filtered output to another file. Typically, a specified amount of data is passed to the filter in each loop iteration. The file reader object uses a state to keep track of where in the file to begin the next data read. Likewise, the file writer object keeps tracks of where it last wrote data to the output file so that data is not overwritten. The filter object maintains its own internal states to assure that the filtering is performed correctly. This diagram represents a single loop of the system.



Many System objects support:

- Fixed-point arithmetic (requires a Fixed-Point Designer™ license)
- C code generation (requires a MATLAB Coder™ or Simulink Coder license)
- HDL code generation (requires an HDL Coder™ license)
- Executable files or shared libraries generation (requires a MATLAB Compiler™ license)

Note: Check your product documentation to confirm fixed-point, code generation, and MATLAB Compiler support for the specific System objects you want to use.

In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See “Define System Objects”.

System Objects vs. MATLAB Functions

In this section...

"System Objects vs. MATLAB Functions" on page 4-5
"Process Audio Data Using Only MATLAB Functions Code" on page 4-5
"Process Audio Data Using System Objects" on page 4-6

System Objects vs. MATLAB Functions

Many System objects have MATLAB function counterparts. For simple, one-time computations use MATLAB functions. However, if you need to design and simulate a system with many components, use System objects. Using System objects is also appropriate if your computations require managing internal states, have inputs that change over time or process large streams of data.

Building a dynamic system with different execution phases and internal states using only MATLAB functions would require complex programming. You would need code to initialize the system, validate data, manage internal states, and reset and terminate the system. System objects perform many of these managerial operations automatically during execution. By combining System objects in a program with other MATLAB functions, you can streamline your code and improve efficiency.

Process Audio Data Using Only MATLAB Functions Code

This example shows how to write MATLAB function-only code for reading audio data.

The code reads audio data from a file, filters it, and then plays the filtered audio data. The audio data is read in frames. This code produces the same result as the System objects code in the next example, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Obtain the total number of samples and the sampling rate from the source file.

```
audioInfo = audioinfo(fname);
maxSamples = audioInfo.TotalSamples;
fs = audioInfo.SampleRate;
```

Define the filter to use.

```
b = fir1(160,.15);
```

Initialize the filter states.

```
z = zeros(1,numel(b)-1);
```

Define the amount of audio data to process at one time, and initialize the while loop index.

```
frameSize = 1024;
nIdx = 1;
```

Define the while loop to process the audio data.

```
while nIdx <= maxSamples(1)-frameSize+1
    audio = audioread(fname,[nIdx nIdx+frameSize-1]);
    [y,z] = filter(b,1,audio,z);
    sound(y,fs);
    nIdx = nIdx+frameSize;
end
```

The loop uses explicit indexing and state management, which can be a tedious and error-prone approach. You must have detailed knowledge of the states, such as, sizes and data types. Another issue with this MATLAB-only code is that the `sound` function is not designed to run in real time. The resulting audio is very choppy and barely audible.

Process Audio Data Using System Objects

This example shows how to write System objects code for reading audio data.

The code uses System objects from the DSP System Toolbox software to read audio data from a file, filter it, and then play the filtered audio data. This code produces the same result as the MATLAB code shown previously, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Define the System object to read the file.

```
audioIn = dsp.AudioFileReader(fname,'OutputDataType','single');
```

Define the System object to filter the data.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Define the System object to play the filtered audio data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

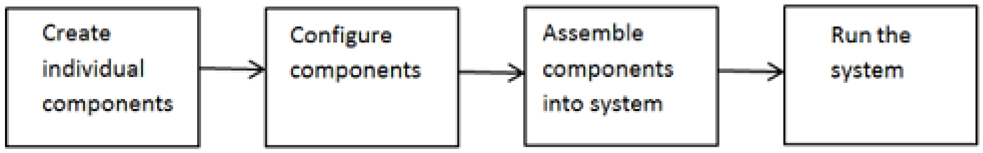
Define the while loop to process the audio data.

```
while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
end
```

This System objects code avoids the issues present in the MATLAB-only code. Without requiring explicit indexing, the file reader object manages the data frame sizes while the filter manages the states. The audio player object plays each audio frame as it is processed.

System Design and Simulation in MATLAB

System objects allow you to design and simulate your system in MATLAB. You use System objects in MATLAB as shown in this diagram.



1. Create individual components — Create the System objects to use in your system. See "Create Components for Your System" on page 4-22 for information. In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See "Define System Objects".
2. Configure components — If necessary, change the objects' property values to model your particular system. All System object properties have default values that you may be able to use without changing them. See "Configure Components for Your System" on page 4-23 for information.
3. Assemble components into system — Write a MATLAB program that includes those System objects, connecting them using MATLAB variables as inputs and outputs to simulate your system. See "Assemble Components to Create Your System" on page 4-25 for information.
4. Run the system — Run your program, which uses the `step` method to run your system's System objects. You can change tunable properties while your system is running. See "Run Your System" on page 4-26 and "Reconfigure Your System During Runtime" on page 4-27 for information.

System Design and Simulation in Simulink

You can use System objects in your model to simulate in Simulink.

1 Create a System object to be used in your model. See "Define New Kinds of System Objects for Use in Simulink" on page 4-29 for information.
2 Test your new System object in MATLAB. See "Test New System Objects in MATLAB" on page 4-34
3 Add the System object to your model using the `MATLAB System` block. See "Add System Objects to Your Simulink Model" on page 4-35 for information.
4 Add other Simulink blocks as needed and connect the blocks to construct your system.
5 Run the system

System Objects in MATLAB Code Generation

In this section...

