MATLAB® Compiler SDK™

Getting Started Guide

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Revision History

Contents

4

Overview of MATLAB Compiler SDK

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, Java® classes, Python® packages, and Docker® container-based microservices 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.

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.

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			■		

Task	MATLAB Compiler and MATLAB Compiler SDK	MATLAB Coder™	Simulink®	HDL Coder™	MATLAB External Interfaces
Generate hardware description language (HDL) from MATLAB code				■	
Integrate custom C code into MATLAB with MEX files					■
Call MATLAB from C and Fortran programs					■
Task	MATLAB Compiler and MATLAB Compiler SDK	MATLAB Coder	Simulink	HDL Coder	MATLAB External Interfaces

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

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".

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 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 `mxArray`. 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 `mxArray`.

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 lose 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 `mxArray`.

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.

Files Generated After Packaging MATLAB Functions

When the packaging process is complete, three folders are generated in the target folder location: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

The file `PackagingLog.html` generated in the target folder location contains information on the `mcc` command used and output from the packaging process.

for_redistribution Folder

Distribute the `for_redistribution` folder to users who do not have MATLAB installed on their machines.

The folder contains the file `MyAppInstaller_web.exe` that installs the application and the MATLAB Runtime (if it is included in the application at the time of packaging). It installs all the files that enable use of the packaged application on the target platform with the target language in the target folder.

for_redistribution_files_only Folder

Distribute the `for_redistribution_files_only` folder to users who do not have MATLAB installed on their machines. This folder contains specific files that enable use of the packaged application on the target platform with the target language.

C Shared Library

File	Description
`GettingStarted.html`	HTML file containing packaging information.
filename`.lib`	Import library for user-written shared library.
filename`.h`	Header file for user-written shared library.
filename`.dll`	Code for user-written shared library.

C++ Shared Library

File	Description
GettingStarted.html	HTML file containing packaging information.
filename.lib	Import library for user-written mwArray API shared library.
filename.h	Header file for user-written mwArray API shared library.
filename.dll	Code for user-written mwArray API shared library.
v2: • generic_interface: • *filename*.ctf • readme.txt	The folder v2 contains another folder generic_interface. It contains a ctf file, which is the deployable archive for MATLAB Data API library. It also contains a readme.txt file that has packaging information.

COM Component

File	Description
_install.bat	File that registers the generated dll file.
*filename*_1_0.dll	The generated dll that needs to be registered using mwregsvr.exe or regsvr32.exe.
GettingStarted.html	HTML file containing packaging information.

.NET Assembly

File	Description
filename.dll	File that contains the generated component that can be accessed using mwArray API.
*filename*_overview.html	HTML overview documentation file for the generated component. It contains requirements for accessing the component and for generating arguments using the mwArray class hierarchy.
*filename*Native.dll	File that contains the generated component that can be accessed using native API.
GettingStarted.html	HTML file containing packaging information.

Java Application

File	Description

doc:
 	
• html:
 	
  • *filename*
 	
    • Class1.html
    • Class1Remote.html
    • *Filename*MCRFactory.html
    • packageframe.html
    • package-summary.html
    • packagetree.html
  • allclasses-frame.html
  • allclasses-noframe.html
  • constantvalues.html
  • deprecated-list.html
  • help-doc.html
  • index.html
  • index-all.html
  • overview-tree.html
  • package-list
  • script.js
  • stylesheet.css
	The folder doc contains another folder html which contains HTML documentation for all classes in the packaged Java application.

Filename.jar	Java archive for user-written application.
GettingStarted.html	HTML file containing packaging information.

Python Application

File	Description
filename: • _init_.py • *filename*.ctf	The folder *Filename* contains the following files: • File used during initialization of the Python package. • Deployable archive for the Python package.
setup.py	File that installs the Python packaged application.
GettingStarted.html	HTML file containing packaging information.

for_testing Folder

Use the files in this folder to test your application. The folder contains all the intermediate and final artifacts such as binaries, JAR files, header files, and source files for a specific target. The final artifacts created during the packaging process are the same files as described in "for_redistribution_files_only Folder". You use these files to test your application.

The intermediate artifacts generated are a result of packaging of the MATLAB files. They are not significant to the user.

This folder also contains three text files. `readme.txt` contains usage information, `mccExcludedFiles.txt` lists the files excluded from packaged application, and `requiredMCRProducts.txt`, contains product IDs of products required by MATLAB Runtime to run the application. For more information on product IDs, see `matlab.codetools.requiredFilesAndProducts`.

See Also

`mcc` | `deploytool` | `compiler.build.Results`

More About

- "Distribute Files to Application Developers"

Distribute Files to Application Developers

In this section...
"Distribute COM Components" on page 1-14
"Distribute C/C++ Shared Libraries" on page 1-14
"Distribute Java Packages" on page 1-14
"Distribute .NET Assemblies" on page 1-15
"Distribute Python Packages" on page 1-15

After you create a component using MATLAB Compiler SDK, distribute files and integrate them in an application in the target language.

The `deploytool` apps generate an installer that packages all of the binary artifacts required for distributing a compiled component. The installer is located in the `for_redistribution` folder of the compiler project.

You can also generate an installer using the `compiler.package.installer` function.

If you do not create an installer, distribute the set of files required to integrate the component according to the component type. In order to run the application, the target machine must have access to MATLAB Runtime that matches the version of MATLAB used to compile the component, at the same update level or newer.

Distribute COM Components

Distribute the following files to integrate a component in an application:

- Function signatures of the deployed MATLAB functions
- *packageName*`.dll` — generated COM component
- `_install.bat` — generated script that registers the component (to register manually, see "Register COM Component")

Distribute C/C++ Shared Libraries

Distribute the following files to integrate a C/C++ shared library in an application:

- Function signatures of the deployed MATLAB functions
- *libraryName*`.lib/.dylib/.so` — generated library
- *libraryName*`.h` — generated header file

Distribute Java Packages

Distribute the following files to integrate a Java package in an application:

- Function signatures of the deployed MATLAB functions
- *packageName*`.jar` — generated Java package

Distribute .NET Assemblies

Distribute the following files to integrate a .NET assembly in an application:

- Function signatures of the deployed MATLAB functions
- *assemblyName*.dll — generated assembly file
- *assemblyName*.xml — generated documentation files
- *assemblyName*.pdb — optionally generated program database file containing debugging information

Distribute Python Packages

Distribute the following files to integrate a Python package in an application:

- Function signatures of the deployed MATLAB functions
- _init_.py — initialization script for the Python package
- setup.py — generated Python installer

See Also
"Files Generated After Packaging MATLAB Functions" on page 1-10

2

Examples

- "Create a C Shared Library with MATLAB Code" on page 2-2
- "Generate a C++ mwArray API Shared Library and Build a C++ Application" on page 2-10
- "Generate a C++ MATLAB Data API Shared Library and Build a C++ Application" on page 2-16
- "Generate .NET Assembly and Build .NET Application" on page 2-22
- "Create a Generic COM Component with MATLAB Code" on page 2-30
- "Generate Java Package and Build Java Application" on page 2-35
- "Generate a Python Package and Build a Python Application" on page 2-42

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. The target system does not require a licensed copy of MATLAB.

Create Functions in MATLAB

1   In MATLAB, examine the MATLAB code that you want packaged.

For this example, copy the `matrix` folder that ships with MATLAB to your work folder.

```
copyfile(fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix'),'matrix')
```

Navigate to the new `matrix` subfolder in your work folder.

2   Examine and test the functions `addmatrix.m`, `multiplymatrix.m`, and `eigmatrix.m`.

addmatrix.m

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

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)

m =  a1*a2;
```

At the MATLAB command prompt, enter `multiplymatrix([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)

    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

Build a C shared library using the **Library Compiler** app. Alternatively, if you want to create a C shared library from the MATLAB command window using a programmatic approach, see "Create C Shared Library Using compiler.build.cSharedLibrary" on page 2-6.

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 **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

a In the **Exported Functions** section of the toolstrip, click ⊕.

b 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 exported function files. Repeat this step to package multiple files in the same application.

Add `addmatrix.m`, `multiplymatrix.m`, and `eigmatrix.m` to the list of main files.

3 In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

- **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.

- **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer.

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

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application 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** — Default installation path for the generated installer and custom logo selection. See "Change the Installation Path" on page 3-3.

- **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 3-4.

- **Files installed for your end user** — Files that are installed with your application.

  See "Specify Files to Install with Application" on page 3-6.

Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

1   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 packaging process is complete, examine the generated output in the target folder.

   • Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

      For more information about the files generated in these folders, see "Files Generated After Packaging MATLAB Functions" on page 1-10.

   • The log file `PackagingLog.html` contains packaging results.

Create C Shared Library Using compiler.build.cSharedLibrary

As an alternative to the Library Compiler app, you can create a C shared library using a programmatic approach. If you have already created a library using the Library Compiler, see "Implement C Shared Library in C Application" on page 2-6.

- Build the C shared library using the `compiler.build.cSharedLibrary` function. Use name-value arguments to specify the library name and enable verbose output.

```
buildResults = compiler.build.cSharedLibrary(["addmatrix.m", ...
"eigmatrix.m","multiplymatrix.m"], ...
'LibraryName','libmatrix', ...
'Verbose','on');
```

You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.cSharedLibrary`.

The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

The function generates the following files within a folder named `libmatrixcSharedLibrary` in your current working directory:

- `GettingStarted.html` — HTML file that contains information on integrating your shared library.
- `includedSupportPackages.txt` — Text file that lists all support files included in the library.
- `libmatrix.c` — C source code file.
- `libmatrix.def` — Module-definition file that provides the linker with module information.
- `libmatrix.dll` — Dynamic-link library file.
- `libmatrix.exports` — Exports file that contains all nonstatic function names.
- `libmatrix.h` — C header file.
- `libmatrix.lib` — Import library file. The file extension is `.dylib` on Mac and `.so` on UNIX®.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Text file that contains packaging information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Note The generated library does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Implement C Shared Library in C Application

After packaging your C shared library, you can call it from a C application. The C application code calls the functions included in the shared library.

1 Locate the `matrix.c` file located in *matlabroot*\extern\examples\compilersdk\c_cpp \matrix or your work folder.

matrix.c

```
/*==================================================================
 *
 * MATRIX.C Sample driver code that calls a shared library created
 *          using MATLAB Compiler SDK. Refer to the MATLAB Compiler
 *          SDK documentation for more information.
 *
 * Copyright 1984-2022 The MathWorks, Inc.
 *
 *==================================================================*/

#include <stdio.h>

/* Include the MATLAB Runtime header file and the library specific header file
 * as generated by MATLAB Compiler SDK. */
#include "libmatrix.h"

/* This function is used to display a double matrix stored in an mxArray */
void display(const mxArray* in);

int run_main(int argc, const char **argv)
{
    mxArray *in1, *in2; /* Define input parameters */
    mxArray *out = NULL;/* and output parameters to be passed to the library functions */

    double data[] = {1,2,3,4,5,6,7,8,9};

    /* Create the input data */
    in1 = mxCreateDoubleMatrix(3,3,mxREAL);
    in2 = mxCreateDoubleMatrix(3,3,mxREAL);
    memcpy(mxGetPr(in1), data, 9*sizeof(double));
    memcpy(mxGetPr(in2), data, 9*sizeof(double));

    /* Call the library intialization routine and make sure that the
     * library was initialized properly. */
    if (!libmatrixInitialize()){
        fprintf(stderr,"Could not initialize the library.\n");
        return -2;
    }
    else
    {
        /* Call the library function */
        mlfAddmatrix(1, &out, in1, in2);
        /* Display the return value of the library function */
        printf("The sum of the matrix with itself is:\n");
        display(out);
        /* Destroy the return value since this variable will be reused in
         * the next function call. Since we are going to reuse the variable,
         * we must set it to NULL. Refer to MATLAB Compiler SDK documentation
         * for more information. */
        mxDestroyArray(out);
        out=0;
        mlfMultiplymatrix(1, &out, in1, in2);
        printf("The product of the matrix with itself is:\n");
```

```
            display(out);
            mxDestroyArray(out);
            out=0;
            mlfEigmatrix(1, &out, in1);
            printf("The eigenvalues of the original matrix are:\n");
            display(out);
            mxDestroyArray(out);
            out=0;

            /* Call the library termination routine */
            libmatrixTerminate();

            /* Free the memory created */
            mxDestroyArray(in1);
            in1=0;
            mxDestroyArray(in2);
            in2=0;
        }

        /* Note that you should call mclTerminateApplication at the end of
         * your application.
         */
        mclTerminateApplication();
        return 0;
}


/*DISPLAY This function will display the double matrix stored in an mxArray.
 * This function assumes that the mxArray passed as input contains double
 * array.
 */
void display(const mxArray* in)
{
    size_t i=0, j=0; /* loop index variables */
    size_t r=0, c=0; /* variables to store the row and column length of the matrix */
    double *data; /* variable to point to the double data stored within the mxArray */

    /* Get the size of the matrix */
    r = mxGetM(in);
    c = mxGetN(in);
    /* Get a pointer to the double data in mxArray */
    data = mxGetPr(in);

    /* Loop through the data and display it in matrix format */
    for( i = 0; i < c; i++ )
    {
        for( j = 0; j < r; j++)
        {
            printf("%4.2f\t",data[j*c+i]);
        }
        printf("\n");
    }
    printf("\n");
}

int main(int argc, const char ** argv)
{
    /* Call the mclInitializeApplication routine. Make sure that the application
```

```
 * was initialized properly by checking the return status. This initialization
 * has to be done before calling any MATLAB APIs or MATLAB Compiler SDK
 * generated shared library functions.  */
if( !mclInitializeApplication(NULL,0) )
{
    fprintf(stderr, "Could not initialize the application.\n");
    return -1;
}
return mclRunMain((mclMainFcnType)run_main, argc, argv);
}
```

Copy and paste this file into the folder that contains your C shared library `libmatrix.lib`. If you used the Library Compiler, it is located in the `for_testing` folder.

Note The `.lib` extension is for Windows. On Mac, the file extension is `.dylib`, and on UNIX it is `.so`.

2 At the MATLAB command prompt, navigate to the folder where you copied `matrix.c`.

3 Use `mbuild` to compile and link the application.

