

MATLAB® Compiler SDK™

Getting Started Guide



MATLAB®

R2018a



How to Contact MathWorks



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



Phone: 508-647-7000



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

MATLAB® Compiler SDK™ Getting Started Guide

© COPYRIGHT 2012–2018 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

March 2015	Online only	New for Version 6.0 (Release R2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release R2016b)
March 2017	Online only	Revised for Version 6.3.1 (Release R2017a)
September 2017	Online only	Revised for Version 6.4 (Release R2017b)
March 2018	Online only	Revised for Version 6.5 (Release R2018a)

Overview of MATLAB Compiler SDK

1

MATLAB Compiler SDK Product Description	1-2
Key Features	1-2
Appropriate Tasks for MATLAB Compiler Products	1-3
Deployment Product Terms	1-6

Examples

2

Create a C Shared Library with MATLAB Code	2-2
Create Functions in MATLAB	2-2
Create C Shared Using Library Compiler App	2-4
Customize the Application and Its Appearance	2-5
Package the Application	2-6
Create a C++ Shared Library with MATLAB Code	2-8
Create Functions in MATLAB	2-8
Create a C++ Shared Library using the Library Compiler App	2-10
Customize the Application and Its Appearance	2-11
Package the Application	2-12
Create a .NET Assembly with MATLAB Code	2-14
Create Function in MATLAB	2-14
Create .NET Assembly Using Library Compiler App	2-14
Customize the Application and Its Appearance	2-16
Package the Application	2-17
Build a .NET Application	2-18

Create a Generic COM Component with MATLAB Code	2-22
Create Function in MATLAB	2-22
Create Generic COM Component Using Library Compiler App	2-22
Customize the Application and Its Appearance	2-23
Package the Application	2-25
Create a Java Package with MATLAB Code	2-27
Create Function in MATLAB	2-27
Create Java Application Using Library Compiler App	2-27
Customize the Application and Its Appearance	2-28
Package the Application	2-30
Install and Implement MATLAB Generated Java Application	2-31
Create a Python Package with MATLAB Code	2-34
Create Function in MATLAB	2-34
Create Python Application Using Library Compiler App	2-34
Customize the Application and Its Appearance	2-36
Package the Application	2-37
Install and Run MATLAB Generated Python Application	2-38

Customizing a Compiler Project

3

Customize the Application	3-2
Customize the Installer	3-2
Manage Required Files in Compiler Project	3-5
Specify Files to Install with Application	3-6
Additional Runtime Settings	3-6
API Selection for C++ Shared Library	3-7
Manage Support Packages	3-9
Using a Compiler App	3-9
Using the Command Line	3-10

Create a Deployable Archive for MATLAB Production Server	4-2
Create a C# Client	4-5
Create a Java Client	4-9
Create a C++ Client	4-13
Create a Python Client	4-19

Overview of MATLAB Compiler SDK

- “MATLAB Compiler SDK Product Description” on page 1-2
- “Appropriate Tasks for MATLAB Compiler Products” on page 1-3
- “Deployment Product Terms” on page 1-6

MATLAB Compiler SDK Product Description

Build software components from MATLAB programs

MATLAB Compiler SDK extends the functionality of MATLAB Compiler™ to let you build C/C++ shared libraries, Microsoft® .NET assemblies, and Java® classes from MATLAB programs. These components can be integrated with custom applications and then deployed to desktop, web, and enterprise systems.

MATLAB Compiler SDK includes a development version of MATLAB Production Server™ for testing and debugging application code and Excel® add-ins before deploying them to web applications and enterprise systems.

Applications created using software components from MATLAB Compiler SDK can be shared royalty-free with users who do not need MATLAB. These applications use the MATLAB Runtime, a set of shared libraries that enables the execution of compiled MATLAB applications or components.

Key Features

- Packaging of your MATLAB programs as C/C++ shared libraries, Microsoft .NET assemblies, and Java classes
- Royalty-free distribution of software components to users who do not need MATLAB
- Development and test framework for MATLAB Production Server for integration with web and enterprise systems
- Encryption of MATLAB code to protect your intellectual property

Appropriate Tasks for MATLAB Compiler Products

MATLAB Compiler generates standalone applications and Excel add-ins. MATLAB Compiler SDK generates C/C++ shared libraries, deployable archives for use with MATLAB Production Server, Java packages, .NET assemblies, and COM components.

While MATLAB Compiler and MATLAB Compiler SDK let you run your MATLAB application outside the MATLAB environment, it is not appropriate for all external tasks you may want to perform. Some tasks require other products or MATLAB external interfaces. Use the following table to determine if MATLAB Compiler or MATLAB Compiler SDK is appropriate to your needs.

MATLAB Compiler Task Matrix

Task	MATLAB Compiler and MATLAB Compiler SDK	MATLAB Coder™	Simulink®	HDL Coder™	MATLAB External Interfaces
Package MATLAB applications for deployment to users who do not have MATLAB	■				
Package MATLAB applications for deployment to MATLAB Production Server	■				
Build non-MATLAB applications that include MATLAB functions	■				
Generate readable and portable C/C++ code from MATLAB code		■			
Generate MEX functions from MATLAB code for code verification and acceleration.		■			
Integrate MATLAB code into Simulink			■		
Generate hardware description language (HDL) from MATLAB code				■	
Integrate custom C code into MATLAB with MEX files					■
Call MATLAB from C and Fortran programs					■

Note Components generated by MATLAB Compiler and MATLAB Compiler SDK cannot be used in the MATLAB environment.

For information on MATLAB Coder, see “MATLAB Coder”.

For information on Simulink, see “Simulink”.

For information on HDL Coder, see “HDL Coder”.

For information on MATLAB external interfaces, see “MATLAB API for Other Languages” (MATLAB).

Deployment Product Terms

A

Add-in — A Microsoft Excel add-in is an executable piece of code that can be actively integrated into a Microsoft Excel application. Add-ins are front-ends for COM components, usually written in some form of Microsoft Visual Basic®.

Application program interface (API) — A set of classes, methods, and interfaces that is used to develop software applications. Typically an API is used to provide access to specific functionality. See *MWArray*.

Application — An end user-system into which a deployed functions or solution is ultimately integrated. Typically, the end goal for the deployment customer is integration of a deployed MATLAB function into a larger enterprise environment application. The deployment products prepare the MATLAB function for integration by wrapping MATLAB code with enterprise-compatible source code, such as C, C++, C# (.NET), F#, and Java code.

Assembly — An executable bundle of code, especially in .NET.

B

Binary — See *Executable*.

Boxed Types — Data types used to wrap opaque C structures.

Build — See *Compile*.

C

Class — A user-defined type used in C++, C#, and Java, among other object-oriented languages, that is a prototype for an object in an object-oriented language. It is analogous to a derived type in a procedural language. A class is a set of objects which share a common structure and behavior. Classes relate in a class hierarchy. One class is a specialization (a subclass) of another (one of its *superclasses*) or comprises other classes. Some classes use other classes in a client-server relationship. Abstract classes have no members, and concrete classes have one or more members. Differs from a MATLAB class

Compile — In MATLAB Compiler and MATLAB Compiler SDK, to compile MATLAB code involves generating a binary that wraps around MATLAB code, enabling it to execute in various computing environments. For example, when MATLAB code is compiled into a

Java package, a Java wrapper provides Java code that enables the MATLAB code to execute in a Java environment.

COM component — In MATLAB Compiler, the executable back-end code behind a Microsoft Excel add-in. In MATLAB Compiler SDK, an executable component, to be integrated with Microsoft COM applications.

Console application — Any application that is executed from a system command prompt window.

D

Data Marshaling — Data conversion, usually from one type to another. Unless a MATLAB deployment customer is using type-safe interfaces, data marshaling—as from mathematical data types to MathWorks® data types such as represented by the `MWArray` API—must be performed manually, often at great cost.

Deploy — The act of integrating MATLAB code into a larger-scale computing environment, usually to an enterprise application, and often to end users.

Deployable archive — The deployable archive is embedded by default in each binary generated by MATLAB Compiler or MATLAB Compiler SDK. It houses the deployable package. All MATLAB-based content in the deployable archive uses the Advanced Encryption Standard (AES) cryptosystem. See “Additional Details” (MATLAB Compiler).

DLL — Dynamic link library. Microsoft's implementation of the shared library concept for Windows®. Using DLLs is much preferred over the previous technology of static (or non-dynamic) libraries, which had to be manually linked and updated.

E

Empties — Arrays of zero (0) dimensions.

Executable — An executable bundle of code, made up of binary bits (zeros and ones) and sometimes called a *binary*.

F

Fields — For this definition in the context of MATLAB Data Structures, see *Structs*.

Fields and Properties — In the context of .NET, *Fields* are specialized classes used to hold data. *Properties* allow users to access class variables as if they were accessing member fields directly, while actually implementing that access through a class method.

I

Integration — Combining deployed MATLAB code's functionality with functionality that currently exists in an enterprise application. For example, a customer creates a mathematical model to forecast trends in certain commodities markets. In order to use this model in a larger-scale financial application (one written with the Microsoft .NET Framework, for instance) the deployed financial model must be integrated with existing C# applications, run in the .NET enterprise environment.

Instance — For the definition of this term in context of MATLAB Production Server software, see *MATLAB Production Server Server Instance*.

J

JAR — Java archive. In computing software, a JAR file (or Java Archive) aggregates many files into one. Software developers use JARs to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.). Computer users can create or extract JAR files using the `jar` command that comes with a Java Development Kit (JDK).

Java-MATLAB Interface — Known as the *JMI Interface*, this is the Java interface built into MATLAB software.

JDK — The Java Development Kit is a free Oracle® product which provides the environment required for programming in Java.

JMI Interface — see *Java-MATLAB Interface*.

JRE — Java Run-Time Environment is the part of the Java Development Kit (JDK) required to run Java programs. It comprises the Java Virtual Machine, the Java platform core classes, and supporting files. It does not include the compiler, debugger, or other tools present in the JDK™. The JRE™ is the smallest set of executables and files that constitute the standard Java platform.

M

Magic Square — A square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.

MATLAB Runtime — An execution engine made up of the same shared libraries. MATLAB uses these libraries to enable the execution of MATLAB files on systems without an installed version of MATLAB.

MATLAB Runtime singleton — See *Shared MATLAB Runtime instance*.

MATLAB Runtime workers — A MATLAB Runtime session. Using MATLAB Production Server software, you have the option of specifying more than one MATLAB Runtime session, using the `--num-workers` options in the server configurations file.

MATLAB Production Server Client — In the MATLAB Production Server software, clients are applications written in a language supported by MATLAB Production Server that call deployed functions hosted on a server.

MATLAB Production Server Configuration — An instance of the MATLAB Production Server containing at least one server and one client. Each configuration of the software usually contains a unique set of values in the server configuration file, `main_config` (MATLAB Production Server).

MATLAB Production Server Server Instance — A logical server configuration created using the `mps-new` command in MATLAB Production Server software.

MATLAB Production Server Software — Product for server/client deployment of MATLAB programs within your production systems, enabling you to incorporate numerical analytics in enterprise applications. When you use this software, web, database, and enterprise applications connect to MATLAB programs running on MATLAB Production Server via a lightweight client library, isolating the MATLAB programs from your production system. MATLAB Production Server software consists of one or more servers and clients.

Marshaling — See *Data Marshaling*.

mbuild — MATLAB Compiler SDK command that compiles and links C and C++ source files into standalone applications or shared libraries. For more information, see the `mbuild` function reference page.

mcc — The MATLAB command that invokes the compiler. It is the command-line equivalent of using the compiler apps.

Method Attribute — In the context of .NET, a mechanism used to specify declarative information to a .NET class. For example, in the context of client programming with MATLAB Production Server software, you specify method attributes to define MATLAB structures for input and output processing.

mxArray interface — The MATLAB data type containing all MATLAB representations of standard mathematical data types.

MWArray interface — A proxy to `mxAarray`. An application program interface (API) for exchanging data between your application and MATLAB. Using `MWArray`, you marshal data from traditional mathematical types to a form that can be processed and understood by MATLAB data type `mxAarray`. There are different implementations of the `MWArray` proxy for each application programming language.

P

Package — The act of bundling the deployed MATLAB code, along with the MATLAB Runtime and other files, into an installer that can be distributed to others. The compiler apps place the installer in the `for_redistribution` subfolder. In addition to the installer, the compiler apps generate a number of loose artifacts that can be used for testing or building a custom installer.

PID File — See *Process Identification File (PID File)*.

Pool — A pool of threads, in the context of server management using MATLAB Production Server software. Servers created with the software do not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a pool, or group, of available threads. The server configuration file option `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process.

Process Identification File (PID File) — A file that documents informational and error messages relating to a running server instance of MATLAB Production Server software.

Program — A bundle of code that is executed to achieve a purpose. Programs usually are written to automate repetitive operations through computer processing. Enterprise system applications usually consist of hundreds or even thousands of smaller programs.

Properties — For this definition in the context of .NET, see *Fields and Properties*.

Proxy — A software design pattern typically using a class, which functions as an interface to something else. For example, `MWArray` is a proxy for programmers who need to access the underlying type `mxAarray`.

S

Server Instance — See MATLAB Production Server Server Instance.

Shared Library — Groups of files that reside in one space on disk or memory for fast loading into Windows applications. Dynamic-link libraries (DLLs) are Microsoft's implementation of the shared library concept for Microsoft Windows.

Shared MATLAB Runtime instance — When using MATLAB Compiler SDK, you can create a shared MATLAB Runtime instance, also known as a singleton. When you invoke MATLAB Compiler with the `-S` option through the compiler (using either `mcc` or a compiler app), a single MATLAB Runtime instance is created for each COM component or Java package in an application. You reuse this instance by sharing it among all subsequent class instances. Such sharing results in more efficient memory usage and eliminates the MATLAB Runtime startup cost in each subsequent class instantiation. All class instances share a single MATLAB workspace and share global variables in the deployed MATLAB files. MATLAB Compiler SDK creates singletons by default for .NET assemblies. MATLAB Compiler creates singletons by default for the COM components used by the Excel add-ins.

