

MATLAB® Coder™
Getting Started Guide



MATLAB®

R2023a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

MATLAB® Coder™ Getting Started Guide

© COPYRIGHT 2011–2023 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)
March 2015	Online only	Revised for Version 2.8 (Release 2015a)
September 2015	Online only	Revised for Version 3.0 (Release 2015b)
October 2015	Online only	Rereleased for Version 2.8.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 3.1 (Release 2016a)
September 2016	Online only	Revised for Version 3.2 (Release 2016b)
March 2017	Online only	Revised for Version 3.3 (Release 2017a)
September 2017	Online only	Revised for Version 3.4 (Release 2017b)
March 2018	Online only	Revised for Version 4.0 (Release 2018a)
September 2018	Online only	Revised for Version 4.1 (Release 2018b)
March 2019	Online only	Revised for Version 4.2 (Release 2019a)
September 2019	Online only	Revised for Version 4.3 (Release 2019b)
March 2020	Online only	Revised for Version 5.0 (Release 2020a)
September 2020	Online only	Revised for Version 5.1 (Release 2020b)
March 2021	Online only	Revised for Version 5.2 (Release 2021a)
September 2021	Online only	Revised for Version 5.3 (Release 2021b)
March 2022	Online only	Revised for Version 5.4 (Release R2022a)
September 2022	Online only	Revised for Version 5.5 (Release R2022b)
March 2023	Online only	Revised for Version 5.6 (Release R2023a)

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/. In the search bar, type the phrase "Incorrect Code Generation" to obtain a report of known bugs that produce code that might compile and execute, but still produce wrong answers. To save a search, click Save Search.

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.

1	Product Overview	
	MATLAB Coder Product Description	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	1-4
	Code Generation for Fixed-Point Algorithms	1-5
	Installing Prerequisite Products	1-6
	Related Products	1-7
	Setting Up the C or C++ Compiler	1-8
	Expected Background	1-9
	Code Generation Workflow	1-10
	See Also	1-10
	Input Type Specification for Code Generation	1-11
	Differences in Appearance of Generated Code and MATLAB Code	1-14
	Mapping MATLAB Functions to C/C++ Functions	1-14
	Representation of Function Outputs	1-14
	Constant Values Removed in Generated Code	1-15
	Accessing Matrix Elements	1-16
	Math Operations and Other Function Calls	1-16
	Variable-Size Arrays	1-16
	Local Variables in Generated Code	1-17
	Cell Arrays in Generated Code	1-17
	Initialize and Terminate Functions	1-17

2	Tutorials	
	Generate C Code by Using the MATLAB Coder App	2-2
	Tutorial Files: Euclidean Distance	2-2

Description of Tutorial Files	2-2
Generate C Code for the MATLAB Function	2-4
Generate C Code for Variable-Size Inputs	2-14
Next Steps	2-15
Generate C Code at the Command Line	2-17
Tutorial Files: Euclidean Distance	2-17
Description of Tutorial Files	2-17
Generate C Code for the MATLAB Function	2-19
Generate C Code for Variable-Size Inputs	2-23
Next Steps	2-24
Accelerate MATLAB Algorithm by Generating MEX Function	2-26
Tutorial Files: Euclidean Distance	2-26
Description of Tutorial Files	2-26
Generate MEX Function for the MATLAB Function	2-28
Generate MEX Function for Variable-Size Inputs	2-31
Next Steps	2-33
Hello World	2-34
Generate Code for an Averaging Filter	2-35
Code Generation Guide: Generate Deployable C/C++ Code	2-40
Prepare MATLAB Code for Code Generation	2-43
Initialize Variables for Code Generation	2-43
Screen Code for Unsupported Functions and Language Features	2-43
Tips	2-44
Generate C/C++ Code from MATLAB Code	2-45
Specify Input Types	2-45
Check for Run-Time Issues	2-45
Configure Code Generation Build Settings	2-45
Generate Standalone C/C++ Code	2-45
Understand Generated Code	2-46
Tips	2-47
Test Generated C/C++ Code	2-49
Test MEX Code to Verify Behavior	2-49
Test Standalone Code by Using Software-in-the-Loop and Processor-in-the-Loop	2-49
Tips	2-49
Deploy Generated C/C++ Code	2-51
Edit Generated Main Function and Interfaces	2-51
Build Executable Applications by Using MATLAB Coder	2-52
Target Specific Code Generation	2-52
Tips	2-53

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 1-4
- “Code Generation for Fixed-Point Algorithms” on page 1-5
- “Installing Prerequisite Products” on page 1-6
- “Related Products” on page 1-7
- “Setting Up the C or C++ Compiler” on page 1-8
- “Expected Background” on page 1-9
- “Code Generation Workflow” on page 1-10
- “Input Type Specification for Code Generation” on page 1-11
- “Differences in Appearance of Generated Code and MATLAB Code” on page 1-14

MATLAB Coder Product Description

Generate C and C++ code from MATLAB code

MATLAB Coder generates C and C++ code from MATLAB code for a variety of hardware platforms, from desktop systems to embedded hardware. It supports most of the MATLAB language and a wide range of toolboxes. You can integrate the generated code into your projects as source code, static libraries, or dynamic libraries. The generated code is readable and portable. You can combine it with key parts of your existing C and C++ code and libraries. You can also package the generated code as a MEX-function for use in MATLAB.

When used with Embedded Coder[®], MATLAB Coder provides code customizations, target-specific optimizations, code traceability, and software-in-the-loop (SIL) and processor-in-the-loop (PIL) verification.

To deploy MATLAB programs as standalone applications, use MATLAB Compiler[™]. To generate software components for integration with other programming languages, use MATLAB Compiler SDK[™].

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

- “Set Up a MATLAB Coder Project”
- “Generate C Code by Using the MATLAB Coder App” on page 2-2

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
- “Generate C Code at the Command Line” on page 2-17
- “Accelerate MATLAB Algorithm by Generating MEX Function” on page 2-26

Code Generation for Embedded Software Applications

The Embedded Coder product extends the MATLAB Coder product with features that you can use for embedded software development. With the Embedded Coder 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 executes efficiently for embedded systems.
- Customize the appearance of the generated code.
- Optimize generated code for a specific target environment.
- Integrate existing applications, functions, and data.
- Enable tracing, reporting, and testing options that facilitate code verification activities.

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

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® website.

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

To use MATLAB Coder to generate code for deep learning networks, you must also install:

- Deep Learning Toolbox™
- MATLAB Coder Interface for Deep Learning

To generate code for deep learning networks, additional third-party libraries and setup steps might be necessary. See “Prerequisites for Deep Learning with MATLAB Coder”.

The MATLAB Coder Interface for Deep Learning is not supported for MATLAB Online™.

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 website.

You can use `mex -setup` to change the default compiler. See “Change Default Compiler”. If you generate C++ code, see “Choose a C++ Compiler”.

Expected Background

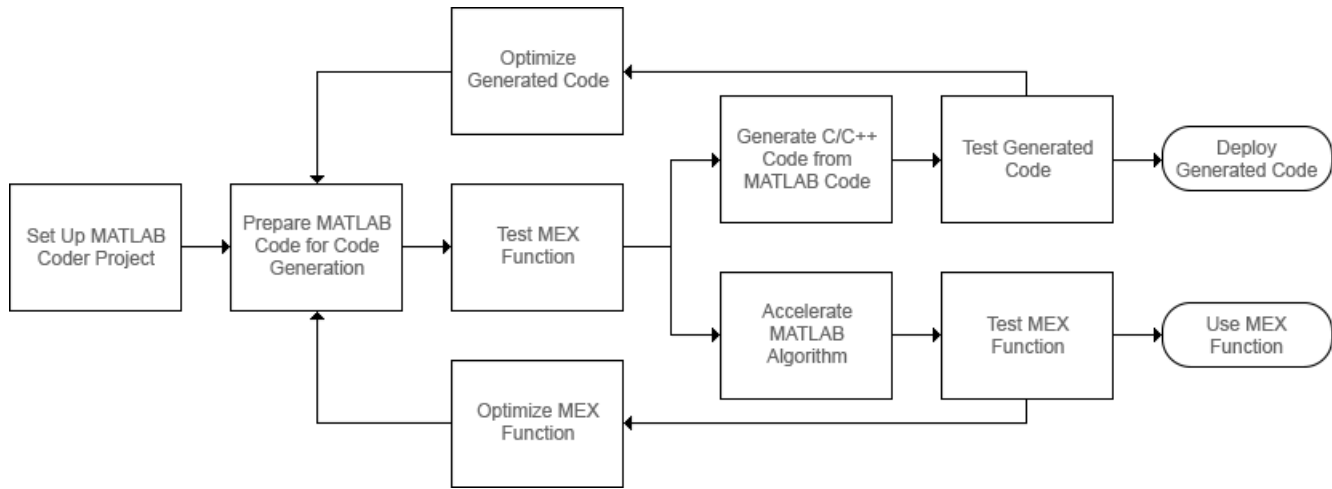
You should be familiar with :

- MATLAB software
- MEX functions
- 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.

Code Generation Workflow



See Also

- “Set Up a MATLAB Coder Project”
- “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”

Input Type Specification for Code Generation

C/C++ and MATLAB handle variables differently. Some of these differences that affect the code generation workflow are:

- C/C++ source code includes type declarations for all variables. The C/C++ compiler uses these declarations to determine the types of all variables at compile time. MATLAB code does not include explicit type declarations. The MATLAB execution engine determines the types of variables at run time.
- In C/C++, the memory for arrays can be either statically declared at compile time (fixed size arrays), or dynamically allocated at run time (variable-size arrays). All MATLAB arrays use dynamically allocated memory and are of variable size.

To allow the generation of C/C++ code with specific types, you must specify the properties (class, size, and complexity) of all input variables to the MATLAB entry-point functions during C/C++ or MEX code generation. An entry-point function is a top-level MATLAB function from which you generate code. The code generator uses these input properties to determine the properties of all variables in the generated code. Different input type specifications can cause the same MATLAB code to produce different versions of the generated code.

If you generate code by using the `codegen` command, you use the `-args` option to specify the input types. If you generate code by using the MATLAB Coder app, you specify the input types in the **Define Input Types** page.

To see how input type specification affects the generated code, consider a simple MATLAB function `myMultiply` that multiplies two quantities `a` and `b` and returns the value of the product.

```
function y = myMultiply(a,b)
y = a*b;
end
```

Generate static C library code for three different type specifications for the input arguments `a` and `b`. In each case, inspect the generated code.

- Specify `a` and `b` as real double scalars. To generate code for these inputs, run these commands:

```
a = 1;
codegen -config:lib myMultiply -args {a,a}
```

The generated C source file `myMultiply.c` contains the C function:

```
double myMultiply(double a, double b)
{
    return a * b;
}
```

- Specify `a` and `b` as real double 5-by-5 matrices. To generate code for these inputs, run these commands:

```
a = zeros(5,5);
codegen -config:lib myMultiply -args {a,a}
```

The generated C source file `myMultiply.c` contains the C function:

```
void myMultiply(const double a[25], const double b[25], double y[25])
{
```

```
int i;
int i1;
double d;
int i2;
for (i = 0; i < 5; i++) {
    for (i1 = 0; i1 < 5; i1++) {
        d = 0.0;
        for (i2 = 0; i2 < 5; i2++) {
            d += a[i + 5 * i2] * b[i2 + 5 * i1];
        }
        y[i + 5 * i1] = d;
    }
}
```

`const double a[25]` and `const double b[25]` correspond to the inputs `a` and `b` in the MATLAB code. The size of the one-dimensional arrays `a` and `b` in the C code is 25, which is equal to the total number of elements in example input arrays that you used when you called the `codegen` function.

The C function has one more argument: the one-dimensional array `y` of size 25. It uses this array to return the output of the function.

You can also generate code that has the same array dimensions as the MATLAB code. See “Generate Code That Uses N-Dimensional Indexing”.

- Finally, you generate code for `myMultiply` that can accept input arrays of many different sizes. To specify variable-size inputs, you can use the `coder.typeof` function. `coder.typeof(A,B,1)` specifies a variable-size input with the same class and complexity as `A` and upper bounds given by the corresponding element of the size vector `B`.

Specify `a` and `b` as real double arrays that are of variable-size, with a maximum size of 10 on either dimension. To generate code, run these commands:

```
a = coder.typeof(1,[10 10],1);
codegen -config:lib myMultiply -args {a,a}
```

The signature of the generated C function is:

```
void myMultiply(const double a_data[], const int a_size[2], const double b_data[],
               const int b_size[2], double y_data[], int y_size[2])
```

The arguments `a_data`, `b_data`, and `y_data` correspond to the input arguments `a` and `b` and the output argument `y` in the original MATLAB function. The C function now accepts three additional arguments, `a_size`, `b_size`, and `y_size`, that specify the sizes of `a_data`, `b_data`, and `y_data` at run time.

See Also

`codegen` | `coder.typeof`

More About

- “Specify Properties of Entry-Point Function Inputs”
- “Generate C Code by Using the MATLAB Coder App” on page 2-2

- “Generate C Code at the Command Line” on page 2-17
- “Generate Code That Uses N-Dimensional Indexing”

Differences in Appearance of Generated Code and MATLAB Code

MATLAB Coder translates and optimizes dynamically typed MATLAB code into statically typed C/C++. Statically typed languages require variable types to be explicitly declared and these types are determined at compile time. Certain changes and optimizations performed by the code generator enable the use of MATLAB data types and features in the generated code. For more information on data types and features supported for code generation, see “Data Definition” and “MATLAB Language Features Supported for C/C++ Code Generation”.

These changes and optimizations cause the generated code to appear differently than the MATLAB code. The generated code might not map in a one-to-one manner with your MATLAB code due to any of the following:

- “Mapping MATLAB Functions to C/C++ Functions” on page 1-14
- “Representation of Function Outputs” on page 1-14
- “Constant Values Removed in Generated Code” on page 1-15
- “Accessing Matrix Elements” on page 1-16
- “Math Operations and Other Function Calls” on page 1-16
- “Variable-Size Arrays” on page 1-16
- “Local Variables in Generated Code” on page 1-17
- “Cell Arrays in Generated Code” on page 1-17
- “Initialize and Terminate Functions” on page 1-17

Note Depending on your source code, these cases might occur slightly differently than how they are shown here.

Mapping MATLAB Functions to C/C++ Functions

MATLAB Coder generates standalone C/C++ code and MEX code from MATLAB code. A single function in your MATLAB code might be translated into multiple functions in the generated code. Two or more functions in your MATLAB code might also become one function body in the generated code. This process is called function inlining. By default, the code generator uses internal heuristics to determine whether to inline your functions or not. For more information, see `coder.inline` and “Control Inlining to Fine-Tune Performance and Readability of Generated Code”.

Representation of Function Outputs

Outputs of a MATLAB function might become return values in C or might become pass-by-reference inputs. . One scalar output in your MATLAB code is treated as a return value in the generated code.

- The function `addOne` has an input variable `x` and output variable `y`. For this example, `x` is of type `double`.

```
function y = addOne(x)
y = x + 1;
end
```


The code generated for the snippet is shown here:

```
double addOne(double x)
{
    return x + 1.0;
}
```

The input to the function `addOne`, `x`, is treated as a pass-by-value variable in the generated code. The output of the MATLAB function is returned by value in the generated code.

