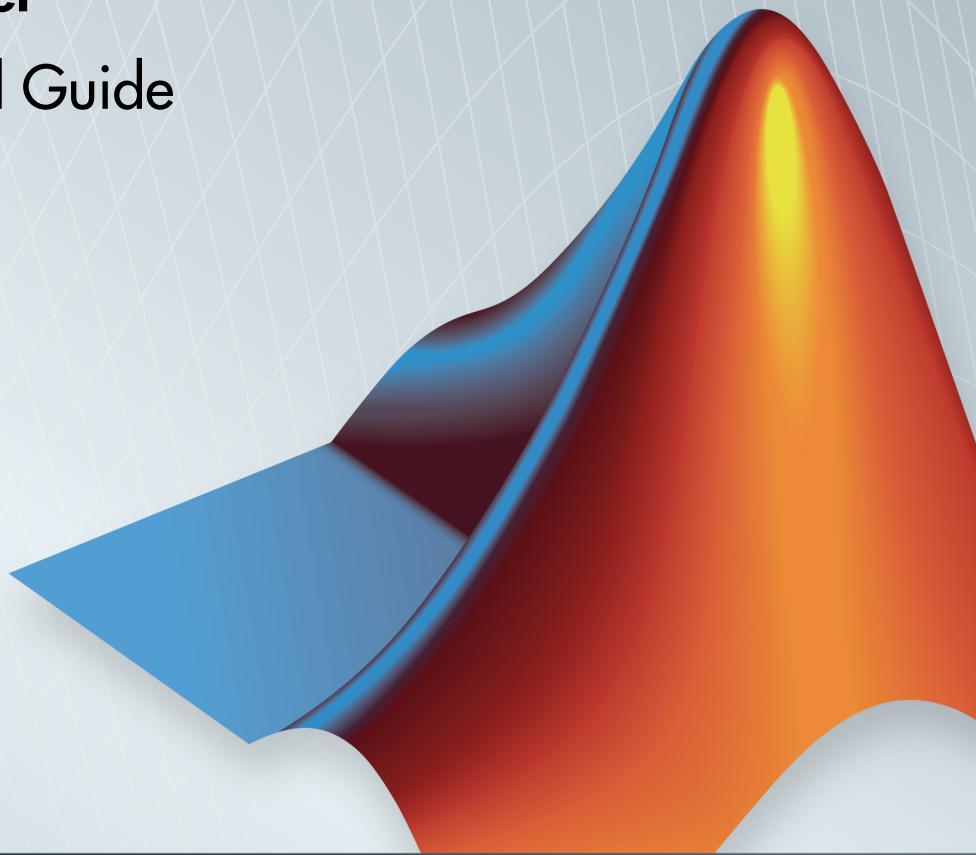


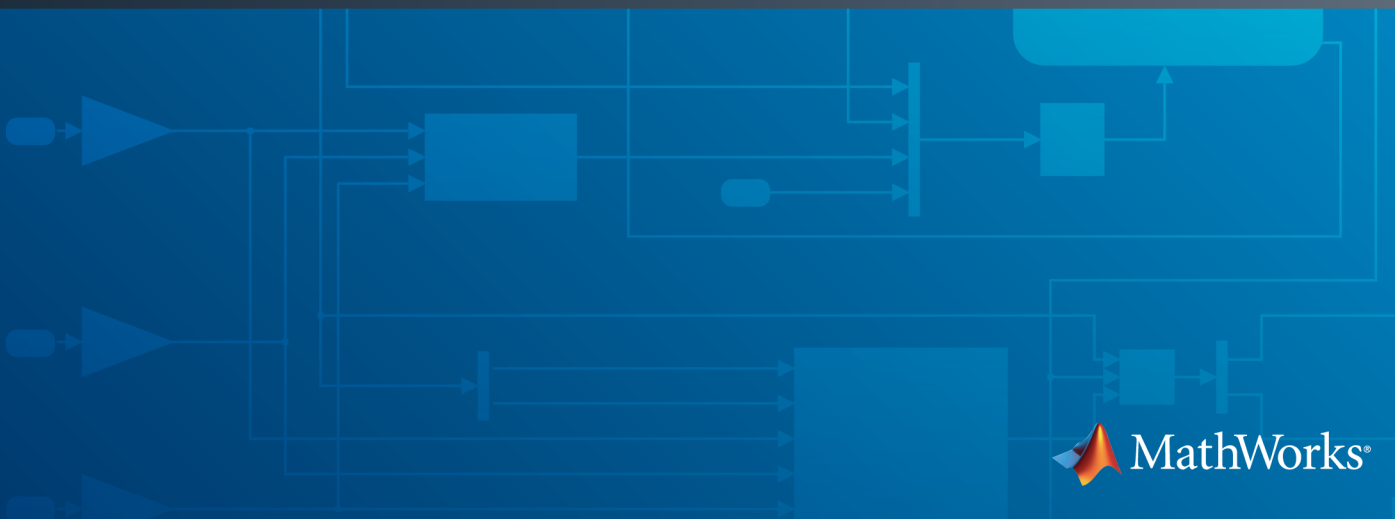
MATLAB[®] Coder[™]

Getting Started Guide

R2014b



MATLAB[®]



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

MATLAB[®] Coder[™] Getting Started Guide

© COPYRIGHT 2011–2014 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

April 2011	Online only	New for R2011a
September 2011	Online only	Revised for Version 2.1 (Release 2011b)
March 2012	Online only	Revised for Version 2.2 (Release 2012a)
September 2012	Online only	Revised for Version 2.3 (Release 2012b)
March 2013	Online only	Revised for Version 2.4 (Release 2013a)
September 2013	Online only	Revised for Version 2.5 (Release 2013b)
March 2014	Online only	Revised for Version 2.6 (Release 2014a)
October 2014	Online only	Revised for Version 2.7 (Release 2014b)

Check Bug Reports for Issues and Fixes

Software is inherently complex and is not free of errors. The output of a code generator might contain bugs, some of which are not detected by a compiler. MathWorks reports critical known bugs brought to its attention on its Bug Report system at www.mathworks.com/support/bugreports/. Use the Saved Searches and Watched Bugs tool with the search phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers.

The bug reports are an integral part of the documentation for each release. Examine periodically all bug reports for a release, as such reports may identify inconsistencies between the actual behavior of a release you are using and the behavior described in this documentation.

In addition to reviewing bug reports, you should implement a verification and validation strategy to identify potential bugs in your design, code, and tools.

Product Overview

1

MATLAB Coder Product Description	1-2
Key Features	1-2
About MATLAB Coder	1-3
When to Use MATLAB Coder	1-3
What You Can Do with the Project Interface	1-3
When to Use the Command Line (codegen function)	1-3
Code Generation for Embedded Software Applications	1-5
Code Generation for Fixed-Point Algorithms	1-6
Installing Prerequisite Products	1-7
Related Products	1-8
Setting Up the C or C++ Compiler	1-9
Expected Background	1-10
Workflow Overview	1-11
See Also	1-11

Tutorials

2

C Code Generation Using the Project Interface	2-2
Learning Objectives	2-2
Tutorial Prerequisites	2-2

Example: The Kalman Filter	2-3
Files for the Tutorial	2-5
Design Considerations When Writing MATLAB Code for Code Generation	2-7
Tutorial Steps	2-8
Key Points to Remember	2-29
Learn More	2-29
C Code Generation at the Command Line	2-31
Learning Objectives	2-31
Tutorial Prerequisites	2-31
Example: The Kalman Filter	2-32
Files for the Tutorial	2-34
Design Considerations When Writing MATLAB Code for Code Generation	2-36
Tutorial Steps	2-37
Key Points to Remember	2-56
Best Practices Used in This Tutorial	2-56
Where to Learn More	2-57
MEX Function Generation at the Command Line	2-59
Learning Objectives	2-59
Tutorial Prerequisites	2-59
Example: Euclidean Minimum Distance	2-60
Files for the Tutorial	2-62
Tutorial Steps	2-63
Key Points to Remember	2-80
Best Practices Used in This Tutorial	2-80
Where to Learn More	2-81

Best Practices for Working with MATLAB Coder

3

Recommended Compilation Options for codegen	3-2
-c Generate Code Only	3-2
-report Generate Code Generation Report	3-2
Testing MEX Functions in MATLAB	3-3

Comparing C Code and MATLAB Code Using Tiling in the MATLAB Editor	3-4
Using Build Scripts	3-5
Check Code Using the MATLAB Code Analyzer	3-6
Separating Your Test Bench from Your Function Code	3-7
Preserving Your Code	3-8
File Naming Conventions	3-9

Product Overview

- “MATLAB Coder Product Description” on page 1-2
- “About MATLAB Coder” on page 1-3
- “Code Generation for Embedded Software Applications” on page 1-5
- “Code Generation for Fixed-Point Algorithms” on page 1-6
- “Installing Prerequisite Products” on page 1-7
- “Related Products” on page 1-8
- “Setting Up the C or C++ Compiler” on page 1-9
- “Expected Background” on page 1-10
- “Workflow Overview” on page 1-11

MATLAB Coder Product Description

Generate C and C++ code from MATLAB code

MATLAB[®] Coder[™] generates standalone C and C++ code from MATLAB code. The generated source code is portable and readable. MATLAB Coder supports a subset of core MATLAB language features, including program control constructs, functions, and matrix operations. It can generate MEX functions that let you accelerate computationally intensive portions of MATLAB code and verify the behavior of the generated code.

Key Features

- ANSI[®]/ISO[®] compliant C and C++ code generation
- MEX function generation for fixed-point and floating-point math
- Project management tool for specifying entry points, input data properties, and other code-generation configuration options
- Static or dynamic memory allocation for variable-size data
- Code generation support for many functions and System objects in Communications System Toolbox[™], DSP System Toolbox[™], Computer Vision System Toolbox[™], and Phased Array System Toolbox[™]
- Support for common MATLAB language features, including matrix operations, subscripting, program controls statements (if, switch, for, while), and structures

About MATLAB Coder

When to Use MATLAB Coder

Use MATLAB Coder to:

- Generate readable, efficient, standalone C/C++ code from MATLAB code.
- Generate MEX functions from MATLAB code to:
 - Accelerate your MATLAB algorithms.
 - Verify generated C code within MATLAB.
- Integrate custom C/C++ code into MATLAB.

What You Can Do with the Project Interface

- Specify the MATLAB files from which you want to generate code
- Specify the data types for the inputs to these MATLAB files
- Select an output type:
 - MEX function
 - C/C++ Static Library
 - C/C++ Dynamic Library
 - C/C++ Executable
- Configure build settings to customize your environment for code generation
- Open the code generation report to view build status, generated code, and compile-time information for the variables and expressions in your MATLAB code

See Also

- “MATLAB Coder Project Set Up Workflow”
- “C Code Generation Using the Project Interface”

When to Use the Command Line (codegen function)

Use the command line if you use build scripts to specify input parameter types and code generation options.

See Also

- The `codegen` function reference page
- “C Code Generation at the Command Line”
- “MEX Function Generation at the Command Line”

Code Generation for Embedded Software Applications

The Embedded Coder[®] product extends the MATLAB Coder product with features that are important for embedded software development. Using the Embedded Coder add-on product, you can generate code that has the clarity and efficiency of professional handwritten code. For example, you can:

- Generate code that is compact and fast, which is essential for real-time simulators, on-target rapid prototyping boards, microprocessors used in mass production, and embedded systems.
- Customize the appearance of the generated code.
- Optimize the generated code for a specific target environment.
- Enable tracing options that help you to verify the generated code.
- Generate reusable, reentrant code.

Code Generation for Fixed-Point Algorithms

Using the Fixed-Point Designer™ product, you can generate:

- MEX functions to accelerate fixed-point algorithms.
- Fixed-point code that provides a bit-wise match to MEX function results.

Installing Prerequisite Products

To generate C and C++ code using MATLAB Coder, you must install the following products:

- MATLAB

Note: If MATLAB is installed on a path that contains non 7-bit ASCII characters, such as Japanese characters, MATLAB Coder might not work because it cannot locate code generation library functions.

- MATLAB Coder
- C or C++ compiler

For most platforms, a default compiler is supplied with MATLAB.

MATLAB Coder automatically locates and uses a supported installed compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks® Web site.

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

For instructions on installing MathWorks products, see the MATLAB installation documentation for your platform. If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window.

Related Products

- Embedded Coder
- Simulink[®] Coder

Setting Up the C or C++ Compiler

MATLAB Coder automatically locates and uses a supported installed compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks Web site.

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

Expected Background

You should be familiar with :

- MATLAB software
- MEX functions

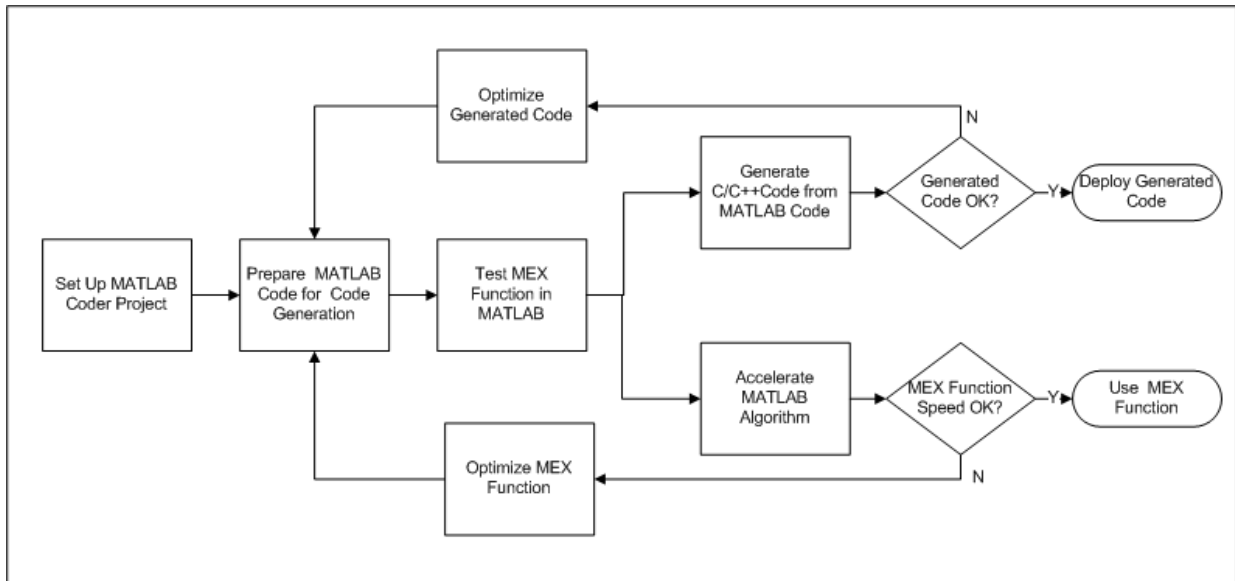
For more information, see “Introducing MEX-Files” in the MATLAB External Interfaces documentation.

- C/C++ programming concepts

To generate C code on embedded targets, you should also be familiar with how to re-compile the generated code in the target environment.

To integrate the generated code into external applications, you should be familiar with the C/C++ compilation and linking process.

Workflow Overview



See Also

- “MATLAB Coder Project Set Up Workflow”
- “Workflow for Preparing MATLAB Code for Code Generation”
- “Workflow for Testing MEX Functions in MATLAB”
- “Code Generation Workflow”
- “Workflow for Accelerating MATLAB Algorithms”
- “Optimization Strategies”
- “Accelerate MATLAB Algorithms”

Tutorials

- “C Code Generation Using the Project Interface” on page 2-2
- “C Code Generation at the Command Line” on page 2-31
- “MEX Function Generation at the Command Line” on page 2-59

C Code Generation Using the Project Interface

In this section...
“Learning Objectives” on page 2-2
“Tutorial Prerequisites” on page 2-2
“Example: The Kalman Filter” on page 2-3
“Files for the Tutorial” on page 2-5
“Design Considerations When Writing MATLAB Code for Code Generation” on page 2-7
“Tutorial Steps” on page 2-8
“Key Points to Remember” on page 2-29
“Learn More” on page 2-29

Learning Objectives

In this tutorial, you will learn how to:

- Create and set up a MATLAB Coder project.
- Automatically generate a MEX function from your MATLAB code and use this MEX function to validate your algorithm in MATLAB before generating C code.
- Automatically generate C code from your MATLAB code.
- Define function input properties.
- Specify variable-size inputs when generating code.
- Specify code generation properties.
- Generate a code generation report that you can use to debug your MATLAB code and verify that it is suitable for code generation.

Tutorial Prerequisites

- “What You Need to Know” on page 2-2
- “Required Products” on page 2-3

What You Need to Know

To complete this tutorial, you should have basic familiarity with MATLAB software.

Required Products

To complete this tutorial, you must install the following products:

- MATLAB
- MATLAB Coder
- C compiler

For most platforms, a default compiler is supplied with MATLAB.

MATLAB Coder automatically locates and uses a supported installed compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks Web site.

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

For instructions on installing MathWorks products, see the MATLAB installation documentation for your platform. If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window.

Example: The Kalman Filter

- “Description” on page 2-3
- “Algorithm” on page 2-4
- “Filtering Process” on page 2-5
- “Reference” on page 2-5

Description

This section describes the example used by the tutorial. You do not have to be familiar with the algorithm to complete the tutorial.

The example for this tutorial uses a Kalman filter to estimate the position of an object moving in a two-dimensional space from a series of noisy inputs based on past positions. The position vector has two components, x and y , indicating its horizontal and vertical coordinates.