State — The present condition of MATLAB, or the MATLAB Runtime. MATLAB functions often carry state in the form of variable values. The MATLAB workspace itself also maintains information about global variables and path settings. When deploying functions that carry state, you must often take additional steps to ensure state retention when deploying applications that use such functions.

Structs — MATLAB Structures. Structs are MATLAB arrays with elements that you access using textual field designators. Fields are data containers that store data of a specific MATLAB type.

System Compiler — A key part of Interactive Development Environments (IDEs) such as Microsoft Visual Studio®.

T

Thread — A portion of a program that can run independently of and concurrently with other portions of the program. See *pool* for additional information on managing the number of processing threads available to a server instance.

Type-safe interface — An API that minimizes explicit type conversions by hiding the `MWArray` type from the calling application.

W

Web Application Archive (WAR) —In computing, a Web Application Archive is a JAR file used to distribute a collection of JavaServer pages, servlets, Java classes, XML files, tag libraries, and static web pages that together constitute a web application.

Webfigure — A MathWorks representation of a MATLAB figure, rendered on the web. Using the WebFigures feature, you display MATLAB figures on a website for graphical

manipulation by end users. This enables them to use their graphical applications from anywhere on the web, without the need to download MATLAB or other tools that can consume costly resources.

Windows Communication Foundation (WCF) — The Windows Communication Foundation™ is an application programming interface in the .NET Framework for building connected, service-oriented, web-centric applications. WCF is designed in accordance with service oriented architecture principles to support distributed computing where services are consumed by client applications.

Examples

- “Create a C Shared Library with MATLAB Code” on page 2-2
- “Create a C++ Shared Library with MATLAB Code” on page 2-8
- “Create a .NET Assembly with MATLAB Code” on page 2-14
- “Create a Generic COM Component with MATLAB Code” on page 2-22
- “Create a Java Package with MATLAB Code” on page 2-27
- “Create a Python Package with MATLAB Code” on page 2-34

Create a C Shared Library with MATLAB Code

Supported platform: Windows, Linux®, Mac

This example shows how to create a C shared library using a MATLAB function. You can then pass the generated package to the developer who is responsible for integrating it into an application. The target system does not require a licensed copy of MATLAB.

Create Functions in MATLAB

In MATLAB, examine the MATLAB code that you want packaged. For this example, open `addmatrix.m`, `multiplymatrix.m`, and `eigmatrix.m` located in `matlabroot\extern\examples\compilersdk\c_cpp`.

`addmatrix.m`

```
function a = addmatrix(a1, a2)
%ADDMATRIX Add two matrices
%   This function adds the two matrices passed as input. This function is
%   used to demonstrate the functionality of MATLAB Compiler SDK. Refer to the
%   documentation for MATLAB Compiler SDK C/C++ shared library for more information.

% Copyright 2003-2017 The MathWorks, Inc.

a = a1 + a2;
```

At the MATLAB command prompt, enter `addmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

The output is:

```
ans =
     2     8    14
     4    10    16
     6    12    18
```

`multiplymatrix.m`

```
function m = multiplymatrix(a1, a2)
%MULTIPLYMATRIX Multiplies two matrices
%   This function multiplies the two matrices passed as input. This
%   function is used to demonstrate the functionality of MATLAB Compiler SDK.
%   Refer to the MATLAB Compiler SDK C/C++ shared library documentation for more
```

```
% information.  
% Copyright 2003-2016 The MathWorks, Inc.  
m = a1*a2;
```

At the MATLAB command prompt, enter `multiplmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

The output is:

```
ans =  
    30    66   102  
    36    81   126  
    42    96   150
```

eigmatrix.m

```
function e = eigmatrix(a1)  
%EIGMATRIX Returns the eigenvalues of the given matrix  
% This function returns the eigenvalues of the input matrix. This  
% function is used to demonstrate the functionality of MATLAB Compiler SDK.  
% Refer to the MATLAB Compiler SDK C/C++ shared library documentation for more  
% information.  
% Copyright 2003-2016 The MathWorks, Inc.  
  
try  
    %Tries to calculate the eigenvalues and return them.  
    e = eig(a1);  
catch  
    %Returns a -1 on error.  
    e = -1;  
end
```

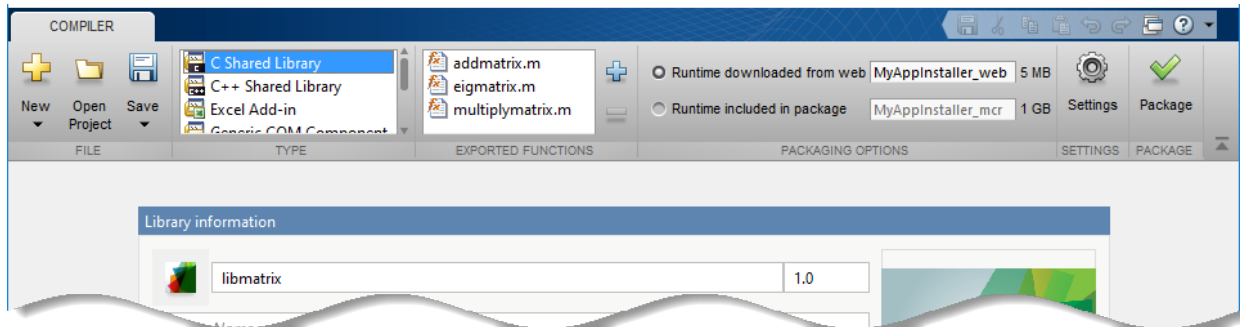
At the MATLAB command prompt, enter `eigmatrix([1 4 7; 2 5 8; 3 6 9])`.

The output is:


```
ans =  
    16.1168  
    -1.1168  
    -0.0000
```

Create C Shared Using Library Compiler App

- 1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**. In the **MATLAB Compiler** project window, click **C Shared Library**.



Alternately, you can open the **Library Compiler** app by entering `libraryCompiler` at the MATLAB prompt.

- 2 In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.
 - 1 In the **Exported Functions** section of the toolstrip, click .
 - 2 In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function is added to the list of main files.

Add all three functions to the list of main files.

- 3 Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:
 - **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.
 - **Runtime included in package** — Generates an application that includes the MATLAB Runtime installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer or obtain a CD if you do not have internet access.

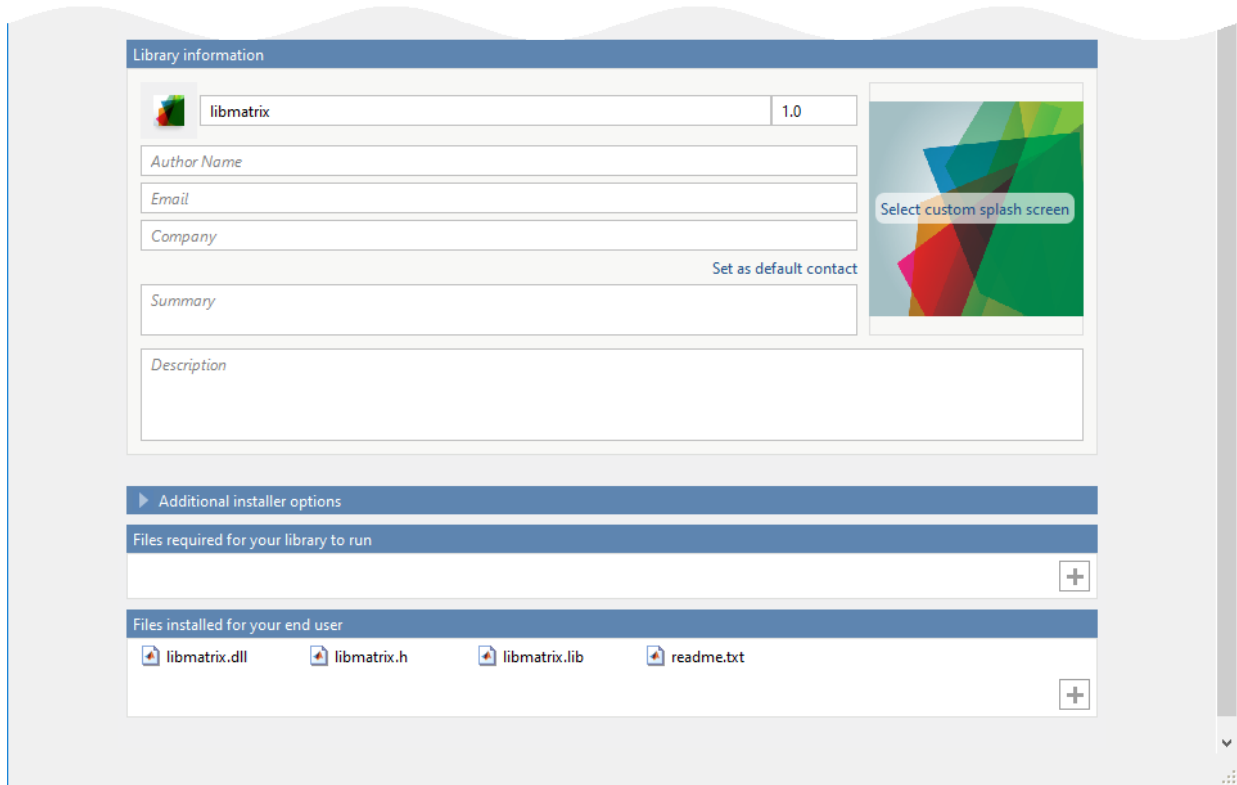
- 4 In the **Library Name** field, rename the packaged shared library as `libmatrix`. The same name is followed through in the implementation of the shared library.

Customize the Application and Its Appearance

You can customize the installer, customize your application, and add more information about the application as follows:

- **Library information** — Editable information about the deployed application. You can also customize the applications appearance by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer” on page 3-2.
- **Additional installer options** — Edit the default installation path for the generated installer and selecting custom logo. See “Change the Installation Path” on page 3-4 .
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project” on page 3-5.
- **Files installed for your end user** — Files that are installed with your application. These files include:
 - Generated `readme.txt`
 - Generated executable for the target platform

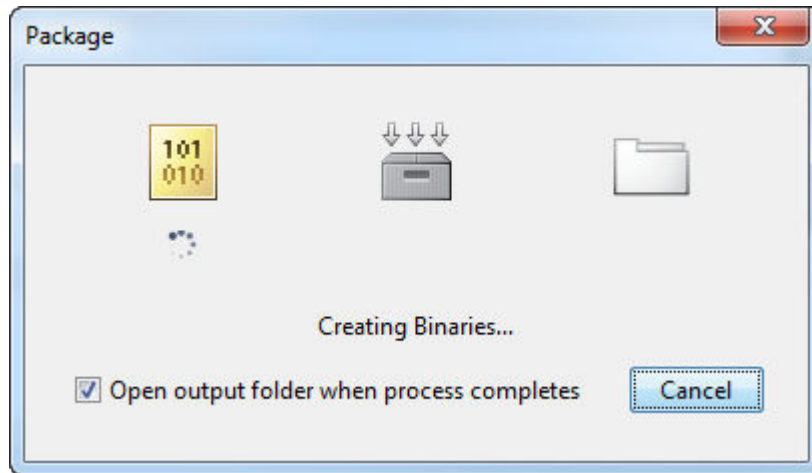
See “Specify Files to Install with Application” on page 3-6



Package the Application

- 1 To generate the packaged application, click **Package**.

In the Save Project dialog box, specify the location to save the project.



- 2 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — Folder containing the file that installs the application and the MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc`, such as binaries and JAR, header, and source files for a specific target. Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler.

See Also

`deploytool` | `libraryCompiler` | `mcc`

Related Examples

- “Integrate a C Shared Library into an Application”

Create a C++ Shared Library with MATLAB Code

Supported platform: Windows, Linux, Mac

This example shows how to create a C++ shared library using a MATLAB function. You can then pass the generated package to the developer who is responsible for integrating it into an application. The target system does not require a licensed copy of MATLAB.

Create Functions in MATLAB

In MATLAB, examine the MATLAB code that you want packaged. For this example, open `addmatrix.m`, `multiplymatrix.m`, and `eigmatrix.m` located in `matlabroot\extern\examples\compilersdk\c_cpp`.

`addmatrix.m`

```
function a = addmatrix(a1, a2)
%ADDMATRIX Add two matrices
%   This function adds the two matrices passed as input. This function is
%   used to demonstrate the functionality of MATLAB Compiler SDK. Refer to the
%   documentation for MATLAB Compiler SDK C/C++ shared library for more information.

% Copyright 2003-2017 The MathWorks, Inc.

a = a1 + a2;
```

At the MATLAB command prompt, enter `addmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

The output is:

```
ans =
     2     8    14
     4    10    16
     6    12    18
```

`multiplymatrix.m`

```
function m = multiplymatrix(a1, a2)
%MULTIPLYMATRIX Multiplies two matrices
%   This function multiplies the two matrices passed as input. This
%   function is used to demonstrate the functionality of MATLAB Compiler SDK.
%   Refer to the MATLAB Compiler SDK C/C++ shared library documentation for more
```

```
% information.  
% Copyright 2003-2016 The MathWorks, Inc.  
m = a1*a2;
```

At the MATLAB command prompt, enter `multiplmatrix([1 4 7; 2 5 8; 3 6 9], [1 4 7; 2 5 8; 3 6 9])`.