- For arrays, outputs might be passed-by-reference. The code snippet shown here uses a `double` input `x` and an array output `y`.

```
function y = addMat(x)
z = [1:100];
y = z + x;
end
```

The output variable `y` is translated into a pass-by-reference array variable in the generated code shown here:

```
void addMat(double x, double y[100])
{
    int i;
    for (i = 0; i < 100; i++) {
        y[i] = ((double)i + 1.0) + x;
    }
}
```

- For entry-point functions that have multiple output variables, outputs might be passed-by-reference in the generated code. This code snippet has two `double` scalar outputs, `y` and `z`, with a `double` scalar input `x`.

```
function [z,y] = splitOne(x)
y = x + 1;
z = x + 2;
end
```

The output variables `y` and `z` are translated into a pass-by-reference variables in the generated code:

```
void splitOne(double x, double *z, double *y)
{
    *y = x + 1.0;
    *z = x + 2.0;
}
```

For more information on argument passing behavior of entry-point functions in the generated code, see “Deploy Generated Code”.

Constant Values Removed in Generated Code

Constant values in your code might not be preserved in generated code. These values might get removed to optimize the generated code. Constant folding removes the computations that might have been present in your MATLAB code and replaces the computations with the result instead. For more information, see “Constant Folding”.

Consider this code snippet:

```
function y = removeConst
x = ones(10);
y = x + 1;
end
```

The code generator removes the constant matrix `x` to save memory and assigns the constant value as the result. The generated code looks like this code:

```
void removeConst(double y[100])
{
    int i;
    for (i = 0; i < 100; i++) {
        y[i] = 2.0;
    }
}
```

Unused inputs or constant inputs to nonentry-point functions in your MATLAB code are removed from the function bodies in the generated code.

Function specializations by the code generator can change the function to a version where the input type, size, complexity, or value might be customized for a particular invocation of the function. This is done to produce efficient C code at the expense of code duplication. For more information, see “Specialized Functions or Classes”.

Accessing Matrix Elements

In the preceding instance, accessing a matrix requires extra lines of code in C/C++. A 10-by-10 matrix is represented as an array of 100 `double` elements in the generated code. A `for` loop is used to access all the array elements in this case. C/C++ do not support many matrix operations, so the code generator converts the matrices and the operations on matrices to arrays and methods like `for` loops to access those arrays.

Math Operations and Other Function Calls

The generated code might use standard C libraries to carry out the math operations or other functions in your MATLAB code. For a list of supported language functions, see “MATLAB Language Features Supported for C/C++ Code Generation”.

Variable-Size Arrays

For code generation, an array can be fixed-size or variable-size. Variable-size arrays might appear in different formats in the generated code. Code can be generated for a fixed-size array if the code generator can determine the size of the array. Code generation is also valid for a fixed-size array with an upper bound. Dynamically allocated arrays are also generated in certain cases. See “Code Generation for Variable-Size Arrays”.

Code generation for fixed-size and variable-size arrays might yield the following variable declarations in the generated code:

```
double x[10];           // Fixed-size array
double y_data[20];
int y_size[2];         // y_data and y_size denote an upper-bounded array
mxArrayReal_T *z;     // Dynamically allocated array
```

Local Variables in Generated Code

If your MATLAB code contains local variables that occupy a lot of memory, then in the generated code they might be declared as local variables, static local variables, or as variables in a `struct` that is passed into your entry-point function in your generated code. You can control this transformation by controlling the memory that is allocated for the generated code. See “Control Stack Space Usage”.

Cell Arrays in Generated Code

To implement cell arrays in generated code, the code generator might translate them as a `struct`, static array, or dynamic array. For more information, see “Code Generation for Cell Arrays”.

Initialize and Terminate Functions

The code generator might produce two housekeeping functions, `initialize` and `terminate`, if they are needed. You can find these functions in the **Generated Code** tab in the code generation report. The `initialize` function initializes the state on which the generated C/C++ entry-point functions operate. The `terminate` function frees allocated memory and performs other cleanup operations. For more information, see “Use Generated Initialize and Terminate Functions”

See Also

`coder.inline`

More About

- “Generate C Code by Using the MATLAB Coder App” on page 2-2
- “Generate C Code at the Command Line” on page 2-17
- “Language, Function, and Object Support”
- “Optimization Strategies”

Tutorials

- “Generate C Code by Using the MATLAB Coder App” on page 2-2
- “Generate C Code at the Command Line” on page 2-17
- “Accelerate MATLAB Algorithm by Generating MEX Function” on page 2-26
- “Hello World” on page 2-34
- “Generate Code for an Averaging Filter” on page 2-35
- “Code Generation Guide: Generate Deployable C/C++ Code” on page 2-40
- “Prepare MATLAB Code for Code Generation” on page 2-43
- “Generate C/C++ Code from MATLAB Code” on page 2-45
- “Test Generated C/C++ Code” on page 2-49
- “Deploy Generated C/C++ Code” on page 2-51

Generate C Code by Using the MATLAB Coder App

In this tutorial, you use the MATLAB Coder app to generate a static C library for a MATLAB function. You first generate C code that can accept only inputs that have fixed preassigned size. You then generate C code that can accept inputs of many different sizes.

You can also generate code at the MATLAB command line by using the `codegen` command. For a tutorial on this workflow, see “Generate C Code at the Command Line” on page 2-17.

The MATLAB Coder app is not supported in MATLAB Online. To generate C/C++ code in MATLAB Online, use the `codegen` command.

Tutorial Files: Euclidean Distance

Open this example to obtain the files for this tutorial.

Description of Tutorial Files

This tutorial uses the `euclidean_data.mat`, `euclidean.m`, and `test.m` files.

- The MATLAB data file `euclidean_data.mat` contains two pieces of data: a single point in three-dimensional Euclidean space and a set of several other points in three-dimensional Euclidean space. More specifically:
 - `x` is a 3-by-1 column vector that represents a point in three-dimensional Euclidean space.
 - `cb` is a 3-by-216 array. Each column in `cb` represents a point in three-dimensional Euclidean space.
- The MATLAB file `euclidean.m` contains the function `euclidean` that implements the *core algorithm* in this example. The function takes `x` and `cb` as inputs. It calculates the Euclidean distance between `x` and each point in `cb` and returns these quantities:
 - The column vector `y_min`, which is equal to the column in `cb` that represents the point that is closest to `x`.
 - The column vector `y_max`, which is equal to the column in `cb` that represents the point that is farthest from `x`.
 - The 2-dimensional vector `idx` that contains the column indices of the vectors `y_min` and `y_max` in `cb`.
 - The 2-dimensional vector `distance` that contains the calculated smallest and largest distances to `x`.

```
function [y_min,y_max,idx,distance] = euclidean(x,cb)
% Initialize minimum distance as distance to first element of cb
% Initialize maximum distance as distance to first element of cb
idx(1)=1;
idx(2)=1;

distance(1)=norm(x-cb(:,1));
distance(2)=norm(x-cb(:,1));

% Find the vector in cb with minimum distance to x
```

```

% Find the vector in cb with maximum distance to x
for index=2:size(cb,2)
    d=norm(x-cb(:,index));
    if d < distance(1)
        distance(1)=d;
        idx(1)=index;
    end
    if d > distance(2)
        distance(2)=d;
        idx(2)=index;
    end
end

% Output the minimum and maximum distance vectors
y_min=cb(:,idx(1));
y_max=cb(:,idx(2));

end

```

- The MATLAB script `test.m` loads the data file `euclidean_data.mat` into the workspace. It then calls the function `euclidean` to calculate `y_min`, `y_max`, `idx`, and `distance`. The script then displays the calculated quantities at the command line.

Loading `euclidean_data.mat` is the preprocessing step that is executed before calling the core algorithm. Displaying the results is the post-processing step.

```

% Load test data
load euclidean_data.mat

% Determine closest and farthest points and corresponding distances
[y_min,y_max,idx,distance] = euclidean(x,cb);

% Display output for the closest point
disp('Coordinates of the closest point are: ');
disp(num2str(y_min));
disp(['Index of the closest point is ', num2str(idx(1))]);
disp(['Distance to the closest point is ', num2str(distance(1))]);

disp(newline);

% Display output for the farthest point
disp('Coordinates of the farthest point are: ');
disp(num2str(y_max));
disp(['Index of the farthest point is ', num2str(idx(2))]);
disp(['Distance to the farthest point is ', num2str(distance(2))]);

```

Tip You can generate code from MATLAB functions by using MATLAB Coder. Code generation from MATLAB scripts is not supported.

Use test scripts to separate the pre- and post-processing steps from the function implementing the core algorithm. This practice enables you to easily reuse your algorithm. You generate code for the MATLAB function that implements the core algorithm. You do not generate code for the test script.

Generate C Code for the MATLAB Function

Run the Original MATLAB Code

Run the test script `test.m` in MATLAB. The output displays `y`, `idx`, and `distance`.

```
Coordinates of the closest point are:
0.8      0.8      0.4
Index of the closest point is 171
Distance to the closest point is 0.080374
```

```
Coordinates of the farthest point are:
0 0 1
Index of the farthest point is 6
Distance to the farthest point is 1.2923
```

Make the MATLAB Code Suitable for Code Generation

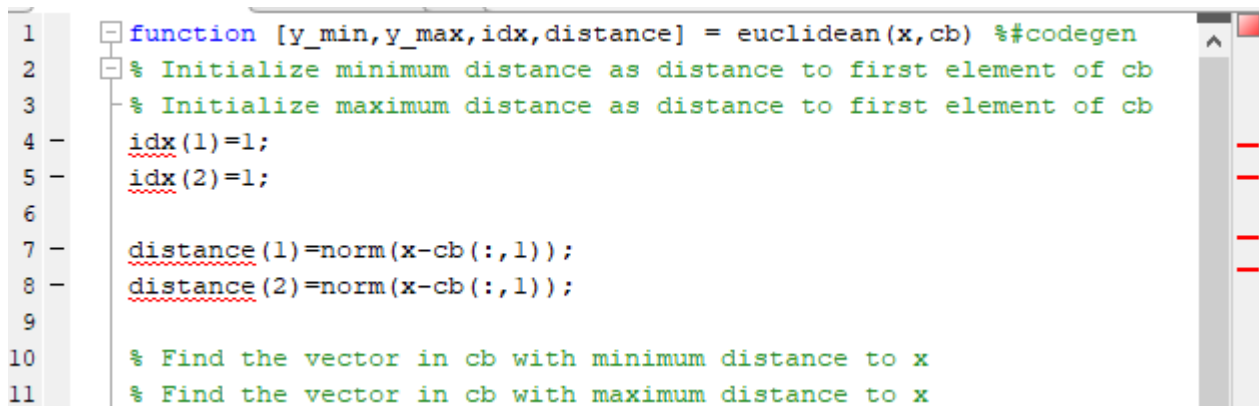
The Code Analyzer in the MATLAB Editor continuously checks your code as you enter it. It reports issues and recommends modifications to maximize performance and maintainability.

- 1 Open `euclidean.m` in the MATLAB Editor. The Code Analyzer message indicator in the top right corner of the MATLAB Editor is green. The analyzer did not detect errors, warnings, or opportunities for improvement in the code.
- 2 After the function declaration, add the `%#codegen` directive:

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

The `%#codegen` directive prompts the Code Analyzer to identify warnings and errors specific to code generation.

The Code Analyzer message indicator becomes red, indicating that it has detected code generation issues.



```

1 function [y_min,y_max,idx,distance] = euclidean(x,cb) %#codegen
2 % Initialize minimum distance as distance to first element of cb
3 % Initialize maximum distance as distance to first element of cb
4 idx(1)=1;
5 idx(2)=1;
6
7 distance(1)=norm(x-cb(:,1));
8 distance(2)=norm(x-cb(:,1));
9
10 % Find the vector in cb with minimum distance to x
11 % Find the vector in cb with maximum distance to x

```

- 3 To view the warning messages, move your cursor to the underlined code fragments. The warnings indicate that code generation requires the variables `idx` and `distance` to be fully defined before subscripting them. These warnings appear because the code generator must determine the sizes of these variables at their first appearance in the code. To fix this issue, use the `ones` function to simultaneously allocate and initialize these arrays.

```

% Initialize minimum distance as distance to first element of cb
% Initialize maximum distance as distance to first element of cb

```



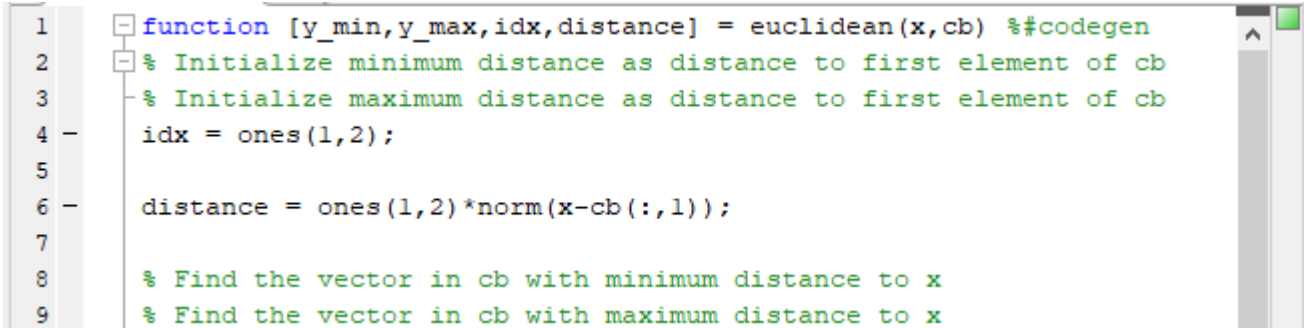
```

idx = ones(1,2);

distance = ones(1,2)*norm(x-cb(:,1));

```

The Code Analyzer message indicator becomes green again, indicating that it does not detect any more code generation issues.



```

1 function [y_min,y_max,idx,distance] = euclidean(x,cb) %#codegen
2 % Initialize minimum distance as distance to first element of cb
3 % Initialize maximum distance as distance to first element of cb
4 idx = ones(1,2);
5
6 distance = ones(1,2)*norm(x-cb(:,1));
7
8 % Find the vector in cb with minimum distance to x
9 % Find the vector in cb with maximum distance to x

```

For more information about using the Code Analyzer, see “Check Code for Errors and Warnings Using the Code Analyzer”.