Kalman filters have a wide range of applications, including control, signal and image processing; radar and sonar; and financial modeling. They are recursive filters that

estimate the state of a linear dynamic system from a series of incomplete or noisy measurements. The Kalman filter algorithm relies on the state-space representation of filters and uses a set of variables stored in the state vector to characterize completely the behavior of the system. It updates the state vector linearly and recursively using a state transition matrix and a process noise estimate.

Algorithm

This section describes the algorithm of the Kalman filter and is implemented in the MATLAB version of the filter supplied with this tutorial.

The algorithm predicts the position of a moving object based on its past positions using a Kalman filter estimator. It estimates the present position by updating the Kalman state vector, which includes the position (x and y), velocity (V_x and V_y), and acceleration (A_x and A_y) of the moving object. The Kalman state vector, `x_est`, is a persistent variable.

```
% Initial conditions
persistent x_est p_est
if isempty(x_est)
    x_est = zeros(6, 1);
    p_est = zeros(6, 6);
end
```

`x_est` is initialized to an empty 6x1 column vector and updated each time the filter is used.

The Kalman filter uses the laws of motion to estimate the new state:

$$X = X_0 + V_x dt$$

$$Y = Y_0 + V_y dt$$

$$V_x = V_{x_0} + A_x dt$$

$$V_y = V_{y_0} + A_y dt$$

These laws of motion are captured in the state transition matrix **A**, which is a matrix that contains the coefficient values of x , y , V_x , V_y , A_x , and A_y .

```
% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];
```


Filtering Process

The filtering process has two phases:

- Predicted state and covariance

The Kalman filter uses the previously estimated state, x_{est} , to predict the current state, x_{prd} . The predicted state and covariance are calculated in:

```
% Predicted state and covariance
x_prd = A * x_est;
p_prd = A * p_est * A' + Q;
```

- Estimation

The filter also uses the current measurement, z , and the predicted state, x_{prd} , to estimate a closer approximation of the current state. The estimated state and covariance are calculated in:

```
% Measurement matrix
H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
Q = eye(6);
R = 1000 * eye(2);

% Estimation
S = H * p_prd' * H' + R;
B = H * p_prd';
klm_gain = (S \ B)';

% Estimated state and covariance
x_est = x_prd + klm_gain * (z - H * x_prd);
p_est = p_prd - klm_gain * H * p_prd;

% Compute the estimated measurements
y = H * x_est;
```

Reference

Haykin, Simon. *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

Files for the Tutorial

- “About the Tutorial Files” on page 2-6

- “Location of Files” on page 2-6
- “Names and Descriptions of Files” on page 2-6

About the Tutorial Files

The tutorial uses the following files:

- Example MATLAB code files for each step of the tutorial.

Throughout this tutorial, you work with MATLAB files that contain a simple Kalman filter algorithm.

- Build scripts that you use to compile your function code.
- Test files that:
 - Perform the preprocessing functions.
 - Call the Kalman filter.
 - Perform the post-processing functions.
- A MAT-file that contains input data.

Location of Files

The tutorial files are available in the following folder: `docroot\toolbox\coder\examples\kalman`. To run the tutorial, you must copy these files to a local folder. For instructions, see “Copying Files Locally” on page 2-38.

Names and Descriptions of Files

Type	Name	Description
Function code	<code>kalman01.m</code>	Baseline MATLAB implementation of a scalar Kalman filter.
	<code>kalman02.m</code>	Version of the original algorithm that is suitable for code generation.
	<code>kalman03.m</code>	Kalman filter suitable for use with frame-based and packet-based inputs.
Test scripts	<code>test01_ui.m</code>	Tests the scalar Kalman filter and plots the trajectory.
	<code>test02_ui.m</code>	Tests the frame-based Kalman filter.

Type	Name	Description
	test03_ui.m	Tests the variable-size (packet-based) Kalman filter.
MAT-file	position.mat	Contains the input data used by the algorithm.
Plot function	plot_trajectory.m	Plots the trajectory of the object and the Kalman filter estimated position.

Design Considerations When Writing MATLAB Code for Code Generation

When writing MATLAB code that you want to convert into efficient, standalone C/C++ code, you must consider the following:

- Data types

C and C++ use static typing. To determine the types of your variables before use, MATLAB Coder requires a complete assignment to each variable.

- Array sizing

Variable-size arrays and matrices are supported for code generation. You can define inputs, outputs, and local variables in MATLAB functions to represent data that varies in size at run time.

- Memory

You can choose whether the generated code uses static or dynamic memory allocation.

With dynamic memory allocation, you potentially use less memory at the expense of time to manage the memory. With static memory, you get the best speed, but with higher memory usage. Most MATLAB code takes advantage of the dynamic sizing features in MATLAB, therefore dynamic memory allocation typically enables you to generate code from existing MATLAB code without modifying it much. Dynamic memory allocation also allows some programs to compile even when upper bounds cannot be found.

Static allocation reduces the memory footprint of the generated code, and therefore is suitable for applications where there is a limited amount of available memory, such as embedded applications.

- Speed

Because embedded applications must run in real time, the code must be fast enough to meet the required clock rate.

To improve the speed of the generated code:

- Choose a suitable C/C++ compiler. The default compiler that MathWorks supplies with MATLAB for Windows® 32-bit platforms is not a good compiler for performance.
- Consider disabling run-time checks.

By default, for safety, the code generated for your MATLAB code contains memory integrity checks and responsiveness checks. Generally, these checks result in more generated code and slower simulation. Disabling run-time checks usually results in streamlined generated code and faster simulation. Disable these checks only if you have verified that array bounds and dimension checking is unnecessary.

See Also

- “Data Definition Basics”
- “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data”
- “Control Dynamic Memory Allocation”
- “Control Run-Time Checks”

Tutorial Steps

- “Copying Files Locally” on page 2-9
- “Running the Original MATLAB Code” on page 2-9
- “Setting Up Your C Compiler” on page 2-11
- “Considerations for Making Your Code Suitable for Code Generation” on page 2-12
- “Making the MATLAB Code Suitable for Code Generation” on page 2-13
- “Setting Up a MATLAB Coder Project” on page 2-15
- “Generating a MEX Function Using MATLAB Coder” on page 2-17
- “Verifying the MEX Function Behavior” on page 2-18
- “Generating C Code Using MATLAB Coder” on page 2-19

- “Comparing the Generated C Code to Original MATLAB Code” on page 2-20
- “Modifying the Filter to Accept a Fixed-Size Input” on page 2-21
- “Using the Filter to Accept a Variable-Size Input” on page 2-26

Copying Files Locally

Copy the tutorial files to a local working folder:

1 Create a local *solutions* folder, for example, `c:\coder\kalman\solutions`.

2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```

3 Copy the contents of the `kalman` subfolder to your local *solutions* folder, specifying the full path name of the *solutions* folder:

```
copyfile('kalman', 'solutions')
```

Your *solutions* folder now contains a complete set of solutions for the tutorial. If you do not want to perform the steps for each task in the tutorial, you can view the solutions to see how the code should look.

4 Create a local *work* folder, for example, `c:\coder\kalman\work`.

5 Copy the following files from your *solutions* folder to your *work* folder.

- `kalman01.m`
- `position.mat`
- Test scripts `test01_ui.m` through `test03_ui.m`
- `plot_trajectory.m`

Your *work* folder now contains the files that you need to get started with the tutorial.

Running the Original MATLAB Code

In this tutorial, you work with a MATLAB function that implements a Kalman filter algorithm, which predicts the position of a moving object based on its past positions. Before generating C code for this algorithm, you make the MATLAB version suitable for code generation and generate a MEX function. Then you test the resulting MEX function to validate the functionality of the modified code. As you work through the tutorial, you refine the design of the algorithm to accept variable-size inputs.

First, use the script `test01_ui.m` to run the original MATLAB function to see how the Kalman filter algorithm works. This script loads the input data and calls the Kalman filter algorithm to estimate the location. It then calls a plot function, `plot_trajectory`, which plots the trajectory of the object and the Kalman filter estimated position.

- 1 Set your MATLAB current folder to the work folder that contains your files for this tutorial. At the MATLAB command line, enter:

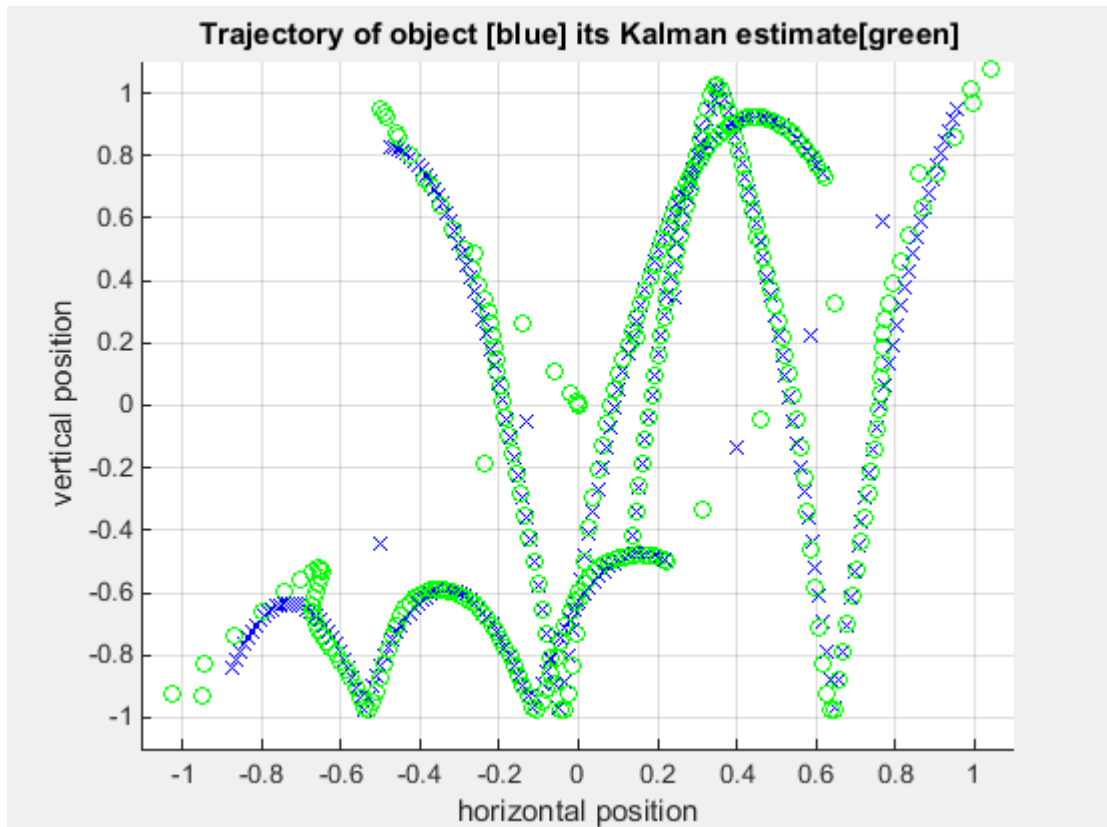
```
cd work
```

where *work* is the full path name of the work folder containing your files. For more information, see “Files and Folders that MATLAB Accesses”.

- 2 At the MATLAB command line, enter:

```
test01_ui
```

The test script runs and plots the trajectory of the object in blue and the Kalman filter estimated position in green. Initially, you see that it takes a short time for the estimated position to converge with the actual position of the object. Then three sudden shifts in position occur—each time the Kalman filter readjusts and tracks the object after a few iterations.



Setting Up Your C Compiler

MATLAB Coder automatically locates and uses a supported installed compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks Web site.

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

Considerations for Making Your Code Suitable for Code Generation

Designing for Code Generation

Before generating code, you must prepare your MATLAB code for code generation. The first step is to eliminate unsupported constructs.

Checking for Violations at Design Time

There are two tools that help you detect code generation issues at design time: the code analyzer and the code generation readiness tool.

You use the code analyzer in the MATLAB Editor to check for code violations at design time, minimizing compilation errors. The code analyzer continuously checks your code as you enter it. It reports problems and recommends modifications to maximize performance and maintainability.

To use the code analyzer to identify warnings and errors specific to MATLAB for code generation, you must add the `%#codegen` directive (or pragma) to your MATLAB file. A complete list of MATLAB for Code Generation code analyzer messages is available in the MATLAB Code Analyzer preferences. See “Running the Code Analyzer Report” for more details.

Note: The code analyzer might not detect all MATLAB for code generation issues. After eliminating the errors or warnings that the code analyzer detects, compile your code with MATLAB Coder to determine if the code has other compliance issues.

The code generation readiness tool screens MATLAB code for features and functions that are not supported for code generation. The tool provides a report that lists the source files that contain unsupported features and functions and an indication of how much work is required to make the MATLAB code suitable for code generation.

You can access the code generation readiness tool in the following ways:

- In the current folder browser — by right-clicking a MATLAB file
- At the command line — by using the `coder.screener` function.
- In a project — when you add a MATLAB file to a project, if MATLAB Coder detects code generation issues, it provides a link to the code generation readiness report. For more information, see “Code Generation Readiness Tool”.

Checking for Violations at Code Generation Time

You can use MATLAB Coder to check for violations at code generation time. MATLAB Coder checks that your MATLAB code is suitable for code generation.

When MATLAB Coder detects errors or warnings, it automatically generates an error report that describes the issues and provides links to the offending MATLAB code. For more information, see “Code Generation Reports”.

After code generation, MATLAB Coder generates a MEX function that you can use to test your implementation in MATLAB.

Checking for Violations at Run Time

You can use MATLAB Coder to generate a MEX function and check for violations at run time. The MEX function generated for your MATLAB functions includes run-time checks. Disabling run-time checks and extrinsic calls usually results in streamlined generated code and faster simulation. Disabling run-time checks allows bugs in your code to crash MATLAB. For more information, see “Control Run-Time Checks”.

If you encounter run-time errors in your MATLAB functions, a run-time stack appears automatically in the MATLAB Command Window. See “Debug Run-Time Errors”.

Making the MATLAB Code Suitable for Code Generation

- “Making Your Code Suitable for Code Generation” on page 2-13
- “Where to Go Next” on page 2-15

Making Your Code Suitable for Code Generation

To modify the code yourself, work through the exercises in this section. Otherwise, open the supplied file `kalman02.m` in your *solutions* subfolder to see the modified algorithm.

To begin the process of making your MATLAB code suitable for code generation, you work with the file `kalman01.m`. This code is a MATLAB version of a scalar Kalman filter that estimates the state of a dynamic system from a series of noisy measurements.

- 1 Set your MATLAB current folder to the work folder that contains your files for this tutorial. At the MATLAB command line, enter:

```
cd work
```

where *work* is the full path name of the work folder containing your files. See “Files and Folders that MATLAB Accesses”.

- 2 Open `kalman01.m` in the MATLAB Editor. At the MATLAB command line, enter:

```
edit kalman01.m
```

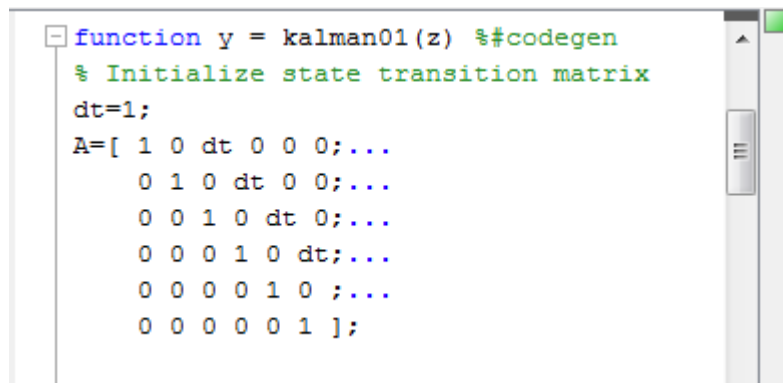
Tip Before modifying your code, it is best practice to preserve the current version by backing it up.

The file opens in the MATLAB Editor. The code analyzer message indicator in the top right corner of the MATLAB Editor is green, which indicates that it has not detected errors, warnings, or opportunities for improvement in the code.

- 3 Turn on MATLAB for code generation error checking by adding the `%#codegen` directive after the function declaration.

```
function y = kalman01(z) %#codegen
```

The code analyzer message indicator remains green, indicating that it has not detected code generation related issues.



```
function y = kalman01(z) %#codegen
% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];
```

For more information on using the code analyzer, see “Running the Code Analyzer Report”.

- 4 Save the file.

You are now ready to compile your code using MATLAB Coder. By default, MATLAB Coder checks that your MATLAB code is suitable for code generation. Then, after compilation, it generates a MEX function that you can test in MATLAB.

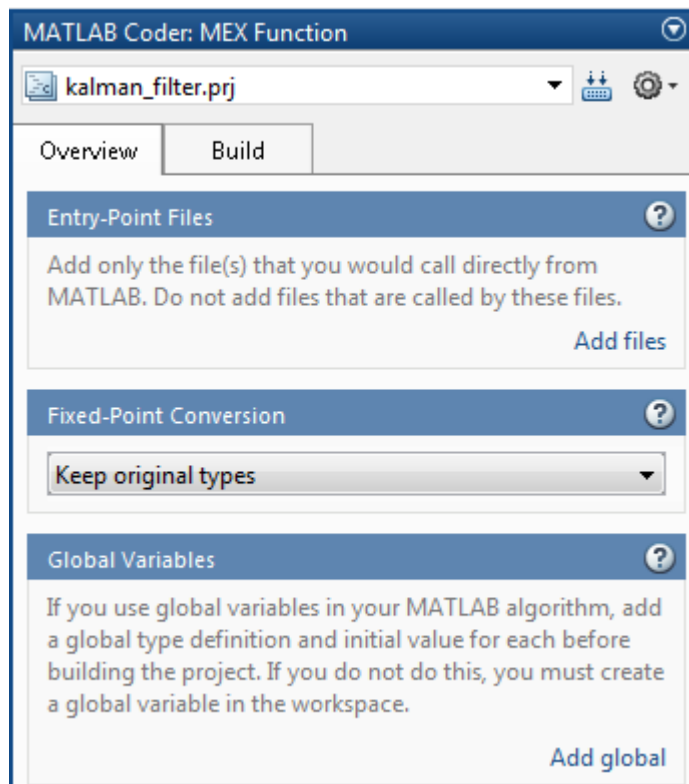
Where to Go Next

The next part of the tutorial, “Setting Up a MATLAB Coder Project” on page 2-15, shows you how to set up a MATLAB Coder project.