```
mbuild matrix.c libmatrix.lib
```

4 From the system command prompt, run the application.

```
matrix
The sum of the matrix with itself is:
2.00         8.00          14.00
4.00         10.00          16.00
6.00         12.00          18.00

The product of the matrix with itself is:
30.00         66.00          102.00
36.00         81.00          126.00
42.00         96.00          150.00

The eigenvalues of the original matrix are:
16.12         -1.12          -0.00
```

Note You may need to give the application executable permissions on UNIX systems by running

```
chmod u+x matrix
```

See Also

compiler.build.cSharedLibrary | **Library Compiler** | deploytool | mxArray (C)

More About

•   "Call a C Shared Library"

•   "Generate a C++ mwArray API Shared Library and Build a C++ Application" on page 2-10

•   "Generate a C++ MATLAB Data API Shared Library and Build a C++ Application" on page 2-16

Generate a C++ mwArray API Shared Library and Build a C++ Application

Supported platform: Windows, Linux, Mac

This example shows how to create a C++ shared library from MATLAB functions. You can integrate the generated library into a C++ application. This example also shows how to call the C++ shared library from a C++ application. The target system does not require a licensed copy of MATLAB.

Create Functions in MATLAB

1   In MATLAB, examine the MATLAB code that you want packaged.

For this example, Copy the `matrix` folder that ships with MATLAB to your work folder.

```
copyfile(fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix'),'matrix')
```

Navigate to the new `matrix` subfolder in your work folder.

2   Examine and test the functions `addmatrix.m`, `multiplymatrix.m`, and `eigmatrix.m`.
3   Create MATLAB sample code that calls the functions. Sample files are used to generate a sample application in the target language. For more information and limitations, see "Sample Driver File Creation" on page 3-5.

Save the following code in a sample file named `libmatrixSample.m`:

```
% Sample script to demonstrate execution of functions
% addmatrix, eigmatrix, and multiplymatrix
a1 = [1 4 7; 2 5 8; 3 6 9]; % Initialize a1 here
a2 = a1; % Initialize a2 here
a = addmatrix(a1, a2);
e = eigmatrix(a1);
m = multiplymatrix(a1, a2);
```

You may instead choose to not include a sample driver file at all during the packaging step. If you create your own C++ application code, you can move it to the appropriate directory and compile it using `mbuild` after the MATLAB functions are packaged.

Create a C++ Shared Library 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**.

Alternatively, you can open the **Library Compiler** app from the MATLAB command prompt by entering:

```
libraryCompiler
```

2   In the **Type** section of the toolstrip, click **C++ Shared Library**.

In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

a   In the **Exported Functions** section of the toolstrip, click ⊞.

b   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 exported function files. Repeat this step to package multiple files in the same application.

Add `addmatrix.m`, `multiplymatrix.m`, and `eigmatrix.m` to the list of main files.

3   In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

- **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.

- **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer.

Specify Shared Library Settings

1   The **Library Name** field is automatically populated with `addmatrix` as the name of the packaged shared library. Rename it as `libmatrix`. The same name is followed through in the implementation of the shared library.

2   Add the MATLAB sample file `libmatrixSample.m` you created earlier. Although C++ driver files are not necessary to create shared libraries, they are used to demonstrate how to "Implement C++ mwArray API Shared Library with C++ Sample Application" on page 2-13.

In the **Samples** section, select **Add Existing Sample**, and select `libmatrixSample.m`.

3   Select the type of API for the generated C++ shared libraries. In the **API selection** section at the bottom, select **Create interface that uses the mwArray API**. You may also leave it on the default option to create both interfaces. For more information, see "API Selection for C++ Shared Library" on page 3-7.

Customize the Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application 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** — Default installation path for the generated installer and custom logo selection. See "Change the Installation Path" on page 3-3.

- **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 3-4.

- **Files installed for your end user** — Files that are installed with your application.

See "Specify Files to Install with Application" on page 3-6.

Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

1 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 packaging process is complete, examine the generated output in the target folder.

- Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

 For more information about the files generated in these folders, see "Files Generated After Packaging MATLAB Functions" on page 1-10.

- The log file `PackagingLog.html` contains packaging results.

Create C++ Shared Library Using compiler.build.cppSharedLibrary

As an alternative to the Library Compiler app, you can create a C++ shared library using a programmatic approach. If you have already created a library using the Library Compiler, see "Implement C++ mwArray API Shared Library with C++ Sample Application" on page 2-13.

1 Save the list of function files in a cell array.

```
functionfiles = {'addmatrix.m', 'multiplymatrix.m', 'eigmatrix.m'}
```

2 Build the C++ shared library using the `compiler.build.cppSharedLibrary` function. Use name-value arguments to add the sample file and specify the library name and interface API.

```
buildResults = compiler.build.cppSharedLibrary(functionfiles,...
'LibraryName','libmatrix',...
'Interface','mwarray',...
'SampleGenerationFiles','libmatrixSample.m');
```

You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.cppSharedLibrary`.

The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

The function generates the following files within a folder named `libmatrixcppSharedLibrary` in your current working directory:

- `samples\libmatrixSample1_mwarray.cpp` — C++ sample application that calls the `addmatrix` function.
- `samples\libmatrixSample2_mwarray.cpp` — C++ sample application that calls the `eigmatrix` function.
- `samples\libmatrixSample3_mwarray.cpp` — C++ sample application that calls the `multiplymatrix` function.
- `GettingStarted.html` — HTML file that contains information on integrating your shared library.
- `includedSupportPackages.txt` — Text file that lists all support files included in the library.

- `libmatrix.cpp` — C++ source code file.
- `libmatrix.def` — Module-definition file that provides the linker with module information.
- `libmatrix.dll` — Dynamic-link library file.
- `libmatrix.exports` — Exports file that contains all nonstatic function names.
- `libmatrix.h` — C++ header file.
- `libmatrix.lib` — Import library file.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Text file that contains packaging information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Note The generated library does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Implement C++ mwArray API Shared Library with C++ Sample Application

Note To call the library using a more advanced application that calls all three functions and handles errors, use the C++ application `matrix_mwarray.cpp` located in the folder

matlabroot`\extern\examples\compilersdk\c_cpp\matrix`

For more details, see "Integrate C++ Shared Libraries with mwArray".

Before starting, make sure that you "Install and Configure MATLAB Runtime", and that you have a C++ compiler installed.

After packaging C++ shared libraries, you can call them from a C++ application. The C++ applications generated in the `samples` folder are based on the sample MATLAB file you created.

1 Copy and paste the generated C++ code file `libmatrixSample1_mwarray.cpp` from the `samples` folder into the folder that contains `libmatrix.lib`.

The program listing for `libmatrixSample1_mwarray.cpp` is shown below.

```
/*==================================================================
 *
 * LIBMATRIXSAMPLE1
 * CPP Sample driver code for libmatrix that calls a shared library
 * created using MATLAB Compiler SDK.
 * Refer to the MATLAB Compiler SDK documentation for more information.
 *
 *================================================================*/
// Include the library specific header file as generated by the
// MATLAB Compiler
#include <iostream>
#include "libmatrix.h"

void addmatrixSample() {
    try {
```

```
            mxDouble a1InData[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
            mwArray a1In(3, 3, mxDOUBLE_CLASS);
            a1In.SetData(a1InData, 9);
            mxDouble a2InData[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0};
            mwArray a2In(3, 3, mxDOUBLE_CLASS);
            a2In.SetData(a2InData, 9);
            mwArray aOut;
            addmatrix(1, aOut, a1In, a2In);
            std::cout << aOut << '\n';
        } catch (const mwException& e) {
            std::cerr << e.what() << std::endl;
        } catch (...) {
            std::cerr << "Unexpected error thrown" << std::endl;
        }
    }

    int run_main(int argc, const char** argv) {
        if (!libmatrixInitialize()) {
            std::cerr << "Could not initialize the library properly" << std::endl;
            return 2;
        } else {
            addmatrixSample();
            // Call the application and library termination routine
            libmatrixTerminate();
        }
        // Note that you should call mclTerminateApplication at the end of
        // your application to shut down all MATLAB Runtime instances.
        mclTerminateApplication();
        return 0;
    }

    // The main routine. On macOS, the main thread runs the system code, and
    // user code must be processed by a secondary thread. On other platforms,
    // the main thread runs both the system code and the user code.
    int main(int argc, const char** argv) {
        /* Call the mclInitializeApplication routine. Make sure that the application
         * was initialized properly by checking the return status. This initialization
         * has to be done before calling any MATLAB APIs or MATLAB Compiler SDK
         * generated shared library functions.
         */
        if (!mclInitializeApplication(nullptr, 0)) {
            std::cerr << "Could not initialize the application." << std::endl;
            return 1;
        }
        return mclRunMain(static_cast<mclMainFcnType>(run_main), argc, argv);
    }
```

2 At the system command prompt, navigate to the folder where you copied `libmatrixSample1_mwarray.cpp`.

3 Compile and link the application using `mbuild` at the MATLAB prompt or your system command prompt.

```
mbuild libmatrixSample1_mwarray.cpp libmatrix.lib
```

> **Note** The `.lib` extension is used on Windows. On macOS, the file extension is `.dylib`, and on Linux it is `.so`.

4 From the system command prompt, run the application. If you used sample MATLAB code in the packaging steps, the sample C++ application returns the same output as the MATLAB code.

```
libmatrixSample1_mwarray.exe
```

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

5 (Optional) Compile and link the other sample C++ applications using `mbuild`. You can also use the generated C++ code as a guide to create your own application.

For further details, see "Integrate C++ Shared Libraries with mwArray".

See Also

compiler.build.cppSharedLibrary | **Library Compiler** | mcc | deploytool

Related Examples

- "Integrate C++ Shared Libraries with mwArray"
- "Generate a C++ MATLAB Data API Shared Library and Build a C++ Application" on page 2-16

Generate a C++ MATLAB Data API Shared Library and Build a C++ Application

Supported platform: Windows, Linux, Mac

This example shows how to create a C++ shared library from MATLAB functions. You can integrate the generated library into a C++ application. This example also shows how to call the C++ shared library from a C++ application. The target system does not require a licensed copy of MATLAB to run the application.

Create Functions in MATLAB

1   In MATLAB, examine the MATLAB code that you want to package.

Copy the `matrix` folder that ships with MATLAB to your work folder.

```
copyfile(fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix'),'matrix')
```

Navigate to the new `matrix` subfolder in your work folder.
2   Examine and test the functions `addmatrix.m`, `multiplymatrix.m`, and `eigmatrix.m`.

Create a C++ Shared Library Using Library Compiler App

Compile the functions into a C++ shared library using the **Library Compiler** app. Alternatively, if you want to create a shared library from the MATLAB command window using a programmatic approach, see "Create C++ Shared Library Using compiler.build.cppSharedLibrary" on page 2-18.

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

Alternatively, you can open the **Library Compiler** app from the MATLAB command prompt by entering:

```
libraryCompiler
```
2   In the **Type** section of the toolstrip, click **C++ Shared Library**.

In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

a   In the **Exported Functions** section of the toolstrip, click .
b   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 exported function files. Repeat this step to package multiple files in the same application.

Add `addmatrix.m`, `multiplymatrix.m`, and `eigmatrix.m` to the list of main files.
3   In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

- **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.
- **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer.

Specify Shared Library Settings

1 The **Library Name** field is automatically populated with `addmatrix` as the name of the packaged shared library. Rename it as `libmatrix`. The same name is followed through in the implementation of the shared library.
2 Select the type of API for the generated C++ shared libraries. In the **API selection** section at the bottom, select **Create interface that uses the MATLAB Data API**. You may also leave it on the default option to create both interfaces. For more information, see "API Selection for C++ Shared Library" on page 3-7.

Create Sample Driver File

Create MATLAB sample files that call the functions. Sample files are used to generate a sample application in the target language. You can use any MATLAB file in the project to generate sample code. Although sample files are not necessary to create shared libraries, you can use them as a guide to implement your own application.

You may instead choose to not include a sample driver file at all during the packaging step. If you create your own C++ application code, you can move it to the appropriate directory and compile it using `mbuild` after the MATLAB functions are packaged.

Create sample files for each MATLAB function in the project.

For instance, to create a sample file for `eigmatrix`, in the **Samples** section, select **Create New Sample** and choose `eigmatrix.m`. A MATLAB file opens for you to edit.

