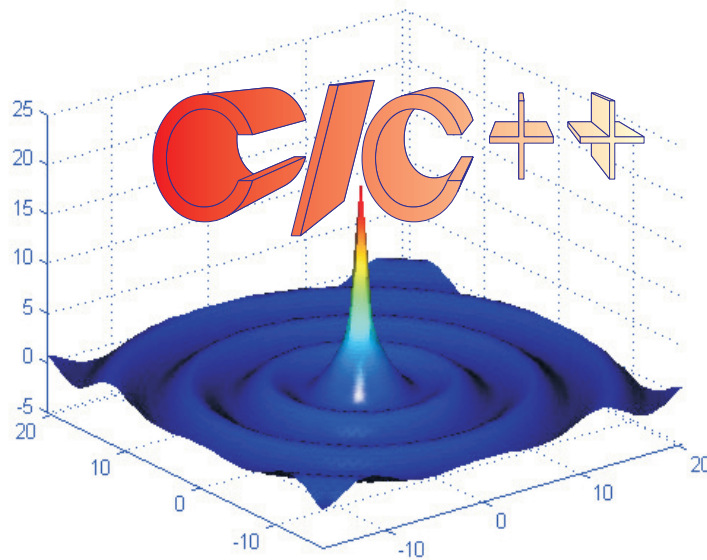


# MATLAB® C/C++ Book

Third Edition



This book is a great tutorial for C/C++ programmers who use MATLAB to develop applications and solutions

# **MATLAB<sup>®</sup> C/C++ Book**

**Copyright © 2004 by LePhan Publishing**

All rights reserved. No part of this CD-ROM should be reproduced, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher.

**ISBN 0-9725794-3-5**

## **Trademark**

MATLAB is a registered trademark of The MathWorks, Inc.  
Microsoft is a registered trademark of Microsoft Corporation.

## **Disclaimer**

The programs and applications on this CD-ROM have been carefully tested, but are not guaranteed for any particular purpose. The publisher does not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from use of such information.

# Contents

<b>Preface</b>	<b>v</b>
<b>Part I:</b>	
<b>Setting up MATLAB and C++ Compilers</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Introduction . . . . .	3
1.2 Projects and Analysis . . . . .	3
1.3 Computer Software and Book Features . . . . .	4
1.4 MATLAB C/C++ Types . . . . .	4
1.5 Differences between C/C++ and MATLAB C/C++ . . . . .	5
1.6 Reference Manuals . . . . .	5
<b>2 Opening a C++ File in Microsoft Visual C++ 6.0</b>	<b>7</b>
2.1 Opening a New C++ File . . . . .	7
2.2 Adding a Header File to a Project . . . . .	12
2.3 Writing a Code in a Header File . . . . .	13
2.4 Writing a Code in a C++ File . . . . .	15
2.5 Building and Executing a C++ Project . . . . .	16
<b>3 Setting Up a Microsoft Visual C++ 6.0 Project with MATLAB Compiler 4</b>	<b>17</b>
3.1 Procedure of Project Setting . . . . .	17
3.2 Testing of Project Setting . . . . .	20
<b>4 Opening a C++ File in Microsoft Visual C++ .Net</b>	<b>23</b>
4.1 Creating a New Microsoft Visual C++ .Net Project . . . . .	23
4.2 Adding a C++ Source File to the Project . . . . .	26
4.3 Adding a Header File to the Project . . . . .	29
4.4 Building and Executing the Project . . . . .	31

<b>5</b>	<b>Setting Up a Microsoft Visual C++ .Net Project with MATLAB Compiler 4</b>	<b>33</b>
5.1	Procedure of Project Setting . . . . .	33
5.2	Testing of Project Setting . . . . .	43
5.2.1	Writing a code for testing . . . . .	43
5.2.2	Building and executing the project . . . . .	44
<b>Part II:</b>		
<b>Creating and Using C/C++ Shared Libraries to Solve Mathematical Problems</b>		<b>45</b>
<b>6</b>	<b>Generating C and C++ Shared Libraries from MATLAB M-Files for Using in Microsoft Visual C++ .Net</b>	<b>47</b>
6.1	Generating a C Shared Library from a MATLAB M-File . . . . .	48
6.2	Writing a Code to Call Functions in a C Shared Library . . . . .	51
6.3	Generated Functions from MATLAB Compiler 4 . . . . .	53
6.4	Using Multiple C Shared Libraries . . . . .	54
6.5	Generating a C++ Shared Library From a MATLAB M-File . . . . .	57
6.6	Writing a Code to Call Functions in a C++ Shared Library . . . . .	57
6.7	Generated C++ Functions from MATLAB Compiler 4 . . . . .	59
<b>7</b>	<b>Transfer of Values between C/C++ double, mxArray, and mxArray</b>	<b>61</b>
7.1	Transfer of Values between C/C++ double and mxArray . . . . .	61
7.2	Transfer of Values from C/C++ double to mxArray . . . . .	66
7.3	Transfer of Values from mxArray to C/C++ double . . . . .	69
7.4	The Code of the Utility File mxUtilityCompilerVer4.h . . . . .	72
7.5	The Code of the Utility File mwUtilityCompilerVer4.h . . . . .	83
<b>8</b>	<b>Matrix Computations</b>	<b>91</b>
8.1	Matrix Addition . . . . .	94
8.2	Matrix Subtraction . . . . .	98
8.3	Matrix Multiplication . . . . .	99
8.4	Matrix Determinant . . . . .	102
8.5	Inverse Matrix . . . . .	104
8.6	Transpose Matrix . . . . .	106
8.7	Assigning Directly Values for a Matrix . . . . .	107
8.8	Assigning Values for a Matrix from a File . . . . .	111



<b>9</b>	<b>Linear System Equations</b>	<b>115</b>
9.1	Linear System Equations . . . . .	118
9.2	Sparse Linear System . . . . .	126
9.3	Tridiagonal System Equations . . . . .	131
9.4	Band Diagonal System Equations . . . . .	140
<b>10</b>	<b>Ordinary Differential Equations</b>	<b>151</b>
10.1	First Order ODE . . . . .	154
10.2	Second Order ODE . . . . .	165
10.2.1	Analysis of second order ODE . . . . .	166
10.2.2	Using a second order ODE function . . . . .	167
<b>11</b>	<b>Integration</b>	<b>173</b>
11.1	Single Integration . . . . .	175
11.2	Double-Integration . . . . .	179
<b>12</b>	<b>Curve Fitting and Interpolations</b>	<b>183</b>
12.1	Polynomial Curve Fitting . . . . .	187
12.2	One-Dimensional Polynomial Interpolation . . . . .	192
12.3	Two-Dimensional Polynomial Interpolation for Grid Points . . . . .	195
12.4	Two-Dimensional Polynomial Interpolation for Non-Grid Points . . . . .	210
<b>13</b>	<b>Roots of Equations</b>	<b>227</b>
13.1	Roots of Polynomials . . . . .	229
13.2	The Root of a Nonlinear-Equation . . . . .	234
<b>14</b>	<b>Fast Fourier Transform</b>	<b>237</b>
14.1	One-Dimensional Fast Fourier Transform . . . . .	239
14.2	Two-Dimensional Fast Fourier Transform . . . . .	247
<b>15</b>	<b>Eigenvalues and Eigenvectors</b>	<b>255</b>
15.1	Eigenvalues and Eigenvectors . . . . .	256
<b>16</b>	<b>Random Numbers</b>	<b>265</b>
16.1	Uniform Random Numbers . . . . .	266
16.1.1	Generating Uniform Random Numbers in Range [0,1] . . . . .	267
16.1.2	Generating Uniform Random Numbers in Range [a,b] . . . . .	271
16.1.3	Generating a Matrix of Uniform Random Numbers in Range [0,1] . . . . .	274
16.1.4	Generating a Matrix of Uniform Random Numbers in Range [a,b] . . . . .	277
16.2	Normal Random Numbers . . . . .	280

16.2.1	Generating Normal Random Numbers with mean=0 and variance=1 . . .	280
16.2.2	Generating Normal Random Numbers with mean=a and variance=b . . .	283
16.2.3	Generating a Matrix of Normal Random Numbers with mean=0 and variance=1 . . . . .	286
16.2.4	Generating a Matrix of Normal Random Numbers with mean=a and variance=b . . . . .	288

### Part III:

#### MATLAB Engine: Calling MATLAB Workspace in C/C++ Functions

#### MEX-File: Calling C Functions in MATLAB Workspace

#### Generating Stand Alone Applications from MATLAB M-Files 291

#### 17 Calling MATLAB Workspace in C/C++ Functions 295

17.1	Calling MATLAB Workspace with Input/Output as a Scalar . . . . .	295
17.2	Calling MATLAB Workspace with Input/Output as a Vector and a Matrix . . .	298
17.3	Generating a MATLAB Graphic from a C/C++ Function . . . . .	301

#### 18 MEX-Files, Calling a C Function in MATLAB Workspace 305

18.1	MEX-File with Input/Output as Scalars . . . . .	305
18.2	MEX-File with Input/Output as Vectors . . . . .	307
18.3	MEX-File with Input/Output as Matrixes . . . . .	308
18.4	MEX-Function Analysis . . . . .	313

#### 19 Stand-Alone Applications 315

19.1	Installing MATLAB Component Runtime to a target machine . . . . .	315
19.2	Stand-Alone Application for an Addition Operator . . . . .	317
19.3	Stand-Alone Application for Linear Equations . . . . .	319
19.4	Stand-Alone Application for Using Matlab Plots . . . . .	320
19.5	Stand-Alone Application for Calculating an Integration . . . . .	322

#### References 325

#### Index 327

# Preface

At the time this book is written, most students and engineers know the basics of MATLAB and C/C++ programming. Their projects are often supported by C/C++ and/or MATLAB, but not many of them know how to use the C/C++ programming with MATLAB support to handle their problems. MATLAB is one of most powerful mathematical softwares used to solve student, engineering, and scientific problems, and C/C++ programming is one of the most used programming languages in the world with numerous applications. Therefore, the combination of both tools, C/C++ and MATLAB, has the potential to become one of the best tools for solving technical problems.

MATLAB provides a toolbox MATLAB Compiler to handle the works between MATLAB and C/C++. This book implements the combination of C/C++ and MATLAB to solve the problems. The features of this book are designed to handle the following projects:

- Common mathematical libraries were created from MATLAB M-files to use in C/C++ functions.
- The MATLAB workspace is called to perform particular tasks in C/C++ functions.
- A C function is called into the MATLAB workspace by writing a MEX-function.
- Stand-alone applications were created to use in the target machine which doesn't have the MATLAB software.

The book contains all C/C++ programming codes in all chapters, that quickly help users solve their problems. This book tries to support C/C++ programmers, especially students and engineers who use C/C++ and MATLAB to develop applications and solutions for their projects and designs.

LePhan Publishing

October 2004



**Part I:**  
**Setting up MATLAB**  
**and C<sup>++</sup> Compilers**



# Chapter 1

## Introduction

### 1.1 Introduction

MATLAB is a special mathematical software that includes many toolboxes. MATLAB Compiler is the most important toolbox that supports C/C++ programmers. We can use MATLAB Compiler 4.0 to create C/C++ functions from MATLAB M-files. These generated C/C++ functions will then be called by another C/C++ functions in a C/C++ file. In addition, we can use a C/C++ function to call the MATLAB workspace to perform specific tasks by using MATLAB Engine, and we can write a MEX-file for a C function to call it in the MATLAB workspace. MATLAB Compiler also provides a feature that can create stand-alone applications using in the target machine which doesn't have the MATLAB software.

There are some C/C++ compilers working with MATLAB Compiler 4.0. In this book, the compilers which were used to compile with MATLAB Compiler 4.0 are Microsoft Visual C++ 6.0, Microsoft Visual C++.Net 2002 (ver. 7.0), and Microsoft Visual C++.Net 2003 (ver. 7.1).

### 1.2 Projects and Analysis

The features of this book are designed to handle following projects:

1. Common mathematical libraries were created from MATLAB M-files to use in C/C++ functions. This is the popular purpose of using MATLAB Compiler. In this project, C/C++ functions will call mathematical functions in a generated mathematical library to solve the mathematical problems. These solutions are explained from Chapter 8 to Chapter 16. In these chapters, the used functions are chosen based on the typical problems, therefore the user can choose another options or another functions to solve his/her particular problems to satisfy requirements.
2. Working on C/C++ programming, we wants to call the MATLAB workspace to perform

particular tasks in the MATLAB workspace then transfer back results to a C/C++ function. This solution is explained in Chapter 17.

3. From an existing C function, we want to call this function in MATLAB by writing a MEX-function. In this project, we will write a MEX-function for an existing C function, then call this function into the MATLAB workspace. This solution is explained in Chapter 18.
4. From existing M-files, we want to generate stand-alone applications. In this project, MATLAB Compiler 4 will be used to generate stand-alone applications from existing MATLAB M-files. These generated stand-alone applications then will be used in a target machine which doesn't have the MATLAB software. This solution is explained in Chapter 19.

### 1.3 Computer Software and Book Features

MATLAB Compiler had some versions and there are some changes of its features in different versions. The focus of this book is only on MATLAB Compiler 4. The example codes in this book are developed, compiled, and tested in Windows 2000, Microsoft Visual C++ 6.0, Microsoft Visual C++ .Net (2002 and 2003), MATLAB 7, and MATLAB Compiler 4.0. These examples are intended to establish common works for C/C++ programming and MATLAB. The example codes are working on scalars, vectors, and matrixes that are inputs/outputs of functions for every application. In addition, the example codes are portable and presented in the step-by-step method, therefore the user can easily reuse the codes or write his/her own codes by following the step-by-step procedure while solving the problems.

The most C/C++ common functions in the examples are `void` functions (return type is `void`) to avoid ambiguity and to emphasize the topic being explained. The book also includes the settings of C++ compilers with MATLAB, and contains utility files to transfer values in different types. These files are very helpful in using MATLAB for C/C++ programming.

### 1.4 MATLAB C/C++ Types

MATLAB Compiler 4 has two principle types, `mwArray` and `mxArray`. When coding, you can use `mwArray` and `mxArray` in input/output as the new types in C/C++ functions, or make transfers of values between C/C++ `double`, `mwArray`, and `mxArray`. Chapter 7 shows transfers between C/C++ `double`, `mwArray`, and `mxArray`. These transfers are very useful in working on the MATLAB Compiler toolbox.



## 1.5 Differences between C/C++ and MATLAB C/C++

There are many differences between C/C++ and MATLAB C/C++ [4]. The most important difference to know is:

- C/C++ stores its two-dimensional arrays in the row-major order, whereas MATLAB C/C++ stores arrays in the column-major order. You must, therefore, remember this when setting up matrix data.

## 1.6 Reference Manuals

In working with MATLAB Compiler 4 you may need more information to help your task. We refer here several manuals from the MATLAB website that you can download for more information.

[http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/compiler/Compiler4.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/compiler/Compiler4.pdf)

[http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/apiext.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiext.pdf)

[http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/apiref.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiref.pdf)

[http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/refbook.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/refbook.pdf)

[http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/refbook2.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/refbook2.pdf)

[http://www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/refbook3.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/refbook3.pdf)

If you couldn't find these files at the time you are looking for, The MathWorks Inc. may change URL of these files, but you can find its somewhere in The MathWorks website [www.mathworks.com](http://www.mathworks.com).



## Chapter 2

# Opening a C++ File in Microsoft Visual C++ 6.0

In order to help users who have not used Microsoft Visual C++ version 6.0 (MSVC), this chapter contains a tutorial for opening and compiling a C++ file in MSVC. If you are familiar with MSVC you can skip this chapter.

### 2.1 Opening a New C++ File

To open Microsoft Visual C++ version 6.0, click Start, click Programs, click Microsoft Visual C++ 6.0, and click Microsoft Visual C++ 6.0. You will obtain Fig. 2.1.

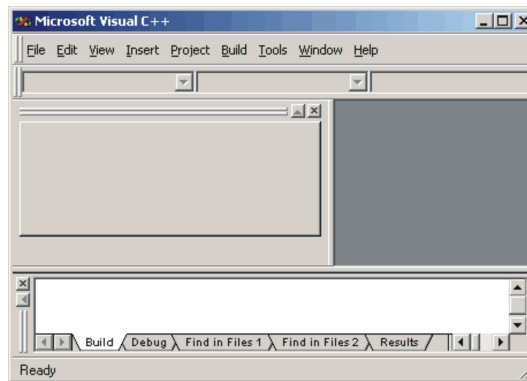


Figure 2.1: Opening Microsoft Visual C++ version 6.0.

In Fig. 2.1, on menu bar, click File, New (obtain Fig. 2.2).

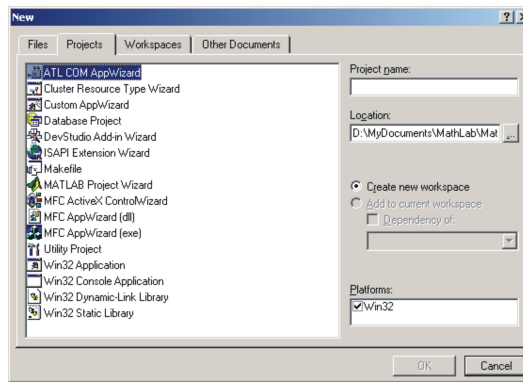


Figure 2.2: Opening a new file

In Fig. 2.2, click Files tab, click C++ Source File (obtain Fig. 2.3).

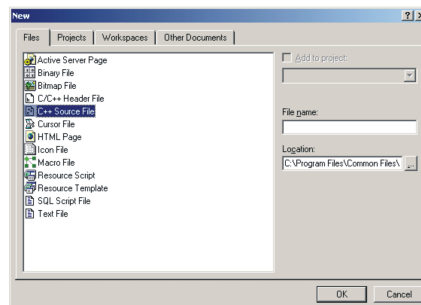


Figure 2.3: Opening a C++ source file

In Fig. 2.3, click on the browse button to choose a folder, then type a file name, say `Myfile.cpp` (see Fig. 2.4), click OK.

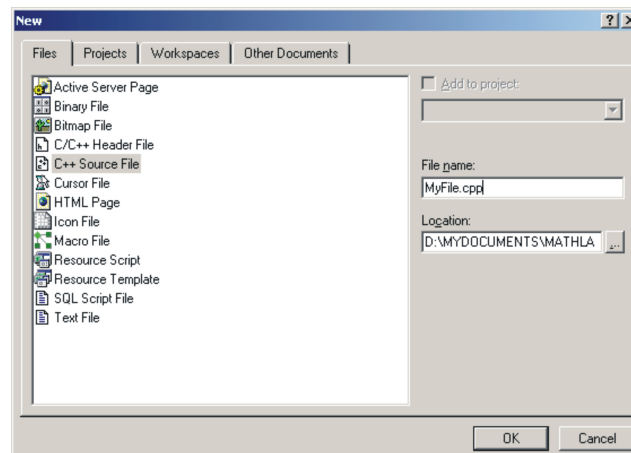


Figure 2.4: Creating a new C++ file

You now have the blank file Myfile.cpp. Write the following code into this file (Fig. 2.5) as follows:

Listing code

---

```
#include <iostream.h>

void main() {

cout << "Hello World" << endl ;
}
```

---

end code

---

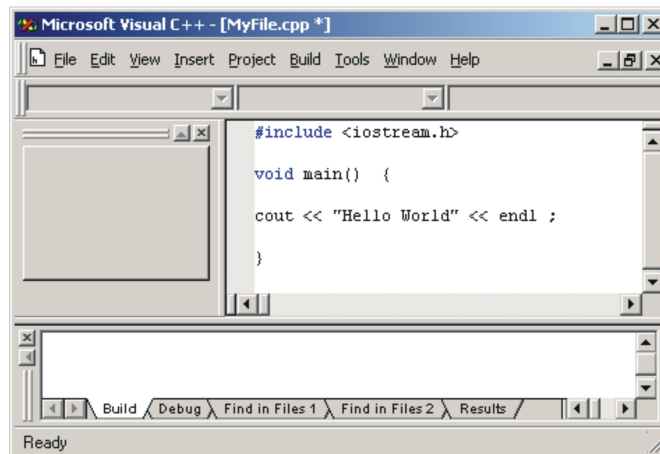


Figure 2.5: A simple code in the C++ file

On menu bar (Fig. 2.5), click Build, then click Build (obtain Fig. 2.6).

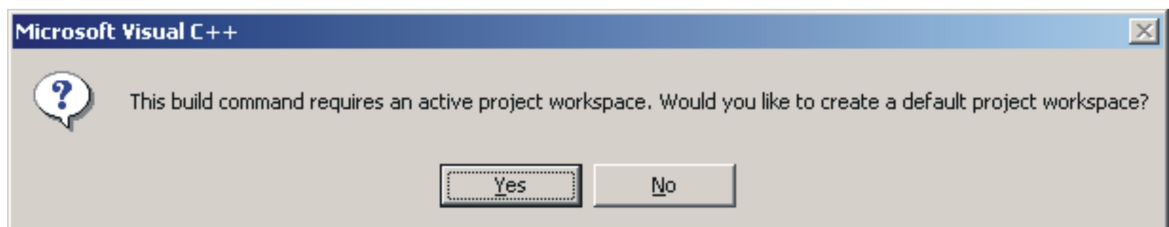


Figure 2.6: Building and executing the file

In Fig. 2.6, click Yes (to create a project), you will obtain Fig. 2.7.

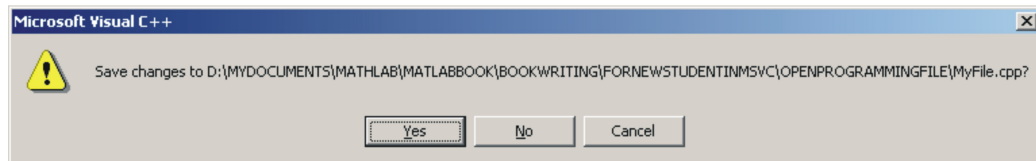


Figure 2.7: Creating an MSVC project

In Fig. 2.7, click Yes. You will obtain a new project named **MyFile** (Fig. 2.8).

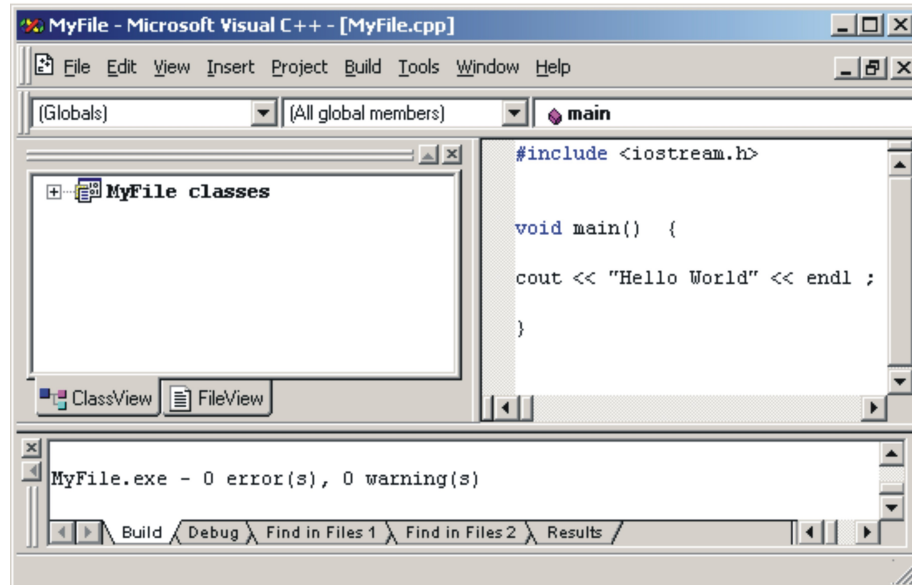


Figure 2.8: Creating an MSVC project (continued)

On the menu bar (Fig. 2.8), click Build, click Rebuild All.

On the menu bar (Fig. 2.8), click Build, click Execute MyFile.exe (obtain an output result, see Fig. 2.9).

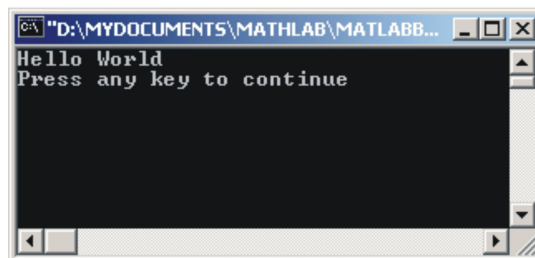


Figure 2.9: A programming output

To copy the result, drag words to highlight, click on left-top corner button (C:\), click Edit, click Copy (Fig. 2.10).

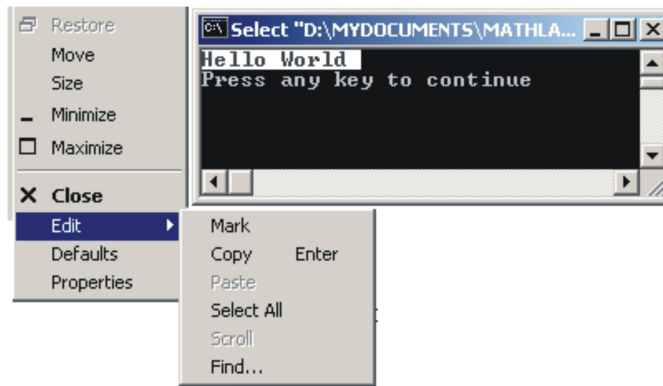


Figure 2.10: Copying an output

To debug an error, on the menu bar (Fig. 2.8) click Tool, click Customize (obtain the customize dialog as shown in Fig. 2.11). In the customize dialog (Fig. 2.11), click Commands tab, Edit (in

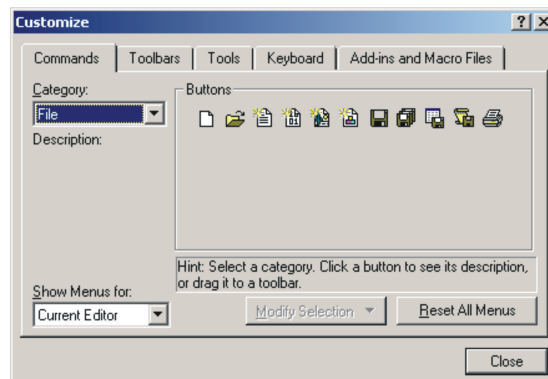


Figure 2.11: the Customize dialog

Category), drag and drop two hammer-icons into the icon bar (Fig. 2.12).

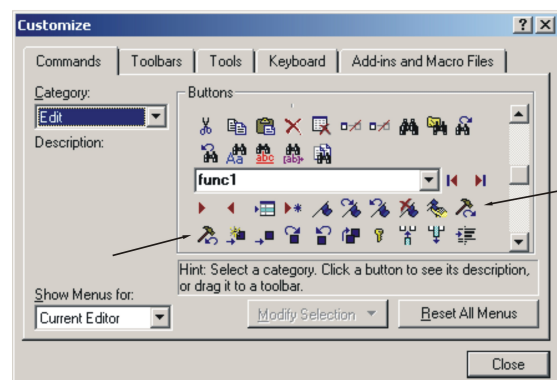


Figure 2.12: The Customize dialog (continued)

If you have errors in your file when building Myfile.exe file, click on these hammer-icons, these will reveal your errors. Figure 2.13 shows an example of the error with missing a semi colon at the end of the line.

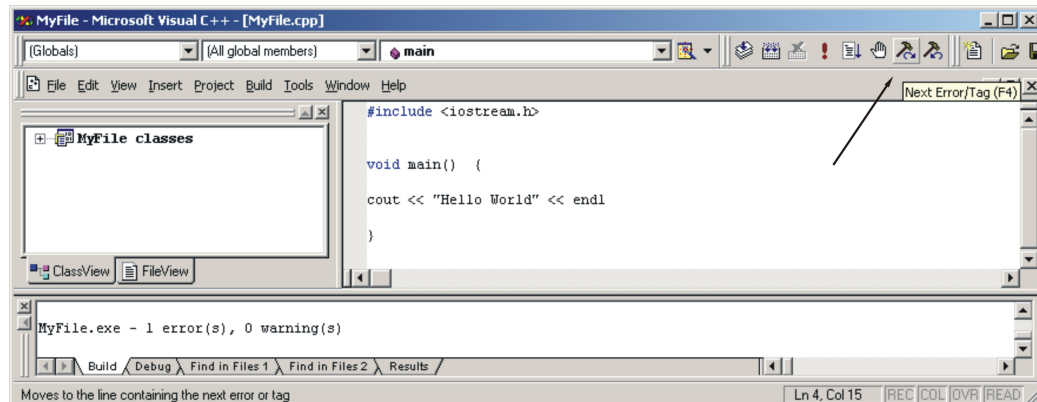


Figure 2.13: Debugging errors

There are many helpful features in MSVC that you can find by clicking on **Help** in the menu bar. Since the purpose of this chapter is to help users who are new to MSVC, a number of the most basic features will be presented in the next sections.

## 2.2 Adding a Header File to a Project

To add a header file into the project, in the project workspace (Fig. 2.14), click **FileView** tab, right click on **MyFile files**, click "Add Files to Project .." (obtain Fig. 2.15), then type Test.h in this dialog (see Fig. 2.15).

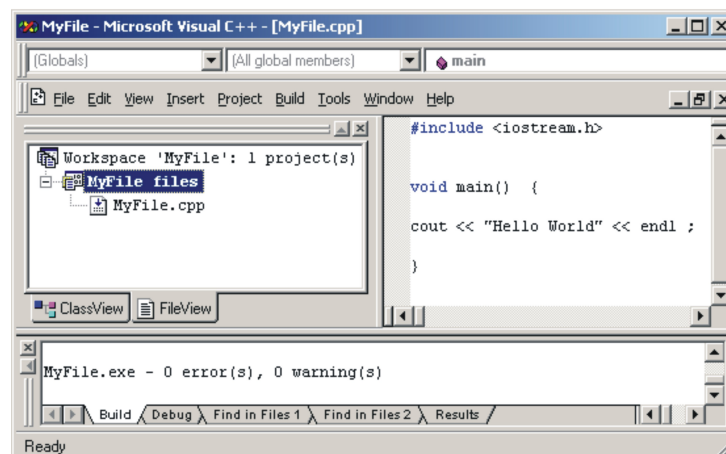


Figure 2.14: Adding a header file



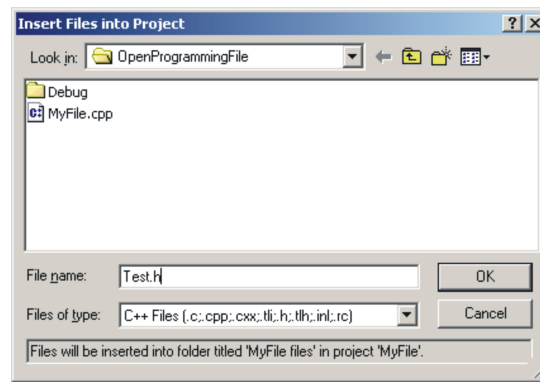


Figure 2.15: Adding a header file (continued)

In Fig. 2.15, click OK (obtain Fig. 2.16). In Fig. 2.16, click Yes (to generate the new header file named Test.h into the project. See Fig. 2.17).

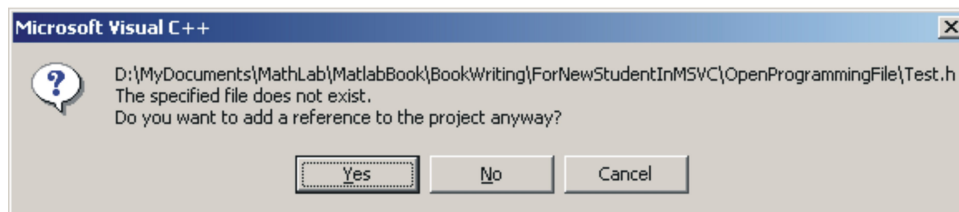


Figure 2.16: Adding a header file (continued)

## 2.3 Writing a Code in a Header File

This section describes how to write a simple code in the header file.

In the project workspace (Fig. 2.17), double-click on Test.h (obtain Fig. 2.18).

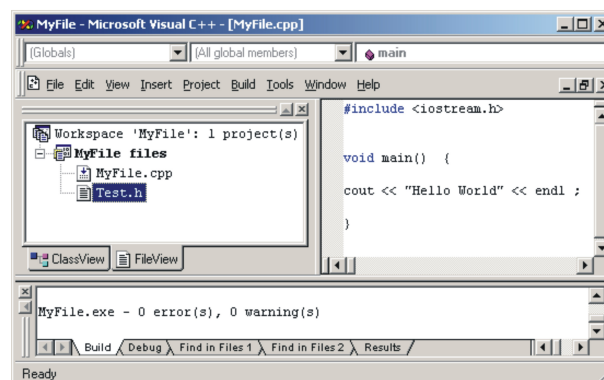


Figure 2.17: Writing a code in the header file

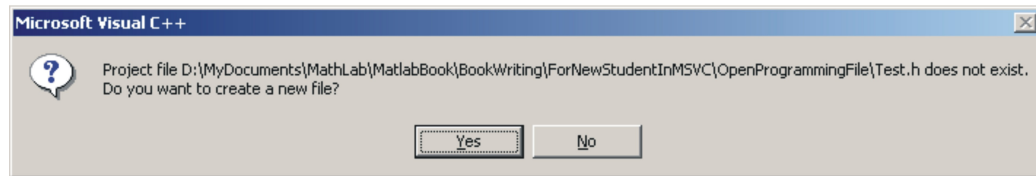


Figure 2.18: Writing a code in the header file (continued)

In Fig. 2.18, click Yes (obtain a blank header file, see Fig. 2.19).

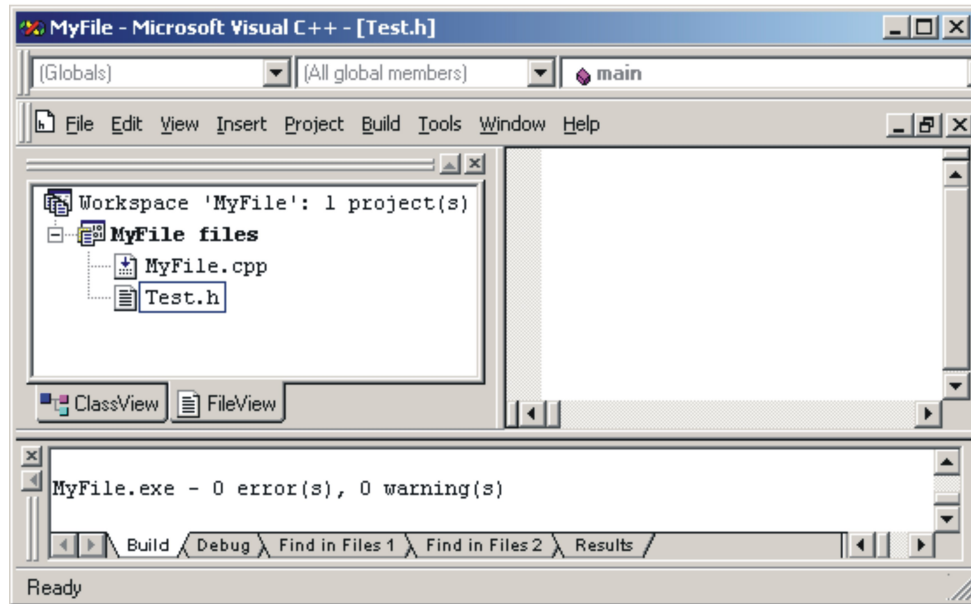


Figure 2.19: Writing a code in the header file (continued)

Now you will write a simple code in this header file Test.h (see Fig 2.20), as follows:

Listing code

---

```
class Test {

public:
void TestFunc () ;

    Test () { ; }
    ~Test () { ; }

} ;
```

```
/* ***** */
```

```
void Test::TestFunc() {
```

```
    cout << "This is a Test Function" << endl ;
}
```

---

end code

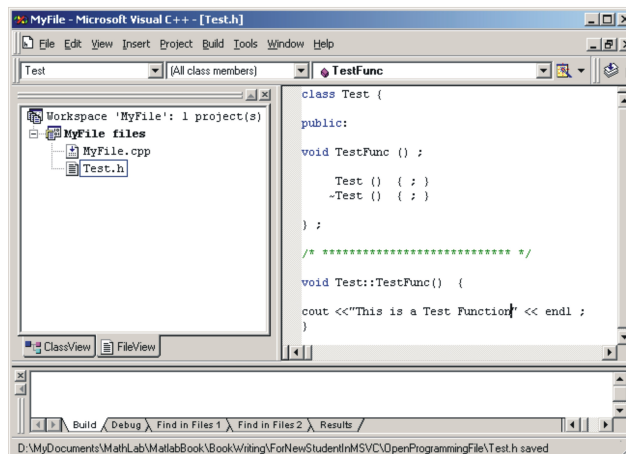


Figure 2.20: Writing a code in a header file (continued)

## 2.4 Writing a Code in a C++ File

Modify the lines of the code in the C++ file MyFile.cpp (Fig. 2.21), as follows:

```
#include <iostream.h>
#include "Test.h"
```

```
void main() {
```

```
    cout << "Hello World" << endl ;
```

```
    Test obj ;
    obj.TestFunc() ;
```

```
}
```

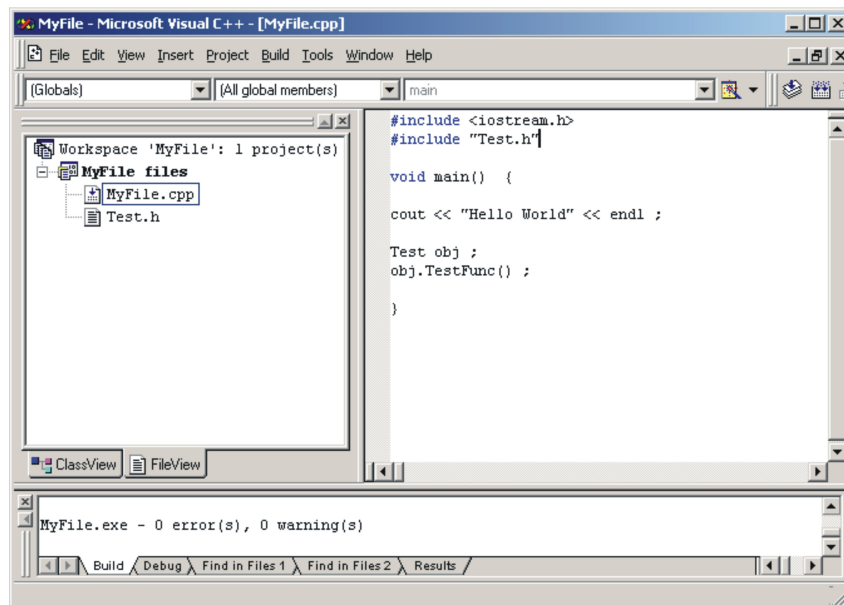


Figure 2.21: Modifying the code in the main function

## 2.5 Building and Executing a C++ Project

On the menu bar (Fig. 2.21), click Build, click Rebuild All.