The output is:

```
ans =  
    30    66   102  
    36    81   126  
    42    96   150
```

eigmatrix.m

```
function e = eigmatrix(a1)  
%EIGMATRIX Returns the eigenvalues of the given matrix  
% This function returns the eigenvalues of the input matrix. This  
% function is used to demonstrate the functionality of MATLAB Compiler SDK.  
% Refer to the MATLAB Compiler SDK C/C++ shared library documentation for more  
% information.  
% Copyright 2003-2016 The MathWorks, Inc.  
  
try  
    %Tries to calculate the eigenvalues and return them.  
    e = eig(a1);  
catch  
    %Returns a -1 on error.  
    e = -1;  
end
```

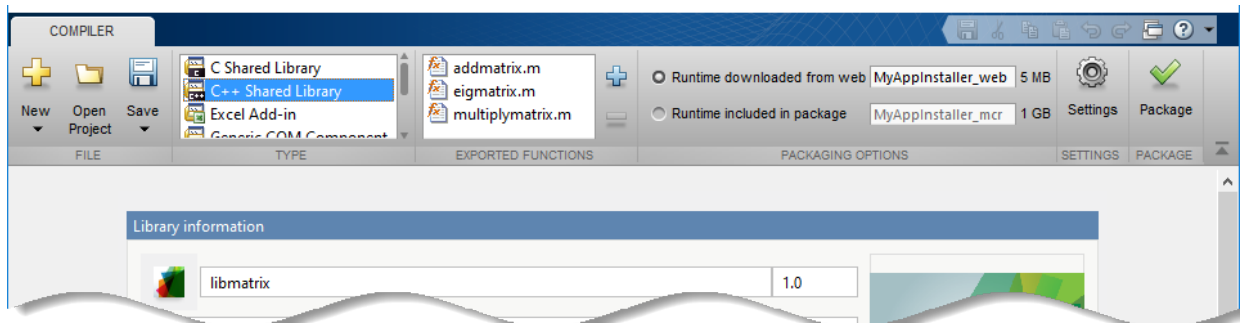
At the MATLAB command prompt, enter `eigmatrix([1 4 7; 2 5 8; 3 6 9])`.

The output is:


```
ans =  
    16.1168  
    -1.1168  
    -0.0000
```

Create a C++ Shared Library using the Library Compiler App

- 1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**. In the **MATLAB Compiler** project window, click **C++ Shared Library**.



Alternately, you can open the **Library Compiler** app by entering `libraryCompiler` at the MATLAB prompt.

- 2 In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.
 - 1 In the **Exported Functions** section of the toolstrip, click .
 - 2 In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function is added to the list of main files.

Add all three functions to the list of main files.

- 3 Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:
 - **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.
 - **Runtime included in package** — Generates an application that includes the MATLAB Runtime installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer or obtain a CD if you do not have internet access.

- 4 In the **Library Name** field, rename the packaged shared library as `libmatrix`. The same name is followed through in the implementation of the shared library.

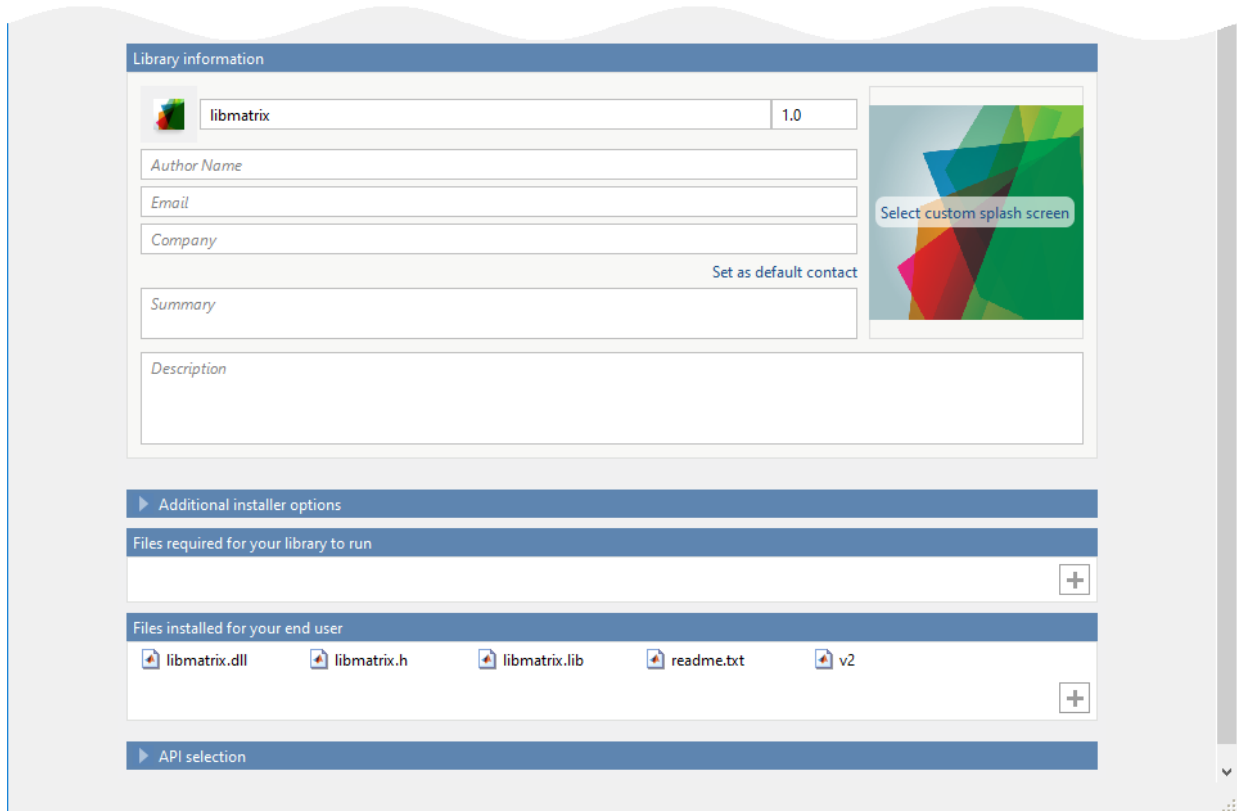
Customize the Application and Its Appearance

You can customize the installer, customize your application, and add more information about the application as follows:

- **Library information** — Editable information about the deployed application. You can also customize the applications appearance by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer” on page 3-2.
- **Additional installer options** — Edit the default installation path for the generated installer and selecting custom logo. See “Change the Installation Path” on page 3-4 .
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project” on page 3-5.
- **Files installed for your end user** — Files that are installed with your application. These files include:
 - Generated `readme.txt`
 - Generated executable for the target platform

See “Specify Files to Install with Application” on page 3-6

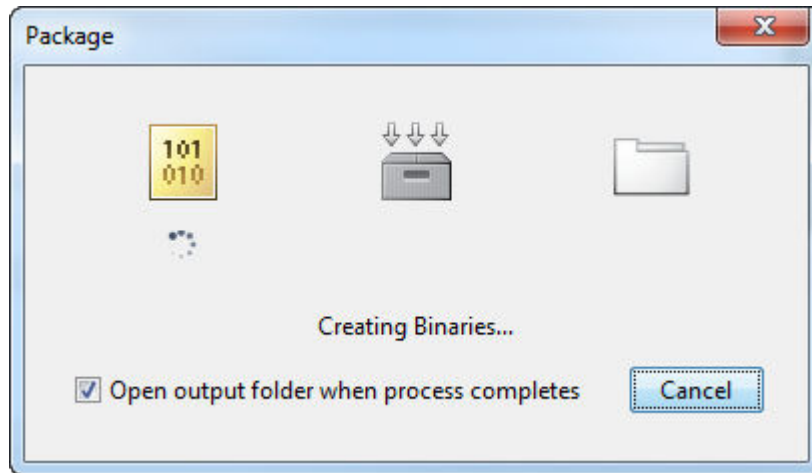
- **API selection** — You can select the type of API for the generated C++ shared libraries. For more information, see “API Selection for C++ Shared Library” on page 3-7.



Package the Application

- 1 To generate the packaged application, click **Package**.

In the Save Project dialog box, specify the location to save the project.



- 2 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — Folder containing the file that installs the application and the MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc`, such as binaries and JAR, header, and source files for a specific target. Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler.

See Also

`deploytool` | `libraryCompiler` | `mcc`

Related Examples

- “Integrate a C++ `mwArray` API Shared Library into an Application”
- “Integrate a C++ MATLAB Data API Shared Library into an Application”

Create a .NET Assembly with MATLAB Code

Supported platform: Windows

This example shows how to create a .NET Assembly using a MATLAB function. It packages a MATLAB function into a .NET assembly, which contains the `MLTestClass` and other files required to deploy your application. You can then pass the generated package to the developer who is responsible for integrating it into an application. The target system does not require a licensed copy of MATLAB.

Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want packaged. For this example, open `makesquare.m` located in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MagicSquareExample\MagicSquareComp`.

```
function y = makesquare(x)
```

```
y = magic(x);
```

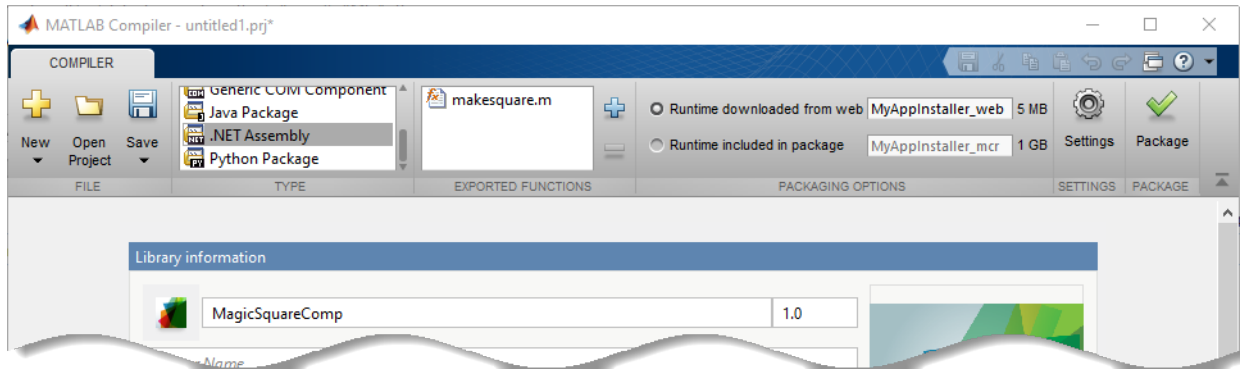
At the MATLAB command prompt, enter `makesquare(5)`.

The output is:


```
17    24     1     8    15
23     5     7    14    16
 4     6    13    20    22
10    12    19    21     3
11    18    25     2     9
```

Create .NET Assembly Using Library Compiler App

- 1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**. In the **MATLAB Compiler** project window, click **.NET Assembly**.



Alternately, you can open the **Library Compiler** app by entering `libraryCompiler` at the MATLAB prompt.

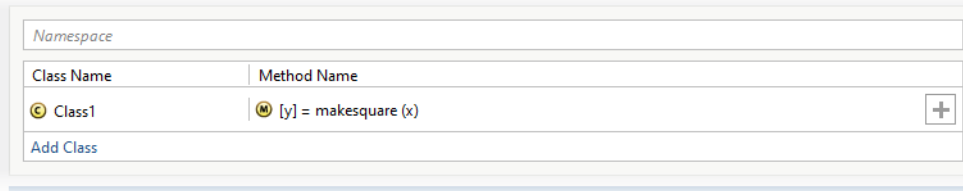
- 2 In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.
 - 1 In the **Exported Functions** section of the toolbar, click .
 - 2 In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function is added to the list of main files.

- 3 Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:
 - **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.
 - **Runtime included in package** — Generates an application that includes the MATLAB Runtime installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer or obtain a CD if you do not have internet access.

- 4 In the **Library Name** field, replace `makesquare` with `MagicSquareComp`.
- 5 Verify that the function defined in `makesquare.m` is mapped into `Class1`.



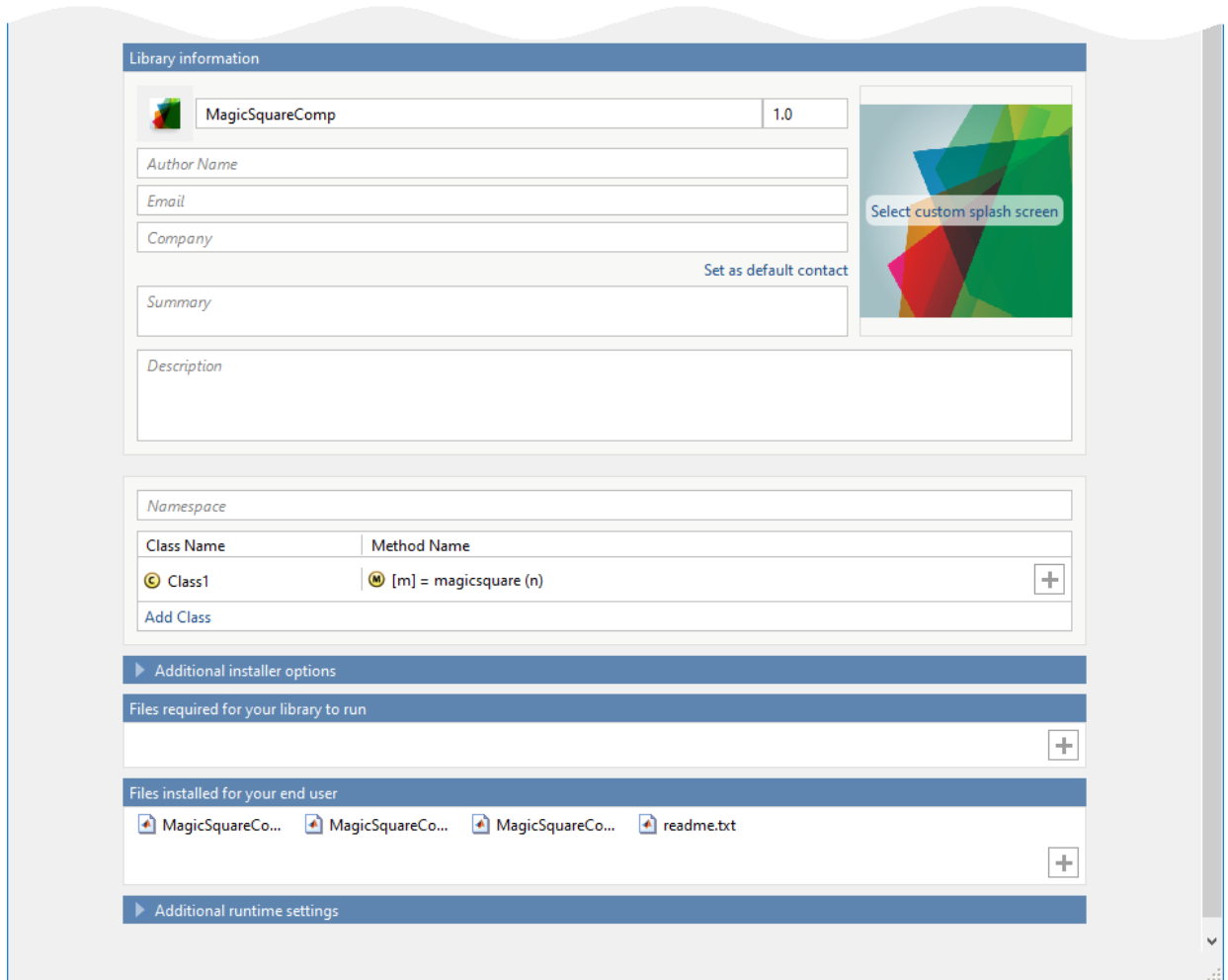
Customize the Application and Its Appearance