```
% Sample script to demonstrate execution of function e = eigmatrix(a1)
a1 = 0; % Initialize a1 here
e = eigmatrix(a1);
```

Change `a1 = 0` to `a1 = [1 4 7; 2 5 8; 3 6 9]`, save the file, and return to the **Library Compiler** app.

Caution You **must** edit the MATLAB sample files to output your desired result. Generated target language sample files use the same inputs and outputs as the sample MATLAB file.

The compiler converts this MATLAB code to C++ code during packaging. For more information and limitations, see "Sample Driver File Creation" on page 3-5.

Customize the Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

2-17

- **Library information** — Information about the deployed application. You can also customize the appearance of the application 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** — Default installation path for the generated installer and custom logo selection. See "Change the Installation Path" on page 3-3.
- **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 3-4.
- **Files installed for your end user** — Files that are installed with your application.

  See "Specify Files to Install with Application" on page 3-6.

Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

1 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 packaging process is complete, examine the generated output in the target folder.

- Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

  For more information about the files generated in these folders, see "Files Generated After Packaging MATLAB Functions" on page 1-10.

- The log file `PackagingLog.html` contains packaging results.

Create C++ Shared Library Using compiler.build.cppSharedLibrary

As an alternative to the **Library Compiler** app, you can create a C++ shared library using a programmatic approach. If you have already created a library using the **Library Compiler**, see "Implement C++ MATLAB Data API Shared Library with Sample Application" on page 2-19.

1 Save the list of function files in a cell array.

```
functionfiles = {'addmatrix.m', 'multiplymatrix.m', 'eigmatrix.m'}
```

2 Create MATLAB sample code that calls the functions. Sample files are used to generate a sample application in the target language. For more information and limitations, see "Sample Driver File Creation" on page 3-5.

  Save the following code in a sample file named `libmatrixSample.m`:

```matlab
% Sample script to demonstrate execution of functions
% addmatrix, eigmatrix, and multiplymatrix
a1 = [1 4 7; 2 5 8; 3 6 9]; % Initialize a1 here
a2 = a1; % Initialize a2 here
a = addmatrix(a1, a2);
e = eigmatrix(a1);
m = multiplymatrix(a1, a2);
```

You may instead choose to not include a sample driver file at all during the packaging step. If you create your own C++ application code, you can move it to the appropriate directory and compile it using `mbuild` after the MATLAB functions are packaged.

3   Build the C++ shared library using the `compiler.build.cppSharedLibrary` function. Use name-value arguments to specify the library name and add the sample file.

```
buildResults = compiler.build.cppSharedLibrary(functionfiles,...
'LibraryName','libmatrix',...
'SampleGenerationFiles','libmatrixSample.m');
```

You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.cppSharedLibrary`.

The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

4   This syntax generates the following files within a folder named `libmatrixcppSharedLibrary` in your current working directory:

- `samples\libmatrixSample1_mda.cpp` — C++ sample application that calls the `addmatrix` function.

- `samples\libmatrixSample2_mda.cpp` — C++ sample application that calls the `eigmatrix` function.

- `samples\libmatrixSample3_mda.cpp` — C++ sample application that calls the `multiplymatrix` function.

- `v2\generic_interface\libmatrix.ctf` — Component technology file that contains the deployable archive.

- `v2\generic_interface\readme.txt` — Text file that contains packaging information.

- `GettingStarted.html` — HTML file that contains information on integrating your shared library.

- `includedSupportPackages.txt` — Text file that lists all support files included in the library.

- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.

- `readme.txt` — Text file that contains packaging and interface information.

- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.

- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Note   The generated library does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Implement C++ MATLAB Data API Shared Library with Sample Application

Note   To call the library using a more advanced application that calls all three functions and handles errors, use the C++ application `matrix_mda.cpp` located in the folder

matlabroot`\extern\examples\compilersdk\c_cpp\matrix`

For more details, see "Integrate C++ Shared Libraries with MATLAB Data API".

Before starting, make sure that you "Install and Configure MATLAB Runtime" and that you have a C++ compiler installed.

After packaging C++ shared libraries, you can call them from a C++ application. The C++ code generated in the `samples` folder is based on the sample MATLAB file you created.

1 Copy and paste the generated file `libmatrix.ctf` from the `v2\generic_interface` folder into the `samples` folder that contains `libmatrixSample1_mda.cpp`.

The program listing for `libmatrixSample1_mda.cpp` is shown below.

```
/*=================================================================
 *
 * ADDMATRIXSAMPLE1
 * Sample driver code that uses the generic interface and
 * MATLAB Data API to call a C++ shared library created using
 * MATLAB Compiler SDK.
 * Refer to the MATLAB Compiler SDK documentation for more
 * information.
 *
 *=================================================================*/

// Include the header file required to use the generic
// interface for the C++ shared library generated by the
// MATLAB Compiler SDK.
#include "MatlabCppSharedLib.hpp"
#include <iostream>

namespace mc = matlab::cpplib;
namespace md = matlab::data;

std::shared_ptr<mc::MATLABApplication> setup()
{
    auto mode = mc::MATLABApplicationMode::IN_PROCESS;
    // Specify MATLAB startup options
    std::vector<std::u16string> options = {};
    std::shared_ptr<mc::MATLABApplication> matlabApplication = mc::initMATLABApplication(mode, options);
    return matlabApplication;
}

int mainFunc(std::shared_ptr<mc::MATLABApplication> app, const int argc, const char * argv[])
{
    md::ArrayFactory factory;
    md::TypedArray<double> a1In = factory.createArray<double>({3, 3}, {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0});
    md::TypedArray<double> a2In = factory.createArray<double>({3, 3}, {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0});
    try {
        // The path to the CTF (library archive file) passed to
        // initMATLABLibrary or initMATLABLibraryAsync may be either absolute
        // or relative. If it is relative, the following will be prepended
        // to it, in turn, in order to find the CTF:
        // - the directory named by the environment variable
        // CPPSHARED_BASE_CTF_PATH, if defined
        // - the working directory
        // - the directory where the executable is located
        // - on Mac, the directory three levels above the directory
        // where the executable is located

        // If the CTF is not in one of these locations, do one of the following:
        // - copy the CTF
        // - move the CTF
        // - change the working directory ("cd") to the location of the CTF
        // - set the environment variable to the location of the CTF
        // - edit the code to change the path
        auto lib = mc::initMATLABLibrary(app, u"libmatrix.ctf");
        std::vector<md::Array> inputs{a1In, a2In};
        auto result = lib->feval(u"addmatrix", 1, inputs);
    } catch (const std::exception & exc) {
        std::cerr << exc.what() << std::endl;
        return -1;
    }
    return 0;
}
```

```
// The main routine. On the Mac, the main thread runs the system code, and
// user code must be processed by a secondary thread. On other platforms,
// the main thread runs both the system code and the user code.
int main(const int argc, const char * argv[])
{
    int ret = 0;
    try {
        auto matlabApplication = setup();
        ret = mc::runMain(mainFunc, std::move(matlabApplication), argc, argv);
        // Calling reset() on matlabApplication allows the user to control
        // when it is destroyed, which automatically cleans up its resources.
        // Here, the object would go out of scope and be destroyed at the end
        // of the block anyway, even if reset() were not called.
        // Whether the matlabApplication object is explicitly or implicitly
        // destroyed, initMATLABApplication() cannot be called again within
        // the same process.
        matlabApplication.reset();
    } catch(const std::exception & exc) {
        std::cerr << exc.what() << std::endl;
        return -1;
    }
    return ret;
}
```

2 At the MATLAB command prompt or your system command prompt, navigate to the `samples` folder where you copied `libmatrix.ctf`.

3 Compile and link the application using `mbuild` at the system command prompt.

```
mbuild libmatrixSample1_mda.cpp
```

4 Run the application from the system command prompt.

```
addmatrixSample1_mda.exe
```

By default, the generated C++ code does not display any output.

5 (Optional) Compile and link the other sample C++ applications using `mbuild`. You can also use the generated C++ code as a guide to create your own application.

For further details, see "Integrate C++ Shared Libraries with MATLAB Data API".

Note For an example on how to retrieve and display a struct array, a cell array, or a character vector from an `feval` call, see the files `subtractmatrix.m` and `subtractmatrix_mda.cpp` located in *matlabroot*\extern\examples\compilersdk\c_cpp\matrix.

See Also

compiler.build.cppSharedLibrary | **Library Compiler** | mcc | deploytool

Related Examples

- "Integrate C++ Shared Libraries with MATLAB Data API"
- "Call MATLAB Compiler SDK API Functions from C/C++"
- "Generate a C++ mwArray API Shared Library and Build a C++ Application" on page 2-10

Generate .NET Assembly and Build .NET Application

Supported platform: Windows

This example shows how to create a .NET assembly from a MATLAB function and integrate the generated assembly into a .NET application.

Prerequisites

- Verify that you have met all of the MATLAB Compiler SDK .NET target requirements. For details, see "MATLAB Compiler SDK .NET Target Requirements".

- Verify that you have Microsoft Visual Studio installed.

- End users must have an installation of MATLAB Runtime to run the application. For details, see "Install and Configure MATLAB Runtime".

  For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime.

Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want to package. For this example, create a MATLAB script named `makesquare.m`.

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

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

The output is a 5-by-5 matrix.

```
    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

Package the function into a .NET assembly using the **Library Compiler** app. Alternatively, if you want to create a .NET assembly from the MATLAB command window using a programmatic approach, see "Create .NET Assembly Using compiler.build.dotNETAssembly" on page 2-26.

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

   Alternatively, you can open the **Library Compiler** app from the MATLAB command prompt.

   ```
   libraryCompiler
   ```
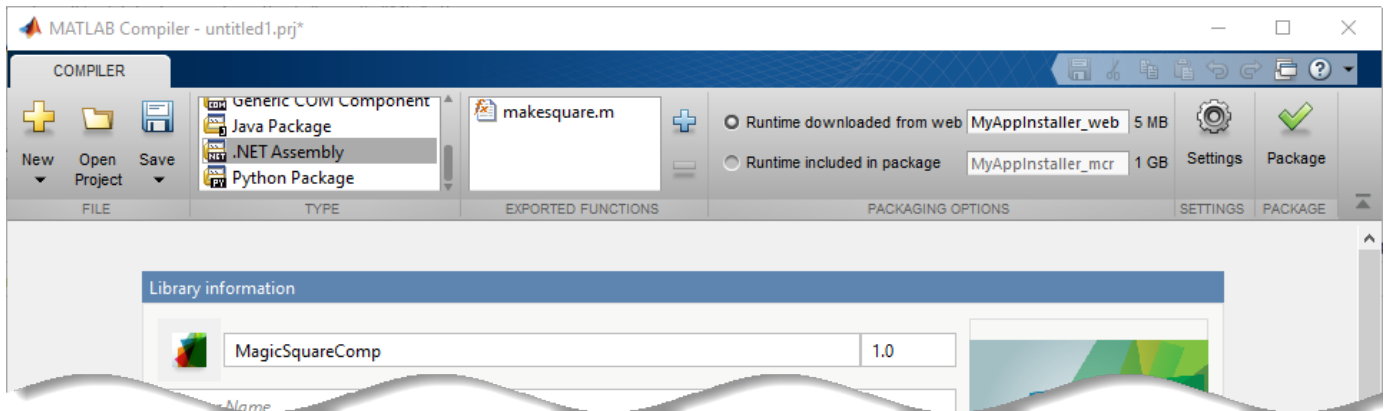
2   In the **Type** section of the toolstrip, click **.NET Assembly**.

In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

a   In the **Exported Functions** section of the toolstrip, click .

b   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 exported function files. Repeat this step to package multiple files in the same application.

For this example, select the file `makesquare.m`.

3   In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

- **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.

- **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer.

Specify Assembly File Settings

Next, define the name of your assembly and verify the class mapping for the `.m` file that you are building into your application.

1   The **Library Name** field is automatically populated with `makesquare` as the name of the assembly. Rename it as `MagicSquareComp`. The same name is followed through in the implementation of the assembly.

The **Library Name** field does not support spaces or special characters.

2   Verify that the function defined in `makesquare.m` is mapped into `MagicSquareClass`. Double-click on the class to change the class name.

Create Sample MATLAB File

You can use any MATLAB file in the project to generate a sample MATLAB file, which is used to generate a .NET sample driver file. Although .NET sample files are not necessary to create an assembly, you can use them as a guide to implement a .NET application in the target language, as shown in "Integrate .NET Assembly Into .NET Application" on page 2-27.

In the **Samples** section, select **Create New Sample**, and click `makesquare.m`. A MATLAB file opens for you to edit.

```matlab
% Sample script to demonstrate execution of function y = makesquare(x)
x = 0; % Initialize x here
y = makesquare(x);
```

Change `x = 0` to `x = 5`, save the file, and return to the **Library Compiler** app.