On the menu bar (Fig. 2.21), click Build, click Execute MyFile.exe.

It should contain no errors and should give the output result (Fig. 2.22).

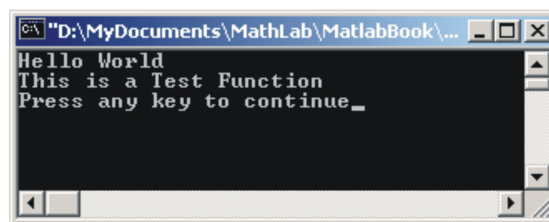


Figure 2.22: The output result

At this point you can use Microsoft Visual C++ 6.0 to open and compile a C++ file.

## Chapter 3

# Setting Up a Microsoft Visual C++ 6.0 Project with MATLAB Compiler 4

This chapter describes how to set up a Microsoft Visual C++ version 6.0 (MSVC) with MATLAB Compiler 4.

### 3.1 Procedure of Project Setting

The following procedure is to set up a Microsoft Visual C++ 6.0 (MSVC) project for working with MATLAB Compiler 4.

1. From an MSVC project (as described in Chapter 2), click Project from the menu bar, click Setting. You will obtain the dialog box as shown in Fig. 3.1.

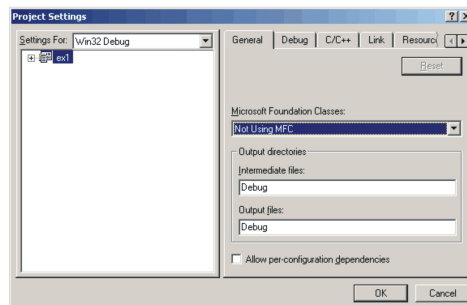


Figure 3.1: The Project Settings dialog

2. In Fig. 3.1, click the **C/C++** tab (see Fig. 3.2).

From **Category** shown in Fig. 3.2, choose Code Generation.

In **Use run-time Library**, choose Multithreaded DLL (Fig. 3.2).

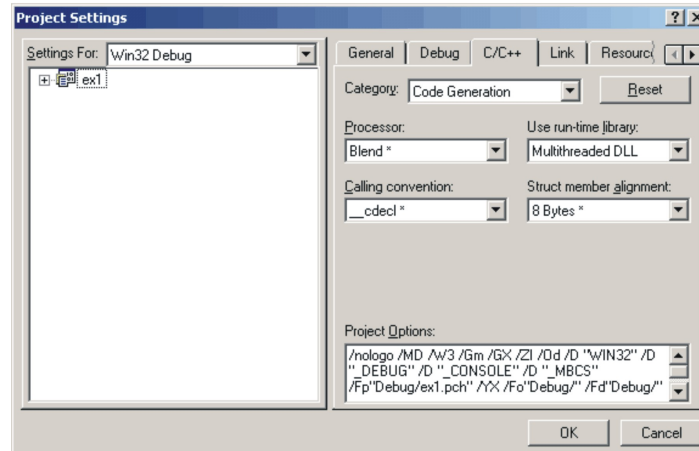


Figure 3.2: The C/C++ tab in Project Settings

3. In **Category**, choose Preprocessor (see Fig. 3.3).

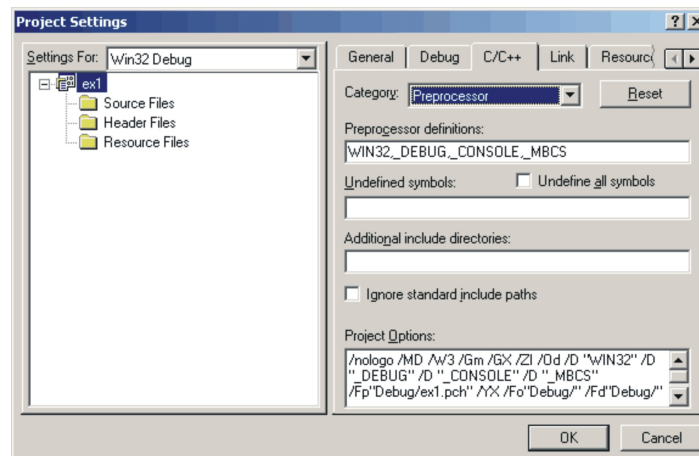


Figure 3.3: Preprocessor in Project Settings

4. In **Preprocessor definitions** text box (Fig. 3.3), replace all with (see Fig. 3.4):

WIN32,\_DEBUG,\_CONSOLE,\_MBCS,\_WINDOWS,\_AFXDLL,IBMP,MSVC,MSWIND, \_\_STDC\_\_

5. Suppose that your MATLAB path is C:\MATLAB7.

In **Additional include directories**, add (see Fig. 3.4):

C:\MATLAB7\extern\include

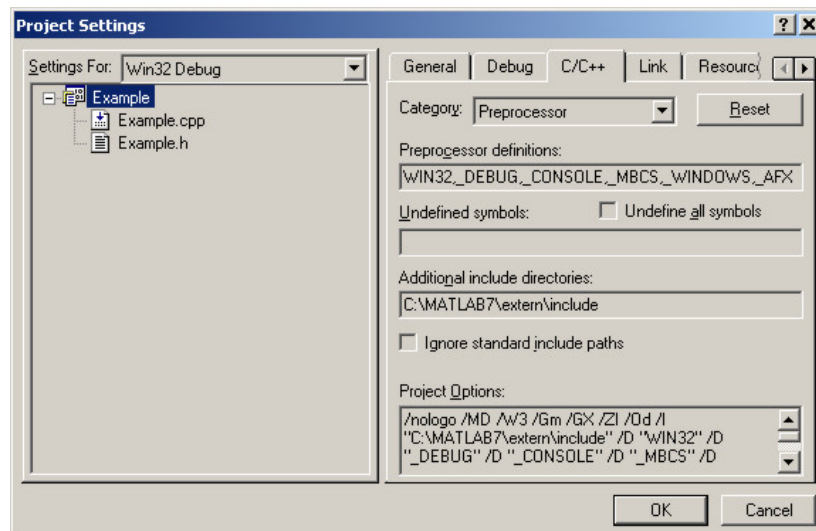


Figure 3.4: Project Settings (continued)

6. Click the **Link** tab (see Fig. 3.5).

In **Category** (Fig. 3.5), choose Input.

In **Object/library modules**, add these libraries (Fig. 3.6):

```
libeng.lib libfixedpoint.lib libmat.lib libmex.lib libmwservices.lib libmx.lib
libut.lib mclcom.lib mclcommmain.lib mclmcr.lib mclmcrrt.lib mclxlmain.lib
```

Note that they are separated by a space.

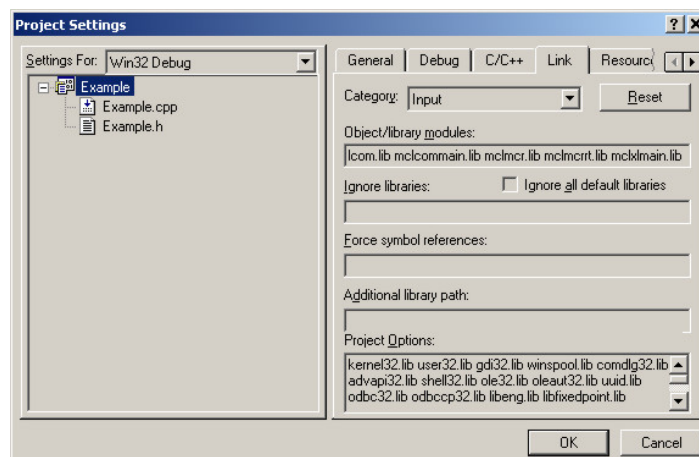


Figure 3.5: The Link tab in Project Settings

7. Suppose that your MATLAB path is C:\MATLAB7.

In **Additional library path**, add (see Fig. 3.6):

```
C:\MATLAB7\extern\lib\win32\microsoft\msvc60
```

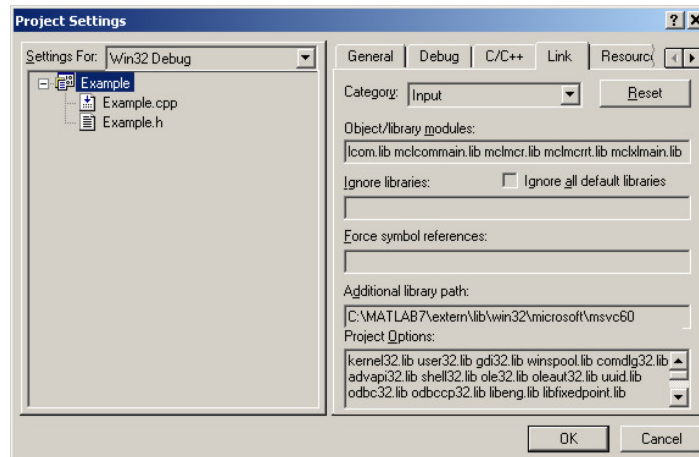


Figure 3.6: Project Settings (continued)

8. In Fig. 3.6, click **OK** to finish the project setting.

## 3.2 Testing of Project Setting

To test the project setting, create an MSVC project then perform the project setting as described in the above procedure. Write the simple following code in a C++ file, then build and execute the project.

The following is the code for testing the project setting.

After building and executing the project, you should have no errors.

Listing code

---

```
#include <iostream.h>
#include "mclcppclass.h"
int main() {

    cout << " Testing setting-up MSVC 6.0 " << endl ;
    mwArray mw_test(1, 2, mxDOUBLE_CLASS) ;

    mw_test(1, 1) = 1.1 ;
    mw_test(1, 2) = 2.2 ;

    std::cout << mw_test << std::endl ;

    return 0 ;
}
```



## Remarks

1. You can use the Microsoft Visual C++ 6.0 (MSVC) project set up in this chapter as a template for another MSVC projects in working with MATLAB Compiler 4.
2. Most of our example projects are Microsoft Visual C++ .Net projects, therefore for working with Microsoft Visual C++ 6.0, do these following steps:
  - Copy the project that was set up as described in this chapter. This project is a current your project.
  - Copy the files Example.cpp, Example.h, and the utility file (mxUtilityCompilerVer4.h or mwUtilityCompilerVer4.h) to your current project.
  - Remove the top line :

```
#pragma warning(disable : 4995)
```

, then you have a project working in Microsoft Visual C++ 6.0.



## Chapter 4

# Opening a C++ File in Microsoft Visual C++ .Net

In order to help students who have not used Microsoft Visual C++ .Net (MSVC.Net), this chapter contains a tutorial for opening and compiling a C++ file in MSVC++.Net. If you are familiar with MSVC++.Net you can skip this chapter.

### 4.1 Opening a New Microsoft Visual C++ .Net Project

To open Microsoft Visual C++ .Net, click Start, Programs, Microsoft Visual Studio .Net, Microsoft Visual Studio .Net. You will obtain Fig. 4.1.

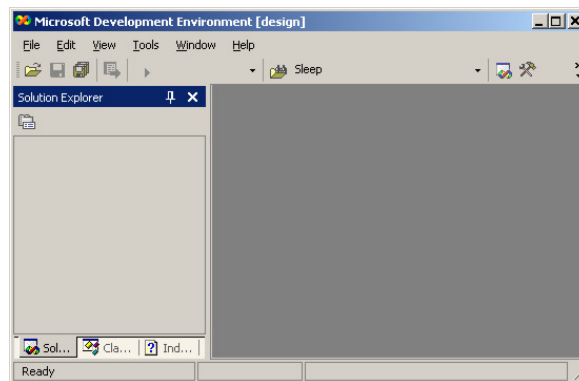


Figure 4.1: Opening Microsoft Visual C++ .Net

In Fig. 4.1, on menu bar, click File, New, Project (obtain Fig. 4.2).

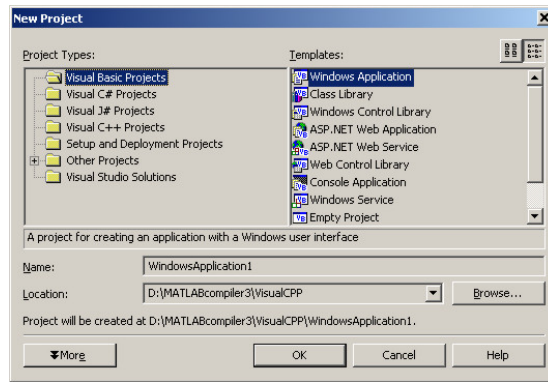


Figure 4.2: Opening a new C++ file in Microsoft Visual C++ .Net

In Fig. 4.2, click Visual C++ Projects, click Win32 Projects.

Click the Browse button to choose a folder, then type the project name, say **Example**, as in Fig. 4.3. In Fig. 4.3 click OK . You will obtain Fig. 4.4

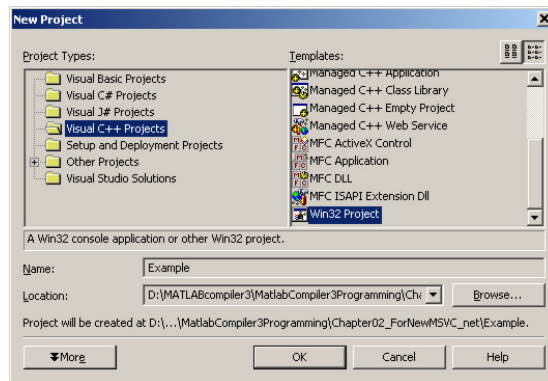


Figure 4.3: Opening a new C++ file (continued)

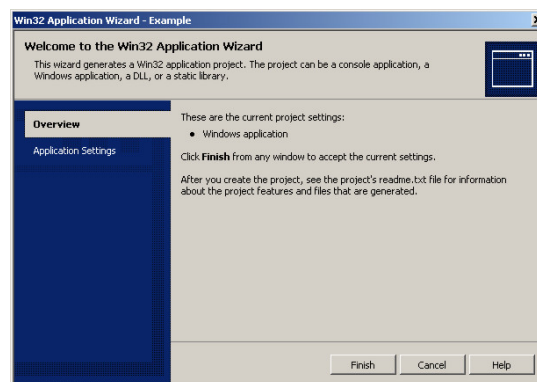


Figure 4.4: Opening a new C++ file (continued)

In Fig. 4.4, click Application Settings, select Console application. Select Empty project (see Fig. 4.5). Click Finish to create the project. You'll obtain Fig. 4.6.

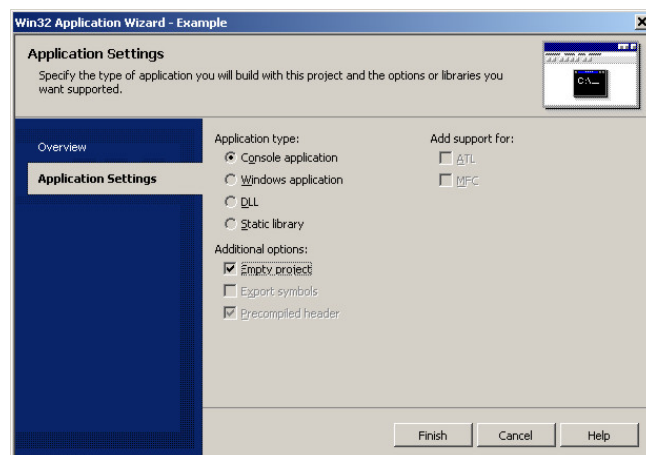


Figure 4.5: Opening a new C++ file (continued)

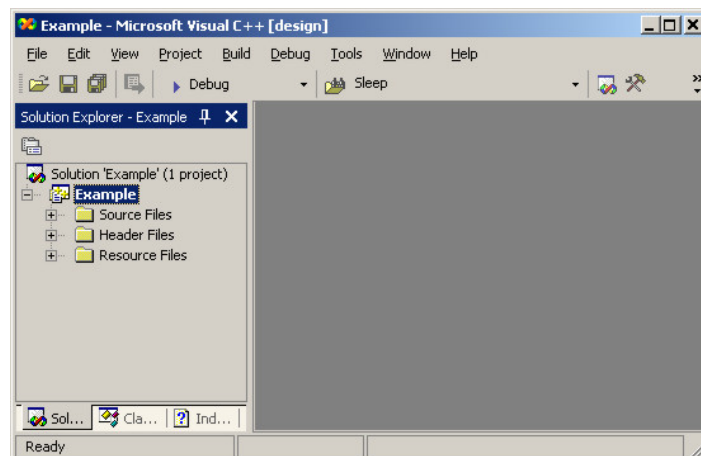


Figure 4.6: Opening a new C++ file (continued)

## 4.2 Adding a C++ Source File to the Project

You now have the blank project and you will add a C++ file to this project.

In Fig. 4.7, right click **Example**, click Add, click Add New Item to open a dialog box as in Fig. 4.18

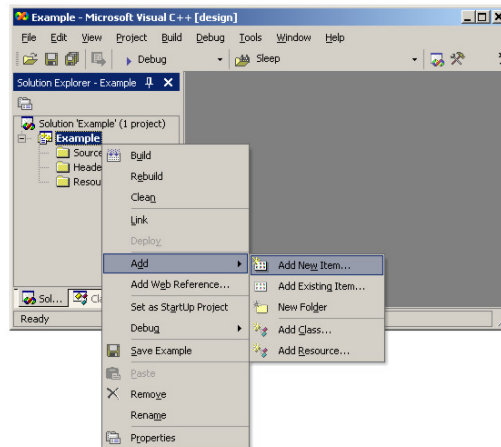


Figure 4.7: Adding a C++ source file

In Fig. 4.8 then type the name of file Example.cpp. Click C++ File from the Templates list on the right, as Fig. 4.8.

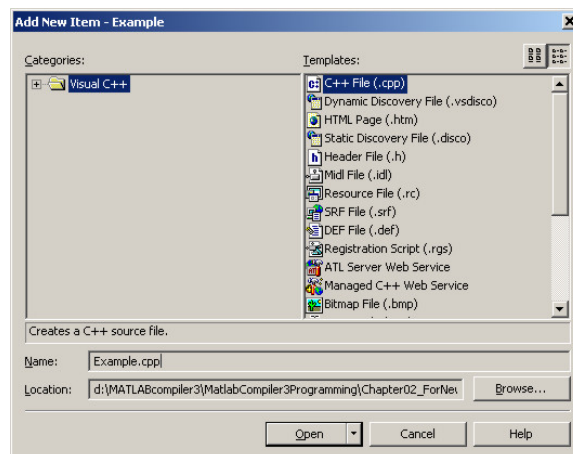


Figure 4.8: Adding a C++ source file (continued)

In Fig. 4.8 click Open, you will obtain a blank file Example.cpp (Fig. 4.9).

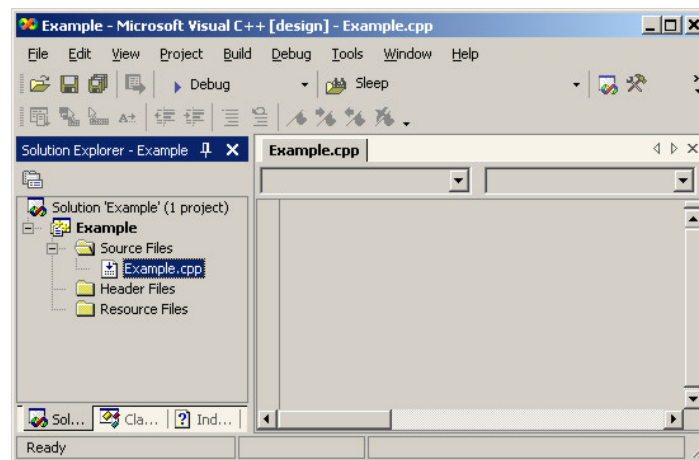


Figure 4.9: Adding a C++ source file (continued)

Write the following code into this file Example.cpp, (Fig. 4.10) as follows:

Listing code

---

```
#include <iostream.h>

int main() {
    cout << "Hello World!" << endl ;

    return 0 ;
}
```

---

end code

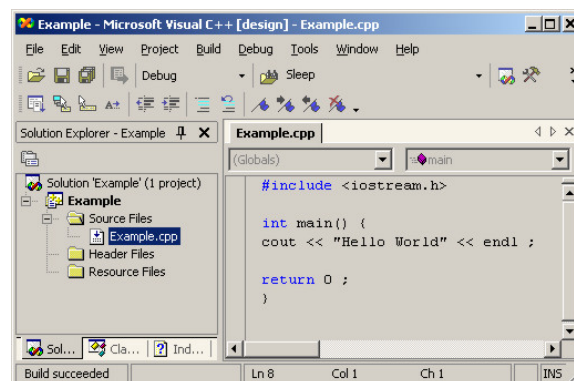


Figure 4.10: Adding a C++ source file (continued)

To build the executive file, click Build on the menu, click Build Solution.

You will have no error, but one warning as follows:

```
: warning C4995: '_OLD_IOSTREAMS_ARE_DEPRECATED':  
name was marked as #pragma deprecated"
```

To ignore this warning, add this line code (Fig. 4.11):

```
#pragma warning(disable : 4995)
```

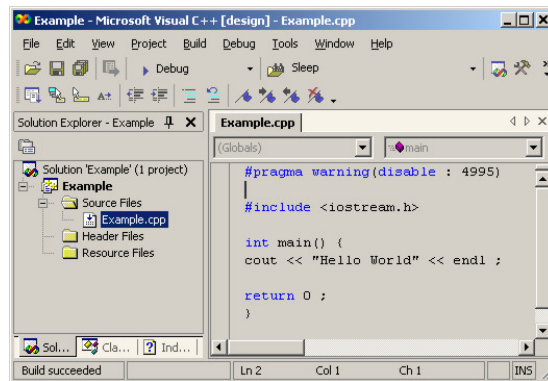


Figure 4.11: Adding a C++ source file (continued)

To execute again, click Build on the menu, click Build Solution. You will have no errors and no warning.

To see the result, on the menu bar click Debug, click Start Without Debugging (Fig. 4.13).

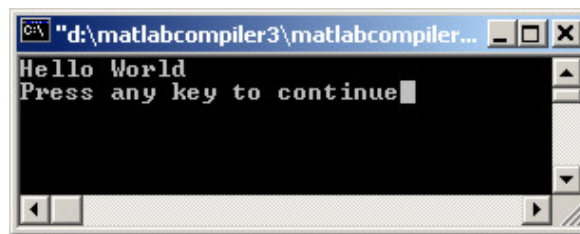


Figure 4.12: The output result

#### Note:

You can avoid the above warning by using Standard Library, but this will give more difficulties in working with MATLAB Compiler, therefore within the scope of working with MATLAB Compiler we don't recommend using Standard Library in MSVC++.Net.

For more information of this warning, go to the Microsoft website:

<http://msdn.microsoft.com/library/>



### 4.3 Adding a Header File to the Project

To add a header file to the project, right click **Example**, click Add, click Add New Item to open a dialog box (see Fig. 4.7).

Type the name for the header file `Example.h`, click Header File from the Templates list on the right as in Fig. 4.13.

Click Open, you will obtain the blank header file `Example.h` (Fig. 4.14).

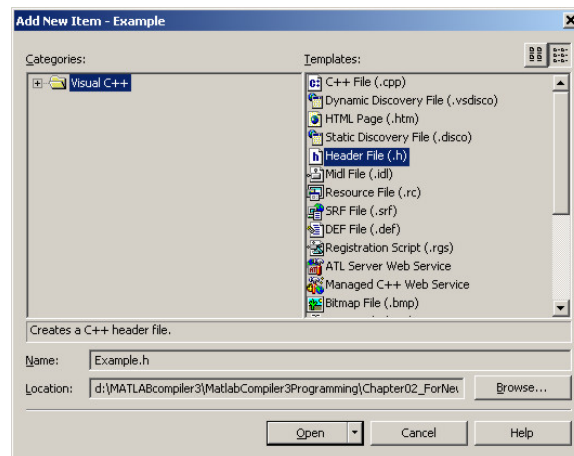


Figure 4.13: Adding a header file

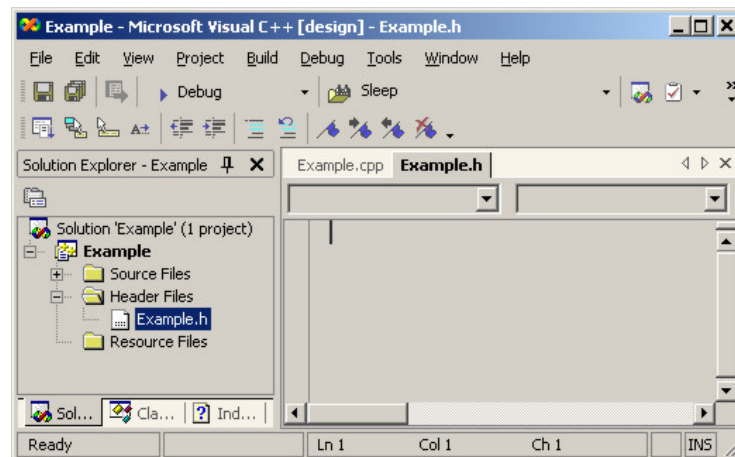


Figure 4.14: Adding a header file (continued)

Now you will write a simple code in this header file Example.h (see Fig 4.15), as follows:

Listing code

---

```
class Test {

public:
void TestFunc () ;

    Test () { ; }
    ~Test () { ; }

} ;

/* ***** */

void Test::TestFunc() {

cout <<"This is a Test Function" << endl ;

}
```

---

end code

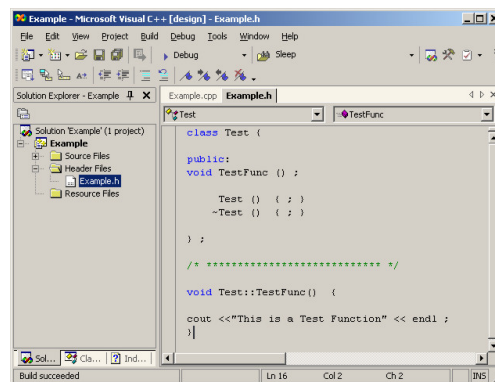


Figure 4.15: Writing a code in the header file (continue)

Modify lines of the code in the C++ file Example.cpp (Fig. 4.16), as follows:

Listing code

---

```
#pragma warning(disable : 4995)

#include <iostream.h>
#include "Example.h"
```

```

int main() {
    cout << "Hello World" << endl ;

    Test obj ;
    obj.TestFunc() ;

    return 0 ;
}

```

---

end code

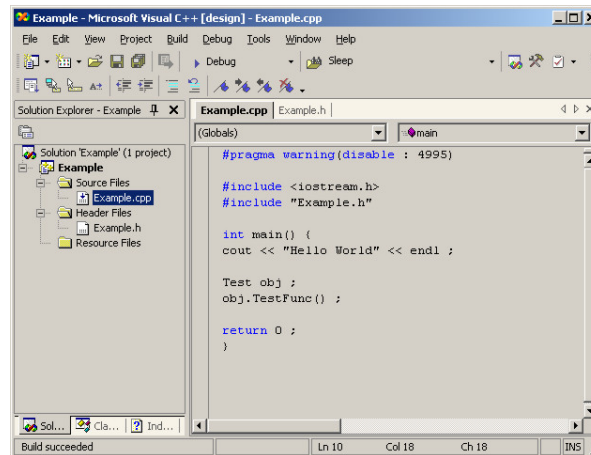


Figure 4.16: Modifying the code in the C++ file

## 4.4 Building and Executing the Project

In Fig. 4.16, on the menu bar click Build, click Build Solution.

In Fig. 4.16, on the menu bar click Debug, click Start Without Debugging.

It should contain no errors and give the output result as shown in Fig. 4.17

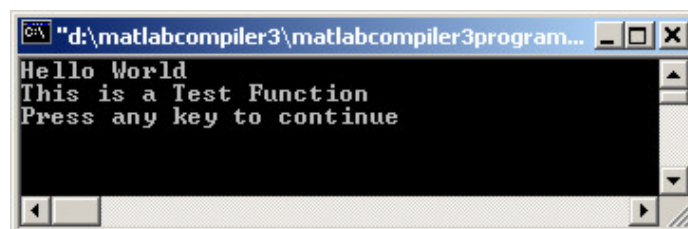


Figure 4.17: The output result

At this point you can use Microsoft Visual C++ .Net to open and compile a C++ file.

**Note**

If you have errors in your file when building, the error message will show in the Task List as in Fig. 4.18. Double click on this message, it will show the error in the code.

There are many helpful features in MSVC++.Net that you can find in Help menu.

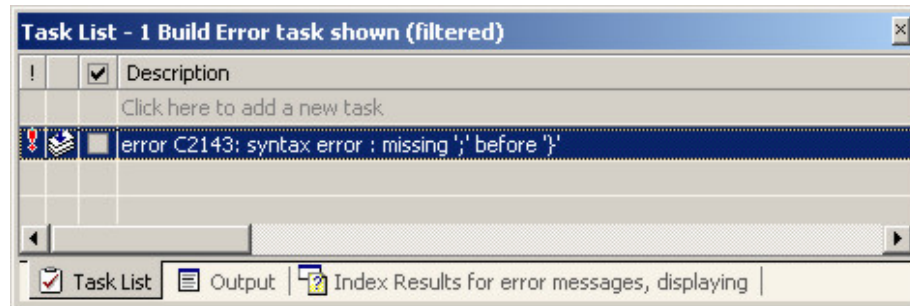


Figure 4.18: An example of an error message

## Chapter 5

# Setting Up a Microsoft Visual C++ .Net Project with MATLAB Compiler 4

This chapter describes how to set up a Microsoft Visual C++ .Net (MSVC++.Net) project with MATLAB Compiler 4.

### 5.1 Procedure of Project Setting

The following procedure is to set up an MSVC++.Net project for working with MATLAB Compiler 4.

1. Create an MSVC++.Net project as described in Chapter 4. This project is shown in Fig. 5.1.

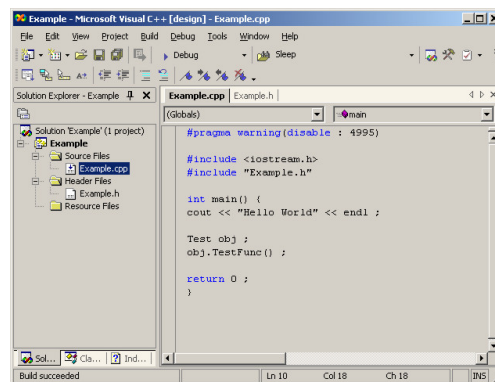


Figure 5.1: A Microsoft Visual C++ .Net project

2. In Fig. 5.1, right click on **Example**, click Properties (Fig. 5.2), you'll obtain a property dialog as shown in Fig. 5.3.

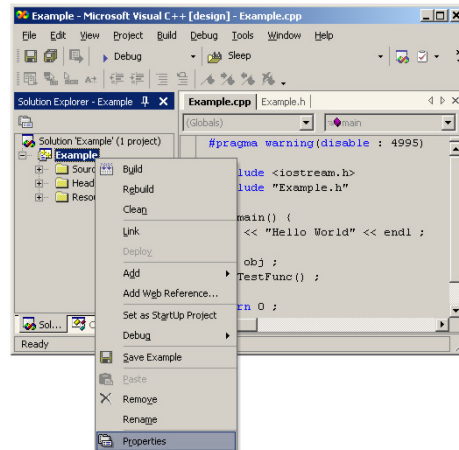


Figure 5.2: Opening Property Pages

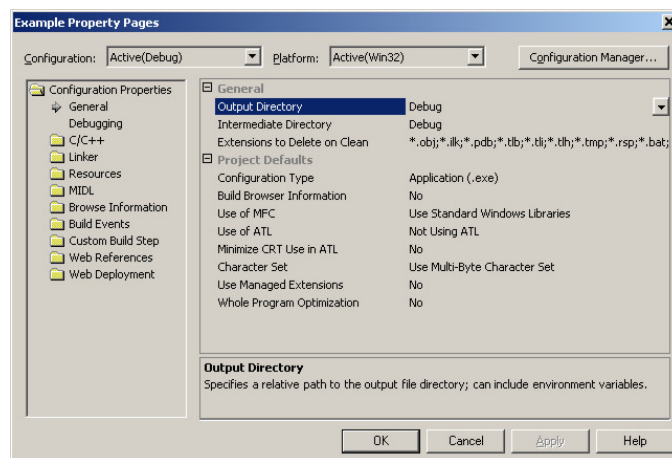


Figure 5.3: Property Pages

3. In Fig. 5.3, click C/C++, click General, click Additional Include Directories (see Fig. 5.4). In Fig. 5.4 click on the browse button, you'll obtain the Additional Include Directories dialog box as shown in Fig. 5.5

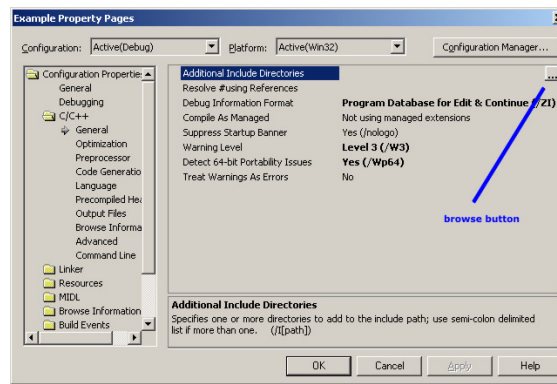


Figure 5.4: Property Pages (continued)

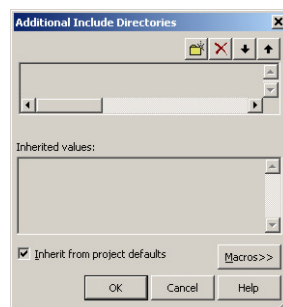


Figure 5.5: The Additional Include Directories dialog box

4. Suppose that your MATLAB path is C:\MATLAB7.

In Fig. 5.5, add the following directory to this dialog (Fig. 5.6):

C:\MATLAB7\extern\include

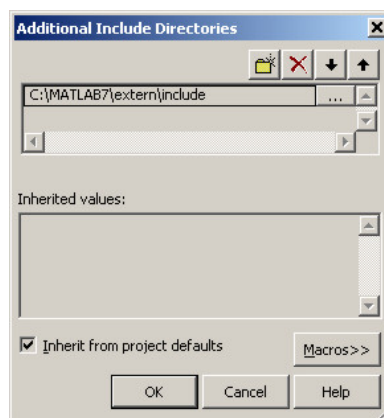


Figure 5.6: The Additional Include Directories dialog box (continued)

In Fig. 5.6, click OK. These directories are added to the Property Pages (see Fig 5.7).

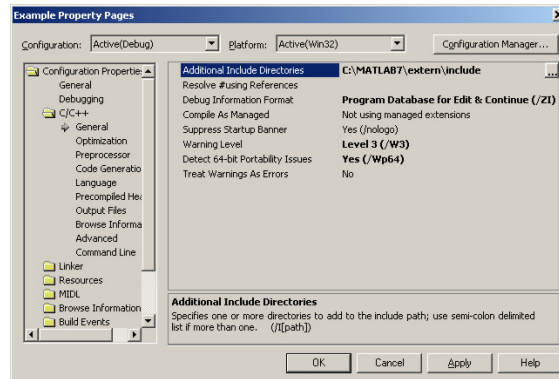


Figure 5.7: Property Pages (continued)

5. In Fig. 5.7, click Preprocessor, click Preprocessor Definitions (as shown in Fig. 5.8). Click the browse button, you'll obtain Fig. 5.9.