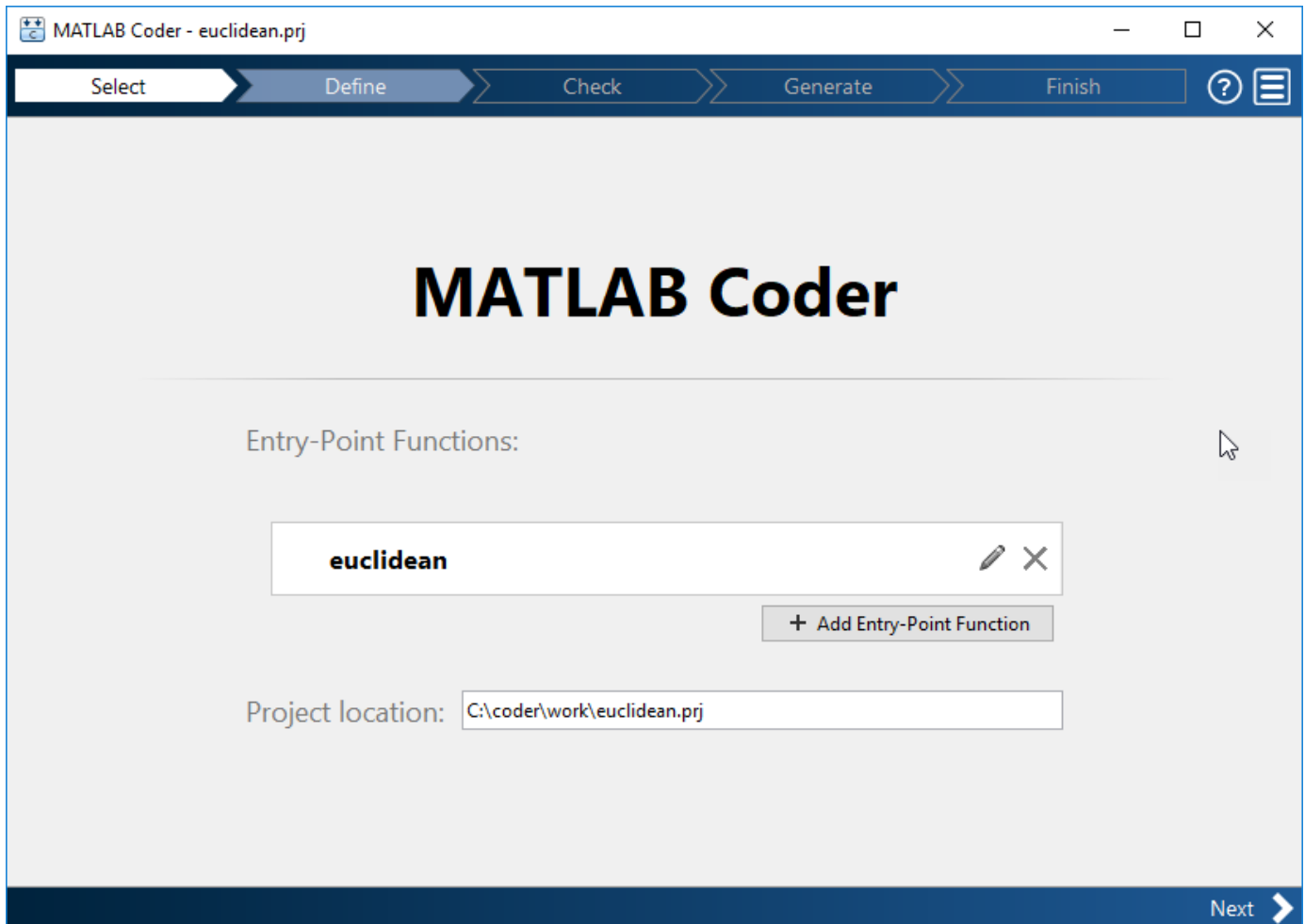
- 4 Save the file.

You are now ready to compile your code by using the MATLAB Coder app. Here, compilation refers to the generation of C/C++ code from your MATLAB code.

Note Compilation of MATLAB code refers to the generation of C/C++ code from the MATLAB code. In other contexts, the term compilation could refer to the action of a C/C++ compiler.

Open the MATLAB Coder App and Select Source Files

- 1 On the MATLAB toolstrip **Apps** tab, under **Code Generation**, click the MATLAB Coder app icon. The app opens the **Select Source Files** page.
- 2 In the **Select Source Files** page, enter or select the name of the entry-point function `euclidean`. An entry-point function is a top-level MATLAB function from which you generate code. The app creates a project with the default name `euclidean.prj` in the current folder.



- 3 Click **Next** to go to the **Define Input Types** step. The app runs the Code Analyzer (that you already ran in the previous step) and the Code Generation Readiness Tool on the entry-point function. The Code Generation Readiness Tool screens the MATLAB code for features and functions that are not supported for code generation. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page. For more information, see “Code Generation Readiness Tool”.

Note The Code Analyzer and the Code Generation Readiness Tool might not detect all code generation issues. After eliminating the errors or warnings that these two tools detect, generate code with MATLAB Coder to determine if your MATLAB code has other compliance issues.

Certain MATLAB built-in functions and toolbox functions, classes, and System objects that are supported for C/C++ code generation have specific code generation limitations. These limitations and related usage notes are listed in the **Extended Capabilities** sections of their corresponding reference pages. For more information, see “Functions and Objects Supported for C/C++ Code Generation”.

Define Input Types

Because C uses static typing, the code generator must determine the class, size, and complexity of all variables in the MATLAB files at code generation time, also known as *compile time*. Therefore, you must specify the properties of all entry-point function inputs. To specify input properties, you can:

- Instruct the app to automatically determine input properties by providing a script that calls the entry-point functions with sample inputs.
- Specify properties directly.

In this example, to define the properties of the inputs `x` and `cb`, specify the test file `test.m` that the code generator can use to define types automatically:

- 1 Enter or select the test file `test.m`.
- 2 Click **Autodefine Input Types**.

The test file, `test.m`, calls the entry-point function, `euclidean`, with the expected input types. The app determines that the input `x` is `double(3x1)` and the input `cb` is `double(3x216)`.

Define Input Types

To convert MATLAB to C, you must define the type of each input for every entry point function. [Learn more](#)

To **automatically define input types**, call `euclidean` or enter a script that calls `euclidean` in the MATLAB prompt below:

```
>> test
```

Autodefine Input Types

euclidean.m Number of outputs: 4

x	double(3 x 1)
cb	double(3 x 216)

Add global


Back Next

- 3 Click **Next** to go to the **Check for Run-Time Issues** step.

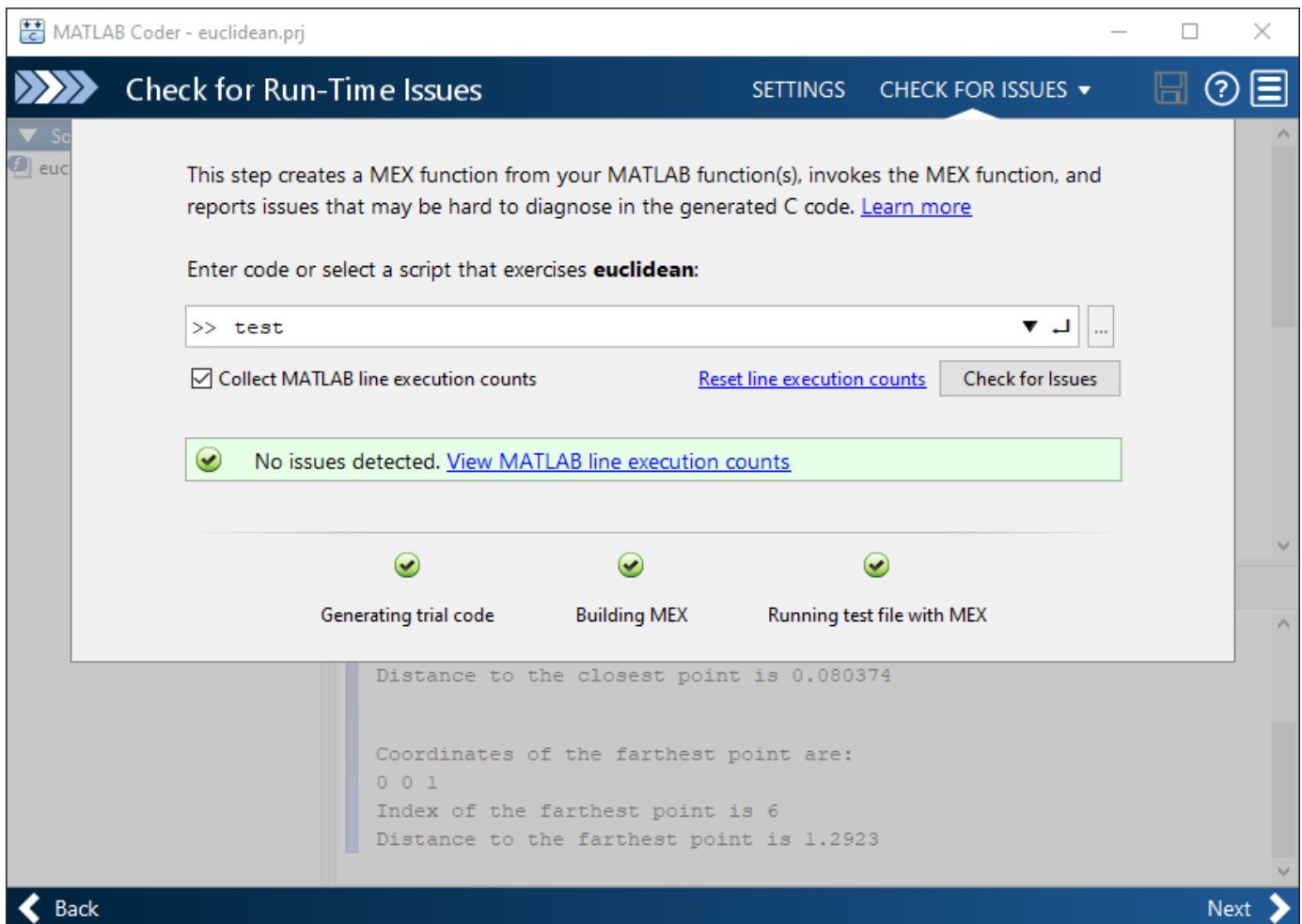
Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. A MEX function is generated code that can be called from inside MATLAB. It is a best practice to perform this step because you can detect and fix run-time errors that are harder to diagnose in the generated C code. By default, the MEX function includes memory integrity checks. These checks perform array bounds and dimension checking. The checks detect violations of memory integrity in code generated for MATLAB functions. For more information, see “Control Run-Time Checks”.

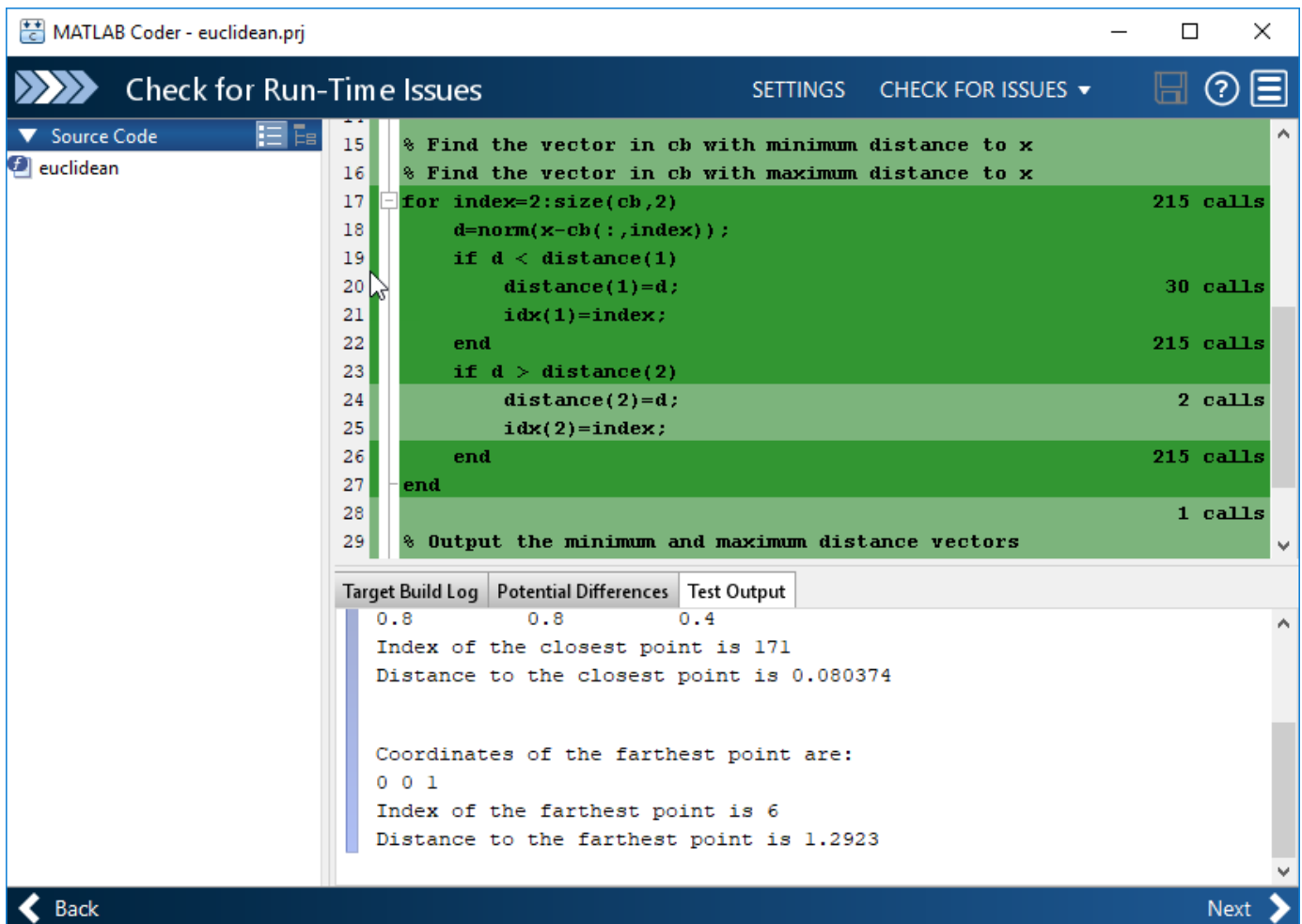
To convert MATLAB code to efficient C/C++ source code, the code generator introduces optimizations that, in certain situations, cause the generated code to behave differently than the original source code. See “Differences Between Generated Code and MATLAB Code”.

- 1** To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow .
- 2** In the **Check for Run-Time Issues** dialog box, specify a test file or enter code that calls the entry-point function with example inputs. For this example, use the test file `test` that you used to define the input types.
- 3** Click **Check for Issues**.

The app generates a MEX function. It runs the test script `test` replacing calls to `euclidean` with calls to the generated MEX. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. Click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues.



- 4 By default, the app collects line execution counts. These counts help you to see how well the test file `test.m` exercised the `euclidean` function. To view line execution counts, click **View MATLAB line execution counts**. The app editor displays a color-coded bar to the left of the code. To extend the color highlighting over the code and to see line execution counts, place your cursor over the bar.




A particular shade of green indicates that the line execution count for this code falls in a certain range. In this case, the `for`-loop executes 215 times. For information about how to interpret line execution counts and turn off collection of the counts, see “Collect and View Line Execution Counts for Your MATLAB Code”.