Caution You must edit the MATLAB sample file to output your desired result. Generated target language sample files use the same inputs and outputs as the sample MATLAB file.

For more information and limitations, see "Sample Driver File Creation" on page 3-5.

Customize Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application 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** — Default installation path for the generated installer and custom logo selection. See "Change the Installation Path" on page 3-3.

- **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 3-4.

- **Files installed for your end user** — Files that are installed with your application.

  See "Specify Files to Install with Application" on page 3-6.

- **Additional runtime settings** — Platform-specific options for controlling the generated executable. For details, see "Additional Runtime Settings" on page 3-7.

Library information

MagicSquareComp   1.0

Author Name

Email

Company

Set as default contact

Summary

Select custom splash screen

Description

Namespace

Class Name	Method Name
MagicSquareClass	[y] = makesquare (x)

Add Class

▼ Samples

Add MATLAB files that demonstrate how to execute the exported functions. These files will be used to generate sample driver files in the target language.

makesquareSampl...

Create New Sample   Add Existing Sample

▶ Additional installer options

Files required for your library to run

Files installed for your end user

MagicSquareCom...   MagicSquareCom...   MagicSquareCom...

▶ Additional runtime settings

Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

1   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 packaging process is complete, examine the generated output in the target folder.

- Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

  For more information about the files generated in these folders, see "Files Generated After Packaging MATLAB Functions" on page 1-10.

- The log file `PackagingLog.html` contains packaging results.

Create .NET Assembly Using compiler.build.dotNETAssembly

As an alternative to the **Library Compiler** app, you can create a .NET assembly using a programmatic approach. If you have already created an assembly using the **Library Compiler**, see "Integrate .NET Assembly Into .NET Application" on page 2-27.

1   If you have not already created the file `makesquare.m`, copy the example file located in *matlabroot*\toolbox\dotnetbuilder\Examples\VS15\NET\MagicSquareExample \MagicSquareComp.

```
copyfile(fullfile(matlabroot,'toolbox','dotnetbuilder','Examples',...
'VS15','NET','MagicSquareExample','MagicSquareComp','makesquare.m'));
```

2   Save the following code in a sample file named `makesquareSample1.m`:

```
x = 5;
y = makesquare(x);
```

3   Build the .NET assembly using the `compiler.build.dotNETAssembly` function. Use name-value arguments to specify the assembly name, class name, and sample file.

```
buildResults = compiler.build.dotNETAssembly(appFile,...
'AssemblyName','MagicSquareComp',...
'ClassName','MagicSquareClass',...
'SampleGenerationFiles','makesquareSample1.m');
```

You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.dotNETAssembly`.

The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

The function generates the following files within a folder named `MagicSquareCompdotNETAssembly` in your current working directory:

- `samples\makesquareSample1.cs` — C# .NET sample file.
- `GettingStarted.html` — HTML file that contains steps on compiling .NET driver applications from the command line.
- `includedSupportPackages.txt` — Text file that lists all support files included in the assembly.
- `MagicSquareComp.dll` — Dynamic-link library file that can be accessed using the `mwArray` API.
- `MagicSquareComp.xml` — XML file that contains documentation for the `mwArray` assembly.
- `MagicSquareComp_overview.html` — HTML file that contains requirements for accessing the assembly and for generating arguments using the `mwArray` class hierarchy.

- `MagicSquareCompNative.dll` — Dynamic-link library file that can be accessed using the native API.
- `MagicSquareCompNative.xml` — XML file that contains documentation for the native assembly.
- `MagicSquareCompVersion.cs` — C# file that contains version information.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Text file that contains packaging and interface information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

> **Note** The generated assembly does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Integrate .NET Assembly Into .NET Application

After creating your .NET assembly, you can integrate it into any .NET application. This example uses the sample C# application code generated during packaging. You can use this sample .NET application code as a guide to write your own application.

> **Note** To call the assembly using a more advanced application that takes an input argument, use the C# or Visual Basic application `MagicSquareApp` located in the corresponding subfolder of:
>
> *matlabroot*\toolbox\dotnetbuilder\Examples\VS15\NET\MagicSquareExample\

1 Open Microsoft Visual Studio and create a C# **Console App (.NET Framework)** called `MainApp`.
2 Remove any source code files that were created within your project, if necessary.
3 Add the sample C# application code `makesquareSample1.cs` that was generated in the `samples` folder to the project.

The program listing is shown below.

```
using System;
using System.Collections.Generic;
using System.Text;
using MathWorks.MATLAB.NET.Arrays;
using MathWorks.MATLAB.NET.Utility;
using MagicSquareComp;

/// <summary>
/// Sample driver code that integrates a compiled MATLAB function
/// generated by MATLAB Compiler SDK
///
/// Refer to the MATLAB Compiler SDK documentation for more
/// information.
/// </summary>
class makesquareSample1 {
    static MagicSquareClass MagicSquareClassInstance;

    static void Setup() {
        MagicSquareClassInstance = new MagicSquareClass();
    }
```

```
/// <summary>
/// Example of using the makesquare function.
/// </summary>
public static void makesquareSample() {
    double xInData = 5.0;
    MWNumericArray yOut = null;
    Object[] results = null;
    try {
        MWNumericArray xIn = new MWNumericArray(xInData);
        results = MagicSquareClassInstance.makesquare(1, xIn);
        if (results[0] is MWNumericArray) {
            yOut = (MWNumericArray) results[0];
        }
        Console.WriteLine(yOut);
    } catch (Exception e) {
        Console.WriteLine(e);
    }
}

/// <summary>
/// The main entry point for the application.
/// </summary>
static void Main(string[] args) {
    try {
        Setup();
    } catch (Exception e) {
        Console.WriteLine(e);
        Environment.Exit(1);
    }
    try {
        makesquareSample();
    } catch (Exception e) {
        Console.WriteLine(e);
        Environment.Exit(1);
    }
}
}
```

The program does the following:

- Uses a `try-catch` block to handle exceptions

- Creates an `MWNumericArray` array to store the input data

- Instantiates the `MagicSquareClass` object `results`

- Calls the `makesquare` method, where the first parameter specifies the number of output arguments and the following parameters are passed to the function in order as input arguments

- Writes the function output to the console

4 In Visual Studio, add a reference to your assembly file `MagicSquareComp.dll` located in the folder where you generated or installed the assembly.

5 Add a reference to the `MWArray` API.

If MATLAB is installed on your system	*matlabroot*\toolbox\dotnetbuilder\bin\win64\<*framework_version*>\MWArray.dll
If MATLAB Runtime is installed on your system	<*MATLAB_RUNTIME_INSTALL_DIR*>\toolbox\dotnetbuilder\bin\win64\<*framework_version*>\MWArray.dll

6 Go to **Build**, then **Configuration Manager**, and change the platform from **Any CPU** to **x64**.

7 After you finish adding your code and references, build the application with Visual Studio.

The build process generates an executable named `makesquareSample1.exe`.

8 Run the application from Visual Studio, in a command window, or by double-clicking the generated executable.

The application returns the same output as the sample MATLAB code.

```
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
```

See Also

compiler.build.dotNETAssembly | **Library Compiler** | deploytool | mcc

Related Examples

- "Build .NET Core Application That Runs on Linux and macOS"
- "Integrate Simple MATLAB Function into .NET 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 and integrate it into an application. The target system does not require a licensed copy of MATLAB.

Prerequisites

- Verify that you have the Windows 10 SDK kit installed. For details, see Windows 10 SDK.
- Verify that you have MinGW-w64 installed. To install it from the MathWorks File Exchange, see MATLAB Support for MinGW-w64 C/C++ Compiler.

  To ensure that MATLAB detects the Windows 10 SDK kit and MinGW-w64, use the following command:

  `mbuild -setup -client mbuild_com`
- Verify that you have Microsoft Visual Studio installed.
- End users must have an installation of MATLAB Runtime to run the application. For details, see "Install and Configure MATLAB Runtime".

  For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime.

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\VS*Version*\COM\MagicSquareExample\MagicSquareComp.

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

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

```
    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

Package the function into a COM component using the **Library Compiler** app. Alternatively, if you want to create a COM component from the MATLAB command window using a programmatic approach, see "Create COM Component Using compiler.build.COMComponent" on page 2-33.

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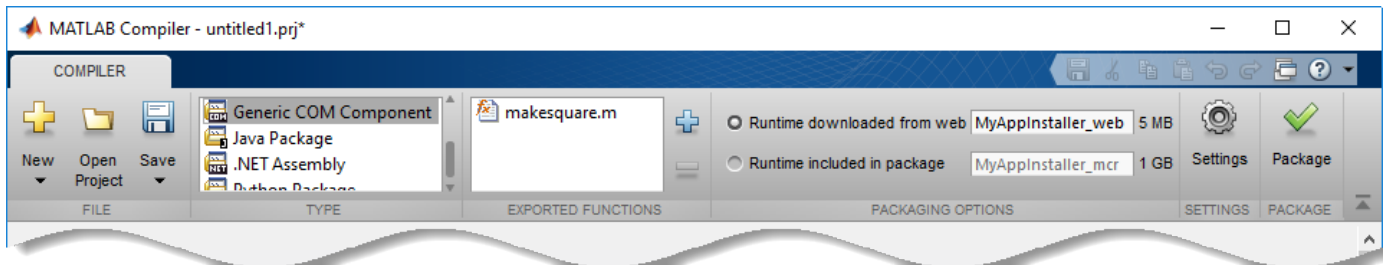
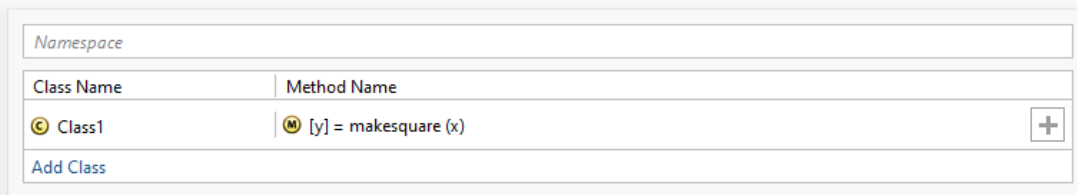Alternately, you can open the **Library Compiler** app by entering `libraryCompiler` at the MATLAB prompt.

2 In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

a In the **Exported Functions** section of the toolstrip, click [+].

b 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 exported function files. Repeat this step to package multiple files in the same application.

3 In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

- **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.

- **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer.

---

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

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application by changing the application icon and splash screen. The generated

2-31

installer uses this information to populate the installed application metadata. See "Customize the Installer" on page 3-2.

- **Additional installer options** — Default installation path for the generated installer and custom logo selection. See "Change the Installation Path" on page 3-3.
- **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 3-4.
- **Files installed for your end user** — Files that are installed with your application.

  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 3-7.

Library information

🔲	MagicSquareComp	1.0

Author Name

Email

Company

Set as default contact

Summary

Description

Select custom splash screen

Class Name	Method Name	
Ⓒ Class1	Ⓜ [m] = magicsquare (n)	+
Add Class		

▶ Additional installer options

Files required for your library to run

+

Files installed for your end user

📄 _install.bat   📄 MagicSquareCo...   📄 readme.txt

+

▶ Additional runtime settings

Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

1   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 packaging process is complete, examine the generated output in the target folder.

- Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

  For more information about the files generated in these folders, see "Files Generated After Packaging MATLAB Functions" on page 1-10.

- The log file `PackagingLog.html` contains packaging results.

Create COM Component Using compiler.build.COMComponent

As an alternative to the **Library Compiler** app, you can create a COM component using a programmatic approach. If you have already created a component using the **Library Compiler**, see "Integrate into COM Application" on page 2-34.

1   Save the path to the file `makesquare.m` located in *matlabroot*\toolbox\dotnetbuilder \Examples\VS*Version*\COM\MagicSquareExample\MagicSquareComp. For example, if you are using Visual Studio version 15, type:

```
appFile = fullfile(matlabroot,'toolbox','dotnetbuilder','Examples', ...
    'VS15','COM','MagicSquareExample','MagicSquareComp','makesquare.m');
```

2   Build the COM component using the `compiler.build.comComponent` function. Use name-value arguments to specify the component name and class name.