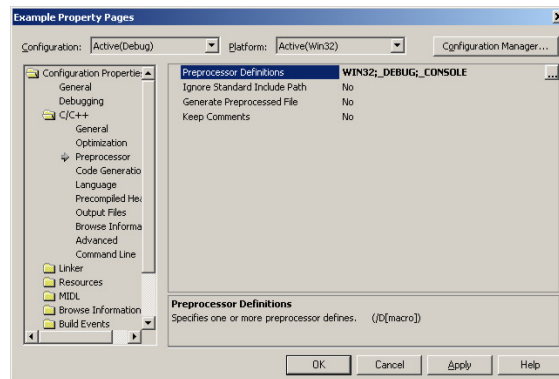


Figure 5.8: Property Pages (continued)

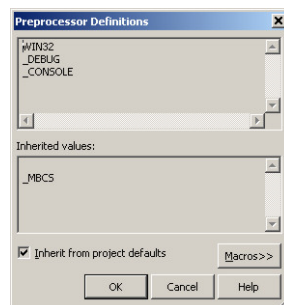


Figure 5.9: Preprocessor Definitions



6. In Fig 5.9, add the following lines to this dialog (see Fig. 5.10):

```
_WINDOWS
_AFXDLL
IBMPCL
MSVC
MSWIND
__STDC__
```

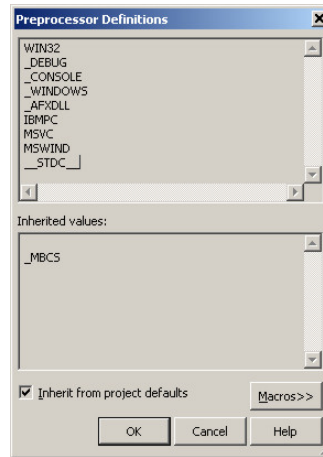


Figure 5.10: Preprocessor Definitions (continued)

7. In Fig. 5.10 click OK. The Preprocessor Definitions is changed as in Fig. 5.11.

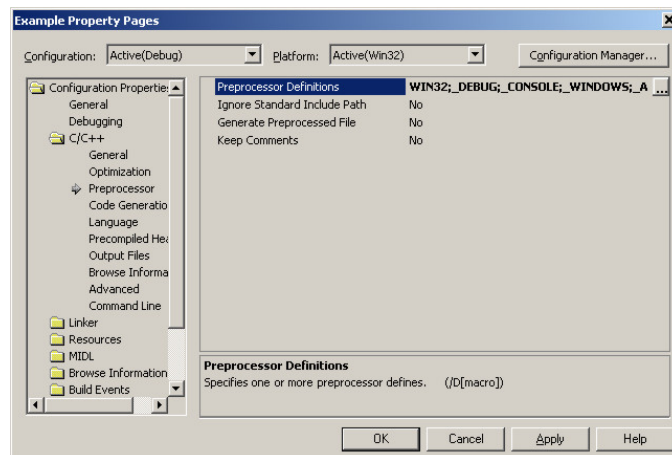


Figure 5.11: Property Pages (continued)

8. In Fig. 5.11, click Code Generation, click Runtime Library (as shown in Fig. 5.12).

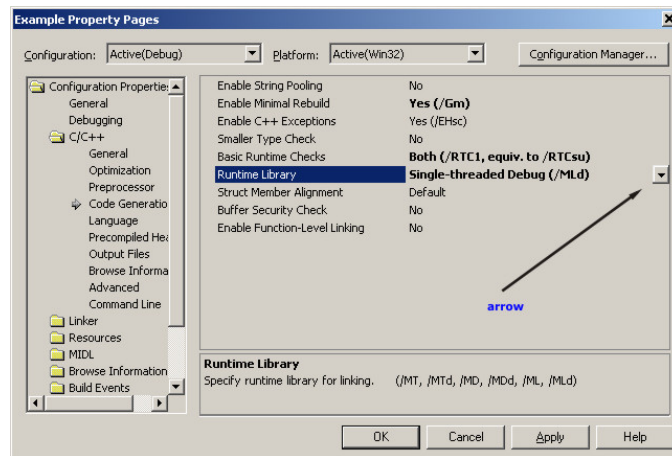


Figure 5.12: Property Pages (continued)

9. In Fig 5.12, click on the arrow button to select the option of Runtime Library: Multi-threaded Debug DLL (/MDd) as shown in Fig. 5.13.

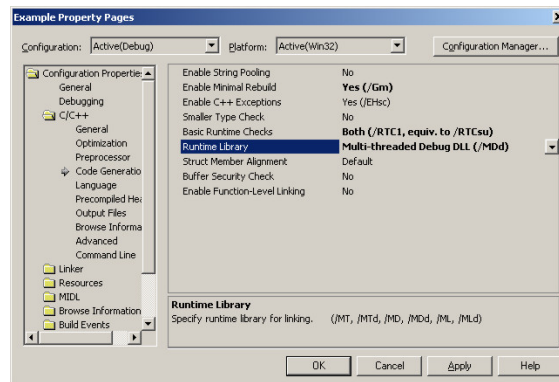


Figure 5.13: Property Pages (continued)

10. In Fig. 5.13, click Precompiled Headers, click Create/Use Precompiled Header, click on the arrow button to select **Not Using Precompiled Headers**, as shown in Fig. 5.14.

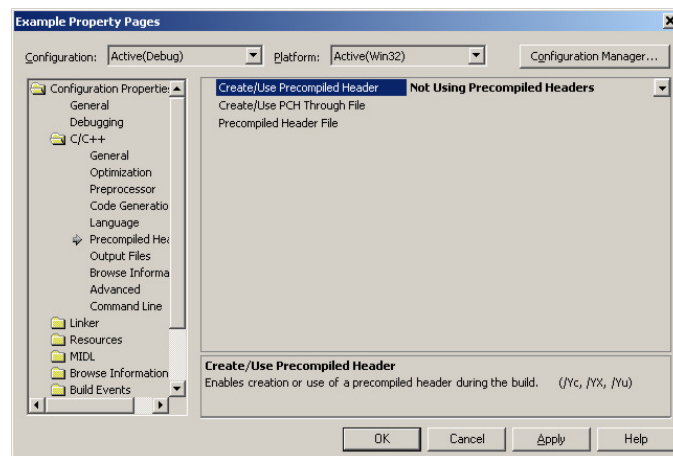


Figure 5.14: Property Pages (continued)

11. In Fig. 5.14, click Linker, click General, click Additional Library Directory, as shown in Fig. 5.15.

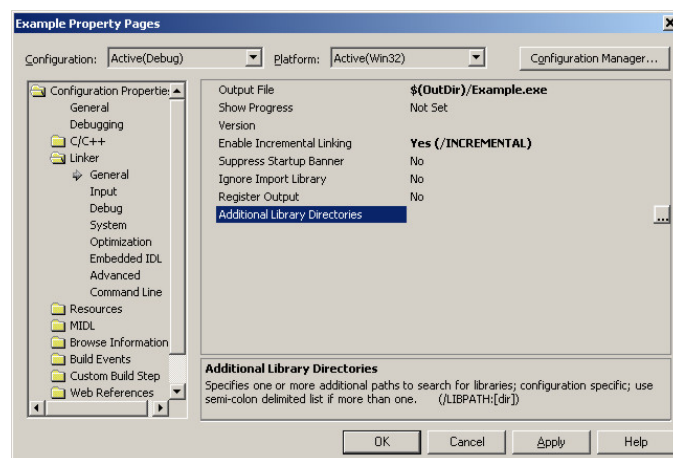


Figure 5.15: Property Pages (continued)

In Fig. 5.15, click on the browse button, you'll obtain the Additional Library Directory dialog box (Fig. 5.16).

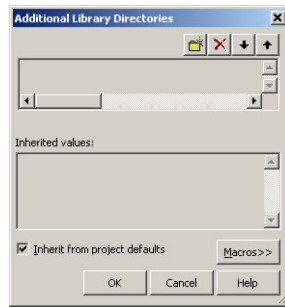


Figure 5.16: The Additional Library Directory dialog box

12. In Fig. 5.16, add the following directory to this dialog (see Fig. 5.17):

C:\MATLAB7\extern\lib \win32\microsoft\msvc70

Add C:\MATLAB7\extern\lib \win32\microsoft\msvc71 if you're using MSVC .Net 2003.

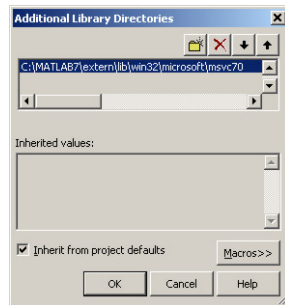


Figure 5.17: Additional Library Directory dialog box (continued)

Click OK. These directories are added in Property Pages as shown in Fig. 5.18.

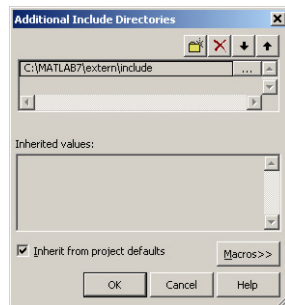


Figure 5.18: Property Pages (cont.)

13. In Fig. 5.15, click Input, click Additional Dependencies as shown in Fig. 5.19.

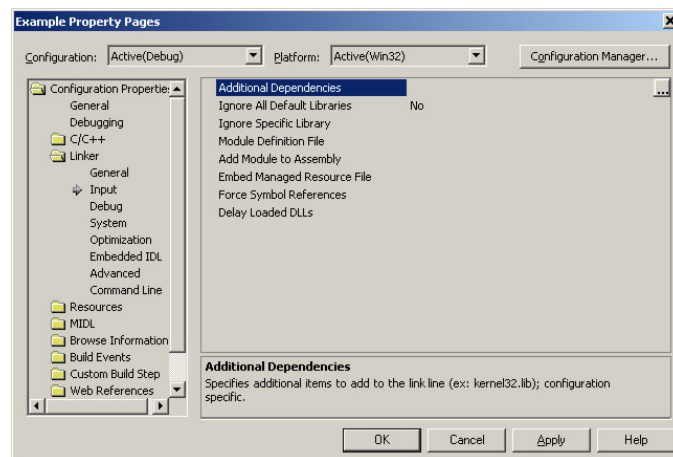


Figure 5.19: Property Pages (continued)

14. In Fig. 5.19, click the browse button, you'll obtain the Additional Dependencies dialog box (Fig. 5.20).

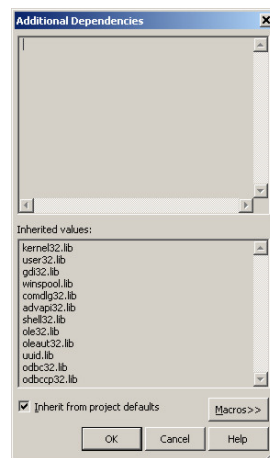


Figure 5.20: The Additional Dependencies dialog box

15. In Fig. 5.20, add the following libraries to this dialog (see Fig. 5.21).

```
libeng.lib
libfixedpoint.lib
libmat.lib
libmex.lib
libmwservices.lib
```

```

libmx.lib
libut.lib
mclcom.lib
mclcommmain.lib
mclmcr.lib
mclmcrrt.lib
mclxlmain.lib

```

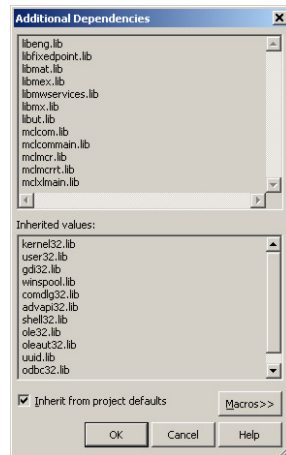


Figure 5.21: The Additional Dependencies dialog box (continued)

16. In Fig. 5.21 click OK. These libraries are added in Property Pages as shown in Fig. 5.22.

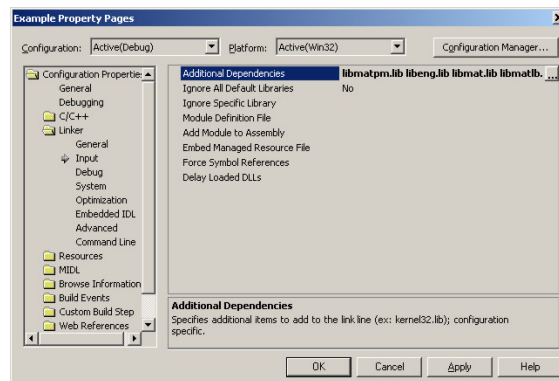


Figure 5.22: Property Pages (continued)

17. In Fig. 5.22 click Apply, then click OK.

You finished the setting of MSVC++.Net with MATLAB Compiler 4.

## 5.2 Testing of Project Setting

### 5.2.1 Writing a code for testing

After setting up the procedure as above, write the following simple code in a C++ file Example.cpp (see Fig. 5.23), then build and execute the file.

Listing code

---

```
#pragma warning(disable : 4995)

#include <iostream.h>
#include "mclcppclass.h"

int main() {

    cout << " Testing setting-up " << endl ;

    mwArray mw_test(1, 2, mxDOUBLE_CLASS) ;
    mw_test(1, 1) = 1.1 ;
    mw_test(1, 2) = 2.2 ;

    std::cout << mw_test << std::endl ;
    return 0 ;
}
```

---

end code

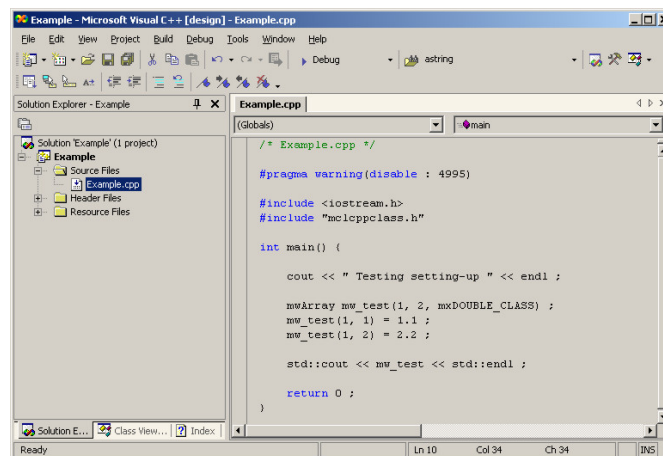


Figure 5.23: Testing the project setting

**Note**

In the code, the line:

```
#pragma warning(disable : 4995)
```

is used to ignore a warnings 4995.

To have more information about this warning, go to the Microsoft website:

<http://msdn.microsoft.com/library/>

### 5.2.2 Building and executing the project

1. In Fig. 5.23, click Build, click Rebuild Solution. Click Debug, click Start Without Debugging. You have no error and warning and an output result as shown in Fig.5.24.

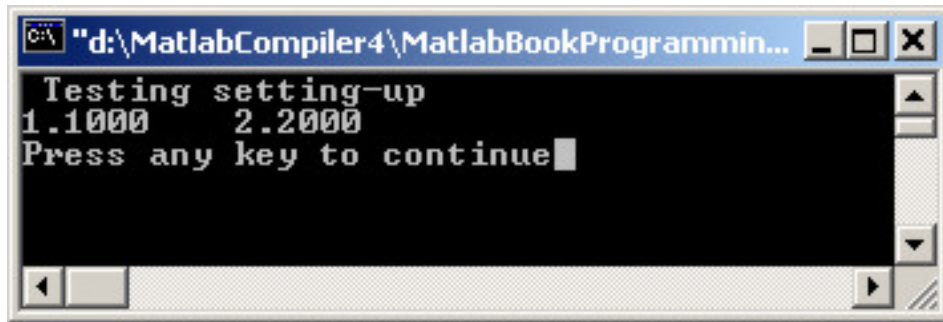


Figure 5.24: The output result

You completely finished the setting of the MSVC++.Net project with MATLAB Compiler

4. You can use this MSVC++.Net project as a template for using in another MSVC .Net projects.



**Part II:**  
**Creating and Using C/C++ Shared  
Libraries to Solve Mathematical  
Problems**



## Chapter 6

# Generating C and C<sup>++</sup> Shared Libraries from MATLAB M-Files for Using in Microsoft Visual C<sup>++</sup> .Net

This chapter describes how to generate a C and C<sup>++</sup> shared libraries from MATLAB M-files and use it in Microsoft Visual C<sup>++</sup> .Net (MSVC.Net). The MATLAB Compiler 4 will generate C/C<sup>++</sup> functions from M-files. These functions then be used as library functions in another C/C<sup>++</sup> functions.

If we compiler all mathematical functions in M-files we will have a mathematical library for C/C<sup>++</sup> functions. The main steps of the procedure to generate a C or C<sup>++</sup> shared library from M-files and use in Microsoft Visual C<sup>++</sup>.Net are:

1. Write the command to generate an dll-file from the MATLAB M-files.
2. Set up a project in Microsoft Visual C<sup>++</sup>.Net (MSVC.Net) for working with MATLAB Compiler 4 as described in Chapter 5.
3. Add the generated files in appropriate directories, set up the project property, and write the code to call the generated C/C<sup>++</sup> functions (in the dll-file).

The following sections describe an example for compiling an M-file. The procedure to compile another M-files is the same.

## 6.1 Generating a C Shared Library from a MATLAB M-File

The following are the steps of the procedure to generate a C shared library from a MATLAB M-file and use it in Microsoft Visual C++ .Net.

1. Create an M-file `myplus.m` as follows:

```
function y = myplus(x, y)
    z = x + y ;
```

2. Open the command prompt, go to the current directory, and write the command (Fig. 6.1):  
`mcc -B csharedlib:mypluslib myplus.m`

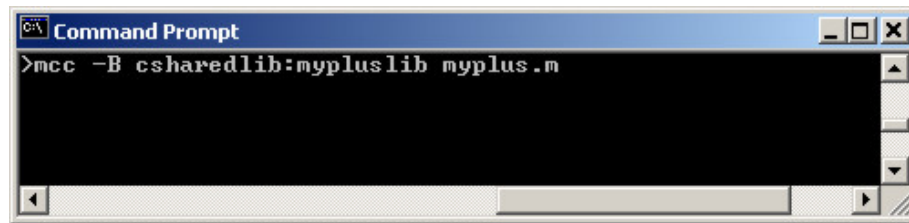


Figure 6.1: Command of an dll-file generation

This step will create 8 files in the current folder:

<code>mypluslib.c</code>	<code>mypluslib.exp</code>	<code>mypluslib.lib</code>
<code>mypluslib.ctf</code>	<code>mypluslib.exports</code>	<code>mypluslib_mcc_component_data.c</code>
<code>mypluslib.dll</code>	<code>mypluslib.h</code>	

3. Create a project named **Example** in Microsoft Visual C++.Net (MSVC.Net) and set up for working with MATLAB Compiler 4 as described in Chapter 5.
4. Copy two files `mypluslib.dll` and `mypluslib.ctf` into the folder **Debug** (Fig. 6.2).

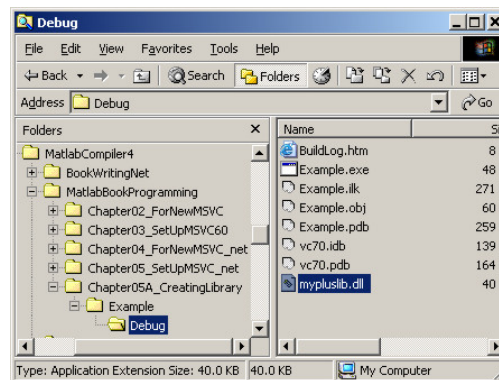


Figure 6.2: Copying the dll-file into the folder **Debug**

5. Copy two files `mypluslib.h` and `mypluslib.lib` into the folder **Example** (see Fig. 6.3).

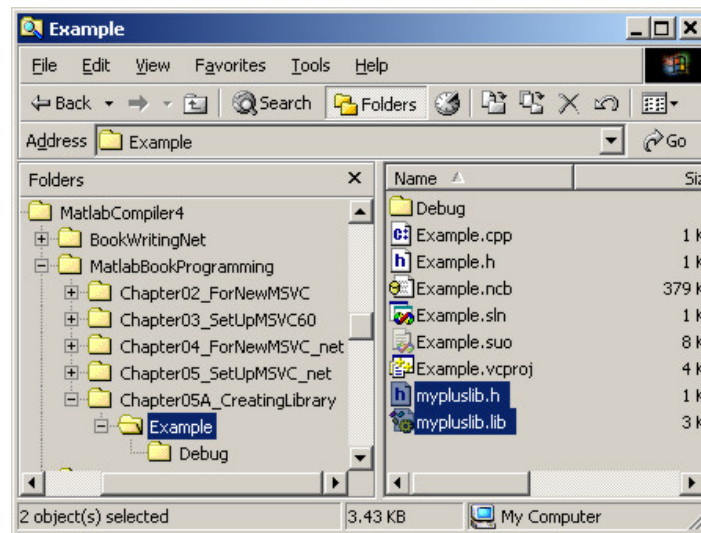


Figure 6.3: Copying two files into the folder **Example**

6. In Solution Explorer, right click on **Example** (see Fig. 6.4), click Property, and the Property Page dialog will appear (Fig. 6.5).

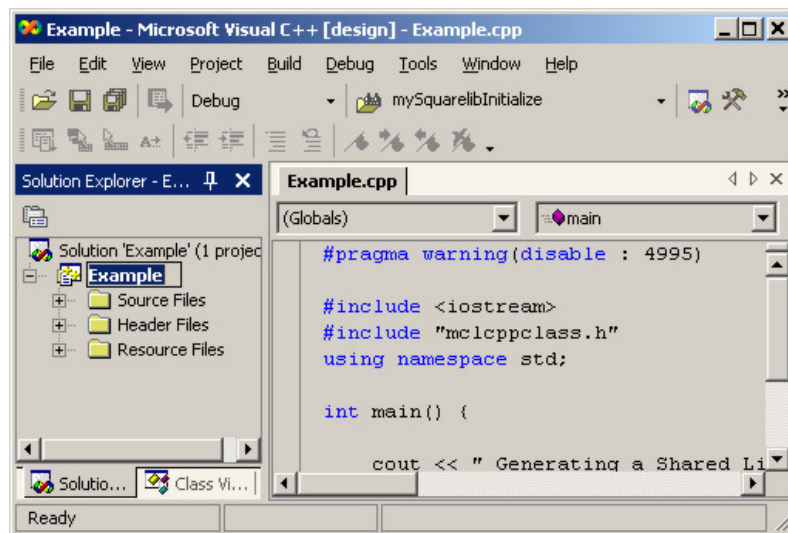


Figure 6.4: An MSVC.Net Project

In Fig. 6.5 click Linker, click Input, click Additional Dependencies, and click the button at the top-right corner. The Additional Dependencies dialog will appear, then add *myplus-lib.lib* to this dialog (Fig. 6.6). Click OK and click OK to close the dialogs.

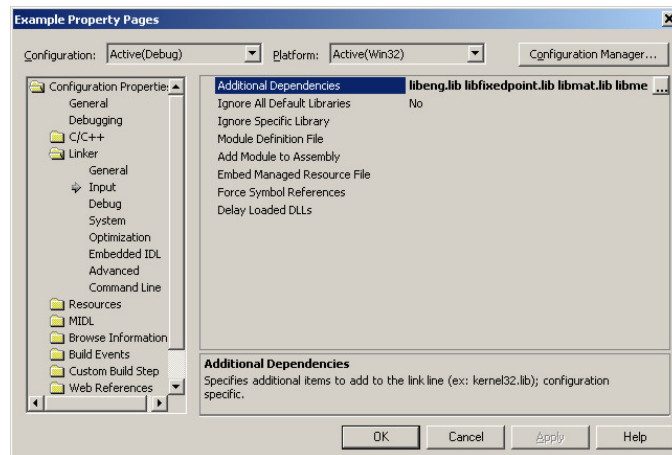


Figure 6.5: The MSVC.Net Project property

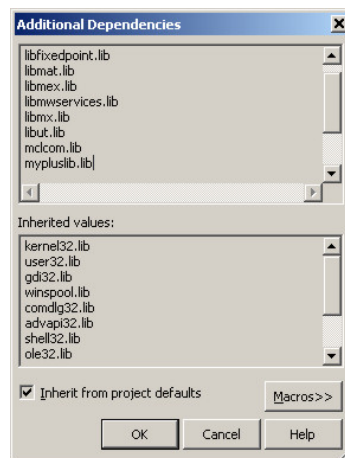


Figure 6.6: Adding the .lib file to the Additional Dependencies dialog

7. From the M-function *myplus.m* MATLAB Compiler 4 has generated a function (see this name in *mypluslib.h* file),

```
void mlfMyplus(int nargsout, mxArray** y, mxArray* a, mxArray* b);
```

,with a rule we'll discuss in Section 6.3. In this function *mlfMyplus(..)*, the arguments are:

nargout	: number of output (in this is case nargout = 1)
y	: output variable
a	: input variable
b	: input variable

Note that the MATLAB Compiler has capitalized **M** in the function name *mlfMyplus*.

## 6.2 Writing a Code to Call Functions in a C Shared Library

Following is the code in an MSVC .Net to call the function `mlfMyplus(..)`. Note that this MSVC .Net has set with MATLAB as described in Chapter 5.

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include <iostream.h>
#include "Example.h"

int main() {

    cout << " Generating a C Shared Library from MATLAB M-Files " ;

    Test obj;
    cout << endl ;
    cout << obj.CalculatePlus(3.4, 2.1) << endl ;

    return 0 ;
}
```

---

```
/* Example.h */

#include "mypluslib.h"
#include "mxUtilityCompilerVer4.h"

class Test {

public:
```

```

Test () {
    mclInitializeApplication(NULL,0);
    mypluslibInitialize();
}

~Test () {
    mypluslibTerminate();
    mclTerminateApplication();
}

double CalculatePlus(double a, double b) ;

} ;

/* ***** */

double Test::CalculatePlus(double a, double b) {

    /* step 1 : declare mxArray variables */
    mxArray *mx_a ;
    mxArray *mx_b ;
    mxArray *mx_y = NULL ;

    /* step 2 : assign memories */
    mx_a = mxCreateDoubleMatrix(1, 1, mxREAL);
    mx_b = mxCreateDoubleMatrix(1, 1, mxREAL);
    mx_y = mxCreateDoubleMatrix(1, 1, mxREAL);

    /* step 3 : convert C/C++ double to mxArray */
    double2mxArray_scalarReal(a, mx_a) ;
    double2mxArray_scalarReal(b, mx_b) ;

    /* step 4 : call the implemental function */
    mlfMyplus(1, &mx_y, mx_a, mx_b);

    /* step 5 : convert back mxArray to C/C++ double */
    double result = mxArray2double_scalarReal(mx_y) ;

    /* step 6 : free memories */
    mxDestroyArray(mx_a) ;

```



```

    mxDestroyArray(mx_b) ;
    mxDestroyArray(mx_y) ;

    return result ;

}

```

---

end code

## Remarks

1. See the file `mxUtilityCompilerVer4.h` in Chapter 7.
2. From the eight generated files, we used only four files, `mypluslib.h`, `mypluslib.lib`, `mypluslib.dll`, and `mypluslib.ctf`.
3. After you built your project, MATLAB also created a folder **mypluslib\_mcr** in the folder **Debug**.
4. If you want to use multiple libraries see Section 6.4.
5. To generate a C shared library from multiple M-files, we just normally add the adding M-files. For example:

```
mcc -B csharedlib:mymathlib myplus.m mymtimes.m
```

## 6.3 Generated Functions from MATLAB Compiler 4

The C function generated by MATLAB Compiler 4 from an M-function has a form depends on the M-function.

- With an M-functions with no return values, the C function has the form:

```
void mlf<function-name>(<list_of_input_variables>);
```

- With an M-function with at least one return value, the C function has the form:

```
void mlf<function-name>(int number_of_return_values,
                        <list_of_pointer_to_return_variables>,
                        <list_of_input_variables>);
```

for example:

```
void mlfMyplus(int nargout, mxArray** y, mxArray* a, mxArray* b)
```

This generated C function has the pattern:

1. the return type is always *void*
2. the first argument, *nargout*, is the number of output variables in the original M-function.
3. the next argument(s) are output variables in the original M-function. These output variables have the type double-pointer to mxArray, for example mxArray\*\* *y*.
4. the next arguments are input variables in the original M-function. These variables have the type pointer to mxArray, for example, mxArray\* *a*.

## 6.4 Using Multiple C Shared Libraries

We often use the multiple C shared libraries in a C/C++ project. Adding and setting multiple C shared libraries in MSVC .Net are same as in single shared library (described in above). For example, suppose that we have two C shared libraries *myplus* and *myminus*. We'll set up the *myminus* library same as we did for the *myplus* library as described in the previous sections:

- add two more files myminuslib.dll and myminuslib.ctf into the folder **Debug** (see Fig. 6.2).
- add two more files myminuslib.h and myminuslib.lib into the folder **Example** (see Fig. 6.3).
- add myminuslib.lib to the Additional Dependencies dialog (see Fig. 6.6).

The following is the code to call functions in the multiple C shared libraries.

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << " Using functions in multiple C shared libraries" ;
    cout << endl ;

    Test obj;
    obj.Calculate(3.4, 2.1) ;

    return 0 ;
}
```

```
}
```

---

```
/* Example.h */
```

```
#include <iostream.h>
```

```
#include "mypluslib.h"
```

```
#include "myminuslib.h"
```

```
#include "mxUtilityCompilerVer4.h"
```

```
class Test {
```

```
public:
```

```
    Test () {
```

```
        mclInitializeApplication(NULL,0);
```

```
        mypluslibInitialize();
```

```
        myminuslibInitialize();
```

```
    }
```

```
    ~Test () {
```

```
        mypluslibTerminate();
```

```
        myminuslibTerminate();
```

```
        mclTerminateApplication();
```

```
    }
```

```
    void Calculate (double a, double b) ;
```

```
} ;
```

```
/* ***** */
```

```
void Test::Calculate(double a, double b) {
```

```
    /* step 1 : declare mxArray variables */
```

```
    mxArray *mx_a ;
```

```
    mxArray *mx_b ;
```

```

mxArray *mx_y  = NULL ;
mxArray *mx_y2 = NULL ;

/* step 2 : assign memories */
mx_a    = mxCreateDoubleMatrix(1, 1, mxREAL);
mx_b    = mxCreateDoubleMatrix(1, 1, mxREAL);
mx_y    = mxCreateDoubleMatrix(1, 1, mxREAL);
mx_y2   = mxCreateDoubleMatrix(1, 1, mxREAL);

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal(a, mx_a) ;
double2mxArray_scalarReal(b, mx_b) ;

/* step 4 : call the implemental function */
mlfMyplus (1, &mx_y , mx_a, mx_b);
mlfMyminus(1, &mx_y2, mx_a, mx_b);

/* step 5: convert back mxArray to C/C++ double */
double resultPlus  = mxArray2double_scalarReal(mx_y)  ;
double resultMinus = mxArray2double_scalarReal(mx_y2) ;

/* step 6: free memories */
mxDestroyArray(mx_a) ;
mxDestroyArray(mx_b) ;
mxDestroyArray(mx_y) ;
mxDestroyArray(mx_y2) ;

cout << "Result of plus : \t" << resultPlus  << endl ;
cout << "Result of minus: \t" << resultMinus << endl ;

}

```

---

end code

See the file mxUtilityCompilerVer4.h in Chapter 7.

## 6.5 Generating a C++ Shared Library From a MATLAB M-File

The steps of generating a C++ shared library are the same as generating a C shared library.

1. Write an M-file, for example `myplus.m`:

```
function y = myplus(x, y)
z = x + y ;
```

2. Open the command prompt, go to the current directory, and write the command:  
`mcc -W cpplib:cppmypluslib -T link:lib myplus.m`
3. Set up the generated files as the same in Section 6.1.
4. The MATLAB Compiler 4 creates an implement function (see this name in `cppmypluslib.h` file),

```
void myplus(int nargout, mxArray& y, const mxArray& a, const mxArray& b);
```

, with the rule we'll discuss in Section 6.7. In this function `myplus(..)`, the arguments are:

<code>nargout</code>	: number of output (in this is case <code>nargout = 1</code> )
<code>y</code>	: output variable
<code>a</code>	: input variable
<code>b</code>	: input variable

## 6.6 Writing a Code to Call Functions in a C++ Shared Library

Following is the code in an MSVC .Net to call the function `myplus(..)`. Note that this MSVC .Net has set with MATLAB as described in Chapter 5.

Listing code

---

```

/* Example.cpp */

#pragma warning(disable : 4995)
#include <iostream.h>
#include "Example.h"

int main() {

    cout << " Generating a C++ Shared Library from MATLAB M-Files " ;

    Test obj;
    cout << endl ;
    cout << obj.CalculatePlus(3.4, 2.1) << endl ;

    return 0 ;
}

```

---

```

/* Example.h */

#include "cppmypluslib.h"
#include "mwUtilityCompilerVer4.h"

class Test {

public:

    Test () {
        mclInitializeApplication(NULL,0);
        cppmypluslibInitialize();
    }

    ~Test () {
        cppmypluslibTerminate();
        mclTerminateApplication();
    }

    double CalculatePlus(double a, double b) ;

```

```

} ;

/* ***** */

double Test::CalculatePlus(double a, double b) {

    /* declare mxArray variables */
    mxArray mw_y(1, 1, mxDOUBLE_CLASS) ;
    mxArray mw_a(1, 1, mxDOUBLE_CLASS) ;
    mxArray mw_b(1, 1, mxDOUBLE_CLASS) ;

    /* convert C/C++ double to mxArray */
    mw_a(1,1) = a ;
    mw_b(1,1) = b ;

    /* call the implemental function */
    myplus(1, mw_y, mw_a, mw_b);

    /* convert back mxArray to C/C++ double */
    double result = (double) mw_y(1,1) ;

    return result ;

}

```

---

end code

See the file `mwUtilityCompilerVer4.h` in Chapter 7.

## 6.7 Generated C++ Functions from MATLAB Compiler 4

The C++ function generated by MATLAB Compiler 4 from an M-function has a form depends on the M-function.

- With an M-function with no return values, the C function has the form:

```
void <function-name>(<list_of_input_variables>);
```

- With an M-function with at least one return value, the C function has the form:

```
void <function-name>(int number_of_return_values,
                    <list_of_pointer_to_return_variables>,
                    <list_of_input_variables>);
```

for example:

```
void myplus(int nargout, mxArray& y, const mxArray& a, const mxArray& b)
```

This generated C++ function has the pattern:

1. the return type is always *void*
2. the first argument, *nargout*, is the number of output variables in the original M-function.
3. the next argument(s) are output variables in the original M-function.
4. the next arguments are input variables in the original M-function.



## Chapter 7

# Transfer of Values between C/C++ double, mxArray, and mwArray

MATLAB Compiler has two principle types, `mwArray` and `mxArray`. We can use `mwArray` and `mxArray` as the new types in the C/C++ code or make a transfer of values between C/C++ double, `mwArray`, and `mxArray`. This chapter shows transfers between C/C++ double, `mwArray`, and `mxArray`. These transfers are very useful in working on MATLAB Compiler 4 and are used in Chapter 8 to Chapter 16. For more information of `mxArray` and `mwArray` see [3] and [2].

### 7.1 Transfer of Values between C/C++ double and `mxArray`

This section shows how to transfer values between C/C++ *double* type and `mxArray` through the example code. We write an utility file `mxUtilityCompilerVer4.h` for convenience in using the transfer of values between C/C++ double and `mxArray`. The following example code use functions in this file `mxUtilityCompilerVer4.h` to implement the transfers. The `mxUtilityCompilerVer4.h` file is shown at the end of this chapter.

#### 1. scalar transfer:

##### a. real scalar

```
double    db_scalar  = 1.1    ;  
mxArray *mx_scalar  = NULL   ;  
mx_scalar = mxCreateDoubleMatrix(1, 1, mxREAL) ;
```

```
double2mxArray_scalarReal(db_scalar, mx_scalar) ;

double db_scalarReturn = mxArray2double_scalarReal(mx_scalar) ;
cout << " db_scalarReturn = " << db_scalarReturn << endl ;

mxDestroyArray(mx_scalar) ;
```

#### b. complex scalar

```
double db_Real = 1.1 ;
double db_Imag = 2.2 ;

mxArray *mx_Complex = NULL ;
mx_Complex = mxCreateDoubleMatrix(1, 1, mxCOMPLEX) ;

double2mxArray_scalarComplex(db_Real, db_Imag, mx_Complex) ;

double db_returnReal, db_returnImag ;
mxArray2double_scalarComplex(mx_Complex, db_returnReal, db_returnImag) ;
cout << " db_returnReal = " << db_returnReal << endl ;
cout << " db_returnImag = " << db_returnImag << endl ;

mxDestroyArray(mx_Complex) ;
```

### 2. vector transfer:

#### a. real vector

```
double db_vector[3] = { 1.1, 2.2, 3.3 } ;
int vectorSize = 3 ;

/* row vector */
mxArray *mx_vector = NULL ;
mx_vector = mxCreateDoubleMatrix(vectorSize, 1, mxREAL) ;

double2mxArray_vectorReal(db_vector, mx_vector) ;

double *db_vectorReturn = new double [vectorSize] ;
mxArray2double_vectorReal(mx_vector, db_vectorReturn) ;
```

```

int i ;
for (i=0; i<vectorSize; i++) {
    cout << db_vectorReturn[i] << endl ;
}

```

```

mxDestroyArray(mx_vector) ;
delete [] db_vectorReturn ;

```

#### b. complex vector

```

double db_Real[3] = { 1.1, 2.2, 3.3 } ;
double db_Imag[3] = { 4.4, 5.5, 6.6 } ;
int vectorSize = 3 ;

/* row vector */
mxArray *mx_complex = NULL ;
mx_complex = mxCreateDoubleMatrix(vectorSize, 1, mxCOMPLEX) ;

double2mxArray_vectorComplex(db_Real, db_Imag, mx_complex) ;

double *db_returnReal = new double [vectorSize] ;
double *db_returnImag = new double [vectorSize] ;

mxArray2double_vectorComplex(mx_complex, db_returnReal, db_returnImag) ;

int i ;
for (i=0; i<vectorSize; i++) {
    cout << db_returnReal[i] << " + " ;
    cout << db_returnImag[i] << "i" << endl ;
}

mxDestroyArray(mx_complex) ;
delete [] db_returnReal ;
delete [] db_returnImag ;

```

### 3. matrix transfer:

#### a. real matrix

```
double db_A[3][3] = {{ 1.1, 2.2, 3.3} , {4.4, 5.5, 6.6} , {7.7, 8.8, 9.9} } ;
```

```
int row = 3 ;
```

```
int col = 3 ;
```

```
mxArray *mx_A = NULL ;
```

```
mx_A = mxCreateDoubleMatrix(row, col, mxREAL) ;
```

```
double2mxArray_matrixReal(&db_A[0][0], mx_A) ;
```

```
double **db_ReturnA ;
```

```
db_ReturnA = new double* [row] ;
```

```
int i ;
```

```
    for (i=0; i<row; i++) {
```

```
        db_ReturnA[i] = new double [col] ;
```

```
    }
```

```
mxArray2double_matrixReal(mx_A, db_ReturnA) ;
```

```
printMatrix(db_ReturnA, row, col) ;
```

```
mxDestroyArray(mx_A) ;
```

```
delete [] db_ReturnA ;
```

#### b. complex matrix

```
double db_Real[3][3] = {{ 1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}, {7.7, 8.8, 9.9} } ;
```

```
double db_Imag[3][3] = {{ 11 , 12 , 13 }, {14 , 15 , 16 }, {17 , 18 , 19 } } ;
```

```
int row = 3 ;
```

```
int col = 3 ;
```

```
mxArray *mx_complex = NULL ;
```

```
mx_complex = mxCreateDoubleMatrix(row, col, mxCOMPLEX) ;
```

```
double2mxArray_matrixComplex(&db_Real[0][0], &db_Imag[0][0], mx_complex) ;
```

```

double **db_returnReal = new double* [row] ;
double **db_returnImag = new double* [row] ;

int i ;
    for (i=0; i<row; i++) {
        db_returnReal[i] = new double [col] ;
        db_returnImag[i] = new double [col] ;
    }

mxArray2double_matrixComplex(mx_complex, db_returnReal, db_returnImag) ;
printMatrix(db_returnReal, row, col) ;
cout << endl ;

printMatrix(db_returnImag, row, col) ;

mxDestroyArray(mx_complex) ;
delete [] db_returnReal ;
delete [] db_returnImag ;

```

### Remark

If a matrix in the double-pointer type, we pass to the function as follow:

```

double ** db_Real = new double* [row] ;
double ** db_Imag = new double* [row] ;

for(i=0; i<row; i++) {
    db_Real[i] = new double [col] ;
    db_Imag[i] = new double [col] ;
}

...
mxArray *mx_complex = NULL ;
mx_complex = mxCreateDoubleMatrix(row, col, mxCOMPLEX) ;

double2mxArray_matrixComplex(db_Real, db_Imag, mx_complex) ;

...
mxDestroyArray(mx_complex) ;
delete [] db_Real ;
delete [] db_Imag ;

```

## 7.2 Transfer of Values from C/C++ double to mxArray

This section shows how to transfer values from C/C++ *double* type to mxArray through the example code.