You can customize the installer, customize your application, and add more information about the application as follows:

- **Library information** — Editable information about the deployed application. You can also customize the applications appearance by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer” on page 3-2.
- **Additional installer options** — Edit the default installation path for the generated installer and selecting custom logo. See “Change the Installation Path” on page 3-4 .
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project” on page 3-5.
- **Files installed for your end user** — Files that are installed with your application. These files include:
 - Generated readme.txt
 - Generated executable for the target platform

See “Specify Files to Install with Application” on page 3-6

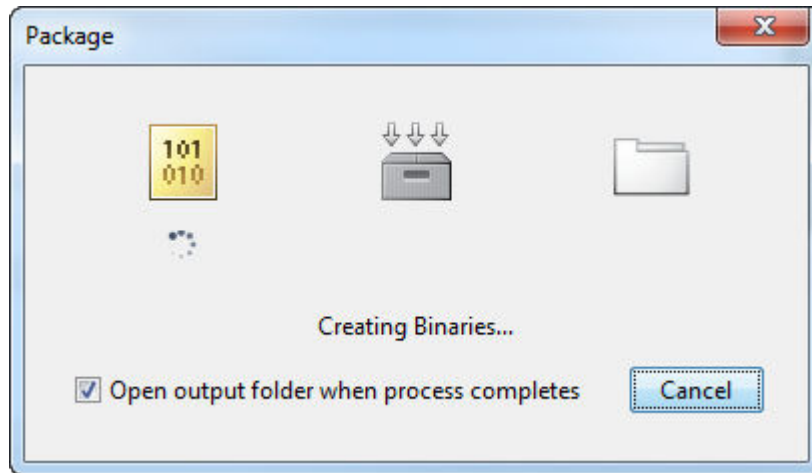
- **Additional runtime settings** — Platform-specific options for controlling the generated executable. See “Additional Runtime Settings” on page 3-6.



Package the Application

- 1 To generate the packaged application, click **Package**.

In the Save Project dialog box, specify the location to save the project.



- 2 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — Folder containing the file that installs the application and the MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc`, such as binaries and JAR, header, and source files for a specific target. Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler.

Build a .NET Application

- 1 Install the .NET assembly from the `for_redistribution` folder.

The generated shared libraries and support files are located in the `for_testing` folder.

- 2 Open Microsoft Visual Studio and create a project. For this example, create a C# Console Application called **MainApp**, and create a reference to your assembly file `MagicSquareComp.dll`.

Ensure that the assembly is located in the application folder created where you installed the component.

- 3 Create a reference to the `MWArray` API. The location of the API within MATLAB Runtime is:

```
matlabrootMATLAB Runtime\v##\toolbox\dotnetbuilder\bin\arch  
\version\MWArray.dll
```

- 4 Go to **Build > Configuration Manager**, and change the platform from **Any CPU** to **x64**.
- 5 Copy the following C# code into the project, and save it.

C# Code to Implement Application

```
// Make .NET Namespaces available to your generated component.
using System;

using MagicSquareComp;
using MathWorks.MATLAB.NET.Arrays;

// Initialize your classes before you use them.
namespace MainApp
{
    class Program
    {
        static void Main(string[] args)
        {
            MLTestClass obj = null;
            MWNumericArray input = null;
            MWNumericArray output = null;
            MWArray[] result = null;

            // Because class instantiation and method invocation make their exceptions
            // you should enclose your code in a try-catch block to handle errors.
            try
            {
                // Instantiate your component class.
                obj = new MLTestClass();

                // Invoke your component.
                input = 5;
                result = obj.makesquare(1, input);

                // Extract the Magic Square you created from the first index of result
                output = (MWNumericArray)result[0];

                // print the output.
                Console.WriteLine(output);
            }
            catch
            {
                throw;
            }
        }
    }
}
```

```
}  
}
```

- 6 After you finish writing your code, build and run it with Microsoft Visual Studio.

See Also

`deploytool` | `libraryCompiler` | `mcc`

More About

- “Integrate a .NET Assembly Into a C# Application”

Create a Generic COM Component with MATLAB Code

Supported platform: Windows

This example shows how to create a generic COM component using a MATLAB function. You can then pass the generated package to the developer who is responsible for integrating it into an application. The target system does not require a licensed copy of MATLAB.

Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want packaged. For this example, open `makesquare.m` located in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\COM\MagicSquareExample\MagicSquareComp`.

```
function y = makesquare(x)

y = magic(x);
```

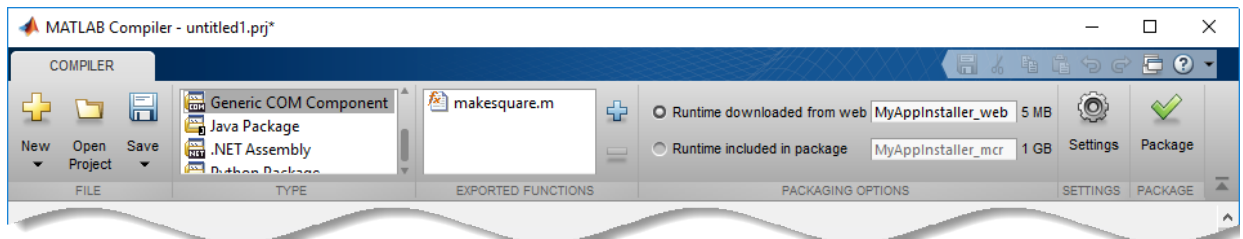
At the MATLAB command prompt, enter `makesquare(5)`.

The output is:

```
17    24     1     8    15
23     5     7    14    16
 4     6    13    20    22
10    12    19    21     3
11    18    25     2     9
```

Create Generic COM Component Using Library Compiler App

- 1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**. In the **MATLAB Compiler** project window, click **Generic COM Component**.



Alternately, you can open the **Library Compiler** app by entering `libraryCompiler` at the MATLAB prompt.

- 2 In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

- 1 In the **Exported Functions** section of the toolstrip, click .

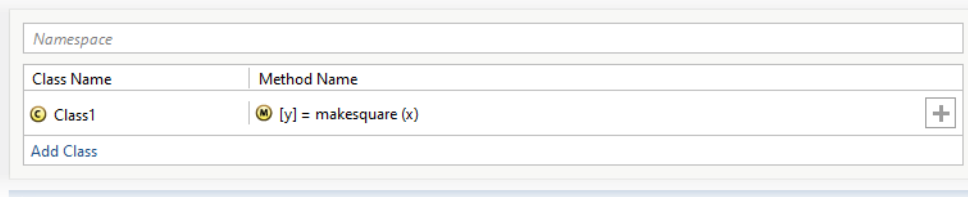
- 2 In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function is added to the list of main files.

- 3 Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:
 - **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.
 - **Runtime included in package** — Generates an application that includes the MATLAB Runtime installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer or obtain a CD if you do not have internet access.

- 4 In the **Library Name** field, replace `makesquare` with `MagicSquareComp`.
- 5 Verify that the function defined in `makesquare.m` is mapped into `Class1`.



Customize the Application and Its Appearance

You can customize the installer, customize your application, and add more information about the application as follows:

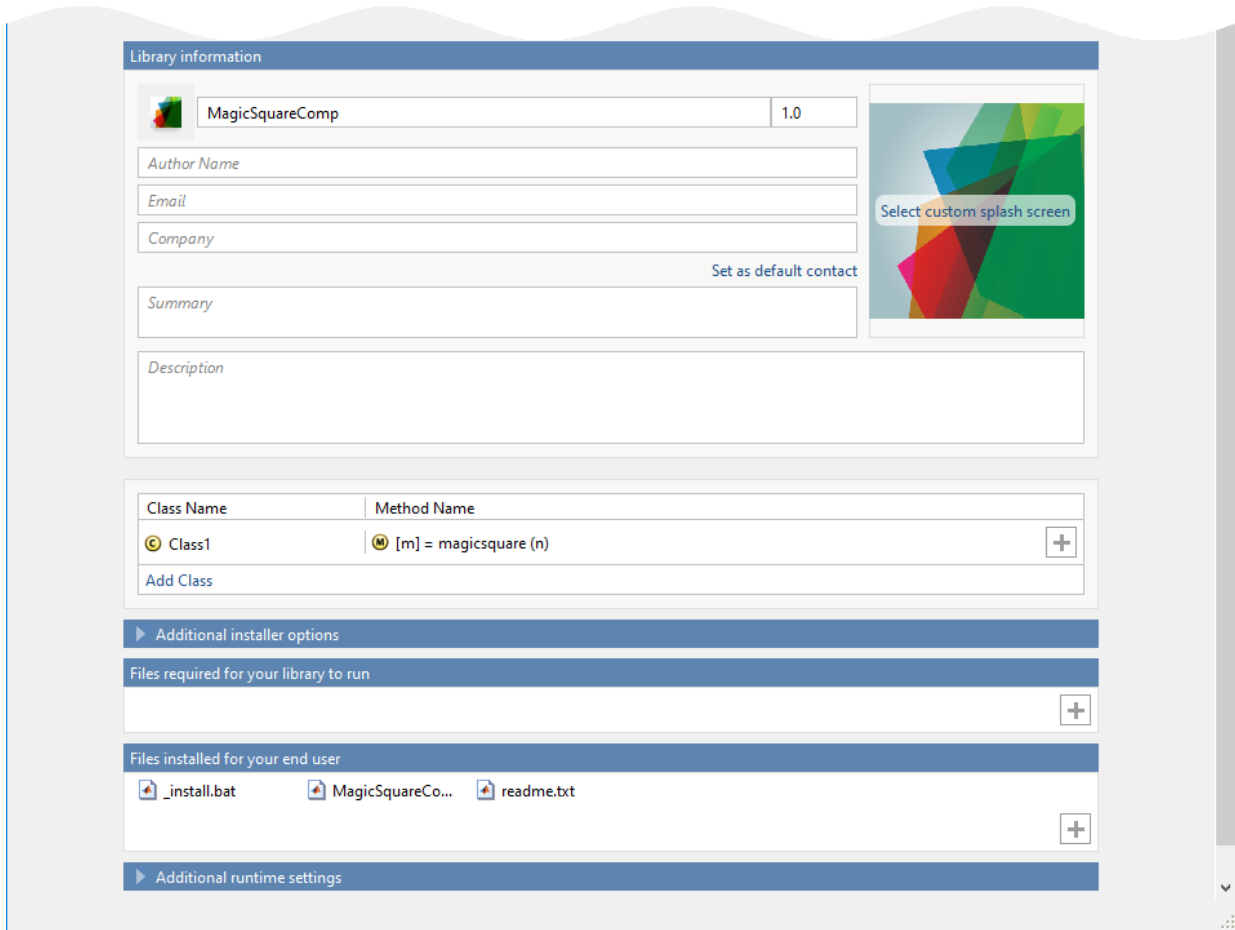
- **Library information** — Editable information about the deployed application. You can also customize the applications appearance by changing the application icon and

splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer” on page 3-2.

- **Additional installer options** — Edit the default installation path for the generated installer and selecting custom logo. See “Change the Installation Path” on page 3-4 .
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project” on page 3-5.
- **Files installed for your end user** — Files that are installed with your application. These files include:
 - Generated `readme.txt`
 - Generated executable for the target platform

See “Specify Files to Install with Application” on page 3-6

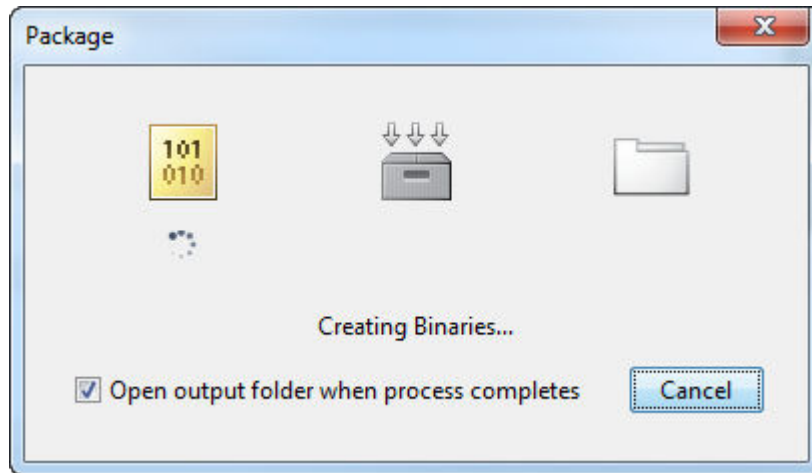
- **Additional runtime settings** — Platform-specific options for controlling the generated executable. See “Additional Runtime Settings” on page 3-6.