“System Objects in Generated Code” on page 4-10
“System Objects in codegen” on page 4-14
“System Objects in the MATLAB Function Block” on page 4-15
“System Objects in the MATLAB System Block” on page 4-15
“System Objects and MATLAB Compiler Software” on page 4-15

System Objects in Generated Code

You can generate C/C++ code in MATLAB from your system that contains System objects by using the MATLAB Coder product. Using this product, you can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms. You do not need the MATLAB Coder product to generate code in Simulink.

System Objects Code with Persistent Objects for Code Generation

```
function ex_system_codegen
% Find corresponding interest points between a pair of images using local
% neighborhoods.

% Declare System objects as persistent.
persistent colorSpaceConverter

% Initialize persistent System objects only once
% Do this with 'if isempty(persistent variable).'
% This condition will be false after the first time.
if isempty(colorSpaceConverter)

    % Create system objects. Pass property value arguments as constructor
    % arguments. Property values must be constants during compile time.

    colorSpaceConverter = vision.ColorSpaceConverter('Conversion',...
            'RGB to intensity');
end

% Declare functions called into MATLAB that do not generate
% code as extrinsic.
coder.extrinsic('imread');

% The output of an extrinsic function is an mxArray - also called a MATLAB
% array. To use mxArrays returned by extrinsic functions, assign the
% mxArray to a variable whose type and size is defined.
imgLeft  = zeros([300 400 3],'uint8');
imgRight = zeros([300 400 3],'uint8');

% Call extrinsic function
imgLeft  = imread('viprectification_deskLeft.png');
imgRight = imread('viprectification_deskRight.png');

% Convert RGB to grayscale
I1 = step(colorSpaceConverter,imgLeft);
I2 = step(colorSpaceConverter,imgRight);

% Find corners
points1 = detectHarrisFeatures(I1);
points2 = detectHarrisFeatures(I2);

% Extract neighborhood features
[features1, valid_points1] = extractFeatures(I1,points1);
[features2, valid_points2] = extractFeatures(I2,points2);

% Match features
index_pairs = matchFeatures(features1, features2);

% Retrieve locations of corresponding points for each image
matchedPoints1 = valid_points1.Location(index_pairs(:,1),:);
matchedPoints2 = valid_points2.Location(index_pairs(:,2),:);

% Visualize corresponding points
coder.extrinsic('showMatchedFeatures')
figure;
showMatchedFeatures(I1,I2,matchedPoints1,matchedPoints2);
```

For another detailed code generation example, see “Generate Code for MATLAB Handle Classes and System Objects” in the MATLAB Coder product documentation.

Usage Rules and Limitations for System Objects for Generating Code

The following usage rules and limitations apply to using System objects in code generated from MATLAB.

Object Construction and Initialization

- If objects are stored in persistent variables, initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty( )`.
- Set arguments to System object constructors as compile-time constants.
- You cannot initialize System objects properties with other MATLAB class objects as default values in code generation. You must initialize these properties in the constructor.

Inputs and Outputs

- System objects accept a maximum of 32 inputs. A maximum of 8 dimensions per input is supported.
- The data type of the inputs should not change.
- If you want the size of inputs to change, verify that variable-size is enabled. Code generation support for variable-size data also requires that the `Enable variable sizing` option is enabled, which is the default in MATLAB.

  Note: Variable-size properties in `MATLAB Function` block in Simulink are not supported. System objects predefined in the software do not support variable-size if their data exceeds the `DynamicMemoryAllocationThreshold` value.

- Do not set System objects to become outputs from the `MATLAB Function` block.
- Do not use the Save and Restore Simulation State as SimState option for any System object in a `MATLAB Function` block.
- Do not pass a System object as an example input argument to a function being compiled with `codegen`.
- Do not pass a System object to functions declared as extrinsic (functions called in interpreted mode) using the `coder.extrinsic` function. System objects returned from extrinsic functions and scope System objects that automatically become extrinsic can be used as inputs to another extrinsic function, but do not generate code.

Tunable and Nontunable Properties

- The value assigned to a nontunable property must be a constant and there can be at most one assignment to that property (including the assignment in the constructor).
- For most System objects, the only time you can set their nontunable properties during code generation is when you construct the objects.

  - For System objects that are predefined in the software, you can set their tunable properties at construction time or using dot notation after the object is locked.
  - For System objects that you define, you can change their tunable properties at construction time or using dot notation during code generation. For `getNumInputsImpl` and `getNumOutputsImpl` methods, if you set the return argument from an object property, that object property must have the `Nontunable` attribute.
- Objects cannot be used as default values for properties.
- In MATLAB simulations, default values are shared across all instances of an object. Two instances of a class can access the same default value if that property has not been overwritten by either instance.

Cell Arrays and Global Variables

- System objects can contain cell arrays, but cell arrays cannot contain System objects.
- Global variables are allowed in a System object, unless you will be using that System object in Simulink via the MATLAB System block. To avoid syncing global variables between a MEX file and the workspace, use a coder configuration object. For example:

  ```
  f = coder.MEXConfig;
  f.GlobalSyncMethod = 'NoSync'
  ```
  Then, include `'-config f'` in your `codegen` command.