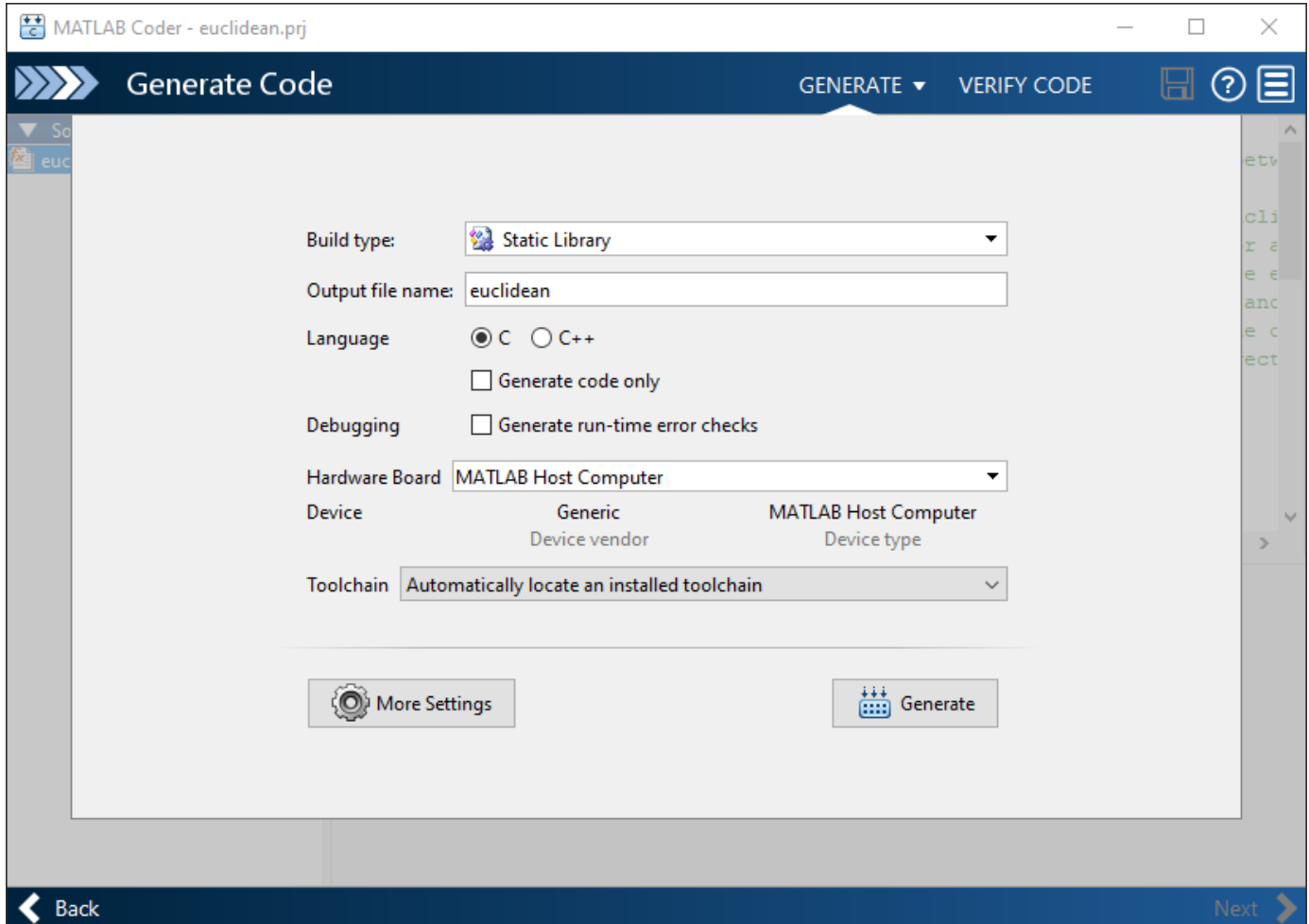
- 5 Click **Next** to go to the **Generate Code** step.

Note Before generating standalone C/C++ code from your MATLAB code, generate a MEX function. Run the generated MEX function and make sure it has the same run-time behavior as your MATLAB function. If the generated MEX function produces answers that are different from MATLAB, or produces an error, you must fix these issues before proceeding to standalone code generation. Otherwise, the standalone code that you generate might be unreliable and have undefined behavior.

Generate C Code

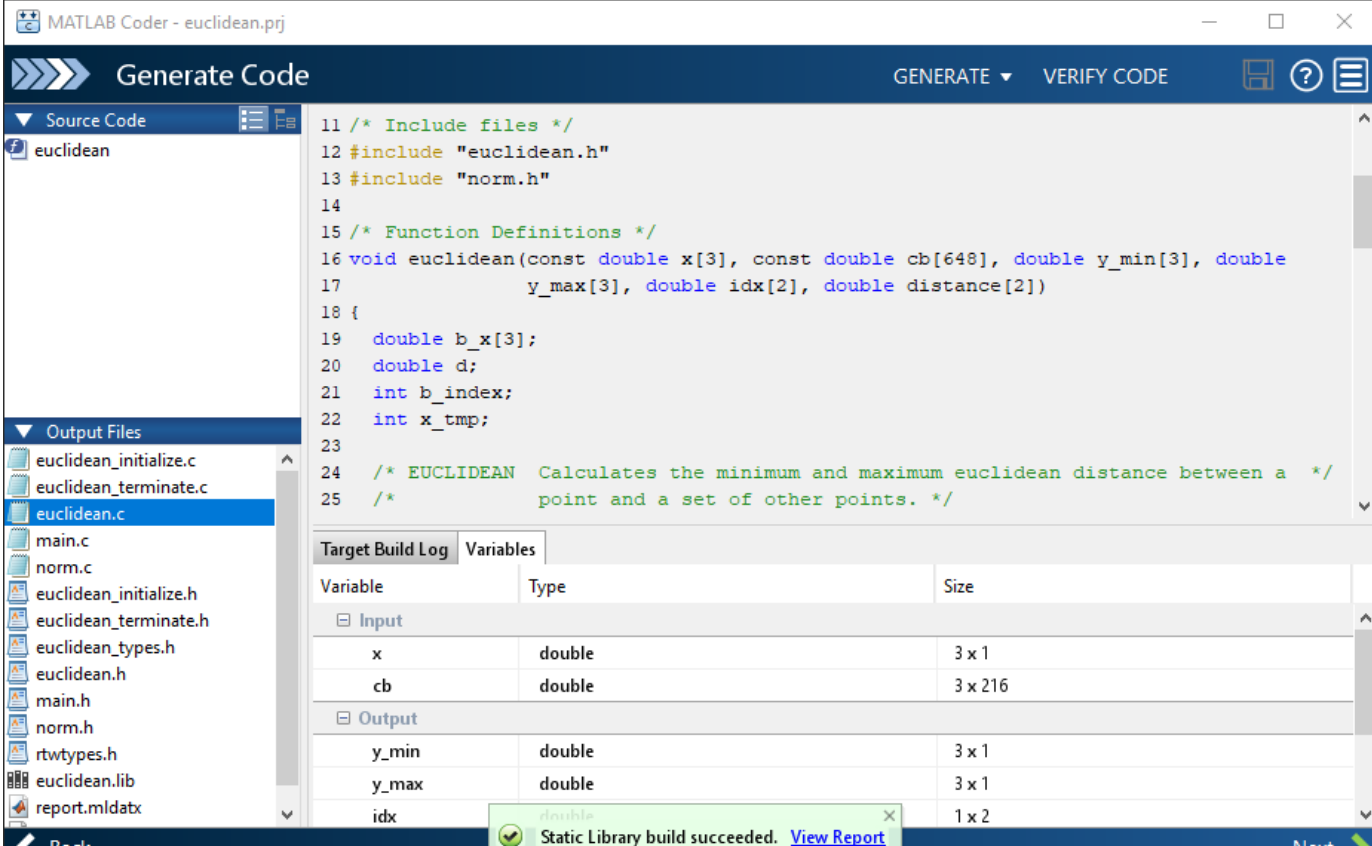
- 1 To open the **Generate** dialog box, click the **Generate** arrow .
- 2 In the **Generate** dialog box, set **Build type** to Static Library (.lib) and **Language** to C. Use the default values for the other project build configuration settings.

Instead of generating a C static library, you can choose to generate a MEX function or other C/C++ build types. Different project settings are available for the MEX and C/C++ build types. When you switch between MEX and C/C++ code generation, verify the settings that you choose.



3 Click **Generate**.

MATLAB Coder generates a standalone C static library `euclidean` in the `work\codegen\lib\euclidean`. `work` is the folder that contains your tutorial files. The MATLAB Coder app indicates that code generation succeeded. It displays the source MATLAB files and generated output files on the left side of the page. On the **Variables** tab, it displays information about the MATLAB source variables. On the **Target Build Log** tab, it displays the build log, including C/C++ compiler warnings and errors. By default, in the code window, the app displays the C source code file, `euclidean.c`. To view a different file, in the **Source Code** or **Output Files** pane, click the file name.



The screenshot shows the MATLAB Coder interface for a project named 'euclidean.pj'. The 'Generate Code' tab is active, displaying the C source code for the 'euclidean' function. The code includes headers for 'euclidean.h' and 'norm.h', and defines a function that calculates the minimum and maximum Euclidean distance between a point and a set of other points. The function signature is: `void euclidean(const double x[3], const double cb[648], double y_min[3], double y_max[3], double idx[2], double distance[2])`. The code defines variables for `b_x`, `d`, `b_index`, and `x_tmp`.

The 'Output Files' list on the left shows the generated files, including `euclidean_initialize.c`, `euclidean_terminate.c`, `euclidean.c`, `main.c`, `norm.c`, `euclidean_initialize.h`, `euclidean_terminate.h`, `euclidean_types.h`, `euclidean.h`, `main.h`, `norm.h`, `rtwtypes.h`, `euclidean.lib`, and `report.mldatx`.

The 'Target Build Log' table shows the following variables and their sizes:

Variable	Type	Size
Input		
<code>x</code>	double	3 x 1
<code>cb</code>	double	3 x 216
Output		
<code>y_min</code>	double	3 x 1
<code>y_max</code>	double	3 x 1
<code>idx</code>	double	1 x 2

A notification at the bottom of the interface states: 'Static Library build succeeded. [View Report](#)'.

- 4 Click **View Report** to view the report in the Report Viewer. If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code. For more information, see “Code Generation Reports”.
- 5 Click **Next** to open the **Finish Workflow** page.

Review the Finish Workflow Page

The **Finish Workflow** page indicates that code generation succeeded. It provides a project summary and links to generated output.

MATLAB Coder - euclidean.prj

Finish Workflow PACKAGE

Static Library Generated Successfully

You can now use the library in your applications. [Learn more](#)

Project Summary

Functions euclidean.m

Project Type MATLAB Coder

Project File euclidean.prj

Generated Output

C Code C:\coder\work\codegen\lib\euclidean

Binaries C:\coder\work\codegen\lib\euclidean\euclidean.lib

Example main Files C:\coder\work\codegen\lib\euclidean\examples

Reports Code Generation Report

Back

Compare the Generated C Code to Original MATLAB Code

To compare your generated C code to the original MATLAB code, open the C file, `euclidean.c`, and the `euclidean.m` file in the MATLAB Editor.

Important information about the generated C code:

- The function signature is:

```
void euclidean(const double x[3], const double cb[648], double y_min[3], double
              y_max[3], double idx[2], double distance[2])
```

`const double x[3]` corresponds to the input `x` in your MATLAB code. The size of `x` is 3, which corresponds to the total size (3 x 1) of the example input that you used when you generated code from your MATLAB code.

`const double cb[648]` corresponds to the input `cb` in your MATLAB code. The size of `cb` is 648, which corresponds to the total size (3 x 216) of the example input that you used when you generated code from your MATLAB code. In this case, the generated code uses a one-dimensional array to represent a two-dimensional array in the MATLAB code.

The generated code has four additional input arguments: the arrays `y_min`, `y_max`, `idx`, and `distance`. These arrays are used to return the output values. They correspond to the output arguments `y_min`, `y_max`, `idx`, and `distance` in the original MATLAB code.

- The code generator preserves your function name and comments. When possible, the code generator preserves your variable names.

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.

With Embedded Coder, you can interactively trace between MATLAB code and generated C/C++ code. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).

Generate C Code for Variable-Size Inputs

The C function that you generated for `euclidean.m` can accept only inputs that have the same size as the sample inputs that you specified during code generation. However, the input arrays to the corresponding MATLAB function can be of any size. In this part of the tutorial, you generate C code from `euclidean.m` that accepts variable-size inputs.

Suppose that you want the dimensions of `x` and `cb` in the generated C code to have these properties:

- The first dimension of both `x` and `cb` can vary in size up to 3.
- The second dimension of `x` is fixed and has the value 1.
- The second dimension of `cb` can vary in size up to 216.

To specify these input properties:

- 1 In the **Define Input Types** step, enter the test file `test.m` and click **Autodefine Input Types** as before. The test file calls the entry-point function, `euclidean.m`, with the expected input types. The app determines that the input `x` is `double(3x1)` and the input `cb` is `double(3x216)`. These types specify fixed-size inputs.
- 2 Click the input type specifications and edit them. You can specify variable size, up to a specified limit, by using the `:` prefix. For example, `:3` implies that the corresponding dimension can vary in size up to 3. Change the types to `double(:3 x 1)` for `x` and `double(:3 x :216)` for `cb`.

Define Input Types

To convert MATLAB to C, you must define the type of each input for every entry point function. [Learn more](#)

To **automatically define input types**, call euclidean or enter a script that calls euclidean in the MATLAB prompt below:

>> test

Autodefine Input Types

euclidean.m Number of outputs: 4

x	double(3 x 1)
cb	double(3 x :216)

Add global

Back Next

You can now generate code by following the same steps as before. The function signature for the generated C code in `euclidean.c` now reads:

```
void euclidean(const double x_data[], const int x_size[1], const double cb_data[],
              const int cb_size[2], double y_min_data[], int y_min_size[1],
              double y_max_data[], int y_max_size[1], double idx[2], double
              distance[2])
```

The arguments `x_data`, `cb_data`, `y_min_data`, and `y_max_data` correspond to the input arguments `x` and `cb` and the output arguments `y_min` and `y_max` in the original MATLAB function. The C function now accepts four additional input arguments `x_size`, `cb_size`, `y_min_size`, and `y_max_size` that specify the sizes of `x_data`, `cb_data`, `y_min_data`, and `y_max_data` at run time.

Next Steps

Goal	More Information
Learn about code generation support for MATLAB built-in functions and toolbox functions, classes, and System objects	"Functions and Objects Supported for C/C++ Code Generation"

Goal	More Information
Generate C++ code	"C++ Code Generation"
Generate and modify an example C main function and use it to build a C executable program	"Use an Example C Main in an Application"
Package generated files into a compressed file	"Package Code for Other Development Environments"
Optimize the execution speed or memory usage of generated code	"Optimization Strategies"
Integrate your custom C/C++ code into the generated code	"Call Custom C/C++ Code from the Generated Code"
Learn about the code generation report	"Code Generation Reports" "Interactively Trace Between MATLAB Code and Generated C/C++ Code" (Embedded Coder)

Generate C Code at the Command Line

In this tutorial, you use the MATLAB Coder `codegen` command to generate a static C library for a MATLAB function. You first generate C code that can accept only inputs that have fixed preassigned size. You then generate C code that can accept inputs of many different sizes.

You can also generate code by using the MATLAB Coder app. For a tutorial on this workflow, see “Generate C Code by Using the MATLAB Coder App” on page 2-2.

Tutorial Files: Euclidean Distance

Open this example to obtain the files for this tutorial.

Description of Tutorial Files

This tutorial uses the `euclidean_data.mat`, `euclidean.m`, `test.m`, `build_lib_fixed.m`, and `build_lib_variable.m` files.

- The MATLAB data file `euclidean_data.mat` contains two pieces of data: a single point in three-dimensional Euclidean space and a set of several other points in three-dimensional Euclidean space. More specifically:
 - `x` is a 3-by-1 column vector that represents a point in three-dimensional Euclidean space.
 - `cb` is a 3-by-216 array. Each column in `cb` represents a point in three-dimensional Euclidean space.
- The MATLAB file `euclidean.m` contains the function `euclidean` that implements the *core algorithm* in this example. The function takes `x` and `cb` as inputs. It calculates the Euclidean distance between `x` and each point in `cb` and returns these quantities:
 - The column vector `y_min`, which is equal to the column in `cb` that represents the point that is closest to `x`.
 - The column vector `y_max`, which is equal to the column in `cb` that represents the point that is farthest from `x`.
 - The 2-dimensional vector `idx` that contains the column indices of the vectors `y_min` and `y_max` in `cb`.
 - The 2-dimensional vector `distance` that contains the calculated smallest and largest distances to `x`.

```
function [y_min,y_max,idx,distance] = euclidean(x,cb)
% Initialize minimum distance as distance to first element of cb
% Initialize maximum distance as distance to first element of cb
idx(1)=1;
idx(2)=1;

distance(1)=norm(x-cb(:,1));
distance(2)=norm(x-cb(:,1));

% Find the vector in cb with minimum distance to x
% Find the vector in cb with maximum distance to x
for index=2:size(cb,2)
```

```

    d=norm(x-cb(:,index));
    if d < distance(1)
        distance(1)=d;
        idx(1)=index;
    end
    if d > distance(2)
        distance(2)=d;
        idx(2)=index;
    end
end

% Output the minimum and maximum distance vectors
y_min=cb(:,idx(1));
y_max=cb(:,idx(2));

end

```

- The MATLAB script `test.m` loads the data file `euclidean_data.mat` into the workspace. It then calls the function `euclidean` to calculate `y_min`, `y_max`, `idx`, and `distance`. The script then displays the calculated quantities at the command line.