Package the Application

- 1 To generate the packaged application, click **Package**.

In the Save Project dialog box, specify the location to save the project.



- 2 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — Folder containing the file that installs the application and the MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc`, such as binaries and JAR, header, and source files for a specific target. Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler.

See Also

`deploytool` | `libraryCompiler` | `mcc`

More About

- “Call COM Objects in Visual C++ Programs”

Create a Java Package with MATLAB Code

Supported platform: Windows, Linux, Mac

This example shows how to create a Java package using a MATLAB function. You can then pass the generated package to the developer who is responsible for integrating it into an application. The target system does not require a licensed copy of MATLAB.

Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want packaged. For this example, open `makesqr.m` located in `matlabroot\toolbox\javabuilder\Examples\MagicSquareExample\MagicDemoComp`.

```
function y = makesqr(x)

y = magic(x);
```

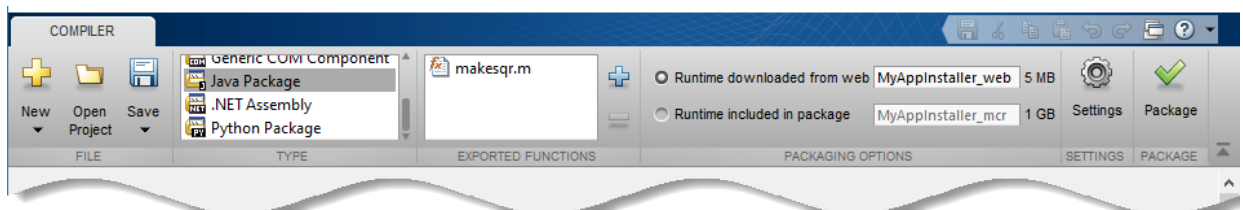
At the MATLAB command prompt, enter `makesqr(5)`.

The output is:

```
17    24     1     8    15
23     5     7    14    16
 4     6    13    20    22
10    12    19    21     3
11    18    25     2     9
```


Create Java Application Using Library Compiler App

- 1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**. In the **MATLAB Compiler** project window, click **Java Package**.



Alternately, you can open the **Library Compiler** app by entering `libraryCompiler` at the MATLAB prompt.

- 2 In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

- 1 In the **Exported Functions** section of the toolstrip, click .

- 2 In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function is added to the list of main files.

- 3 Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:
 - **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.
 - **Runtime included in package** — Generates an application that includes the MATLAB Runtime installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer or obtain a CD if you do not have internet access.

- 4 Verify that the function defined in `makesqr.m` is mapped into `Class1`.



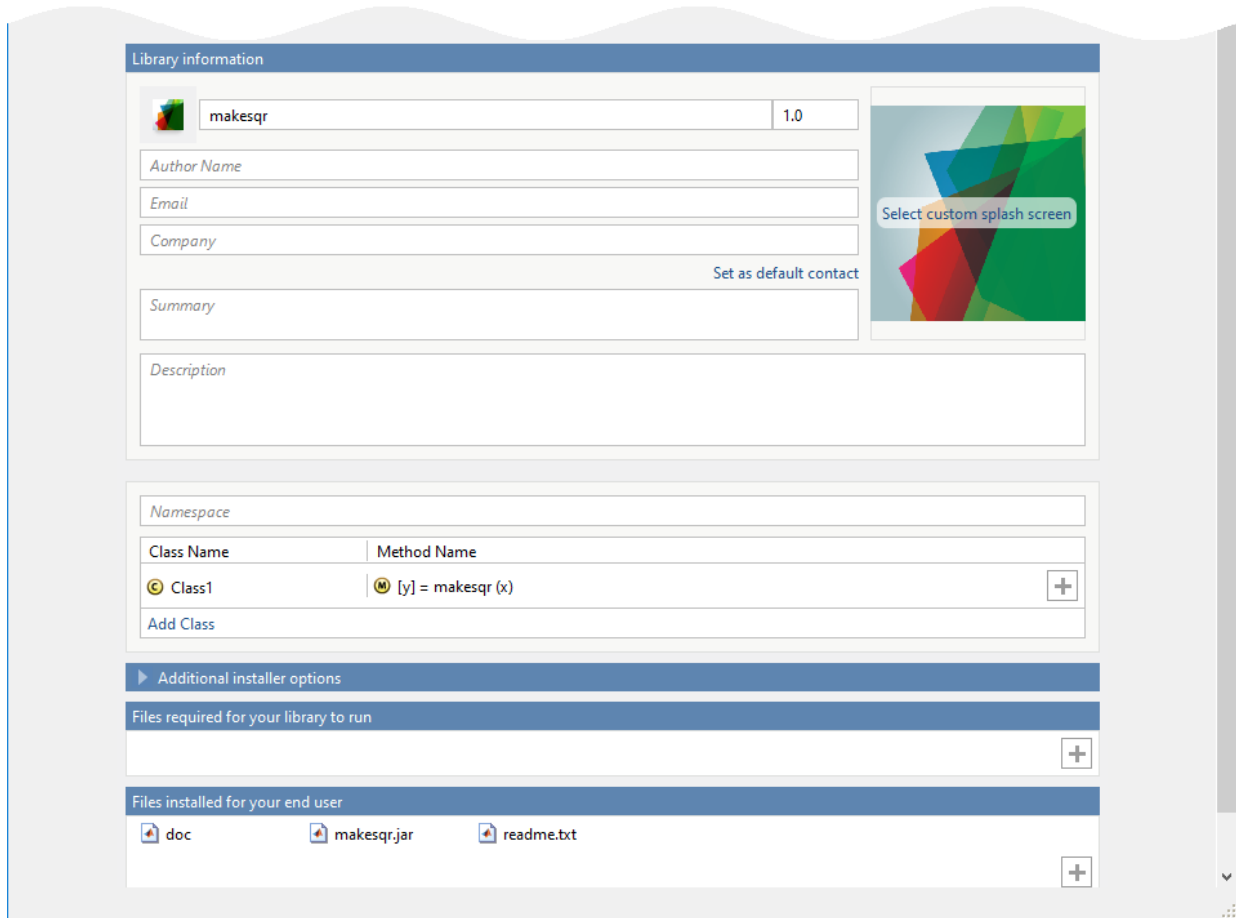
Customize the Application and Its Appearance

You can customize the installer, customize your application, and add more information about the application as follows:

- **Library information** — Editable information about the deployed application. You can also customize the applications appearance by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer” on page 3-2.

- **Additional installer options** — Edit the default installation path for the generated installer and selecting custom logo. See “Change the Installation Path” on page 3-4 .
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project” on page 3-5.
- **Files installed for your end user** — Files that are installed with your application. These files include:
 - Generated `readme.txt`
 - Generated executable for the target platform

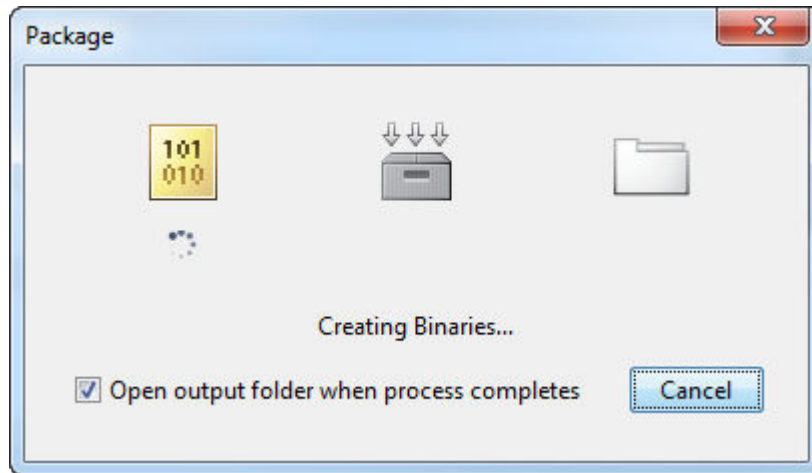
See “Specify Files to Install with Application” on page 3-6



Package the Application

- 1 To generate the packaged application, click **Package**.

In the Save Project dialog box, specify the location to save the project.



- 2 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — Folder containing the file that installs the application and the MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc`, such as binaries and JAR, header, and source files for a specific target. Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler.

Install and Implement MATLAB Generated Java Application

- 1 To install the Java application, in the `for_redistribution` folder, double-click `MyAppInstaller_web` file.
- 2 After the installation is complete, locate the folder containing the generated JAR files, and create a file called `getmagic.java`.
- 3 Open `getmagic.java` using a text editor, and paste the following code into the file.

Java Code to Implement Application

```
import com.mathworks.toolbox.javabuilder.*;
import makesqr.*;

class getmagic
{
    public static void main(String[] args)
    {
        MWNumericArray n = null;
        Object[] result = null;
        Class1 theMagic = null;

        if (args.length == 0)
        {
            System.out.println("Error: must input a positive integer");
            return;
        }

        try
        {
            n = new MWNumericArray(Double.valueOf(args[0]),
                                   MWClassID.DOUBLE);

            theMagic = new Class1();

            result = theMagic.makesqr(1, n);
            System.out.println(result[0]);
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e.toString());
        }
        finally
        {
            MWArray.disposeArray(n);
            MWArray.disposeArray(result);
            theMagic.dispose();
        }
    }
}
```

- 4** Compile the Java application using javac.

```
javac -classpath "mcrroot\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr.jar .\
```

Note On UNIX® platforms, use colon (:) as the class path delimiter instead of semicolon (;).

mcrroot is the path to the MATLAB Runtime installation on your system. If you have MATLAB installed on your system instead, you can use the path to your MATLAB installation.

- 5 From the system command prompt, run the application.

```
java -classpath .;"mcrroot\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr.jar g
  17   24   1   8   15
  23   5   7  14  16
   4   6  13  20  22
  10  12  19  21   3
  11  18  25   2   9
```

Place a dot (.) in the first position of the class path. If it is not there, you get a message stating that Java cannot load the class.

Note On UNIX platforms, use colon (:) as the class path delimiter instead of semicolon (;).

See Also

deploytool | libraryCompiler | mcc

Create a Python Package with MATLAB Code

Supported platform: Windows, Linux, Mac

This example shows how to create a Python package using a MATLAB function. You can then pass the generated package to the developer who is responsible for integrating it into an application. The target system does not require a licensed copy of MATLAB.

Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want packaged. For this example, write a function `makesqr.m` as follows:

```
function y = makesqr(x)

y = magic(x);
```

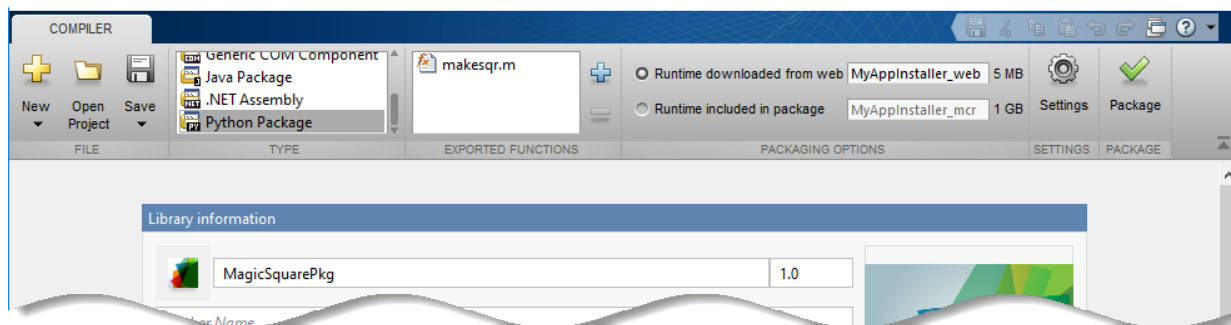
At the MATLAB command prompt, enter `makesqr(5)`.

The output is:


```
17    24     1     8    15
23     5     7    14    16
 4     6    13    20    22
10    12    19    21     3
11    18    25     2     9
```

Create Python Application Using Library Compiler App

- 1 On the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Library Compiler**. In the **MATLAB Compiler** project window, click **Python Package**.



Alternately, you can open the **Library Compiler** app by entering `libraryCompiler` at the MATLAB prompt.

- 2 In the **MATLAB Compiler** project window, specify the main file of the MATLAB application that you want to deploy.
 - 1 In the **Exported Functions** section of the toolbar, click .
 - 2 In the **Add Files** window, browse to the example folder, and select the function you want to package. Click **Open**.

The function is added to the list of main files.

- 3 Decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the two options in the **Packaging Options** section:
 - **Runtime downloaded from web** — Generates an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application.
 - **Runtime included in package** — Generates an application that includes the MATLAB Runtime installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer or obtain a CD if you do not have internet access.