Methods

- Code generation support is available only for these System object methods:
  - `get`
  - `getNumInputs`
  - `getNumOutputs`
  - `isDone` (for sources only)
  - `isLocked`
  - `release`
  - `reset`
  - `set` (for tunable properties)
  - `step`
- For System objects that you define,

Code generation support is available only for these methods:

- `getDiscreteStateImpl`
- `getNumInputsImpl`
- `getNumOutputsImpl`
- `infoImpl`
- `isDoneImpl`
- `isInputDirectFeedThroughImpl`
- `outputImpl`
- `processTunedPropertiesImpl`
- `releaseImpl` — Code is not generated automatically for the this method. To release an object, you must explicitly call the `release` method in your code.
- `resetImpl`
- `setupImpl`
- `stepImpl`
- `updateImpl`
- `validateInputsImpl`
- `validatePropertiesImpl`

- Code generation support for using dot notation depends on whether the System object is predefined in the software or is one that you defined.
  - For System objects that are predefined in the software, you cannot use dot notation to call methods.
  - For System objects that you define, you can use dot notation or function call notation, with the System object as first argument, to call methods.

System Objects in codegen

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See “Getting Started with MATLAB Coder” and “MATLAB Classes” for more information.

Note: Most, but not all, System objects support code generation. Refer to the particular object's reference page for information.

System Objects in the MATLAB Function Block

Using the `MATLAB Function` block, you can include any System object and any MATLAB language function in a Simulink model. This model can then generate embeddable code. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see "What Is a MATLAB Function Block?" in the Simulink documentation.

System Objects in the MATLAB System Block

Using the `MATLAB System` block, you can include in a Simulink model individual System objects that you create with a class definition file . The model can then generate embeddable code. For more information, see "What Is the MATLAB System Block?" in the Simulink documentation.

System Objects and MATLAB Compiler Software

MATLAB Compiler software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

System Objects Methods That Support Code Generation

In this section...

"Code Generation Supported System Objects Methods" on page 4-16
"Simulation-Only System Objects Methods" on page 4-16

Code Generation Supported System Objects Methods

Only the following methods are supported in code generation.

- getDiscreteStateImpl
- getNumInputsImpl
- getNumOutputsImpl
- isDoneImpl
- infoImpl
- isInputDirectFeedthroughImpl
- outputImpl
- processTunedPropertiesImpl
- releaseImpl — Code is not generated automatically for the this method. To release an object, you must explicitly call the `release` method in your code.
- resetImpl
- setupImpl
- stepImpl
- updateImpl
- validateInputsImpl
- validatePropertiesImpl

Simulation-Only System Objects Methods

The following methods are for simulation only and do not support code generation.

- getDiscreteStateSpecificationImpl
- getHeaderImpl

- getInputNamesImpl
- getIconImpl
- getOutputDataTypeImpl
- getOutputNamesImpl
- getOutputSizeImpl
- getPropertyGroupsImpl
- isInactivePropertyImpl
- isOutputComplexImpl
- loadObjectImpl
- propagatedInputComplexity
- propagatedInputDataType
- propagatedInputFixedSize
- propagatedInputSize
- saveObjectImpl
- supportsMultipleInstanceImpl

System Objects in Simulink

In this section...
"System Objects in the MATLAB Function Block" on page 4-18
"System Objects in the MATLAB System Block" on page 4-18

System Objects in the MATLAB Function Block

You can include System object code in Simulink models using the `MATLAB Function` block. Your function can include one or more System objects. Portions of your system may be easier to implement in the MATLAB environment than directly in Simulink. Many System objects have Simulink block counterparts with equivalent functionality. Before writing MATLAB code to include in a Simulink model, check for existing blocks that perform the desired operation.

System Objects in the MATLAB System Block

You can include individual System objects that you create with a class definition file into Simulink using the `MATLAB System` block. This provides one way to add your own algorithm blocks into your Simulink models. For information, see "System Object Integration" in the Simulink documentation.

System Object Methods

In this section...

"What Are System Object Methods?" on page 4-19
"The Step Method" on page 4-19
"Common Methods" on page 4-20

What Are System Object Methods?

After you create a System object, you use various object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<method>(<handle>)`, such as `step(H)`, plus possible extra input arguments.

System objects use a minimum of two commands to process data—a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

The Step Method

The `step` method is the key System object method. You use `step` to process data using the algorithm defined by that object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object. For more information about the step method and other available methods, see the descriptions in "Common Methods" on page 4-20.

Common Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
`step`	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the `step` method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The `step` method returns regular MATLAB variables. Example: `Y = step(H,X)`
`release`	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. For System objects, use the `release` method instead of a destructor.
`reset`	Resets the internal states of a locked object to the initial values for that object and leaves the object locked
`getNumInputs`	Returns the number of inputs (excluding the object itself) expected by the `step` method. This number varies for an object depending on whether any properties enable additional inputs.
`getNumOutputs`	Returns the number of outputs expected from the `step` method. This number varies for an object depending on whether any properties enable additional outputs.
`getDiscreteState`	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the `step` method on it or after you have released the object), the states are empty. If the object has no discrete states, `getDiscreteState` returns an empty structure.
`clone`	Creates another object of the same type with the same property values
`isLocked`	Returns a logical value indicating whether the object is locked.

Method	Description
`isDone`	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns `false`.
`info`	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.

For a complete list of methods for writing new System objects, see “System Objects Methods for Defining New Objects”.

System Design in MATLAB Using System Objects

In this section...

"Create Components for Your System" on page 4-22

"Configure Components for Your System" on page 4-23

"Assemble Components to Create Your System" on page 4-25

"Run Your System" on page 4-26

"Reconfigure Your System During Runtime" on page 4-27

Create Components for Your System

This example shows how to create components for a system that finds the edges of objects in a video stream.