```
buildResults = compiler.build.comComponent(appFile, ...
'ComponentName','MagicSquareComp', ...
'ClassName','Class1');
```

You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.comComponent`.

The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

The function generates the following files within a folder named `MagicSquareCompcomComponent` in your current working directory:

- `magicsquare.def`
- `magicsquare.rc`
- `magicsquare_1_0.dll`
- `readme.txt`
- `requiredMCRProducts.txt`
- `unresolvedSymbols.txt`

- `Class1_com.cpp` — C++ source code file that defines the class.
- `Class1_com.hpp` — C++ header file that defines the class.
- `dlldata.c` — C source code file that contains entry points and data structures required by the class factory for the DLL.

- `GettingStarted.html` — HTML file that contains steps on installing COM components.
- `includedSupportPackages.txt` — Text file that contains information on included support packages.
- `MagicSquareComp.def` — Module definition file that defines which functions to include in the DLL export table.
- `MagicSquareComp.rc` — Resource script file that describes the resources used by the component.
- `MagicSquareComp_1_0.dll` — Dynamic-link library file.
- `MagicSquareComp_dll.cpp` — C++ source code file that contains helper functions.
- `MagicSquareComp_idl.h` — C++ header file.
- `MagicSquareComp_idl.idl` — Interface definition language file.
- `MagicSquareComp_idl.tlb` — Type library file that contains information about the COM object properties and methods.
- `MagicSquareComp_idl_i.c` — C source code file that contains the IIDs and CLSIDs for the IDL interface.
- `MagicSquareComp_idl_p.c` — C source code file that contains the proxy stub code for the IDL interface.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `mwcomtypes.h` — C++ header file that contains the definitions for the interfaces.
- `mwcomtypes_i.c` — C source code file that contains the IIDs and CLSIDs.
- `mwcomtypes_p.c` — C source code file that contains the proxy stub code.
- `readme.txt` — Text file that contains deployment information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Note The generated component does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Integrate into COM Application

To integrate your COM component into an application, see "Creating the Microsoft Visual Basic Project".

See Also
`compiler.build.comComponent` | **Library Compiler** | `mcc` | `deploytool`

More About
- "Call COM Objects in Visual C++ Programs"

Generate Java Package and Build Java Application

Supported platforms: Windows, Linux, Mac

This example shows how to create a Java package from a MATLAB function and generate sample Java code.

Prerequisites

- Verify that you have a version of Java installed that is compatible with MATLAB Compiler SDK. For information on supported Java versions, see MATLAB Interfaces to Other Languages.

  For information on configuring your development environment after installation, see "Configure Your Environment for Generating Java Packages".

- End users must have an installation of MATLAB Runtime to run the application. For details, see "Install and Configure MATLAB Runtime".

  For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime.

Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want to package. 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 a 5-by-5 matrix.

```
    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
```
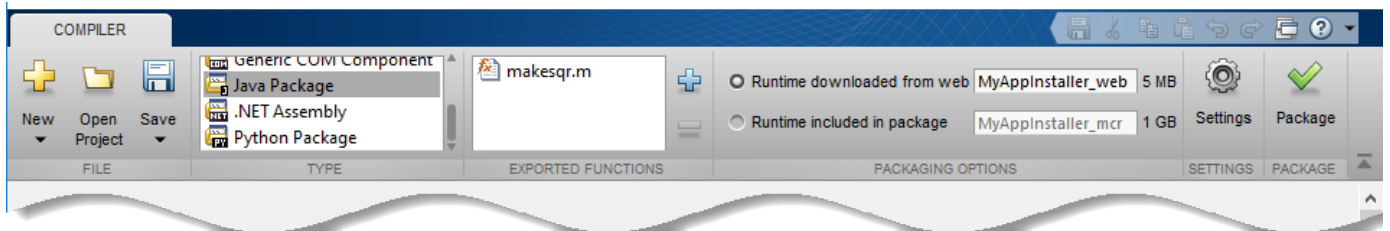
Create Java Package Using Library Compiler App

Compile the function into a Java package using the **Library Compiler** app. Alternatively, if you want to create a Java package from the MATLAB command window using a programmatic approach, see "Create Java Package Using compiler.build.javaPackage" on page 2-39.

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

   Alternatively, you can open the **Library Compiler** app from the MATLAB command prompt by entering:

   ```
   libraryCompiler
   ```

2   In the **Type** section of the toolstrip, click **Java Package**.

In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

a   In the **Exported Functions** section of the toolstrip, click ⊞.

b   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 exported function files. Repeat this step to package multiple files in the same application.

For this example, select the file `makesqr.m`.

3   In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

• **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.

• **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

Note The first time you select this option, you are prompted to download the MATLAB Runtime installer.

Specify Package Settings

Next, define the name of your Java package and verify the class mapping for the `.m` file that you are building into your application.

1   Choose a name for your package. The **Library Name** field is automatically populated with `makesqr` as the name of the package. The same name is followed through in the package implementation steps below.

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

Create Sample Driver File

You can use any MATLAB file in the project to generate sample Java driver files. Although Java driver files are not necessary to create a package, you can use them to implement a Java application, as shown in "Compile and Run MATLAB Generated Java Application" on page 2-40.

In the **Samples** section, select **Create New Sample**, and click `makesqr.m`. A MATLAB file opens for you to edit.

```
% Sample script to demonstrate execution of function y = makesqr(x)
x = 0; % Initialize x here
y = makesqr(x);
```

Change `x = 0` to `x = 5`, save the file, and return to the **Library Compiler** app.

Caution You must edit the MATLAB sample file to output your desired result. Generated target language sample files use the same inputs and outputs as the sample MATLAB file.

The compiler converts this MATLAB code to Java code during packaging. For more information and limitations, see "Sample Driver File Creation" on page 3-5.

Customize the Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.
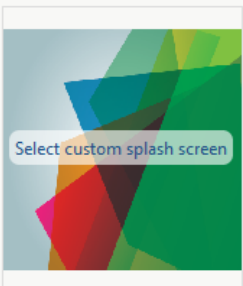
- **Library information** — Information about the deployed application. You can also customize the appearance of the application by changing the application icon and splash screen. The generated installer uses this information to populate the installed application metadata. See "Customize the Installer" on page 3-2.
- **Additional installer options** — Default installation path for the generated installer and custom logo selection. See "Change the Installation Path" on page 3-3.
- **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 3-4.
- **Files installed for your end user** — Files that are installed with your application.

  See "Specify Files to Install with Application" on page 3-6.

Library information

makesqr   1.0

Author Name

Email

Company

                                          Set as default contact

Summary

Description

Select custom splash screen

Namespace

Class Name	Method Name
Ⓒ Class1	Ⓜ [y] = makesqr (x)

Add Class

▼ Samples

Add MATLAB files that demonstrate how to execute the exported functions. These files will be used to generate sample driver files in the target language.

makesqrSample1...

Create New Sample   Add Existing Sample

▶ Additional installer options

Files required for your library to run

Files installed for your end user

doc   makesqr.jar

Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

1 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 packaging process is complete, examine the generated output in the target folder.

- Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

  For more information about the files generated in these folders, see "Files Generated After Packaging MATLAB Functions" on page 1-10.

- The log file `PackagingLog.html` contains packaging results.

Create Java Package Using compiler.build.javaPackage

As an alternative to the **Library Compiler** app, you can create a Java package using a programmatic approach. If you have already created a package using the **Library Compiler**, see "Compile and Run MATLAB Generated Java Application" on page 2-40.

1 Save the path to the `makesqr.m` file located in *matlabroot*\toolbox\javabuilder \Examples\MagicSquareExample\MagicDemoComp.

```
appFile = fullfile(matlabroot,'toolbox','javabuilder','Examples', ...
    'MagicSquareExample','MagicDemoComp','makesqr.m');
```

2 Save the following code in a sample file named `makesqrSample1.m`:

```
x = 5;
y = makesqr(x);
```

3 Build the Java package using the `compiler.build.javaPackage` function. Use name-value arguments to add a sample file and enable verbose output.

```
buildResults = compiler.build.javaPackage(appFile, ...
'SampleGenerationFiles','makesqrSample1.m', ...
'Verbose','on');
```

You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.javaPackage`.

The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

The function generates the following files and folders within a folder named `makesqrjavaPackage` in your current working directory:

- `classes` — Folder that contains the Java class files and the deployable archive CTF file.
- `doc` — Folder that contains HTML documentation for all classes in the package.
- `example` — Folder that contains Java source code files.
- `samples` — Folder that contains the Java sample driver file `makesqrSample1.java`.
- `GettingStarted.html` — File that contains information on integrating your package.
- `includedSupportPackages.txt` — Text file that lists all support files included in the package.
- `makesqr.jar` — Java archive file.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see Functions Not Supported For Compilation.

- `readme.txt` — Text file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Note The generated package does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Compile and Run MATLAB Generated Java Application

After creating your Java package, you can call it from a Java application. This example uses the sample Java code generated during packaging. You can use this sample Java application code as a guide to write your own application.

1 Copy and paste the generated Java file `makesqrSample1.java` from the `samples` folder into the folder that contains the `makesqr.jar` package. If you used the Library Compiler, `makesqr.jar` is located in the `for_testing` folder.
2 At the system command prompt, navigate to the folder that contains `makesqrSample1.java` and `makesqr.jar`.
3 Compile the application using `javac`. In the classpath argument, you specify the paths to `javabuilder.jar`, which contains the `com.mathworks.toolbox.javabuilder` package, and your generated Java package `makesqr.jar`.

- On Windows, type:

  ```
  javac -classpath "matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr.jar makesqrSample1.java
  ```
- On UNIX, type:

  ```
  javac -classpath "matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./makesqr.jar makesqrSample1.java
  ```

  Replace *matlabroot* with the path to your MATLAB or MATLAB Runtime installation folder. For example, on Windows, the path may be `C:\Program Files\MATLAB\R2023a`.

  Note If `makesqr.jar` or `makesqrSample1.java` is not in the current directory, specify the full or relative path in the command. If the path contains spaces, surround it with double quotes.

4 Run the application using `java`.

- On Windows, type:

  ```
  java -classpath .;"matlabroot\toolbox\javabuilder\jar\javabuilder.jar";.\makesqr.jar makesqrSample1
  ```
- On UNIX, type:

  ```
  java -classpath .:"matlabroot/toolbox/javabuilder/jar/javabuilder.jar":./makesqr.jar makesqrSample1
  ```

  Note The dot (`.`) in the first position of the class path represents the current working directory. If it is not there, you get a message stating that Java cannot load the class.

The application returns the same output as the sample MATLAB code.

```
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
```

See Also

Library Compiler | `compiler.build.javaPackage` | `mcc` | `deploytool`

Related Examples

- "Integrate Simple MATLAB Function into Java Application"
- "Display MATLAB Plot in Java Application"

Generate a Python Package and Build a Python Application

Supported platforms: Windows, Linux, Mac

This example shows how to create a Python package from a MATLAB function and integrate the generated package into a Python application.

Prerequisites

- Verify that you have a version of Python installed that is compatible with MATLAB Compiler SDK. For details, see MATLAB Supported Interfaces to Other Languages.
- End users must have an installation of MATLAB Runtime to run the application. For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime. For details, see "Install and Configure MATLAB Runtime".

Create Function in MATLAB

In MATLAB, examine the MATLAB code that you want packaged. For this example, create a function named `makesqr.m` that contains the following code:

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

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

The output is a 5-by-5 matrix.

```
    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

Compile the function into a Python package using the **Library Compiler** app. Alternatively, if you want to create a Python package from the MATLAB command window using a programmatic approach, see "Create Python Package Using compiler.build.pythonPackage" on page 2-46.

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

Alternatively, you can open the **Library Compiler** app from the MATLAB command prompt.

```
libraryCompiler
```
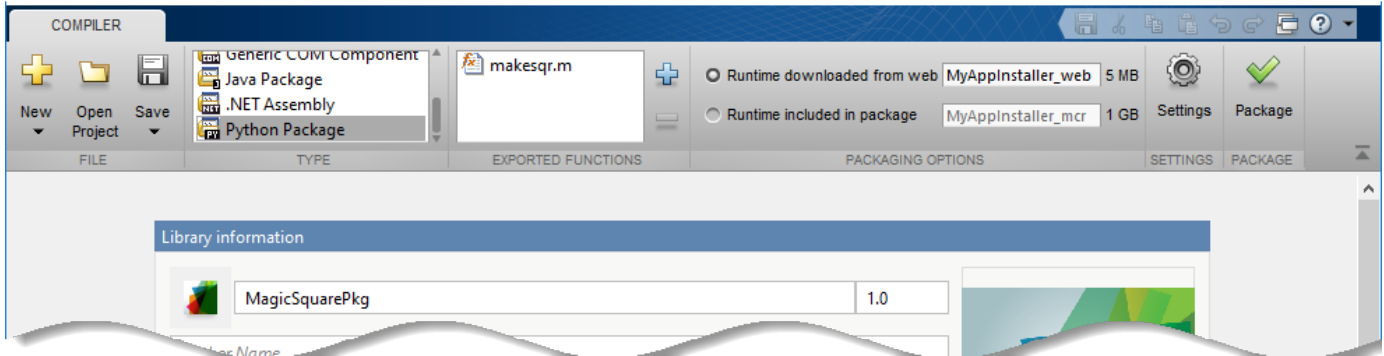
2 In the **Type** section of the toolstrip, click **Python Package**.

In the **Library Compiler** app project window, specify the files of the MATLAB application that you want to deploy.

   a In the **Exported Functions** section of the toolstrip, click ![plus icon].

   b 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 exported function files. Repeat this step to package multiple files in the same application.

For this example, select the `makesqr.m` file that you wrote earlier.

3 In the **Packaging Options** section of the toolstrip, decide whether to include the MATLAB Runtime installer in the generated application by selecting one of the options:

- **Runtime downloaded from web** — Generate an installer that downloads the MATLAB Runtime and installs it along with the deployed MATLAB application. You can specify the file name of the installer.

- **Runtime included in package** — Generate an application that includes the MATLAB Runtime installer. You can specify the file name of the installer.