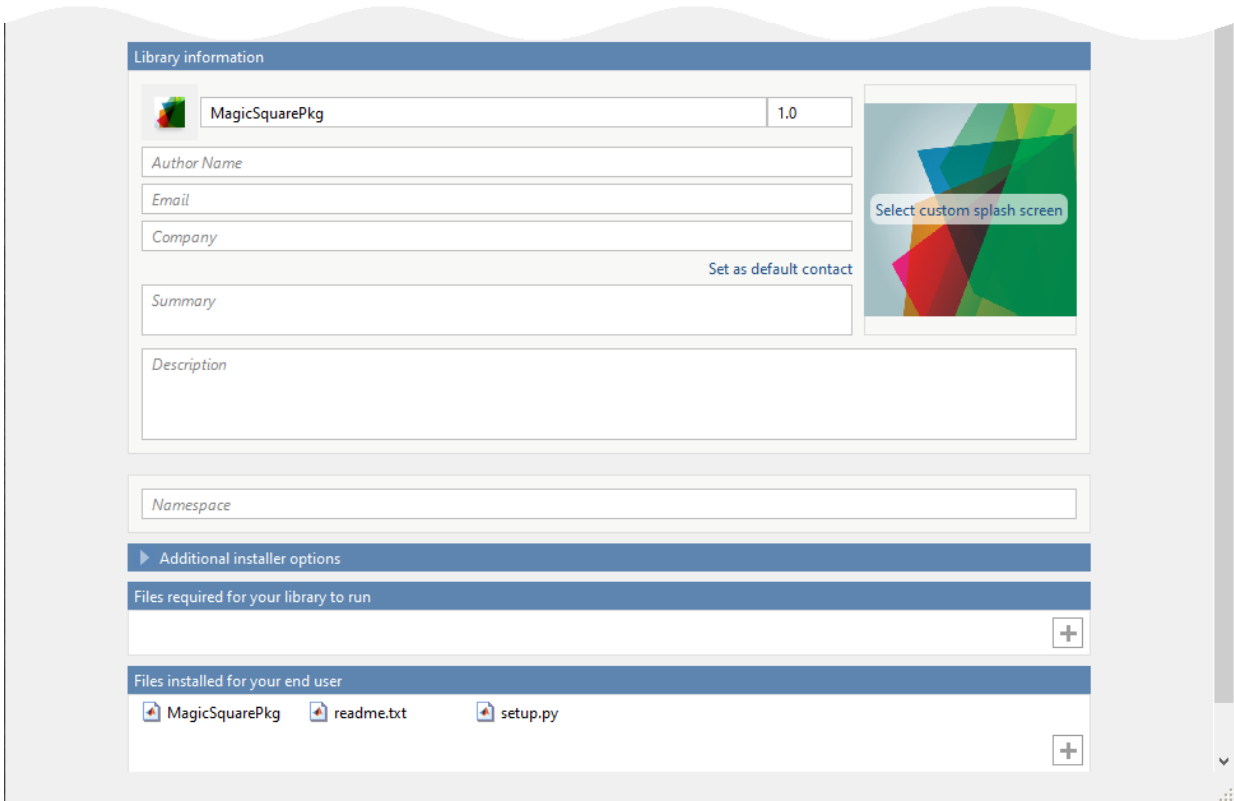
- 4 In the **Library Name** field, replace `makesqr` with `MagicSquarePkg`. For more information on naming requirements for the Python package, see “Import Compiled Python Packages”.

Customize the Application and Its Appearance

You can customize the installer, customize your application, and add more information about the application as follows:

- **Library information** — Editable information about the deployed application. You can also customize the applications appearance by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See “Customize the Installer” on page 3-2.
- **Additional installer options** — Edit the default installation path for the generated installer and selecting custom logo. See “Change the Installation Path” on page 3-4 .
- **Files required for your library to run** — Additional files required by the generated application to run. These files are included in the generated application installer. See “Manage Required Files in Compiler Project” on page 3-5.
- **Files installed for your end user** — Files that are installed with your application. These files include:
 - Generated `readme.txt`
 - Generated executable for the target platform

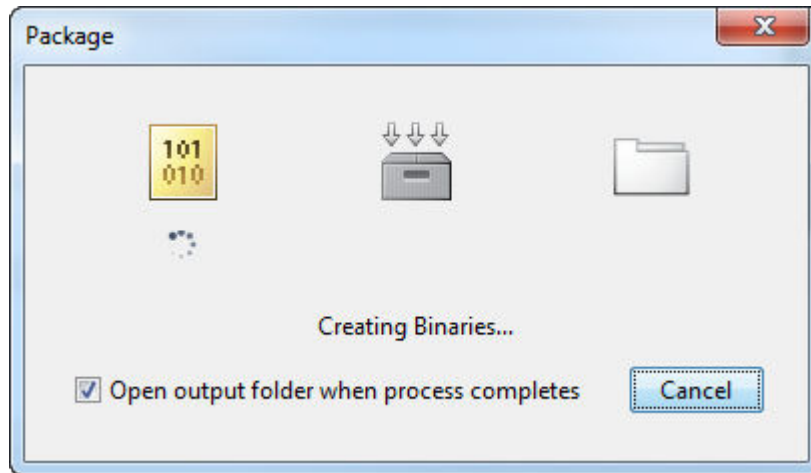
See “Specify Files to Install with Application” on page 3-6



Package the Application

- 1 To generate the packaged application, click **Package**.

In the Save Project dialog box, specify the location to save the project.



- 2 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

When the deployment process is complete, examine the generated output.

- `for_redistribution` — Folder containing the file that installs the application and the MATLAB Runtime.
- `for_testing` — Folder containing all the artifacts created by `mcc`, such as binaries and JAR, header, and source files for a specific target. Use these files to test the installation.
- `for_redistribution_files_only` — Folder containing the files required for redistributing the application. Distribute these files to users who have MATLAB or MATLAB Runtime installed on their machines.
- `PackagingLog.txt` — Log file generated by MATLAB Compiler.

Install and Run MATLAB Generated Python Application

- 1 To install the Python application, open a command prompt in the `for_redistribution_files_only` folder and run the script:


```
python setup.py install
```
- 2 Create a file called `getmagic.py`, and include the following code in it.

```
import MagicSquarePkg
myMagic = MagicSquarePkg.initialize()
```



```
print(myMagic.makesqr(3))  
myMagic.terminate()
```

- 3 At the system command prompt, navigate to the folder containing `getmagic.py` and run the application as follows:

```
python getmagic.py
```

The following output is displayed:

```
[[8.0,1.0,6.0],  
 [3.0,5.0,7.0],  
 [4.0,9.0,2.0]]
```

Note On Mac OS X, you must use the `mwpython` script. The `mwpython` script is located in the `matlabroot/bin` folder. `matlabroot` is the location of your MATLAB installation.

For example, `mwpython getmagic.py`

See Also

`deploytool` | `libraryCompiler` | `mcc` | `mwpython`

Customizing a Compiler Project

- “Customize the Application” on page 3-2
- “Manage Support Packages” on page 3-9

Customize the Application

You can customize the application in several ways: customize the installer, manage files in the project, or add a custom installer path using the **Application Compiler** app or the **Library Compiler** app.

Customize the Installer

Change Application Icon

To change the default icon, click the graphic to the left of the **Library name** or **Application name** field to preview the icon.



Click **Select icon**, and locate the graphic file to use as the application icon. Select the **Use mask** option to fill any blank spaces around the icon with white or the **Use border** option to add a border around the icon.

To return to the main window, click **Save and Use**.

Add Library or Application Information

You can provide further information about your application as follows:

- **Library/Application Name:** The name of the installed MATLAB artifacts. For example, if the name is `foo`, the installed executable is `foo.exe`, and the Windows start menu entry is **foo**. The folder created for the application is `InstallRoot/foo`.

The default value is the name of the first function listed in the **Main File(s)** field of the app.

- **Version:** The default value is 1.0.
- **Author name:** Name of the developer.

- Support email address: Email address to use for contact information.
- Company name: The full installation path for the installed MATLAB artifacts. For example, if the company name is *bar*, the full installation path would be *InstallRoot/bar/ApplicationName*.
- Summary: Brief summary describing the application.
- Description: Detailed explanation about the application.

All information is optional and, unless otherwise stated, is only displayed on the first page of the installer. On Windows systems, this information is also displayed in the Windows **Add/Remove Programs** control panel.

Change the Splash Screen

The installer splash screen displays after the installer has started. It is displayed along with a status bar while the installer initializes.

You can change the default image by clicking the **Select custom splash screen**. When the file explorer opens, locate and select a new image.

You can drag and drop a custom image onto the default splash screen.

Change the Installation Path

This table lists the default path the installer uses when installing the packaged binaries onto a target system.

Windows	C:\Program Files\ <i>companyName</i> \ <i>appName</i>
Mac OS X	/Applications/ <i>companyName</i> / <i>appName</i>
Linux	/usr/ <i>companyName</i> / <i>appName</i>

You can change the default installation path by editing the **Default installation folder** field under **Additional installer options**.



A text field specifying the path appended to the root folder is your installation folder. You can pick the root folder for the application installation folder. This table lists the optional custom root folders for each platform:

Windows	C:\Users\ <i>userName</i> \AppData
Linux	/usr/local

Change the Logo

The logo displays after the installer has started. It is displayed on the right side of the installer.

You change the default image in **Additional Installer Options** by clicking **Select custom logo**. When the file explorer opens, locate and select a new image. You can drag and drop a custom image onto the default logo.

Edit the Installation Notes

Installation notes are displayed once the installer has successfully installed the packaged files on the target system. They provide useful information concerning any additional setup that is required to use the installed binaries and instructions for how to run the application.

Manage Required Files in Compiler Project

The compiler uses a dependency analysis function to automatically determine what additional MATLAB files are required for the application to package and run. These files are automatically packaged into the generated binary. The compiler does not generate any wrapper code that allows direct access to the functions defined by the required files.

If you are using one of the compiler apps, the required files discovered by the dependency analysis function are listed in the **Files required for your application to run** or **Files required for your library to run** field.

To add files, click the plus button in the field, and select the file from the file explorer. To remove files, select the files, and press the **Delete** key.

Caution Removing files from the list of required files may cause your application to not package or not to run properly when deployed.

Using mcc


If you are using `mcc` to package your MATLAB code, the compiler does not display a list of required files before running. Instead, it packages all the required files that are discovered by the dependency analysis function and adds them to the generated binary file.

You can add files to the list by passing one or more `-a` arguments to `mcc`. The `-a` arguments add the specified files to the list of files to be added into the generated binary. For example, `-a hello.m` adds the file `hello.m` to the list of required files and `-a ./foo` adds all the files in `foo` and its subfolders to the list of required files.

Specify Files to Install with Application

The compiler packages files to install along with the ones it generates. By default the installer includes a readme file with instructions on installing the MATLAB Runtime and configuring it.

These files are listed in the **Files installed for your end user** section of the app.

To add files to the list, click , and select the file from the file explorer.

JAR files are added to the application class path as if you had called `javaaddpath`.

To remove files, select the files and press the **Delete** key.

Caution Removing the binary targets from the list results in an installer that does not install the intended functionality.

When installed on a target computer, the files listed in the **Files installed for your end user** are saved in the application folder.

Additional Runtime Settings

Type of Packaged Application	Description	Additional Runtime Settings Options
Generic COM Components	<ul style="list-style-type: none"> • Register the resulting component for you only on the development machine — This option exclusively registers the packaged component for one user on the development machine. 	<div data-bbox="795 1121 1337 1312"> <p>▼ Additional runtime settings</p> <p><input type="checkbox"/> Register the resulting component for you only on the de</p> </div>

Type of Packaged Application	Description	Additional Runtime Settings Options
.NET Assembly	<ul style="list-style-type: none"> • Create Shared Assembly — Enables sharing MATLAB Runtime installer instances for multiple .NET assemblies. • Enable .NET Remoting — Enables you to remotely access MATLAB functionality, as a part of a distributed system. For more information, see “Create a Remotable .NET Assembly”. • Enable Type Safe API — Enables the type safe API for the packaged .NET assembly. 	<p>▼ Additional runtime settings</p> <p>What .NET versions are supported?</p> <p>Assembly Type</p> <p><input type="checkbox"/> Create Shared Assembly</p> <p><input type="checkbox"/> Enable .NET Remoting</p> <p>Type Safe API</p> <p><input type="checkbox"/> Enable type safe API</p>

API Selection for C++ Shared Library

▼ API selection

C++ Shared Library API

Create all interfaces

Create interface that uses the mxArray API

Create interface that uses the MATLAB Data API

- **Create all interfaces** — Creates interfaces for shared libraries using both the `mwArray` API and the MATLAB Data API .
- **Create interface that uses the `mwArray` API** — Creates an interface for a shared library using the `mwArray` API. The interface uses C-style functions to initialize the MATLAB Runtime, load the compiled MATLAB functions into the MATLAB Runtime, and manage data that is passed between the C++ code and the MATLAB Runtime. The interface only supports C++03 functionality. For more information, see “Integrate a C++ `mwArray` API Shared Library into an Application”.
- **Create interface that uses the MATLAB Data API** — Creates an interface for a shared library using MATLAB Data API. It uses a generic interface that has modern C++ semantics. The interface supports C++11 functionality. For more information, see “Integrate a C++ MATLAB Data API Shared Library into an Application”.

See Also

`applicationCompiler` | `libraryCompiler`

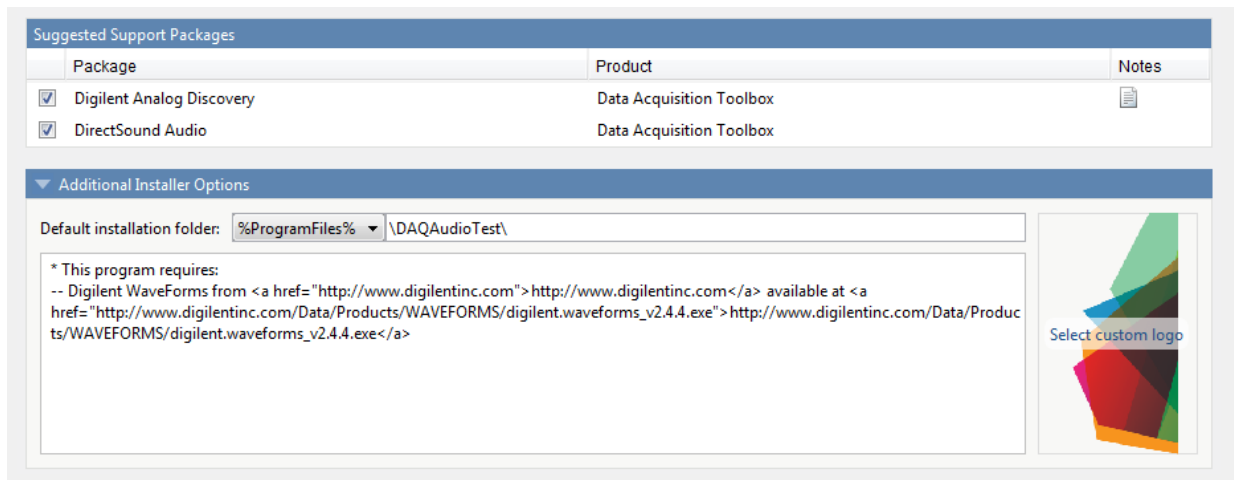
More About

- “Create a C Shared Library with MATLAB Code” on page 2-2
- “Create a C++ Shared Library with MATLAB Code” on page 2-8
- “Create a .NET Assembly with MATLAB Code” on page 2-14
- “Create a Generic COM Component with MATLAB Code” on page 2-22
- “Create a Java Package with MATLAB Code” on page 2-27
- “Create a Python Package with MATLAB Code” on page 2-34

Manage Support Packages

Using a Compiler App

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, the app displays a **Suggested Support Packages** section.