### 1. scalar transfer:

#### a. real scalar

```
mxArray mw_scalar(1, 1, mxDOUBLE_CLASS) ;
mw_scalar(1,1) = 1.4 ;

/* or */
double db_scalar2 = 1.2 ;
mxArray mw_scalar2(1, 1, mxDOUBLE_CLASS) ;
mw_scalar2 = db_scalar2 ;
```

#### b. complex scalar

```
mxArray mw_scalarComplex(1, 1, mxDOUBLE_CLASS, mxCOMPLEX) ;
mw_scalarComplex(1,1).Real() = 2.2 ;
mw_scalarComplex(1,1).Imag() = 3.3 ;

/* or */
double db_scalarReal = 4.4 ;
double db_scalarImag = 5.5 ;

mxArray mw_scalarComplex2(1, 1, mxDOUBLE_CLASS, mxCOMPLEX) ;
mw_scalarComplex2(1,1).Real() = db_scalarReal ;
mw_scalarComplex2(1,1).Imag() = db_scalarImag ;
```

### 2. vector transfer:

#### a. real vector

```
/* row vector */
double db_vector[6] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0} ;
int vectorSize = 6 ;

mxArray mw_vector(vectorSize, 1, mxDOUBLE_CLASS) ;
mw_vector.SetData(db_vector, vectorSize) ;
```

### b. complex vector

```
double realdata[4] = {1.0, 2.0, 3.0, 4.0};
double imagdata[4] = {10.0, 20.0, 30.0, 40.0};
int aSize = 4 ;

mwArray mw_vectorComplex(aSize, 1, mxDOUBLE_CLASS, mxCOMPLEX);
mw_vectorComplex.Real().SetData(realdata, aSize);
mw_vectorComplex.Imag().SetData(imagdata, aSize);
```

## 3. matrix transfer:

### a. real matrix

```
int i,j ;

double db_matrix[3][2] = { {1.0, 2.0} , {3.0, 4.0}, {5.0, 6.0} } ;
int arow = 3 ;
int acol = 2 ;

mwArray mw_matrix = double2mwArray_matrixReal(&db_matrix[0][0], arow, acol) ;
std::cout << mw_matrix << std::endl ;

/* or */

/* if matrix in double-pointer */
double **db_matrixA ;

db_matrixA = new double*[arow] ;
for(i=0; i<arow; i++) {
    db_matrixA[i] = new double [acol] ;
}

for (i=0; i<arow; i++) {
    for (j=0; j<acol; j++) {
        db_matrixA[i][j] = 1.2 + i+j ; // assign a number
    }
}
```

```

mWArray mw_matrixA = double2mWArray_matrixReal(db_matrixA, arow, acol) ;
std::cout << mw_matrixA << std::endl ;

```

```

delete [] db_matrixA ;

```

See the functions `double2mWArray_matrixReal(..)` in `mwUtilityCompilerVer4.h` at the end of this chapter.

## b. complex matrix

```

double db_AReal[3][4] = { { 1.1, 2.2 , 3.3 , 4.4 } ,\
                          { 5.5, 6.6 , 7.7 , 8.8 } ,\
                          { 9.9, 10.10, 11.11, 12.12 } } ;

```

```

double db_AImag[3][4] = { { 0.1, 0.2 , 0.3 , 0.4 } ,\
                          { 0.5, 0.6 , 0.7 , 0.8 } ,\
                          { 0.9, 0.21, 0.21 , 0.22 } } ;

```

```

int row = 3 ;

```

```

int col = 4 ;

```

```

mWArray mw_complex = double2mWArray_matrixComplex(&db_AReal[0][0],
                                                    &db_AImag[0][0], row, col) ;

```

```

cout << " mw_complex " << endl ;

```

```

std::cout << mw_complex << std::endl ;

```

```

/* or */

```

```

/* if matrix in double-pointer */

```

```

double ** db_Real = new double* [row] ;

```

```

double ** db_Imag = new double* [row] ;

```

```

for(i=0; i<row; i++) {
    db_Real[i] = new double [col] ;
    db_Imag[i] = new double [col] ;
}

```

```

for (i=0; i<row; i++) {

```



```

        db_Real[i] = &db_AReal[i][0] ;
        db_Imag[i] = &db_AImag[i][0] ;
    }

    mxArray mw_complexOther = double2mwArray_matrixComplex(db_Real, db_Imag
                                                            , row, col) ;

    cout << " mw_complex other: " << endl ;
    std::cout << mw_complexOther << std::endl ;

    delete [] db_Real ;
    delete [] db_Imag ;

```

See the functions `double2mwArray_matrixComplex(..)` in `mwUtilityCompilerVer4.h` at the end of this chapter.

## 7.3 Transfer of Values from mxArray to C/C++ double

This section shows how transfer values from mxArray to C/C++ *double* type through the example code.

### 1. scalar transfer:

#### a. real scalar

```

    mxArray mw_scalar(1, 1, mxDOUBLE_CLASS) ;
    mw_scalar(1,1) = 1.4 ;
    double db_scalar = (double) mw_scalar(1,1) ;

```

#### b. complex scalar

```

    mxArray mw_scalarComplex(1, 1, mxDOUBLE_CLASS, mxCOMPLEX) ;
    mw_scalarComplex(1,1).Real() = 2.2 ;
    mw_scalarComplex(1,1).Imag() = 3.3 ;

    double db_scalarReal = (double) mw_scalarComplex(1,1).Real() ;
    double db_scalarImag = (double) mw_scalarComplex(1,1).Imag() ;

```

### 2. vector transfer:

#### a. real vector

```

...
/* suppose that mw_vector already had values */
int vectorSize = 6 ;
double *db_vector2 = new double[vectorSize] ;
mwArray2double_vectorReal(mw_vector, db_vector2) ;

for (i=0; i<vectorSize; i++) {
    cout << db_vector2[i] << endl ;
}

delete [] db_vector2 ;

```

See the function `mwArray2double_vectorReal(..)` in `mwUtilityCompilerVer4.h` at the end of this chapter.

#### **b. complex vector**

```

...
/* suppose that mw_vectorComplex already had values */

int aSize = 4 ;

double* db_vectorReal = new double [aSize] ;
double* db_vectorImag = new double [aSize] ;
mwArray2double_vectorComplex(mw_vectorComplex, db_vectorReal, db_vectorImag) ;

for (i=0; i<aSize; i++) {
    cout << db_vectorReal[i] << " + " << db_vectorImag[i] << "i" << endl ;
}
cout << endl ;

delete [] db_vectorReal ;
delete [] db_vectorImag ;

```

See the function `mwArray2double_vectorComplex(..)` in `mwUtilityCompilerVer4.h` at the end of this chapter.

### **3. matrix transfer:**

#### **a. real matrix**

```

int arow = 3 ;
int acol = 2 ;

double **db_matrixA = new double* [arow] ;
for (i=0; i<arow; i++) {
    db_matrixA[i] = new double [acol] ;
}

...

/* suppose that mw_matrix already had values */
mwArray2double_matrixReal(mw_matrix, db_matrixA) ;

delete [] db_matrixA ;

```

See the function `mwArray2double_matrixReal(..)` in `mwUtilityCompilerVer4.h` at the end of this chapter.

#### **b. complex matrix**

```

int row = 3 ;
int col = 4 ;

double ** db_Real = new double* [row] ;
double ** db_Imag = new double* [row] ;

for(i=0; i<row; i++) {
    db_Real[i] = new double [col] ;
    db_Imag[i] = new double [col] ;
}

...

/* suppose that mw_matrix already had values */
mwArray2double_matrixComplex(mw_complex, db_Real, db_Imag) ;

delete [] db_Real ;
delete [] db_Imag ;

```

See the function `mwArray2double_matrixComplex(..)` in `mwUtilityCompilerVer4.h` at the end of this chapter.

## 7.4 The Code of the Utility File mxUtilityCompilerVer4.h

```

/* mxUtilityCompilerVer4.h */

/* ***** */
/* I. Transfer values from C/C++ double to mxArray */

/* 1a. transfer a C/C++ double scalar to a real mxArray */
void double2mxArray_scalarReal (double cpp, mxArray* mx_pointer) {

    double db_bufx[1] ;
    db_bufx[0] = cpp ;

    memcpy( mxGetPr(mx_pointer), db_bufx, 1*sizeof(double) ) ;

}

/* ***** */
/* 1b. transfer C/C++ double scalars to a complex mxArray */
void double2mxArray_scalarComplex (double cppReal, double cppImag, mxArray* mx_pointer) {

    double db_bufReal[1] ;
    double db_bufImag[1] ;

    db_bufReal[0] = cppReal ;
    db_bufImag[0] = cppImag ;

    memcpy( mxGetPr(mx_pointer), db_bufReal, 1*sizeof(double) ) ;
    memcpy( mxGetPi(mx_pointer), db_bufImag, 1*sizeof(double) ) ;

}

/* ***** */

/* 2a. transfer a C/C++ double vector to a real mxArray */
void double2mxArray_vectorReal (double* db_vector, mxArray* mx_pointer) {

    /* row vector has row =1 */

    int row = mxGetM(mx_pointer) ; /* number of rows */
    int col = mxGetN(mx_pointer) ; /* number of columns */

```

```

    int vectorSize ;

    if ( row > col )    { vectorSize = row ;}
    else                { vectorSize = col ;}

    memcpy(mxGetPr(mx_pointer), db_vector,vectorSize*sizeof(double));

}

/* ***** */

/* 2b. transfer C/C++ double vectors to a complex mxArray */
void double2mxArray_vectorComplex (double* db_vectorReal, double* db_vectorImag
                                   , mxArray* mx_pointer) {

    /* row vector has row =1 */

    int row = mxGetM(mx_pointer) ; /* number of rows    */
    int col = mxGetN(mx_pointer) ; /* number of columns */

    int vectorSize ;

    if ( row > col )    { vectorSize = row ;}
    else                { vectorSize = col ;}

    memcpy(mxGetPr(mx_pointer), db_vectorReal, vectorSize*sizeof(double));
    memcpy(mxGetPi(mx_pointer), db_vectorImag, vectorSize*sizeof(double));

}

/* ***** */

/* 3a. transfer a C/C++ double matrix to a real mxArray */
void double2mxArray_matrixReal (double** db_matrix, mxArray* mx_pointer) {

    int row = mxGetM(mx_pointer) ; /* number of rows    */
    int col = mxGetN(mx_pointer) ; /* number of columns */

    double* db_vector ;
    db_vector = new double [row*col] ;

```

```

int i, j, index ;

for(j=0; j<col; j++) {
    for(i=0; i<row; i++) {

        index = j*row + i ;

        db_vector[index] = db_matrix[i][j] ;

    }
}

memcpy(mxGetPr(mx_pointer), db_vector, row*col*sizeof(double));

delete[] db_vector ;

}

/* ***** */

/* 3a. transfer a C/C++ double matrix to a real mxArray */
void double2mxArray_matrixReal(double* addressMatrix00, mxArray* mx_pointer) {

    int row = mxGetM(mx_pointer) ; /* number of rows */
    int col = mxGetN(mx_pointer) ; /* number of columns */

    /*assign memories for a buffer */
    int i, j ;
    double **db_matrixbuf ;
    db_matrixbuf = new double*[row] ;

    for(i=0; i<row; i++) {
        db_matrixbuf[i] = new double [col] ;
    }

    /* set address for rows */
    for(i=0; i<row; i++) {
        db_matrixbuf[i] = addressMatrix00 + i*col ;
    }
}

```

```

double* db_vector ;
db_vector = new double [row*col] ;

int index ;

    for(j=0; j<col; j++) {
        for(i=0; i<row; i++) {

            index = j*row + i ;
            db_vector[index] = db_matrixbuf[i][j] ;

        }
    }

memcpy(mxGetPr(mx_pointer), db_vector, row*col*sizeof(double));

delete[] db_vector ;
delete[] db_matrixbuf ;

}

/* ***** */
/* 3b. transfer C/C++ double matrixes to a complex mxArray */
void double2mxArray_matrixComplex (double** db_matrixReal, double** db_matrixImag
                                   , mxArray* mx_pointer) {

    int row = mxGetM(mx_pointer) ; /* number of rows */
    int col = mxGetN(mx_pointer) ; /* number of columns */

    double* db_vectorReal ;
    double* db_vectorImag ;

    db_vectorReal = new double [row*col] ;
    db_vectorImag = new double [row*col] ;

    int i, j, index ;

```

```

        for(j=0; j<col; j++) {
            for(i=0; i<row; i++) {
                index = j*row + i ;

                db_vectorReal[index] = db_matrixReal[i][j] ;
                db_vectorImag[index] = db_matrixImag[i][j] ;

            }
        }

        memcpy(mxGetPr(mx_pointer), db_vectorReal, row*col*sizeof(double));
        memcpy(mxGetPi(mx_pointer), db_vectorImag, row*col*sizeof(double));

        delete[] db_vectorReal ;
        delete[] db_vectorImag ;

    }

/* ***** */
/* 3b. transfer C/C++ double matrixes to a complex mxArray */
void double2mxArray_matrixComplex(double* addressReal00, double* addressImag00
                                   , mxArray* mx_pointer) {

    int row = mxGetM(mx_pointer) ; /* number of rows */
    int col = mxGetN(mx_pointer) ; /* number of columns */

    /*assign memories for a buffer */
    int i, j ;
    double **db_bufReal ;
    double **db_bufImag ;

    db_bufReal = new double*[row] ;
    db_bufImag = new double*[row] ;

    for(i=0; i<row; i++) {
        db_bufReal[i] = new double [col] ;
        db_bufImag[i] = new double [col] ;
    }

    /* set address for rows */

```



```

for(i=0; i<row; i++) {
    db_bufReal[i] = addressReal00 + i*col ;
    db_bufImag[i] = addressImag00 + i*col ;
}

double* db_vectorReal ;
double* db_vectorImag ;

db_vectorReal = new double [row*col] ;
db_vectorImag = new double [row*col] ;

int index ;

for(j=0; j<col; j++) {
    for(i=0; i<row; i++) {

        index = j*row + i ;
        db_vectorReal[index] = db_bufReal[i][j] ;
        db_vectorImag[index] = db_bufImag[i][j] ;

    }
}

memcpy(mxGetPr(mx_pointer), db_vectorReal, row*col*sizeof(double));
memcpy(mxGetPi(mx_pointer), db_vectorImag, row*col*sizeof(double));

delete[] db_vectorReal ;
delete[] db_vectorImag ;

delete[] db_bufReal ;
delete[] db_bufImag ;
}

/* ***** */
/* ***** */
/* ***** */
/* ***** */

```

```

/* II. Transfer values from mxArray to C/C++ double */

/* ***** */
/* 1a. transfer a real mxArray to a C/C++ double scalar */
double mxArray2double_scalarReal (mxArray* mx_pointer) {

    double db_scalar = mxGetScalar(mx_pointer) ;

    return db_scalar ;

}

/* ***** */
/* 1b. transfer a complex mxArray to C/C++ double scalars */
void mxArray2double_scalarComplex (mxArray* mx_pointer, double &db_scalarReal
                                   , double &db_scalarImag) {

    double* bufferReal ;
    bufferReal      = (double *)mxGetPr(mx_pointer) ;
    db_scalarReal   = bufferReal[0] ;

    double* bufferImag ;

    if( mxGetPi(mx_pointer)!= NULL ) {
        bufferImag      = (double *)mxGetPi(mx_pointer) ;
        db_scalarImag   = bufferImag[0] ;
    }

    else {
        db_scalarImag   = 0 ;
    }

}

/* ***** */
/* 2a. transfer a real mxArray to a C/C++ double vector */
void mxArray2double_vectorReal (mxArray* mx_pointer, double* cpp) {

    /* row vector has row =1 */
    int i ;

```

```

int row = mxGetM(mx_pointer) ; /* number of rows */
int col = mxGetN(mx_pointer) ; /* number of columns */

int vectorSize ;

if ( row > col )    { vectorSize = row ;}
else                { vectorSize = col ;}

double* buffer ;
buffer = mxGetPr(mx_pointer) ;

for (i=0; i<vectorSize; i++) {

    cpp[i] = buffer[i] ;

}

}

/* ***** */

/* 2b. transfer a complex mxArray to C/C++ double vectors */
void mxArray2double_vectorComplex (mxArray* mx_pointer, double* cppReal, double* cppImag)
{
    /* row vector has row =1 */
    int i ;
    int row = mxGetM(mx_pointer) ; /* number of rows */
    int col = mxGetN(mx_pointer) ; /* number of columns */

    int vectorSize ;

    if ( row > col )    { vectorSize = row ;}
    else                { vectorSize = col ;}

    double* bufferReal ;
    bufferReal = mxGetPr(mx_pointer) ;

    for (i=0; i<vectorSize; i++) {
        cppReal[i] = bufferReal[i] ;
    }
}

```

```

double* bufferImag ;
if( mxGetPi(mx_pointer)!= NULL ) {

    bufferImag = mxGetPi(mx_pointer) ;
    for (i=0; i<vectorSize; i++) {
        cppImag[i] = bufferImag[i] ;
    }
}
else {
    for (i=0; i<vectorSize; i++) {
        cppImag[i] = 0.0 ;
    }
}

}

/* ***** */
/* 3a. transfer a real mxArray to a C/C++ double matrix */
void mxArray2double_matrixReal (mxArray* mx_pointer, double** db_matrix) {

    int i, j, index ;
    int row = mxGetM(mx_pointer) ; /* number of rows */
    int col = mxGetN(mx_pointer) ; /* number of columns */

    double* buffer ;
    buffer = mxGetPr(mx_pointer) ;

    for(j=0; j<col; j++) {
        for(i=0; i<row; i++) {

            index = j*row + i ;
            db_matrix[i][j] = buffer[index] ;

        }
    }
}

```

```

}

/* ***** */

/* 3b. transfer a complex mxArray to C/C++ double matrixes */
void mxArray2double_matrixComplex (mxArray* mx_pointer, double** db_matrixReal
                                   , double** db_matrixImag) {

    int i, j, index ;
    int row = mxGetM(mx_pointer) ; /* number of rows */
    int col = mxGetN(mx_pointer) ; /* number of columns */

    double* bufferReal ;
    bufferReal = mxGetPr(mx_pointer) ;

    for(j=0; j<col; j++) {
        for(i=0; i<row; i++) {

            index = j*row + i ;
            db_matrixReal[i][j] = bufferReal[index] ;

        }
    }

    double* bufferImag ;
    if( mxGetPi(mx_pointer) != NULL ) {

        bufferImag = mxGetPi(mx_pointer) ;

        for(j=0; j<col; j++) {
            for(i=0; i<row; i++) {

                index = j*row + i ;
                db_matrixImag[i][j] = bufferImag[index] ;

            }
        }
    }
}

```

```

    }

}

else {

    for(j=0; j<col; j++) {
        for(i=0; i<row; i++) {
            db_matrixImag[i][j] = 0.0 ;

        }
    }

}

}

/* ***** */
/* ***** */
/* ***** */
void printMatrix(double** matrix, int row, int col) {

int i, j;

    for (i=0; i<row; i++) {
        for (j=0; j<col; j++) {
            cout << matrix[i][j] << "\t" ;

        }
        cout << endl ;
    }

}

/* ***** */

```

## 7.5 The Code of the Utility File mwUtilityCompilerVer4.h

```

/* mwUtilityCompilerVer4.h */

/* ***** */
/* III. Transfer values from C/C++ double to mxArray */
/* ***** */

/* 3a. Transfer a C/C++ double matrix to a real mxArray */
mwArray double2mwArray_matrixReal(double** db_matrix, int row, int col) {

    mxArray mw_matrix(row, col, mxDOUBLE_CLASS) ;

    for(int i=0; i<row; i++) {
        for(int j=0; j<col; j++) {

            mw_matrix(i+1, j+1) = db_matrix[i][j] ;

        }
    }

    return mw_matrix ;
}

/* ***** */
/* 3a. Transfer a C/C++ double matrix to a real mxArray */
mwArray double2mwArray_matrixReal(double* addressMatrix00, int row, int col) {

/*assign memories for a buffer */
    int i, j ;
    double **db_matrixbuf ;
    db_matrixbuf = new double*[row] ;

    for(i=0; i<row; i++) {
        db_matrixbuf[i] = new double [col] ;
    }

/* set address for rows */
    for(i=0; i<row; i++) {
        db_matrixbuf[i] = addressMatrix00 + i*col ;
    }
}

```





```

/*assign memories for buffers */
    int i, j ;
    double **db_bufReal ;
    double **db_bufImag ;

    db_bufReal = new double*[row] ;
    db_bufImag = new double*[row] ;

    for(i=0; i<row; i++) {
        db_bufReal[i] = new double [col] ;
        db_bufImag[i] = new double [col] ;
    }

/* set address for rows */
    for(i=0; i<row; i++) {
        db_bufReal[i] = addressReal00 + i*col ;
        db_bufImag[i] = addressImag00 + i*col ;
    }

/* transfer to mxArray */
    mxArray mw_matrix(row, col, mxDOUBLE_CLASS, mxCOMPLEX) ;

    for(i=0; i<row; i++) {
        for(j=0; j<col; j++) {

            mw_matrix(i+1, j+1).Real() = db_bufReal[i][j] ;
            mw_matrix(i+1, j+1).Imag() = db_bufImag[i][j] ;

        }
    }

    delete[] db_bufReal ;
    delete[] db_bufImag ;

    return mw_matrix ;
}

/* ***** */
/* ***** */

```

```

/* ***** */

/* IV. Transfer values from mxArray to C/C++ double */

/* 2a. Transfer a real mxArray to C/C++ double vector */
void mxArray2double_vectorReal (mxArray mw, double* cpp) {

    int i ;

    int vectorSize = mw.NumberOfElements() ;

    mxArray dim = mw.GetDimensions() ;
    int row = (int) dim(1,1) ;
    int col = (int) dim(1,2) ;

    /* row case */
    if ( row > col )    {
        for (i=0; i<vectorSize; i++) {
            cpp[i] = (double)mw(i+1, 1) ;
        }
    }

    /* col case */
    else    {
        for (i=0; i<vectorSize; i++) {
            cpp[i] = (double)mw(1, i+1) ;
        }
    }

}

/* ***** */

/* 2b. Transfer a complex mxArray to C/C++ double vectors */

void mxArray2double_vectorComplex(mxArray mw, double* cppReal, double *cppImag) {

    int i ;

    int vectorSize = mw.NumberOfElements() ;

```

```

mwArray dim = mw.GetDimensions() ;
int row = (int) dim(1,1) ;
int col = (int) dim(1,2) ;

/* row case */
if ( row > col )    {
    for (i=0; i<vectorSize; i++) {
        cppReal[i] = (double)mw(i+1, 1).Real() ;
        if( mw.IsComplex() ) { cppImag[i] = (double)mw(i+1, 1).Imag() ; }
        else {                  cppImag[i] = 0.0 ;                  }
    }

}

/* col case */
else    {
    for (i=0; i<vectorSize; i++) {
        cppReal[i] = (double)mw(1, i+1).Real() ;
        if( mw.IsComplex() ) { cppImag[i] = (double)mw(1, i+1).Imag() ; }
        else {                  cppImag[i] = 0.0 ;                  }

    }

}

}

/* ***** */
/* 3a. Transfer a real mxArray to a C/C++ double matrix */

void mxArray2double_matrixReal (mwArray mw_matrix, double** cpp) {

    int i,j ;

    mxArray dim = mw_matrix.GetDimensions() ;
    int row = (int) dim(1,1) ;
    int col = (int) dim(1,2) ;

```

```

    for (i=0; i<row; i++) {
        for (j=0; j<col; j++) {

            cpp[i][j] = (double)mw_matrix(i+1, j+1) ;

        }
    }

}

/* ***** */
/* 3b. Transfer a complex mxArray to C/C++ double matrixes */

void mxArray2double_matrixComplex (mxArray mw_matrix, double** cppReal, double** cppImag) {

    int i,j ;

    mxArray dim = mw_matrix.GetDimensions() ;
    int row = (int) dim(1,1) ;
    int col = (int) dim(1,2) ;

    for (i=0; i<row; i++) {
        for (j=0; j<col; j++) {

            cppReal[i][j] = (double)mw_matrix(i+1, j+1).Real() ;

            if( mw_matrix.IsComplex() ) { cppImag[i][j] = (double)mw_matrix(i+1, j+1).Imag() ; }
            else {
                cppImag[i][j] = 0.0 ; }

        }
    }

}

/* ***** */
/* ***** */
/* ***** */

```

```
void printMatrix(double** matrix, int row, int col) {  
  
    int i, j;  
  
    for (i=0; i<row; i++) {  
        for (j=0; j<col; j++) {  
            cout << matrix[i][j] << "\t" ;  
  
        }  
        cout << endl ;  
    }  
  
}
```



## Chapter 8

# Matrix Computations

In this chapter we'll generate a C shared library ***matrixcomputationslib*** and a C++ shared library ***cppmatrixcomputationslib*** from common M-files working on matrix computation problems. The generated functions of these libraries will be used in a MSVC .Net project to solve matrix computation problems.

Following are steps to create a C shared library `matrixcomputationslib.dll` and a C++ shared library `cppmatrixcomputationslib.dll` which will be used to solve matrix computation problems in the next sections.

We will write the M-files as shown below. These files are used to generate the C and C++ shared libraries. These files are:

`mydet.m`, `myinv.m`, `myminus.m`, `mymtimes.m`, `myplus.m`, and `mytranspose.m`

---

```
function y = mydet(a)
```

```
y = det(a) ;
```

---

```
function y = myinv(a)
```

```
y = inv(a) ;
```

---

```
function y = myminus(a, b)
```

```
y = a - b ;
```

---

```
function y = mymtimes(a, b)
```

```
y = a*b ;
```

---

```
function y = myplus(a, b)
```

```
y = a + b ;
```

---

```
function y = mytranspose( x )
```

```
y = x' ;
```

---

## A. FOR C SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C shared library

***matrixcomputationslib*** :

```
mcc -B csharedlib:matrixcomputationslib mydet.m myinv.m myminus.m  
mymtimes.m myplus.m mytranspose.m
```

2. From this command MATLAB Compiler 4 will create eight files for this C shared library:

```
matrixcomputationslib.c    matrixcomputationslib.ctf  
matrixcomputationslib.dll  matrixcomputationslib.exp  
matrixcomputationslib.exports  matrixcomputationslib.h  
matrixcomputationslib.lib  matrixcomputationslib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the common problems in the matrix computations (open the file `matrixcomputationslib.h` to see the names of these functions):

```
void mlfMydet      (int nargout, mxArray** y, mxArray* a);  
void mlfMyinv      (int nargout, mxArray** y, mxArray* a);  
void mlfMyminus    (int nargout, mxArray** y, mxArray* a, mxArray* b);
```



```

void mlfMymtimes    (int nargout, mxArray** y, mxArray* a, mxArray* b);
void mlfMyplus      (int nargout, mxArray** y, mxArray* a, mxArray* b);
void mlfMytranspose(int nargout, mxArray** y, mxArray* x);

```

## B. FOR C++ SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C++ shared library *cppmatrixcomputationslib* :

```

mcc -W cpplib:cppmatrixcomputationslib -T link:lib mydet.m myinv.m myminus.m mym-
times.m myplus.m mytranspose.m

```

2. From this command MATLAB Compiler 4 will create eight files for C++ shared library:

```

cppmatrixcomputationslib.cpp      cppmatrixcomputationslib.ctf
cppmatrixcomputationslib.dll      cppmatrixcomputationslib.exp
cppmatrixcomputationslib.exports  cppmatrixcomputationslib.h
cppmatrixcomputationslib.lib      cppmatrixcomputationslib_mcc_component_data.c

```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the common problems in the matrix computations (open the file *cppmatrixcomputationslib.h* to see the names of these functions):

```

void mydet    (int nargout, mxArray& y, const mxArray& a);
void myinv    (int nargout, mxArray& y, const mxArray& a);
void myminus  (int nargout, mxArray& y, const mxArray& a, const mxArray& b);
void mymtimes (int nargout, mxArray& y, const mxArray& a, const mxArray& b);
void myplus   (int nargout, mxArray& y, const mxArray& a, const mxArray& b);
void mytranspose (int nargout, mxArray& y, const mxArray& x);

```

## 8.1 Matrix Addition

### Problem 1

**input**     Matrix **A** and **B**

$$\mathbf{A} = \begin{bmatrix} 1.1 & 2.2 & 3.3 \\ 4.4 & 5.5 & 6.6 \\ 7.7 & 8.8 & 9.9 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 11 & 12 & 13 \\ 14 & 15 & 16 \\ 17 & 18 & 19 \end{bmatrix}$$

**output**     Finding the matrix addition  $\mathbf{C} = \mathbf{A} + \mathbf{B}$

### A. FOR C SHARED LIBRARY

The functions `mlfMyplus(..)` in the generated *matrixcomputationslib* library will be used in the following code to solve Problem 1 .

#### Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Matrix Computations" << endl;
    Test obj;

    cout << "Matrix addition" << endl ;
    obj.addMatrix() ;

    return 0 ;
}
```

---

```
/* Example.h */
```

```

#include <iostream.h>
#include "matrixcomputationslib.h"
#include "mxUtilityCompilerVer4.h"

class Test {

public:
    void addMatrix()          ;

    Test () {
        mclInitializeApplication(NULL,0);
        matrixcomputationslibInitialize();
    }

    ~Test () {
        matrixcomputationslibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */
void Test::addMatrix() {
    int i;
    double A[3][3] = {{ 1.1, 2.2, 3.3} , {4.4, 5.5, 6.6} , {7.7, 8.8, 9.9} } ;
    double B[3][3] = {{ 11 , 12 , 13 } , {14 , 15 , 16 } , {17 , 18 , 19 } } ;

    int row = 3 ;
    int col = 3 ;

    /* step 1 : declare mxArray variables */
    mxArray *mx_A = NULL ;
    mxArray *mx_B = NULL ;
    mxArray *mx_C = NULL ;

    /* step 2 : assign memory */
    mx_A = mxCreateDoubleMatrix(row, col, mxREAL) ;
    mx_B = mxCreateDoubleMatrix(row, col, mxREAL) ;
    mx_C = mxCreateDoubleMatrix(row, col, mxREAL) ;

```

```

/* step 3 : convert C/C++ matrix to mxArray */
double2mxArray_matrixReal(&A[0][0], mx_A) ;
double2mxArray_matrixReal(&B[0][0], mx_B) ;

/* step 4 : call an implemental function */
mlfMyplus(1, &mx_C, mx_A, mx_B );

/* step 5 : convert back to C/C++ double */
double **db_matrixC ;
db_matrixC = new double *[row] ;
for(i=0; i<row; i++) {
    db_matrixC[i] = new double [col] ;
}
mxArray2double_matrixReal(mx_C, db_matrixC) ;

/* step 6 : print out */
printMatrix(db_matrixC, row, col) ;

/* step 7 : free memories */
mxDestroyArray(mx_A) ;
mxDestroyArray(mx_B) ;
mxDestroyArray(mx_C) ;

delete [] db_matrixC ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The functions `myplus(..)` in the generated *cppmatrixcomputationslib* library will be used in the following code to solve Problem 1 .

Listing code

---

```

/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

```

```

int main() {

    cout << "Matrix Computations" << endl;
    Test obj;

    cout << endl ;
    cout << "Matrix addition" << endl ;
    obj.addMatrix() ;

    return 0 ;
}



---



/* Example.h */

#include <iostream.h>
#include "cppmatrixcomputationslib.h"
#include "mwUtilityCompilerVer4.h"

class Test {

public:
    void addMatrix()          ;

    Test () {
        mclInitializeApplication(NULL,0);
        cppmatrixcomputationslibInitialize();
    }

    ~Test () {
        cppmatrixcomputationslibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */

void Test::addMatrix() {
    int i;

    double db_A[3][3] = {{ 1.1, 2.2, 3.3} , {4.4, 5.5, 6.6} , {7.7, 8.8, 9.9} } ;

```

```

double db_B[3][3] = {{ 11 , 12 , 13 } , {14 , 15 , 16 } , {17 , 18 , 19 } } ;

int row = 3 ;
int col = 3 ;

/* convert C/C++ matrix to mxArray */
mwArray mw_A = double2mwArray_matrixReal(&db_A[0][0], row, col) ;
mwArray mw_B = double2mwArray_matrixReal(&db_B[0][0], row, col) ;

/* call an implemental function */
mwArray mw_C ;
myplus (1, mw_C, mw_A, mw_B);
std::cout << mw_C << std::endl ;

/* convert back to C/C++ double */
double **db_C ;
db_C = new double* [row] ;

    for (i=0; i<row; i++) {
        db_C[i] = new double [col] ;
    }

mwArray2double_matrixReal(mw_C, db_C) ;
/* print out */
printMatrix(db_C, row, col) ;

/* free memories */
delete[] db_C ;
}

```

---

end code

## 8.2 Matrix Subtraction

### A. FOR C SHARED LIBRARY

The functions `mlfMyminus(..)` in the generated ***matrixcomputations*** library will be used to handle the matrix subtraction. The code of using this function is identical to the above section, except at the step *call an implemental function*, we write:

```
mlfMyminus(1, &mx_C, mx_A, mx_B );
```

## B. FOR C++ SHARED LIBRARY

The functions `myminus(..)` in the generated *cppmatrixcomputations* library will be used to handle the matrix subtraction. The code of using this function is identical to the above section, except at the step *call an implemental function*, we write:

```
mlfMyminus(1, &mx_C, mx_A, mx_B );
```

## 8.3 Matrix Multiplication

### Problem 2

**input**      Matrix **A** and **B**

$$\mathbf{A} = \begin{bmatrix} 1.1 & 2.2 & 3.3 & 4.4 \\ 5.5 & 6.6 & 7.7 & 8.8 \\ 9.9 & 10.10 & 11.11 & 12.12 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 10 & 11 \\ 12 & 13 \\ 14 & 15 \\ 16 & 17 \end{bmatrix}$$

**output**      Finding the product matrix    **C = A \* B**

## A. FOR C SHARED LIBRARY

This following code describes how to use the functions `mlfMymtimes(..)` in the generated *matrixcomputationslib* library to calculate the matrix multiplication in Problem 2.

Listing code

---

```
void Test::multipleMatrix() {
    int i ;
    double A[3][4] = { { 1.1, 2.2 , 3.3 , 4.4 } ,\
                       { 5.5, 6.6 , 7.7 , 8.8 } ,\
                       { 9.9, 10.10, 11.11, 12.12 } } ;

    double B[4][2] = {{ 10, 11}, {12, 13}, {14, 15}, {16, 17} } ;

    int rowA = 3 ;
    int colA = 4 ;

    int rowB = 4 ;
    int colB = 2 ;
```

```

int rowC = rowA ;
int colC = colB ;

/* step 1 : declare mxArray variables */
mxArray *mx_A = NULL ;
mxArray *mx_B = NULL ;
mxArray *mx_C = NULL ;

/* step 2 : assign memory */
mx_A = mxCreateDoubleMatrix(rowA, colA, mxREAL) ;
mx_B = mxCreateDoubleMatrix(rowB, colB, mxREAL) ;
mx_C = mxCreateDoubleMatrix(rowC, colC, mxREAL) ;

/* step 3 : convert C/C++ matrix to mxArray */
double2mxArray_matrixReal(&A[0][0], mx_A) ;
double2mxArray_matrixReal(&B[0][0], mx_B) ;

/* step 4 : call an implemental function */
mlfMymtimes(1, &mx_C, mx_A, mx_B );

/* step 5 : convert back to C/C++ double */
double **db_matrixC ;
db_matrixC = new double *[rowC] ;
for(i=0; i<rowC; i++) {
    db_matrixC[i] = new double [colC] ;
}

mxArray2double_matrixReal(mx_C, db_matrixC) ;

/* step 6 : print out */
printMatrix(db_matrixC, rowC, colC) ;

/* step 7 : free memories */
mxDestroyArray(mx_A) ;
mxDestroyArray(mx_B) ;
mxDestroyArray(mx_C) ;
delete [] db_matrixC ;
}

```



## B. FOR C++ SHARED LIBRARY

This following code describes how to use the functions `mymtimes(..)` in the generated *cppmatrixcomputationslib* library to calculate the matrix multiplication in Problem 2.

Listing code

---

```
void Test::multipleMatrix() {
    int i ;

    double db_A[3][4] = { { 1.1, 2.2 , 3.3 , 4.4 } , \
                          { 5.5, 6.6 , 7.7 , 8.8 } , \
                          { 9.9, 10.10, 11.11, 12.12 } } ;

    double db_B[4][2] = {{ 10, 11}, {12, 13}, {14, 15}, {16, 17} } ;

    int rowA = 3 ;
    int colA = 4 ;

    int rowB = 4 ;
    int colB = 2 ;

    int rowC = rowA ;
    int colC = colB ;

    /* convert C/C++ matrix to mxArray */
    mxArray mw_A = double2mxArray_matrixReal(&db_A[0][0], rowA, colA) ;
    mxArray mw_B = double2mxArray_matrixReal(&db_B[0][0], rowB, colB) ;

    /* call an implemental function */
    mxArray mw_C ;
    mymtimes (1, mw_C, mw_A, mw_B);
    std::cout << mw_C << std::endl ;

    /* convert back to C/C++ double */
    double **db_C ;
    db_C = new double* [rowC] ;

    for (i=0; i<rowC; i++) {
        db_C[i] = new double [colC] ;
    }
}
```

```

mwArray2double_matrixReal(mw_C, db_C) ;
/* print out */
printMatrix(db_C, rowC, colC) ;

/* free memories */
delete[] db_C ;

}

```

---

end code

## 8.4 Matrix Determinant

### Problem 3

**input**      Matrix **A**

$$\mathbf{A} = \begin{bmatrix} 1.1 & 2.2 & 3.3 \\ 7.7 & 4.4 & 9.9 \\ 4.4 & 5.5 & 8.8 \end{bmatrix}$$

**output**      Finding the determinant of this matrix **A**

### A. FOR C SHARED LIBRARY

This following code describes how to use the functions `mlfMydet(..)` in the generated *matrixcomputationslib* library to find the matrix determinant in Problem 3.

#### Listing code

---

```

void Test::determinantMatrix() {

double A[3][3] = {{ 1.1, 2.2, 3.3}, {7.7, 4.4, 9.9} , {4.4, 5.5, 8.8} } ;

int row = 3 ;
int col = 3 ;

/* step 1 : declare mxArray variables */
mxArray *mx_A      = NULL ;
mxArray *mx_detA = NULL ;

```

```

/* step 2 : assign memory */
mx_A      = mxCreateDoubleMatrix(row, col, mxREAL)      ;
mx_detA   = mxCreateDoubleMatrix( 1 , 1 , mxREAL)      ;

/* step 3 : convert C/C++ matrix to mxArray */
double2mxArray_matrixReal(&A[0][0], mx_A) ;

/* step 4 : call an implemental function */
mlfMydet(1, &mx_detA, mx_A );

/* step 5 : convert back to C/C++ double */
double db_detA ;
db_detA = mxArray2double_scalarReal(mx_detA) ;

/* step 6 : print out */
    cout<< db_detA << endl  ;

/* step 7 : free memories */
mxDestroyArray(mx_A)      ;
mxDestroyArray(mx_detA)   ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

This following code describes how to use the functions `mydet(..)` in the generated *cppmatrixcomputationslib* library to find the matrix determinant in Problem 3.