> **Note** The first time you select this option, you are prompted to download the MATLAB Runtime installer.

Specify Package Settings

Next, define the name of your Python package.

- Choose a name for your package. The **Library Name** field is automatically populated with `makesqr` as the name of the package. Rename it as `MagicSquarePkg`. For more information on naming requirements for the Python package, see "Install and Import MATLAB Compiler SDK Python Packages".

Create Sample Driver File

You can add MATLAB files to the project to generate sample Python driver files. Although Python driver files are not necessary to create a package, you can use them to implement a Python application, as shown in "Install and Run MATLAB Generated Python Application" on page 2-47.

In the **Samples** section, select **Create New Sample**, and click `makesqr.m`. A MATLAB file opens for you to edit.

```
% Sample script to demonstrate execution of function y = makesqr(x)
x = 0; % Initialize x here
y = makesqr(x);
```

Change `x = 0` to `x = 5`, save the file, and return to the **Library Compiler** app.

Caution You must edit the MATLAB sample file to output your desired result. Generated target language sample files use the same inputs and outputs as the sample MATLAB file.

The compiler converts this MATLAB code to Python code during packaging. For more information and limitations, see "Sample Driver File Creation" on page 3-5.

Customize the Application and Its Appearance

In the **Library Compiler** app, you can customize the installer, customize your application, and add more information about the application.

- **Library information** — Information about the deployed application. You can also customize the appearance of the application 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** — Default installation path for the generated installer and custom logo selection. See "Change the Installation Path" on page 3-3.
- **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 3-4.
- **Files installed for your end user** — Files that are installed with your application.

  See "Specify Files to Install with Application" on page 3-6.

Package the Application

When you are finished selecting your packaging options, save your **Library Compiler** project and generate the packaged application.

1   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 packaging process is complete, examine the generated output in the target folder.

- Three folders are generated: `for_redistribution`, `for_redistribution_files_only`, and `for_testing`.

  For more information about the files generated in these folders, see "Files Generated After Packaging MATLAB Functions" on page 1-10.

- The log file `PackagingLog.html` contains packaging results.

Create Python Package Using compiler.build.pythonPackage

As an alternative to the **Library Compiler** app, you can create a Python package using a programmatic approach. If you have already created a package using the **Library Compiler**, see "Install and Run MATLAB Generated Python Application" on page 2-47.

1 Save the following code in a sample file named `makesqrSample1.m`:

```
x = 5;
y = makesqr(x);
```
2 Build the Python package using the `compiler.build.pythonPackage` function and the `makesqr.m` file that you wrote earlier. Use name-value arguments to specify the package name and add a sample file.

```
buildResults = compiler.build.pythonPackage('makesqr.m',...
'PackageName','MagicSquarePkg',...
'SampleGenerationFiles','makesqrSample1.m',...
'Verbose','on');
```

You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.pythonPackage`.

The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

3 The function generates the following files within a folder named `MagicSquarePkgpythonPackage` in your current working directory:

- `samples\makesqrSample1.py` — Python sample application file.
- `GettingStarted.html` — HTML file that contains information on integrating your package.
- `includedSupportPackages.txt` — Text file that lists all support files included in the package.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `pyproject.toml` — Configuration file that contains build system requirements and information, which are used by `pip` to build the package. For details, see `pip.pypa.io/en/stable/reference/build-system/pyproject-toml`.
- `readme.txt` — Text file that contains packaging and interface information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `setup.py` — Python file that installs the package.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Note The generated package does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Install and Run MATLAB Generated Python Application

After creating your Python package, you can call it from a Python application. This example uses the sample Python code generated during packaging. You can use this sample Python application code as a guide to write your own application.

1 Copy and paste the generated Python file `makesqrSample1.py` from the `samples` folder into the folder that contains the `setup.py` file.

The program listing for `makesqrSample1.py` is shown below.

```python
#!/usr/bin/env python
"""
Sample script that uses the MagicSquarePkg package created using
MATLAB Compiler SDK.

Refer to the MATLAB Compiler SDK documentation for more information.
"""

import MagicSquarePkg
# Import the matlab module only after you have imported
# MATLAB Compiler SDK generated Python modules.
import matlab

try:
    my_MagicSquarePkg = MagicSquarePkg.initialize()
except Exception as e:
    print('Error initializing MagicSquarePkg package\\n:{}'.format(e))
    exit(1)

try:
    xIn = matlab.double([5], size=(1, 1))
    yOut = my_MagicSquarePkg.makesqr(xIn)
    print(yOut, sep='\\n')
except Exception as e:
    print('Error occurred during program execution\\n:{}'.format(e))

my_MagicSquarePkg.terminate()
```

2 At the system command prompt, navigate to the folder that contains `makesqrSample1.py` and `setup.py`.

3 Install the application using the `python` command.

```
python setup.py install
```

To install to a location other than the default, consult "Installing Python Modules" in the official Python documentation.

4 Run the application at the system command prompt.

```
python makesqrSample1.py
```

If you used sample MATLAB code in the packaging steps, this application returns the same output as the sample code.

```
[[17.0,24.0,1.0,8.0,15.0],[23.0,5.0,7.0,14.0,16.0],[4.0,6.0,13.0,20.0,22.0],
[10.0,12.0,19.0,21.0,3.0],[11.0,18.0,25.0,2.0,9.0]]
```

Note On macOS, you must use the `mwpython` script instead of `python`. For example, `mwpython makesqrSample1.py`.

The `mwpython` script is located in the *`matlabroot`*`/bin` folder, where *`matlabroot`* is the location of your MATLAB or MATLAB Runtime installation.

See Also

`mwpython` | `compiler.build.pythonPackage` | **Library Compiler** | `mcc` | `deploytool`

Related Examples

- "Install and Import MATLAB Compiler SDK Python Packages"
- "Create Python Application with Multiple MATLAB Functions"

Customizing a Compiler Project

Customize an Application

You can customize an 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.
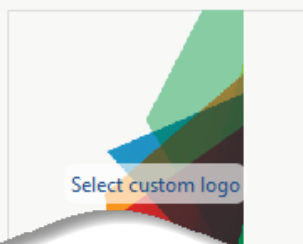
Windows	`C:\Program Files\`*companyName*`\`*appName*
Mac OS X	`/Applications/`*companyName*`/`*appName*
Linux	`/usr/`*companyName*`/`*appName*

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*userName*\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. You can 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.

Sample Driver File Creation

Sample driver files are used to implement the generated component into an application in the target language.

The following target types support sample driver file creation in MATLAB Compiler SDK:

- C++ shared library
- Java package
- .NET assembly
- Python package
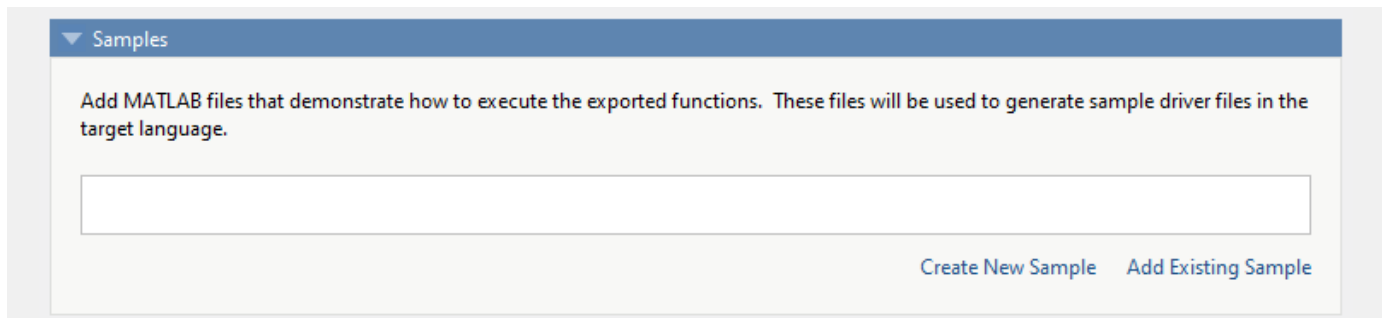
The sample file creation feature in **Library Compiler** uses MATLAB code to generate sample files in the target language. In the app, click **Create New Sample** to automatically generate a new MATLAB script, or click **Add Existing Sample** to upload a MATLAB script that you have already written. After you package your functions, a sample file in the target language is generated from your MATLAB script and is saved in a folder named `samples`. Sample files are also included in the installer.

To automatically generate a new MATLAB file, click **Create New Sample**. This opens up a MATLAB file for you to edit. The sample file serves as a starting point, and you should edit it as necessary based on the behavior of your exported functions.

The sample MATLAB files must follow these guidelines:

- The sample file must be a MATLAB script, not a function.
- The sample file code must use only exported functions. Any user-defined function called in the script must be a top-level exported function.
- Each exported function must be in a separate sample file.
- Each call to the same exported function must be a separate sample file.
- The input parameters of the top-level function are analyzed during the process. An input parameter cannot be a field in a struct.
- The output of the exported function must be an n-dimensional numeric, char, logical, struct, or cell array.
- Data must be saved as a local variable and then passed to the exported function in the sample file code.
- Sample file code should not require user interaction.
- The sample script is executed as part of the process of generating the target language sample code. Any errors in execution (for instance, undefined variables) will prevent a sample from being generated, although the build target will still be generated.

Additional considerations specific to the target language are as follows:

- C++ mwArray API — `varargin` and `varargout` are not supported.
- .NET — Type-safe API is not supported.
- Python — Cell arrays and char arrays must be of size 1xN and struct arrays must be scalar. There are no restrictions on numeric or logical arrays, other than that they must be rectangular, as in MATLAB.

To upload a MATLAB file that you have already written, click **Add Existing Sample**. The MATLAB code should demonstrate how to execute the exported functions. The required MATLAB code can be only a few lines:

```
input1 = [1 4 7; 2 5 8; 3 6 9];
input2 = [1 4 7; 2 5 8; 3 6 9];
addoutput = addmatrix(input1,input2);
```

This code must also follow all the same guidelines outlined for the **Create New Sample** option.

If you have already created a MATLAB sample file, you can include it in a `compiler.build` function for the supported targets using the `'SampleGenerationFiles'` option.

You can also choose not to include a sample file at all during the packaging step. If you create your own code in the target language, you can later copy and paste it into the appropriate directory once the MATLAB functions are packaged.

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`.


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	Additional Runtime Settings Options
Generic COM Components	• **Register the component for the current user (Recommended for non-admin users)** —This option enables registering the component for the current user account. It is provided for users without admin rights.
.NET Assembly	• **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 Remotable .NET Assembly". • **Enable Type Safe API** — Enables the type safe API for the packaged .NET assembly.

API Selection for C++ Shared Library
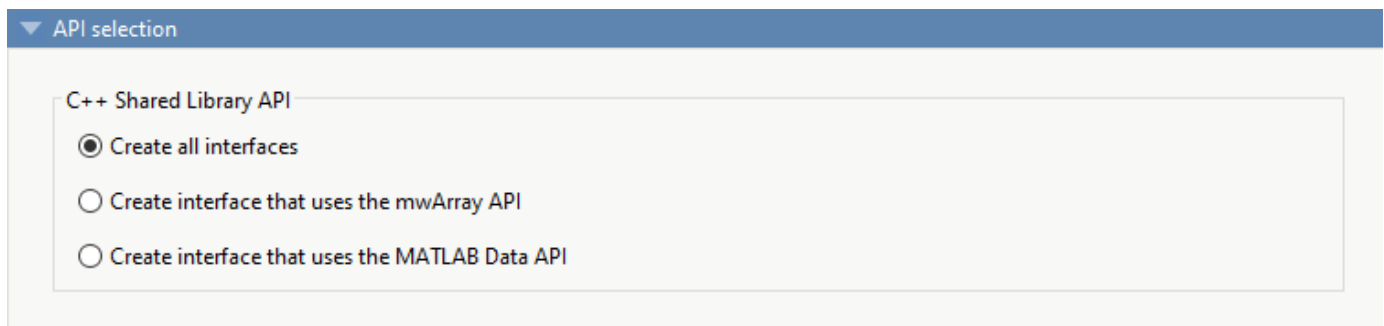
- **Create all interfaces** — Create interfaces for shared libraries using both the `mwArray` API and the MATLAB Data API.
- **Create interface that uses the mwArray API** — Create an interface for a shared library using the `mwArray` API. The interface uses C-style functions to initialize the MATLAB 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 supports only C++03 functionality. For an example, see "Generate a C++ mwArray API Shared Library and Build a C++ Application" on page 2-10.
- **Create interface that uses the MATLAB Data API** — Create 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 "Generate a C++ MATLAB Data API Shared Library and Build a C++ Application" on page 2-16.
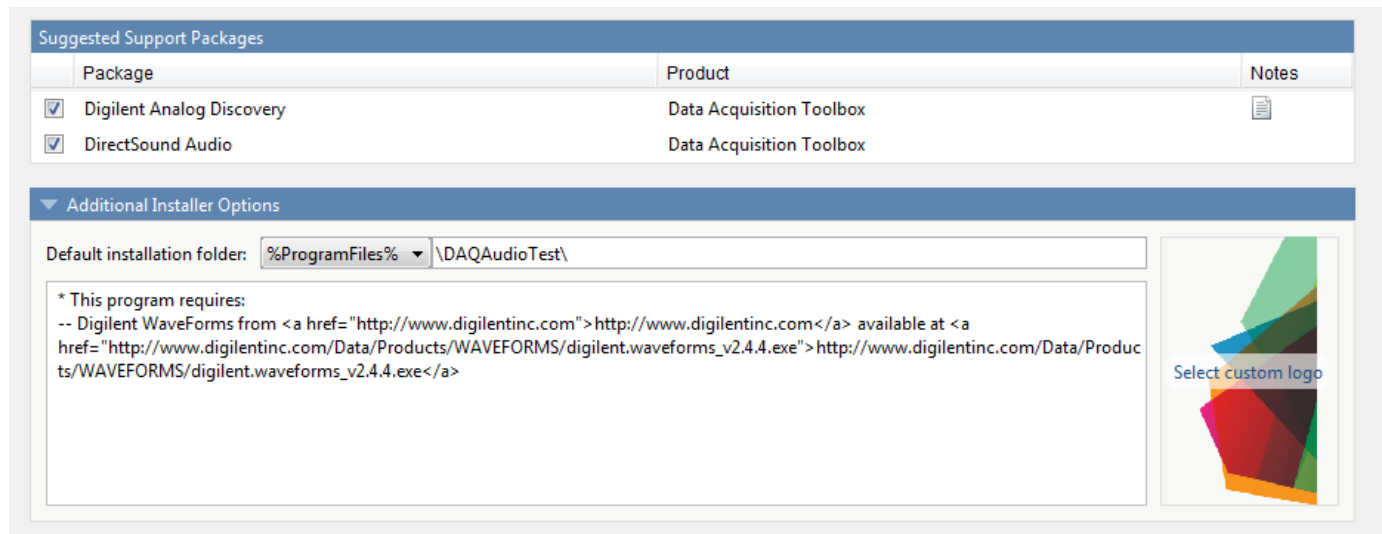
See Also
Application Compiler | Library Compiler