This example shows how to use System objects that are predefined in the software. You can also create your own System objects (see "Define System Objects"). If you use a function to create and use a System object, specify the object creation using conditional code. This will prevent errors if that function is called within a loop.

This example shows how to set up your system. The particular predefined components you need are:

- `vision.VideoFileReader` — Read the file of video data
- `vision.EdgeDetector` — Detect the edges in the video data
- `vision.AlphaBlender` — Overlay edges onto the original video images
- `vision.VideoPlayer` — Play the video

First, you create the component objects, using default property settings. You create three `VideoPlayer` objects to play the original video, the edges, and the edges overlaid on the original video.

```
hVideoSrc = vision.VideoFileReader;
hEdge = vision.EdgeDetector;
hAB = vision.AlphaBlender;
hVideoOrig = vision.VideoPlayer;
hVideoEdges = vision.VideoPlayer;
hVideoOverlay = vision.VideoPlayer
```

Next, you configure each System object for your system.

Configure Components for Your System

When to Configure Components

If you did not set an object's properties when you created it and do not want to use default values, you must explicitly set those properties. Some properties allow you to change their values while your system is running. See “Reconfigure Your System During Runtime” on page 4-27 for information.

Most properties are independent of each other. However, some System object properties enable or disable another property or limit the values of another property. To avoid errors or warnings, you should set the controlling property before setting the dependent property.

Display Component Property Values

To display the current property values for an object, type that object’s handle name at the command line (such as `audioIn`). To display the value of a specific property, type `objecthandle.propertyname` (such as `audioIn.FileName`).

Configure Component Property Values

This example shows how to configure System object property values.

For the video file reader object, specify the file to read and set the image color space.

```
hVideoSrc.Filename = 'vipmen.avi';
hVideoSrc.ImageColorSpace = 'Intensity';
```

For the edge detector object, specify the edge detection method, the threshold and threshold source, and whether to use edge thinning.

```
hEdge.Method = 'Prewitt';
hEdge.ThresholdSource = 'Property';
hEdge.Threshold = 15/256;
hEdge.EdgeThinning = true;
```

For the alpha blender object, specify the type of operation to use.

```
hAB.Operation = 'Highlight selected pixels';
```

For the video player objects, specify the names, the window size, and the window position.

```
WindowSize = [190 150];

hVideoOrig.Name = 'Original';
hVideoOrig.Position = [10 hVideoOrig.Position(2) WindowSize];

hVideoEdges.Name = 'Edges';
hVideoEdges.Position = [210 hVideoOrig.Position(2) WindowSize];

hVideoOverlay.Name = 'Overlay';
hVideoOverlay.Position = [410 hVideoOrig.Position(2) WindowSize];
```

Create and Configure Components at the Same Time

This example shows how to create your System object components and configure the desired properties at the same time. To avoid errors or warnings for dependent properties, you should set the controlling property before setting the dependent property. Use this procedure if you have not already created your components.

For the video file reader object, specify the file to read and set the image color space.

```
hVideoSrc = vision.VideoFileReader('vipmen.avi', ...
     'ImageColorSpace', 'Intensity');
```

For the edge detector object, specify the edge detection method, the threshold and threshold source, and whether to use edge thinning.

```
hEdge = vision.EdgeDetector('Method','Prewitt',...
     'ThresholdSource','Property', ...
     'Threshold',15/256,'EdgeThinning',true);
```

For the alpha blender object, specify the type of operation to use.

```
hAB = vision.AlphaBlender('Operation', 'Highlight selected pixels');
```

For the video player objects, specify the names, the window size, and the window position.

```
WindowSize = [190 150];
hVideoOrig = vision.VideoPlayer('Name', 'Original');
hVideoOrig.Position = [10 hVideoOrig.Position(2) WindowSize];

hVideoEdges = vision.VideoPlayer('Name', 'Edges');
```

```
hVideoEdges.Position = [210 hVideoOrig.Position(2) WindowSize];

hVideoOverlay = vision.VideoPlayer('Name', 'Overlay');
hVideoOverlay.Position = [410 hVideoOrig.Position(2) WindowSize];
```

After you create the components, you can assemble them in your system.

Assemble Components to Create Your System

- "Connect Inputs and Outputs" on page 4-25
- "Code for the Whole System" on page 4-25

Connect Inputs and Outputs

After you have determined the components you need and have created and configured your System objects, assemble your system. You use the System objects like other MATLAB variables and include them in MATLAB code. You can pass MATLAB variables into and out of System objects.

The main difference between using System objects and using functions is the `step` method. The `step` method is the processing command for each System object and is customized for that specific System object. This method initializes your objects and controls data flow and state management of your system. You typically use `step` within a loop.

You use the output from an object's `step` method as the input to another object's `step` method. For some System objects, you can use properties of those objects to change the number of inputs or outputs. To verify that the appropriate number of input and outputs are being used, you can use `getNumInputs` and `getNumOutputs` on any System object. For information on all available System object methods, see "System Object Methods" on page 4-19.

Code for the Whole System

This example shows how to write the full code for edge detection.