Setting Up a MATLAB Coder Project

- 1 In MATLAB, select the **Apps** tab and then click **MATLAB Coder**.
- 2 The MATLAB Coder Project dialog box opens.
- 3 In this dialog box, enter a name for the project, for example, `kalman_filter`, and click **OK**.


MATLAB Coder creates the project, `kalman_filter.prj`, in the current folder and, by default, opens the project in the right side of the MATLAB workspace.



You are now ready to add the `kalman01.m` file to your project.

- 4 On the MATLAB Coder project **Overview** tab, click **Add files**.
- 5 In the **Add Files** dialog box, select `kalman01.m` and click **Open**.

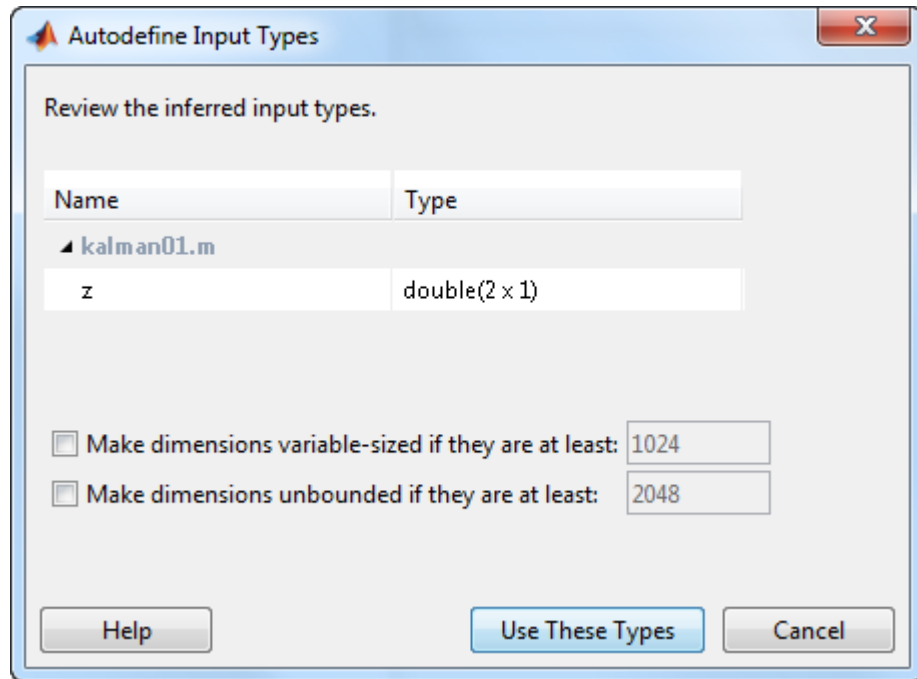
MATLAB Coder adds the file to the project. `kalman01` function has one input parameter, `z`, which appears below the file name. Note that its type is undefined.

- 6 Because C uses static typing, MATLAB Coder must determine the properties of all variables in the MATLAB files at compile time. Therefore, you must specify the properties of all function inputs at the same time as you compile the file. To compile `kalman01.m`, you must specify the size of the input vector `z`. Here, you specify a test file so that MATLAB Coder can autodefine types for `z` :
 - a Click the **Autodefine types** link.
 - b In the **Autodefine Input Types** dialog box, click the  button to add a test file to the project.
 - c Select `test01_ui.m` and click **Open**.

The test file, `test01_ui.m`, calls the entry-point function, `kalman01.m`, with the expected input types. For more information on using test files to autodefine input types, see “Autodefine Input Types”.

- d In the **Autodefine Input Types** dialog box, click the **Run** button.

The test file runs, plots the output, and infers that input `z` is `double(2x1)`.



- e Click **Use These Types**.

MATLAB Coder assigns this type to `z`.

You are now ready to build the project to generate a MEX function for `kalman01.m`.

Generating a MEX Function Using MATLAB Coder

- 1 Click the MATLAB Coder project **Build** tab.

By default, the project will generate a MEX function named `kalman01_mex`.

Note: For the purpose of this tutorial, use the default build settings. To fine tune the MEX code generation, you can click **More settings** to open the **Project Settings** dialog box and configure build settings.

- 2 Click the **Build** button.

The Build progress dialog box appears. When the build is complete, MATLAB Coder generates a MEX function `kalman01_mex` in the current folder.

Note: The file extension is platform dependent.

You have proved that the Kalman filter example code is suitable for code generation using MATLAB Coder. You are ready to begin the next task in this tutorial, “Verifying the MEX Function” on page 2-45.

Verifying the MEX Function Behavior

In this part of the tutorial, you test the MEX function to verify that it provides the same functionality as the original MATLAB code.

In addition, simulating your algorithm in MATLAB before generating C code enables you to detect and fix run-time errors that would be much harder to diagnose in the generated C code. By default, the following run-time checks execute when you simulate your MEX function in MATLAB:

- Memory integrity checks. These checks perform array bounds and dimension checking and detect violations of memory integrity in code generated for MATLAB functions. If a violation is detected, MATLAB stops execution with a diagnostic message.
- Responsiveness checks in code generated for MATLAB functions. These checks enable periodic checks for **Ctrl+C** breaks in code generated for MATLAB functions, allowing you to terminate execution with **Ctrl+C**.

For more information, see “Control Run-Time Checks”.

Running the Generated MEX Function

You run the MEX function, `kalman01_mex`, using the same test file that you used in “Running the Original MATLAB Code” on page 2-39. The MATLAB Coder software automatically replaces calls to the MATLAB algorithm with calls to the MEX function.

- 1 On the **Build** tab **Verification** pane, verify that the **Test file** is set to `test01_ui.m`.
- 2 Select **Redirect entry-point calls to MEX function**.
- 3 Click **Run**.

MATLAB Coder runs the test file and automatically replaces calls to the MATLAB algorithm with calls to the MEX function. The test runs and plots the trajectory of the object and the Kalman filter estimated position as before.

You have generated a MEX function for your MATLAB code, verified that it is functionally equivalent to your original MATLAB code, and checked for run-time errors. Now you are ready to begin the next task in this tutorial, “Generating C Code Using codegen” on page 2-46.

Generating C Code Using MATLAB Coder

In this task, you use MATLAB Coder to generate C code for your MATLAB filter algorithm. You then view the generated C code using the code generation report and compare the generated C code with the original MATLAB code.

How to Generate C Code

- 1 On the **Build** tab, from the **Output type** drop-down list, select **C/C++ Static Library**.

MATLAB Coder is now ready to generate a static library for `kalman01`. The default name for the library is `kalman01`.

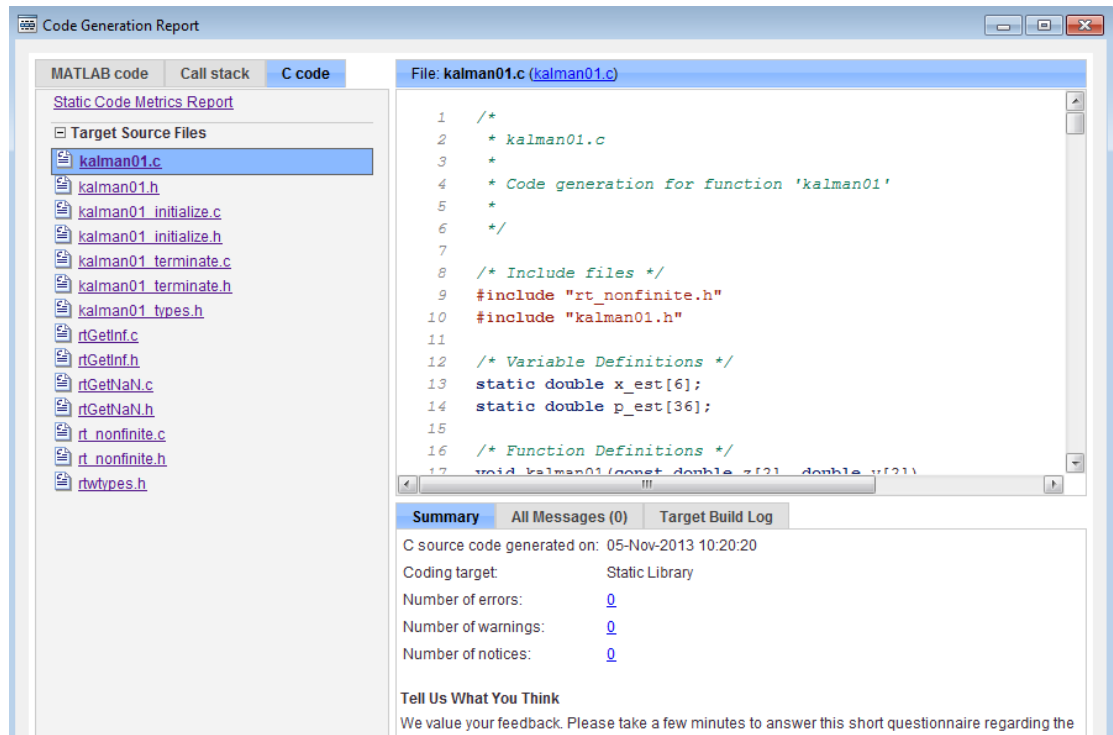
Note: For the purpose of this tutorial, use the default build settings. Different project settings are available for MEX and C/C++ output types. When you switch between MEX and C/C++ code generation, you should verify these settings.

- 2 On the **Build** tab, click the **Build** button.

The Build progress dialog box is displayed. MATLAB Coder generates a standalone C static library `kalman01` in the `work\codegen\lib\kalman01`, where `work` is the folder that contains your tutorial files, and provides a link to the code generation report.

- 3 To view the code generation report, click *View report*.

The Code Generation Report opens and, in the right pane, displays the generated C code, `kalman01.c`. It also provides a hyperlink to open the C code in the MATLAB Editor.



To learn more about the report, see “Code Generation Reports”.

Comparing the Generated C Code to Original MATLAB Code

To compare your generated C code to the original MATLAB code, open the C file, `kalman01.c`, and the `kalman01.m` file in the MATLAB Editor. View the files side by side by selecting the **VIEW** tab and then clicking **Left/Right**.

Here are some important points about the generated C code:

- The function signature is:

```
void kalman01(const double z[2], double y[2])
```

`z` corresponds to the input `z` in your MATLAB code. The size of `z` is 2, which corresponds to the total size (2 x 1) of the example input you used when you compiled your MATLAB code.

- You can easily compare the generated C code to your original MATLAB code. In the generated C code:
 - Your function name is unchanged.
 - Your comments are preserved in the same position.
 - Your variable names are the same as in the original MATLAB code.

Note: If a variable in your MATLAB code is set to a constant value, it does not appear as a variable in the generated C code. Instead, the generated C code contains the actual value of the variable.

Modifying the Filter to Accept a Fixed-Size Input

The filter you have worked on so far in this tutorial uses a simple batch process that accepts one input at a time, so you must call the function repeatedly for each input. In this part of the tutorial, you learn how to modify the algorithm to accept a fixed-sized input, which makes the algorithm suitable for frame-based processing.

Modifying Your MATLAB Code

To modify the code yourself, work through the exercises in this section. Otherwise, open the supplied file `kalman03.m` in your *solutions* subfolder to see the modified algorithm.

The filter algorithm you have used so far in this tutorial accepts only one input. You can now modify the algorithm to process a vector containing more than one input. You need to find the length of the vector and call the filter code for each element in the vector in turn. You do this by calling the filter algorithm in a `for`-loop.

- 1 Open `kalman01.m` in the MATLAB Editor.

```
edit kalman01.m
```

Tip Before modifying your code, it is best practice to preserve the current version by backing it up.

- 2 Add a `for`-loop around the filter code.
 - a Before the comment

```
% Predicted state and covariance
```

```
insert:
```

```
for i=1:size(z,2)
```

b After

```
% Compute the estimated measurements
```

```
y = H * x_est;
```

```
insert:
```

```
end
```

Your filter code should now look like this:

```
for i=1:size(z,2)
```

```
% Predicted state and covariance
```

```
x_prd = A * x_est;
```

```
p_prd = A * p_est * A' + Q;
```

```
% Estimation
```

```
S = H * p_prd' * H' + R;
```

```
B = H * p_prd';
```

```
klm_gain = (S \ B)';
```

```
% Estimated state and covariance
```

```
x_est = x_prd + klm_gain * (z - H * x_prd);
```

```
p_est = p_prd - klm_gain * H * p_prd;
```

```
% Compute the estimated measurements
```

```
y = H * x_est;
```

```
end
```

- 3** Modify the line that calculates the estimated state and covariance to use the i^{th} element of input Z .

Change

```
x_est = x_prd + klm_gain * (z - H * x_prd);
```

to

```
x_est = x_prd + klm_gain * (z(:,i) - H * x_prd);
```

- 4** Modify the line that computes the estimated measurements to append the result to the i^{th} element of the output y .

Change

```

y = H * x_est;
to

y(:,i) = H * x_est;

```

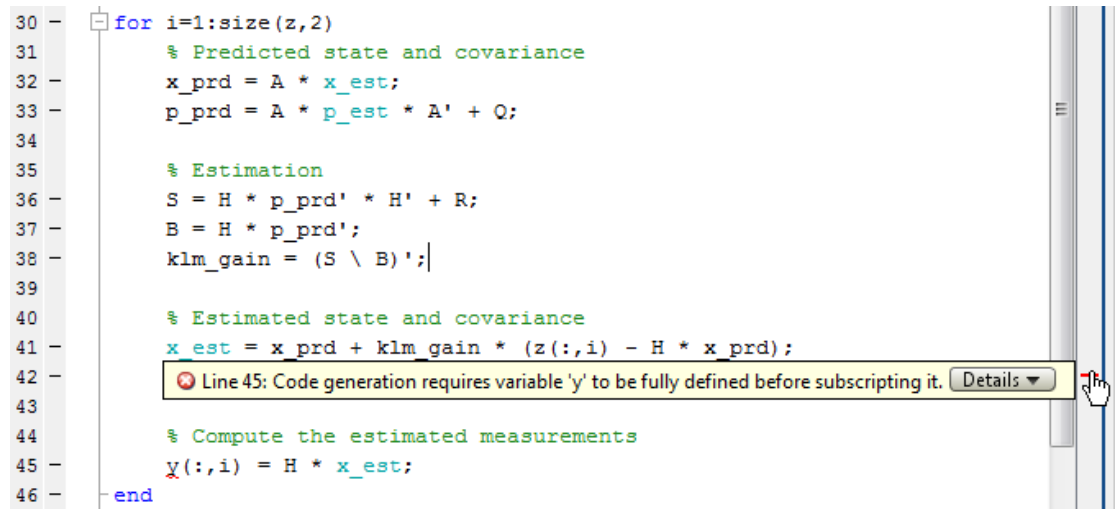
The code analyzer message indicator in the top right turns red to indicate that the code analyzer has detected an error. The code analyzer underlines the offending code in red and places a red marker to the right.

- 5 Move your pointer over the red marker to view the error.

The code analyzer reports that code generation requires variable `y` to be fully defined before subscripting it.

Why Preallocate the Outputs?

You must preallocate outputs here because code generation does not support increasing the size of an array through indexing. Repeatedly expanding the size of an array over time can adversely affect the performance of your program. See “Preallocating Memory”.



```

30 - for i=1:size(z,2)
31 -     % Predicted state and covariance
32 -     x_prd = A * x_est;
33 -     p_prd = A * p_est * A' + Q;
34 -
35 -     % Estimation
36 -     S = H * p_prd' * H' + R;
37 -     B = H * p_prd';
38 -     klm_gain = (S \ B)';
39 -
40 -     % Estimated state and covariance
41 -     x_est = x_prd + klm_gain * (z(:,i) - H * x_prd);
42 -     Line 45: Code generation requires variable 'y' to be fully defined before subscripting it.
43 -
44 -     % Compute the estimated measurements
45 -     y(:,i) = H * x_est;
46 - end

```

- 6 To address the error, preallocate memory for the output `y`, which is the same size as the input `z`. Add this code before the `for`-loop.

```
% Pre-allocate output signal:  
y=zeros(size(z));
```

The red error marker disappears and the code analyzer message indicator in the top right edge of the code turns green, which indicates that you have fixed the errors and warnings detected by the code analyzer.


For more information on using the code analyzer, see “Running the Code Analyzer Report”.

7 Save the file.

You are ready to begin the next task in the tutorial, “Generating C Code for Your Modified Algorithm” on page 2-24.

Generating C Code for Your Modified Algorithm

You modified the algorithm to expect fixed-size input, so you must first define the input type for the updated `kalman01` function. Use the test file `test02_ui.m` to autodefine input types for the updated `kalman01.m`. This script sets the frame size to 10 and calculates the number of frames in the example input. It then calls the Kalman filter and plots the results for each frame in turn.