More About

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\R2023a\toolbox\daq\supportpackages\daqaudio ...
-a 'C:\MATLAB\SupportPackages\R2023a\resources\daqaudio'
```

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 Deployable Archive for MATLAB Production Server

Supported platform: Windows, Linux, Mac

Note To create a deployable archive, you need an installation of the MATLAB Compiler SDK product.

This example shows how to create a deployable archive using a MATLAB function. You can then deploy the generated archive on MATLAB Production Server.

Create MATLAB Function

In MATLAB, examine the MATLAB program that you want to package.

For this example, write a function `addmatrix.m` as follows.

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

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
```

Create Deployable Archive with Production Server Compiler App

Package the function into a deployable archive using the Production Server Compiler app. Alternatively, if you want to create a deployable archive from the MATLAB command window using a programmatic approach, see "Create Deployable Archive Using compiler.build.productionServerArchive".

1   To open the **Production Server Compiler** app, type `productionServerCompiler` at the MATLAB prompt.

    Alternatively, on the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Production Server Compiler**. In the **Production Server Compiler** project window, click **Deployable Archive (.ctf)**.

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

    1   In the **Exported Functions** section, click .

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

        Click **Open**.

    Doing so adds the function `addmatrix.m` to the list of main files.
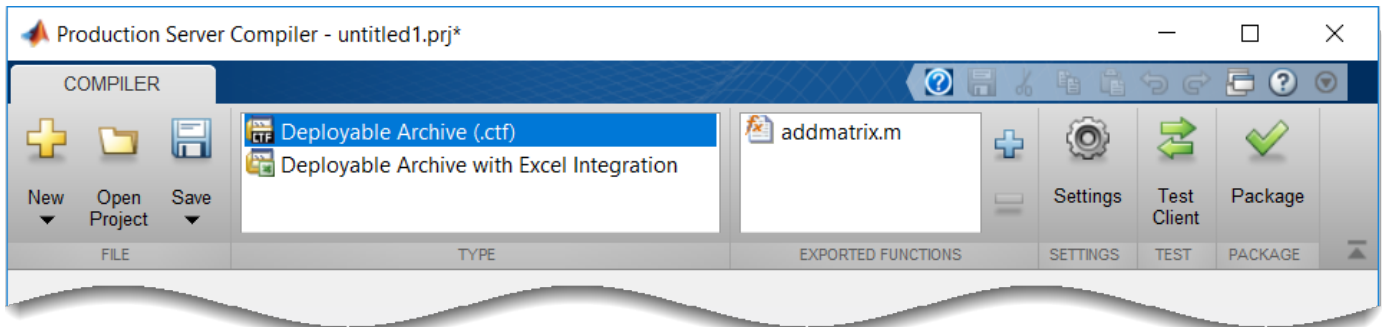
Customize Application and Its Appearance

Customize your deployable archive and add more information about the application.

- **Archive information** — Editable information about the deployed archive.
- **Additional files required for your archive to run** — Additional files required to run the generated archive. These files are included in the generated archive installer. See "Manage Required Files in Compiler Project" on page 3-4.
- **Files packaged for redistribution** — Files that are installed with your archive. These files include:

  - Generated deployable archive
  - Generated `readme.txt`

  See "Specify Files to Install with Application" on page 3-6.

- **Include MATLAB function signature file** — Add or create a function signature file to help clients use your MATLAB functions. See "MATLAB Function Signatures in JSON".
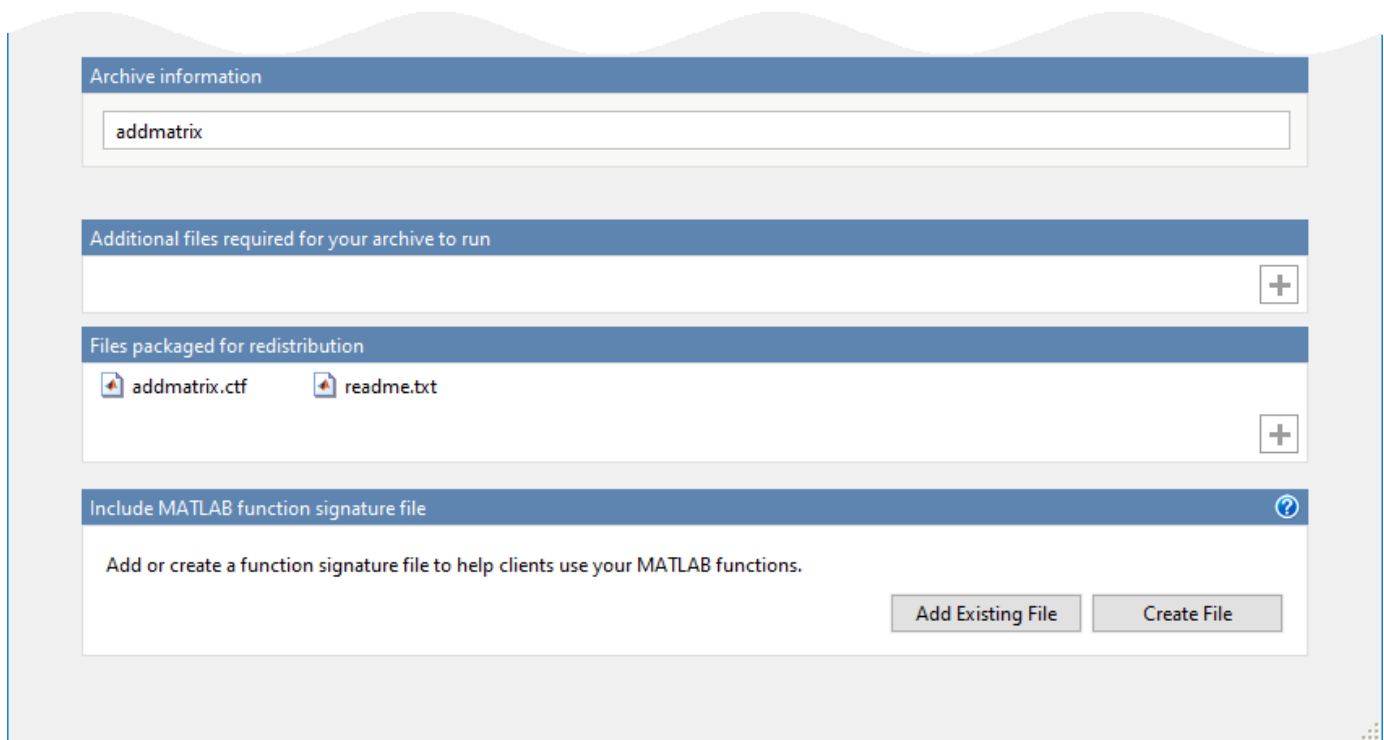
Package Application

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

In the Save Project dialog box, specify the location to save the project.
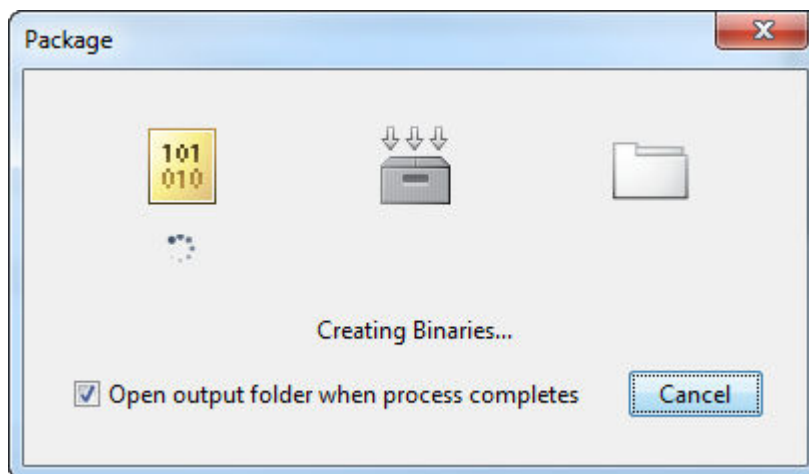
2 In the **Package** dialog box, verify that **Open output folder when process completes** is selected.

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

- `for_redistribution` — Folder containing the archive *archiveName*`.ctf`
- `for_testing` — Folder containing the raw generated files to create the installer

- `PackagingLog.html` — Log file generated by MATLAB Compiler SDK

Create Deployable Archive Using compiler.build.productionServerArchive

As an alternative to the **Production Server Compiler** app, you can create a deployable archive using a programmatic approach.

- Build the deployable archive using the `compiler.build.productionServerArchive` function.

  Optionally, you can add a function signature file to help clients use your MATLAB functions. For more details, see "MATLAB Function Signatures in JSON".

  ```
  buildResults = compiler.build.productionServerArchive('addmatrix.m',...
  'FunctionSignatures','addmatrixFunctionSignatures.json',...
  'Verbose','on');

  buildResults =

    Results with properties:

                   BuildType: 'productionServerArchive'
                       Files: {'/home/mluser/Work/magicarchiveproductionServerArchive/addmatri
      IncludedSupportPackages: {}
                     Options: [1×1 compiler.build.ProductionServerArchiveOptions]
  ```

  You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.productionServerArchive`.

  The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

  The function generates the following files within a folder named `addmatrixproductionServerArchive` in your current working directory:

- `addmatrix.ctf` — Deployable archive file.
- `includedSupportPackages.txt` — Text file that lists all support files included in the assembly.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Text file that contains packaging and deployment information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Compatibility Considerations

In most cases, you can generate the deployable archive on one platform and deploy to a server running on any other supported platform. Unless you add operating system-specific dependencies or content, such as MEX files or Simulink simulations to your applications, the generated archives are platform-independent.

See Also

`compiler.build.productionServerArchive` | `deploytool` | **Production Server Compiler** | `mcc`

More About

- "Test Client Data Integration Against MATLAB"
- Production Server Compiler
- "Deploy Archive to MATLAB Production Server" (MATLAB Production Server)
- "MATLAB Function Signatures in JSON"
- "JSON Representation of MATLAB Data Types" (MATLAB Production Server)

Create and Install a Deployable Archive with Excel Integration for MATLAB Production Server

Supported Platform: Microsoft Windows only.

This example shows how to create a deployable archive with Excel integration using a MATLAB function. You can then deploy the generated archive on MATLAB Production Server.

Prerequisites

MATLAB Compiler SDK requires .NET framework 4.0 or later to build Excel add-ins for MATLAB Production Server.

To generate the Excel add-in file (.xla), enable **Trust access to the VBA project object model** in Excel. If you do not do this, you can manually create the add-in by importing the `.bas` file into Excel.

Create Function in MATLAB

In MATLAB, examine the MATLAB program that you want to package.

For this example, write a function `mymagic.m` as follows.

```matlab
function y = mymagic(x)

y = magic(x);
```

At the MATLAB command prompt, enter `mymagic(3)`.

The output is:

```
ans =
    8    1    6
    3    5    7
    4    9    2
```

Create Deployable Archive with Excel Integration Using Production Server Compiler App

Package the function into a deployable archive with Excel integration using the Production Server Compiler app. Alternatively, if you want to create a deployable archive from the MATLAB command window using a programmatic approach, see "Create Deployable Archive with Excel Integration Using compiler.build.excelClientForProductionServer".