You can type this code on the command line or put it into a program file.

```
hVideoSrc = vision.VideoFileReader('vipmen.avi', ...
     'ImageColorSpace','Intensity');
hEdge = vision.EdgeDetector('Method','Prewitt', ...
      'ThresholdSource','Property', ...
      'Threshold',35/256,'EdgeThinning',true);
hAB = vision.AlphaBlender('Operation','Highlight selected pixels');

WindowSize = [190 150];
hVideoOrig = vision.VideoPlayer('Name','Original');
hVideoOrig.Position = [10 hVideoOrig.Position(2) WindowSize];

hVideoEdges = vision.VideoPlayer('Name','Edges');
hVideoEdges.Position = [210 hVideoOrig.Position(2) WindowSize];

hVideoOverlay = vision.VideoPlayer('Name','Overlay');
hVideoOverlay.Position = [410 hVideoOrig.Position(2) WindowSize];

while ~isDone(hVideoSrc)
    frame     = step(hVideoSrc);          % Read input video
    edges     = step(hEdge, frame);       % Edge detection
    composite = step(hAB, frame, edges)   % AlphaBlender

    step(hVideoOrig,frame);              % Display original
    step(hVideoEdges,edges);             % Display edges
    step(hVideoOverlay,composite);       % Display edges overlayed
end
release(hVideoSrc);
```

The while loop uses the `isDone` method to read through the entire file. The `step` method is used on each object inside the loop.

Now, you are ready to run your system.

Run Your System

- "How to Run Your System" on page 4-26
- "What You Cannot Change While Your System Is Running" on page 4-26

How to Run Your System

Run your code either by typing directly at the command line or running a file containing your program. When you run the code for your system, the `step` method instructs each object to process data through that object.

What You Cannot Change While Your System Is Running

The first call to the `step` method initializes and then locks your object. When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. Use the `isLocked` method to verify whether an object is locked. When the object is locked, you cannot change:

- Number of inputs or outputs
- Data type of inputs or outputs
- Data type of any tunable property
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data
- Value of any nontunable property

To make changes to your system while it is running, see "Reconfigure Your System During Runtime" on page 4-27.

Reconfigure Your System During Runtime

- "When Can You Change Component Properties?" on page 4-27
- "Change a Tunable Property in Your System" on page 4-27
- "Change Input Complexity or Dimensions" on page 4-28

When Can You Change Component Properties?

When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. You can use `isLocked` on any System object to verify whether it is locked or not. When processing is complete, you can use the `release` method to unlock a System object.

Some object properties are *tunable*, which enables you to change them even if the object is locked. Unless otherwise specified, System objects properties are nontunable. Refer to the object's reference page to determine whether an individual property is tunable. Typically, tunable properties are not critical to how the System object processes data.

Change a Tunable Property in Your System

This example shows how to change a tunable property.

You can change the threshold value as your code is running by replacing the while loop with the following while loop. The change takes effect the next time the `step` method is called (such as, at the next iteration of the while loop).

```
while ~isDone(hVideoSrc)
```

```
    frame     = step(hVideoSrc);           % Read input video
    edges     = step(hEdge, frame);        % Edge detection
    composite = step(hAB, frame, edges)    % AlphaBlender

    hEdge.Threshold = hEdge.Threshold-0.0005; % Tune threshold

    step(hVideoOrig,frame);               % Display original
    step(hVideoEdges,edges);              % Display edges
    step(hVideoOverlay,composite);        % Display edges overlayed
end
```

Change Input Complexity or Dimensions

During simulation, some System objects do not allow complex data if the object was initialized with real data. You cannot change any input complexity during code generation.

You can change the value of a tunable property without a warning or error being produced. For all other changes at run time, an error occurs.

System Design in Simulink Using System Objects

In this section...

"Define New Kinds of System Objects for Use in Simulink" on page 4-29
"Test New System Objects in MATLAB" on page 4-34
"Add System Objects to Your Simulink Model" on page 4-35

Define New Kinds of System Objects for Use in Simulink

- "Define System Object with Block Customizations" on page 4-29
- "Define System Object with Nondirect Feedthrough" on page 4-32

A System object is a component you can use to create your system in MATLAB. You can write the code in MATLAB and use that code to create a block in Simulink. To define your own System object, you write a class definition file, which is a text-based MATLAB file that contains the code defining your object. See "System Object Integration" in the Simulink documentation.

Define System Object with Block Customizations

This example shows how to create a System object for use in Simulink. The example performs system identification using a least mean squares (LMS) adaptive filter and is similar to the System Identification Using MATLAB System Blocks Simulink example.

This example shows how to create a class definition text file to define your System object. The code in this example creates a least mean squares (LMS) filter and includes customizations to the block icon and dialog appearance.

Note: Instead of manually creating your class definition file, you can use the **New** > **System Object** > **Simulink Extension** menu option to open a template. This template includes customizations of the System object for use in the Simulink MATLAB System block. You edit the template file, using it as guideline, to create your own System object.

On the first line of the class definition file, specify the name of your System object and subclass from both `matlab.System` and `matlab.system.mixin.CustomIcon`. The `matlab.System` base class enables you to use all the basic System object methods and specify the block input and output names, title, and property groups. The `CustomIcon` mixin class enables the method that lets you specify the block icon.

Add the appropriate basic System object methods to set up, reset, set the number of inputs and outputs, and run your algorithm. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.

- Use the `setupImpl` method to perform one-time calculations and initialize variables.
- Use the `stepImpl` method to implement the block's algorithm.
- Use the `resetImpl` method to reset the state properties or `DiscreteState` properties.
- Use the `getNumInputsImpl` and `getNumOutputsImpl` methods to specify the number of inputs and outputs, respectively.

Add the appropriate `CustomIcon` methods to define the appearance of the MATLAB System block in Simulink. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.

- Use the `getHeaderImpl` method to specify the title and description to display on the block dialog.
- Use the `getPropertyGroupsImpl` method to specify groups of properties to display on the block dialog.
- Use the `getIconImpl` method to specify the text to display on the block icon.
- Use the `getInputNamesImpl` and `getOutputNamesImpl` methods to specify the labels to display for the block input and output ports.