Listing code

---

```

void Test::determinantMatrix() {

double db_A[3][3] = {{ 1.1, 2.2, 3.3}, {7.7, 4.4, 9.9} , {4.4, 5.5, 8.8} } ;

int row = 3 ;
int col = 3 ;

/* convert C/C++ matrix to mxArray */
mwArray mw_A = double2mwArray_matrixReal(&db_A[0][0], row, col) ;

```

```

/* call an implemental function */
mwArray mw_detA ;
mydet(1, mw_detA, mw_A);

/* convert back to C/C++ double */
double db_detA = (double) mw_detA(1,1) ;

/* print out */
cout << db_detA ;

}

```

---

end code

## 8.5 Inverse Matrix

### A. FOR C SHARED LIBRARY

This following code describes how to use the functions `mlfMyinv(..)` in the generated *matrixcomputationslib* library to find a matrix inversion.

Listing code

---

```

void Test::inverseMatrix() {

int i ;
double A[3][3] = {{ -1 , 1 , 2} , {3 , -1 , 1} , {-1 , 3 , 4} } ;

int row = 3 ;
int col = 3 ;

/* step 1 : declare mxArray variables */
mxArray *mx_A          = NULL ;
mxArray *mx_inverseA = NULL ;

/* step 2 : assign memory */
mx_A          = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_inverseA = mxCreateDoubleMatrix(row, col, mxREAL) ;

/* step 3 : convert C/C++ matrix to mxArray */
double2mxArray_matrixReal(&A[0][0], mx_A) ;

```

```

/* step 4 : call an implemental function */
mlfMyinv(1, &mx_inverseA, mx_A );

/* step 5 : convert back to C/C++ double */
double **db_inverseA;
db_inverseA = new double *[row] ;
for(i=0; i<row; i++) {
    db_inverseA[i] = new double [col] ;
}

mxArray2double_matrixReal(mx_inverseA, db_inverseA) ;

/* step 6 : print out */
printMatrix(db_inverseA, row, col) ;

/* step 7 : free memories */
mxDestroyArray(mx_A)          ;
mxDestroyArray(mx_inverseA)   ;

delete [] db_inverseA ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

This following code describes how to use the functions `myinv(..)` in the generated *cppmatrixcomputationslib* library to find a matrix inversion.

Listing code

---

```

void Test::inverseMatrix() {

    int i ;
    double db_A[3][3] = {{ -1 , 1 , 2} , {3 , -1 , 1} , {-1 , 3 , 4} } ;

    int row = 3 ;
    int col = 3 ;

    /* convert C/C++ matrix to mxArray */

```

```

mwArray mw_A = double2mwArray_matrixReal(&db_A[0][0], row, col) ;
/* call an implemental function */
mwArray mw_invA ;
myinv(1, mw_invA, mw_A);

/* convert back to C/C++ double */
double **db_invA ;
db_invA = new double* [row] ;
    for (i=0; i<row; i++) {
        db_invA[i] = new double [col] ;
    }

mwArray2double_matrixReal(mw_invA, db_invA) ;

/* print out */
printMatrix(db_invA, row, col) ;

/* free memories */
delete[] db_invA ;

}

```

---

end code

## 8.6 Transpose Matrix

### A. FOR C SHARED LIBRARY

The functions `mlfMytranspose(..)` in the generated *matrixcomputationslib* library will be used to find the transpose matrix. The code of using this function is identical to the section **Inverse Matrix**, except at the step *call an implemental function*, we write:

```

/* call an implemental function */
mlfMytranspose(1, &mx_transposeA, mx_A ) ;

```

### B. FOR C++ SHARED LIBRARY

The functions `mytranspose(..)` in the generated *cppmatrixcomputationslib* library will be used to find the transpose matrix. The code of using this function is identical to the section **Inverse Matrix**, except at the step *call an implemental function*, we write:

```

mytranspose(1, mw_transposeA, mw_A);

```

## 8.7 Assigning Directly Values for a Matrix

In the above sections, we used a method to assign a matrix to an mxArray or mxArray for using in the generated functions. When the matrix declares in the double-pointer, we can assign directly values of the matrix by using the same name function double2mxArray\_matrixReal(..) or double2mxArray\_matrixReal(..). The follow code implements these transfers.

### A. FOR C SHARED LIBRARY

Listing code

---

```
void Test::addMatrixOtherMethod( ) {
    int i ;

    double **db_matrixA ;
    double **db_matrixB ;

    /* step 1 : assign memories for buffers */
    int row = 3 ;
    int col = 3 ;

    db_matrixA = new double*[row] ;
    db_matrixB = new double*[row] ;

    for(i=0; i<row; i++) {
        db_matrixA[i] = new double [col] ;
        db_matrixB[i] = new double [col] ;
    }

    db_matrixA[0][0] = 1.1 ;
    db_matrixA[0][1] = 2.2 ;
    db_matrixA[0][2] = 3.3 ;

    db_matrixA[1][0] = 4.4 ;
    db_matrixA[1][1] = 5.5 ;
    db_matrixA[1][2] = 6.6 ;

    db_matrixA[2][0] = 7.7 ;
    db_matrixA[2][1] = 8.8 ;
    db_matrixA[2][2] = 9.9 ;
```

```

db_matrixB[0][0] = 11 ;
db_matrixB[0][1] = 12 ;
db_matrixB[0][2] = 13 ;

db_matrixB[1][0] = 14 ;
db_matrixB[1][1] = 15 ;
db_matrixB[1][2] = 16 ;

db_matrixB[2][0] = 17 ;
db_matrixB[2][1] = 18 ;
db_matrixB[2][2] = 19 ;

/* step 2 : declare mxArray variables */
mxArray *mx_A = NULL ;
mxArray *mx_B = NULL ;
mxArray *mx_C = NULL ;

/* step 3 : assign memory */
mx_A = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_B = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_C = mxCreateDoubleMatrix(row, col, mxREAL) ;

/* step 4 : convert C/C++ matrix to mxArray */
double2mxArray_matrixReal(db_matrixA, mx_A) ;
double2mxArray_matrixReal(db_matrixB, mx_B) ;

/* step 5 : call an implemental function */
mlfMyplus(1, &mx_C, mx_A, mx_B );

/* step 6 : convert back to C/C++ double */
double **db_matrixC ;
db_matrixC = new double *[row] ;
for(i=0; i<row; i++) {
    db_matrixC[i] = new double [col] ;
}

mxArray2double_matrixReal(mx_C, db_matrixC) ;

/* setp 7 : print out */

```



```

printMatrix(db_matrixC, row, col) ;

/* step 8 : free memories */
mxDestroyArray(mx_A)      ;
mxDestroyArray(mx_B)      ;
mxDestroyArray(mx_C)      ;

delete [] db_matrixA ;
delete [] db_matrixB ;
delete [] db_matrixC ;
}

```

---

end code

## B. FOR C++ SHARED LIBRARY

Listing code

---

```

/* ***** */
void Test::addMatrixOtherMethod( ) {
    int i ;

    double **db_matrixA ;
    double **db_matrixB ;

    int row = 3 ;
    int col = 3 ;

    db_matrixA = new double*[row] ;
    db_matrixB = new double*[row] ;

    for(i=0; i<row; i++) {
        db_matrixA[i] = new double [col] ;
        db_matrixB[i] = new double [col] ;
    }

    db_matrixA[0][0] = 1.1 ;
    db_matrixA[0][1] = 2.2 ;
    db_matrixA[0][2] = 3.3 ;

```

```

db_matrixA[1][0] = 4.4 ;
db_matrixA[1][1] = 5.5 ;
db_matrixA[1][2] = 6.6 ;

db_matrixA[2][0] = 7.7 ;
db_matrixA[2][1] = 8.8 ;
db_matrixA[2][2] = 9.9 ;

db_matrixB[0][0] = 11 ;
db_matrixB[0][1] = 12 ;
db_matrixB[0][2] = 13 ;

db_matrixB[1][0] = 14 ;
db_matrixB[1][1] = 15 ;
db_matrixB[1][2] = 16 ;

db_matrixB[2][0] = 17 ;
db_matrixB[2][1] = 18 ;
db_matrixB[2][2] = 19 ;

/* convert C/C++ matrix to mxArray */
mwArray mw_A = double2mwArray_matrixReal(db_matrixA, row, col) ;
mwArray mw_B = double2mwArray_matrixReal(db_matrixB, row, col) ;

/* call an implemental function */
mwArray mw_C ;
myplus (1, mw_C, mw_A, mw_B);

/* convert back to C/C++ double */
double **db_C ;
db_C = new double* [row] ;
    for (i=0; i<row; i++) {
        db_C[i] = new double [col] ;
    }

mwArray2double_matrixReal(mw_C, db_C) ;

/* print out */
printMatrix(db_C, row, col) ;

```

```

/* free memories */
delete[] db_C ;

}

```

---

end code

## 8.8 Assigning Values for a Matrix from a File

We can also assign values for a matrix from a file. The following is the code that get values for the matrix from a data file matrixA.dat.

### A. FOR C SHARED LIBRARY

Listing code

---

```

void Test::transposeMatrixOtherMethod() {

/* matrixA.dat
1.1  2.2  3.3
4.4  5.5  6.6
7.7  8.8  9.9
*/

int i,j ;

/* step 1 : declare matrix */
double **db_matrixA ;
int row = 3 ;
int col = 3 ;

/* step 2 : assign memory */
db_matrixA = new double*[row] ;

for(i=0; i<row; i++) {
    db_matrixA[i] = new double [col] ;
}

/* step 3 : assign values for matrix */

```

112

```
ifstream f ;
f.open("matrixA.dat", ios::in | ios::nocreate );
if(!f) {
    f.close() ;
    delete [] db_matrixA ;

    cout << "You don't have a file matrixA.dat" << endl ;
    return ;
}

/* read the file */
for (i=0; i<row; i++) {
    for (j=0; j<col; j++) {

        f>> db_matrixA[i][j] ;
    }
}
f.close() ;

/* step 4 : declare mxArray variables */
mxArray *mx_A          = NULL ;
mxArray *mx_transposeA = NULL ;

/* step 5 : assign memory */
mx_A          = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_transposeA = mxCreateDoubleMatrix(row, col, mxREAL) ;

/* step 6 : convert C/C++ matrix to mxArray */
double2mxArray_matrixReal(db_matrixA, mx_A) ;

/* step 7 : call an implemental function */
mlfMytranspose(1, &mx_transposeA, mx_A );

/* step 8 : convert back to C/C++ double */
double **db_transposeA;
db_transposeA = new double *[row] ;
for(i=0; i<row; i++) {
    db_transposeA[i] = new double [col] ;
}
```

```

mxArray2double_matrixReal(mx_transposeA, db_transposeA) ;

/* setp 9 : print out */
printMatrix(db_transposeA, row, col) ;

/* step 10 : free memories */
mxDestroyArray(mx_A) ;
mxDestroyArray(mx_transposeA) ;

delete [] db_matrixA ;
delete [] db_transposeA ;
}

```

---

end code

## B. FOR C++ SHARED LIBRARY

Listing code

---

```

void Test::transposeMatrixOtherMethod() {
/* matrixA.dat
1.1  2.2  3.3
4.4  5.5  6.6
7.7  8.8  9.9
*/

int i,j ;
/* step 1 : declare matrix */
double **db_matrixA ;
int row = 3 ;
int col = 3 ;

/* step 2 : assign memory */
db_matrixA = new double*[row] ;

for(i=0; i<row; i++) {
    db_matrixA[i] = new double [col] ;
}

/* step 3 : assign values for matrix */

```

114

```
ifstream f ;
f.open("matrixA.dat", ios::in | ios::nocreate );
if(!f) {
    f.close() ;
    delete [] db_matrixA ;

    cout << "You don't have a file matrixA.dat" << endl ;
    return ;
}

/* read the file */
for (i=0; i<row; i++) {
    for (j=0; j<col; j++) {

        f>> db_matrixA[i][j] ;
    }
}
f.close() ;

/* convert C/C++ matrix to mxArray */
mwArray mw_A = double2mwArray_matrixReal(db_matrixA, row, col) ;

/* call an implemental function */
mwArray mw_transposeA ;
mytranspose(1, mw_transposeA, mw_A);

/* convert back to C/C++ double */
double **db_transposeA ;
db_transposeA = new double* [row] ;
    for (i=0; i<row; i++) {
        db_transposeA[i] = new double [col] ;
    }
mwArray2double_matrixReal(mw_transposeA, db_transposeA) ;

/* print out */
printMatrix(db_transposeA, row, col) ;
/* free memories */
delete [] db_matrixA ;
delete [] db_transposeA ;
}
```

## Chapter 9

# Linear System Equations

The problem of linear system equations involves solving the equation  $\mathbf{Ax} = \mathbf{b}$ . This chapter focuses on linear system equations in which the matrix  $\mathbf{A}$  is a square matrix or a sparse matrix,  $\mathbf{A} \in \mathbb{R}^{n \times n}$ ,  $\mathbf{x} \in \mathbb{R}^n$ , and  $\mathbf{b} \in \mathbb{R}^n$ .

In this chapter we'll generate a C shared library *linearsytemlib* and C++ shared library *cpplinearsytemlib* from common M-files working on problems of linear system equations. The generated functions of these libraries will be used in MSVC .Net project to solve the linear system equations.

Following are steps to create a C shared library linearsytemlib.dll and a C++ shared library cpplinearsytemlib.dll which will be used to solve problems in the next sections.

We will write the M-files as shown below. These files will be used to generate the C and C++ shared libraries.

mydiag.m, myfull.m, mylu.m, mymldivide.m, mymrdivide.m,  
mysparse.m, and myspdiags.m

---

```
function X = mydiag(v,k)
```

```
X = diag(v,k) ;
```

---

```
function B = myextractmatrix(A, rowa, rowb, cola, colb)
```

```
B = A(rowa:rowb, cola:colb) ;
```

```
% extract from row a to row b, and from col a to col b
```

---

```
function A = myfull(S)
```

```
A = full(S) ;
```

---

```
function [L,U,P] = mylu(A)
```

```
[L,U,P] = lu(A) ;
```

---

```
function x = mymldivide(A, b)
```

```
%solve equation Ax = b
```

```
x = A\b ;
```

---

```
function x = mymrdivide(A, b)
```

```
%solve equation xA = b ==>
```

```
x = A/b ;
```

---

```
function S = mysparse(A)
```

```
S = sparse(A) ;
```

---

```
function A = myspdiags(B,d,m,n)
```

```
A = spdiags(B,d,m,n)
```

---

## A. FOR C SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C shared library

***linearsytemlib*** :

```
mcc -B csharedlib:linearsystemlib mydiag.m myextractmatrix.m myfull.m mylu.m
```

```
mymldivide.m mymrdivide.m mysparse.m myspdiags.m
```

2. MATLAB Compiler 4 will create eight files:

```
linearsystemlib.c      linearsystemlib.ctf      linearsystemlib.dll
linearsystemlib.exp    linearsystemlib.exports  linearsystemlib.h
linearsystemlib.lib    linearsystemlib_mcc_component_data.c
```



Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the common problems in the linear system equations (open the file `linearsystemlib.h` to see the names of these functions):

```
void mlfMydiag      (int nargout, mxArray** X, mxArray* v, mxArray* k);
void mlfMyextractmatrix(int nargout, mxArray** B, mxArray* A, mxArray* rowa,
                        mxArray* rowb, mxArray* cola, mxArray* colb);

void mlfMyfull      (int nargout, mxArray** A, mxArray* S);
void mlfMyLU (int nargout, mxArray** L, mxArray** U, mxArray** P, mxArray* A);
void mlfMyLdivide(int nargout, mxArray** x, mxArray* A, mxArray* b);
void mlfMyMdivide(int nargout, mxArray** x, mxArray* A, mxArray* b);
void mlfMySparse (int nargout, mxArray** S, mxArray* A);
void mlfMySpdiags (int nargout, mxArray** A, mxArray* B, mxArray* d,
                  mxArray* m, mxArray* n);
```

## B. FOR C++ SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C++ shared library *cpplinearsystemlib* :  
`mcc -W cpplib:cpplinearsystemlib -T link:lib mydiag.m myextractmatrix.m myfull.m myLU.m mymldivide.m mymdivide.m mysparse.m myspdiags.m`
2. MATLAB Compiler 4 will create eight files:

```
cpplinearsystemlib.cpp    cpplinearsystemlib.ctf        cpplinearsystemlib.dll
cpplinearsystemlib.exp    cpplinearsystemlib.exports    cpplinearsystemlib.h
cpplinearsystemlib.lib    cpplinearsystemlib_mcc_component_data.c
```

3. In the following sections, we'll use the following implemental functions in this library to solve the common problems in the linear system equations (open the file `cpplinearsystemlib.h` to see the names of these functions):

```
void mydiag(int nargout, mxArray& X
            , const mxArray& v, const mxArray& k);

void myextractmatrix(int nargout, mxArray& B, const mxArray& A
                    , const mxArray& rowa, const mxArray& rowb
                    , const mxArray& cola, const mxArray& colb);
```

```

void myfull(int nargout, mxArray& A, const mxArray& S);
void mylu (int nargout, mxArray& L, mxArray& U
           , mxArray& P, const mxArray& A);
void mymldivide(int nargout, mxArray& x
                , const mxArray& A, const mxArray& b);
void mymrdivide(int nargout, mxArray& x
                , const mxArray& A, const mxArray& b);
void mysparse (int nargout, mxArray& S, const mxArray& A);
void myspsdiags (int nargout, mxArray& A, const mxArray& B
                 , const mxArray& d, const mxArray& m, const mxArray& n);

```

## 9.1 Linear System Equations

In general, the form of linear system equations (size  $n \times n$ ) is:

$$\begin{aligned}
 a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
 a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
 a_{31}x_1 + a_{32}x_2 + \cdots + a_{3n}x_n &= b_3 \\
 \cdots &\quad \cdots \\
 a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
 \end{aligned} \tag{9.1}$$

### Problem 1

**input**      Matrix **A** and vector **b**

$$\mathbf{A} = \begin{bmatrix} 1.1 & 5.6 & 3.3 \\ 4.4 & 12.3 & 6.6 \\ 7.7 & 8.8 & 9.9 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 12.5 \\ 32.2 \\ 45.6 \end{bmatrix}$$

**output**    . Finding the solution **x** of linear system equations, **Ax = b**  
              . Finding the lower **L** and upper **U** of the matrix **A**

### A. FOR C SHARED LIBRARY

The functions `mlfMymldivide(..)` and `mlfMyLU(..)` in the generated *linearsystemlib* library will

be used in the following code to solve Problem 1.

#### Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Linear System Equations" ;
    cout << endl ;

    Test obj;
    obj.LinearSystemEquations() ;
    obj.LU_decompression()      ;

    return 0 ;
}
```

---

```
/* Example.h */

#include <iostream.h>
#include "linearsystemlib.h"
#include "mxUtilityCompilerVer4.h"

class Test {

public:
    void LinearSystemEquations() ;
    void LU_decompression()      ;

    Test () {
        mclInitializeApplication(NULL,0);
        linearsystemlibInitialize();
    }

    ~Test () {
```

```

        linearsystemlibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */
void Test::LinearSystemEquations() {

    int i ;

    /* Solve general linear system equations  $Ax = b$  */

    double db_A[3][3]      = { {1.1,  5.6, 3.3} ,\
                                {4.4, 12.3, 6.6} ,\
                                {7.7,  8.8, 9.9} };

    double db_vectorb[3] = { 12.5, 32.2 , 45.6 } ;

    int row = 3 ;
    int col = 3 ;

    /* step 1 : declare mxArray variables */
    mxArray *mx_A = NULL ;
    mxArray *mx_b = NULL ;
    mxArray *mx_x = NULL ;

    /* step 2 : assign memory */
    mx_A = mxCreateDoubleMatrix(row, col, mxREAL) ;
    mx_b = mxCreateDoubleMatrix(row, 1 , mxREAL) ;
    mx_x = mxCreateDoubleMatrix(row, 1 , mxREAL) ;

    /* note: we create mx_b and mx_x are column vectors */

    /* step 3 : convert C/C++ double to mxArray */
    double2mxArray_matrixReal(&db_A[0][0], mx_A) ;
    double2mxArray_vectorReal(db_vectorb, mx_b) ;

    /* step 4 : call an implemental function */
    mlfMymldivide(1, &mx_x, mx_A, mx_b);

    /* step 5 : convert back to C/C++ double */

```

```

double *db_vectorx = new double[col] ;
mxArray2double_vectorReal(mx_x, db_vectorx) ;

/* setp 9 : print out */
for(i=0; i<col; i++) {
    cout<< *(db_vectorx + i) << endl ;
}

/* step 7 : free memories */
mxDestroyArray(mx_A) ;
mxDestroyArray(mx_b) ;
mxDestroyArray(mx_x) ;

delete [] db_vectorx ;

}

/* ***** */
void Test::LU_decompression () {

/* find lower and upper matrixes */

double db_A[3][3] = { {1.1, 5.6, 3.3} ,\
                      {4.4, 12.3, 6.6} ,\
                      {7.7, 8.8, 9.9} };

int row = 3 ;
int col = 3 ;
int i ;

/* step 1 : declare mxArray variables */
mxArray *mx_A = NULL ;
mxArray *mx_L = NULL ;
mxArray *mx_U = NULL ;
mxArray *mx_P = NULL ;

/* step 2 : assign memory */
mx_A = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_L = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_U = mxCreateDoubleMatrix(row, col, mxREAL) ;

```

```

mx_P = mxCreateDoubleMatrix(row, col, mxREAL)    ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_matrixReal(&db_A[0][0], mx_A) ;

/* step 4 : call an implemental function */
mlfMyLU(3, &mx_L, &mx_U, &mx_P, mx_A);

/* step 5 : convert back to C/C++ double */
double **db_L = new double *[row] ;
double **db_U = new double *[row] ;

for(i=0; i<row; i++) {
    db_L[i] = new double [col] ;
    db_U[i] = new double [col] ;
}

mxArray2double_matrixReal(mx_L, db_L) ;
mxArray2double_matrixReal(mx_U, db_U) ;

/* setp 9 : print out */
cout << endl << "The lower matrix:" << endl ;
printMatrix(db_L, row, col) ;

cout << endl << "The upper matrix:" << endl ;
printMatrix(db_U, row, col) ;

/* step 7 : free memories */
mxDestroyArray(mx_A)      ;
mxDestroyArray(mx_U)      ;
mxDestroyArray(mx_L)      ;
mxDestroyArray(mx_P)      ;

delete [] db_L ;
delete [] db_U ;

}

```

## B. FOR C++ SHARED LIBRARY

The functions `mymldivide(..)` and `mylu(..)` in the generated *cpplinearsystemlib* library will be used in the following code to solve Problem 1.

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Linear System Equations" ;
    cout << endl ;

    Test obj;
    obj.LinearSystemEquations() ;

    obj.LU_decompression() ;
    return 0 ;
}
```

---

```
/* Example.h */

#include <iostream.h>
#include "cpplinearsystemlib.h"
#include "mwUtilityCompilerVer4.h"

class Test {

public:

    void LinearSystemEquations() ;
    void LU_decompression () ;

    Test () {
        mclInitializeApplication(NULL,0);
```

```

    cpplinearsystemlibInitialize();
}

~Test () {
    cpplinearsystemlibTerminate();
    mclTerminateApplication();
}

};

/* ***** */
void Test::LinearSystemEquations() {

    int i ;

    /* Solve general linear system equations  $Ax = b$  */

    double db_A[3][3]      = { {1.1,  5.6, 3.3} ,\
                                {4.4, 12.3, 6.6} ,\
                                {7.7,  8.8, 9.9} };

    double db_vectorb[3] = { 12.5, 32.2 , 45.6 } ;

    int row = 3 ;
    int col = 3 ;

    /* convert C/C++ matrix to mxArray */
    mxArray mw_A = double2mxArray_matrixReal(&db_A[0][0], row, col) ;
    mxArray mw_vectorb(row, 1, mxDOUBLE_CLASS) ;
    mw_vectorb.SetData(db_vectorb, row) ;

    /* call an implemental function */
    mxArray mw_x(row, 1, mxDOUBLE_CLASS) ;
    mymldivide(1, mw_x, mw_A, mw_vectorb) ;

    /* convert back to C/C++ double */
    double *db_vectorx = new double[col] ;
    mxArray2double_vectorReal(mw_x, db_vectorx) ;

```



```

/* print out */
cout << "Solution x: " << endl ;
for(i=0; i<col; i++) {
    cout<< db_vectorx[i] << endl ;
}

/* free memories */
delete [] db_vectorx ;

}

/* ***** */
void Test::LU_decompression () {

/* find lower and upper matrixes */

double db_A[3][3] = { {1.1, 5.6, 3.3} ,\
                      {4.4, 12.3, 6.6} ,\
                      {7.7, 8.8, 9.9} };

int row = 3 ;
int col = 3 ;
int i ;

/* convert C/C++ double to mxArray */
mwArray mw_A = double2mwArray_matrixReal(&db_A[0][0], row, col) ;

/* call an implemental function */
mwArray mw_L (row, col, mxDOUBLE_CLASS) ;
mwArray mw_U (row, col, mxDOUBLE_CLASS) ;
mwArray mw_P (row, col, mxDOUBLE_CLASS) ;

mylu(3, mw_L, mw_U, mw_P, mw_A);

/* convert back to C/C++ double */
double **db_L = new double *[row] ;
double **db_U = new double *[row] ;

for(i=0; i<row; i++) {
    db_L[i] = new double [col] ;
    db_U[i] = new double [col] ;
}

```

```

}

mwArray2double_matrixReal(mw_L, db_L) ;
mwArray2double_matrixReal(mw_U, db_U) ;

/* print out */
cout << endl << "The lower matrix:" << endl ;
printMatrix(db_L, row, col) ;

cout << endl << "The upper matrix:" << endl ;
printMatrix(db_U, row, col) ;

/* free memories */
delete [] db_L ;
delete [] db_U ;
}

```

---

end code

The code of the file mxUtilityCompilerVer4.h is in Chapter 7.

## 9.2 Sparse Linear System

The sparse linear system is a common system created to solve a technical problem. In this system the main matrix is a sparse matrix (a matrix that has numbers where the nonzero is minor). To obtain an accurate solution and a better computational simulation in the sparse system, MATLAB provided specified functions to handle this task.

### Problem 2

**input**      Sparse matrix **A** and vector **b**

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1.1 \\ 0 & 2.2 & 0 & 0 & 0 \\ 3.3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6.6 & 0 \\ 0 & 0 & 5.5 & 0 & 0 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 11.1 \\ 0 \\ 22.2 \\ 0 \\ 33.3 \end{bmatrix}$$

**output**      Finding the solution **x** of the sparse system equations **Ax = b**

## A. FOR C SHARED LIBRARY

The common steps to solve a sparse linear system using functions in the C shared library *matrixcomputations* are:

1. Establishing the spare matrix by using the function `mlfMysparse(..)`.
2. Solving the sparse system by using the function `mlfMymldivide(..)`.

The following is the code to solve Problem 2 by using the functions, `mlfMysparse(..)` and `mlfMymldivide(..)`

Listing code

---

```
void Test::sparseSystem() {
/*
A =   0      0      0      0      1.1
      0      2.2    0      0      0
      3.3    0      0      0      0
      0      0      0      6.6    0
      0      0      5.5    0      0

b =   11.1  0      22.2  0      33.3
*/

/* Solve general linear system equations Ax = b */
double db_A[5][5] = { {0 ,    0 ,    0 ,    0 ,    1.1 } ,\
                      {0 ,    2.2,    0 ,    0 ,    0 } ,\
                      {3.3,    0 ,    0 ,    0 ,    0 } ,\
                      {0 ,    0 ,    0 ,    6.6,    0 } ,\
                      {0 ,    0 ,    5.5,    0 ,    0 } } ;

double db_vectorb[5] = { 11.1, 0, 22.2, 0, 33.3 } ;

int row = 5 ;
int col = 5 ;
int i ;

/* step 1 : declare mxArray variables */
mxArray *mx_A      = NULL ;
mxArray *mx_b      = NULL ;
```

```

mxArray *mx_x      = NULL ;

/* step 2 : assign memory */
mx_A      = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_b      = mxCreateDoubleMatrix(row, 1, mxREAL) ;
mx_x      = mxCreateDoubleMatrix(row, 1, mxREAL) ;

/* note: we create mx_b and mx_x are column vectors */

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_matrixReal(&db_A[0][0], mx_A) ;
double2mxArray_vectorReal(db_vectorb, mx_b) ;

/* step 4 : call an implemental function */
mlfMysparse(1, &mx_A, mx_A);
mlfMysparse(1, &mx_b, mx_b);

mlfMymldivide(1, &mx_x, mx_A, mx_b);
mlfMyfull    (1, &mx_x, mx_x);

/* step 5 : convert back to C/C++ double */
double *db_vectorx = new double[col] ;

mxArray2double_vectorReal(mx_x, db_vectorx) ;

/* setp 6 : print out */
cout << endl ;
cout << "Solution x :" << endl ;
for(i=0; i<col; i++) {
    cout<< *(db_vectorx + i) << endl ;
}

/* step 7 : free memories */
mxDestroyArray(mx_A) ;
mxDestroyArray(mx_b) ;
mxDestroyArray(mx_x) ;

delete [] db_vectorx ;
}

```

## B. FOR C++ SHARED LIBRARY

The common steps to solve a sparse linear system using functions in the C++ shared library *cppmatrixcomputations* are:

1. Establishing the spare matrix by using the function `myparse(..)`.
2. Solving the sparse system by using the function `mymldivide(..)`.

The following is the code to solve Problem 2 by using the functions, `myparse(..)` and `mymldivide(..)`

Listing code

---

```
void Test::sparseSystem() {
/*
A =   0      0      0      0      1.1
      0      2.2    0      0      0
      3.3    0      0      0      0
      0      0      0      6.6    0
      0      0      5.5    0      0

b =   11.1  0      22.2  0      33.3
*/

/* Solve general linear system equations Ax = b */
double db_A[5][5] = { {0 ,    0 ,    0 ,    0 ,    1.1 } ,\
                      {0 ,    2.2,    0 ,    0 ,    0 } ,\
                      {3.3,    0 ,    0 ,    0 ,    0 } ,\
                      {0 ,    0 ,    0 ,    6.6,    0 } ,\
                      {0 ,    0 ,    5.5,    0 ,    0 } } ;

double db_vectorb[5] = { 11.1, 0, 22.2, 0, 33.3 } ;

int row = 5 ;
int col = 5 ;
int i ;

/* note: we create mw_b and mw_x are column vectors */
/* convert C/C++ double to mxArray */
mwArray mw_A = double2mwArray_matrixReal(&db_A[0][0], row, col) ;
```

```

mwArray mw_b(row, 1, mxDOUBLE_CLASS) ;
mw_b.SetData(db_vectorb, row) ;

/* call an implemental function */
mysparse(1, mw_A, mw_A);
mysparse(1, mw_b, mw_b);

mwArray mw_x(row, 1, mxDOUBLE_CLASS) ;
mymldivide(1, mw_x, mw_A, mw_b);
myfull(1, mw_x, mw_x);

/* convert back to C/C++ double */
double *db_vectorx = new double[col] ;

mwArray2double_vectorReal(mw_x, db_vectorx) ;

/* print out */
cout << endl ;
cout << "Solution x :" << endl ;
for(i=0; i<col; i++) {
    cout<< db_vectorx[i] << endl ;
}

/* free memories */
delete [] db_vectorx ;

}

```

---

end code

### 9.3 Tridiagonal System Equations

This section focuses on finding the solution of tridiagonal linear system equations  $\mathbf{Ax} = \mathbf{d}$ , as follows:

$$\begin{bmatrix} a_1 & b_1 & 0 & \cdots & 0 \\ c_2 & a_2 & b_2 & \cdots & 0 \\ 0 & c_3 & a_3 & b_3 & \cdots & 0 \\ \cdots & 0 & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \cdots & 0 & 0 & c_n & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \cdots \\ d_{n-1} \\ d_n \end{bmatrix} \quad (9.2)$$

#### Problem 3

**input** Matrix **B** includes column vectors **c**, **a**, and **b**.  
Vector right-hand side **d**

$$\mathbf{B} = \begin{bmatrix} c_1 & a_1 & b_1 \\ c_2 & a_2 & b_2 \\ c_3 & a_3 & b_3 \\ c_4 & a_4 & b_4 \\ c_5 & a_5 & b_5 \\ c_6 & a_6 & b_6 \end{bmatrix} = \begin{bmatrix} 1.1 & 4.1 & 2.1 \\ 1.2 & 4.2 & 2.2 \\ 1.3 & 4.3 & 2.3 \\ 1.4 & 4.4 & 2.4 \\ 1.5 & 4.5 & 2.5 \\ 1.6 & 4.6 & 2.6 \end{bmatrix}, \quad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \end{bmatrix} = \begin{bmatrix} 1.2 \\ 4.5 \\ 5.6 \\ 12.4 \\ 7.8 \\ 6.8 \end{bmatrix} \quad (9.3)$$

**output** Finding the solution **x** of tridiagonal system equations (Eq. 9.2)

#### A. FOR C SHARED LIBRARY

The steps to solve Problem 3 are:

1. Establishing a buffer matrix **bufferA** (in Eq. 9.4) from given matrix **B** (in Eq. 9.3) by using the function in the library *linearsytemlib*:

```
void mlfMyspdiaqs (int nargout, mxArray** bufferA, mxArray* B, mxArray* d,
                  mxArray* m, mxArray* n);
```

This function `mlfMyspdiaags(..)` creates an m-by-n sparse matrix **bufferA** by taking the columns of B and placing them along the diagonals specified by d as follows:

$$\mathbf{bufferA} = \begin{bmatrix} c_1 & a_1 & b_1 & 0 & & \cdots & 0 & 0 \\ 0 & c_2 & a_2 & b_2 & & \cdots & 0 & 0 \\ 0 & 0 & c_3 & a_3 & b_3 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & c_{n-1} & a_{n-1} & b_{n-1} & 0 \\ 0 & 0 & \cdots & 0 & 0 & c_n & a_n & b_n \end{bmatrix} \quad (9.4)$$

2. Obtaining the matrix A as in Eq. 9.2 by extracting from the matrix **bufferA**
3. Using the functions in the library *linearsytemlib* to solve the tridiagonal linear system equations.

The following is the code to solve Problem 3 by using the functions in the library *linearsytemlib*.

Listing code

---

```
void Test::tridiagonalSystem() {

double B[6][3] ;

B[0][0] = 1.1 ;
B[1][0] = 1.2 ;
B[2][0] = 1.3 ;
B[3][0] = 1.4 ;
B[4][0] = 1.5 ;
B[5][0] = 1.6 ;