Loading `euclidean_data.mat` is the preprocessing step that is executed before calling the core algorithm. Displaying the results is the post-processing step.

```

% Load test data
load euclidean_data.mat

% Determine closest and farthest points and corresponding distances
[y_min,y_max,idx,distance] = euclidean(x,cb);

% Display output for the closest point
disp('Coordinates of the closest point are: ');
disp(num2str(y_min));
disp(['Index of the closest point is ', num2str(idx(1))]);
disp(['Distance to the closest point is ', num2str(distance(1))]);

disp(newline);

% Display output for the farthest point
disp('Coordinates of the farthest point are: ');
disp(num2str(y_max));
disp(['Index of the farthest point is ', num2str(idx(2))]);
disp(['Distance to the farthest point is ', num2str(distance(2))]);

```

- The build scripts `build_lib_fixed.m` and `build_lib_variable.m` contain commands for generating static C libraries from your MATLAB code that accept fixed-size and variable-size inputs, respectively. The contents of these scripts are shown later in the tutorial, when you generate the C code.

Tip You can generate code from MATLAB functions by using MATLAB Coder. Code generation from MATLAB scripts is not supported.

Use test scripts to separate the pre- and post-processing steps from the function implementing the core algorithm. This practice enables you to easily reuse your algorithm. You generate code for the MATLAB function that implements the core algorithm. You do not generate code for the test script.

Generate C Code for the MATLAB Function

Run the Original MATLAB Code

Run the test script `test.m` in MATLAB. The output displays `y`, `idx`, and `distance`.

```
Coordinates of the closest point are:  
0.8      0.8      0.4  
Index of the closest point is 171  
Distance to the closest point is 0.080374
```

```
Coordinates of the farthest point are:  
0 0 1  
Index of the farthest point is 6  
Distance to the farthest point is 1.2923
```

Make the MATLAB Code Suitable for Code Generation

To make your MATLAB code suitable for code generation, you use the Code Analyzer and the Code Generation Readiness Tool. The Code Analyzer in the MATLAB Editor continuously checks your code as you enter it. It reports issues and recommends modifications to maximize performance and maintainability. The Code Generation Readiness Tool screens the MATLAB code for features and functions that are not supported for code generation.

Certain MATLAB built-in functions and toolbox functions, classes, and System objects that are supported for C/C++ code generation have specific code generation limitations. These limitations and related usage notes are listed in the **Extended Capabilities** sections of their corresponding reference pages. For more information, see “Functions and Objects Supported for C/C++ Code Generation”.

- 1 Open `euclidean.m` in the MATLAB Editor. The Code Analyzer message indicator in the top right corner of the MATLAB Editor is green. The analyzer did not detect errors, warnings, or opportunities for improvement in the code.
- 2 After the function declaration, add the `%#codegen` directive:

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

The `%#codegen` directive prompts the Code Analyzer to identify warnings and errors specific to code generation.

The Code Analyzer message indicator becomes red, indicating that it has detected code generation issues.

```

1 function [y_min,y_max,idx,distance] = euclidean(x,cb) %#codegen
2 % Initialize minimum distance as distance to first element of cb
3 % Initialize maximum distance as distance to first element of cb
4 -   idx(1)=1;
5 -   idx(2)=1;
6
7 -   distance(1)=norm(x-cb(:,1));
8 -   distance(2)=norm(x-cb(:,1));
9
10 % Find the vector in cb with minimum distance to x
11 % Find the vector in cb with maximum distance to x

```

- 3 To view the warning messages, move your cursor to the underlined code fragments. The warnings indicate that code generation requires the variables `idx` and `distance` to be fully defined before subscripting them. These warnings appear because the code generator must determine the sizes of these variables at their first appearance in the code. To fix this issue, use the `ones` function to simultaneously allocate and initialize these arrays.

```

% Initialize minimum distance as distance to first element of cb
% Initialize maximum distance as distance to first element of cb
idx = ones(1,2);

distance = ones(1,2)*norm(x-cb(:,1));

```

The Code Analyzer message indicator becomes green again, indicating that it does not detect any more code generation issues.

```

1 function [y_min,y_max,idx,distance] = euclidean(x,cb) %#codegen
2 % Initialize minimum distance as distance to first element of cb
3 % Initialize maximum distance as distance to first element of cb
4 -   idx = ones(1,2);
5
6 -   distance = ones(1,2)*norm(x-cb(:,1));
7
8 % Find the vector in cb with minimum distance to x
9 % Find the vector in cb with maximum distance to x

```

For more information about using the Code Analyzer, see “Check Code for Errors and Warnings Using the Code Analyzer”.

- 4 Save the file.
- 5 To run the Code Generation Readiness Tool, call the `coder.screener` function from the MATLAB command line.

```
coder.screener('euclidean')
```

The tool does not detect any code generation issues for `euclidean`. For more information, see “Code Generation Readiness Tool”.

The Code Generation Readiness Tool is not supported in MATLAB Online.

Note The Code Analyzer and the Code Generation Readiness Tool might not detect all code generation issues. After eliminating the errors or warnings that these tools detect, generate code by using MATLAB Coder to determine if your MATLAB code has other compliance issues.

You are now ready to compile your code by using the MATLAB Coder app. Here, compilation refers to the generation of C/C++ code from your MATLAB code.

Note Compilation of MATLAB code refers to the generation of C/C++ code from the MATLAB code. In other contexts, the term compilation could refer to the action of a C/C++ compiler.

Defining Input Types

Because C uses static typing, the code generator must determine the class, size, and complexity of all variables in the MATLAB files at code generation time, also known as *compile time*. Therefore, when you generate code for the files, you must specify the properties of all input arguments to the entry-point functions. An entry-point function is a top-level MATLAB function from which you generate code.

When you generate code by using the `codegen` command, use the `-args` option to specify sample input parameters to the entry-point functions. The code generator uses this information to determine the properties of the input arguments.

In the next step, you use the `codegen` command to generate a MEX file from your entry-point function `euclidean`.

Check for Run-Time Issues

You generate a MEX function from your entry-point function. A MEX function is generated code that can be called from inside MATLAB. You run the MEX function and check whether the generated MEX function and the original MATLAB function have the same functionality.

It is a best practice to perform this step because you can detect and fix run-time errors that are harder to diagnose in the generated C code. By default, the MEX function includes memory integrity checks. These checks perform array bounds and dimension checking. The checks detect violations of memory integrity in code generated for MATLAB functions. For more information, see “Control Run-Time Checks”.

To convert MATLAB code to efficient C/C++ source code, the code generator introduces optimizations that, in certain situations, cause the generated code to behave differently than the original source code. See “Differences Between Generated Code and MATLAB Code”.

- 1 Generate a MEX file for `euclidean.m` by using the `codegen` command. To verify the MEX function, run the test script `test` with calls to the MATLAB function `euclidean` replaced with calls to the generated MEX function.

```
codegen euclidean.m -args {x,cb} -test test
```

- By default, `codegen` generates a MEX function named `euclidean_mex` in the current folder.
- You use the `-args` option to specify sample input parameters to the entry-point function `euclidean`. The code generator uses this information to determine the properties of the input arguments.
- You use the `-test` option to run the test file `test.m`. This option replaces the calls to `euclidean` in the test file with calls to `euclidean_mex`.

The output is:

```
Running test file: 'test' with MEX function 'euclidean_mex'.
Coordinates of the closest point are:
0.8      0.8      0.4
Index of the closest point is 171
Distance to the closest point is 0.080374
```

```
Coordinates of the farthest point are:
0  0  1
Index of the farthest point is 6
Distance to the farthest point is 1.2923
```

This output matches the output that was generated by the original MATLAB function and verifies the MEX function. Now you are ready to generate standalone C code for `euclidean`.

Note Before generating standalone C/C++ code from your MATLAB code, generate a MEX function. Run the generated MEX function and make sure it has the same run-time behavior as your MATLAB function. If the generated MEX function produces answers that are different from MATLAB, or produces an error, you must fix these issues before proceeding to standalone code generation. Otherwise, the standalone code that you generate might be unreliable and have undefined behavior.

Generate C Code

The build script `build_lib_fixed.m` contains the commands that you use to generate code for `euclidean.m`.

```
% Load the test data
load euclidean_data.mat
% Generate code for euclidean.m with codegen. Use the test data as example input.
codegen -report -config:lib euclidean.m -args {x, cb}
```

Note that:

- `codegen` reads the file `euclidean.m` and translates the MATLAB code into C code.
- The `-report` option instructs `codegen` to generate a code generation report that you can use to debug code generation issues and verify that your MATLAB code is suitable for code generation.
- The `-config:lib` option instructs `codegen` to generate a static C library instead of generating the default MEX function.
- The `-args` option instructs `codegen` to generate code for `euclidean.m` using the class, size, and complexity of the sample input parameters `x` and `cb`.

Instead of generating a C static library, you can choose to generate a MEX function or other C/C++ build types by using suitable options with the `codegen` command. For more information on the various code generation options, see `codegen`.

- 1 Run the build script.

MATLAB processes the build file and outputs the message:

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

The code generator produces a standalone C static library `euclidean` in `work\codegen\lib\euclidean`. Here, `work` is the folder that contains your tutorial files.

- To view the code generation report in the Report Viewer, click **View report**.

If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code. See “Code Generation Reports”.

Tip Use a build script to generate code at the command line. A build script automates a series of MATLAB commands that you perform repeatedly at the command line, saving you time and eliminating input errors.

Compare the Generated C Code to Original MATLAB Code

To compare your generated C code to the original MATLAB code, open the C file, `euclidean.c`, and the `euclidean.m` file in the MATLAB Editor.

Important information about the generated C code:

- The function signature is:

```
void euclidean(const double x[3], const double cb[648], double y_min[3], double
              y_max[3], double idx[2], double distance[2])
```

`const double x[3]` corresponds to the input `x` in your MATLAB code. The size of `x` is 3, which corresponds to the total size (3 x 1) of the example input you used when you generated code for your MATLAB code.

`const double cb[648]` corresponds to the input `cb` in your MATLAB code. The size of `cb` is 648, which corresponds to the total size (3 x 216) of the example input you used when you generated code for your MATLAB code. In this case, the generated code uses a one-dimensional array to represent a two-dimensional array in the MATLAB code.

The generated code has four additional input arguments: the arrays `y_min`, `y_max`, `idx`, and `distance`. These arrays are used to return the output values. They correspond to the output arguments `y_min`, `y_max`, `idx`, and `distance` in the original MATLAB code.

- The code generator preserves your function name and comments. When possible, the code generator preserves your variable names.

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.

With Embedded Coder, you can interactively trace between MATLAB code and generated C/C++ code. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).

Generate C Code for Variable-Size Inputs

The C function that you generated for `euclidean.m` can accept only inputs that have the same size as the sample inputs that you specified during code generation. However, the input arrays to the corresponding MATLAB function can be of any size. In this part of the tutorial, you generate C code from `euclidean.m` that accepts variable-size inputs.

Suppose that you want the dimensions of `x` and `cb` in the generated C code to have these properties:

- The first dimension of both `x` and `cb` can vary in size up to 3.
- The second dimension of `x` is fixed and has the value 1.
- The second dimension of `cb` can vary in size up to 216.

To specify these input properties, use the `coder.typeof` function. `coder.typeof(A,B,1)` specifies a variable-size input with the same class and complexity as `A` and upper bounds given by the corresponding element of the size vector `B`. Use the build script `build_lib_variable.m` that uses `coder.typeof` to specify the properties of variable-size inputs in the generated C library.

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

% Use coder.typeof to specify variable-size inputs
eg_x=coder.typeof(x,[3 1],1);
eg_cb=coder.typeof(cb,[3 216],1);

% Generate code for euclidean.m using coder.typeof to specify
% upper bounds for the example inputs
codegen -report -config:lib euclidean.m -args {eg_x,eg_cb}
```

You can now generate code by following the same steps as before. The function signature for the generated C code in `euclidean.c` now reads:

```
void euclidean(const double x_data[], const int x_size[1], const double cb_data[],
              const int cb_size[2], double y_min_data[], int y_min_size[1],
              double y_max_data[], int y_max_size[1], double idx[2], double
              distance[2])
```

The arguments `x_data`, `cb_data`, `y_min_data`, and `y_max_data` correspond to the input arguments `x` and `cb` and the output arguments `y_min` and `y_max` in the original MATLAB function. The C function now accepts four additional input arguments `x_size`, `cb_size`, `y_min_size` and `y_max_size` that specify the sizes of `x_data`, `cb_data`, `y_min_data`, and `y_max_data` at run time.

Next Steps

Goal	More Information
Learn about code generation support for MATLAB built-in functions and toolbox functions, classes, and System objects	"Functions and Objects Supported for C/C++ Code Generation"
Generate C++ code	"C++ Code Generation"
Create and edit input types interactively	"Create and Edit Input Types by Using the Coder Type Editor"
Generate and modify an example C main function and use it to build a C executable program	"Use an Example C Main in an Application"
Package generated files into a compressed file	"Package Code for Other Development Environments"
Optimize the execution speed or memory usage of generated code	"Optimization Strategies"

Goal	More Information
Integrate your custom C/C++ code into the generated code	"Call Custom C/C++ Code from the Generated Code"
Learn about the code generation report	"Code Generation Reports" "Interactively Trace Between MATLAB Code and Generated C/C++ Code" (Embedded Coder)

See Also

codegen | coder.screener

Accelerate MATLAB Algorithm by Generating MEX Function

You can use MATLAB Coder to generate a MEX function from your MATLAB code. A MEX function is a MATLAB executable. It is generated code that can be called from inside MATLAB. While working inside the MATLAB environment, use MEX functions to accelerate the computationally intensive portions of your MATLAB code. Generate a MEX function from your MATLAB code by using the MATLAB Coder app or by using `codegen` at the MATLAB command line.

In this tutorial, you use the MATLAB Coder `codegen` command to generate a MEX function for a MATLAB function. You first generate a MEX function that can accept only inputs that have fixed, preassigned size. You then generate another MEX function that can accept inputs of many different sizes.

Tutorial Files: Euclidean Distance

Open this example to obtain the files for this tutorial.

Description of Tutorial Files

This tutorial uses the `euclidean_data.mat`, `euclidean.m`, `test.m`, `test_2d.m`, `build_mex_fixed.m`, and `build_mex_variable.m` files.

- The MATLAB data file `euclidean_data.mat` contains two pieces of data: a single point in three-dimensional Euclidean space and a set of several other points in three-dimensional Euclidean space. More specifically:
 - `x` is a 3-by-1 column vector that represents a point in three-dimensional Euclidean space.
 - `cb` is a 3-by-216 array. Each column in `cb` represents a point in three-dimensional Euclidean space.
- The MATLAB file `euclidean.m` contains the function `euclidean` that implements the *core algorithm* in this example. The function takes `x` and `cb` as inputs. It calculates the Euclidean distance between `x` and each point in `cb` and returns these quantities:
 - The column vector `y_min`, which is equal to the column in `cb` that represents the point closest to `x`.
 - The column vector `y_max`, which is equal to the column in `cb` that represents the point farthest from `x`.
 - The 2-dimensional vector `idx` that contains the column indices of the vectors `y_min` and `y_max` in `cb`.
 - The 2-dimensional vector `distance` that contains the calculated smallest and largest distances to `x`.