The full class definition file for the least mean squares filter is:

```
classdef lmsSysObj < matlab.System &...
      matlab.system.mixin.CustomIcon
   % lmsSysObj Least mean squares (LMS) adaptive filtering.
   % #codegen

   properties
      % Mu Step size
      Mu = 0.005;
   end

   properties (Nontunable)
      % Weights  Filter weights
      Weights = 0;
      % N  Number of filter weights
      N = 32;
   end
```

```
properties (DiscreteState)
   X;
   H;
end

methods (Access = protected)
   function setupImpl(obj)
      obj.X = zeros(obj.N,1);
      obj.H = zeros(obj.N,1);
   end

   function [y, e_norm] = stepImpl(obj,d,u)
      tmp = obj.X(1:obj.N-1);
      obj.X(2:obj.N,1) = tmp;
      obj.X(1,1) = u;
      y = obj.X'*obj.H;
      e = d-y;
      obj.H = obj.H + obj.Mu*e*obj.X;
      e_norm = norm(obj.Weights'-obj.H);
   end

   function resetImpl(obj)
      obj.X = zeros(obj.N,1);
      obj.H = zeros(obj.N,1);
   end

end

% Block icon and dialog customizations
methods (Static, Access = protected)
   function header = getHeaderImpl
      header = matlab.system.display.Header(...
           'lmsSysObj', ...
           'Title', 'LMS Adaptive Filter');
   end

   function groups = getPropertyGroupsImpl
      upperGroup = matlab.system.display.SectionGroup(...
           'Title','General',...
           'PropertyList',{'Mu'});

      lowerGroup = matlab.system.display.SectionGroup(...
           'Title','Coefficients', ...
           'PropertyList',{'Weights','N'});
```

```
         groups = [upperGroup,lowerGroup];
      end
   end

   methods (Access = protected)
      function icon = getIconImpl(~)
         icon = sprintf('LMS Adaptive\nFilter');
      end
      function [in1name, in2name] = getInputNamesImpl(~)
         in1name = 'Desired';
         in2name = 'Actual';
      end
      function [out1name, out2name] = getOutputNamesImpl(~)
         out1name = 'Output';
         out2name = 'EstError';
      end
   end
end
```

Define System Object with Nondirect Feedthrough

This example shows how to create a System object for use in Simulink. The example performs system identification using a least mean squares (LMS) adaptive filter and uses feedback loops. It is similar to the System Identification Using MATLAB System Blocks Simulink example. For information on feedback loops, see “Use System Objects in Feedback Loops”.

This example shows how to create a class definition text file to define your System object. The code in this example creates an integer delay and includes customizations to the block icon. It implements a System object that you can use for nondirect feedthrough.

On the first line of the class definition file, subclass from `matlab.System` and `matlab.system.mixin.CustomIcon`. The `matlab.System` base class enables you to use all the basic System object methods and specify the block input and output names, title, and property groups. The `CustomIcon` mixin class enables the method that lets you specify the block icon. The `Nondirect` mixin enables the methods that let you specify how the block is updated and what it outputs.

Add the appropriate basic System object methods to set up and reset the object and set and validate the properties. Since this object supports nondirect feedthrough, you do not implement the `stepImpl` method. You implement the `updateImpl` and `outputImpl` methods instead. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.

- Use the `setupImpl` method to initialize some of the object's properties.
- Use the `resetImpl` method to reset the property states.
- Use the `validatePropertiesImpl` method to check that the property values are valid.

Add the following `Nondirect` mixin class methods instead of the `stepImpl` method to specify how the block updates its state and its output. See the reference pages and the full class definition file below for the implementation of each of these methods.

- Use the `outputImpl` method to implement code to calculate the block output.
- Use the `updateImpl` method to implement code to update the block's internal states.
- Use the `isInputDirectFeedthroughImpl` method to specify that the block is not direct feedthrough. Its inputs do not directly affect its outputs.

Add the `getIconImpl` method to define the block icon when it is used in Simulink via the MATLAB System block. See the reference page and the full class definition file below for the implementation of this method.

The full class definition file for the delay is:

```
classdef intDelaySysObj < matlab.System &...
     matlab.system.mixin.Nondirect &...
     matlab.system.mixin.CustomIcon
   % intDelaySysObj Delay input by specified number of samples.
   % #codegen

   properties
      % InitialOutput Initial output
      InitialOutput = 0;
   end

   properties (Nontunable)
      % NumDelays Number of delays
      NumDelays = 1;
   end

   properties (DiscreteState)
      PreviousInput;
   end
```

```
    methods (Access = protected)
       function setupImpl(obj, ~)
          obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
       end

       function [y] = outputImpl(obj, ~)
          % Output does not directly depend on input
          y = obj.PreviousInput(end);
       end

       function updateImpl(obj, u)
          obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
       end

       function flag = isInputDirectFeedthroughImpl(~,~)
          flag = false;
       end

       function validatePropertiesImpl(obj)
          if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
             error('Number of delays must be positive non-zero scalar value.');
          end
          if (numel(obj.InitialOutput)>1)
             error('Initial output must be scalar value.');
          end
       end

       function resetImpl(obj)
          obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
       end

       function icon = getIconImpl(~)
          icon = sprintf('Integer\nDelay');
       end
    end
 end
```

Test New System Objects in MATLAB

1. Create an instance of your new System object. For example, create an instance of the `lmsSysObj`.

   ```
   s = lmsSysObj;
   ```

2 Run the step method on the object multiple times with different inputs. This tests for syntax errors and other possible issues before you add it to Simulink. For example,

```
desired = 0;
actual = 0.2;
step(s,desired,actual);
```

Add System Objects to Your Simulink Model

1 Add your System objects to your Simulink model by using the `MATLAB System` block as described in "Mapping System Objects to Block Dialog Box".
2 Add other Simulink blocks, connect them, and configure any needed parameters to complete your model as described in the Simulink documentation. See the System Identification for an FIR System Using MATLAB System Blocks Simulink example.
3 Run your model in the same way you run any Simulink model.