- 1 On the MATLAB Coder project **Overview** tab, click the **Autodefine types** link.
- 2 In the **Autodefine Input Types** dialog box, click the  button to add a test file to the project.
- 3 Select `test02_ui.m` and click **Open**.

The test file `test02_ui.m` sets the frame size to 10 and calculates the number of frames in the example input. It then calls the Kalman filter and plots the results for each frame in turn.

Contents of `test02_ui.m`

```
% Figure setup  
clear all;  
% Load position data  
load position.mat  
% Set up the frame size  
numPts = 300;  
frame=10;
```

```

numFrms=300/frame;

figure;hold;grid;
% Kalman filter loop
for i = 1: numFrms
    % Generate the location data
    z = position(:,frame*(i-1)+1:frame*i);

    % Use Kalman filter to estimate the location
    y = kalman01(z);

    % Plot the results
    for n=1:frame
        plot_trajectory(z(:,n),y(:,n));
    end
end
hold;

```

- 4 In the **Autodefine Input Types** dialog box, click the **Run** button.

The test file runs and plots the trajectory of the object and the Kalman filter estimated position as before. MATLAB Coder infers that the input type of `z` is `double(2x10)`.

- 5 Click **Use These Types**.

MATLAB Coder assigns this type to `z`.

You are now ready to build the project to generate and test a MEX function for `kalman01.m`.

To generate and test a MEX function:

- 1 On the **Build** tab, set **Output type** to **MEX Function**.
- 2 On the **Build** tab **Verification** pane, set **Test file** to `test02_ui.m`.
- 3 Select **Redirect entry-point calls to MEX function**.
- 4 Click **Run**.

MATLAB Coder generates a MEX function. It then runs the test file and automatically replaces calls to the MATLAB algorithm with calls to the MEX function. The test runs and plots the trajectory of the object and the Kalman filter estimated position as before.

To generate C code:

- 1 On the **Build** tab, set **Output type** to **C/C++ Static Library** and select **Generate code only**.

This option instructs MATLAB Coder to generate code only without invoking the `make` command. If this option is used, MATLAB Coder does not generate compiled object code. This option saves you time during the development cycle when you want to iterate rapidly between modifying MATLAB code and generating C code and are mainly interested in inspecting the C code.

- 2 Click the **Build** button.

The Build progress dialog box appears. MATLAB Coder generates C code in the `work\codegen\lib\kalman01`, where `work` is the folder that contains your tutorial files subfolder and provides a link to the code generation report.

- 3 To view the code generation report, click *View report*

The Code Generation Report opens and displays the generated code, `kalman01.c`.

- 4 Compare the generated C code with the C code for the scalar Kalman filter. You see that the code is almost identical except that there is now a `for`-loop for the frame processing.

Here are some important points about the generated C code:

- The function signature is now:

```
void kalman01(const double z[20], double y[20])
```

The size of `z` and `y` is now 20, which corresponds to the size of the example input `z` (2x10) used to compile your MATLAB code.

- The filtering now takes place in a `for`-loop. The `for`-loop iterates over all 10 inputs.

```
for(i = 0; i < 10; i++)  
{  
    /* Predicted state and covariance */ ...
```

Using the Filter to Accept a Variable-Size Input

The algorithm you have used so far in this tutorial is suitable for processing input data that consists of fixed-size frames. In this part of the tutorial, you test your algorithm with variable-size inputs and see that the algorithm is suitable for processing packets of data of varying size. You then learn how to generate code for a variable-size input.

Testing the Algorithm with Variable-Size Inputs

Use the test script `test03_ui.m` to test the filter with variable-size inputs.

The test script calls the filter algorithm in a loop, passing a different size input to the filter each time. Each time through the loop, the test script calls the `plot_trajectory` function for every position in the input.

To run the test script, at the MATLAB command line, enter:


```
test03_ui
```

The test script runs and plots the trajectory of the object and the Kalman filter estimated position as before.

You have created an algorithm that accepts variable-size inputs. You are ready to begin the next task in the tutorial, “Generating C Code for a Variable-Size Input” on page 2-54.

Note: Before generating C code, it is best practice to generate a MEX function that you can execute within the MATLAB environment to test your algorithm and check for run-time errors.

How to Generate C Code for a Variable-Size Input

- 1 On the MATLAB Coder project **Overview** tab, click the **Autodefine types** link.
- 2 In the **Autodefine Input Types** dialog box, click the  button to add a test file to the project.
- 3 Select `test03_ui.m` and click **Open**.
- 4 In the **Autodefine Input Types** dialog box, click **Run**.

The test file runs and plots the trajectory of the object and the Kalman filter estimated position as before. MATLAB Coder infers that the input type of `z` is `double(2x:100)`. The `:` in front of the second dimension indicates that this dimension is variable size.

Because the test file calls `kalman01` multiple times with different sized inputs, MATLAB Coder takes the union of the inputs and infers that the inputs are variable size, with an upper bound equal to the size of the largest input.

5 Click **Use These Types**.

MATLAB Coder assigns this type to `z`.

You are now ready to build the project to generate and test a MEX function for `kalman01.m`.

To generate and test a MEX function:

- 1 On the **Build** tab, set **Output type** to **MEX Function**.
- 2 On the **Build** tab **Verification** pane, set **Test file** to `test03_ui.m`.
- 3 Select **Redirect entry-point calls to MEX function**.
- 4 Click **Run**.

MATLAB Coder generates a MEX function. It then runs the test file and automatically replaces calls to the MATLAB algorithm with calls to the MEX function. The test runs and plots the trajectory of the object and the Kalman filter estimated position as before.

To generate C code:

- 1 On the **Build** tab, set **Output type** to **C/C++ Static Library** and select **Generate code only**.

This option instructs MATLAB Coder to generate code only without invoking the `make` command. If this option is used, MATLAB Coder does not generate compiled object code. This option saves you time during the development cycle when you want to iterate rapidly between modifying MATLAB code and generating C code and are mainly interested in inspecting the C code.

- 2 On the **Build** tab, click the **Build** button.

The Build progress dialog box appears. MATLAB Coder generates a standalone C library `kalman01` in the `work\codegen\lib\kalman01`, where `work` is the folder that contains your tutorial files. subfolder and provides a link to the code generation report.

- 3 View the generated C code as before.

Here are some important points about the generated C code:

- The generated C code can process inputs from 2×1 to 2×100 . The function signature is now:


```
void kalman01(double z_data[200], ...
             int z_size[2], ...
             double y_data[200], int y_size[2])
```

Because `y` and `z` are variable size, the generated code contains two pieces of information about each of them: the data and the actual size of the sample. For example, for variable `z`, the generated code contains:

- The data `z_data[200]`, where `200` is the maximum size of input `z`.
- `z_size[2]`, which contains the actual size of the input data. This information varies each time the filter is called.
- To maximize efficiency, the actual size of the input data `z_size` is used when calculating the estimated position. The filter processes only the number of samples available in the input.

```
for (i = 0; i <= z_size[1]; i++) {
    /* Predicted state and covariance */
    for(k = 0; k < 6; k++) {
        ...
    }
}
```

Key Points to Remember

- Back up your MATLAB code before you modify it.
- Use test scripts to separate the pre- and post-processing from the core algorithm.
- Generate a MEX function before generating C code. Use this MEX function to simulate your algorithm in MATLAB to validate its operation and check for run-time errors.
- Use the `Autodefine types` option to specify input parameters if you have a test file that calls the entry-point function with the required class, size, and complexity.
- Create a code generation report.

Learn More

- “Next Steps” on page 2-30
- “Product Help” on page 2-30
- “MathWorks Online” on page 2-30

Next Steps

To...	See...
Learn how to integrate your MATLAB code with Simulink models	“Track Object Using MATLAB Code”
Learn more about using MATLAB for code generation	“MATLAB Algorithm Design”
Use variable-size data	“Variable-Size Data Definition for Code Generation”
Speed up fixed-point MATLAB code	<code>fiaccel</code>
Integrate custom C code into MATLAB code and generate embeddable code	“Specify External File Locations”
Integrate custom C code into a MATLAB function for code generation	<code>coder.ceval</code>
Generate HDL from MATLAB code	www.mathworks.com/products/slhdlcoder

Product Help

MathWorks product documentation is available from the Help menu on the MATLAB desktop.

MathWorks Online

For additional information and support, visit the MATLAB Coder page on the MathWorks Web site at:

www.mathworks.com/products/matlab-coder

C Code Generation at the Command Line

In this section...

“Learning Objectives” on page 2-31

“Tutorial Prerequisites” on page 2-31

“Example: The Kalman Filter” on page 2-32

“Files for the Tutorial” on page 2-34

“Design Considerations When Writing MATLAB Code for Code Generation” on page 2-36

“Tutorial Steps” on page 2-37

“Key Points to Remember” on page 2-56

“Best Practices Used in This Tutorial” on page 2-56

“Where to Learn More” on page 2-57

Learning Objectives

In this tutorial, you will learn how to:

- Automatically generate a MEX function from your MATLAB code and use this MEX function to validate your algorithm in MATLAB before generating C code.
- Automatically generate C code from your MATLAB code.
- Define function input properties at the command line.
- Specify variable-size inputs when generating code.
- Specify code generation properties.
- Generate a code generation report that you can use to debug your MATLAB code and verify that it is suitable for code generation.

Tutorial Prerequisites

- “What You Need to Know” on page 2-31
- “Required Products” on page 2-32

What You Need to Know

To complete this tutorial, you should have basic familiarity with MATLAB software.

Required Products

To complete this tutorial, you must install the following products:

- MATLAB
- MATLAB Coder
- C compiler

For most platforms, a default compiler is supplied with MATLAB.

MATLAB Coder automatically locates and uses a supported installed compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks Web site.

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

For instructions on installing MathWorks products, see the MATLAB installation documentation for your platform. If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window.

Example: The Kalman Filter

- “Description” on page 2-32
- “Algorithm” on page 2-33
- “Filtering Process” on page 2-34
- “Reference” on page 2-34

Description

This section describes the example used by the tutorial. You do not have to be familiar with the algorithm to complete the tutorial.

The example for this tutorial uses a Kalman filter to estimate the position of an object moving in a two-dimensional space from a series of noisy inputs based on past positions. The position vector has two components, x and y , indicating its horizontal and vertical coordinates.

Kalman filters have a wide range of applications, including control, signal and image processing; radar and sonar; and financial modeling. They are recursive filters that

estimate the state of a linear dynamic system from a series of incomplete or noisy measurements. The Kalman filter algorithm relies on the state-space representation of filters and uses a set of variables stored in the state vector to characterize completely the behavior of the system. It updates the state vector linearly and recursively using a state transition matrix and a process noise estimate.

Algorithm

This section describes the algorithm of the Kalman filter and is implemented in the MATLAB version of the filter supplied with this tutorial.

The algorithm predicts the position of a moving object based on its past positions using a Kalman filter estimator. It estimates the present position by updating the Kalman state vector, which includes the position (x and y), velocity (Vx and Vy), and acceleration (Ax and Ay) of the moving object. The Kalman state vector, `x_est`, is a persistent variable.

```
% Initial conditions
persistent x_est p_est
if isempty(x_est)
    x_est = zeros(6, 1);
    p_est = zeros(6, 6);
end
```

`x_est` is initialized to an empty 6x1 column vector and updated each time the filter is used.

The Kalman filter uses the laws of motion to estimate the new state:

$$X = X_0 + Vx.dt$$

$$Y = Y_0 + Vy.dt$$

$$Vx = Vx_0 + Ax.dt$$

$$Vy = Vy_0 + Ay.dt$$

These laws of motion are captured in the state transition matrix **A**, which is a matrix that contains the coefficient values of x , y , V_x , V_y , A_x , and A_y .

```
% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];
```

Filtering Process

The filtering process has two phases:

- Predicted state and covariance

The Kalman filter uses the previously estimated state, x_{est} , to predict the current state, x_{prd} . The predicted state and covariance are calculated in:

```
% Predicted state and covariance
x_prd = A * x_est;
p_prd = A * p_est * A' + Q;
```

- Estimation

The filter also uses the current measurement, z , and the predicted state, x_{prd} , to estimate a closer approximation of the current state. The estimated state and covariance are calculated in:

```
% Measurement matrix
H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
Q = eye(6);
R = 1000 * eye(2);

% Estimation
S = H * p_prd' * H' + R;
B = H * p_prd';
klm_gain = (S \ B)';

% Estimated state and covariance
x_est = x_prd + klm_gain * (z - H * x_prd);
p_est = p_prd - klm_gain * H * p_prd;

% Compute the estimated measurements
y = H * x_est;
```

Reference

Haykin, Simon. *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.

Files for the Tutorial

- “About the Tutorial Files” on page 2-35

- “Location of Files” on page 2-35
- “Names and Descriptions of Files” on page 2-35

About the Tutorial Files

The tutorial uses the following files:

- Example MATLAB code files for each step of the tutorial.

Throughout this tutorial, you work with MATLAB files that contain a simple Kalman filter algorithm.

- Build scripts that you use to compile your function code.
- Test files that:
 - Perform the preprocessing functions.
 - Call the Kalman filter.
 - Perform the post-processing functions.
- A MAT-file that contains input data.

Location of Files

The tutorial files are available in the following folder: `docroot\toolbox\coder\examples\kalman`. To run the tutorial, you must copy these files to a local folder. For instructions, see “Copying Files Locally” on page 2-38.

Names and Descriptions of Files

Type	Name	Description
Function code	<code>kalman01.m</code>	Baseline MATLAB implementation of a scalar Kalman filter.
	<code>kalman02.m</code>	Version of the original algorithm that is suitable for code generation.
	<code>kalman03.m</code>	Kalman filter suitable for use with frame-based and packet-based inputs.
Build scripts	<code>build01.m</code>	Generates MEX function for the original Kalman filter.
	<code>build02.m</code>	Generates C code for the original Kalman filter.

Type	Name	Description
	build03.m	Generates C code for the frame-based Kalman filter.
	build04.m	Generates C code for the variable-size (packet-based) Kalman filter.
Test scripts	test01.m	Tests the scalar Kalman filter and plots the trajectory.
	test02.m	Tests MEX function for the original Kalman filter and plots the trajectory.
	test03.m	Tests the frame-based Kalman filter.
	test04.m	Tests the variable-size (packet-based) Kalman filter.
MAT-file	position.mat	Contains the input data used by the algorithm.
Plot function	plot_trajectory.m	Plots the trajectory of the object and the Kalman filter estimated position.

Design Considerations When Writing MATLAB Code for Code Generation

When writing MATLAB code that you want to convert into efficient, standalone C/C++ code, you must consider the following:

- Data types

C and C++ use static typing. To determine the types of your variables before use, MATLAB Coder requires a complete assignment to each variable.

- Array sizing

Variable-size arrays and matrices are supported for code generation. You can define inputs, outputs, and local variables in MATLAB functions to represent data that varies in size at run time.

- Memory

You can choose whether the generated code uses static or dynamic memory allocation.

With dynamic memory allocation, you potentially use less memory at the expense of time to manage the memory. With static memory, you get the best speed, but with

higher memory usage. Most MATLAB code takes advantage of the dynamic sizing features in MATLAB, therefore dynamic memory allocation typically enables you to generate code from existing MATLAB code without modifying it much. Dynamic memory allocation also allows some programs to compile even when upper bounds cannot be found.

Static allocation reduces the memory footprint of the generated code, and therefore is suitable for applications where there is a limited amount of available memory, such as embedded applications.

- **Speed**

Because embedded applications must run in real time, the code must be fast enough to meet the required clock rate.

To improve the speed of the generated code:

- Choose a suitable C/C++ compiler. The default compiler that MathWorks supplies with MATLAB for Windows 32-bit platforms is not a good compiler for performance.
- Consider disabling run-time checks.

By default, for safety, the code generated for your MATLAB code contains memory integrity checks and responsiveness checks. Generally, these checks result in more generated code and slower simulation. Disabling run-time checks usually results in streamlined generated code and faster simulation. Disable these checks only if you have verified that array bounds and dimension checking is unnecessary.

See Also

- “Data Definition Basics”
- “Variable-Size Data”
- “Bounded Versus Unbounded Variable-Size Data”
- “Control Dynamic Memory Allocation”
- “Control Run-Time Checks”

Tutorial Steps

- “Copying Files Locally” on page 2-38
- “Running the Original MATLAB Code” on page 2-39

- “Setting Up Your C Compiler” on page 2-40
- “Considerations for Making Your Code Suitable for Code Generation” on page 2-41
- “Making the MATLAB Code Suitable for Code Generation” on page 2-42
- “Generating a MEX Function Using codegen” on page 2-44
- “Verifying the MEX Function” on page 2-45
- “Generating C Code Using codegen” on page 2-46
- “Comparing the Generated C Code to Original MATLAB Code” on page 2-48
- “Modifying the Filter to Accept a Fixed-Size Input” on page 2-49
- “Modifying the Filter to Accept a Variable-Size Input” on page 2-53
- “Testing the Algorithm with Variable-Size Inputs” on page 2-54
- “Generating C Code for a Variable-Size Input” on page 2-54

Copying Files Locally

Copy the tutorial files to a local working folder:

- 1 Create a local *solutions* folder, for example, `c:\coder\kalman\solutions`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```

- 3 Copy the contents of the `kalman` subfolder to your local *solutions* folder, specifying the full path name of the *solutions* folder:

```
copyfile('kalman', 'solutions')
```

Your *solutions* folder now contains a complete set of solutions for the tutorial. If you do not want to perform the steps for each task in the tutorial, you can view the solutions to see how the code should look.