```
function [y_min,y_max,idx,distance] = euclidean(x,cb)
% Initialize minimum distance as distance to first element of cb
% Initialize maximum distance as distance to first element of cb
idx(1)=1;
idx(2)=1;

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

```

distance(2)=norm(x-cb(:,1));

% Find the vector in cb with minimum distance to x
% Find the vector in cb with maximum distance to x
for index=2:size(cb,2)
    d=norm(x-cb(:,index));
    if d < distance(1)
        distance(1)=d;
        idx(1)=index;
    end
    if d > distance(2)
        distance(2)=d;
        idx(2)=index;
    end
end

% Output the minimum and maximum distance vectors
y_min=cb(:,idx(1));
y_max=cb(:,idx(2));

end

```

- The MATLAB script `test.m` loads the data file `euclidean_data.mat` into the workspace. It calls the function `euclidean` to calculate `y_min`, `y_max`, `idx`, and `distance`. The script then displays the calculated quantities at the command line.

Loading `euclidean_data.mat` is the preprocessing step that is executed before calling the core algorithm. Displaying the results is the post-processing step.

```

% Load test data
load euclidean_data.mat

% Determine closest and farthest points and corresponding distances
[y_min,y_max,idx,distance] = euclidean(x,cb);

% Display output for the closest point
disp('Coordinates of the closest point are: ');
disp(num2str(y_min));
disp(['Index of the closest point is ', num2str(idx(1))]);
disp(['Distance to the closest point is ', num2str(distance(1))]);

disp(newline);

% Display output for the farthest point
disp('Coordinates of the farthest point are: ');
disp(num2str(y_max));
disp(['Index of the farthest point is ', num2str(idx(2))]);
disp(['Distance to the farthest point is ', num2str(distance(2))]);

```

- The MATLAB script `test_2d.m` is a modification of `test.m` for points in two-dimensional Euclidean space. The contents of `test_2d.m` are shown later in the tutorial, when you use it to test the MEX function for variable-size inputs.
- The build scripts `build_mex_fixed.m` and `build_mex_variable.m` contain commands for generating MEX functions from your MATLAB code that accept fixed-size and variable-size inputs, respectively. The contents of these scripts are shown later in the tutorial, when you generate the C code.

Tip You can generate code from MATLAB functions by using MATLAB Coder. Code generation from MATLAB scripts is not supported.

Use test scripts to separate the pre- and post-processing steps from the function that implements the core algorithm. This practice enables you to easily reuse your algorithm. You generate code for the MATLAB function implementing the core algorithm. You do not generate code for the test script.

Generate MEX Function for the MATLAB Function

Run the Original MATLAB Code

Run the test script `test.m` in MATLAB. The output displays `y`, `idx`, and `distance`.

```
Coordinates of the closest point are:  
0.8      0.8      0.4  
Index of the closest point is 171  
Distance to the closest point is 0.080374
```

```
Coordinates of the farthest point are:  
0 0 1  
Index of the farthest point is 6  
Distance to the farthest point is 1.2923
```

Make the MATLAB Code Suitable for Code Generation

To make your MATLAB code suitable for code generation, you use the Code Analyzer and the Code Generation Readiness Tool. The Code Analyzer in the MATLAB Editor continuously checks your code as you enter it. It reports issues and recommends modifications to maximize performance and maintainability. The Code Generation Readiness Tool screens the MATLAB code for features and functions that are not supported for code generation.

Certain MATLAB built-in functions and toolbox functions, classes, and System objects that are supported for C/C++ code generation have specific code generation limitations. These limitations and related usage notes are listed in the **Extended Capabilities** sections of their corresponding reference pages. For more information, see “Functions and Objects Supported for C/C++ Code Generation”.

- 1 Open `euclidean.m` in the MATLAB Editor. The Code Analyzer message indicator in the top right corner of the MATLAB Editor is green. The analyzer did not detect errors, warnings, or opportunities for improvement in the code.
- 2 After the function declaration, add the `%#codegen` directive:

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

The `%#codegen` directive prompts the Code Analyzer to identify warnings and errors specific to code generation.

The Code Analyzer message indicator becomes red, indicating that it has detected code generation issues.


```

1  function [y_min,y_max,idx,distance] = euclidean(x,cb) %#codegen
2  % Initialize minimum distance as distance to first element of cb
3  % Initialize maximum distance as distance to first element of cb
4  -  idx(1)=1;
5  -  idx(2)=1;
6
7  -  distance(1)=norm(x-cb(:,1));
8  -  distance(2)=norm(x-cb(:,1));
9
10 % Find the vector in cb with minimum distance to x
11 % Find the vector in cb with maximum distance to x

```

- 3 To view the warning messages, move your cursor to the underlined code fragments. The warnings indicate that code generation requires the variables `idx` and `distance` to be fully defined before subscripting them. This warning appears because the code generator must determine the sizes of these variables at their first appearance in the code. To fix this issue, use the `ones` function to simultaneously allocate and initialize these arrays.

```

% Initialize minimum distance as distance to first element of cb
% Initialize maximum distance as distance to first element of cb
idx = ones(1,2);

distance = ones(1,2)*norm(x-cb(:,1));

```

The Code Analyzer message indicator becomes green again, indicating that it does not detect any more code generation issues.

```

1  function [y_min,y_max,idx,distance] = euclidean(x,cb) %#codegen
2  % Initialize minimum distance as distance to first element of cb
3  % Initialize maximum distance as distance to first element of cb
4  -  idx = ones(1,2);
5
6  -  distance = ones(1,2)*norm(x-cb(:,1));
7
8  % Find the vector in cb with minimum distance to x
9  % Find the vector in cb with maximum distance to x

```

For more information about using the Code Analyzer, see “Check Code for Errors and Warnings Using the Code Analyzer”.

- 4 Save the file.
- 5 To run the Code Generation Readiness Tool, call the `coder.screener` function from the MATLAB command line:

```
coder.screener('euclidean')
```

The tool does not detect any code generation issues for `euclidean`. For more information, see “Code Generation Readiness Tool”.

The Code Generation Readiness Tool is not supported in MATLAB Online.

Note The Code Analyzer and the Code Generation Readiness Tool might not detect all code generation issues. After eliminating the errors or warnings that these tools detect, generate code by using MATLAB Coder to determine if your MATLAB code has other compliance issues.

You are now ready to compile your code by using the `codegen` command. Here, compilation refers to the generation of C/C++ code from your MATLAB code.

Note Compilation of MATLAB code refers to the generation of C/C++ code from the MATLAB code. In other contexts, the term compilation could refer to the action of a C/C++ compiler.

Defining Input Types

Because C uses static typing, the code generator must determine the class, size, and complexity of all variables in the MATLAB files at code generation time, also known as *compile time*. Therefore, when you generate code for the files, you must specify the properties of all input arguments to the entry-point functions. An entry-point function is a top-level MATLAB function from which you generate code.

When you generate code by using the `codegen` command, use the `-args` option to specify sample input parameters to the entry-point functions. The code generator uses this information to determine the properties of the input arguments.

In the next step, you use the `codegen` command to generate a MEX file from your entry-point function `euclidean`.

Generate and Validate the MEX Function

The build script `build_mex_fixed.m` contains the commands that you use to generate and validate a MEX function for `euclidean.m`. To validate the MEX function, you run the test script `test` with calls to the MATLAB function `euclidean` replaced with calls to the generated MEX function.

```
% Load the test data
load euclidean_data.mat
% Generate code for euclidean.m with codegen. Use the test data as example input. Validate MEX by
codegen -report euclidean.m -args {x, cb} -test test
```

Note that:

- By default, `codegen` generates a MEX function named `euclidean_mex` in the current folder.
- The `-report` option instructs `codegen` to generate a code generation report that you can use to debug code generation issues and verify that your MATLAB code is suitable for code generation.
- The `-args` option specifies sample input parameters to the entry-point function `euclidean`. The code generator uses this information to determine the class, size, and complexity of the input arguments.
- You use the `-test` option to run the test file `test.m`. This option replaces the calls to `euclidean` in the test file with calls to `euclidean_mex`.

For more information on the code generation options, see `codegen`.

- 1 Run the build script `build_mex_fixed.m`.

The code generator produces a MEX function `euclidean_mex` in the current working folder.

The output is:

```
Code generation successful: View report.
Running test file: 'test' with MEX function 'euclidean_mex'.
Coordinates of the closest point are:
0.8      0.8      0.4
Index of the closest point is 171
Distance to the closest point is 0.080374
```

```
Coordinates of the farthest point are:
0 0 1
Index of the farthest point is 6
Distance to the farthest point is 1.2923
```

This output matches the output that was generated by the original MATLAB function and verifies the MEX function.

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

If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code. See “Code Generation Reports”.

Tip Use a build script to generate code at the command line. A build script automates a series of MATLAB commands that you perform repeatedly at the command line, saving you time and eliminating input errors.

Generate MEX Function for Variable-Size Inputs

The MEX function that you generated for `euclidean.m` can accept only inputs that have the same size as the sample inputs that you specified during code generation. However, the input arrays to the corresponding MATLAB function can be of any size. In this part of the tutorial, you generate a MEX function from `euclidean.m` that accepts variable-size inputs.

Suppose that you want the dimensions of `x` and `cb` in the generated MEX function to have these properties:

- The first dimension of both `x` and `cb` can vary in size up to 3.
- The second dimension of `x` is fixed and has the value 1.
- The second dimension of `cb` can vary in size up to 216.

To specify these input properties, you use the `coder.typeof` function. `coder.typeof(A,B,1)` specifies a variable-size input with the same class and complexity as `A` and upper bounds given by the corresponding element of the size vector `B`. Use the build script `build_mex_variable.m` that uses `coder.typeof` to specify the properties of variable-size inputs in the generated MEX function.

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

```
% Use coder.typeof to specify variable-size inputs
eg_x=coder.typeof(x,[3 1],1);
eg_cb=coder.typeof(cb,[3 216],1);
```

```
% Generate code for euclidean.m using coder.typeof to specify
```

```
% upper bounds for the example inputs
codegen -report euclidean.m -args {eg_x, eg_cb}
```

You can verify that the new MEX function `euclidean_mex` accepts inputs of dimensions different from those of `x` and `cb`. The test script `test_2d.m` creates the input arrays `x2d` and `cb2d` that are two-dimensional versions of `x` and `cb`, respectively. It then calls the MATLAB function `euclidean` by using these input parameters.

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

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

% Determine closest and farthest points and corresponding distances
[y_min,y_max,idx,distance] = euclidean(x2d,cb2d);

% Display output for the closest point
disp('Coordinates of the closest point are: ');
disp(num2str(y_min));
disp(['Index of the closest point is ', num2str(idx(1))]);
disp(['Distance to the closest point is ', num2str(distance(1))]);

disp(newline);

% Display output for the farthest point
disp('Coordinates of the farthest point are: ');
disp(num2str(y_max));
disp(['Index of the farthest point is ', num2str(idx(2))]);
disp(['Distance to the farthest point is ', num2str(distance(2))]);
```

Running `test_2d.m` produces the output:

```
Coordinates of the closest point are:
0.8      0.8
Index of the closest point is 29
Distance to the closest point is 0.078672
```

```
Coordinates of the farthest point are:
0      0
Index of the farthest point is 1
Distance to the farthest point is 1.1357
```

To run the test script `test_2d.m` with the calls to `euclidean` replaced with calls to `euclidean_mex`, use `coder.runTest`.

```
coder.runTest('test_2d','euclidean')
```

The output matches the output generated by the original MATLAB function. This verifies the fact that the new MEX function can accept inputs of dimensions different from those of `x` and `cb`.

Next Steps

Goal	More Information
Learn about code generation support for MATLAB built-in functions and toolbox functions, classes, and System objects	"Functions and Objects Supported for C/C++ Code Generation"
Generate C++ MEX code	"C++ Code Generation"
Create and edit input types interactively	"Create and Edit Input Types by Using the Coder Type Editor"
Optimize the execution speed or memory usage of generated code	"Optimization Strategies"
Learn about the code generation report	"Code Generation Reports"
See execution times and code coverage for generated MEX functions in MATLAB Profiler	"Profile MEX Functions by Using MATLAB Profiler"

See Also

`codegen` | `coder.screener` | `coder.runTest`

Hello World

This example shows how to generate a MEX function from a simple MATLAB® function using the `codegen` command. You can use `codegen` to check that your MATLAB code is suitable for code generation and, in many cases, to accelerate your MATLAB algorithm. You can run the MEX function to check for run-time errors.

Prerequisites

There are no prerequisites for this example.

About the 'hello_world' Function

The `hello_world.m` function simply returns the string 'Hello World!'.

```
type hello_world

function y = hello_world
%#codegen
y = 'Hello World!';
```

The `%#codegen` directive indicates that the MATLAB code is intended for code generation.

Generate the MEX Function

First, generate a MEX function using the command `codegen` followed by the name of the MATLAB file to compile.

```
codegen hello_world

Code generation successful.
```

By default, `codegen` generates a MEX function named `hello_world_mex` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Run the MEX Function

Run the MEX function to compare its behavior to that of the original MATLAB function and to check for run-time errors.