The list displays all installed support packages that your MATLAB code requires. The list is determined using these criteria:

- the support package is installed
- your code has a direct dependency on the support package
- your code is dependent on the base product of the support package
- your code is dependent on at least one of the files listed as a dependency in the `mcc.xml` file of the support package, and the base product of the support package is MATLAB

Deselect support packages that are not required by your application.

Some support packages require third-party drivers that the compiler cannot package. In this case, the compiler adds the information to the installation notes. You can edit

installation notes in the **Additional Installer Options** section of the app. To remove the installation note text, deselect the support package with the third-party dependency.

Caution Any text you enter beneath the generated text will be lost if you deselect the support package.

Using the Command Line

Many MATLAB toolboxes use support packages to interact with hardware or to provide additional processing capabilities. If your MATLAB code uses a toolbox with an installed support package, use the `-a` flag with `mcc` command when packaging your MATLAB code to specify supporting files in the support package folder. For example, if your function uses the `OS Generic Video Interface` support package, run the following command:

```
mcc -m -v test.m -a C:\MATLAB\SupportPackages\R2016b\toolbox\daq\supportpackages\daqau
```

Some support packages require third-party drivers that the compiler cannot package. In this case, you are responsible for downloading and installing the required drivers.

Using MATLAB Production Server

- “Create a Deployable Archive for MATLAB Production Server” on page 4-2
- “Create a C# Client” on page 4-5
- “Create a Java Client” on page 4-9
- “Create a C++ Client” on page 4-13
- “Create a Python Client” on page 4-19

Create a Deployable Archive for MATLAB Production Server

This example shows how to create a deployable archive for MATLAB Production Server using a MATLAB function. You can then hand the generated archive to a system administrator who will deploy it into MATLAB Production Server.

To create a deployable archive:

- 1 Create a MATLAB function `addmatrix.m` that adds two matrices.

```
function a = addmatrix(a1, a2)
```

```
a = a1 + a2;
```

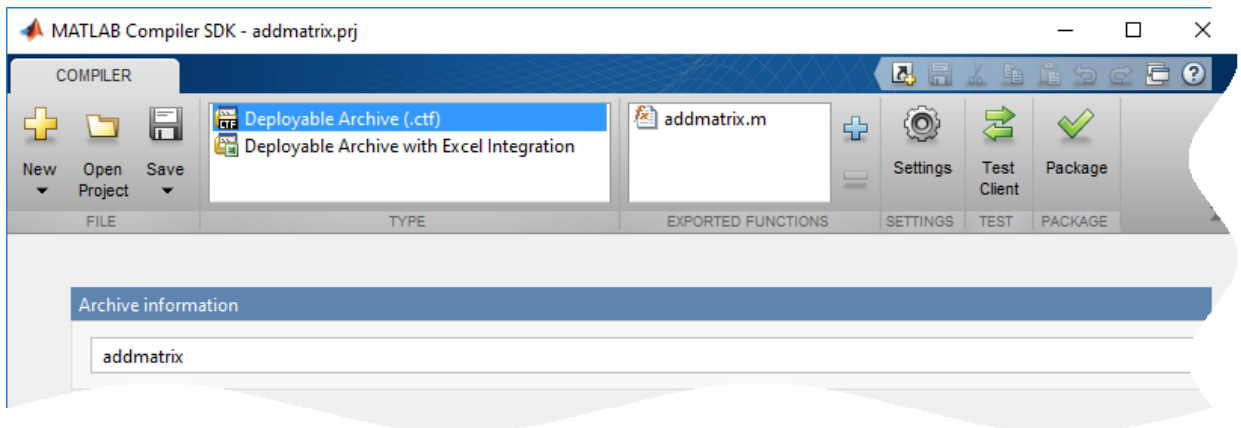
At the MATLAB command prompt, enter `addmatrix(1,2)`.

The output appears as follows:

```
ans =
```

```
3
```

- 2 Open the **Production Server Compiler** app.
 - a On the toolstrip, select the **Apps** tab.
 - b Click the arrow on the far right of the tab to open the apps gallery.
 - c Click **Production Server Compiler**.



- 3 In the **Application Type** section of the toolstrip, select **Deployable Archive** from the list.

Note If the **Application Type** section of the toolstrip is collapsed, you can expand it by clicking the down arrow .

- 4 Specify the MATLAB functions you want to deploy.
 - a In the **Exported Functions** section of the toolstrip, click the plus button.

Note If the **Exported Functions** section of the toolstrip is collapsed, you can expand it by clicking the down arrow.

- b Using the file explorer, locate and select the `addmatrix.m` file.
 - c Click **Open** to select the file and close the file explorer.
- 5 Explore the main body of the project window.

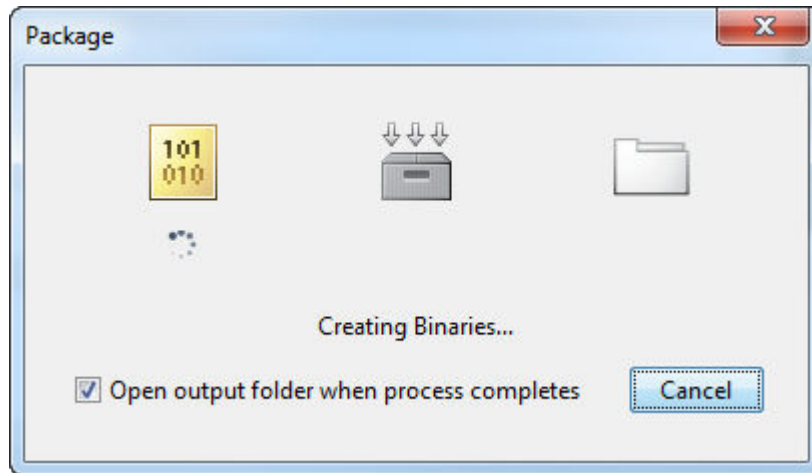
The project window is divided into the following areas:

- **Archive Information** — Editable information about the deployed archive.
- **Files required for your archive to run** — Additional files required by the archive. These files will be included in the generated archive. See “Manage Required Files in Compiler Project” (MATLAB Production Server).
- **Files packaged with the archive** — Files that are packaged with your archive. These files include:
 - `readme.txt`
 - `addmatrix.ctf` file

See “Specify Files to Install with Application” (MATLAB Production Server).

- 6 Click **Package**.

The Package window opens while the library is being generated.



- 7 Select the **Open output folder when process completes** check box.

When the deployment process is complete, a file explorer opens and displays the generated output.

- 8 Verify the contents of the generated output:
 - `for_redistribution` — A folder containing the installer to redistribute the archive to the system administrator responsible for the MATLAB Production Server
 - `for_testing` — A folder containing the raw files generated by the compiler
 - `PackagingLog.txt` — A log file generated by the compiler.
- 9 Click **Close** on the Package window.

To learn more about MATLAB Production Server see “MATLAB Production Server”

Create a C# Client

This example shows how to call a deployed MATLAB function from a C# application using MATLAB Production Server.

In your C# code, you must:

- Create a Microsoft Visual Studio Project.
- Create a Reference to the Client Run-Time Library.
- Design the .NET interface in C#.
- Write, build, and run the C# application.

This task is typically performed by .NET application programmer. This part of the tutorial assumes you have Microsoft Visual Studio and .NET installed on your computer.

Create a Microsoft Visual Studio Project

- 1 Open Microsoft Visual Studio.
- 2 Click **File > New > Project**.
- 3 In the New Project dialog, select the project type and template you want to use. For example, if you want to create a C# Console Application, select **Windows** in the **Visual C#** branch of the **Project Type** pane, and select the **C# Console Application** template from the **Templates** pane.
- 4 Type the name of the project in the **Name** field (*Magic*, for example).
- 5 Click **OK**. Your *Magic* source shell is created, typically named `Program.cs`, by default.

Create a Reference to the Client Run-Time Library

Create a reference in your `MainApp` code to the MATLAB Production Server client runtime library. In Microsoft Visual Studio, perform the following steps:

- 1 In the Solution Explorer pane within Microsoft Visual Studio (usually on the right side), select the name of your project, *Magic*, highlighting it.
- 2 Right-click *Magic* and select **Add Reference**.
- 3 In the Add Reference dialog box, select the **Browse** tab. Browse to the MATLAB Production Server client runtime, installed at `matlabroot\toolbox\compiler_sdk\mps_client\dotnet`. Select `MathWorks.MATLAB.ProductionServer.Client.dll`.

- 4 Click **OK**. `MathWorks.MATLAB.ProductionServer.Client.dll` is now referenced by your Microsoft Visual Studio project.

Design the .NET Interface in C#

In this example, you invoke `mymagic.m`, hosted by the server, from a .NET client, through a .NET interface.

To match the MATLAB function `mymagic.m`, design an interface named `Magic`.

For example, the interface for the `mymagic` function:

```
function m = mymagic(in)
    m = magic(in);
```

might look like this:

```
public interface Magic
{
    double[,] mymagic(int in1);
}
```

Note the following:

- The .NET interface has the same number of inputs and outputs as the MATLAB function.
- You are deploying one MATLAB function, therefore you define one corresponding .NET method in your C# code.
- Both MATLAB function and .NET interface process the same types: input type `int` and the output type two-dimensional `double`.
- You specify the name of your deployable archive (`magic`, which resides in your `auto_deploy` folder) in your URL, when you call `CreateProxy` ("`http://localhost:9910/magic`").

Write, Build, and Run the .NET Application

Create a C# interface named `Magic` in Microsoft Visual Studio by doing the following:

- 1 Open the Microsoft Visual Studio project, `MagicSquare`, that you created earlier.
- 2 In `Program.cs` tab, paste in the code below.

Note The URL value ("http://localhost:9910/mymagic_deployed") used to create the proxy contains three parts:

- the server address (localhost).
 - the port number (9910).
 - the archive name (mymagic_deployed)
-

```
using System;
using System.Net;
using MathWorks.MATLAB.ProductionServer.Client;

namespace Magic
{
    public class MagicClass
    {
        public interface Magic
        {
            double[,] mymagic(int in1);
        }

        public static void Main(string[] args)
        {
            MWClient client = new MWHttpClient();
            try
            {
                Magic me = client.CreateProxy<Magic>
                    (new Uri("http://localhost:9910/mymagic_deployed"));
                double[,] result1 = me.mymagic(4);
                print(result1);
            }
            catch (MATLABException ex)
            {
                Console.WriteLine("{0} MATLAB exception caught.", ex);
                Console.WriteLine(ex.StackTrace);
            }
            catch (WebException ex)
            {
                Console.WriteLine("{0} Web exception caught.", ex);
                Console.WriteLine(ex.StackTrace);
            }
            finally
            {
                client.Dispose();
            }
            Console.ReadLine();
        }

        public static void print(double[,] x)
        {
```

```
int rank = x.Rank;
int [] dims = new int[rank];

for (int i = 0; i < rank; i++)
{
    dims[i] = x.GetLength(i);
}

for (int j = 0; j < dims[0]; j++)
{
    for (int k = 0; k < dims[1]; k++)
    {
        Console.Write(x[j,k]);
        if (k < (dims[1] - 1))
        {
            Console.Write(",");
        }
    }
    Console.WriteLine();
}
}
```

- 3 Build the application. Click **Build > Build Solution**.
- 4 Run the application. Click **Debug > Start Without Debugging**. The program returns the following console output:

```
16,2,3,13
5,11,10,8
9,7,6,12
4,14,15,1
```

Create a Java Client

This example shows how to write a MATLAB Production Server client using the Java client API. In your Java code, you will:

- Define a Java interface that represents the MATLAB function.
- Instantiate a proxy object to communicate with the server.
- Call the deployed function in your Java code.

To create a Java MATLAB Production Server client application:

- 1** Create a new file called `MPSClientExample.java`.
- 2** Using a text editor, open `MPSClientExample.java`.
- 3** Add the following import statements to the file:

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;
```

- 4** Add a Java interface that represents the deployed MATLAB function.

The interface for the `addmatrix` function

```
function a = addmatrix(a1, a2)
```

```
a = a1 + a2;
```

looks like this:

```
interface MATLABAddMatrix {
    double[][] addmatrix(double[][] a1, double[][] a2)
        throws MATLABException, IOException;
}
```

When creating the interface, note the following:

- You can give the interface any valid Java name.
- You must give the method defined by this interface the same name as the deployed MATLAB function.
- The Java method must support the same inputs and outputs supported by the MATLAB function, in both type and number. For more information about data type

conversions and how to handle more complex MATLAB function signatures, see “Java Client Programming” (MATLAB Production Server).

- The Java method must handle MATLAB exceptions and I/O exceptions.

5 Add the following class definition:

```
public class MPSClientExample
{
}
```

This class now has a single main method that calls the generated class.

6 Add the `main()` method to the application.