- 4 Create a local *work* folder, for example, `c:\coder\kalman\work`.
- 5 Copy the following files from your *solutions* folder to your *work* folder.
 - `kalman01.m`
 - `position.mat`
 - Build files `build01.m` through `build04.m`
 - Test scripts `test01.m` through `test04.m`

- `plot_trajectory.m`

Your *work* folder now contains the files that you need to get started with the tutorial.

Running the Original MATLAB Code

In this tutorial, you work with a MATLAB function that implements a Kalman filter algorithm, which predicts the position of a moving object based on its past positions. Before generating C code for this algorithm, you make the MATLAB version suitable for code generation and generate a MEX function. Then you test the resulting MEX function to validate the functionality of the modified code. As you work through the tutorial, you refine the design of the algorithm to accept variable-size inputs.

First, use the script `test01.m` to run the original MATLAB function to see how the Kalman filter algorithm works. This script loads the input data and calls the Kalman filter algorithm to estimate the location. It then calls a plot function, `plot_trajectory`, which plots the trajectory of the object and the Kalman filter estimated position.

- 1 Set your MATLAB current folder to the work folder that contains your files for this tutorial. At the MATLAB command line, enter:

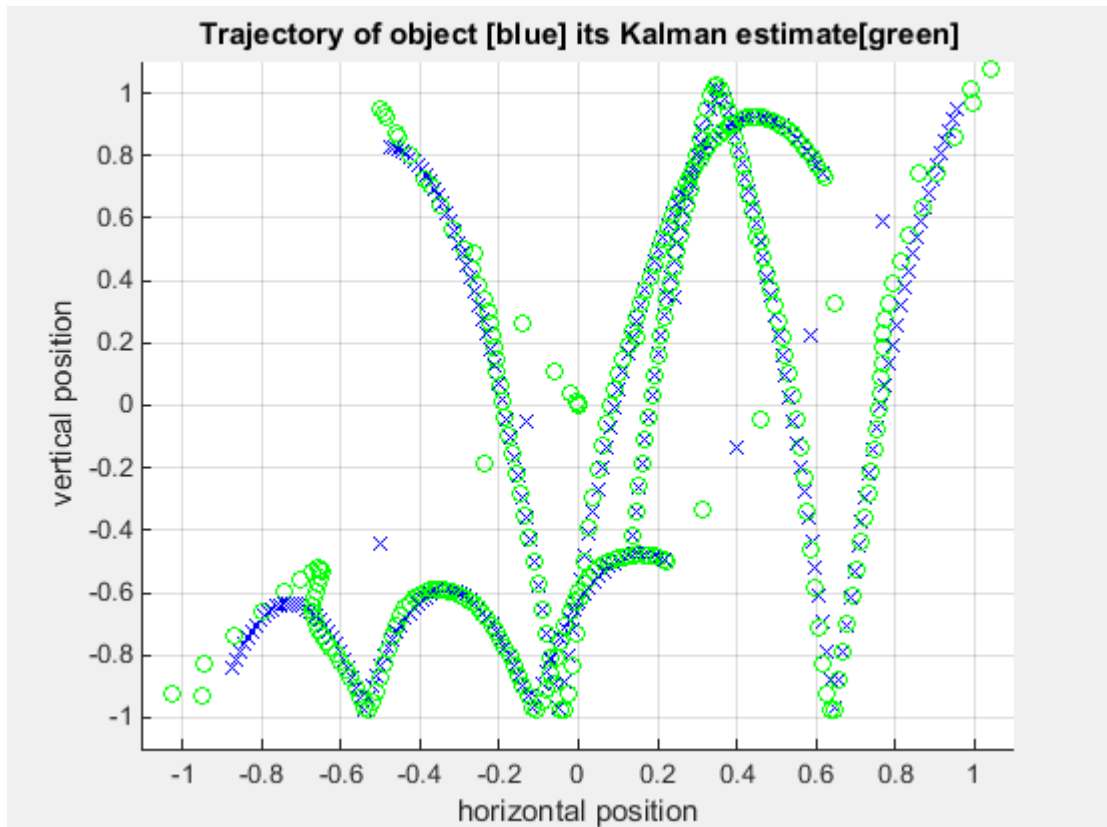
```
cd work
```

where *work* is the full path name of the work folder containing your files. For more information, see “Files and Folders that MATLAB Accesses”.

- 2 At the MATLAB command line, enter:

```
test01
```

The test script runs and plots the trajectory of the object in blue and the Kalman filter estimated position in green. Initially, you see that it takes a short time for the estimated position to converge with the actual position of the object. Then three sudden shifts in position occur—each time the Kalman filter readjusts and tracks the object after a few iterations.



Setting Up Your C Compiler

MATLAB Coder automatically locates and uses a supported installed compiler. For the current list of supported compilers, see [Supported and Compatible Compilers](#) on the MathWorks Web site.

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

Considerations for Making Your Code Suitable for Code Generation

Designing for Code Generation

Before generating code, you must prepare your MATLAB code for code generation. The first step is to eliminate unsupported constructs.

Checking for Violations at Design Time

There are two tools that help you detect code generation issues at design time: the code analyzer and the code generation readiness tool.

You use the code analyzer in the MATLAB Editor to check for code violations at design time, minimizing compilation errors. The code analyzer continuously checks your code as you enter it. It reports problems and recommends modifications to maximize performance and maintainability.

To use the code analyzer to identify warnings and errors specific to MATLAB for code generation, you must add the `%#codegen` directive (or pragma) to your MATLAB file. A complete list of MATLAB for Code Generation code analyzer messages is available in the MATLAB Code Analyzer preferences. See “Running the Code Analyzer Report” for more details.

Note: The code analyzer might not detect all MATLAB for code generation issues. After eliminating errors or warnings that the code analyzer detects, compile your code with MATLAB Coder to determine if the code has other compliance issues.

The code generation readiness tool screens MATLAB code for features and functions that are not supported for code generation. The tool provides a report that lists the source files that contain unsupported features and functions and an indication of how much work is required to make the MATLAB code suitable for code generation.

You can access the code generation readiness tool in the following ways:

- In the current folder browser — by right-clicking a MATLAB file
- At the command line — by using the `coder.screener` function.
- In a project — when you add a MATLAB file to a project, if MATLAB Coder detects code generation issues, it provides a link to the code generation readiness report.

Checking for Violations at Code Generation Time

You can use `codegen` to check for violations at code generation time. `codegen` checks that your MATLAB code is suitable for code generation.

When `codegen` detects errors or warnings, it automatically generates an error report that describes the issues and provides links to the offending MATLAB code. For more information, see “Code Generation Reports”.

After code generation, `codegen` generates a MEX function that you can use to test your implementation in MATLAB.

Checking for Violations at Run Time

You can use `codegen` to generate a MEX function and check for violations at run time. In simulation, the code generated for your MATLAB functions includes the run-time checks. Disabling run-time checks and extrinsic calls usually results in streamlined generated code and faster simulation. You control run-time checks using the MEX configuration object, `coder.MexCodeConfig`. For more information, see “Control Run-Time Checks”.

If you encounter run-time errors in your MATLAB functions, a run-time stack appears automatically in the MATLAB Command Window. See “Debug Run-Time Errors”.

Making the MATLAB Code Suitable for Code Generation

Making Your Code Suitable for Code Generation

To modify the code yourself, work through the exercises in this section. Otherwise, open the supplied file `kalman02.m` in your *solutions* subfolder to see the modified algorithm.

To begin the process of making your MATLAB code suitable for code generation, you work with the file `kalman01.m`. This code is a MATLAB version of a scalar Kalman filter that estimates the state of a dynamic system from a series of noisy measurements.

- 1 Set your MATLAB current folder to the work folder that contains your files for this tutorial. At the MATLAB command line, enter:

```
cd work
```

where *work* is the full path name of the work folder containing your files. See “Files and Folders that MATLAB Accesses”.

- 2 Open `kalman01.m` in the MATLAB Editor. At the MATLAB command line, enter:

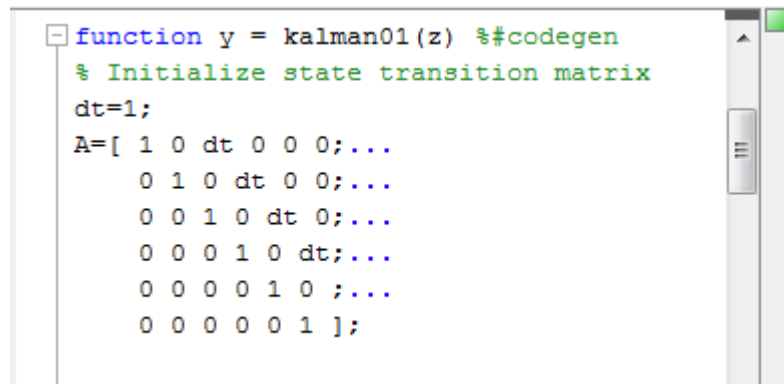
```
edit kalman01.m
```

The file opens in the MATLAB Editor. The code analyzer message indicator in the top right corner of the MATLAB Editor is green, which indicates that it has not detected errors, warnings, or opportunities for improvement in the code.

- 3 Turn on MATLAB for code generation error checking by adding the `%#codegen` directive after the function declaration.

```
function y = kalman01(z) %#codegen
```

The code analyzer message indicator remains green, indicating that it has not detected code generation related issues.



```
function y = kalman01(z) %#codegen
% Initialize state transition matrix
dt=1;
A=[ 1 0 dt 0 0 0;...
    0 1 0 dt 0 0;...
    0 0 1 0 dt 0;...
    0 0 0 1 0 dt;...
    0 0 0 0 1 0 ;...
    0 0 0 0 0 1 ];
```

For more information on using the code analyzer, see “Running the Code Analyzer Report”.

- 4 Save the file in the current folder as `kalman02.m`:
 - a To match the function name to the file name, change the function name to `kalman02`.

```
function y = kalman02(z)
```

- b In the MATLAB Editor, select **Save As** from the **File** menu.
- c Enter `kalman02.m` as the new file name.

Note: If you do not match the file name to the function name, the code analyzer warns you that these names are not the same and highlights the function name

in orange to indicate that it can provide an automatic correction. For more information, see “Changing Code Based on Code Analyzer Messages”.

- d** Click **Save**.

You are now ready to compile your code using `codegen`. By default, `codegen` checks that your MATLAB code is suitable for code generation. Then, after compilation, `codegen` generates a MEX function that you can test in MATLAB.

Generating a MEX Function Using `codegen`

Because C uses static typing, `codegen` must determine the properties of all variables in the MATLAB files at compile time. Therefore, you must specify the properties of all function inputs at the same time as you compile the file with `codegen`.

To compile `kalman02.m`, you must specify the size of the input vector `y`.

- 1** Load the `position.mat` file into your MATLAB workspace.

```
load position.mat
```

This command loads a matrix `position` containing the `x` and `y` coordinates of 310 points in Cartesian space.

- 2** Get the first vector in the position matrix.

```
z = position(1:2,1);
```

- 3** Compile the file `kalman02.m` using `codegen`.

```
codegen -report kalman02.m -args {z}
```

`codegen` reports that the code generation is complete. By default, it generates a MEX function, `kalman02_mex`, in the current folder and provides a link to the code generation report.

Note that:

- The `-report` option instructs `codegen` to generate a code generation report, which you can use to debug your MATLAB code and verify that it is suitable for code generation.
- The `-args` option instructs `codegen` to compile the file `kalman02.m` using the class, size, and complexity of the sample input parameter `z`.

You have proved that the Kalman filter example code is suitable for code generation using `codegen`. You are ready to begin the next task in this tutorial, “Verifying the MEX Function” on page 2-45.

Verifying the MEX Function

In this part of the tutorial, you test the MEX function to verify that it provides the same functionality as the original MATLAB code.

In addition, simulating your algorithm in MATLAB before generating C code enables you to detect and fix run-time errors that would be much harder to diagnose in the generated C code. By default, the following run-time checks execute when you simulate your MEX function in MATLAB:

- Memory integrity checks. These checks perform array bounds and dimension checking and detect violations of memory integrity in code generated for MATLAB functions. If a violation is detected, MATLAB stops execution with a diagnostic message.
- Responsiveness checks in code generated for MATLAB functions. These checks enable periodic checks for **Ctrl+C** breaks in code generated for MATLAB functions, allowing you to terminate execution with **Ctrl+C**.

For more information, see “Control Run-Time Checks”.

Running the Generated MEX Function

You run the MEX function, `kalman02_mex`, using `coder.runTest` to call the test file, `test02`. This test file is the same as `test01` that you used in “Running the Original MATLAB Code” on page 2-39 except that it calls `kalman02` instead of `kalman01`.

Contents of test02.m

```
% Figure setup
clear all;
load position.mat
numPts = 300;
figure;hold;grid;

% Kalman filter loop
for idx = 1: numPts
    % Generate the location data
```

```
z = position(:,idx);

% Use Kalman filter to estimate the location
y = kalman02(z);

% Plot the results
plot_trajectory(z,y);
end
hold;
```

`coder.runTest` runs the test file and replaces calls to the MATLAB algorithm with calls to the MEX function.

```
coder.runTest('test02','kalman02')
```

`coder.runTest` runs the MEX function, `kalman02_mex`, using the same inputs you used in “Running the Original MATLAB Code” on page 2-39.

The test script runs and plots the trajectory of the object and the Kalman filter estimated position as before.

You have generated a MEX function for your MATLAB code, verified that it is functionally equivalent to your original MATLAB code, and checked for run-time errors. Now you are ready to begin the next task in this tutorial, “Generating C Code Using `codegen`” on page 2-46.

Generating C Code Using `codegen`

In this task, you use `codegen` to generate C code for your MATLAB filter algorithm. You then view the generated C code using the MATLAB Coder code generation report and compare the generated C code with the original MATLAB code. You use the supplied build script `build02.m` to generate code.

About the Build Script

A build script automates a series of MATLAB commands that you want to perform repeatedly from the command line, saving you time and eliminating input errors.

The build script `build02.m` contains:

```
% Load the position vector
load position.mat
```

```
% Get the first vector in the position matrix
% to use as an example input
z = position(1:2,1);
% Generate C code only, create a code generation report
codegen -c -d build02 -config coder.config('lib')
        -report kalman02.m -args {z}
```

Note that:

- `codegen` opens the file `kalman02.m` and automatically translates the MATLAB code into C source code.
- The `-c` option instructs `codegen` to generate code only, without compiling the code to an object file. This option enables you to iterate rapidly between modifying MATLAB code and generating C code.
- The `-config coder.config('lib')` option instructs `codegen` to generate embeddable C code suitable for targeting a static library instead of generating the default MEX function. For more information, see `coder.config`.
- The `-d` option instructs `codegen` to generate code in the output folder `build02`.
- The `-report` option instructs `codegen` to generate a code generation report that you can use to debug your MATLAB code and verify that it is suitable for code generation.
- The `-args` option instructs `codegen` to compile the file `kalman01.m` using the class, size, and complexity of the sample input parameter `z`.

How to Generate C Code

- 1 Run the build script.

```
build02
```

MATLAB processes the build file and outputs the message:

```
Code generation successful: View report.
codegen generates files in the folder, build02.
```

- 2 To view the code generation report, click *View report*.

The MATLAB Coder Code Generation Report opens and displays the generated code, `kalman02.c`.

The file appears in the right pane. The code generation report provides a hyperlink to open the C code in the MATLAB Editor.

To learn more about the report, see “Code Generation Reports”.

Comparing the Generated C Code to Original MATLAB Code

To compare your generated C code to the original MATLAB code, open the C file, `kalman02.c`, and the `kalman02.m` file in the MATLAB Editor. View the files side by side by selecting the **VIEW** tab and then clicking **Left/Right**.

```

8 function y = kalman02(z) %#codegen
9 % Initialize state transition matrix
10 dt=1;
11 A=[ 1 0 dt 0 0 0;...
12     0 1 0 dt 0 0;...
13     0 0 1 0 dt 0;...
14     0 0 0 1 0 dt;...
15     0 0 0 0 1 0;...
16     0 0 0 0 0 1 ];
17
18 % Measurement matrix
19 H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
20 Q = eye(6);
21 R = 1000 * eye(2);
22
23 % Initial conditions
24 persistent x_est p_est
25 if isempty(x_est)
26     x_est = zeros(6, 1);
27     p_est = zeros(6, 6);
28 end
29
26 void kalman02(const real_T z[2], real_T y[2])
27 {
28     int32_T r2;
29     int8_T Q[36];
30     int32_T r1;
31     real_T x_prd[6];
32     static int8_T iv0[36] = { 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
33                             0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1 };
34
35     real_T klm_gain[36];
36     int32_T k;
37     real_T a21;
38     static int8_T iv1[36] = { 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0,
39                             0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1 };
40
41     real_T p_prd[36];
42     real_T b_klm_gain[12];
43     static int8_T iv2[12] = { 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 };
44
45     static int8_T iv3[12] = { 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 };
46
47     static int16_T iv4[4] = { 1000, 0, 0, 1000 };

```

Here are some important points about the generated C code:

- The function signature is:

```
void kalman02(const double z[2], double y[2])
```

`Z` corresponds to the input `z` in your MATLAB code. The size of `z` is 2, which corresponds to the total size (2 x 1) of the example input you used when you compiled your MATLAB code.

- You can easily compare the generated C code to your original MATLAB code. In the generated C code:
 - Your function name is unchanged.
 - Your comments are preserved in the same position.
 - Your variable names are the same as in the original MATLAB code.

Note: If a variable in your MATLAB code is set to a constant value, it does not appear as a variable in the generated C code. Instead, the generated C code contains the actual value of the variable.