/* columns 2 */
B[0][1] = 4.1 ;
B[1][1] = 4.2 ;
B[2][1] = 4.3 ;
B[3][1] = 4.4 ;
B[4][1] = 4.5 ;
B[5][1] = 4.6 ;

/* columns 3 */
B[0][2] = 2.1 ;
B[1][2] = 2.2 ;
```



```

B[2][2] = 2.3 ;
B[3][2] = 2.4 ;
B[4][2] = 2.5 ;
B[5][2] = 2.6 ;

double db_vectord [6] ;
db_vectord[0] = 1.2 ;
db_vectord[1] = 4.5 ;
db_vectord[2] = 5.6 ;
db_vectord[3] = 12.4 ;
db_vectord[4] = 7.8 ;
db_vectord[5] = 6.8 ;

int i ;
int one = 1 ;
int two = 2 ;
int seven = 7 ;

int row = 6 ;
int col = 6 ;

int band = 3 ; /* tridiagonal */
int m = row ;
int n = col + (band-1) ;

double d[3] = {0, 1, 2} ; /* start from 0 */

int rowB = row ;
int colB = band ;

/* step 1 : declare mxArray variables */
mxArray *mx_B = NULL ;
mxArray *mx_bufferA = NULL ;
mxArray *mx_A = NULL ;

mxArray *mx_d = NULL ;
mxArray *mx_m = NULL ;
mxArray *mx_n = NULL ;

mxArray *mx_one = NULL ;

```

```

mxArray *mx_two      = NULL ;
mxArray *mx_seven    = NULL ;
mxArray *mx_row      = NULL ;

mxArray *mx_vectord = NULL ;
mxArray *mx_x        = NULL ;

/* step 2 : assign memory */
mx_B      = mxCreateDoubleMatrix( rowB , colB , mxREAL) ;
mx_bufferA = mxCreateDoubleMatrix( m   , n   , mxREAL) ;
mx_A      = mxCreateDoubleMatrix( row  , col , mxREAL) ;

mx_d      = mxCreateDoubleMatrix( 1 , band , mxREAL) ;
mx_m      = mxCreateDoubleMatrix( 1 , 1   , mxREAL) ;
mx_n      = mxCreateDoubleMatrix( 1 , 1   , mxREAL) ;

mx_one    = mxCreateDoubleMatrix( 1 , 1 , mxREAL) ;
mx_two    = mxCreateDoubleMatrix( 1 , 1 , mxREAL) ;
mx_seven  = mxCreateDoubleMatrix( 1 , 1 , mxREAL) ;
mx_row    = mxCreateDoubleMatrix( 1 , 1 , mxREAL) ;

mx_vectord = mxCreateDoubleMatrix(row, 1 , mxREAL) ;
mx_x       = mxCreateDoubleMatrix(row, 1 , mxREAL) ;

/* note: we create mx_vectord and mx_x are column vectors */

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal(one , mx_one ) ;
double2mxArray_scalarReal(two , mx_two ) ;
double2mxArray_scalarReal(seven , mx_seven ) ;
double2mxArray_scalarReal(row , mx_row ) ;

double2mxArray_scalarReal(m, mx_m) ;
double2mxArray_scalarReal(n, mx_n) ;

double2mxArray_vectorReal(d , mx_d ) ;
double2mxArray_matrixReal(&B[0][0] , mx_B) ;

double2mxArray_vectorReal(db_vectord, mx_vectord) ;

```

```

/* step 4 : call an implemental function */

/* create a sparse matrix from column-matrix B */
mlfMyspdiaqs (1, &mx_bufferA, mx_B, mx_d , mx_m, mx_n);
mlfMyfull(1, &mx_bufferA, mx_bufferA) ;

/* plot to see */
double **db_bufferA = new double *[m] ;
    for(i=0; i<m; i++) {
        db_bufferA[i] = new double [n] ;

    }

mxArray2double_matrixReal(mx_bufferA, db_bufferA) ;

cout << endl << "The buffer matrix A:" << endl ;
printMatrix(db_bufferA, m, n) ;

/* extract the need-matrix from the buffter matrix,
   from row 1 to row 6 and from column 2 to column 7 */
mlfMyextractmatrix(1, &mx_A, mx_bufferA, mx_one, mx_row, mx_two, mx_seven);

/* plot to see */
double **db_A = new double *[row] ;
    for(i=0; i<row; i++) {
        db_A[i] = new double [col] ;

    }

mxArray2double_matrixReal(mx_A, db_A) ;

cout << endl << "The need-matrix A:" << endl ;
printMatrix(db_A, row, col) ;

/* solve the tridiagnal system equations */
mlfMymldivide(1, &mx_x, mx_A, mx_vectord);

```

```

/* step 5 : convert back to C/C++ double */
double *db_vectorx = new double[col] ;
mxArray2double_vectorReal(mx_x, db_vectorx) ;

/* step 6 : print out */
cout << "Tridiagonal system solution:" << endl ;
for(i=0; i<row; i++) {
    cout<< *(db_vectorx + i) << endl ;
}

/* step 7 : free memories */

mxDestroyArray(mx_B      ) ;
mxDestroyArray(mx_bufferA ) ;
mxDestroyArray(mx_A      ) ;

mxDestroyArray(mx_d) ;
mxDestroyArray(mx_m) ;
mxDestroyArray(mx_n) ;

mxDestroyArray(mx_one  ) ;
mxDestroyArray(mx_two  ) ;
mxDestroyArray(mx_seven ) ;
mxDestroyArray(mx_row  ) ;

mxDestroyArray(mx_vectord) ;
mxDestroyArray(mx_x      ) ;

delete [] db_bufferA ;
delete [] db_A ;
delete [] db_vectorx ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The steps to solve Problem 3 are:

1. Establishing a buffer matrix **bufferA** (in Eq. 9.5) from given matrix **B** (in Eq. 9.3) by using the function in the library *cpplinearsytemlib*:

```
void myspdiags (int nargout, mxArray& A, const mxArray& B
               , const mxArray& d, const mxArray& m, const mxArray& n);
```

This function myspdiags(..) creates an m-by-n sparse matrix **bufferA** by taking the columns of B and placing them along the diagonals specified by d as follows:

$$\mathbf{bufferA} = \begin{bmatrix} c_1 & a_1 & b_1 & 0 & & \cdots & 0 & 0 \\ 0 & c_2 & a_2 & b_2 & & \cdots & 0 & 0 \\ 0 & 0 & c_3 & a_3 & b_3 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & c_{n-1} & a_{n-1} & b_{n-1} & 0 \\ 0 & 0 & \cdots & 0 & 0 & c_n & a_n & b_n \end{bmatrix} \quad (9.5)$$

2. Obtaining the matrix A as in Eq. 9.2 by extracting from the matrix **bufferA**
3. Using the functions in the library *cpplinearsytemlib* to solve the tridiagonal linear system equations.

The following is the code to solve Problem 3 by using the functions in the library *cpplinearsytemlib*.

Listing code

---

```
void Test::tridiagonalSystem() {

double B[6][3] ;
B[0][0] = 1.1 ;
B[1][0] = 1.2 ;
B[2][0] = 1.3 ;
B[3][0] = 1.4 ;
B[4][0] = 1.5 ;
B[5][0] = 1.6 ;

/* columns 2 */
```

```

B[0][1] = 4.1 ;
B[1][1] = 4.2 ;
B[2][1] = 4.3 ;
B[3][1] = 4.4 ;
B[4][1] = 4.5 ;
B[5][1] = 4.6 ;

/* columns 3 */
B[0][2] = 2.1 ;
B[1][2] = 2.2 ;
B[2][2] = 2.3 ;
B[3][2] = 2.4 ;
B[4][2] = 2.5 ;
B[5][2] = 2.6 ;

double db_vectord [6] ;
db_vectord[0] = 1.2 ;
db_vectord[1] = 4.5 ;
db_vectord[2] = 5.6 ;
db_vectord[3] = 12.4 ;
db_vectord[4] = 7.8 ;
db_vectord[5] = 6.8 ;

int i ;

int row = 6 ;
int col = 6 ;

int band = 3 ; /* tridiagonal */
int m = row ;
int n = col + (band-1) ;

double d[3] = {0, 1, 2} ; /* start from 0 */

int rowB = row ;
int colB = band ;

/* note: we create mx_vectord and mx_x are column vectors */

```

```

/* convert C/C++ double to mxArray */
mwArray mw_B = double2mwArray_matrixReal(&B[0][0], rowB, colB) ;

mwArray mw_vectord(row, 1, mxDOUBLE_CLASS) ;
mw_vectord.SetData(db_vectord, row) ;

mwArray mw_d(band, 1, mxDOUBLE_CLASS) ;
mw_d.SetData(d, band) ;

/* call an implemental function */

/* create a sparse matrix, size mxn, from column-matrix B */
mwArray mw_m(1, 1, mxDOUBLE_CLASS) ;
mw_m = m ;

mwArray mw_n(1, 1, mxDOUBLE_CLASS) ;
mw_n = n ;

mwArray mw_bufferA(m, n, mxDOUBLE_CLASS) ;
myspdiags (1, mw_bufferA, mw_B, mw_d , mw_m, mw_n);

myfull(1, mw_bufferA, mw_bufferA) ;

/* plot to see */
cout << endl << "The buffer matrix A:" << endl ;
std::cout << mw_bufferA << std::endl ;

/* extract the need-matrix A from the buffter matrix,
   from row 1 to row 6 and from column 2 to column 7 */
mwArray mw_A(row, col, mxDOUBLE_CLASS) ;

mwArray mw_one(1, 1, mxDOUBLE_CLASS) ;
mw_one = 1;

mwArray mw_row(1, 1, mxDOUBLE_CLASS) ;
mw_row = row;

mwArray mw_two(1, 1, mxDOUBLE_CLASS) ;
mw_two = 2;

```

```

mwArray mw_seven(1, 1, mxDOUBLE_CLASS) ;
mw_seven = 7;

myextractmatrix(1, mw_A, mw_bufferA, mw_one, mw_row, mw_two, mw_seven);

/* plot to see */
cout << endl << "The need-matrix A:" << endl ;
std::cout << mw_A << std::endl ;

/* solve the tridiagonal system equations */
mwArray mw_x(row, 1, mxDOUBLE_CLASS) ;
mymldivide(1, mw_x, mw_A, mw_vectord);

/* convert back to C/C++ double */
double *db_vectorx = new double[row] ;
mwArray2double_vectorReal(mw_x, db_vectorx) ;

/* print out */
cout << "Tridiagonal system solution:" << endl ;
for(i=0; i<row; i++) {
    cout<< db_vectorx[i] << endl ;
}

/* free memories */
delete [] db_vectorx ;

}

```

---

end code

## 9.4 Band Diagonal System Equations

The band diagonal system is a common system in engineering applications. The band diagonal matrix is a matrix with nonzero elements existing only along a few diagonal lines adjacent to the main diagonal (above and below). This section is a study of finding the solution of band diagonal system equations where the *width* = 4. This system is  $\mathbf{Ax} = \mathbf{d}$  as follows:



$$\begin{bmatrix} a_1 & b_1 & e_1 & 0 & & \cdots & 0 \\ c_2 & a_2 & b_2 & e_2 & 0 & \cdots & 0 \\ 0 & c_3 & a_3 & b_3 & e_3 & \cdots & 0 \\ \cdots & \cdots & \cdots & & & \cdots & \cdots \\ 0 & \cdots & 0 & c_{n-2} & a_{n-2} & b_{n-2} & e_{n-2} \\ 0 & \cdots & 0 & 0 & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \cdots & 0 & 0 & 0 & c_n & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \cdots \\ d_{n-2} \\ d_{n-1} \\ d_n \end{bmatrix} \quad (9.6)$$

The procedure to solve band diagonal system equations is similar to the procedure to solve tridiagonal system equations.

**Problem 4**

**input**      Matrix **B** includes columns **c**, **a**, **b**, and **e**.  
Vector **d**

$$\mathbf{B} = \begin{bmatrix} c_1 & a_1 & b_1 & e_1 \\ c_2 & a_2 & b_2 & e_2 \\ c_3 & a_3 & b_3 & e_3 \\ c_4 & a_4 & b_4 & e_4 \\ c_5 & a_5 & b_5 & e_5 \\ c_6 & a_6 & b_6 & e_6 \end{bmatrix} = \begin{bmatrix} 1.1 & 4.1 & 2.1 & 7.1 \\ 1.2 & 4.2 & 2.2 & 7.2 \\ 1.3 & 4.3 & 2.3 & 7.3 \\ 1.4 & 4.4 & 2.4 & 7.4 \\ 1.5 & 4.5 & 2.5 & 7.5 \\ 1.6 & 4.6 & 2.6 & 7.6 \end{bmatrix}, \quad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \end{bmatrix} = \begin{bmatrix} 1.2 \\ 4.5 \\ 5.6 \\ 12.4 \\ 7.8 \\ 6.8 \end{bmatrix} \quad (9.7)$$

**output**      Finding the solution **x** of the band system equations 9.6

The steps to solve this problem are similar to the steps in the tridiagonal problem. These steps are:

1. Establishing a buffer matrix **bufferA** (in Eq. 9.8) from given matrix **B** (in Eq. 9.7) by using a function in the library.

$$\mathbf{bufferA} = \begin{bmatrix} c_1 & a_1 & b_1 & e_1 & 0 & & \cdots & 0 & 0 & 0 \\ 0 & c_2 & a_2 & b_2 & e_2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & c_3 & a_3 & b_3 & e_3 & \cdots & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & & & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & c_{n-2} & a_{n-2} & b_{n-2} & e_{n-2} & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & c_{n-1} & a_{n-1} & b_{n-1} & e_{n-1} & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & c_n & a_n & b_n & e_n \end{bmatrix} \quad (9.8)$$

2. Obtaining the matrix A as in Eq. 9.6 by extracting from the matrix **bufferA**.
3. Using the functions in the library to solve the band system diagonal equations.

## A. FOR C SHARED LIBRARY

The following is the code to solve Problem 4 by using the functions in the library *linearsytemlib*.

Listing code

---

```
void Test::bandMatrixSystem() {

double B[6][4] ;

/* columns 1 */
B[0][0] = 1.1 ;
B[1][0] = 1.2 ;
B[2][0] = 1.3 ;
B[3][0] = 1.4 ;
B[4][0] = 1.5 ;
B[5][0] = 1.6 ;

/* columns 2 */
B[0][1] = 4.1 ;
B[1][1] = 4.2 ;
B[2][1] = 4.3 ;
B[3][1] = 4.4 ;
B[4][1] = 4.5 ;
B[5][1] = 4.6 ;

/* columns 3 */
B[0][2] = 2.1 ;
B[1][2] = 2.2 ;
B[2][2] = 2.3 ;
B[3][2] = 2.4 ;
B[4][2] = 2.5 ;
B[5][2] = 2.6 ;

/* columns 4 */
B[0][3] = 7.1 ;
B[1][3] = 7.2 ;
B[2][3] = 7.3 ;
B[3][3] = 7.4 ;
```

```

B[4][3] = 7.5 ;
B[5][3] = 7.6 ;

double db_vectord [6] ;
db_vectord[0] = 1.2 ;
db_vectord[1] = 4.5 ;
db_vectord[2] = 5.6 ;
db_vectord[3] = 12.4 ;
db_vectord[4] = 7.8 ;
db_vectord[5] = 6.8 ;

int i ;
int one = 1 ;
int two = 2 ;
int seven = 7 ;

int row = 6 ;
int col = 6 ;

int band = 4 ; /* band width */
int m = row ;
int n = col + (band-1) ;

double d[4] = {0, 1, 2, 3} ; /* start from 0 */

int rowB = row ;
int colB = band ;

/* step 1 : declare mxArray variables */
mxArray *mx_B = NULL ;
mxArray *mx_bufferA = NULL ;
mxArray *mx_A = NULL ;

mxArray *mx_d = NULL ;
mxArray *mx_m = NULL ;
mxArray *mx_n = NULL ;

mxArray *mx_one = NULL ;
mxArray *mx_two = NULL ;
mxArray *mx_seven = NULL ;

```

```

mxArray *mx_row      = NULL ;

mxArray *mx_vectord = NULL ;
mxArray *mx_x        = NULL ;

/* step 2 : assign memory */
mx_B      = mxCreateDoubleMatrix( rowB , colB , mxREAL) ;
mx_bufferA = mxCreateDoubleMatrix( m   , n   , mxREAL) ;
mx_A      = mxCreateDoubleMatrix( row  , col , mxREAL) ;

mx_d      = mxCreateDoubleMatrix( 1 , band , mxREAL) ;
mx_m      = mxCreateDoubleMatrix( 1 , 1   , mxREAL) ;
mx_n      = mxCreateDoubleMatrix( 1 , 1   , mxREAL) ;

mx_one     = mxCreateDoubleMatrix( 1 , 1 , mxREAL) ;
mx_two     = mxCreateDoubleMatrix( 1 , 1 , mxREAL) ;
mx_seven   = mxCreateDoubleMatrix( 1 , 1 , mxREAL) ;
mx_row     = mxCreateDoubleMatrix( 1 , 1 , mxREAL) ;

mx_vectord = mxCreateDoubleMatrix(row, 1 , mxREAL) ;
mx_x       = mxCreateDoubleMatrix(row, 1 , mxREAL) ;

/* note: we create mx_vectord and mx_x as column vectors */

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal(one , mx_one ) ;
double2mxArray_scalarReal(two , mx_two ) ;
double2mxArray_scalarReal(seven , mx_seven ) ;
double2mxArray_scalarReal(row , mx_row ) ;

double2mxArray_scalarReal(m, mx_m) ;
double2mxArray_scalarReal(n, mx_n) ;

double2mxArray_vectorReal(d , mx_d ) ;
double2mxArray_matrixReal(&B[0][0] , mx_B ) ;

double2mxArray_vectorReal(db_vectord, mx_vectord) ;

/* step 4 : call an implemental function */

```

```

/* create a sparse matrix, size mxn, from column-matrix B */
mlfMyspdiags (1, &mx_bufferA, mx_B, mx_d , mx_m, mx_n);
mlfMyfull(1, &mx_bufferA, mx_bufferA) ;

/* plot to see */
double **db_bufferA = new double *[m] ;
    for(i=0; i<m; i++) {
        db_bufferA[i] = new double [n] ;

    }

    mxArray2double_matrixReal(mx_bufferA, db_bufferA) ;

cout << endl << "The buffer band-matrix A:" << endl ;
printMatrix(db_bufferA, m, n) ;

/* extract the need-matrix A from the buffter matrix,
    from row 1 to row 6 and from column 2 to column 7 */
mlfMyextractmatrix(1, &mx_A, mx_bufferA, mx_one, mx_row, mx_two, mx_seven);

/* plot to see */
double **db_A = new double *[row] ;
    for(i=0; i<row; i++) {
        db_A[i] = new double [col] ;

    }

    mxArray2double_matrixReal(mx_A, db_A) ;

cout << endl << "The band-need-matrix A:" << endl ;
printMatrix(db_A, row, col) ;

/* solve the tridiagonal system equations */
mlfMymldivide(1, &mx_x, mx_A, mx_vectord);

/* step 5 : convert back to C/C++ double */
double *db_vectorx = new double[col] ;

```

```

mxArray2double_vectorReal(mx_x, db_vectorx) ;

/* step 6 : print out */
cout << "Band matrix system solution:" << endl ;
for(i=0; i<row; i++) {
    cout<< *(db_vectorx + i) << endl ;
}

/* step 7 : free memories */

mxDestroyArray(mx_B      ) ;
mxDestroyArray(mx_bufferA ) ;
mxDestroyArray(mx_A      ) ;

mxDestroyArray(mx_d) ;
mxDestroyArray(mx_m) ;
mxDestroyArray(mx_n) ;

mxDestroyArray(mx_one  ) ;
mxDestroyArray(mx_two  ) ;
mxDestroyArray(mx_seven ) ;
mxDestroyArray(mx_row  ) ;

mxDestroyArray(mx_vectord) ;
mxDestroyArray(mx_x      ) ;

delete [] db_bufferA ;
delete [] db_A      ;
delete [] db_vectorx ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 4 by using the functions in the library

*cpplinarsytemlib*.

Listing code

---

```
void Test::bandMatrixSystem() {
```

```
double B[6][4] ;
```

```
/* columns 1 */
```

```
B[0][0] = 1.1 ;
```

```
B[1][0] = 1.2 ;
```

```
B[2][0] = 1.3 ;
```

```
B[3][0] = 1.4 ;
```

```
B[4][0] = 1.5 ;
```

```
B[5][0] = 1.6 ;
```

```
/* columns 2 */
```

```
B[0][1] = 4.1 ;
```

```
B[1][1] = 4.2 ;
```

```
B[2][1] = 4.3 ;
```

```
B[3][1] = 4.4 ;
```

```
B[4][1] = 4.5 ;
```

```
B[5][1] = 4.6 ;
```

```
/* columns 3 */
```

```
B[0][2] = 2.1 ;
```

```
B[1][2] = 2.2 ;
```

```
B[2][2] = 2.3 ;
```

```
B[3][2] = 2.4 ;
```

```
B[4][2] = 2.5 ;
```

```
B[5][2] = 2.6 ;
```

```
/* columns 4 */
```

```
B[0][3] = 7.1 ;
```

```
B[1][3] = 7.2 ;
```

```
B[2][3] = 7.3 ;
```

```
B[3][3] = 7.4 ;
```

```
B[4][3] = 7.5 ;
```

```

B[5][3] = 7.6 ;

double db_vectord [6] ;
db_vectord[0] = 1.2 ;
db_vectord[1] = 4.5 ;
db_vectord[2] = 5.6 ;
db_vectord[3] = 12.4 ;
db_vectord[4] = 7.8 ;
db_vectord[5] = 6.8 ;

int i ;

int row = 6 ;
int col = 6 ;

int band = 4 ; /* band width */
int m = row ;
int n = col + (band-1) ;

double d[4] = {0, 1, 2, 3} ; /* start from 0 */

int rowB = row ;
int colB = band ;

/* note: we create mw_vectord and mw_x as column vectors */
mwArray mw_B = double2mwArray_matrixReal(&B[0][0], rowB, colB) ;

mwArray mw_vectord(row, 1, mxDOUBLE_CLASS) ;
mw_vectord.SetData(db_vectord, row) ;

mwArray mw_d(band, 1, mxDOUBLE_CLASS) ;
mw_d.SetData(d, band) ;

/* call an implemental function */

/* create a sparse matrix, size mxn, from column-matrix B */
mwArray mw_m(1, 1, mxDOUBLE_CLASS) ;
mw_m = m ;

mwArray mw_n(1, 1, mxDOUBLE_CLASS) ;

```



```

mw_n = n ;

mwArray mw_bufferA(m, n, mxDOUBLE_CLASS) ;
myspdiags (1, mw_bufferA, mw_B, mw_d , mw_m, mw_n);

myfull(1, mw_bufferA, mw_bufferA) ;

/* plot to see */
cout << endl << "The buffer band-matrix A:" << endl ;
std::cout << mw_bufferA << std::endl ;

/* extract the need-matrix A from the buffter matrix,
   from row 1 to row 6 and from column 2 to column 7 */
mwArray mw_A(row, col, mxDOUBLE_CLASS) ;

mwArray mw_one(1, 1, mxDOUBLE_CLASS) ;
mw_one = 1;

mwArray mw_row(1, 1, mxDOUBLE_CLASS) ;
mw_row = row;

mwArray mw_two(1, 1, mxDOUBLE_CLASS) ;
mw_two = 2;

mwArray mw_seven(1, 1, mxDOUBLE_CLASS) ;
mw_seven = 7;

myextractmatrix(1, mw_A, mw_bufferA, mw_one, mw_row, mw_two, mw_seven);

/* plot to see */
cout << endl << "The band-need-matrix A:" << endl ;
std::cout << mw_A << std::endl ;

/* solve the tridiagnal system equations */
mwArray mw_x(row, 1, mxDOUBLE_CLASS) ;
mymldivide(1, mw_x, mw_A, mw_vectord);

/* convert back to C/C++ double */
double *db_vectorx = new double[row] ;

```

```
mwArray2double_vectorReal(mw_x, db_vectorx) ;
```

```
/* print out */
cout << "Band matrix system solution:" << endl ;
for(i=0; i<row; i++) {
    cout<< db_vectorx[i] << endl ;
}
```

```
/* free memories */
delete [] db_vectorx ;

}
```

---

end code

### Note:

1. If all elements in each column of the matrix **B** are equal (for example, one case shown in Eq.9.9), you can create directly the matrix **A** without through the buffer matrix **bufferA** as follows:

#### **A. FOR C SHARED LIBRARY**

```
double d[3] = {-1, 0, 1} ;
mlfMyspdiags (1, &mx_A, mx_B, mx_d , mx_row, mx_row);
```

#### **B. FOR C++ SHARED LIBRARY**

```
double d[3] = {-1, 0, 1} ;
myspdiags (1, mw_A, mw_B, mw_d , mw_row, mw_row);
```

$$\mathbf{B} = \begin{bmatrix} c_1 & a_1 & b_1 \\ c_2 & a_2 & b_2 \\ c_3 & a_3 & b_3 \\ c_4 & a_4 & b_4 \\ c_5 & a_5 & b_5 \\ c_6 & a_6 & b_6 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 1 \\ 1 & 4 & 1 \\ 1 & 4 & 1 \\ 1 & 4 & 1 \\ 1 & 4 & 1 \\ 1 & 4 & 1 \end{bmatrix} \quad (9.9)$$

2. The matrix **A** can also be created by using the MATLAB function `diags(..)` [5].

## Chapter 10

# Ordinary Differential Equations

In this chapter we'll generate a C shared library *odelib* and a C++ shared library *cppodelib* from common M-files working on problems of ordinary differential equations (ODE). The generated functions of these libraries will be used in MSVC .Net project to solve common ODE problems.

The major MATLAB function of M-files used to generate the libraries to solve ODE problems is `ode45(..)`. There are another functions, *ode23(..)*, *ode113(..)*, *ode15s(..)*, and *ode23s(..)*, can be used to solve the ODE problems. Therefore you can choose a function with an options that satisfies your problem requirements. For more information on these functions, refer to the manual [7].

Following are steps to create a C shared library `odelib.dll` and a C++ shared library `cppodelib.dll` which will be used to solve ODE problems in the next sections.

We will write the M-files as shown below. These functions will be used to generate the C and C++ shared libraries.

```
myode45firstorder.m, yourfunc.m  
myode45secondorder.m, yoursecondfunc.m
```

---

```
function [t, y, lengthtime] = myode45firstorder(strfunc, tspan, y0)
```

```
[t, y] = ode45(@yourfunc, tspan, y0, [], strfunc) ;  
lengthtime = length(t) ;
```

---

```
function dydt = yourfunc(t, y, strfunc)
```

```
%trick for a function with/without t, y
strfunction = strcat(strfunc, '+ 0*t + 0*y') ;
```

```
F = inline(strfunction) ;
dydt = feval(F, t, y) ;
```

---

```
function [t, y, lengthtime] = myode45secondorder(strfunc, tspan, y0)
```

```
[t,y] = ode45(@yoursecondfunc, tspan, y0, [], strfunc) ;
lengthtime = length(t) ;
```

---

```
function dy = yoursecondfunc(t, y, strfunc)
```

```
% example:
```

```
%  $y'' - 2y' - 6y = \cos(3t)$ 
```

```
%  $y'' = \cos(3t) + 2y' + 6y$ 
```

```
% write an expression string with replace y' by yprime:
```

```
%       $\cos(3*t) + 2*yprime + 6*y$ 
```

```
f0 = inline('yy') ;
```

```
% it creates a function f(x)=x , as f0(yy) = yy
```

```
dy(1,:) = feval( f0, y(2) ) ;
```

```
%trick for a function with/without t, yprime, y
```

```
strfunction = strcat(strfunc, '+ 0*t + 0*yprime + 0*y') ;
```

```
f1 = inline(strfunction) ;
```

```
dy(2,:) = feval( f1, t , y(1), y(2) ) ;
```

## A. FOR C SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to generate the C shared library *odelib*:

```
mcc -B csharedlib:odelib myode45firstorder.m myode45secondorder.m
```

2. MATLAB Compiler 4.0 will create eight files for this C shared library:

```
odelib.c      odelib.ctf      odelib.dll
odelib.exp    odelib.exports  odelib.h
odelib.lib    odelib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the common ODE problems (open the file *odelib.h* to see the names of these functions):

```
void mlfMyode45firstorder(int nargout, mxArray** t, mxArray** y
                          , mxArray** lengthtime, mxArray* strfunc
                          , mxArray* tspan, mxArray* y0);

void mlfMyode45secondorder(int nargout, mxArray** t, mxArray** y
                           , mxArray** lengthtime, mxArray* strfunc
                           , mxArray* tspan, mxArray* y0);
```

## B. FOR C++ SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to generate the C shared library *cppodelib*:

```
mcc -W cpplib:cppodelib -T link:lib myode45firstorder.m myode45secondorder.m
```

2. MATLAB Compiler 4.0 will create eight files for this C shared library:

```
cppodelib.cpp  cppodelib.ctf      cppodelib.dll
cppodelib.exp  cppodelib.exports  cppodelib.h
cppodelib.lib  cppodelib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the common ODE problems (open the file `cppodelib.h` to see the names of these functions):

```
void myode45firstorder(int nargout, mxArray& t, mxArray& y
                        , mxArray& lengthtime, const mxArray& strfunc
                        , const mxArray& tspan, const mxArray& y0);

void myode45secondorder(int nargout, mxArray& t, mxArray& y
                        , mxArray& lengthtime, const mxArray& strfunc
                        , const mxArray& tspan, const mxArray& y0);
```

## 10.1 First Order ODE

**Problem 1** Find the function,  $y(t)$ , from the ODE function:

$$\frac{dy}{dt} = \cos(t)$$

with initial condition :

$$y_0 = 2.2 \quad \text{at} \quad t_0 = 0.2$$

**Note:**

- In solving first order ODE problems, the MATLAB function `ode45(..)` has an input argument that is an interval `tspan=[a,b]`, and the function outputs are two arrays:
  - array `t[ ]` contains the values of time  $t$ ,  $t \in [a, b]$
  - array `y[ ]` contains the values of the function  $y(t)$

The beginning of the interval is given,  $a = t_0$ . The end of the interval,  $b$ , is chosen by the user to show the time range in the problem.
- The time step is set to the default if you do not provide a time step.
- The time step can be set by providing `tspan=[t0, t2, ..., tn]` as a vector including the values of time. The output value  $y$  will be a column vector. Each row in the solution array  $y$  corresponds to the time in the column vector `tspan`.
- In the following code, we'll call the implement function `mlfMyode45firstorder(..)` or `myode45firstorder(..)` twice. First time, the function is called to obtain the size of the vector  $t$ . Second time, the function is called to get values of  $t$  and  $y$ .

5. The ODE function is passed as an expression string to the generated function `mlfMyode45firstorder(..)` or `myode45firstorder(..)` which is a function-function and has an argument as an expression string. The form of this expression string follows the rule of a MATLAB expression string.

## A. FOR C SHARED LIBRARY

The following is the code to solve Problem 1 by using the function `mlfMyode45firstorder(..)` in the generated *odelib* library with the time step set to default.

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "ODE problems." << endl;

    Test obj ;
    cout << "First order ODE." << endl ;
    obj.FirstOrder() ;

    return 0 ;
}
```

---

```
/* Example.h */

#include <iostream.h>
#include "odelib.h"
#include "mxUtilityCompilerVer4.h"

class Test {

public:
```

```

void FirstOrder() ;

    Test () {
        mclInitializeApplication(NULL,0);
        odelibInitialize();
    }

    ~Test () {
        odelibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */
void Test::FirstOrder() {

    /* Calculating first order ODE */
    const char strfunc[] = "cos(t)" ;
    //const char strfunc[] = "2+y" ;

    double db_y0= 2.2 ; /* initial condition at t0      */

    double db_tspan[2] ;
    db_tspan[0] = 0.2 ; /* begin interval t0 = 0.2      */
    db_tspan[1] = 6.5 ; /* end interval, we choose this */

    int i ;

    /* step 1 : declare mxArray variables */
    mxArray *mx_strfunc ;
    mxArray *mx_y0      = NULL ;
    mxArray *mx_tspan   = NULL ;

    mxArray *mx_length  = NULL ;
    mxArray *mx_t       = NULL ;
    mxArray *mx_y       = NULL ;

    mxArray *mx_dum01   = NULL ;
    mxArray *mx_dum02   = NULL ;

```



```

/* step 2 : assign memory */
mx_y0      = mxCreateDoubleMatrix(1, 1, mxREAL)    ;
mx_tspan    = mxCreateDoubleMatrix(2, 1, mxREAL)    ;
mx_length   = mxCreateDoubleMatrix(1, 1, mxREAL)    ;

mx_dum01    = mxCreateDoubleMatrix(100, 1, mxREAL)  ;
mx_dum02    = mxCreateDoubleMatrix(100, 1, mxREAL)  ;

/* step 3 : convert C/C++ double to mxArray */
mx_strfunc  = mxCreateString(strfunc)              ;

double2mxArray_scalarReal (db_y0    , mx_y0      ) ;
double2mxArray_vectorReal (db_tspan, mx_tspan    ) ;

/* step 4 : call an implemental function */
/* get size of the vector t */

mlfMyode45firstorder(3, &mx_dum01, &mx_dum02, &mx_length, mx_strfunc, mx_tspan, mx_y0);
int int_length = (int)mxAArray2double_scalarReal(mx_length)      ;

cout << "Length = " << int_length << endl;
mx_t    = mxCreateDoubleMatrix(int_length, 1, mxREAL)    ;
mx_y    = mxCreateDoubleMatrix(int_length, 1, mxREAL)    ;

/* solve the problem */
mlfMyode45firstorder(3, &mx_t, &mx_y, &mx_length, mx_strfunc, mx_tspan, mx_y0);

/* step 5 : convert back to C/C++ double */
double* db_t = new double[int_length] ;
double* db_y = new double[int_length] ;

mxArray2double_vectorReal(mx_t, db_t) ;
mxArray2double_vectorReal(mx_y, db_y) ;

cout << "The column of time" << endl ;
for (i=0 ; i<int_length; i++) {
    cout << db_t[i] << endl ;
}

cout << "The column of the function values y" << endl ;

```

```

for (i=0 ; i<int_length; i++) {
    cout << db_y[i] << endl ;
}

```

```

/* step 6 : free memories */

```

```

mxDestroyArray(mx_tspan);
mxDestroyArray(mx_strfunc);
mxDestroyArray(mx_y0)    ;

```

```

mxDestroyArray(mx_t)      ;
mxDestroyArray(mx_y)      ;
mxDestroyArray(mx_length) ;

```

```

mxDestroyArray(mx_dum01)   ;
mxDestroyArray(mx_dum02)   ;

```

```

delete [] db_t            ;
delete [] db_y            ;
}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 1 by using the function `myode45firstorder(..)` in the generated *cppodelib* library with the time step set to default.

Listing code

---

```

/* Example.cpp */

```

```

#pragma warning(disable : 4995)
#include "Example.h"

```

```

int main() {

```

```

    cout << "ODE problems." << endl;

```

```

    Test obj ;

```

```

    cout << "First order ODE." << endl ;

```

```

        obj.FirstOrder()          ;

    return 0 ;
}



---



/* Example.h */

#include <iostream.h>
#include "cppodelib.h"
#include "mwUtilityCompilerVer4.h"

class Test {

public:
    void FirstOrder() ;

    Test () {
        mclInitializeApplication(NULL,0);
        cppodelibInitialize();
    }

    ~Test () {
        cppodelibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */
void Test::FirstOrder() {

    /* Calculating first order ODE */
    int i ;
    mwArray mw_y0(1, 1, mxDOUBLE_CLASS) ;
    mw_y0(1,1) = 2.2 ;

    mwArray mw_tspan(2, 1, mxDOUBLE_CLASS) ;
    mw_tspan(1,1) = 0.2 ; /* begin interval t0 = 0.2      */
    mw_tspan(2,1) = 6.5 ; /* end interval, we choose this */
}

```

```

mwArray mw_dum01(100, 1, mxDOUBLE_CLASS) ;
mwArray mw_dum02(100, 1, mxDOUBLE_CLASS) ;

/* convert C/C++ double to mxArray */
    mxArray mw_strfunc("cos(t)") ;
//mwArray mw_strfunc("2+y") ;

/* call an implemental function */
/* get size of the vector t */
mwArray mw_length(1, 1, mxDOUBLE_CLASS) ;
myode45firstorder(3, mw_dum01, mw_dum02, mw_length, mw_strfunc, mw_tspan, mw_y0);

int int_length = (int)mw_length(1,1) ;
cout << "Length = " << int_length << endl;

mwArray mw_t(int_length, 1, mxDOUBLE_CLASS) ;
mwArray mw_y(int_length, 1, mxDOUBLE_CLASS) ;

/* solve the problem */
myode45firstorder(3, mw_t, mw_y, mw_length, mw_strfunc, mw_tspan, mw_y0);

/* convert back to C/C++ double */
double* db_t = new double[int_length] ;
double* db_y = new double[int_length] ;

mwArray2double_vectorReal(mw_t, db_t) ;
mwArray2double_vectorReal(mw_y, db_y) ;

cout << "The column of time" << endl ;
for (i=0 ; i<int_length; i++) {
    cout << db_t[i] << endl ;
}

cout << "The column of the function values y" << endl ;
for (i=0 ; i<int_length; i++) {
    cout << db_y[i] << endl ;
}

```

```

/* free memories */
delete [] db_t      ;
delete [] db_y      ;

}

```

---

end code

---

**Problem 2** Find the function,  $y(t)$ , from the ODE function:

$$\frac{dy}{dt} = 6.4t^2 - 3.8ty;$$

with initial condition :

$$y_0 = 1.24 \quad \text{at} \quad t_0 = 0.15$$

This Problem 2 is solved similarly to Problem 1. In the code we just change the expression string:

```
const char strfunc[] = "6.4*t.^2 - 3.8*t*y" ;
```

### Remark

To find particular function values we need to set the argument **tspan** as a column vector including the finding time. For example, the following code to find the particular values  $y$  at  $t = 0.15$ , 0.2, 2.6, and 5.0 in Problem 2.

## A. FOR C SHARED LIBRARY

Listing code

---

```

void Test::FirstOrderGetParticularValues() {

/* Calculating first order ODE */
const char strfunc[] = "6.4*t.^2 - 3.8*t*y" ;

double db_y0= 1.24 ; /* initial condition at t0          */

double db_tspan[4] ;
db_tspan[0] = 0.15 ; /* begin interval t0 = 0.15          */
db_tspan[1] = 0.2   ; /* choose a particular time t = 0.2    */
db_tspan[2] = 2.6   ; /* choose a particular time t = 2.6    */
db_tspan[3] = 5.0   ; /* choose a particular time t = 5.0    */
}

```

```

int i ;

/* step 1 : declare mxArray variables */
mxArray *mx_strfunc ;
mxArray *mx_y0      = NULL ;
mxArray *mx_tspan   = NULL ;

mxArray *mx_length  = NULL ;
mxArray *mx_t        = NULL ;
mxArray *mx_y        = NULL ;

mxArray *mx_dum01    = NULL ;
mxArray *mx_dum02    = NULL ;

/* step 2 : assign memory */
mx_y0      = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_tspan   = mxCreateDoubleMatrix(4, 1, mxREAL) ;
mx_length  = mxCreateDoubleMatrix(1, 1, mxREAL) ;

mx_dum01    = mxCreateDoubleMatrix(100, 1, mxREAL) ;
mx_dum02    = mxCreateDoubleMatrix(100, 1, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
mx_strfunc = mxCreateString(strfunc) ;

double2mxArray_scalarReal (db_y0 , mx_y0 ) ;
double2mxArray_vectorReal (db_tspan, mx_tspan ) ;

/* step 4 : call an implemental function */
/* get size of the vector t */

mlfMyode45firstorder(3, &mx_dum01, &mx_dum02, &mx_length, mx_strfunc, mx_tspan, mx_y0);
int int_length = (int)mxAarray2double_scalarReal(mx_length) ;

cout << "Length = " << int_length << endl;
mx_t    = mxCreateDoubleMatrix(int_length, 1, mxREAL) ;
mx_y    = mxCreateDoubleMatrix(int_length, 1, mxREAL) ;

/* solve the problem */
mlfMyode45firstorder(3, &mx_t, &mx_y, &mx_length, mx_strfunc, mx_tspan, mx_y0);

```

```

/* step 5 : convert back to C/C++ double */
double* db_t = new double[int_length] ;
double* db_y = new double[int_length] ;

mxArray2double_vectorReal(mx_t, db_t) ;
mxArray2double_vectorReal(mx_y, db_y) ;

cout << "The column of time" << endl ;
for (i=0 ; i<int_length; i++) {
    cout << db_t[i] << endl ;
}

cout << "The column of the function values y" << endl ;
for (i=0 ; i<int_length; i++) {
    cout << db_y[i] << endl ;
}

/* step 6 : free memories */
mxDestroyArray(mx_tspan);
mxDestroyArray(mx_strfunc);
mxDestroyArray(mx_y0)    ;

mxDestroyArray(mx_t)    ;
mxDestroyArray(mx_y)    ;
mxDestroyArray(mx_length)    ;

mxDestroyArray(mx_dum01)    ;
mxDestroyArray(mx_dum02)    ;

delete [] db_t            ;
delete [] db_y            ;
}

```

---

end code

## B. FOR C++ SHARED LIBRARY

Listing code

---

```
void Test::FirstOrderGetParticularValues() {

    /* Calculating first order ODE */
    int i ;
    mxArray mw_y0(1, 1, mxDOUBLE_CLASS) ;
    mw_y0(1,1) = 1.24 ;

    mxArray mw_tspan(4, 1, mxDOUBLE_CLASS) ;
    mw_tspan(1,1) = 0.2 ; /* begin interval t0 = 0.2          */
    mw_tspan(2,1) = 6.5 ; /* end interval, we choose this    */

    mw_tspan(1,1) = 0.15 ; /* begin interval t0 = 0.15          */
    mw_tspan(2,1) = 0.2   ; /* choose a particular time t = 0.2      */
    mw_tspan(3,1) = 2.6   ; /* choose a particular time t = 2.6      */
    mw_tspan(4,1) = 5.0   ; /* choose a particular time t = 5.0      */

    mxArray mw_dum01(100, 1, mxDOUBLE_CLASS) ;
    mxArray mw_dum02(100, 1, mxDOUBLE_CLASS) ;

    /* convert C/C++ double to mxArray */
    mxArray mw_strfunc("6.4*t.^2 - 3.8*t*y") ;

    /* call an implemental function */
    /* get size of the vector t */
    mxArray mw_length(1, 1, mxDOUBLE_CLASS) ;
    myode45firstorder(3, mw_dum01, mw_dum02, mw_length, mw_strfunc, mw_tspan, mw_y0);

    int int_length = (int)mw_length(1,1) ;
    cout << "Length = " << int_length << endl;

    mxArray mw_t(int_length, 1, mxDOUBLE_CLASS) ;
    mxArray mw_y(int_length, 1, mxDOUBLE_CLASS) ;

    /* solve the problem */
    myode45firstorder(3, mw_t, mw_y, mw_length, mw_strfunc, mw_tspan, mw_y0);
```



```

/* convert back to C/C++ double */
double* db_t = new double[int_length] ;
double* db_y = new double[int_length] ;

mwArray2double_vectorReal(mw_t, db_t) ;
mwArray2double_vectorReal(mw_y, db_y) ;

cout << "The column of time" << endl ;
for (i=0 ; i<int_length; i++) {
    cout << db_t[i] << endl ;
}

cout << "The column of the function values y" << endl ;
for (i=0 ; i<int_length; i++) {
    cout << db_y[i] << endl ;
}

/* free memories */
delete [] db_t ;
delete [] db_y ;

}

```

---

end code

## 10.2 Second Order ODE

**Problem 3** Find the function  $y(t)$ , and its derivative  $y'(t)$  from the ODE function,

$$y'' - 2y' - 6y = \cos(3t)$$

with initial conditions :

$$\text{at } t_0 = 0.12, \quad y_0 = 0.2 \quad \text{and} \quad y'_0 = 1.1$$

### 10.2.1 Analysis of second order ODE

1. To solve Problem 3 by writing M-files, we can write an M-file `mysecondfunc.m` as follows:

```
function dy = mysecondfunc(t, y)
```

```
dy = [y(2) ; cos(3*t) + 2*y(2) + 6*y(1)] ;
```

and in MATLAB Command Window write:

```
>> tspan = [1.2 ; 2.5] ;
```

```
>> ybc = [0.2 ; 1.1] ;
```

```
>> [t,y] = ode45(@mysecondfunc, tspan, ybc)
```

2. To explain the code in the M-file `mysecondfunc.m`, we rewrite and set from the provided equation :

$$y = y_1$$

$$y' = y_2$$

$$y'' = y_2'$$

Problem 3 then becomes:

$$y = y_1$$

$$y' = y_2$$

$$\begin{aligned} y'' &= \cos(3t) + 2y' + 6y \\ &= \cos(3t) + 2y_2 + 6y_1 \end{aligned}$$

This is the second expression in the M-file `mysecondfunc.m`.

3. The function `mysecondfunc(..)` which is passed to the ode function `ode45(..)` has a return including two arrays:

- First array is the first derivative of the function  $y$ , as `y(2)`
- Second array is the second derivative of the function  $y$ , as  
`cos(3*t) + 2*y(2) + 6*y(1) ;`

The M-file `yoursecondfunc.m`, with which we use for creating a C shared library ***odelib*** or a C++ library ***cppodelib***, as shown in above, also has a return including two arrays:

- First array is the first derivative of the function  $y$ , as follows:

```
f0 = inline('yy') ;
```

```
dy(1,:) = feval( f0, y(2) ) ;
```

- Second array is the second derivative of the function  $y$ , as follows:

```
cos(3*t) + 2*y(2) + 6*y(1) ;
```

This second array is represented in the code:

```
f1 = inline(strfunction) ;
```

```
dy(2,:) = feval( f1, t , y(1), y(2) ) ;
```

### 10.2.2 Using a second order ODE function

As explain in above, we have an easy way to use the generated function `mlfMyode45secondorder(..)` in the C shared library ***odelib*** or `myode45secondorder(..)` in the C++ shared library ***cppodelib*** to solve second order ODE problems by following the steps:

1. Write your ode-function with second derivative in the left-hand-side, for example:

$$y'' = \cos(3t) + 2y' + 6y$$

2. Rewrite your ode-function as an MATLAB expression string, for example:

$$y'' = \cos(3*t) + 2*y' + 6*y$$

3. Replace  $y'$  by `yprime`, for example:

$$y'' = \cos(3*t) + 2*yprime + 6*y$$

4. Use the right-hand-side as a string to use in the code, for example:

```
constchar strfunc[] = "cos(3*t) + 2*yprime + 6*y" ;
```

## A. FOR C SHARED LIBRARY

The following is the code to solve Problem 3 by using the function `mlfMyode45secondorder(..)` in the generated ***odelib*** library to solve second order ODE problems.

Listing code

---

```
void Test::SecondOrder() {

/* Calculating second order ODE */
//const char strfunc[] = "cos(t)" ;
const char strfunc[] = "cos(3*t) + 2*yprime + 6*y" ;

double db_ycb[2] ; /* y boundary conditions */
db_ycb[0] = 0.2 ; /* initial condition of y at t0 */
db_ycb[1] = 1.1 ; /* initial condition of y' at t0 */

double db_tspan[2] ;
```

```

db_tspan[0] = 1.2    ; /* begin interval t0 = 1.2          */
db_tspan[1] = 2.5    ; /* end interval, we choose this    */

int i ;

/* step 1 : declare mxArray variables */
mxArray *mx_strfunc ;
mxArray *mx_ybc      = NULL ;
mxArray *mx_tspan    = NULL ;

mxArray *mx_length   = NULL ;
mxArray *mx_t        = NULL ;
mxArray *mx_y        = NULL ;

mxArray *mx_dum01    = NULL ;
mxArray *mx_dum02    = NULL ;

/* step 2 : assign memory */
mx_ybc      = mxCreateDoubleMatrix(2, 1, mxREAL)    ;
mx_tspan    = mxCreateDoubleMatrix(2, 1, mxREAL)    ;
mx_length   = mxCreateDoubleMatrix(1, 1, mxREAL)    ;

mx_dum01    = mxCreateDoubleMatrix(100, 1, mxREAL) ;
mx_dum02    = mxCreateDoubleMatrix(100, 1, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
mx_strfunc = mxCreateString(strfunc)    ;
double2mxArray_vectorReal (db_ybc , mx_ybc      ) ;
double2mxArray_vectorReal (db_tspan, mx_tspan    ) ;

/* step 4 : call an implemental function */
/* get size of the vector t */
mlfMyode45secondorder(3, &mx_dum01, &mx_dum02, &mx_length, mx_strfunc, mx_tspan, mx_ybc);
int int_length = (int)mxAarray2double_scalarReal(mx_length)    ;

mx_t    = mxCreateDoubleMatrix(int_length, 1, mxREAL)    ;
mx_y    = mxCreateDoubleMatrix(int_length, 2, mxREAL)    ;

/* solve the problem */

```

```

mlfMyode45secondorder(3, &mx_t, &mx_y, &mx_length, mx_strfunc, mx_tspan, mx_ybc);

cout << "Length = " << int_length << endl ;

/* step 5 : convert back to C/C++ double */
double* db_t    = new double[int_length]    ;
mxArray2double_vectorReal(mx_t, db_t) ;

double** db_y    ;
db_y = new double*[int_length] ;
    for(i=0; i<int_length; i++) {
        db_y[i] = new double [2] ;
    }

mxArray2double_vectorReal(mx_t, db_t) ;
mxArray2double_matrixReal(mx_y, db_y) ;

cout << "The column of time" << endl ;
for (i=0 ; i<int_length; i++) {
    cout << db_t[i] << endl ;
}

cout << "The column of the function values y :" << endl ;
/* first column of the matrix y */
for (i=0 ; i<int_length; i++) {
    cout << db_y[i][0] << endl ;
}

cout << "The column of the first derivative y' :" << endl ;
/* second column of the matrix y */
for (i=0 ; i<int_length; i++) {
    cout << db_y[i][1] << endl ;
}

/* step 6 : free memories */
mxDestroyArray(mx_strfunc);
mxDestroyArray(mx_tspan);
mxDestroyArray(mx_ybc) ;

mxDestroyArray(mx_t)    ;

```

```

mxDestroyArray(mx_y)      ;
mxDestroyArray(mx_length)  ;

mxDestroyArray(mx_dum01)   ;
mxDestroyArray(mx_dum02)   ;

delete [] db_t             ;
delete [] db_y             ;
}

```

---

end code

---

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 3 by using the function `myode45secondorder(..)` in the generated *cppodelib* library to solve second order ODE problems.

Listing code

---

```

/* ***** */
void Test::SecondOrder() {

/* Calculating second order ODE */
int i ;
mwArray mw_ybc(2, 1, mxDOUBLE_CLASS) ;
mw_ybc(1,1) = 0.2 ;
mw_ybc(2,1) = 1.1 ;

mwArray mw_tspan(2, 1, mxDOUBLE_CLASS) ;
mw_tspan(1,1) = 1.2 ; /* begin interval t0 = 0.2      */
mw_tspan(2,1) = 2.5 ; /* end interval, we choose this */

mwArray mw_dum01(100, 1, mxDOUBLE_CLASS) ;
mwArray mw_dum02(100, 1, mxDOUBLE_CLASS) ;

mwArray mw_strfunc("cos(3*t) + 2*yprime + 6*y") ;

/* call an implemental function */
/* get size of the vector t */
mwArray mw_length(1, 1, mxDOUBLE_CLASS) ;
myode45secondorder(3, mw_dum01, mw_dum02, mw_length, mw_strfunc, mw_tspan, mw_ybc);

```

```

int int_length = (int)mw_length(1,1) ;
cout << "Length = " << int_length << endl;

mwArray mw_t(int_length, 1, mxDOUBLE_CLASS) ;
mwArray mw_y(int_length, 1, mxDOUBLE_CLASS) ;

/* solve the problem */
myode45secondorder(3, mw_t, mw_y, mw_length, mw_strfunc, mw_tspan, mw_ycb);

/* convert back to C/C++ double */
double* db_t = new double[int_length] ;
mwArray2double_vectorReal(mw_t, db_t) ;

double** db_y ;
db_y = new double*[int_length] ;
    for(i=0; i<int_length; i++) {
        db_y[i] = new double [2] ;
    }

mwArray2double_vectorReal(mw_t, db_t) ;
mwArray2double_matrixReal(mw_y, db_y) ;

cout << "The column of time" << endl ;
for (i=0 ; i<int_length; i++) {
    cout << db_t[i] << endl ;
}

cout << "The column of the function values y :" << endl ;
/* first column of the matrix y */
for (i=0 ; i<int_length; i++) {
    cout << db_y[i][0] << endl ;
}

cout << "The column of the first derivative y' :" << endl ;
/* second column of the matrix y */
for (i=0 ; i<int_length; i++) {
    cout << db_y[i][1] << endl ;
}

/* free memories */

```

```
delete [] db_t      ;
delete [] db_y      ;
```

---

— end code —

**Note:**

1. In solving second order ODE problem, the output matrix **y** includes two columns: The first column is values of the function  $y(t)$  and the second column is values of the first derivative  $y'(t)$ .
2. In this chapter we describe the methods to solve ODE problems by passing your ode-function to the C/C++ code. These methods are useful when your function are changing in the run-time or your function is provided in an application. If your ode-function is known in the design time, you can call directly. For example, the M-file myotherode.m as follow below will directly call the M-file mysecondfunc.m:

```
function dy = mysecondfunc(t, y)
dy = [y(2) ; cos(3*t) + 2*y(2) + 6*y(1)] ;
```

---

```
function [t, y, lengthtime] = myotherode(tspan, y0)
```

```
[t,y] = ode45(@mysecondfunc, tspan, y0) ;
lengthtime = length(t) ;
```



# Chapter 11

## Integration

In this chapter we'll generate a C shared library *integrationlib* and a C++ shared library *cppintegrationlib* from common M-files working on problems of single and double integrations. The generated functions of these libraries will be used in a MSVC .Net project to solve the integral problems.

Following are steps to create a C shared library *integrationlib.dll* and a C++ shared library *cpintegrationlib.dll* which will be used to solve integral problems in the next sections.

We will write the M-files *myquad.m* and *mydblquad.m*. These functions will be used to generate the C and C++ shared libraries.

---

```
function y = myquad(strfunc, a, b)
```

```
F = inline(strfunc) ;
```

```
y = quad(F, a, b) ;
```

---

```
function dbint = mydblquad(strfunc, x1, x2, y1, y2)
```

```
F = inline(strfunc) ;
```

```
dbint = dblquad(F, x1, x2, y1, y2) ;
```

## A. FOR C SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C shared library *integrationlib*.

```
mcc -B csharedlib:integrationlib myquad.m mydblquad.m
```

2. MATLAB Compiler 4.0 will create eight files for this library:

```
integrationlib.c      integrationlib.ctf      integrationlib.dll
integrationlib.exp    integrationlib.exports  integrationlib.h
integrationlib.lib    integrationlib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

In the following sections, we'll use the following implemental functions in this library to solve the common problems in the integration (open the file `integrationlib.h` to see the names of these functions):

```
void mlfMyquad(int nargout, mxArray** y, mxArray* strfunc
               , mxArray* a, mxArray* b);

void mlfMydblquad(int nargout, mxArray** dbint
                  , mxArray* strfunc, mxArray* x1
                  , mxArray* x2, mxArray* y1, mxArray* y2);
```

## B. FOR C++ SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C++ shared library *integrationlib*.

```
mcc -W cpplib:cppintegrationlib -T link:lib myquad.m mydblquad.m
```

2. MATLAB Compiler 4 will create eight files for this library:

```
cppintegrationlib.c    cppintegrationlib.ctf    cppintegrationlib.dll
cppintegrationlib.exp  cppintegrationlib.exports  cppintegrationlib.h
cppintegrationlib.lib  cppintegrationlib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

In the following sections, we'll use the following implemental functions in this library to solve the common problems in the integration (open the file `cppintegrationlib.h` to see the names of these functions):

```

void myquad(int nargout, mxArray& y, const mxArray& strfunc
            , const mxArray& a, const mxArray& b);

void mydblquad(int nargout, mxArray& dbint, const mxArray& strfunc
               , const mxArray& x1, const mxArray& x2
               , const mxArray& y1, const mxArray& y2);

```

## 11.1 Single Integration

**Problem 1** Calculate the integration :

$$I = \int_0^{3\pi} (\sin(x) + x^2) dx$$

### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 1 by using the functions `mlfMyquad(..)` in the C shared library *integrationlib*. The function `mlfMyquad(..)` used a MATLAB function `quad(..)` with default as shown in the M-file `myquad.m`. To use more options see this function `quad(..)` refer to [7].

Listing code

---

```

/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Single integration:" << endl;
    Test obj ;
    obj.singleIntegration() ;
    cout << endl;

    return 0 ;
}

```

---

```

/* Example.h */

#include <iostream.h>
#define _USE_MATH_DEFINES
#include <math.h>

#include "integrationlib.h"
#include "mxUtilityCompilerVer4.h"

class Test {

public:
void singleIntegration();
void dbIntegration() ;

    Test () {
        mclInitializeApplication(NULL,0);
        integrationlibInitialize();
    }

    ~Test () {
        integrationlibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */

void Test::singleIntegration()
{
    const char strfunc[] = "sin(x) + x.^2" ;

    double db_beginInterval = 0 ;
    double db_endInterval   = 3*M_PI ; // using the value pi in math.h

    /* step 1 : declare mxArray variables */
    mxArray *mx_strfunc ;
    mxArray *mx_beginInterval = NULL ;

```

```

mxArray *mx_endInterval    = NULL ;
mxArray *mx_y              = NULL ;

/* step 2 : assign memory */
mx_beginInterval    = mxCreateDoubleMatrix(1, 1, mxREAL)    ;
mx_endInterval      = mxCreateDoubleMatrix(1, 1, mxREAL)    ;
mx_y                = mxCreateDoubleMatrix(1, 1, mxREAL)    ;

/* step 3 : convert C/C++ double to mxArray */
mx_strfunc = mxCreateString(strfunc)    ;
double2mxArray_scalarReal (db_beginInterval, mx_beginInterval) ;
double2mxArray_scalarReal (db_endInterval , mx_endInterval ) ;

/* step 4 : call an implemental function */
mlfMyquad(1, &mx_y, mx_strfunc, mx_beginInterval, mx_endInterval);

/* step 5 : convert back to C/C++ double */
double db_y ;
db_y = mxArray2double_scalarReal(mx_y) ;
cout << " I = " << db_y ;

/* step 6 : free memories */
mxDestroyArray(mx_beginInterval )    ;
mxDestroyArray(mx_endInterval )    ;
mxDestroyArray(mx_y )    ;
mxDestroyArray(mx_strfunc )    ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 1 by using the functions `myquad(..)` in the C++ library *cppintegrationlib*. The function `myquad(..)` used a MATLAB function `quad(..)` with an option as shown in the M-file `myquad.m`. To use more options see this function `quad(..)` refer to [7].

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Single integration:" << endl;
    Test obj ;
    obj.singleIntegration() ;

    return 0 ;
}
```

---

```
/* Example.h */

#include <iostream.h>
#define _USE_MATH_DEFINES
#include <math.h>

#include "cppintegrationlib.h"
#include "mwUtilityCompilerVer4.h"

class Test {
public:
    void singleIntegration() ;

    Test () {
        mclInitializeApplication(NULL,0);
        cppintegrationlibInitialize();
    }

    ~Test () {
        cppintegrationlibTerminate();
        mclTerminateApplication();
    }

} ;
```

```

/* ***** */
void Test::singleIntegration() {

double db_beginInterval = 0 ;
double db_endInterval   = 3*M_PI ; // using the value pi in math.h

/* assign memory */
mwArray mw_beginInterval(1,1, mxDOUBLE_CLASS) ;
mw_beginInterval = db_beginInterval ;

mwArray mw_endInterval(1,1, mxDOUBLE_CLASS) ;
mw_endInterval = db_endInterval ;

mwArray mw_strfunc("sin(x) + x.^2");

/* call an implemental function */
mwArray mw_y(1, 1, mxDOUBLE_CLASS) ;
myquad(1, mw_y, mw_strfunc, mw_beginInterval, mw_endInterval);

/* convert back to C/C++ double */
double db_y = (double) mw_y(1,1) ;
cout << " I = " << db_y << endl ;

}

```

---

end code

### Note

1. The generated function `mlfMyquad(..)` or `myquade(..)` is a function-function and has an argument as an expression string. The form of this expression string follows the rule of a MATLAB expression string.
2. See the MATLAB function `quad(..)` for the other method to calculate the single integration.

## 11.2 Double-Integration

**Problem 2**      Calculate the double-integration:

$$I = \int_0^{\frac{\pi}{2}} \int_0^{\pi} \left( \sin(x) + x^2 + y^3 \right) dx \, dy$$

## A. FOR C SHARED LIBRARY

The following is the code to solve Problem 2 using the functions `mlfMydblquad(..)` in the C library *integrationlib*. The function `mlfMydblquad(..)` used a MATLAB function `dblquad(..)` with an option as shown in the M-file `mydblquad.m`. To use more options see this function `dblquad(..)` refer to [7].

Listing code

---

```
void Test::dbIntegration()
{

    const char  strfunc[] = "sin(x) + x.^2 + y.^3" ;

    double db_x1 = 0 ;
    double db_x2 = 3*M_PI ; // using the value pi in math.h

    double db_y1 = 0 ;
    double db_y2 = M_PI ;

    /* step 1 : declare mxArray variables */
    mxArray *mx_strfunc ;
    mxArray *mx_x1 = NULL ;
    mxArray *mx_x2 = NULL ;

    mxArray *mx_y1 = NULL ;
    mxArray *mx_y2 = NULL ;

    mxArray *mx_II  = NULL ;

    /* step 2 : assign memory */
    mx_x1 = mxCreateDoubleMatrix(1, 1, mxREAL) ;
    mx_x2 = mxCreateDoubleMatrix(1, 1, mxREAL) ;

    mx_y1 = mxCreateDoubleMatrix(1, 1, mxREAL) ;
    mx_y2 = mxCreateDoubleMatrix(1, 1, mxREAL) ;

    mx_II = mxCreateDoubleMatrix(1, 1, mxREAL) ;

    /* step 3 : convert C/C++ double to mxArray */
```



```

mx_strfunc = mxCreateString(strfunc)    ;
double2mxArray_scalarReal (db_x1, mx_x1) ;
double2mxArray_scalarReal (db_x2, mx_x2) ;

double2mxArray_scalarReal (db_y1, mx_y1) ;
double2mxArray_scalarReal (db_y2, mx_y2) ;

/* step 4 : call an implemental function */
mlfMydblquad(1, &mx_II, mx_strfunc, mx_x1, mx_x2, mx_y1, mx_y2) ;

/* step 5 : convert back to C/C++ double */
double db_II ;
db_II = mxArray2double_scalarReal(mx_II) ;
cout << " II = " << db_II ;

/* step 6 : free memories */
mxDestroyArray(mx_strfunc) ;
mxDestroyArray(mx_x1 ) ;
mxDestroyArray(mx_x2 ) ;

mxDestroyArray(mx_y1 ) ;
mxDestroyArray(mx_y2 ) ;

mxDestroyArray(mx_II ) ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 2 using the functions mydblquad(..) int the C++ library *cppintegrationlib*. The function mydblquad(..) used a MATLAB function dblquad(..) with an option as shown in the M-file mydblquad.m. To use more options see this function dblquad(..) refer to [7].

Listing code

---

```

void Test::dbIntegration() {

mwArray mw_x1(1,1, mxDOUBLE_CLASS) ;

```

```

mw_x1 = 0 ;

mwArray mw_x2(1,1, mxDOUBLE_CLASS) ;
mw_x2 = 3*M_PI ; // using the value pi in math.h

mwArray mw_y1(1,1, mxDOUBLE_CLASS) ;
mw_y1 = 0 ;

mwArray mw_y2(1,1, mxDOUBLE_CLASS) ;
mw_y2 = M_PI ;

mwArray mw_strfunc("sin(x) + x.^2 + y.^3") ;

/* call an implemental function */
mwArray mw_II(1, 1, mxDOUBLE_CLASS) ;
mydblquad(1, mw_II, mw_strfunc, mw_x1, mw_x2, mw_y1, mw_y2) ;

/* convert back to C/C++ double */
double db_II = (double) mw_II(1,1) ;
cout << " II = " << db_II ;

}

```

---

end code

## Chapter 12

# Curve Fitting and Interpolations

In this chapter we'll generate a C shared library *curvefittinglib* and a C++ shared library *cppcurvefittinglib* from common M-files working on curve fitting problems. The generated functions of these libraries will be used in MSVC .Net project to solve common curve fitting problems.

Following are steps to create a C shared library curvefittinglib.dll and a C++ shared library cp-curvefittinglib.dll which will be used to solve curve fitting problems in the next sections.

We will write the M-files as shown below. These functions will be used to generate the C and C++ shared libraries.

```
myinterp1.m, myinterp2.m, mypolyfit.m, mypolyval.m,  
mymeshgrid.m, mygriddata.m, and myfinemeshgrid.m
```

---

```
function yi = myinterp1(x,y,xi)
```

```
yi = interp1(x,y,xi) ;
```

---

```
function ZI = myinterp2(X,Y,Z,XI,YI,method)
```

```
ZI = interp2(X,Y,Z,XI,YI,method) ;
```

---

```
function p = mypolyfit(x,y,n)
```

```
p = polyfit(x,y,n) ;
```

---

```
function y = mypolyval(p,x)
```

```
y = polyval(p,x) ;
```

---

```
function [X,Y] = mymeshgrid(vectorstepx, vectorstepy)
```

```
%Using two colons to create a vector with increments between  
%first and end elements.
```

```
[X,Y] = meshgrid( vectorstepx(1):vectorstepx(2):vectorstepx(3), ...  
                  vectorstepy(1):vectorstepy(2):vectorstepy(3) ) ;
```

---

```
function ZI = mygriddata(x,y,z,XI,YI)
```

```
ZI = griddata(x,y,z,XI,YI) ;
```

---

```
function [row,col] = myfinemeshgrid(vectorstepx, vectorstepy)
```

```
%Using two colons to create a vector with increments between  
%first and end elements.
```

```
[X,Y] = meshgrid( vectorstepx(1):vectorstepx(2):vectorstepx(3), ...  
                  vectorstepy(1):vectorstepy(2):vectorstepy(3) ) ;
```

```
[row, col] = size(X) ;
```

## A. FOR C SHARED LIBRARY

1. Write the command in Windows Command Prompt to create a C shared library *curvefittinglib* as follows:

```
mcc -B csharedlib:curvefittinglib myinterp1.m myinterp2.m mypolyfit.m  
mypolyval.m mymeshgrid.m mygriddata.m myfinemeshgrid.m
```

2. MATLAB Compiler 4.0 will create eight files for this C shared library:

curvefittinglib.c	curvefittinglib.ctf	curvefittinglib.dll
curvefittinglib.exp	curvefittinglib.exports	curvefittinglib.h
curvefittinglib.lib	curvefittinglib_mcc_component_data.c	

Add and set these files to the MSVC .Net project as described in Chapter 6.

**Note:** In this chapter we use typical functions to solve the particular problems. There are another options and MATLAB functions to solve the curve fitting problems, refer to [7].

3. In the following sections, we'll use the following implemental functions in this library to solve the common curve fitting problem (open the file curvefittinglib.h to see the names of these functions):

```
void mlfMyinterp1(int nargout, mxArray** yi, mxArray* x
                  , mxArray* y, mxArray* xi);

void mlfMyinterp2(int nargout, mxArray** ZI, mxArray* X
                  , mxArray* Y, mxArray* Z, mxArray* XI
                  , mxArray* YI, mxArray* method);

void mlfMypolyfit(int nargout, mxArray** p, mxArray* x
                  , mxArray* y, mxArray* n);

void mlfMypolyval(int nargout, mxArray** y, mxArray* p, mxArray* x);

void mlfMymeshgrid(int nargout, mxArray** X, mxArray** Y
                   , mxArray* x, mxArray* y);

void mlfMygriddata(int nargout, mxArray** ZI, mxArray* x, mxArray* y
                   , mxArray* z, mxArray* XI, mxArray* YI);
```

## B. FOR C++ SHARED LIBRARY

1. Write the command in Windows Command Prompt to create a C++ shared library *cppcurvefittinglib* as follows:

```
mcc -W cpplib:cppcurvefittinglib -T link:lib myinterp1.m myinterp2.m mypolyfit.m
mypolyval.m mymeshgrid.m mygriddata.m myfinemeshgrid.m
```

2. MATLAB Compiler 4 will create eight files for this C++ shared library:



## 12.1 Polynomial Curve Fitting

This section describe how to use the functions in the generated libraries to find the coefficients of a polynomial function that fits a set of data in the least-squares sense. An array **c** representing these coefficients is in the polynomial form:

$$f(x) = c_1x^n + c_2x^{n-1} + c_3x^{n-2} + \cdots + c_nx + c_{n+1} \quad (12.1)$$

**Problem 1** There are two arrays **X** and **Y** which have a relationship via a function,  $y = f(x)$ ,  $x \in \mathbf{X}$  and  $y \in \mathbf{Y}$ .

**input** Array **X** = { 1, 2, 3, 4, 5, 6 }

Array **Y** = { 6.8, 50.2, 140.8, 280.5, 321.4, 428.6 }

**output** Finding an array **c** of the polynomial function in Eq. 12.1 with degree n=3;  
since degree n=3, the function  $y = f(x)$  has the form:

$$y = c_1x^3 + c_2x^2 + c_3x + c_4$$

Calculating the interpolation value of the function  $y(x)$ , at  $x = 2.2$

### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 1. In the code, we will use the function `mlfMypolyfit(..)` with degree of  $n = 3$  to obtain the coefficient array, and use the function `mlfMypolyval(..)` to calculate the function value.

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Curve Fitting:" << endl;
    Test obj ;
    obj.PolynomialFittingCurve() ;

    return 0 ;
}
```

---

```

/* Example.h */

#include <iostream.h>
#include "curvefittinglib.h"
#include "mxUtilityCompilerVer4.h"

class Test {

public:
void PolynomialFittingCurve() ;

    Test () {
        mclInitializeApplication(NULL,0);
        curvefittinglibInitialize();
    }

    ~Test () {
        curvefittinglibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */

void Test::PolynomialFittingCurve() {

double db_X[6] = { 1, 2, 3, 4, 5, 6 } ;
double db_Y[6] = { 6.8, 50.2, 140.8, 280.5, 321.4, 428.6 };

double db_three      = 3      ;
double db_oneValue   = 2.2    ;

/* step 1 : declare mxArray variables */
mxArray *mx_X        = NULL ;
mxArray *mx_Y        = NULL ;
mxArray *mx_coefs    = NULL ;

mxArray *mx_three     = NULL ;

```



```

mxArray *mx_oneValue      = NULL ;
mxArray *mx_funcValue     = NULL ;
/* step 2 : assign memory */
int vectorSize = 6 ;

mx_X      = mxCreateDoubleMatrix(1, vectorSize, mxREAL) ;
mx_Y      = mxCreateDoubleMatrix(1, vectorSize, mxREAL) ;
mx_coefs   = mxCreateDoubleMatrix(1, 4, mxREAL) ;

mx_three   = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_oneValue = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_funcValue = mxCreateDoubleMatrix(1, 1, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_vectorReal (db_X, mx_X) ;
double2mxArray_vectorReal (db_Y, mx_Y) ;

double2mxArray_scalarReal (db_three, mx_three) ;
double2mxArray_scalarReal (db_oneValue, mx_oneValue) ;

/* step 4 : call an implemental function */
mlfMypolyfit(1, &mx_coefs, mx_X, mx_Y, mx_three);

/* step 5 : convert back to C/C++ double */
double *db_coefs = new double [4] ;
mxArray2double_vectorReal(mx_coefs, db_coefs) ;

/* print out */
cout << "The polynomial:" << endl ;
cout << db_coefs[0] << " x^3 " << " + " ;
cout << db_coefs[1] << " x^2 " << " + " ;
cout << db_coefs[2] << " x " << " + " ;
cout << db_coefs[3] << endl;

/* calculate the function value at oneValue */
mlfMypolyval(1, &mx_funcValue, mx_coefs, mx_oneValue);
double db_funcValue = mxArray2double_scalarReal(mx_funcValue) ;
cout << "The function value at 2.2 is: " << db_funcValue << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_X) ;

```

```

mxDestroyArray(mx_Y ) ;
mxDestroyArray(mx_coefs);

mxDestroyArray(mx_three );
mxDestroyArray(mx_oneValue );
mxDestroyArray(mx_funcValue);

delete []db_coefs ;

}

```

---

end code

---

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 1. In the code, we will use the function `mypolyfit(..)` with degree of  $n = 3$  to obtain the coefficient array, and use the function `mypolyval(..)` to calculate the function value.

Listing code

---

```

/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Curve Fitting:" << endl ;
    Test obj ;

    cout << "1. Polynomial" << endl ;
    obj.PolynomialFittingCurve() ;

    return 0 ;
}

```

---

```

/* Example.h */

#include <iostream.h>
#include "cppcurvefittinglib.h"

```

```

#include "mwUtilityCompilerVer4.h"

class Test {

public:
void PolynomialFittingCurve()          ;

    Test () {
        mclInitializeApplication(NULL,0);
        cppcurvefittinglibInitialize();
    }

    ~Test () {
        cppcurvefittinglibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */
void Test::PolynomialFittingCurve() {

double db_X[6] = { 1, 2, 3, 4, 5, 6 } ;
double db_Y[6] = { 6.8, 50.2, 140.8, 280.5, 321.4, 428.6 };

int vectorSize = 6 ;

/* convert C/C++ double to mxArray */
mxArray mw_X(1, vectorSize, mxDOUBLE_CLASS) ;
mw_X.SetData(db_X, vectorSize) ;

mxArray mw_Y(1, vectorSize, mxDOUBLE_CLASS) ;
mw_Y.SetData(db_Y, vectorSize) ;

/* call an implemental function */
mxArray mw_three(1, 1, mxDOUBLE_CLASS) ;
mw_three(1,1) = 3 ;

mxArray mw_coefs(1, 4, mxDOUBLE_CLASS) ;
mypolyfit(1, mw_coefs, mw_X, mw_Y, mw_three);

```

```

/* convert back to C/C++ double */
double *db_coefs = new double [4] ;
mwArray2double_vectorReal(mw_coefs, db_coefs) ;

/* print out */
cout << "The polynomial:" << endl ;
cout << db_coefs[0] << " x^3 " << " + " ;
cout << db_coefs[1] << " x^2 " << " + " ;
cout << db_coefs[2] << " x " << " + " ;
cout << db_coefs[3] << endl;

/* calculate the function value at oneValue */
mwArray mw_funcValue(1, 1, mxDOUBLE_CLASS) ;
mwArray mw_oneValue(1, 1, mxDOUBLE_CLASS) ;
mw_oneValue(1,1) = 2.2 ;

mypolyval(1, mw_funcValue, mw_coefs, mw_oneValue);

/* convert back to C/C++ double */
double db_funcValue = (double)mw_funcValue(1,1) ;
cout << "The function value at 2.2 is: " << db_funcValue << endl ;

/* free memories */
delete []db_coefs ;

}

```

---

end code

## 12.2 One-Dimensional Polynomial Interpolation

This section describe how to use the functions in the generated libraries to solve an one-dimensional interpolation problem. This function uses polynomial techniques to evaluate value of a function at a desired interpolation point.

**Problem 2** There are two arrays  $\mathbf{X}$  and  $\mathbf{Y}$  have a relationship via a function,  $y = f(x)$ ,  $x \in \mathbf{X}$  and  $y \in \mathbf{Y}$ .

**input**     Array  $\mathbf{X} = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$   
               Array  $\mathbf{Y} = \{ 6.8, 24.6, 50.2, 74, 140.8, 280.5, 321.4, 428.6 \}$

**output**    Finding the interpolation function value at  $x = a$ , where  $a = 2.1$

The following is the code to solve Problem 2. In the code, you will use the function `mlfMyinterp1(..)` or `myinterp1(..)` with the default method (liner method) to solve the problem. To learn more about other possible methods see the MATLAB function `interp1(..)` in [6].

## A. FOR C SHARED LIBRARY

Listing code

---

```
void Test::OneDimensionInterpolation() {

double db_X[8] = { 1, 2, 3, 4, 5, 6, 7, 8 } ;
double db_Y[8] = { 6.8, 24.6, 50.2, 74, 140.8, 280.5, 321.4, 428.6 };

double db_oneValue = 2.1 ;

/* step 1 : declare mxArray variables */
mxArray *mx_X      = NULL ;
mxArray *mx_Y      = NULL ;

mxArray *mx_oneValue = NULL ;
mxArray *mx_funcValue = NULL ;

/* step 2 : assign memory */
int vectorSize = 6 ;
mx_X          = mxCreateDoubleMatrix(1, vectorSize, mxREAL) ;
mx_Y          = mxCreateDoubleMatrix(1, vectorSize, mxREAL) ;

mx_oneValue = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_funcValue= mxCreateDoubleMatrix(1, 1, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_vectorReal (db_X, mx_X) ;
double2mxArray_vectorReal (db_Y, mx_Y) ;
```

```

double2mxArray_scalarReal (db_oneValue, mx_oneValue) ;

/* step 4 : call an implemental function */
mlfMyinterp1(1, &mx_funcValue, mx_X, mx_Y, mx_oneValue);

/* step 5 : convert back to C/C++ double */

double db_funcValue = mxArray2double_scalarReal(mx_funcValue) ;
cout << "The function value at 2.1 is: " << db_funcValue << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_X ) ;
mxDestroyArray(mx_Y ) ;

mxDestroyArray(mx_oneValue ) ;
mxDestroyArray(mx_funcValue);

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

Listing code

---

```

void Test::OneDimensionInterpolation() {

double db_X[8] = { 1, 2, 3, 4, 5, 6, 7, 8 } ;
double db_Y[8] = { 6.8, 24.6, 50.2, 74, 140.8, 280.5, 321.4, 428.6 };

int vectorSize = 6 ;

/* declare mxArray variables */
mxArray mw_X(1, vectorSize, mxDOUBLE_CLASS) ;
mw_X.SetData(db_X, vectorSize) ;

mxArray mw_Y(1, vectorSize, mxDOUBLE_CLASS) ;
mw_Y.SetData(db_Y, vectorSize) ;

mxArray mw_oneValue(1, 1, mxDOUBLE_CLASS) ;

```

```

mw_oneValue(1,1) = 2.1 ;

/* call an implemental function */
mwArray mw_funcValue(1, 1, mxDOUBLE_CLASS) ;
myinterp1(1, mw_funcValue, mw_X, mw_Y, mw_oneValue);

/* convert back to C/C++ double */

double db_funcValue = (double) mw_funcValue(1,1) ;
cout << "The function value at 2.1 is: " << db_funcValue << endl ;

}

```

---

end code

## 12.3 Two-Dimensional Polynomial Interpolation for Grid Points

This section describe how to use the functions in the generated libraries to solve a two-dimensional interpolation problem. This function uses polynomial techniques to evaluate value of a function at a desired interpolation point.