```
hello_world_mex

ans =
'Hello World!'
```

Generate Code for an Averaging Filter

This example shows the recommended workflow for generating C code from a MATLAB® function using the `codegen` command. These are the steps:

1. Add the `%#codegen` directive to the MATLAB function to indicate that it is intended for code generation. This directive also enables the MATLAB code analyzer to identify warnings and errors specific to MATLAB for code generation.
2. Generate a MEX function to check that the MATLAB code is suitable for code generation. If errors occur, you should fix them before generating C code.
3. Test the MEX function in MATLAB to ensure that it is functionally equivalent to the original MATLAB code and that no run-time errors occur.
4. Generate C code.
5. Inspect the C code.

Prerequisites

There are no prerequisites for this example.

About the `averaging_filter` Function

The `averaging_filter.m` function acts as an averaging filter on the input signal; it takes an input vector of values and computes an average for each value in the vector. The output vector is the same size and shape as the input vector.

type `averaging_filter`

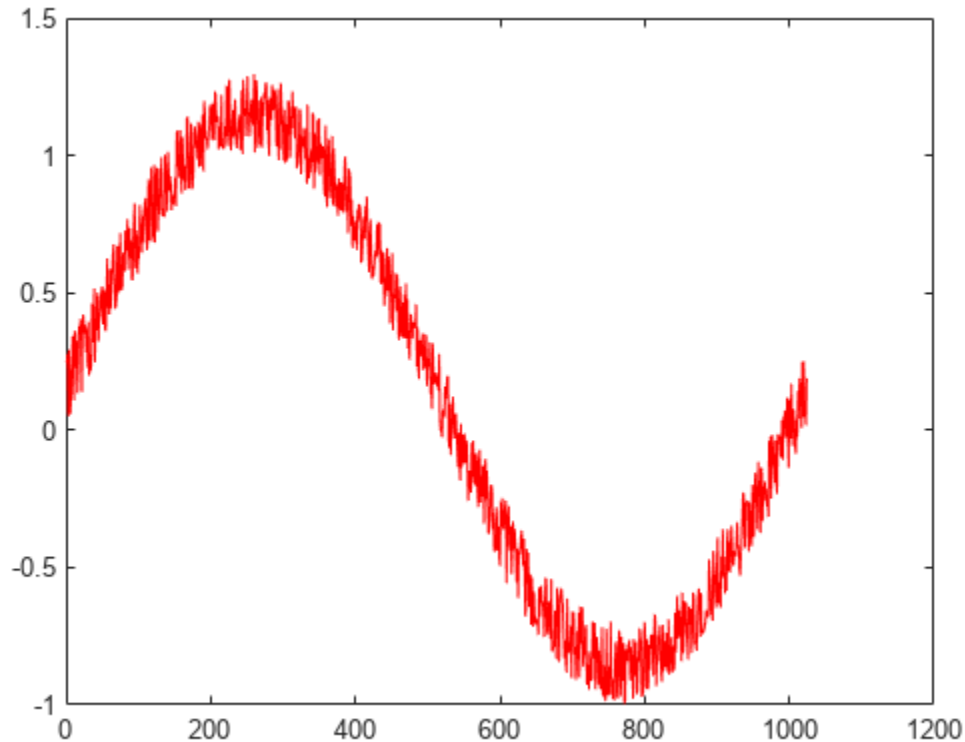
```
% y = averaging_filter(x)
% Take an input vector signal 'x' and produce an output vector signal 'y' with
% same type and shape as 'x' but filtered.
function y = averaging_filter(x) %#codegen
% Use a persistent variable 'buffer' that represents a sliding window of
% 16 samples at a time.
persistent buffer;
if isempty(buffer)
    buffer = zeros(16,1);
end
y = zeros(size(x), class(x));
for i = 1:numel(x)
    % Scroll the buffer
    buffer(2:end) = buffer(1:end-1);
    % Add a new sample value to the buffer
    buffer(1) = x(i);
    % Compute the current average value of the window and
    % write result
    y(i) = sum(buffer)/numel(buffer);
end
```

The `%#codegen` compilation directive indicates that the MATLAB code is intended for code generation.

Create Some Sample Data

Generate a noisy sine wave and plot the result.

```
v = 0:0.00614:2*pi;  
x = sin(v) + 0.3*rand(1,numel(v));  
plot(x, 'red');
```



Generate a MEX Function for Testing

Generate a MEX function using the `codegen` command. The `codegen` command checks that the MATLAB function is suitable for code generation and generates a MEX function that you can test in MATLAB prior to generating C code.

```
codegen averaging_filter -args {x}
```

Code generation successful.

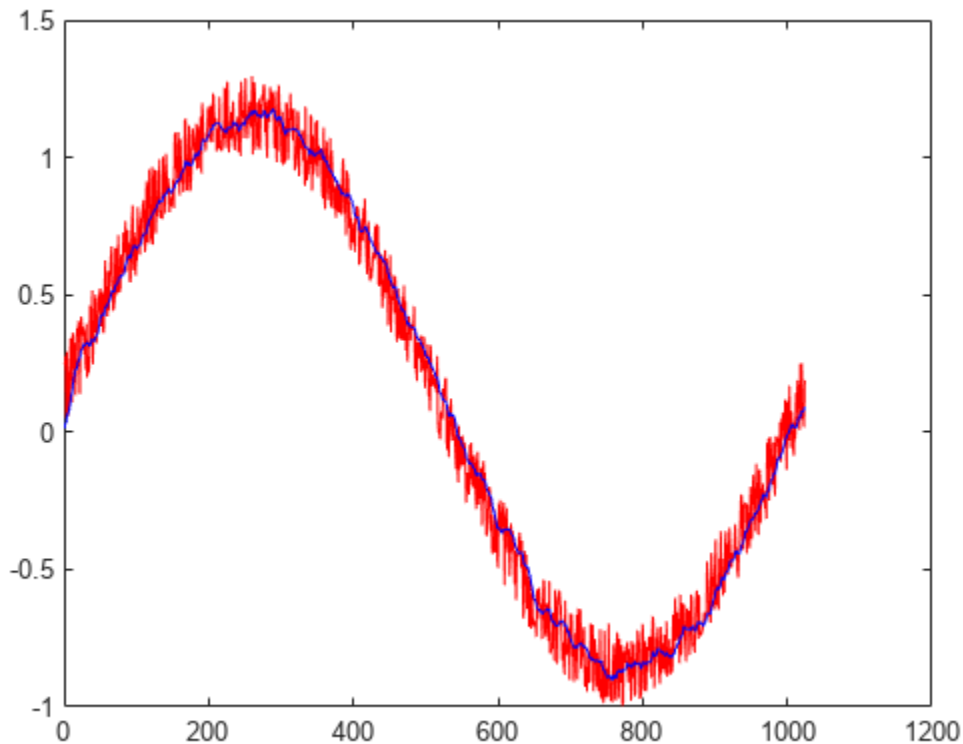
Because C uses static typing, `codegen` must determine the properties of all variables in the MATLAB files at compile time. Here, the `-args` command-line option supplies an example input so that `codegen` can infer new types based on the input types. Using the sample signal created above as the example input ensures that the MEX function can use the same input.

By default, `codegen` generates a MEX function named `averaging_filter_mex` in the current folder. This allows you to test the MATLAB code and MEX function and compare the results.

Test the MEX Function in MATLAB

Run the MEX function in MATLAB

```
y = averaging_filter_mex(x);
% Plot the result when the MEX function is applied to the noisy sine wave.
% The 'hold on' command ensures that the plot uses the same figure window as
% the previous plot command.
hold on;
plot(y, 'blue');
```



Generate C Code

```
codegen -config coder.config('lib') averaging_filter -args {x}
```

Code generation successful.

Inspect the Generated Code

The `codegen` command with the `-config coder.config('lib')` option generates C code packaged as a standalone C library. The generated C code is in the `codegen/lib/averaging_filter/` folder. The files are:

```
dir codegen/lib/averaging_filter/
```

```
.
..
_clang-format
```

Inspect the C Code for the averaging_filter.c Function

```

type codegen/lib/averaging_filter/averaging_filter.c

/*
 * File: averaging_filter.c
 *
 * MATLAB Coder version      : 5.6
 * C/C++ source code generated on : 03-Mar-2023 07:27:45
 */

/* Include Files */
#include "averaging_filter.h"
#include "averaging_filter_data.h"
#include "averaging_filter_initialize.h"
#include <string.h>

/* Variable Definitions */
static double buffer[16];

/* Function Definitions */
/*
 * Use a persistent variable 'buffer' that represents a sliding window of
 * 16 samples at a time.
 *
 * Arguments      : const double x[1024]
 *                  double y[1024]
 * Return Type    : void
 */
void averaging_filter(const double x[1024], double y[1024])
{
    double dv[15];
    int i;
    int k;
    if (!isInitialized_averaging_filter) {
        averaging_filter_initialize();
    }
    /* y = averaging_filter(x) */
    /* Take an input vector signal 'x' and produce an output vector signal 'y'
     * with */
    /* same type and shape as 'x' but filtered. */
    for (i = 0; i < 1024; i++) {
        double b_y;
        /* Scroll the buffer */
        memcpy(&dv[0], &buffer[0], 15U * sizeof(double));
        /* Add a new sample value to the buffer */
        b_y = x[i];
        buffer[0] = b_y;
        /* Compute the current average value of the window and */
        /* write result */
        for (k = 0; k < 15; k++) {
            double d;
            d = dv[k];
            buffer[k + 1] = d;
            b_y += d;
        }
        y[i] = b_y / 16.0;
    }
}

```

```
}

/*
 * Use a persistent variable 'buffer' that represents a sliding window of
 * 16 samples at a time.
 *
 * Arguments    : void
 * Return Type  : void
 */
void averaging_filter_init(void)
{
    memset(&buffer[0], 0, 16U * sizeof(double));
}

/*
 * File trailer for averaging_filter.c
 *
 * [EOF]
 */
```

Code Generation Guide: Generate Deployable C/C++ Code

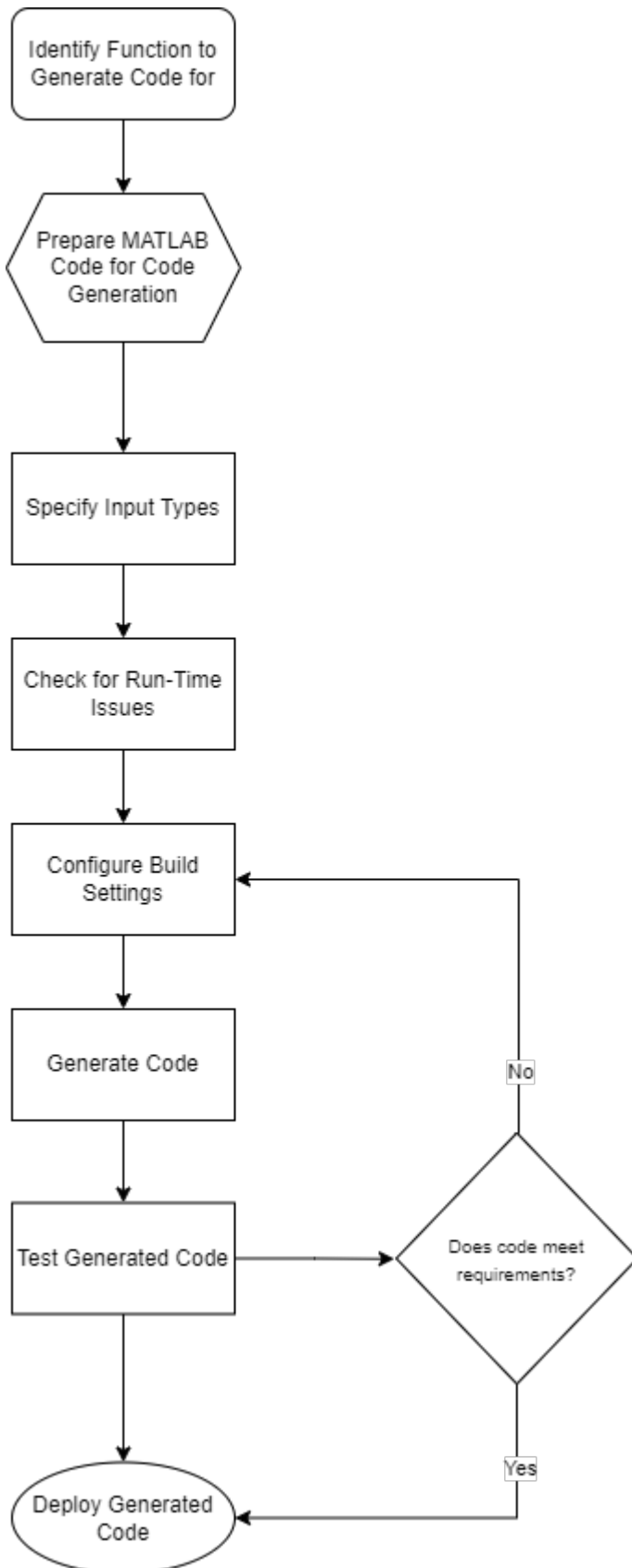
MATLAB Coder enables you to generate C/C++ code for your MATLAB code. You can:

- Generate standalone C/C++ code to use in your projects as source code, static libraries, or dynamic libraries.
- Generate MEX code to accelerate computation-intensive operations.

Follow the tasks in this guide to learn about standalone code generation and its deployment.

- 1** Prepare your MATLAB code for code generation.
- 2** Generate C/C++ code from your MATLAB code.
- 3** Test the generated C/C++ code.
- 4** Deploy the generated code to your existing projects.

To identify the tasks required to complete your code generation process, use this workflow diagram.



To start the tutorial, see “Prepare MATLAB Code for Code Generation” on page 2-43.

See Also

Related Examples

- “Prepare MATLAB Code for Code Generation” on page 2-43
- “Specify Input Types” on page 2-45
- “Configure Code Generation Build Settings” on page 2-45
- “Test Generated C/C++ Code” on page 2-49
- “Deploy Generated C/C++ Code” on page 2-51

Prepare MATLAB Code for Code Generation

To generate C/C++ code, the code generator converts dynamically typed MATLAB code to statically typed C/C++. Dynamically typed MATLAB variables can change their properties at run time. The same variable can hold a value of any class, size, or complexity. Statically typed languages such as C/C++ must determine variable types at compile time.

Before generating code, identify which function to generate code for. This function is termed the *entry-point* function or *primary* function. To prepare your code for code generation:

- 1 Initialize variables for code generation.
- 2 Screen code for unsupported functions and language features.

Initialize Variables for Code Generation

Because the generated code is statically typed, initialize all variables in your code before use to allow the code generator to identify and allocate the variables properly in the generated code. To identify some of these issues, include this line in your code.

```
%#codegen
```

This table lists some common errors that might occur while initializing variables in code intended for code generation.

Original Code	Issue	Modified Code
<code>y = zeros(1,10); y(3) = 1 + 2i;</code>	y is defined as double but assigned complex double value.	<code>y = complex(zeros(1,10)); y(3) = 1 + 2i;</code>
<code>for i = 1:N y(i,i) = i; end</code>	The array y is extended dynamically without being defined.	<code>y = zeros(N,N); for i = 1:N y(i,i) = i; end</code>

For information about data definition for code generation of specific data types, see “Data Definition Considerations for Code Generation” and “Best Practices for Defining Variables for C/C++ Code Generation”.

Screen Code for Unsupported Functions and Language Features

The code generator supports most language features and functions. To check for unsupported functions and language features in your code:

- 1 Start the MATLAB Coder App from the **APPS** tab. Alternatively, type this in the command line:

```
>> coder
```
- 2 Enter the entry-point function name in the app. Do not add sub-functions in this step. The code generator automatically includes all required sub-functions.
- 3 To obtain the Code Generation Readiness Tool Report, click **Next**. If there are unsupported functions or language features in your code, they are reported here.

Alternatively, call the screener on your entry-point function. At the command line, run this command:

```
coder.screener('filename');
```

This tool parses your code and highlights unsupported MATLAB functions and some unsupported language features. See “Functions and Objects Supported for C/C++ Code Generation” & `coder.screener`.

If your code includes unsupported functions, consider these workarounds:

- Check for replacement functions and System objects that support code generation.
- Write custom code for those functions.

Use `coder.ceval` to call a custom C function you have for that function.

- Use `coder.extrinsic` to call the function.

Tips

Set Advanced Code Generation Options at the Command Line

Use the `codegen` function with the configuration object `coder.config`. Depending on the type of build, you can also use `coder.CodeConfig`, `coder.EmbeddedCodeConfig`, and `coder.MexCodeConfig`.

Research Code Generation Considerations for Specific Functions

For functions supported for code generation, their reference pages contain a section titled *Extended Capabilities*. This section lists all special considerations when generating code for those functions. For example, see *Extended Capabilities* in `interp2`.

The `coder.extrinsic` Call

Calls to `coder.extrinsic` declares a function as an extrinsic function. The code generator does not produce code for the body of the extrinsic function and instead uses the MATLAB® engine to execute the call.

See Also

`coder.target` | `coder.screener` | `coder.ceval` | `coder.extrinsic` | `codegen` | `coder.config` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

Related Examples

- “Data Definition Considerations for Code Generation”
- “Best Practices for Defining Variables for C/C++ Code Generation”
- “MATLAB Language Features Supported for C/C++ Code Generation”
- “Use MATLAB Engine to Execute a Function Call in Generated Code”

Generate C/C++ Code from MATLAB Code

After verifying MEX code behavior, generate standalone code for your project.

- 1 Specify input types.
- 2 Check for run-time issues.
- 3 Configure code generation build settings.
- 4 Generate standalone C/C++ code.
- 5 Understand generated code.

Specify Input Types

Before generating code, provide the input types to the code generator. The code generator then determines the data types to use in the generated code.

To automatically define input types, call your function by using example inputs or enter a script that calls your function in the prompt. Provide input types directly by providing example inputs. If your code requires a matrix of doubles of size 3-by-4, the example input can be `zeros(3,4)` or `ones(3,4)`.

For more information, see “Input Type Specification for Code Generation” on page 1-11.

Check for Run-Time Issues

After defining the input types for the code generator, perform an initial code generation and code execution to detect run-time errors that are harder to diagnose in the generated code.

- 1 To generate a MEX file for your code, click on the **Check for Issues** button.
- 2 You can open the Code Generation Report automatically by selecting the **Always create a code generation report** option in the **Debugging** section under **More Settings**.

See “Why Test MEX Functions in MATLAB?”.

Configure Code Generation Build Settings

To create code according to your requirements, you can change the configuration settings of the code generator. In the **Generate Code** tab in the app, select the **More Settings** button at the bottom of the tab. This window lists the configuration settings that modify the generated code.

Use these settings to specify where the generated code should be built, apply target specific optimizations, enable variable-sizing support, include comments in the generated code, and apply other customizations for your generated code.

Generate Standalone C/C++ Code

After checking for run-time issues by generating a MEX file, generate standalone C/C++ code by choosing the required **Build type** under the **Generate Code** tab in the app.

To generate code for your project, click **Generate**.

Understand Generated Code

Access Code Generation Report

Use the code generation reports to view the generated C/C++ code, trace between the MATLAB code and generated C/C++ code, and identify potential issues in the generated code.

After code generation, to open the code generation reports.

- In the MATLAB Coder app, in the **Debugging** settings, select the check box for **Always create a report** and **Automatically launch a report if one is generated**.

For more information, see “Code Generation Reports”.

Array Layout in Generated Code

Programming languages and environments typically assume a single array layout for all data. MATLAB uses column-major layout by default, whereas C and C++ use row-major layout.

To generate row-major code:

- 1 In the app, open the **Generate** dialog box. On the **Generate Code** page, click the **Generate** arrow.
- 2 Click **More Settings**.
- 3 On the **Memory** tab, set **Array layout** to Row-major.

See “Code Design for Row-Major Array Layout”.

Memory Allocation in Generated Code

For code generation, an array dimension is *fixed-size* or *variable-size*. If the code generator can determine the size of the dimension and that the size of the dimension does not change, then the dimension is fixed-size. When all dimensions of an array are fixed-size, the array is a *fixed-size* array.

You can generate code that allocates memory for fixed-size and variable-size arrays on the program stack or on the heap.

Static memory allocation allocates memory for arrays on the program stack at compile time. Static allocation is beneficial when:

- You know the upper bounds of all arrays in use.
- You have a large program stack.
- The arrays are small and take up less space on the program stack.

Dynamic memory allocation allocates memory on the heap as needed at run time, instead of allocating memory statically on the stack. Dynamic memory allocation is beneficial when:

- You do not know the upper bound of an array.
- You do not want to allocate memory on the stack for large arrays.

Dynamic memory allocation can result in slower execution of the generated code. See “Control Memory Allocation for Variable-Size Arrays”.

To dynamically allocate memory for fixed-size and variable-size arrays in the generated code:

- 1 In the app, open the **Generate** dialog box. On the **Generate Code** page, click the **Generate** arrow.
- 2 Click **More Settings**.
- 3 On the **Memory** tab, select a value from the drop-down list for **Dynamic memory allocation for variable size arrays**.
- 4 On the **Memory** tab, select a value from the drop-down list for **Dynamic memory allocation for fixed size arrays**.

Setting these options to 'Threshold' causes arrays that are greater in size (in bytes) than the threshold value to be allocated dynamically.

Tips

Define Maximum Size for Unbounded Arrays

If you are generating code for arrays whose size depends on a user input, you can still set an upper limit for such inputs by using the `assert` function. For example:

```
function inSize(n)
assert(n < 25);
y = zeros(1,n);
end
```

When you cannot use dynamic memory allocation, define the upper bounds of arrays.

File I/O Support

The code generator includes limited support for functions like `coder.load`, `fread`, `fopen`, `fprintf`, and `fclose`.

Invoke Generated Code from Your C Project

The code generator provides an example main function for your reference when you generate static or dynamic libraries. See "Use an Example C Main in an Application".

Generate Code at the Command Line

You can generate code and set all options at the command line. See `codegen` and `coder.config`.

You can also use the code configuration objects to set these options at the command line. The options are properties of the configuration objects that are accessible by dot notation. See `coder.MexCodeConfig`, `coder.CodeConfig`, and `coder.EmbeddedCodeConfig`.

Note To open a dialog box that includes the associated build configuration settings, double-click on the configuration object in the workspace.

Convert Your Project to Script

You can also convert your projects to a script by using the `-tocode` option of the `coder` command.

Optimize the Generated Code

While the code generator produces optimized code for most applications, you can generate efficient C/C++ code for your project by following some of these best practices:

- Pass arguments by reference
- Inline code
- Integrate optimized external code
- Disable run-time checks

For more information, see “Optimization Strategies”.

Create Reports at the Command Line

When generating code at the command line, use these codegen options:

- To generate a report, use the `-report` option.
- To generate and open a report, use the `-launchreport` option.

Alternatively, use configuration object properties:

- To generate a report, set `GenerateReport` to `true`.
- If you want the `codegen` command to open the report for you, set `LaunchReport` to `true`.

See Also

`coder.cstructname` | `coder.MexCodeConfig` | `coder.CodeConfig` |
`coder.EmbeddedCodeConfig` | `coder.load` | `fread` | `fclose` | `fprintf` | `fopen` | `coder.opaque`

Related Examples

- “Specify Objects as Inputs at the Command Line”
- “Code Design for Row-Major Array Layout”
- “Control Memory Allocation for Variable-Size Arrays”

Test Generated C/C++ Code

After generating code for your MATLAB code, verify the run-time behavior of the generated code. To see the generated code and identify potential issues, access the code generation report.

You can test the generated code to verify code behavior, depending on your build type:

- Test MEX code to verify behavior.
- Test standalone code by using software-in-the-loop and processor-in-the-loop execution (requires Embedded Coder).

Test MEX Code to Verify Behavior

If you use the app to generate a MEX function, you can test the MEX function in the app.

- 1 On the **Generate Code** page, click **Verify Code**.
- 2 Type or select the test file name.
- 3 To run the test file without replacing calls to the original MATLAB function with calls to the MEX function, for **Run using**, select **MATLAB code**. Click **Run Generated Code**.
- 4 To run the test file, replacing calls to the original MATLAB function with calls to the MEX function, for **Run using**, select **Generated code**. Click **Run Generated Code**.
- 5 Compare the results of running the original MATLAB function to the results of running the MEX function.

If you have Embedded Coder, you can verify the numeric behavior of generated C/C++ code by using software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution. You can also produce a profile of execution times.

Test Standalone Code by Using Software-in-the-Loop and Processor-in-the-Loop

To test the generated standalone code on your target hardware, you can run unit tests on the generated code. To run unit tests on standalone code in a separate process outside of MATLAB, use software-in-the-loop (SIL) or processor-in-the-loop (PIL) execution. To use SIL or PIL execution, you must have Embedded Coder.

See “Software-in-the-Loop Execution with the MATLAB Coder App” (Embedded Coder) and “Processor-in-the-Loop Execution with the MATLAB Coder App” (Embedded Coder).

Tips

Test MEX Files at the Command Line

If you use `codegen` to generate a MEX function, use the `-test` option. For example:

```
codegen myfunction -test 'myfunction_test'
```

You can also test the MEX function by using `coder.runTest`. For example:

```
coder.runTest('myfunction_test', 'myfunction')
```

Unit Test Generated Code

See “Unit Test Generated Code with MATLAB Coder”.

To unit test external code, see “Unit Test External C Code with MATLAB Coder”.

See Also

`coder.runTest`

Related Examples

- “Code Generation Reports”
- “Software-in-the-Loop Execution with the MATLAB Coder App” (Embedded Coder)
- “Processor-in-the-Loop Execution with the MATLAB Coder App” (Embedded Coder)

Deploy Generated C/C++ Code

Once you have verified that the generated code behaves according to your requirements, you can deploy it. You can deploy the generated code as source code, static libraries, dynamic libraries, or executable applications.

To package all the required source files to build your application on an external platform, use the `packNGo` function.

To build executable applications with the code generator, you must:

- 1 Generate the required source code or library (completed in previous steps).
- 2 Create a main function that calls the generated code.
- 3 Iterate through the code generation process with build type set to `Executable (.exe)`.

Edit Generated Main Function and Interfaces

To create an application, create or use a C/C++ main function to call the C/C++ entry-point functions generated from your MATLAB functions. For example, see “Generating Standalone C/C++ Executables from MATLAB Code”.

Note Do not modify the files `main.c` and `main.h` in the `examples` subfolder. Before using the example main function, copy the example main source and header files to a location outside of the build folder. Modify the files in the new location to meet the requirements of your application.

Use the generated example main as a starting point for creating a main function. The example main provides a clear example for how to pass input to and output from the generated code. For more information and examples, see “Incorporate Generated Code Using an Example Main Function” and “Structure of Generated Example C/C++ Main Function”.

Generated Function Interfaces

To write a main function, you must be familiar with the generated function interfaces. See “Mapping MATLAB Types to Types in Generated Code”.

C/C++ entry-point functions follow these conventions:

- Pass input arrays by reference.
- Return output arrays by reference.
- Pass input scalars by value.
- Return scalars by value for single-output functions.
- Return scalars by reference:
 - For functions with multiple outputs.
 - When you use the same variable as input and output.

If you use the same variable as input and output in your MATLAB code, the generated code passes the scalar by reference. See “Avoid Data Copies of Function Inputs in Generated Code”.

Array Definition

The code generator creates C/C++ array definitions that depend on the array element type and their memory allocation type. See “Representation of Arrays in Generated Code”.

To learn more about the methods associated with arrays in the generated code, see:

- “Use C Arrays in the Generated Function Interfaces”
- “Use Dynamically Allocated C++ Arrays in Generated Function Interfaces”

Initialize and Terminate Functions

Your C/C++ code must call an initialize function and a terminate function that are generated in addition to your C/C++ entry-point functions. By default, the generated C/C++ entry-point function calls the initialize function. The generated example main function calls the terminate function. As you create and edit your own main function, make sure to call both initialize and terminate functions.

For more information, see “Use Generated Initialize and Terminate Functions”.

Build Executable Applications by Using MATLAB Coder

After you create a main file (`main.c`) and main header file (`main.h`), follow these steps to build executable applications by using the app:

- 1 Open the **Generate Code** page in the app.
- 2 Set **Build type** to Executable (`.exe`).
- 3 Click **More Settings**.
- 4 On the **Custom Code** tab, in **Additional source files**, enter `main.c`
- 5 On the **Custom Code** tab, in **Additional include directories**, enter the location of the modified `main.c` and `main.h` files. For example, `c:\myfiles`. Click **Close**.
- 6 To generate the executable, click **Generate**.

The app indicates that code generation succeeded.

- 7 Click **Next** to go to the **Finish Workflow** step.
- 8 Under **Generated Output**, you can see the location of the generated executable `filename.exe`.

After generating code and writing a main file that uses the generated code, you can generate executable applications with the code generator or other build tools. If you want to transfer the generated code to a target platform in an exportable zip file, use the `packNGo` function.

Target Specific Code Generation

To deploy your code to another platform, use the hardware support packages that support generating and building binary code for that platform.

In the MATLAB Coder app, during the **Generate Code** step, select a hardware support package from the **Hardware Board** drop-down list.

For a list of support packages provided for MATLAB Coder, see “MATLAB Coder Supported Hardware”. If you want to specify a custom toolchain for build that is not available from a hardware support package, you can register your own toolchain. See “Custom Toolchain Registration”.

Tips

Choose Hardware at the Command Line

At the command line, specify a hardware support package by using the `coder.hardware` function.

Run Executables in MATLAB

To run the executable in MATLAB on a Windows® platform:

```
system('filename.exe')
```

See Also

`coder.hardware` | `packNGo`

Related Examples

- “Deploy Generated Code”
- “Generating Standalone C/C++ Executables from MATLAB Code”
- “Incorporate Generated Code Using an Example Main Function”
- “Structure of Generated Example C/C++ Main Function”
- “Mapping MATLAB Types to Types in Generated Code”
- “Avoid Data Copies of Function Inputs in Generated Code”
- “Representation of Arrays in Generated Code”
- “Use C Arrays in the Generated Function Interfaces”
- “Use Dynamically Allocated C++ Arrays in Generated Function Interfaces”
- “MATLAB Coder Supported Hardware”
- “Custom Toolchain Registration”

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

-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
- 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

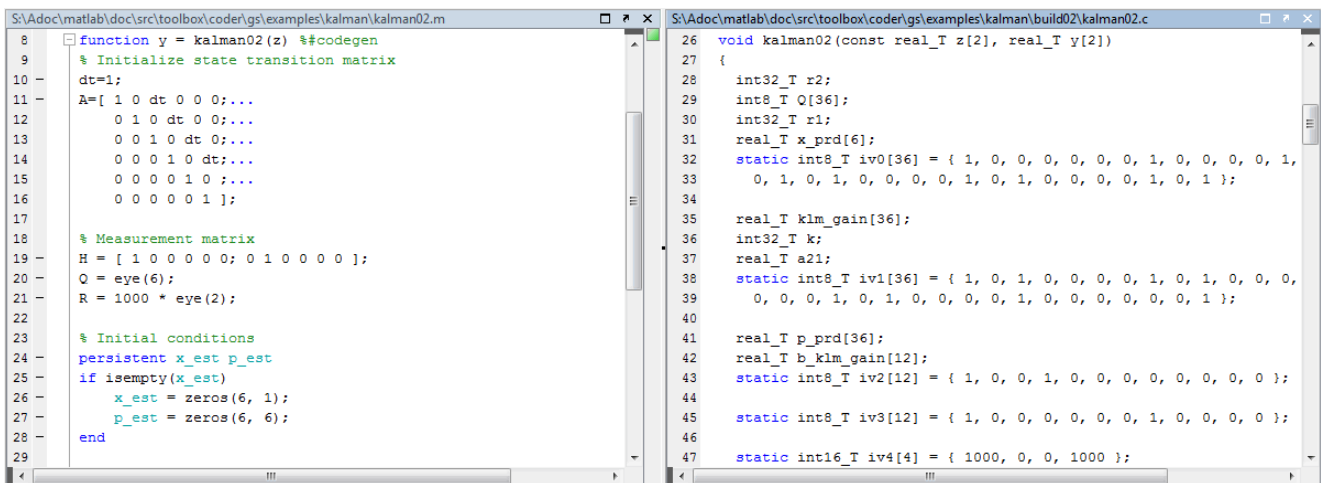
Use the MATLAB Editor's left/right tile feature to compare your generated C code to the original MATLAB code. You can easily compare the generated C code to your original MATLAB code. In the generated C code:

- 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.



```

S:\Adoc\matlab\doc\src\toolbox\coder\examples\kalman\kalman02.m
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

S:\Adoc\matlab\doc\src\toolbox\coder\examples\kalman\build02\kalman02.c
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, 1, 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, 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, 1, 0, 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.