Modifying the Filter to Accept a Fixed-Size Input

The filter you have worked on so far in this tutorial uses a simple batch process that accepts one input at a time, so you must call the function repeatedly for each input. In this part of the tutorial, you learn how to modify the algorithm to accept a fixed-sized input, which makes the algorithm suitable for frame-based processing.

Modifying Your MATLAB Code

To modify the code yourself, work through the exercises in this section. Otherwise, open the supplied file `kalman03.m` in your *solutions* subfolder to see the modified algorithm.

The filter algorithm you have used so far in this tutorial accepts only one input. You can now modify the algorithm to process a vector containing more than one input. You need to find the length of the vector and call the filter code for each element in the vector in turn. You do this by calling the filter algorithm in a `for`-loop.

- 1 Open `kalman02.m` in the MATLAB Editor.

```
edit kalman02.m
```

- 2 Add a `for`-loop around the filter code.

- a Before the comment

```
% Predicted state and covariance
insert:
```

```
for i=1:size(z,2)
```

- b After

```
% Compute the estimated measurements
y = H * x_est;
insert:
```

```
end
```

Your filter code should now look like this:

```
for i=1:size(z,2)
% Predicted state and covariance
x_prd = A * x_est;
p_prd = A * p_est * A' + Q;

% Estimation
```

```
S = H * p_prd' * H' + R;
B = H * p_prd';
klm_gain = (S \ B)';

% Estimated state and covariance
x_est = x_prd + klm_gain * (z - H * x_prd);
p_est = p_prd - klm_gain * H * p_prd;

% Compute the estimated measurements
y = H * x_est;
end
```

- 3** Modify the line that calculates the estimated state and covariance to use the i^{th} element of input z .

Change

```
x_est = x_prd + klm_gain * (z - H * x_prd);
to
```

```
x_est = x_prd + klm_gain * (z(:,i) - H * x_prd);
```

- 4** Modify the line that computes the estimated measurements to append the result to the i^{th} element of the output y .

Change

```
y = H * x_est;
to
```

```
y(:,i) = H * x_est;
```

The code analyzer message indicator in the top right turns red to indicate that the code analyzer has detected an error. The code analyzer underlines the offending code in red and places a red marker to the right.

- 5** Move your pointer over the red marker to view the error.

The code analyzer reports that code generation requires variable y to be fully defined before subscripting it.

Why Preallocate the Outputs?

You must preallocate outputs here because the MATLAB for code generation does not support increasing the size of an array over time. Repeatedly expanding the size of an array over time can adversely affect the performance of your program. See “Preallocating Memory”.

```

30 - for i=1:size(z,2)
31     % Predicted state and covariance
32     x_prd = A * x_est;
33     p_prd = A * p_est * A' + Q;
34
35     % Estimation
36     S = H * p_prd' * H' + R;
37     B = H * p_prd';
38     klm_gain = (S \ B)';
39
40     % Estimated state and covariance
41     x_est = x_prd + klm_gain * (z(:,i) - H * x_prd);
42     % Line 45: Code generation requires variable 'y' to be fully defined before subscripting it. Details
43
44     % Compute the estimated measurements
45     y(:,i) = H * x_est;
46 - end

```

- 6 To address the error, preallocate memory for the output y , which is the same size as the input z . Add this code before the `for`-loop.

```

% Pre-allocate output signal:
y=zeros(size(z));

```

The red error marker disappears and the code analyzer message indicator in the top right edge of the code turns green, which indicates that you have fixed the errors and warnings detected by the code analyzer.

For more information on using the code analyzer, see “Running the Code Analyzer Report”.

- 7 Change the function name to `kalman03` and save the file as `kalman03.m` in the current folder.

You are ready to begin the next task in the tutorial, “Testing Your Modified Algorithm” on page 2-52.

Testing Your Modified Algorithm

Use the test script `test03.m` to test `kalman03.m`. This script sets the frame size to 10 and calculates the number of frames in the example input. It then calls the Kalman filter and plots the results for each frame in turn.

At the MATLAB command line, enter:

```
test03
```

The test script runs and plots the trajectory of the object and the Kalman filter estimated position as before.

You are ready to begin the next task in the tutorial, “Generating C Code for Your Modified Algorithm” on page 2-52.

Note: Before generating C code, it is best practice to generate a MEX function that you can execute within the MATLAB environment to test your algorithm and check for run-time errors.

Generating C Code for Your Modified Algorithm

You use the supplied build script `build03.m` to generate code. The only difference between this build script and the script for the initial version of the filter is the example input used when compiling the file. `build03.m` specifies that the input to the function is a matrix containing five 2x1 position vectors, which corresponds to a frame size of 10.

Contents of `build03.m`

```
% Load the position vector
load position.mat
% Get the first 5 positions in the position matrix to use
% as an example input
z = position(1:2,1:5);
% Generate C code only, create a code generation report
codegen -c -config coder.config('lib') -report kalman03.m -args {z}
```


To generate C code for `kalman03`:

- 1 At the MATLAB command line, enter:

```
build03
```

MATLAB processes the build file and outputs the message:

```
Code generation successful: View report.
```

The generated C code is in `work\codegen\lib\kalman03`, where `work` is the folder that contains your tutorial files.

- 2 To view the generated C code:

- a Click *View report*.

The MATLAB Coder Code Generation Report opens and displays the generated coder, `kalman03.c`.

- 3 Compare the generated C code with the C code for the scalar Kalman filter. You see that the code is almost identical except that there is now a `for`-loop for the frame processing.

Here are some important points about the generated C code:

- The function signature is now:

```
void kalman03(const double z[10], double y[10])
```

The size of `z` and `y` is now 10, which corresponds to the size of the example input `z` (2x5) used to compile your MATLAB code.

- The filtering now takes place in a `for`-loop. The `for`-loop iterates over all 5 inputs.

```
for(i = 0; i < 5; i++)
{
    /* Predicted state and covariance */ ...
}
```

Modifying the Filter to Accept a Variable-Size Input

The algorithm you have used so far in this tutorial is suitable for processing input data that consists of fixed-size frames. In this part of the tutorial, you test your algorithm with variable-size inputs and see that the algorithm is suitable for processing packets of data of varying size. You then learn how to generate code for a variable-size input.

Testing the Algorithm with Variable-Size Inputs

Use the test script `test04.m` to test `kalman03.m` with variable-size inputs.

The test script calls the filter algorithm in a loop, passing a different size input to the filter each time. Each time through the loop, the test script calls the `plot_trajectory` function for every position in the input.

To run the test script, at the MATLAB command line, enter:

```
test04
```

The test script runs and plots the trajectory of the object and the Kalman filter estimated position as before.

You have created an algorithm that accepts variable-size inputs. You are ready to begin the next task in the tutorial, “Generating C Code for a Variable-Size Input” on page 2-54.

Note: Before generating C code, it is best practice to generate a MEX function that you can execute within the MATLAB environment to test your algorithm and check for run-time errors.

Generating C Code for a Variable-Size Input

You use the supplied build script `build04.m` to generate code.

About the Build Script

Contents of `build04.m`

```
% Load the position vector
load position.mat
N=100;
% Get the first N vectors in the position matrix to
% use as an example input
z = position(1:2,1:N);
% Specify the upper bounds of the variable-size input z
% using the coder.typeof declaration - the upper bound
% for the first dimension is 2; the upper bound for
% the second dimension is N. The first dimension is fixed,
% the second is variable.
```

```

eg_z = coder.typeof(z, [2 N], [0 1]);
% Generate C code only
% specify upper bounds for variable-size input z
codegen -c -config coder.config('lib') -report kalman03.m -args {eg_z}

```

This build file:

- Specifies the upper bounds explicitly for the variable-size input using the declaration `coder.typeof(z, [2 N], [0 1])` with the `-args` option on the `codegen` command line. The second input, `[2 N]`, specifies the size and upper bounds of the variable size input `z`. Because $N=100$, `coder.typeof` specifies that the input to the function is a matrix with two dimensions, the upper bound for the first dimension is `2`; the upper bound for the second dimension is `100`. The third input specifies which dimensions are variable. A value of `true` or one means that the corresponding dimension is variable; a value of `false` or zero means that the corresponding dimension is fixed. The value `[0 1]` specifies that the first dimension is fixed, the second dimension is variable. For more information, see “Generating Code for MATLAB Functions with Variable-Size Data”.
- Creates a code configuration object `cfg` and uses it with the `-config` option to specify code generation parameters. For more information, see `coder.config`.

How to Generate C Code for a Variable-Size Input

- 1 Use the build script `build04` to generate C code.

```
build04
```

- 2 View the generated C code as before.

Here are some important points about the generated C code:

- The generated C code can process inputs from 2×1 to 2×100 . The function signature is now:

```

void kalman03(double z_data[200], ...
             int z_size[2], ...
             double y_data[200], int y_size[2])

```

Because `y` and `z` are variable size, the generated code contains two pieces of information about each of them: the data and the actual size of the sample. For example, for variable `z`, the generated code contains:

- The data `z_data[200]`, where `200` is the maximum size specified using `coder.typeof`.

- `z_size[2]`, which contains the actual size of the input data. This information varies each time the filter is called.
- To maximize efficiency, the actual size of the input data `z_size` is used when calculating the estimated position. The filter processes only the number of samples available in the input.

```
for(i = 0; i+1 <= z_size[1]; i++) {  
    /* Predicted state and covariance */  
    for(k = 0; k < 6; k++) {  
        ...  
    }  
}
```

Key Points to Remember

- Back up your MATLAB code before you modify it.
- Decide on a naming convention for your files and save interim versions frequently. For example, this tutorial uses a two-digit suffix to differentiate the various versions of the filter algorithm.
- Use build scripts to build your files.
- Use test scripts to separate the pre- and post-processing from the core algorithm.
- Generate a MEX function before generating C code. Use this MEX function to simulate your algorithm in MATLAB to validate its operation and check for run-time errors.
- Use the `-args` option to specify input parameters at the command line.
- Use the `-report` option to create a code generation report.
- Use `coder.typeof` to specify variable-size inputs.
- Use the code generation configuration object (`coder.config`) to specify parameters for standalone C code generation.

Best Practices Used in This Tutorial

Best Practice — Preserving Your Code

Preserve your code before making further modifications. This practice provides a fallback in case of error and a baseline for testing and validation. Use a consistent file

naming convention. For example, add a two-digit suffix to the file name for each file in a sequence.

Best Practice — Comparing Files

Use the MATLAB Compare `Against` option to compare two MATLAB files to examine differences between files.

Best Practice — Generating a Code Generation Report

Use the `-report` option to generate an HTML report with links to your MATLAB code files and compile-time type information for the variables and expressions in your code. This information simplifies finding sources of error messages and aids understanding of type propagation rules. If you do not specify this option, `codegen` generates a report only if errors or warnings occur. For more information, see .

Best Practice — Using Build Scripts

A build script automates a series of MATLAB commands that you want to perform repeatedly from the command line, saving you time and eliminating input errors. For more information, see .

Best Practice — Separating Your Test Bench from Your Function Code

Separate your core algorithm from your test bench. Create a separate test script to do the pre- and post-processing such as loading inputs, setting up input values, calling the function under test, and outputting test results.

Where to Learn More

- “Next Steps” on page 2-58
- “Product Help” on page 2-58
- “MathWorks Online” on page 2-58

Next Steps

To...	See...
See the compilation options for codegen	codegen
Learn how to integrate your MATLAB code with Simulink models	“Track Object Using MATLAB Code”
Learn more about using MATLAB for code generation	“MATLAB Algorithm Design”
Use variable-size data	“Variable-Size Data Definition for Code Generation”
Speed up fixed-point MATLAB code	fiaccel
Integrate custom C code into MATLAB code and generate standalone code	“Specify External File Locations”
Integrate custom C code into a MATLAB function for code generation	coder.ceval
Generate HDL from MATLAB code	www.mathworks.com/products/slhdlcoder

Product Help

MathWorks product documentation is available from the Help menu on the MATLAB desktop.

MathWorks Online

For additional information and support, visit the MATLAB Coder page on the MathWorks Web site at:

www.mathworks.com/products/matlab-coder

MEX Function Generation at the Command Line

In this section...

- “Learning Objectives” on page 2-59
- “Tutorial Prerequisites” on page 2-59
- “Example: Euclidean Minimum Distance” on page 2-60
- “Files for the Tutorial” on page 2-62
- “Tutorial Steps” on page 2-63
- “Key Points to Remember” on page 2-80
- “Best Practices Used in This Tutorial” on page 2-80
- “Where to Learn More” on page 2-81

Learning Objectives

In this tutorial, you will learn how to:

- Automatically generate a MEX function from your MATLAB code.
- Define function input properties at the command line.
- Specify the upper bounds of variable-size data.
- Specify variable-size inputs.
- Generate a code generation report that you can use to debug your MATLAB code and verify that it is suitable for code generation.

Tutorial Prerequisites

- “What You Need to Know” on page 2-59
- “Required Products” on page 2-59

What You Need to Know

To complete this tutorial, you should have basic familiarity with MATLAB software.

Required Products

To complete this tutorial, you must install the following products:

- MATLAB
- MATLAB Coder
- C compiler

For most platforms, a default compiler is supplied with MATLAB.

MATLAB Coder automatically locates and uses a supported installed compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks Web site.

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

For instructions on installing MathWorks products, refer to the installation documentation. If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window.

Example: Euclidean Minimum Distance

- “Description” on page 2-60
- “Algorithm” on page 2-61

Description

The Euclidean distance between points p and q is the length of the line segment \overline{pq} .

In Cartesian coordinates, if $p = (p_1, p_2, \dots, p_n)$ and $q = (q_1, q_2, \dots, q_n)$ are two points in Euclidean n -space, then the distance from p to q is given by:

$$\begin{aligned}d(p, q) &= \|p - q\| \\ &= \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2} \\ &= \sqrt{\sum_{i=1}^n (p_i - q_i)^2}\end{aligned}$$

In one dimension, the distance between two points, x_1 and x_2 , on a line is simply the absolute value of the difference between the two points:

$$\sqrt{(x_2 - x_1)^2} = |x_2 - x_1|$$

In two dimensions, the distance between $p = (p_1, p_2)$ and $q = (q_1, q_2)$ is:

$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}$$

The example for this tutorial computes the minimum Euclidean distance between a column vector x and a collection of column vectors in the codebook matrix cb . The function has three output variables:

- y , the vector in cb with the minimum distance to x
- idx , the index of the column vector in cb corresponding to the closest vector
- $distance$, the distance between x and y

Algorithm

This algorithm computes the minimum Euclidean distance between a column vector x and a collection of column vectors in the codebook matrix cb . The algorithm computes the minimum distance to x and finds the column vector in cb that is closest to x . It outputs this column vector, y , its index, idx , in cb , and $distance$, the distance between x and y .

The function signature for the algorithm is:

```
function [y,idx,distance] = euclidean(x,cb)
```

The minimum distance is initially set to the first element of cb .

```
idx=1;
distance=norm(x-cb(:,1));
```

The minimum distance calculation is performed in the `for`-loop.

```
for index=2:size(cb,2)
    d=norm(x-cb(:,index));
    if d < distance
        distance=d;
        idx=index;
    end
end
```

The output y is set to the minimum distance vector.

```
y=cb(:,idx);
```

Files for the Tutorial

- “About the Tutorial Files” on page 2-62
- “Location of Files” on page 2-62
- “Names and Descriptions of Files” on page 2-62

About the Tutorial Files

The tutorial uses the following files:

- Example MATLAB code files for each step of the tutorial.

Throughout this tutorial, you work with MATLAB files that contain a simple Euclidean distance algorithm.

- Build scripts that you use to compile your function code.
- Test files that:
 - Perform the preprocessing functions, for example, setting up input data.
 - Call the specified Euclidean function.
 - Perform the post-processing functions, for example, plotting the distances.
- A MAT-file that contains example input data.

Location of Files

The tutorial files are available in the following folder: `docroot\toolbox\coder\examples\euclidean`. To run the tutorial, you must copy these files to a local folder. For instructions, see “Copying Files Locally” on page 2-64.

Names and Descriptions of Files

Type	Name	Description
Function code	<code>euclidean01.m</code>	Baseline MATLAB implementation of Euclidean minimum distance algorithm including plot functions.
	<code>euclidean02.m</code>	Version of the original algorithm suitable for code generation with extrinsic calls to the pause function.
	<code>euclidean03.m</code>	Version of the original algorithm without plotting functions.

Type	Name	Description
	<code>euclidean04.m</code>	Modified algorithm that uses <code>assert</code> to specify the upper bounds of variable <code>N</code> .
Build script	<code>build01.m</code>	Build script for <code>euclidean03.m</code> .
	<code>build02.m</code>	Build script for <code>euclidean03.m</code> specifying two-dimensional inputs.
	<code>build03.m</code>	Build script for <code>euclidean03.m</code> specifying variable-size inputs.
	<code>build04.m</code>	Build script for <code>euclidean04.m</code> .
Test script	<code>test01.m</code>	Initial version of test script, includes plot functions. Tests <code>euclidean03</code> MEX function.
	<code>test02.m</code>	Tests the three-dimensional <code>euclidean03</code> MEX function with two-dimensional inputs.
	<code>test03.m</code>	Tests the two-dimensional <code>euclidean04</code> MEX function with two-dimensional inputs.
	<code>test04.m</code>	Tests <code>euclidean03_varsize</code> MEX function with two-dimensional and three-dimensional inputs.
	<code>test05.m</code>	Tests <code>euclidean04</code> MEX function specifying how many elements of each input to process.
MAT-file	<code>euclidean.mat</code>	Contains the input data used by the algorithm.