1   To open the **Production Server Compiler** app, type `productionServerCompiler` at the MATLAB prompt.

    Alternatively, on the **MATLAB Apps** tab, on the far right of the **Apps** section, click the arrow. In **Application Deployment**, click **Production Server Compiler**. In the **Production Server Compiler** project window, click **Deployable Archive with Excel integration**.
2   In the **Production Server Compiler** project window, specify the main file of the MATLAB application that you want to deploy.

1 In the **Exported Functions** section, click ⊞.

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

Click **Open**.

Doing so adds the function `mymagic.m` to the list of main files.

Customize the Application and Its Appearance

Customize your deployable archive with Excel integration and add more information about the application.

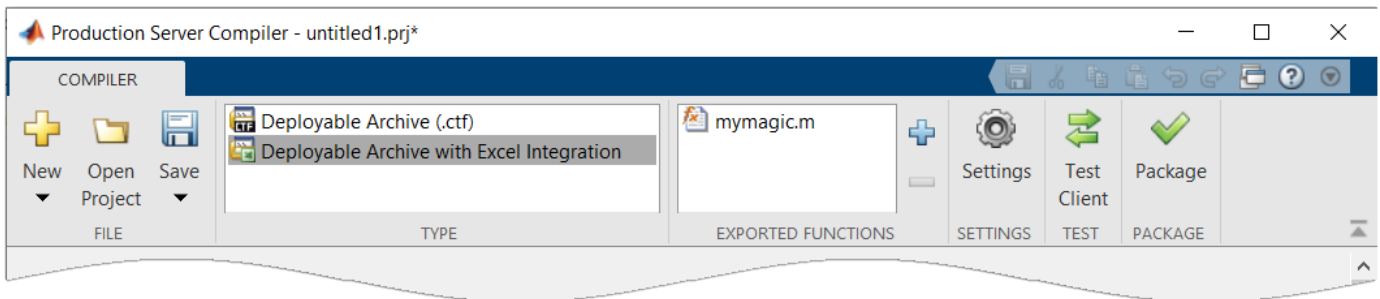- **Archive information** — Editable information about the deployed archive with Excel integration.

- **Client configuration** — Configure the MATLAB Production Server client. Select the **Default Server URL**, decide wait time-out, and maximum size of response for the client, and provide an optional self-signed certificate for `https`.

- **Additional files required for your archive to run** — Additional files required by the generated archive to run. These files are included in the generated archive installer. See "Manage Required Files in Compiler Project" on page 3-4.

- **Files installed with your archive** — Files that are installed with your archive on the client and server. The files installed on the server include:

  - Generated deployable archive (CTF file)
  - Generated `readme.txt`

  The files installed on the client include:

  - `mymagic.bas`
  - `mymagic.dll`
  - `mymagic.xla`
  - `readme.txt`
  - `ServerConfig.dll`

  See "Specify Files to Install with Application" on page 3-6.

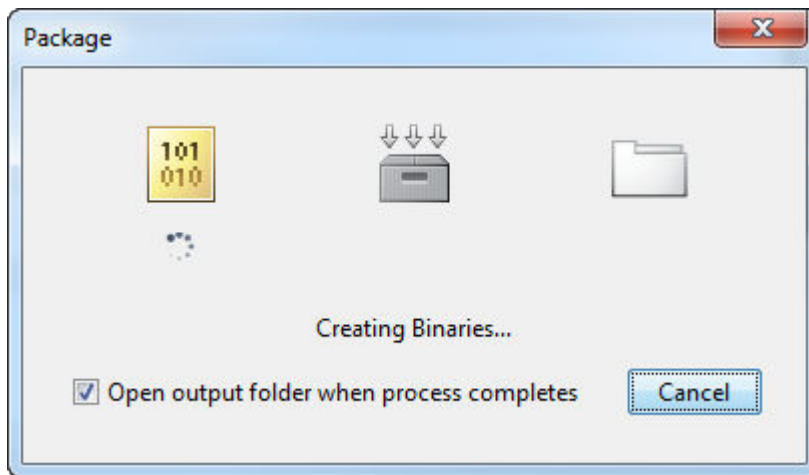- **Options** — The option **Register the resulting component for you only on the development machine** exclusively registers the packaged component for one user on the development machine.

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 installer to distribute the archive on the MATLAB Production Server client and server
    - `for_redistribution_files_only` — Folder containing the files required for redistributing the application on the MATLAB Production Server client and server
    - `for_testing` — Folder containing the raw generated files to create the installer
    - `PackagingLog.html` — Log file generated by MATLAB Compiler SDK

Create Deployable Archive with Excel Integration Using compiler.build.excelClientForProductionServer

As an alternative to the **Production Server Compiler** app, you can create a deployable archive with Excel integration using a programmatic approach.

1   Create a production server archive using `mymagic.m` and save the build results to a `compiler.build.Results` object.

    ```
    buildResults = compiler.build.productionServerArchive('mymagic.m');
    ```
2   Build the deployable archive with Excel integration using the `compiler.build.excelClientForProductionServer` function.

    ```
    mpsxlResults = compiler.build.excelClientForProductionServer(buildResults, ...
    'Verbose','on');
    ```

    You can specify additional options in the `compiler.build` command by using name-value arguments. For details, see `compiler.build.excelClientForProductionServer`.

    The `compiler.build.Results` object `buildResults` contains information on the build type, generated files, included support packages, and build options.

    The function generates the following files within a folder named `mymagicexcelClientForProductionServer` in your current working directory:

    - `includedSupportPackages.txt` — Text file that lists all support files included in the assembly.

- `mymagic.bas` — VBA module file that can be imported into a VBA project.
- `mymagic.dll` — Dynamic library required by the Excel add-in.
- `mymagic.reg` — Text file that contains information on unresolved symbols.
- `mymagic.xla` — Excel add-in that can be installed directly in Excel.
- `mymagicClass.cs` — Text file that contains information on unresolved symbols.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Text file that contains packaging and deployment information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.

Note The generated Excel add-in does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Install the Deployable Archive with Excel Integration

You must deploy the archive to a MATLAB Production Server instance before you can use the add-in in Excel.

To install the deployable archive on a server instance:

1 Locate the archive in the `for_redistribution_files_only\server\` folder if you used the Production Server Compiler, or the `addmatrixproductionServerArchive` folder if you used the `compiler.build.productionServerArchive` function.

   For this example, the file name is `mymagic.ctf`.
2 Copy the archive file to the `auto_deploy` folder of the server instance. The server instance automatically deploys it and makes it available to interested clients.

For more information, see "MATLAB Production Server" documentation.

See Also
Production Server Compiler | `mcc`

Create a C# Client

This example shows how to write a C# application to call a MATLAB function deployed to MATLAB Production Server. The C# application uses the MATLAB Production Server .NET client library.

A .NET application programmer typically performs this task. The tutorial assumes that you have Microsoft Visual Studio and .NET installed on your computer.

Create Microsoft Visual Studio Project

1 Open Microsoft Visual Studio.

2 Click **File > New > Project**.

3 In the New Project dialog box, select the template you want to use. For example, if you want to create a C# console application in Visual Studio 2017, select **Visual C# > Windows Desktop** in the left navigation pane, then select the **Console App (.Net Framework)**.

4 Type the name of the project in the **Name** field (for example, `Magic`).

5 Click **OK**. Your `Magic` source shell is created, typically named `Program.cs`, by default.

Create Reference to Client Runtime Library

Create a reference in your `Magic` project 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), right-click your `Magic` project, select **Add > Browse**.

2 Browse to the MATLAB Production Server .NET client runtime library location.

The library is located in *matlabroot*`\toolbox\compiler_sdk\mps_clients\dotnet`. Select the `MathWorks.MATLAB.ProductionServer.Client.dll` file.

The client library is also available for download at `https://www.mathworks.com/products/matlab-production-server/client-libraries.html`.

3 Click **OK**. Your Microsoft Visual Studio project now references the `MathWorks.MATLAB.ProductionServer.Client.dll`.

Deploy MATLAB Function to Server

Write a MATLAB function `mymagic` that uses the `magic` function to create a magic square, package `mymagic` into a deployable archive called `mymagic_deployed`, then deploy it to a server. The function `mymagic` takes a single `int` input and returns a magic square as a 2-D `double` array. The example assumes that the server instance is running at `http://localhost:9910`.

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

Design .NET Interface in C#

Invoke the deployed MATLAB function `mymagic` from a .NET client through a .NET interface. Design a C# interface `Magic` to match the MATLAB function `mymagic`.

- The .NET interface has the same number of inputs and outputs as the MATLAB function.
- Since you are deploying one MATLAB function on the server, you define one corresponding .NET method in your C# code.

- Both the MATLAB function and the .NET interface process the same data types—input type `int` and output type 2-D `double`.

- In your C# client program, use the interface `Magic` to specify the type of the proxy object reference in the `CreateProxy` method. The `CreateProxy` method requires the URL to the deployable archive that contains the `mymagic` function (`http://localhost:9910/mymagic_deployed`) as an input argument.

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

Write, Build, and Run .NET Application

1 Open the Microsoft Visual Studio project `Magic` that you created earlier.

2 In the `Program.cs` tab, paste in the code below.

```
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();
            }
        }
    }
}
```

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`).

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
```

See Also

More About

- "Create a .NET MATLAB Production Server Client" (MATLAB Production Server)
- "Configure the Client-Server Connection" (MATLAB Production Server)
- "Synchronous RESTful Requests Using Protocol Buffers in .NET Client" (MATLAB Production Server)

Create MATLAB Production Server Java Client Using MWHttpClient Class

This example shows how to write a MATLAB Production Server client using the `MWHttpClient` class from the Java client API. For information on obtaining the Java client library, see "Obtain and Configure Client Library" (MATLAB Production Server). In your Java code, you will:

- Define a Java interface that represents the deployed MATLAB function.
- Instantiate a static 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, for example, `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.

For example, consider the following `addmatrix` function deployed to the server. For information on writing and compiling the function for deployment, see "Create Deployable Archive for MATLAB Production Server" (MATLAB Production Server). For deploying the function to the server, see "Deploy Archive to MATLAB Production Server" (MATLAB Production Server).

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

a = a1 + a2;
```

The interface for the `addmatrix` function follows.

```
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 "Data Conversion with Java and MATLAB Types" (MATLAB Production Server) and "Conversion of Java Types to MATLAB Types" (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 to initialize the variables used by the application:

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

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 `createProxy` method of the client object 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_clients 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 `close()` method of the client object 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 resulting matrix
        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 at the Windows command prompt:

```
javac -classpath "matlabroot\toolbox\compiler_sdk\mps_clients\java\mps_client.jar" MPSClientExample.java
```
14 Run the application using the `java` command or your IDE.

For example, enter the following at the Windows command prompt:

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

To run the application on Linux and macOS systems, use a colon (`:`) to separate multiple paths.

The application returns the following at the console:

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

See Also

More About

*   "Bond Pricing Tool for Java Client" (MATLAB Production Server)
*   "MATLAB Production Server Java Client Basics" (MATLAB Production Server)
*   "Synchronous RESTful Requests Using Protocol Buffers in the Java Client" (MATLAB Production Server)

- "Asynchronous RESTful Requests Using Protocol Buffers in the Java Client" (MATLAB Production Server)

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 "Package Deployable Archives with Production Server Compiler App" and deployed in "Deploy Archive to MATLAB Production Server" (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_clients/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_clients` 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_clients/c/include/mps/`.

To link your application, the linker needs access to the following files stored in *matlabroot*/`toolbox/compiler_sdk/mps_clients/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`	`$arch/lib/` `libmwmpsclient.dylib`
	`$arch/lib/` `libmwcpp11compat.so`	

19 Run the application.

To run your application, add the following files stored in *matlabroot*/toolbox/ compiler_sdk/mps_clients/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.dylib
	$arch/lib/ libmwcpp11compat.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` MATLAB function deployed to a server instance. For information on writing and compiling the function for deployment, see "Create Deployable Archive for MATLAB Production Server" (MATLAB Production Server). For deploying the function to the server, see "Deploy Archive to MATLAB Production Server" (MATLAB Production Server).

Before you write the client application, you must have the MATLAB Production Server Python client libraries installed on your system. For details, see "Install the MATLAB Production Server Python Client" (MATLAB Production Server).

1　Start the Python command line interpreter.
2　Enter the following import statements at the Python command prompt.

```
import matlab
from production_server import client
```
3　Open the connection to the MATLAB Production Server instance and initialize the client runtime.

```
client_obj = client.MWHttpClient("http://localhost:9910")
```
4　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]])
```
5　Call the deployed MATLAB function. To call the function, you must know the name of the deployed archive and the name of the function.

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

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

The output is:

```
matlab.double([[5.0,7.0,9.0],[9.0,7.0,5.0]])
```
6　Close the client connection.

```
client_obj.close()
```

See Also
`matlab.production_server.client.MWHttpClient`

Related Examples
• 　"Create Client Connection" (MATLAB Production Server)
• 　"Invoke Packaged MATLAB Functions" (MATLAB Production Server)