Strategies for Real-Time Video Processing in Simulink

• "Optimizing Your Implementation" on page 5-2
• "Developing Your Models" on page 5-4

Optimizing Your Implementation

Video processing is computationally intensive, and the ability to perform real-time video processing is affected by the following factors:

- Hardware capability
- Model complexity
- Model implementation
- Input data size

Optimizing your implementation is a crucial step toward real-time video processing. The following tips can help improve the performance of your model:

- Minimize the number of blocks in your model.
- Process only the regions of interest to reduce the input data size.
- Use efficient algorithms or the simplest version of an algorithm that achieves the desired result.
- Use efficient block parameter settings. However, you need to decide whether these settings best suit your algorithm. For example, the most efficient block parameter settings might not yield the most accurate results. You can find out more about individual block parameters and their effect on performance by reviewing specific block reference pages.

  The two following examples show settings that make each block's operation the least computationally expensive:

  - Resize block — **Interpolation method** = `Nearest neighbor`
  - Blocks that support fixed point — On the **Fixed-Point** tab, **Overflow mode** = `Wrap`
- Choose data types carefully.
  - Avoid data type conversions.
  - Use the smallest data type necessary to represent your data to reduce memory usage and accelerate data processing.

    In simulation mode, models with floating-point data types run faster than models with fixed-point data types. To speed up fixed-point models, you must run them in accelerator mode. Simulink contains additional code to process all fixed-point data types. This code affects simulation performance. After you run your model in accelerator mode or generate code for your target using the Simulink Coder, the fixed-point data types are specific to the choices you made for the fixed-point parameters. Therefore, the fixed-point model and generated code run faster.

Developing Your Models

Use the following general process guidelines to develop real-time video processing models to run on embedded targets. By optimizing the model at each step, you improve its final performance.

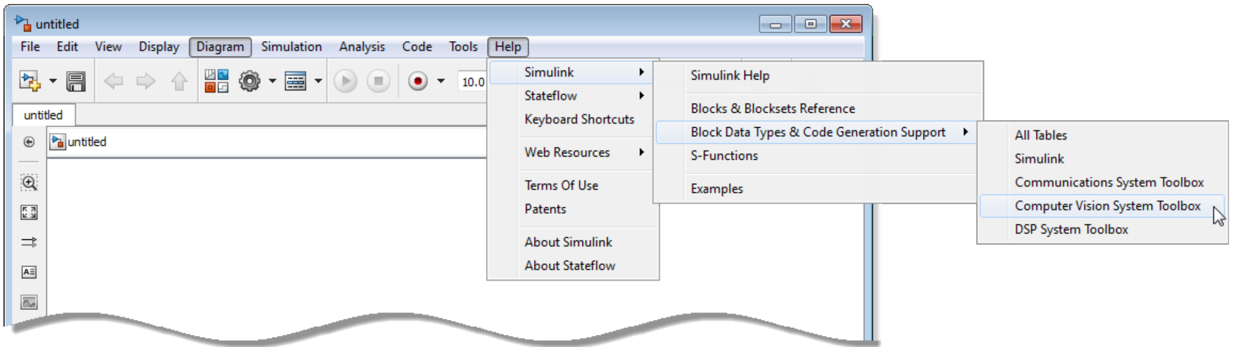
1. Create the initial model and optimize the implementation algorithm. Use floating-point data types so that the model runs faster in simulation mode. If you are working with a floating-point processor, go to step 3.
2. If you are working with a fixed-point processor, gradually change the model data types to fixed point, and run the model after every modification.

   During this process, you can use data type conversion blocks to isolate the floating point sections of the model from the fixed-point sections. You should see a performance improvement if you run the model in accelerator mode.
3. Remove unnecessary sink blocks, including scopes, and blocks that log data to files.
4. Compile the model for deployment on the embedded target.

Data Type Support

Block Data Type Support

The Computer Vision System Toolbox Data Type Support Table is available through the Simulink model Help menu. The table provides information about data type support and code generation coverage for all Computer Vision System Toolbox blocks. Select **Help** > **Simulink**> **Block Data Types & Code Generation Support** > **Computer Vision System Toolbox**.



Fixed-Point Support for MATLAB System Objects

In this section...

"Getting Information About Fixed-Point System Objects" on page 6-3
"Displaying Fixed-Point Properties" on page 6-4
"Setting System Object Fixed-Point Properties" on page 6-5

For information on working with Fixed-Point features, refer to the "Fixed-Point"topic.

Getting Information About Fixed-Point System Objects

System objects that support fixed-point data processing have fixed-point properties, which you can display for a particular object by typing `vision.<ObjectName>.helpFixedPoint` at the command line.

See "Displaying Fixed-Point Properties" on page 6-4 to set the display of System object fixed-point properties.

The following Computer Vision System Toolbox objects support fixed-point data processing.