Tutorial Steps

- “Copying Files Locally” on page 2-64
- “Running the Original MATLAB Code” on page 2-64
- “Setting Up Your C Compiler” on page 2-67
- “Considerations for Making Your Code Compliant” on page 2-67
- “Making the MATLAB Code Suitable for Code Generation” on page 2-68
- “Generating a MEX Function Using `codegen`” on page 2-69
- “Validating the MEX Function” on page 2-71
- “Using Build and Test Scripts” on page 2-72
- “Elaborating the Algorithm to Accept Variable-Size Inputs” on page 2-75

- “Specifying Upper Bounds for Local Variables” on page 2-78

Copying Files Locally

Copy the tutorial files to a local solutions folder and create a local working folder:

- 1 Create a local *solutions* folder, for example, `c:\coder\euclidean\solutions`.
- 2 Change to the `docroot\toolbox\coder\examples` folder. At the MATLAB command line, enter:

```
cd(fullfile(docroot, 'toolbox', 'coder', 'examples'))
```

- 3 Copy the contents of the `euclidean` subfolder to your local *solutions* folder, specifying the full pathname of the *solutions* folder:

```
copyfile('euclidean', 'solutions')
```

Your *solutions* folder now contains a complete set of solutions for the tutorial. If you do not want to perform the steps for each task in the tutorial, you can view the solutions to see how the code should look.

- 4 Create a local *work* folder, for example, `c:\coder\euclidean\work`.
- 5 Copy the following files from your *solutions* folder to your *work* folder.

- `euclidean01.m`
- `euclidean.mat`
- Build files `build01.m` through `build04.m`
- Test scripts `test01.m` through `test05.m`

Your *work* folder now contains the files that you need to get started with the tutorial.

Running the Original MATLAB Code

In this tutorial, you work with a MATLAB function that implements the Euclidean distance minimizing algorithm. You make the MATLAB version of this algorithm suitable for code generation and test the resulting MEX function to validate the functionality of the modified code. As you work through the tutorial, you refine the design of the algorithm to accept variable-size inputs.

Before generating a MEX function, run the original MATLAB function to see how the Euclidean distance minimizing algorithm works.

- 1 Set your MATLAB current folder to the *work* folder that contains your files for this tutorial.

```
cd work
```

work is the full path name of the work folder containing your files. For more information, see “Files and Folders that MATLAB Accesses”.

- 2 Load the `euclidean.mat` file into your MATLAB workspace.

```
load euclidean.mat
```

Your MATLAB workspace now contains:

- A matrix `x` containing 40000 three-dimensional vectors.
- A matrix `cb` containing 216 three-dimensional vectors.

The Euclidean algorithm minimizes the distance between a column vector, `x1`, taken from matrix `x`, and the column vectors in the codebook matrix `cb`. It outputs the column vector in `cb` that is closest to `x1`.

- 3 Create a single input vector `x1` from the matrix `x`.

```
x1=x(:,1)
```

The result is the first vector from `x`:

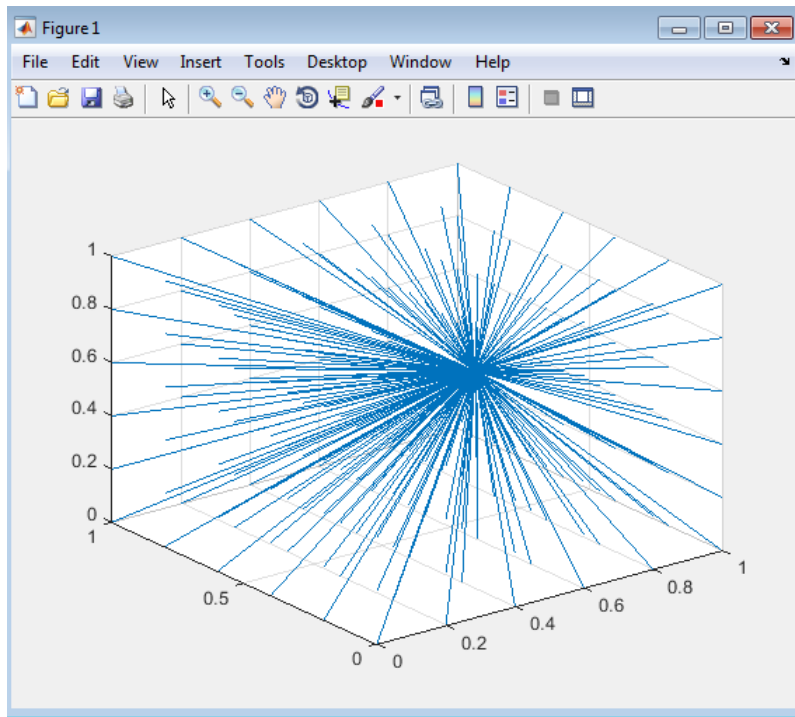
```
x1 =
```

```
    0.8568  
    0.7455  
    0.3835
```

- 4 Use the Euclidean algorithm to find the vector in codebook matrix `cb` that is closest to `x1`. At the MATLAB command line, enter:

```
[y, idx, distance]=euclidean01(x1,cb)
```

The Euclidean algorithm runs and plots the lines from `x1` to each vector in `cb`.



After completing the algorithm, it outputs the coordinates of the point y , which is the vector in cb closest to $x1$, together with the index idx of $x1$ in cb , and the distance, $distance$, between y and $x1$.

```
y =  
    0.8000  
    0.8000  
    0.4000
```

```
idx =  
    171
```

```
distance =  
    0.0804
```

The algorithm computes that the point $y=0.8000, 0.8000, 0.4000$, the 171st vector in cb , is closest to point $x1$. The distance between y and $x1$ is 0.0804.

Where to Go Next

Before continuing with the tutorial, you must set up your C compiler as detailed in “Setting Up Your C Compiler” on page 2-67.

Setting Up Your C Compiler

MATLAB Coder automatically locates and uses a supported installed compiler. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks Web site.

You can use `mex -setup` to change the default compiler. See “Changing Default Compiler”.

Considerations for Making Your Code Compliant

Designing for Code Generation

Before generating code, you must prepare your MATLAB code for code generation. The first step is to eliminate unsupported constructs.

Checking for Violations at Design Time

There are two tools that help you detect code generation issues at design time: the code analyzer and the code generation readiness tool.

You use the code analyzer in the MATLAB Editor to check for code violations at design time, minimizing compilation errors. The code analyzer continuously checks your code as you enter it. It reports problems and recommends modifications to maximize performance and maintainability.

To use the code analyzer to identify warnings and errors specific to MATLAB for code generation, you must add the `%#codegen` directive (or pragma) to your MATLAB file. A complete list of MATLAB for Code Generation code analyzer messages is available in the MATLAB Code Analyzer preferences. See “Running the Code Analyzer Report” for more details.

Note: The code analyzer might not detect all MATLAB for code generation issues. After eliminating errors or warnings that the code analyzer detects, compile your code with MATLAB Coder to determine if the code has other compliance issues.

The code generation readiness tool screens MATLAB code for features and functions that are not supported for code generation. The tool provides a report that lists the source files

that contain unsupported features and functions and an indication of how much work is required to make the MATLAB code suitable for code generation.

You can access the code generation readiness tool in the following ways:

- In the current folder browser — by right-clicking a MATLAB file
- At the command line — by using the `coder.screener` function.
- In a project — when you add a MATLAB file to a project, if MATLAB Coder detects code generation issues, it provides a link to the code generation readiness report.

Checking for Violations at Code Generation Time

You can use `codegen` to check for violations at code generation time. `codegen` checks that your MATLAB code is suitable for code generation.

When `codegen` detects errors or warnings, it automatically generates an error report that describes the issues and provides links to the offending MATLAB code. For more information, see “Code Generation Reports”.

After code generation, `codegen` generates a MEX function that you can use to test your implementation in MATLAB.

Checking for Violations at Run Time

You can use `codegen` to generate a MEX function and check for violations at run time. In simulation, the code generated for your MATLAB functions includes the run-time checks. Disabling run-time checks and extrinsic calls usually results in streamlined generated code and faster simulation. You control run-time checks using the MEX configuration object, `coder.MexCodeConfig`. For more information, see “Control Run-Time Checks”.

If you encounter run-time errors in your MATLAB functions, a run-time stack appears automatically in the MATLAB Command Window. See “Debug Run-Time Errors”.

Where to Go Next

The next section of the tutorial, “Making Your Code Suitable for Code Generation” on page 2-68, shows you how to use the MATLAB code analyzer and `codegen` to make your code suitable for code generation.

Making the MATLAB Code Suitable for Code Generation

Making Your Code Suitable for Code Generation

To begin the process of making your MATLAB code suitable for code generation, you work with the `euclidean01.m` file. This file is a MATLAB version of a three-

dimensional Euclidean example that plots the distances between an input vector `x` and each of the vectors in the codebook matrix `cb`. It determines which vector in `cb` is closest to `x`, and outputs this vector, its position in `cb`, and the distance to `y`.

- 1 In your *work* folder, open `euclidean01.m` in the MATLAB Editor.

```
edit euclidean01.m
```

The file opens. The code analyzer message indicator in the top right corner of the MATLAB Editor is green, which indicates that the code analyzer has not detected errors, warnings, or opportunities for improvement in the code.

- 2 Turn on code generation error checking by adding the `%#codegen` compilation directive after the function declaration.

```
function [ y, idx, distance ] = ...
    euclidean01( x, cb ) %#codegen
```

The code analyzer message indicator remains green, indicating that it has not detected code generation issues.

For more information on using the code analyzer, see “Running the Code Analyzer Report”.

- 3 Change the function name to `euclidean02` and save the file as `euclidean02.m` in the current folder.

You are now ready to compile your code using `codegen`, which checks that your code is suitable for code generation. After code generation, `codegen` generates a MEX function that you can test in MATLAB.

Generating a MEX Function Using `codegen`

About `codegen`

You generate MEX functions using `codegen`, a function that compiles MATLAB code to a MEX function. `codegen` also checks that your MATLAB code is suitable for code generation.

Using `codegen`

Because C uses static typing, `codegen` must determine the properties of all variables in the MATLAB files at compile time. Therefore, you must specify the properties of all function inputs at the same time as you compile the file with `codegen`. To compile `euclidean02.m`, you must specify the size of the input vector `x` and the codebook matrix `cb`.

- 1 Compile the `euclidean02.m` file.

```
codegen euclidean02.m -args {x(:,1), cb}
```

`codegen` reports the following error, which includes a link to the offending line of code and a link to an error report:

```
??? Function 'pause' is not supported for code generation.  
Consider adding coder.extrinsic('pause') at the top of the  
function to bypass code generation.
```

```
Error in ==> euclidean02 Line: 32 Column: 1  
Code generation failed: Open error report.  
Error using codegen
```

Note that:

- By default, `codegen` generates a MEX function named `euclidean02_mex` in the current folder, which allows you to test the original MATLAB code and the MEX function and compare the results.
 - The `-args` option instructs `codegen` to compile the file `euclidean02.m` using the sample input parameters `x(:,1)` and `cb`.
- 2 Click the **euclidean02 Line: 32 Column: 1** link.

The `euclidean02.m` file opens with the cursor at the offending line of code in the `plot_distances` local function.

```
pause;
```

Some MATLAB functions such as `pause` are not supported for code generation. When you call an unsupported MATLAB function, you must declare it to be extrinsic so MATLAB can execute it, but `codegen` does not try to generate code for it. MATLAB Coder automatically treats common MATLAB visualization functions, including `line`, `grid`, `clf`, and `axis` as extrinsic, and excludes them from code generation.

- 3 Declare the function `pause` extrinsic after the function declaration in the local function `plot_distances`:

```
function plot_distances(x,cb)  
% Declare extrinsic function  
coder.extrinsic('pause');
```

- 4 Save the file, then recompile it.

```
codegen -report euclidean02.m -args {x(:,1), cb}
```

This time `codegen` compiles the file and generates a MEX function `euclidean02_mex` in the current folder.

- At the MATLAB command line, click the link to the code generation report and then view the MATLAB code for the `plot_distances` function. The report highlights the extrinsic functions that are supported only within the MATLAB environment.

The screenshot shows the MATLAB IDE interface. On the left, the 'Functions' pane lists `euclidean02` and `euclidean02 > plot_distances`. The main editor displays the MATLAB code for `plot_distances` with line numbers 25 to 35. Line 27 contains a comment: `% Declare extrinsic functions`. Line 28 is `coder.extrinsic('pause');`. Line 33 is `axis([0 1 0 1 0 1]); grid;`. A tooltip points to the `grid` function, stating 'Only supported within the MATLAB environment.'

```

25
26 function plot_distances(x,cb)
27 % Declare extrinsic functions
28 coder.extrinsic('pause');
29
30
31
32
33
34
35

```

The Euclidean minimum distance example code is now suitable for code generation. You are ready to begin the next task in this tutorial, “Validating the MEX Function” on page 2-71.

Validating the MEX Function

In this part of the tutorial, you test the MEX function that you generated in “Generating a MEX Function Using `codegen`” on page 2-69 to verify that it provides the same functionality as the original MATLAB code. You run the MEX function using the same inputs you used in “Running the Original MATLAB Code” on page 2-64.

Running the Generated MEX Function

- Create a single input vector `x1` from the matrix `x`.

```
x1=x(:,1)
```

The result is the first vector in `x`:

```

x1 =
    0.8568
    0.7455
    0.3835

```

- 2 Use the MEX function `euclidean02_mex` to find the vector in codebook matrix `cb` that is closest to `x1`.

```
[y, idx, distance] = euclidean02_mex(x1,cb)
```

The MEX function runs and plots the lines from `x1` to each vector in `cb`. After completing the algorithm, it outputs the coordinates of the point `y`, which is the vector in `cb` closest to `x1`, together with the index `idx` of `y` in `cb`, and the distance, `distance`, between `y` and `x1`.

```
y =  
    0.8000  
    0.8000  
    0.4000
```

```
idx =  
    171
```

```
distance =  
    0.0804
```

The plots and outputs are identical to those generated with the original MATLAB function. The MEX function `euclidean02_mex` is functionally equivalent to the original MATLAB code in `euclidean01.m`.

Using Build and Test Scripts

In “Generating a MEX Function Using MATLAB Coder” on page 2-17, you generated a MEX function for your MATLAB code by calling `codegen` from the MATLAB command line. In this part of the tutorial, you use a build script to generate your MEX function and a test script to test it. The first step is to modify the code in `euclidean02.m` to move the plotting function to a separate test script.

Why Use Build Scripts?

A build script automates a series of MATLAB commands that you want to perform repeatedly from the command line, saving you time and eliminating input errors.

Why Use Test Scripts?

The `euclidean02.m` file contains both the Euclidean minimum distance algorithm and the plot function. It is good practice to separate your core algorithm from your test bench. This practice allows you to reuse your algorithm easily. Create a separate test script to do the pre- and post-processing such as loading inputs, setting up input values, calling the function under test, and outputting test results.

Modifying the Code to Remove the Plot Function

In the file `euclidean02.m`:

- 1 Delete the call to `plot_distances`.
- 2 Delete the local function `plot_distances`.
- 3 Change the function name to `euclidean03` and save the file as `euclidean03.m` in the current folder.

Using the Build Script `build01.m`

Next you use the build script `build01.m` that compiles `euclidean03.m` using `codegen`. This time, use the `-report` option, which instructs `codegen` to generate a code generation report that you can use to debug your MATLAB code and verify that it is suitable for code generation.

Contents of Build File `build01.m`

```
% Load the test data
load euclidean.mat
% Compile euclidean03.m with codegen
codegen -report euclidean03.m -args {x(:,1), cb}
```

At the MATLAB command line, enter:

```
build01
codegen runs and generates a MEX function euclidean03_mex in the current folder.
```

You are ready to test the MEX function `euclidean03_mex`.

Using the Test Script `test01.m`

You use the test script `test01.m` to test the MEX function `euclidean03x`.

The test script:

- Loads the test data from the file `euclidean.mat`.
- Runs the original MATLAB file `euclidean03.m` and plots the distances.
- Runs the MEX function `euclidean03_mex` and plots the distances.

Contents of Test Script `test01.m`

```
% Load test data
load euclidean.mat
% Take a single input vector from the matrix x
x1=x(:,1);
% Run the original MATLAB function
disp('Running MATLAB function euclidean03');
[y, idx, distance] = euclidean03(x1,cb);
disp(['y = ', num2str(y')]);
disp(['idx = ', num2str(idx)]);
disp(['distance = ', num2str(distance)]);
% Visualize the distance minimization
% plot_distances
clf;
for index=1:size(cb,2)
line([x(1,1) cb(1,index)], [x(2,1) cb(2,index)], ...
     [x(3,1) cb(3,index)]);
end
axis([0 1 0 1 0 1]);grid;
pause(.5);
% Run the MEX function euclidean03_mex
disp('Running MEX function euclidean03_mex');
[y, idx, distance] = euclidean03_mex(x1,cb);
disp(['y = ', num2str(y')]);
disp(['idx = ', num2str(idx)]);
disp(['distance = ', num2str(distance)]);
% Visualize the distance minimization
% plot_distances
clf;
for index=1:size(cb,2)
line([x(1,1) cb(1,index)], [x(2,1) cb(2,index)], ...
     [x(3,1) cb(3,index)]);
end
axis([0 1 0 1 0 1]);grid;
pause(.5);
```

Running the Test Script

At the MATLAB command line, enter:

```
test01
```

The test file runs, plots the lines from `x1` to each vector in `cb`, and outputs:

```
Running MATLAB function euclidean03
y = 0.8      0.8      0.4
idx = 171
distance = 0.080374
Running MEX function euclidean03_mex
y = 0.8      0.8      0.4
idx = 171
distance = 0.080374
```

The outputs for the original MATLAB code and the MEX function are identical.