```
public static void main(String[] args)
{
}
```

7 Add the following code to the top of the `main()` method:

```
double[][] a1={{1,2,3},{3,2,1}};
double[][] a2={{4,5,6},{6,5,4}};
```

These statements initialize the variables used by the application.

8 Instantiate a client object using the `MWHttpClient` constructor.

```
MWClient client = new MWHttpClient();
```

This class establishes an HTTP connection between the application and the server instance.

9 Call the client object’s `createProxy` method to create a dynamic proxy.

You must specify the URL of the deployable archive and the name of your interface class as arguments:

```
MATLABAddMatrix m = client.createProxy(new URL("http://localhost:9910/addmatrix"),
MATLABAddMatrix.class);
```

The URL value (“http://localhost:9910/addmatrix”) used to create the proxy contains three parts:

- the server address (`localhost`).
- the port number (`9910`).
- the archive name (`addmatrix`)

For more information about the `createProxy` method, see the Javadoc included in the `matlabroot/toolbox/compiler_sdk/mps_client` folder.

- 10 Call the deployed MATLAB function in your Java application by calling the public method of the interface.

```
double[][] result = m.addmatrix(a1,a2);
```

- 11 Call the client object's `close()` method to free system resources.

```
client.close();
```

- 12 Save the Java file.

The completed Java file should resemble the following:

```
import java.net.URL;
import java.io.IOException;
import com.mathworks.mps.client.MWClient;
import com.mathworks.mps.client.MWHttpClient;
import com.mathworks.mps.client.MATLABException;

interface MATLABAddMatrix
{
    double[][] addmatrix(double[][] a1, double[][] a2)
        throws MATLABException, IOException;
}

public class MPSClientExample {

    public static void main(String[] args){

        double[][] a1={{1,2,3},{3,2,1}};
        double[][] a2={{4,5,6},{6,5,4}};

        MWClient client = new MWHttpClient();

        try{
            MATLABAddMatrix m = client.createProxy(new URL("http://localhost:9910/addmatrix"),
                MATLABAddMatrix.class);
            double[][] result = m.addmatrix(a1,a2);

            // Print the magic square

            printResult(result);

        }catch(MATLABException ex){

            // This exception represents errors in MATLAB
            System.out.println(ex);
        }catch(IOException ex){

            // This exception represents network issues.
            System.out.println(ex);
        }finally{

            client.close();
        }
    }

    private static void printResult(double[][] result){
```

```
        for(double[] row : result){
            for(double element : row){
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
}
```

- 13** Compile the Java application, using the `javac` command or use the build capability of your Java IDE.

For example, enter the following:

```
javac -classpath "matlabroot\toolbox\compiler_sdk\mps_client\java\mps_client.jar" MPSCClientExample.java
```

- 14** Run the application using the `java` command or your IDE.

For example, enter the following:

```
java -classpath .;"matlabroot\toolbox\compiler_sdk\mps_client\java\mps_client.jar" MPSCClientExample
```

The application returns the following at the console:

```
5.0 7.0 9.0
9.0 7.0 5.0
```


Create a C++ Client

This example shows how to write a MATLAB Production Server client using the C client API. The client application calls the `addmatrix` function you compiled in “Create a Deployable Archive for MATLAB Production Server” (MATLAB Production Server) and deployed in “Share a Deployable Archive on the Server Instance” (MATLAB Production Server).

Create a C++ MATLAB Production Server client application:

- 1 Create a file called `addmatrix_client.cpp`.
- 2 Using a text editor, open `addmatrix_client.cpp`.
- 3 Add the following include statements to the file:

```
#include <iostream>
#include <mps/client.h>
```

Note The header files for the MATLAB Production Server C client API are located in the `matlabroot/toolbox/compiler_sdk/mps_client/c/include/mps` folder.

- 4 Add the `main()` method to the application.

```
int main ( void )
{
}
```

- 5 Initialize the client runtime.

```
mpsClientRuntime* mpsruntime = mpsInitializeEx(MPS_CLIENT_1_1);
```

- 6 Create the client configuration.

```
mpsClientConfig* config;
mpsStatus status = mpsruntime->createConfig(&config);
```

- 7 Create the client context.

```
mpsClientContext* context;
status = mpsruntime->createContext(&context, config);
```

- 8 Create the MATLAB data to input to the function.

```
double a1[2][3] = {{1,2,3},{3,2,1}};
double a2[2][3] = {{4,5,6},{6,5,4}};
```

```
int numIn=2;
```

```
mpsArray** inVal = new mpsArray* [numIn];

inVal[0] = mpsCreateDoubleMatrix(2,3,mpsREAL);
inVal[1] = mpsCreateDoubleMatrix(2,3,mpsREAL);

double* data1 = (double *) ( mpsGetData(inVal[0]) );
double* data2 = (double *) ( mpsGetData(inVal[1]) );

for(int i=0; i<2; i++)
{
    for(int j=0; j<3; j++)
    {
        mpsIndex subs[] = { i, j };
        mpsIndex id = mpsCalcSingleSubscript(inVal[0], 2, subs);
        data1[id] = a1[i][j];
        data2[id] = a2[i][j];
    }
}
```

- 9** Create the MATLAB data to hold the output.

```
int numOut = 1;
mpsArray **outVal = new mpsArray* [numOut];
```

- 10** Call the deployed MATLAB function.

Specify the following as arguments:

- client context
- URL of the function
- Number of expected outputs
- Pointer to the mpsArray holding the outputs
- Number of inputs
- Pointer to the mpsArray holding the inputs

```
mpsStatus status = mpsruntime->feval(context,
    "http://localhost:9910/addmatrix/addmatrix",
    numOut, outVal, numIn, (const mpsArray**)inVal);
```

For more information about the `feval` function, see the reference material included in the `matlabroot/toolbox/compiler_sdk/mps_client` folder.

- 11** Verify that the function call was successful using an `if` statement.

```

if (status==MPS_OK)
{
}

```

- 12** Inside the if statement, add code to process the output.

```

double* out = mpsGetPr(outVal[0]);

for (int i=0; i<2; i++)
{
    for (int j=0; j<3; j++)
    {
        mpsIndex subs[] = {i, j};
        mpsIndex id = mpsCalcSingleSubscript(outVal[0], 2, subs);
        std::cout << out[id] << "\t";
    }
    std::cout << std::endl;
}

```

- 13** Add an else clause to the if statement to process any errors.

```

else
{
    mpsErrorInfo error;
    mpsruntime->getLastErrorInfo(context, &error);
    std::cout << "Error: " << error.message << std::endl;
    switch(error.type)
    {
        case MPS_HTTP_ERROR_INFO:
            std::cout << "HTTP: " << error.details.http.responseCode << ": "
                << error.details.http.responseMessage << std::endl;
        case MPS_MATLAB_ERROR_INFO:
            std::cout << "MATLAB: " << error.details.matlab.identifier
                << std::endl;
            std::cout << error.details.matlab.message << std::endl;
        case MPS_GENERIC_ERROR_INFO:
            std::cout << "Generic: " << error.details.general.genericErrorMsg
                << std::endl;
    }

    mpsruntime->destroyLastErrorInfo(&error);
}

```

- 14** Free the memory used by the inputs.

```

for (int i=0; i<numIn; i++)
    mpsDestroyArray(inVal[i]);
delete[] inVal;

```

15 Free the memory used by the outputs.

```
for (int i=0; i<numOut; i++)
    mpsDestroyArray(outVal[i]);
delete[] outVal;
```

16 Free the memory used by the client runtime.

```
mpsruntime->destroyConfig(config);
mpsruntime->destroyContext(context);
mpsTerminate();
```

17 Save the file.

The completed program should resemble the following:

```
#include <iostream>
#include <mps/client.h>

int main ( void )
{
    mpsClientRuntime* mpsruntime = mpsInitializeEx(MPS_CLIENT_1_1);

    mpsClientConfig* config;
    mpsStatus status = mpsruntime->createConfig(&config);

    mpsClientContext* context;
    status = mpsruntime->createContext(&context, config);

    double a1[2][3] = {{1,2,3},{3,2,1}};
    double a2[2][3] = {{4,5,6},{6,5,4}};

    int numIn=2;
    mpsArray** inVal = new mpsArray* [numIn];
    inVal[0] = mpsCreateDoubleMatrix(2,3,mpsREAL);
    inVal[1] = mpsCreateDoubleMatrix(2,3,mpsREAL);
    double* data1 = (double *) ( mpsGetData(inVal[0]) );
    double* data2 = (double *) ( mpsGetData(inVal[1]) );
    for(int i=0; i<2; i++)
    {
        for(int j=0; j<3; j++)
        {
            mpsIndex subs[] = { i, j };
            mpsIndex id = mpsCalcSingleSubscript(inVal[0], 2, subs);
            data1[id] = a1[i][j];
            data2[id] = a2[i][j];
        }
    }

    int numOut = 1;
    mpsArray **outVal = new mpsArray* [numOut];

    status = mpsruntime->feval(context,
        "http://localhost:9910/addmatrix/addmatrix",
        numOut, outVal, numIn, (const mpsArray **)inVal);

    if (status==MPS_OK)
    {
        double* out = mpsGetPr(outVal[0]);
    }
}
```

```

for (int i=0; i<2; i++)
{
    for (int j=0; j<3; j++)
    {
        mpsIndex subs[] = {i, j};
        mpsIndex id = mpsCalcSingleSubscript(outVal[0], 2, subs);
        std::cout << out[id] << "\t";
    }
    std::cout << std::endl;
}
}
else
{
    mpsErrorInfo error;
    mpsruntime->getLastErrorInfo(context, &error);
    std::cout << "Error: " << error.message << std::endl;

    switch(error.type)
    {
    case MPS_HTTP_ERROR_INFO:
        std::cout << "HTTP: "
            << error.details.http.responseCode
            << ": " << error.details.http.responseMessage
            << std::endl;
    case MPS_MATLAB_ERROR_INFO:
        std::cout << "MATLAB: " << error.details.matlab.identifier
            << std::endl;
        std::cout << error.details.matlab.message << std::endl;
    case MPS_GENERIC_ERROR_INFO:
        std::cout << "Generic: "
            << error.details.general.genericErrorMsg
            << std::endl;
    }
    mpsruntime->destroyLastErrorInfo(&error);
}

for (int i=0; i<numIn; i++)
    mpsDestroyArray(inVal[i]);
delete[] inVal;

for (int i=0; i<numOut; i++)
    mpsDestroyArray(outVal[i]);
delete[] outVal;

mpsruntime->destroyConfig(config);
mpsruntime->destroyContext(context);
mpsTerminate();
}

```

18 Compile the application.

To compile your client code, the compiler needs access to `client.h`. This header file is stored in `matlabroot/toolbox/compiler_sdk/mps_client/c/include/mps/`.

To link your application, the linker needs access to the following files stored in `matlabroot/toolbox/compiler_sdk/mps_client/c/`:

Files Required for Linking

Windows	UNIX/Linux	Mac OS X
\$arch\lib \mpsclient.lib	\$arch/lib/ libprotobuf.so	\$arch/lib/ libprotobuf.dylib
	\$arch/lib/ libcurl.so	\$arch/lib/ libcurl.dylib
	\$arch/lib/ libmwmpsclient.so	\$arch/lib/ libmwmpsclient.dyl ib
	\$arch/lib/ libmwcppl1compat.so	

19 Run the application.

To run your application, add the following files stored in *matlabroot/toolbox/compiler_sdk/mps_client/c/* to the application's path:

Files Required for Running

Windows	UNIX/Linux	Mac OS X
\$arch\lib \mpsclient.dll	\$arch/lib/ libprotobuf.so	\$arch/lib/ libprotobuf.dylib
\$arch\lib \libprotobuf.dll	\$arch/lib/ libcurl.so	\$arch/lib/ libcurl.dylib
\$arch\lib \libcurl.dll	\$arch/lib/ libmwmpsclient.so	\$arch/lib/ libmwmpsclient.dyl ib
	\$arch/lib/ libmwcppl1compat.so	

The client invokes `addmatrix` function on the server instance and returns the following matrix at the console:

```
5.0 7.0 9.0
9.0 7.0 5.0
```

Create a Python Client

This example shows how to write a MATLAB Production Server client using the Python client API. The client application calls the `addmatrix` function you compiled in “Create a Deployable Archive for MATLAB Production Server” (MATLAB Production Server) and deployed in “Share a Deployable Archive on the Server Instance” (MATLAB Production Server).

Create a Python MATLAB Production Server client application:

- 1 Copy the contents of the `matlabroot\toolbox\compiler_sdk\mps_clients\python` folder to your development environment.
- 2 Open a command line,
- 3 Change directories into the folder where you copied the MATLAB Production Server Python client.
- 4 Run the following command.

```
python setup.py install
```

- 5 Start the Python command line interpreter.
- 6 Enter the following import statements at the Python command prompt.

```
import matlab
from production_server import client
```

- 7 Open the connection to the MATLAB Production Server instance and initialize the client runtime.

```
client_obj = client.MWHttpClient("http://localhost:9910")
```

- 8 Create the MATLAB data to input to the function.

```
a1 = matlab.double([[1,2,3],[3,2,1]])
a2 = matlab.double([[4,5,6],[6,5,4]])
```

- 9 Call the deployed MATLAB function.

You must know the following:

- Name of the deployed archive
- Name of the function

```
client_obj.addmatrix.addmatrix(a1,a2)
```

```
matlab.double([[5.0,7.0,9.0],[9.0,7.0,5.0]])
```

The syntax for invoking a function is `client.archiveName.functionName(arg1, arg2, ..., [nargout=numOutArgs])`.

- 10** Close the client connection.

```
client_obj.close()
```