Fixed-Point Data Processing Support

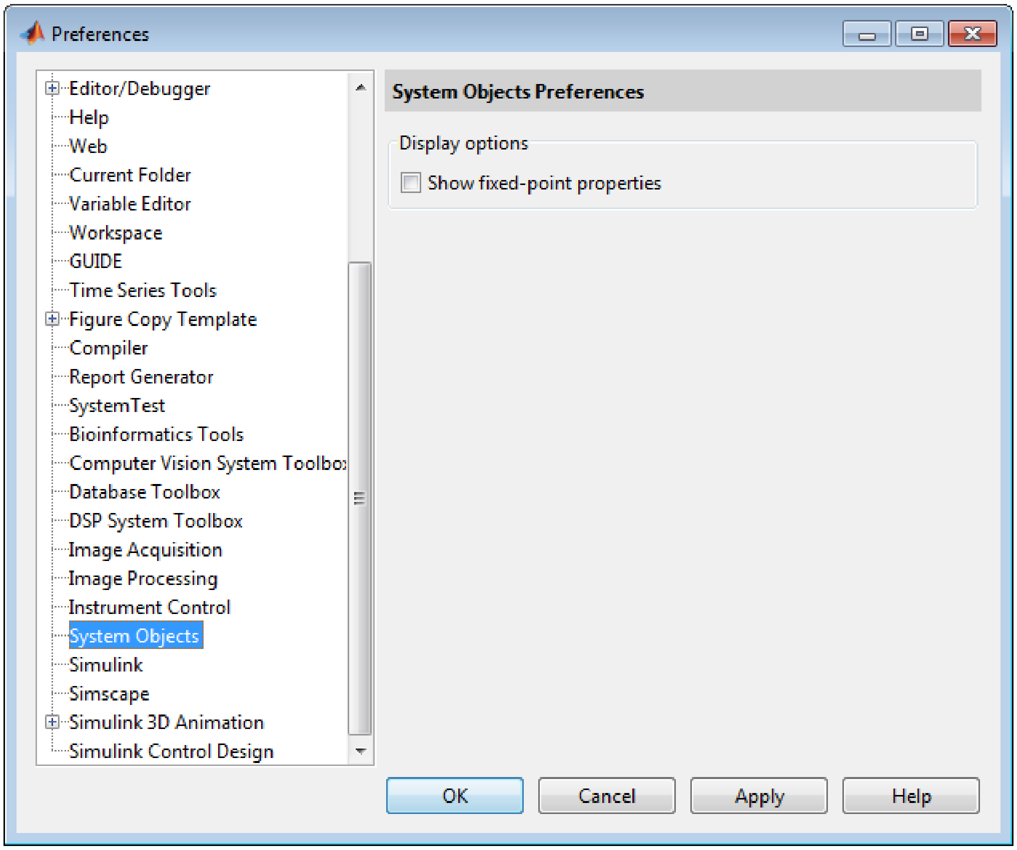
```
vision.AlphaBlender
vision.Autocorrelator
vision.Autothresholder
vision.BlobAnalysis
vision.BlockMatcher
vision.ContrastAdjuster
vision.Convolver
vision.CornerDetector
vision.Crosscorrelator
vision.DCT
vision.Deinterlacer
vision.DemosaicInterpolator
vision.EdgeDetector
vision.FFT
vision.GeometricRotator
vision.GeometricScaler
vision.GeometricTranslator
vision.Histogram
vision.HoughLines
vision.HoughTransform
vision.IDCT
vision.IFFT
vision.ImageDataTypeConverter
vision.ImageFilter
vision.MarkerInserter
vision.Maximum
vision.Mean
vision.Median
vision.MedianFilter
vision.Minimum
vision.OpticalFlow
vision.PSNR
vision.Pyramid
vision.SAD
vision.ShapeInserter
vision.Variance
```

Displaying Fixed-Point Properties

You can control whether the software displays fixed-point properties with either of the following commands:

- `matlab.system.showFixedPointProperties`
- `matlab.system.hideFixedPointProperties`

at the MATLAB command line. These commands set the **Show fixed-point properties** display option. You can also set the display option directly via the MATLAB preferences dialog box. Select the **Preferences** icon from the MATLAB desktop, and then select **System Objects**. Finally, select or deselect **Show fixed-point properties**.



System Objects Preferences

Display options

☐ Show fixed-point properties

If an object supports fixed-point data processing, its fixed-point properties are active regardless of whether they are displayed or not.

Setting System Object Fixed-Point Properties

A number of properties affect the fixed-point data processing used by a System object. Objects perform fixed-point processing and use the current fixed-point property settings when they receive fixed-point input.

You change the values of fixed-point properties in the same way as you change any System object property value. You also use the Fixed-Point Designer `numerictype` object to specify the desired data type as fixed-point, the signedness, and the word- and fraction-lengths.

In the same way as for blocks, the data type properties of many System objects can set the appropriate word lengths and scalings automatically by using full precision. System objects assume that the target specified on the Configuration Parameters Hardware Implementation target is `ASIC/FPGA`.

If you have not set the property that activates a dependent property and you attempt to change that dependent property, a warning message displays. For example, for the `vision.EdgeDetector` object, before you set `CustomProductDataType` to `numerictype(1,16,15)` you must set `ProductDataType` to `'Custom'`.

Note: System objects do not support fixed-point word lengths greater than 128 bits.

For any System object provided in the Toolbox, the fimath settings for any fimath attached to a fi input or a fi property are ignored. Outputs from a System object never have an attached fimath.