You are now ready to begin the next task in this tutorial, “Elaborating the Algorithm to Accept Variable-Size Inputs” on page 2-75.

Elaborating the Algorithm to Accept Variable-Size Inputs

Why Modify the Algorithm?

The algorithm you have used so far in this tutorial is suitable only to process inputs whose dimensions match the dimensions of the example inputs provided using the `-args` option. In this part of the tutorial, you run `euclidean03_mex` to see that it does not accept two-dimensional inputs. You then recompile your code using two-dimensional example inputs and test the resulting MEX function with the two-dimensional inputs.

About the Build and Test Scripts

Contents of test02.m

This test script creates two-dimensional inputs `x2` and `cb2`, then calls `euclidean03_mex` using these input parameters. You run this test script to see that your existing algorithm does not accept two-dimensional inputs.

```
% Load the test data
load euclidean.mat
```

```
% Create 2-D versions of x and cb
x2=x(1:2,:);
x2d=x2(:,47);
cb2d=cb(1:2,1:6:216);

% Run euclidean03_mex with these 2-D inputs
disp('Attempting to run euclidean03_mex with 2-D inputs');
[y, idx, distance] = euclidean03_mex(x2d,cb2d);
```

Contents of build02.m

This build file creates two-dimensional example inputs `x2d` and `cb2d` then uses these inputs to compile `euclidean03.m`.

```
% Load the test data
load euclidean.mat
% Create 2-D versions of x and cb
x2=x(1:2,:);
x2d=x2(:,47);
cb2d=cb(1:2,1:6:216);
% Recompile euclidean03 with 2-D example inputs
% The -o option instructs codegen to name the MEX function euclidean03_2d
disp('Recompiling euclidean03.m with 2-D example inputs');
codegen -o euclidean03_2d -report euclidean03.m -args {x2d, cb2d};
```

Contents of test03.m

This test script runs the MEX function `euclidean03_2d` with two-dimensional inputs.

```
% Load input data
load euclidean.mat
% Create 2-D versions of x and cb
x2=x(1:2,:);
x2d=x2(:,47);
cb2d=cb(1:2,1:6:216);
% Run new 2-D version of euclidean03
disp('Running new 2-D version of MEX function');
[y, idx, distance] = euclidean03_2d(x2d, cb2d);
disp(['y = ', num2str(y')]);
disp(['idx = ', num2str(idx)]);
```



```
disp(['distance = ', num2str(distance)]);
```

Running the Build and Test Scripts

- 1 Run the test script `test02.m` to test `euclidean03x` with two-dimensional inputs.

```
test02
```

MATLAB reports an error indicating that the MEX function does not accept two-dimensional variables for the input `cb`.

```
??? MATLAB expression 'x' is not of the correct size:
expected [3x1] found [2x1].
```

```
Error in ==> euclidean03
```

To process two-dimensional inputs, you must recompile your code providing two-dimensional example inputs.

- 2 Run the build file `build02.m` to recompile `euclidean03.m` with two-dimensional inputs.

```
build02
```

`codegen` compiles the file and generates a MEX function `euclidean03_2d` in the current folder.

- 3 Run the test file `test03.m` to run the resulting MEX function `euclidean03_2d` with two-dimensional inputs.

At the MATLAB command line, enter:

```
test03
```

This time, the MEX function runs and outputs the vector `y` in matrix `cb` that is closest to `x2d` in two dimensions.

```
Running new 2-D version of MEX function
y = 0          0.4
idx = 3
distance = 0.053094
```

This part of the tutorial demonstrates how to create MEX functions to handle inputs with different dimensions. Using this approach, you would need a library of MEX functions, each one suitable only for inputs with specified data types, dimensions, and complexity. Alternatively, you can modify your code to accept variable-size inputs. To learn how, see “Specifying Variable-Size Inputs” on page 2-78.

Specifying Variable-Size Inputs

The original MATLAB function is suitable for many different size inputs. To provide this same flexibility in your generated C code, use `coder.typeof` with the `codegen -args` command-line option.

`coder.typeof(a,b,1)` specifies a variable-size input with the same class and complexity as `a` and same size and upper bounds as the size vector `b`. For more information, see “Specify Variable-Size Inputs at the Command Line”.

- 1 Compile this code using the build file `build03.m`. This build file uses `coder.typeof` to specify variable-size inputs to the `euclidean03` function.

```
build03
```

`codegen` compiles the file without warnings or errors and generates a MEX function `euclidean03_varsize` in the current folder.

- 2 Run the resulting MEX function with two-dimensional and then three-dimensional inputs using the test file `test04.m`.

At the MATLAB command line, enter:

```
test04
```

The test file runs and outputs:

```
Running euclidean03_varsize with 2-D inputs
```

```
y = 0          0.4
```

```
idx = 3
```

```
distance = 0.053094
```

```
Running euclidean04_varsize with 3-D inputs
```

```
y = 0.6        0.8        0.2
```

```
idx = 134
```

```
distance = 0.053631
```

You have created an algorithm that accepts variable-size inputs.

Specifying Upper Bounds for Local Variables

In this part of the tutorial, you modify the algorithm to compute only the distance between the first `N` elements of a given vector `x` and the first `N` elements of every column vector in the matrix `cb`.

To modify the Euclidean minimum distance algorithm, `euclidean03.m`, to accommodate changes in dimensions over which to compute the distances:

- 1 Provide a new input parameter, `N`, to specify the number of elements to consider. The new function signature is:

```
function [y,idx,distance] = euclidean03(x,cb,N)
```

- 2 Specify an upper bound for the variable `N` using `assert`. Add this line after the function declaration.

```
assert(N<=3);
```

The value of the upper bound must correspond to the maximum number of dimensions of matrix `cb`. If you do not specify an upper bound, an array bounds error occurs if you run the MEX function with a value for `N` that exceeds the number of dimensions of matrix `cb`. For more information, see “Specifying Upper Bounds for Variable-Size Data”.

- 3 Modify the line of code that calculates the initial distance to use `N`. Replace the line:

```
distance=norm(x-cb(:,1));
```

with:

```
distance=norm(x(1:N)-cb(1:N,1));
```

- 4 Modify the line of code that calculates each successive distance to use `N`. Replace the line:

```
d=norm(x-cb(:,index));
```

with:

```
d=norm(x(1:N)-cb(1:N,index));
```

- 5 Change the function name to `euclidean04` and save the file as `euclidean04.m` in the current folder.
- 6 Compile this code using the build file `build04.m`.

At the MATLAB command line, enter:

```
build04
```

`codegen` compiles the file without warnings or errors and generates a MEX function `euclidean04x` in the current folder.

- 7 Run the resulting MEX function to process the first two elements of the inputs `x` and `cb`, then to process all three elements of these inputs. Use the test file `test05.m`.

At the MATLAB command line, enter:

test05

The test file runs and outputs:

```
Running euclidean04_mex for first two elements of inputs x and cb
y = 0.8          0.8          0
idx = 169
distance = 0.078672
Running euclidean04_mex for three elements of inputs x and cb
y = 0.8          0.8          0.4
idx = 171
distance = 0.080374
```

Key Points to Remember

- Back up your MATLAB code before you modify it.
- Decide on a naming convention for your files and save interim versions frequently. For example, this tutorial uses a two-digit suffix to differentiate the various versions of the filter algorithm.
- Use build scripts to build your files.
- Use test scripts to separate the pre- and post-processing from the core algorithm.
- Use the `-args` option to specify input parameters at the command line.
- Use the MATLAB `assert` function to specify the upper bounds of variable-size data.
- Use the `-report` option to create a code generation report.
- Use `coder.typeof(a,b,1)` to specify variable-size inputs.

Best Practices Used in This Tutorial

Best Practice — Preserving Your Code

Preserve your code before making further modifications. This practice provides a fallback in case of error and a baseline for testing and validation. Use a consistent file naming convention. For example, add a 2-digit suffix to the file name for each file in a sequence.

Best Practice — Generating a Code Generation Report

Use the `-report` option to generate an HTML report with links to your MATLAB code files and compile-time type information for the variables and expressions in your code. This information simplifies finding sources of error messages and aids understanding of type propagation rules. If you do not specify this option, `codegen` generates a report only if errors or warnings occur. For more information, see “-report Generate Code Generation Report” on page 3-2.

Where to Learn More

- “Next Steps” on page 2-81
- “Product Help” on page 2-81
- “MathWorks Online” on page 2-82

Next Steps

To...	See...
Learn how to generate C code from your MATLAB code	“C Code Generation at the Command Line” on page 2-31
Learn how to integrate your MATLAB code with Simulink models	“Track Object Using MATLAB Code”
Learn more about using code generation from MATLAB	“MATLAB Algorithm Design”
Use variable-size data	“Variable-Size Data Definition for Code Generation”
Speed up fixed-point MATLAB code	<code>fiaccel</code>
Integrate custom C code into MATLAB code and generate embeddable code	“External Code Integration”
Integrate custom C code into a MATLAB function	<code>coder.ceval</code>
Generate HDL from MATLAB code	www.mathworks.com/products/slhdlcoder

Product Help

MathWorks product documentation is available online from the Help menu on the MATLAB desktop.

MathWorks Online

For additional information and support, visit the MATLAB Coder page on the MathWorks Web site at:

www.mathworks.com/products/featured/matlab-coder

Best Practices for Working with MATLAB Coder

- “Recommended Compilation Options for codegen” on page 3-2
- “Testing MEX Functions in MATLAB” on page 3-3
- “Comparing C Code and MATLAB Code Using Tiling in the MATLAB Editor” on page 3-4
- “Using Build Scripts” on page 3-5
- “Check Code Using the MATLAB Code Analyzer” on page 3-6
- “Separating Your Test Bench from Your Function Code” on page 3-7
- “Preserving Your Code” on page 3-8
- “File Naming Conventions” on page 3-9

Recommended Compilation Options for codegen

In this section...
“-c Generate Code Only” on page 3-2
“-report Generate Code Generation Report” on page 3-2

-c Generate Code Only

Use the `-c` option to generate code only without invoking the `make` command. If this option is used, `codegen` does not generate compiled object code. This option saves you time during the development cycle when you want to iterate rapidly between modifying MATLAB code and generating C code and are mainly interested in inspecting the C code.

For more information and a complete list of compilation options, see `codegen`.

-report Generate Code Generation Report

Use the `-report` option to generate a code generation report in HTML format at compile time to help you debug your MATLAB code and verify that it is suitable for code generation. If the `-report` option is not specified, `codegen` generates a report only if compilation errors or warnings occur.

The code generation report contains the following information:

- Summary of compilation results, including type of target and number of warnings or errors
- Target build log that records compilation and linking activities
- Links to generated files
- Error and warning messages

For more information, see `codegen`.

Testing MEX Functions in MATLAB

To prepare your MATLAB code before you generate C code, use `codegen` to convert your MATLAB code to a MEX function. `codegen` generates a platform-specific MEX-file, which you can execute within the MATLAB environment to test your algorithm.


For more information, see `codegen`.

Comparing C Code and MATLAB Code Using Tiling in the MATLAB Editor

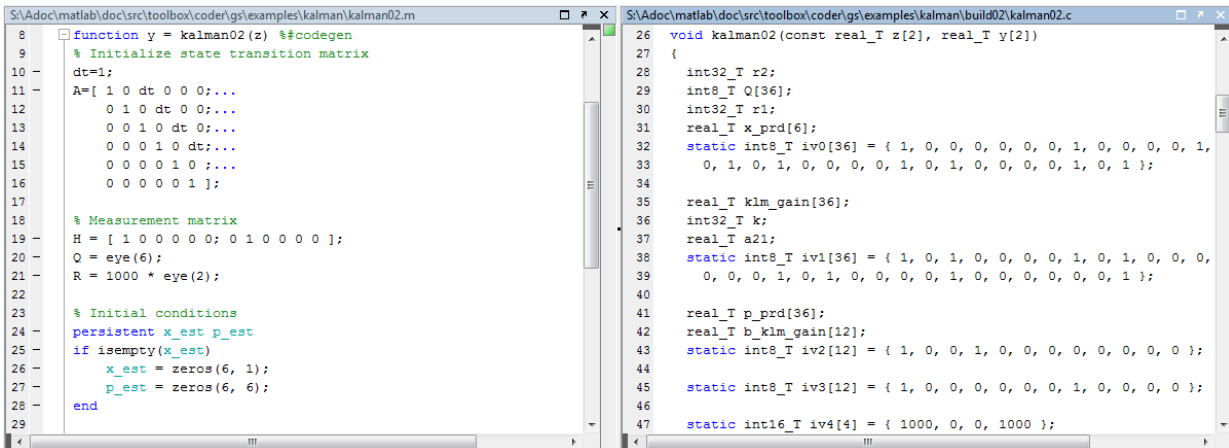
Use the MATLAB Editor's left/right tile feature to compare your generated C code to the original MATLAB code. You can easily compare the generated C code to your original MATLAB code. In the generated C code:

- Your function name is unchanged.
- Your comments are preserved in the same position.

To compare two files, follow these steps:

- 1 Open the C file and the MATLAB file in the Editor. (Dock both windows if they are not docked.)
- 2 Select **Window > Left/Right Tile** (or the  toolbar button) to view the files side by side.

The MATLAB file `kalman02.m` and its generated C code `kalman02.c` are displayed in the following figure.



```

8 function y = kalman02(z) %#codegen
9     % Initialize state transition matrix
10    dt=1;
11    A=[ 1 0 dt 0 0 0;...
12        0 1 0 dt 0 0;...
13        0 0 1 0 dt 0;...
14        0 0 0 1 0 dt;...
15        0 0 0 0 1 0 ;...
16        0 0 0 0 0 1 ];
17
18    % Measurement matrix
19    H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];
20    Q = eye(6);
21    R = 1000 * eye(2);
22
23    % Initial conditions
24    persistent x_est p_est
25    if isempty(x_est)
26        x_est = zeros(6, 1);
27        p_est = zeros(6, 6);
28    end
29
26 void kalman02(const real_T z[2], real_T y[2])
27 {
28     int32_T r2;
29     int8_T Q[36];
30     int32_T r1;
31     real_T x_prd[6];
32     static int8_T iv0[36] = { 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
33         0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1 };
34
35     real_T klm_gain[36];
36     int32_T k;
37     real_T a21;
38     static int8_T iv1[36] = { 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
39         0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1 };
40
41     real_T p_prd[36];
42     real_T b_klm_gain[12];
43     static int8_T iv2[12] = { 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 };
44
45     static int8_T iv3[12] = { 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 };
46
47     static int16_T iv4[4] = { 1000, 0, 0, 1000 };

```

Using Build Scripts

If you use `codegen` to generate code from the command line, use build scripts to call `codegen` to generate MEX functions from your MATLAB function.

A build script automates a series of MATLAB commands that you want to perform repeatedly from the command line, saving you time and eliminating input errors. For instance, you can use a build script to clear your workspace before each build and to specify code generation options.

Here is an example of a build script to run `codegen` to process `lms_02.m`:

```
close all;  
clear all;  
clc;
```

```
N = 73113;
```

```
codegen -report lms_02.m ...  
-args { zeros(N,1) zeros(N,1) }
```

where:

- `close all` deletes figures whose handles are not hidden. See `close` in the MATLAB Graphics function reference for more information.
- `clear all` removes variables, functions, and MEX-files from memory, leaving the workspace empty. It also clears breakpoints.

Note: Remove the `clear all` command from the build scripts if you want to preserve breakpoints for debugging.

- `clc` clears all input and output from the Command Window display, giving you a “clean screen.”
- `N = 73113` sets the value of the variable `N`, which represents the number of samples in each of the two input parameters for the function `lms_02`
- `codegen -report lms_02.m -args { zeros(N,1) zeros(N,1) }` calls `codegen` to generate C code for file `lms_02.m` using the following options:
 - `-report` generates a code generation report
 - `-args { zeros(N,1) zeros(N,1) }` specifies the properties of the function inputs as a cell array of example values. In this case, the input parameters are `N`-by-1 vectors of real doubles.

Check Code Using the MATLAB Code Analyzer

The code analyzer checks your code for problems and recommends modifications. You can use the code analyzer to check your code interactively in the MATLAB Editor while you work.

To verify that continuous code checking is enabled:

- 1** In MATLAB, select the **Home** tab and then click **Preferences**.
- 2** In the **Preferences** dialog box, select **Code Analyzer**.
- 3** In the **Code Analyzer Preferences** pane, verify that **Enable integrated warning and error messages** is selected.

Separating Your Test Bench from Your Function Code

If you use `codegen` to generate code from the command line, separate your core algorithm from your test bench. Create a separate test script to do the pre- and post-processing such as loading inputs, setting up input values, calling the function under test, and outputting test results.

Preserving Your Code

Preserve your code before making further modifications. This practice provides a fallback in case of error and a baseline for testing and validation. Use a consistent file naming convention. For example, add a 2-digit suffix to the file name for each file in a sequence. See “File Naming Conventions” on page 3-9 for more details.

File Naming Conventions

Use a consistent file naming convention to identify different types and versions of your MATLAB files. This approach keeps your files organized and minimizes the risk of overwriting existing files or creating two files with the same name in different folders.

For example, the file naming convention in the Generating MEX Functions getting started tutorial is:

- The suffix `_build` identifies a build script.
- The suffix `_test` identifies a test script.
- A numerical suffix, for example, `_01` identifies the version of a file. These numbers are typically two-digit sequential integers, beginning with 01, 02, 03, and so on.

For example:

- The file `build_01.m` is the first version of the build script for this tutorial.
- The file `test_03.m` is the third version of the test script for this tutorial.