**Problem 3** There are three matrixes,  $\mathbf{x}$ ,  $\mathbf{y}$ , and  $\mathbf{z}$ , that have a relationship via a function,  $z = f(x, y)$ ,  $x \in \mathbf{x}$ ,  $y \in \mathbf{y}$ , and  $z \in \mathbf{z}$ .

**input** Grid points (x,y).

Representation of these grid points are two matrixes:

matrix  $\mathbf{x}$  contains the x-direction values of all grid points

matrix  $\mathbf{y}$  contains the y-direction values of all grid points

Matrix  $\mathbf{z}$  contains the function values  $z(x, y)$  of all grid points

(the values of matrixes  $\mathbf{x}$ ,  $\mathbf{y}$ , and  $\mathbf{z}$  are shown in the next page)

**output** Finding the interpolation function value  $z(x, y)$  at a particular point,  
( $x = a = 2.3$ ,  $y = b = 0.7$ )

### A. Assigning values for a matrix

Suppose that you have grid points as in Fig. 12.1.

The matrix  $\mathbf{x}$ , which contains the x-direction values for all grid points, is:

							( from left to right )
-3	-2	-1	0	1	2	3	from point 1 to 7
-3	-2	-1	0	1	2	3	from point 8 to 14
-3	-2	-1	0	1	2	3	from point 15 to 21
-3	-2	-1	0	1	2	3	from point 22 to 28
-3	-2	-1	0	1	2	3	from point 29 to 35
-3	-2	-1	0	1	2	3	from point 36 to 42
-3	-2	-1	0	1	2	3	from point 43 to 49

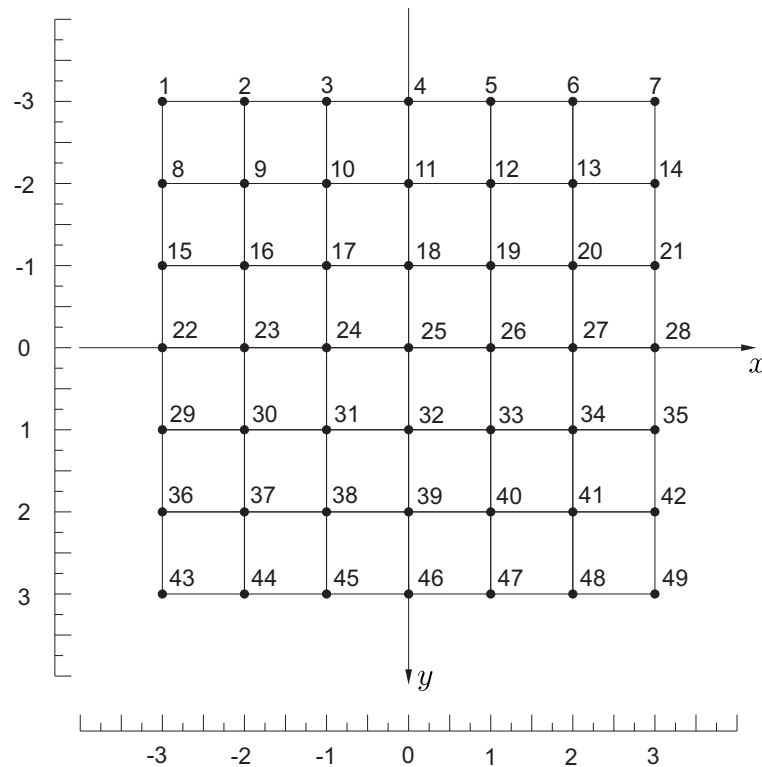


Figure 12.1: Grid points



The matrix  $\mathbf{Y}$ , which contains the y-direction values of all grid points, is:

( from left to right )							
-3	-3	-3	-3	-3	-3	-3	from point 1 to 7
-2	-2	-2	-2	-2	-2	-2	from point 8 to 14
-1	-1	-1	-1	-1	-1	-1	from point 15 to 21
0	0	0	0	0	0	0	from point 22 to 28
1	1	1	1	1	1	1	from point 29 to 35
2	2	2	2	2	2	2	from point 36 to 42
3	3	3	3	3	3	3	from point 43 to 49

The matrix  $\mathbf{z}$ , which contains the function values  $z(x, y)$  for all grid points, is:

( from left to right )							
0.0001	0.0034	-0.0299	-0.2450	-0.1100	-0.0043	-0.0000	(point 1 - 7)
0.0007	0.0468	-0.5921	-4.7596	-2.1024	-0.0616	0.0004	(point 8 -14)
0.0088	-0.1301	1.8559	-0.7239	-0.2729	0.4996	0.0130	(point 15-21)
0.0365	-1.3327	-1.6523	0.9810	2.9369	1.4122	0.0331	(point 22-28)
0.0137	-0.4808	0.2289	3.6886	2.4338	0.5805	0.0125	(point 29-35)
0.0000	0.0797	2.0967	5.8591	2.2099	0.1328	0.0013	(point 36-42)
0.0000	0.0053	0.1099	0.2999	0.1107	0.0057	0.0000	(point 43-49)

## B. Programming code

The following is the code to solve Problem 3. In the code, you will use the generated function `mlfMyinterp2(..)` or `myinterp2(..)` with the *cubic* method to solve the problem. To learn more about other possible methods see the MATLAB function `interp2(..)` in [6]. The procedure of this code is:

1. Assign values for matrix  $\mathbf{x}$  and matrix  $\mathbf{y}$
2. Assign values for matrix  $\mathbf{z}$
3. Call the two-dimensional interpolation function to evaluate the value at the point  $(a, b)$

## A. FOR C SHARED LIBRARY

Listing code

---

```

void Test::TwoDimensionsInterpolation() {

/* function values z at 47 points, (x_i, y_j) are : */
/* matrix z values*/

double z[7][7] = {
{ 0.0001,    0.0034,   -0.0299,   -0.2450,   -0.1100,   -0.0043,    0.0000 } ,\
{ 0.0007,    0.0468,   -0.5921,   -4.7596,   -2.1024,   -0.0616,    0.0004 } ,\
{ -0.0088,   -0.1301,    1.8559,   -0.7239,   -0.2729,    0.4996,    0.0130 } ,\
{ -0.0365,   -1.3327,   -1.6523,    0.9810,    2.9369,    1.4122,    0.0331 } ,\
{ -0.0137,   -0.4808,    0.2289,    3.6886,    2.4338,    0.5805,    0.0125 } ,\
{ 0.0000,    0.0797,    2.0967,    5.8591,    2.2099,    0.1328,    0.0013 } ,\
{ 0.0000,    0.0053,    0.1099,    0.2999,    0.1107,    0.0057,    0.0000 } };

int i, j;
int nSize = 7 ;
double db_vectorstep[3] = {-3, 1, 3} ;
double db_a = 2.3 ;
double db_b = 0.7 ;

int row = nSize ;
int col = nSize ;

/* step 1 : declare mxArray variables */
mxArray *mx_a          = NULL ;
mxArray *mx_b          = NULL ;
mxArray *mx_interp2z   = NULL ;

mxArray *mx_x          = NULL ;
mxArray *mx_y          = NULL ;
mxArray *mx_z          = NULL ;

mxArray *mx_vectorstepx = NULL ;
mxArray *mx_vectorstepy = NULL ;

```

```

mxArray *mx_method ;

/* step 2 : assign memory */
mx_a      = mxCreateDoubleMatrix(1, 1, mxREAL)    ;
mx_b      = mxCreateDoubleMatrix(1, 1, mxREAL)    ;
mx_interp2z = mxCreateDoubleMatrix(1, 1, mxREAL)    ;

mx_x      = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_y      = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_z      = mxCreateDoubleMatrix(row, col, mxREAL) ;

mx_vectorstepx = mxCreateDoubleMatrix(1, 3, mxREAL) ;
mx_vectorstepy = mxCreateDoubleMatrix(1, 3, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal (db_a, mx_a) ;
double2mxArray_scalarReal (db_b, mx_b) ;

/* same for two step-vectors */
double2mxArray_vectorReal (db_vectorstep, mx_vectorstepx) ;
double2mxArray_vectorReal (db_vectorstep, mx_vectorstepy) ;

double2mxArray_matrixReal (&z[0][0], mx_z) ;

/* step 4 : call implemental functions */
/* create values for the matrix x and matrix y */
mlfMymeshgrid(2, &mx_x, &mx_y, mx_vectorstepx, mx_vectorstepy) ;

const char  interp2method[] = "cubic";
mx_method = mxCreateString(interp2method);

mlfMyinterp2(1, &mx_interp2z, mx_x, mx_y, mx_z, mx_a, mx_b, mx_method);

/* step 5 : convert back to C/C++ double */

double **db_x = new double*[row] ;
double **db_y = new double*[row] ;

for(i=0; i<row; i++) {

```

```

        db_x[i] = new double[col] ;
        db_y[i] = new double[col] ;
    }

    mxArray2double_matrixReal(mx_x, db_x) ;
    mxArray2double_matrixReal(mx_y, db_y) ;

    /* print out */
    cout << "Matrix x : " << endl ;
    printMatrix(db_x, row, col) ;
    printMatrix(db_y, row, col) ;

    cout << "Matrui y : " << endl ;
    for (i=0; i<row; i++) {
        for (j=0; j<col; j++) {
            cout << db_y[i][j] << "\t" ;
        }
        cout << endl ;
    }

    double db_interp2z = mxArray2double_scalarReal(mx_interp2z) ;
    cout << "Interpolation in two dimensions with cubic method " << endl;
    cout << " z = " << db_interp2z << endl ;

    /* step 6 : free memories */
    mxDestroyArray(mx_a ) ;
    mxDestroyArray(mx_b ) ;
    mxDestroyArray(mx_interp2z ) ;

    mxDestroyArray(mx_x ) ;
    mxDestroyArray(mx_y ) ;
    mxDestroyArray(mx_z ) ;

    mxDestroyArray(mx_vectorstepx);
    mxDestroyArray(mx_vectorstepy);

    mxDestroyArray(mx_method);

```

```

delete[] db_x ;
delete[] db_y ;

}

```

---

end code

---

## B. FOR C++ SHARED LIBRARY

Listing code

---

```

void Test::TwoDimensionsInterpolation() {

/* function values z at 47 points, (x_i, y_j) are : */
/* matrix z values*/

double z[7][7] = {
{ 0.0001,    0.0034,   -0.0299,   -0.2450,   -0.1100,   -0.0043,    0.0000 } ,\
{ 0.0007,    0.0468,   -0.5921,   -4.7596,   -2.1024,   -0.0616,    0.0004 } ,\
{ -0.0088,   -0.1301,    1.8559,   -0.7239,   -0.2729,    0.4996,    0.0130 } ,\
{ -0.0365,   -1.3327,   -1.6523,    0.9810,    2.9369,    1.4122,    0.0331 } ,\
{ -0.0137,   -0.4808,    0.2289,    3.6886,    2.4338,    0.5805,    0.0125 } ,\
{ 0.0000,    0.0797,    2.0967,    5.8591,    2.2099,    0.1328,    0.0013 } ,\
{ 0.0000,    0.0053,    0.1099,    0.2999,    0.1107,    0.0057,    0.0000 } };

int nSize = 7 ;
double db_vectorstep[3] = {-3, 1, 3} ;

int row = nSize ;
int col = nSize ;

/* declare mwArray variables */
mwArray mw_a(1, 1, mxDOUBLE_CLASS) ;
mw_a(1,1) = 2.3 ;

mwArray mw_b(1, 1, mxDOUBLE_CLASS) ;
mw_b(1,1) = 0.7 ;

mwArray mw_x(row, col, mxDOUBLE_CLASS) ;
mwArray mw_y(row, col, mxDOUBLE_CLASS) ;

```

```

mwArray mw_vectorstepx(1, 3, mxDOUBLE_CLASS) ;
mwArray mw_vectorstepy(1, 3, mxDOUBLE_CLASS) ;

/* same for two step-vectors */
mw_vectorstepx.SetData(db_vectorstep, 3) ;
mw_vectorstepy.SetData(db_vectorstep, 3) ;

/* convert C/C++ double to mxArray */
mwArray mw_z = double2mwArray_matrixReal(&z[0][0], row, col) ;

/* call implemental functions */
/* create values for the matrix x and matrix y */
mymeshgrid(2, mw_x, mw_y, mw_vectorstepx, mw_vectorstepy) ;

mwArray mw_method("cubic") ;
mwArray mw_interp2z(1, 1, mxDOUBLE_CLASS) ;

myinterp2(1, mw_interp2z, mw_x, mw_y, mw_z, mw_a, mw_b, mw_method);

/* print out to see */
cout << "Matriy x : " << endl ;
std::cout << mw_x << std::endl ;

cout << "Matriy y : " << endl ;
std::cout << mw_y << std::endl ;

/* convert back to C/C++ double */
double db_interp2z = (double) mw_interp2z(1,1) ;
cout << "Interpolation in two dimensions with cubic method " << endl;
cout << " z = " << db_interp2z << endl ;

}

```

---

end code

### Remarks

1. The M-file mymeshgrid.m uses the MATLAB function meshgrid(..) which assigns values for matrixes **x** and **y**. These values are assigned from left to right, and from top to bottom.

In Fig. 12.1, the *y* axis direction is from top to bottom, therefore pay attention when

assigning your data into the matrix  $\mathbf{y}$ .

2. The matrix  $\mathbf{z}$  values will be assigned according to the same rules (left to right, top to bottom) as matrixes  $\mathbf{x}$  and  $\mathbf{y}$  (Fig. 12.2), therefore to avoid mistakes and to have the convenience of visibility, you can set up your matrix data as in Fig. 12.1 (**y axis direction from top to bottom**).

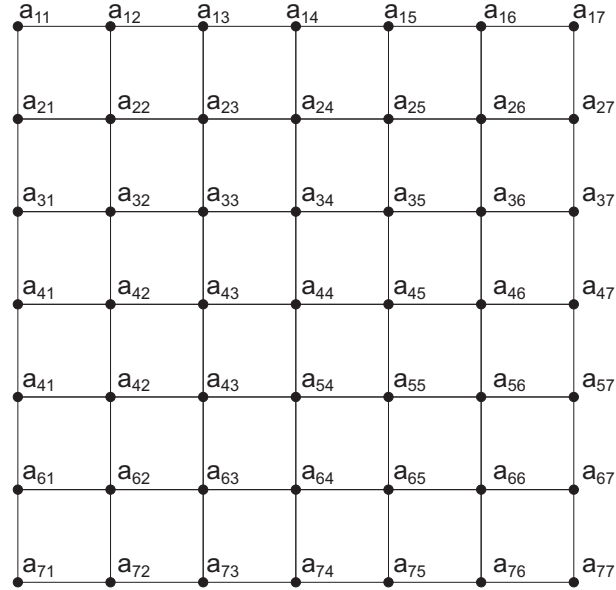


Figure 12.2: A matrix form for grid points in two-dimensional interpolation

3. To receive a better solution in the two-dimensional interpolation you can create a fine grid by using the function `mygriddata(..)` (see the M-file `mygriddata.m` in the beginning of this chapter). This functions uses the MATLAB `griddata(..)` function which fits a surface of the form  $z = f(x, y)$  to the data in the spaced vectors  $(x, y, z)$ . For more information on this function, refer to the MATLAB manual [6]. The following code finds the fine solution of Problem 3 by using the function `mygriddata(..)`.

## A. FOR C SHARED LIBRARY

Listing code

---

```
void Test::TwoDimensionsInterpolationFineSolution() {

/* function values z at 47 points, (x_i, y_j) are : */
/* matrix z values*/
```

```

double z[7][7] = {
{ 0.0001,    0.0034,   -0.0299,   -0.2450,   -0.1100,   -0.0043,    0.0000 } ,\
{ 0.0007,    0.0468,   -0.5921,   -4.7596,   -2.1024,   -0.0616,    0.0004 } ,\
{ -0.0088,   -0.1301,    1.8559,   -0.7239,   -0.2729,    0.4996,    0.0130 } ,\
{ -0.0365,   -1.3327,   -1.6523,    0.9810,    2.9369,    1.4122,    0.0331 } ,\
{ -0.0137,   -0.4808,    0.2289,    3.6886,    2.4338,    0.5805,    0.0125 } ,\
{ 0.0000,    0.0797,    2.0967,    5.8591,    2.2099,    0.1328,    0.0013 } ,\
{ 0.0000,    0.0053,    0.1099,    0.2999,    0.1107,    0.0057,    0.0000 } };

int nSize = 7 ;

double db_vectorstep[3]    = {-3, 1 , 3} ;
double db_finevectorstep[3] = {-3, 0.2, 3} ;

double db_a = 2.3 ;
double db_b = 0.7 ;

int row = nSize ;
int col = nSize ;

/* step 1 : declare mxArray variables */
mxArray *mx_a          = NULL ;
mxArray *mx_b          = NULL ;
mxArray *mx_interp2z   = NULL ;

mxArray *mx_x          = NULL ;
mxArray *mx_y          = NULL ;
mxArray *mx_z          = NULL ;

mxArray *mx_vectorstepx = NULL ;
mxArray *mx_vectorstepy = NULL ;

mxArray *mx_method ;

/* step 1a : declare mxArray variables */
mxArray *mx_finex      = NULL ;
mxArray *mx_finey      = NULL ;
mxArray *mx_finez      = NULL ;

mxArray *mx_finerow     = NULL ;

```



```

mxArray *mx_finecol      = NULL ;

mxArray *mx_finevectorstepx = NULL ;
mxArray *mx_finevectorstepy = NULL ;

/* step 2 : assign memory */
mx_a      = mxCreateDoubleMatrix(1, 1, mxREAL)      ;
mx_b      = mxCreateDoubleMatrix(1, 1, mxREAL)      ;
mx_interp2z = mxCreateDoubleMatrix(1, 1, mxREAL)      ;

mx_x      = mxCreateDoubleMatrix(row, col, mxREAL)    ;
mx_y      = mxCreateDoubleMatrix(row, col, mxREAL)    ;
mx_z      = mxCreateDoubleMatrix(row, col, mxREAL)    ;

mx_vectorstepx = mxCreateDoubleMatrix(1, 3, mxREAL)    ;
mx_vectorstepy = mxCreateDoubleMatrix(1, 3, mxREAL)    ;

/* step 2a : assign memory */
mx_finerow = mxCreateDoubleMatrix(1, 1, mxREAL)      ;
mx_finecol = mxCreateDoubleMatrix(1, 1, mxREAL)      ;

mx_finevectorstepx = mxCreateDoubleMatrix(1, 3, mxREAL)    ;
mx_finevectorstepy = mxCreateDoubleMatrix(1, 3, mxREAL)    ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal (db_a, mx_a) ;
double2mxArray_scalarReal (db_b, mx_b) ;

/* same for two step-vectors */
double2mxArray_vectorReal (db_vectorstep, mx_vectorstepx) ;
double2mxArray_vectorReal (db_vectorstep, mx_vectorstepy) ;

double2mxArray_matrixReal (&z[0][0], mx_z) ;

/* step 3a : convert C/C++ double to mxArray */
double2mxArray_vectorReal (db_finevectorstep, mx_finevectorstepx) ;
double2mxArray_vectorReal (db_finevectorstep, mx_finevectorstepy) ;

/* get size for fine matrixes */

```

```

mlfMyfinemeshgrid(2, &mx_finerow, &mx_finecol, mx_finevectorstepx, mx_finevectorstepy) ;

int finerow = (int) mxArray2double_scalarReal(mx_finerow) ;
int finecol = (int) mxArray2double_scalarReal(mx_finecol) ;

cout << " fine row = " << finerow << endl ;
cout << " fine col = " << finecol << endl ;

mx_finex   = mxCreateDoubleMatrix(finerow, finecol, mxREAL)   ;
mx_finey   = mxCreateDoubleMatrix(finerow, finecol, mxREAL)   ;
mx_finez   = mxCreateDoubleMatrix(finerow, finecol, mxREAL)   ;

/* step 4 : call implemental functions */
/* create values for the matrix x and matrix y */
mlfMymeshgrid(2, &mx_x, &mx_y, mx_vectorstepx, mx_vectorstepy) ;

/* create values for the fine matrix x and fine matrix y */
mlfMymeshgrid(2, &mx_finex, &mx_finey, mx_finevectorstepx, mx_finevectorstepy) ;

/* get a fine matrix mx_finez from mx_z */
mlfMygriddata(1, &mx_finez, mx_x, mx_y, mx_z, mx_finex, mx_finey);

const char  interp2method[] = "cubic"      ;
mx_method = mxCreateString(interp2method)  ;

mlfMyinterp2(1, &mx_interp2z, mx_finex, mx_finey, mx_finez, mx_a, mx_b, mx_method);

/* step 5 : convert back to C/C++ double */
double db_interp2finez = mxArray2double_scalarReal(mx_interp2z) ;
cout << "Interpolation in two dimensions " ;
cout << "with cubic method and a fine grid " << endl;
cout << " z = " << db_interp2finez << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_a )      ;
mxDestroyArray(mx_b )      ;
mxDestroyArray(mx_interp2z ) ;

```

```

mxDestroyArray(mx_x )    ;
mxDestroyArray(mx_y )    ;
mxDestroyArray(mx_z )    ;

mxDestroyArray(mx_vectorstepx);
mxDestroyArray(mx_vectorstepy);

mxDestroyArray(mx_method);

/* for 1a */
mxDestroyArray(mx_findex) ;
mxDestroyArray(mx_finey)  ;
mxDestroyArray(mx_finez)  ;

mxDestroyArray(mx_finerow) ;
mxDestroyArray(mx_finecol) ;

mxDestroyArray(mx_finevectorstepx) ;
mxDestroyArray(mx_finevectorstepy) ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

Listing code

---

```

void Test::TwoDimensionsInterpolationFineSolution() {
/* function values z at 47 points, (x_i, y_j) are : */
/* matrix z values*/

double z[7][7] = {
{ 0.0001,    0.0034,   -0.0299,   -0.2450,   -0.1100,   -0.0043,    0.0000 } ,\
{ 0.0007,    0.0468,   -0.5921,   -4.7596,   -2.1024,   -0.0616,    0.0004 } ,\
{ -0.0088,   -0.1301,    1.8559,   -0.7239,   -0.2729,    0.4996,    0.0130 } ,\
{ -0.0365,   -1.3327,   -1.6523,    0.9810,    2.9369,    1.4122,    0.0331 } ,\
{ -0.0137,   -0.4808,    0.2289,    3.6886,    2.4338,    0.5805,    0.0125 } ,\
{ 0.0000,    0.0797,    2.0967,    5.8591,    2.2099,    0.1328,    0.0013 } ,\
{ 0.0000,    0.0053,    0.1099,    0.2999,    0.1107,    0.0057,    0.0000 } };

```

```

int nSize = 7 ;
int row = nSize ;
int col = nSize ;

mwArray mw_a(1, 1, mxDOUBLE_CLASS) ;
mw_a(1,1) = 2.3 ;

mwArray mw_b(1, 1, mxDOUBLE_CLASS) ;
mw_b(1,1) = 0.7 ;

/* convert C/C++ double to mxArray */
mwArray mw_z = double2mwArray_matrixReal(&z[0][0], row, col) ;

double db_vectorstep[3]      = {-3, 1, 3} ;
double db_finevectorstep[3] = {-3, 0.2, 3} ;

/* declare mxArray variables */
mwArray mw_vectorstepx(1, 3, mxDOUBLE_CLASS) ;
mwArray mw_vectorstepy(1, 3, mxDOUBLE_CLASS) ;
    /* same for two step-vectors */
mw_vectorstepx.SetData(db_vectorstep, 3) ;
mw_vectorstepy.SetData(db_vectorstep, 3) ;

mwArray mw_finevectorstepx(1, 3, mxDOUBLE_CLASS) ;
mwArray mw_finevectorstepy(1, 3, mxDOUBLE_CLASS) ;
    /* same for two fine step-vectors */
mw_finevectorstepx.SetData(db_finevectorstep, 3) ;
mw_finevectorstepy.SetData(db_finevectorstep, 3) ;

mwArray mw_finerow(1, 1, mxDOUBLE_CLASS) ;
mwArray mw_finecol(1, 1, mxDOUBLE_CLASS) ;

/* get size for fine matrixes */
myfinemeshgrid(2, mw_finerow, mw_finecol, mw_finevectorstepx, mw_finevectorstepy) ;

int finerow = (int) mw_finerow(1,1) ;
int finecol = (int) mw_finecol(1,1) ;

```

```

cout << " fine row = " << finerow << endl ;
cout << " fine col = " << finecol << endl ;

mwArray mw_finex(finerow, finecol, mxDOUBLE_CLASS) ;
mwArray mw_finey(finerow, finecol, mxDOUBLE_CLASS) ;
mwArray mw_finez(finerow, finecol, mxDOUBLE_CLASS) ;

/* step 4 : call implemental functions */
/* create values for the matrix x and matrix y */
mwArray mw_x(row, col, mxDOUBLE_CLASS) ;
mwArray mw_y(row, col, mxDOUBLE_CLASS) ;

mymeshgrid(2, mw_x, mw_y, mw_vectorstepx, mw_vectorstepy) ;

/* create values for the fine matrix x and fine matrix y */
mymeshgrid(2, mw_finex, mw_finey, mw_finevectorstepx, mw_finevectorstepy) ;

/* get a fine matrix mw_finez from mw_z */
mygriddata(1, mw_finez, mw_x, mw_y, mw_z, mw_finex, mw_finey);

mwArray mw_method("cubic") ;
mwArray mw_interp2z(1, 1, mxDOUBLE_CLASS) ;

myinterp2(1, mw_interp2z, mw_finex, mw_finey, mw_finez, mw_a, mw_b, mw_method);

/* convert back to C/C++ double */
double db_interp2finez = (double) mw_interp2z(1,1) ;
cout << "Interpolation in two dimensions " ;
cout << "with cubic method and a fine grid " << endl;
cout << " z = " << db_interp2finez << endl ;

}

```

## 12.4 Two-Dimensional Polynomial Interpolation for Non-Grid Points

This section describe how to use the function `mlfMyinterp2(..)` in the generated *curvefittinglib* library or `myinterp2(..)` in the generated *cppcurvefittinglib* library to solve the problem of two-dimensional polynomial interpolation for non-grid points.

### Problem 4

#### input

Suppose that you have values of experimental data that are not in grid points. At each value  $x_i$  there are the corresponding values of array  $\mathbf{y}_i[]$  and array  $\mathbf{z}_i[]$  as shown in Fig. 12.3.

The values  $x$ ,  $y$ , and  $z$  of non-grid points are in Table 12.1

#### output

Finding the interpolation function value at a particular point  $(a, b)$ ,

where  $a = 2.3$ ,  $b = 0.7$

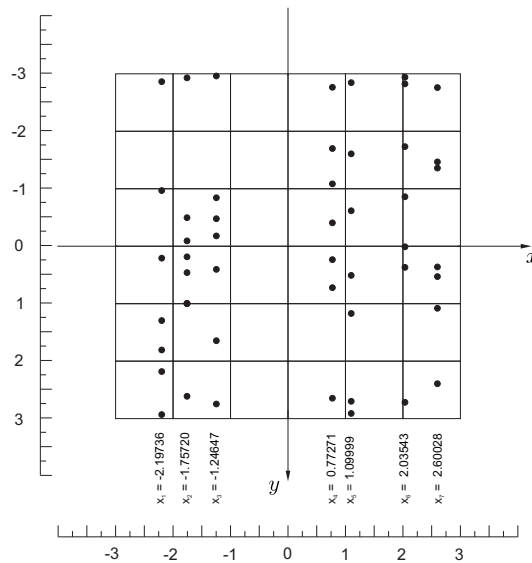


Figure 12.3: Non-grid points in Problem 4

The following is the code to solve Problem 4 by using the *cubic* method in the MATLAB function `interp2(..)` through the M-file `myinterp2.m`. See the MATLAB manual [6] for using other methods in the function `interp2(..)`.

The steps of the procedure for the programming code are:

1. Assign values (from Table 12.1) for matrixes  $\mathbf{y}$  and  $\mathbf{z}$  by assigning values in column arrays.

at $x_1 = -2.19736$		at $x_2 = -1.75720$		at $x_3 = -1.24647$		at $x_4 = 0.77271$	
$y_1$	$z_1$	$y_2$	$z_2$	$y_3$	$z_3$	$y_4$	$z_4$
-2.92946	0.0001	-2.61131	0.0034	-2.74558	-0.0299	-2.64490	-0.2450
-2.18089	0.0007	-0.99629	0.0468	-1.64430	-0.5921	-0.72109	-4.7596
-1.80517	-0.0088	-0.45902	-0.1301	-0.40256	1.8559	-0.23343	-0.7239
-1.29355	-0.0365	-0.18465	-1.3327	0.17894	-1.6523	0.40697	0.9810
-0.20766	-0.0137	0.09307	-0.4808	0.47884	0.2289	1.08507	3.6886
0.96866	0.0080	0.49675	0.0797	0.84316	2.0967	1.69997	5.8591
2.86339	0.0000	2.93001	0.0053	2.96219	0.1099	2.76526	0.2999

at $x_5 = 1.09999$		at $x_6 = 2.03543$		at $x_7 = 2.60028$	
$y_5$	$z_5$	$y_6$	$z_6$	$y_7$	$z_7$
-2.90994	-0.1100	-2.71622	-0.0043	-2.76059	-0.0000
-2.69839	-2.1024	-0.36805	-0.0616	-1.07979	0.0004
-1.17001	-0.2729	-0.01013	0.4996	-0.52828	0.0130
-0.50775	2.9369	0.86095	1.4122	-0.36045	0.0331
0.61721	2.4338	1.73317	0.5805	1.35979	0.0125
1.60770	2.2099	2.82507	0.1328	1.46739	0.0013
2.84620	0.1107	2.94050	0.0057	2.39232	0.0000

Table 12.1: Value  $x$ ,  $y$ , and  $z$  at the non-grid points in Fig. 12.3

2. Choose new grid points as in Fig. 12.4, in which new values of new grid points in the  $x$  direction are the same of old grids, as follows:

$$\begin{array}{llll} x_1 = -2.19736 & x_2 = -1.75720 & x_3 = -1.24647 & x_4 = 0.77271 \\ x_5 = 1.09999 & x_6 = 2.03543 & x_7 = 2.60028 & \end{array}$$

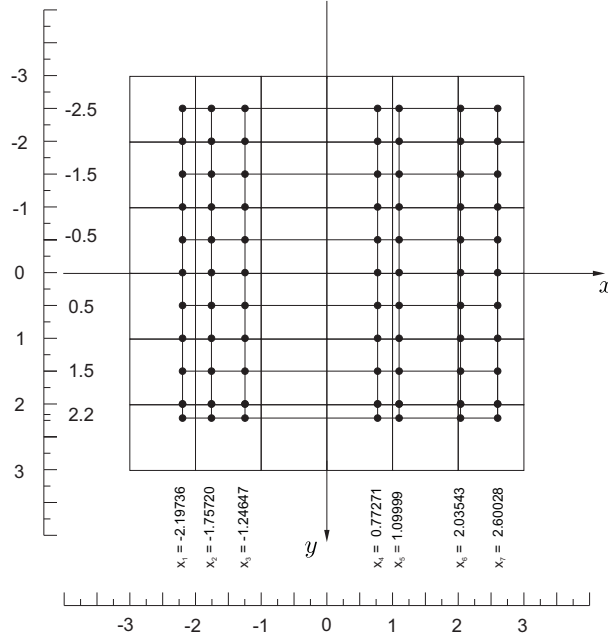


Figure 12.4: New grid points

and new values of new grid points in  $y$  direction are :

$$\begin{array}{llll} y_1 = -2.5 & y_2 = -2.0 & y_3 = -1.5 & y_4 = -1.0 \\ y_5 = -0.5 & y_6 = 0.0 & y_7 = 0.5 & y_8 = 1.0 \\ y_9 = 1.5 & y_{10} = 2.0 & y_{11} = 2.2 & \end{array}$$

Note that these new 11  $y$  values ( $y_{new}$ ) are not outside the original values in each  $y$  array in Table 12.1 ( $y_{i,min} \leq y_{new} \leq y_{i,max}$ ,  $i = 1, 2, \dots, 7$ ). There are 77 ( $= 7 \times 11$ ) new grid points that were created.

3. Calculate  $z$  values for the 77 new grid points by calculating its columns via the one-dimensional interpolation function. This calculation is the one-dimension interpolation of  $y$  and  $z$  in each column (note  $x$  is constant at each current calculating column  $y$  and  $z$ ).

```
At each column, for each number in row {
    z[i] <--- oneDimensionInperpolation(Old_y, Old_z)
}
```



Actually, this calculation is the calculation of the new  $z$  from the old pair array  $(y, z)$ . The figure 12.5 shows the new  $z$  from  $(y, z)$  of the first pair  $(y_1, z_1)$  of Table 12.1.

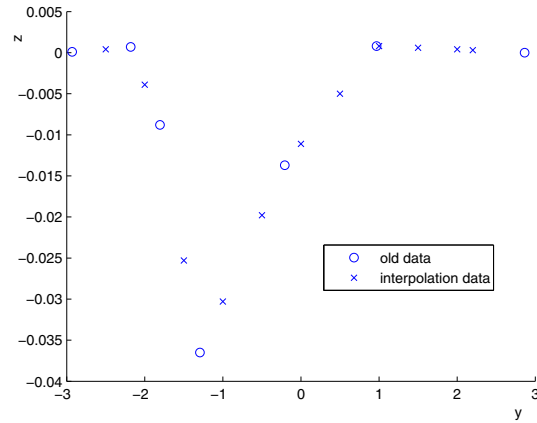


Figure 12.5: New interpolation points from the first pair

4. Generate  $x$  and  $y$  values for the 77 new grid points (assign values for new matrixes  $\mathbf{x}$  and  $\mathbf{y}$ ). The new matrixes are matrixes with 7 rows  $\times$  11 columns.
5. Call the two dimensional interpolation function to evaluate the value at a point  $(a, b)$ .

The following is the code to solve Problem 4 by using the functions in the libraries with the *cubic* method.

## A. FOR C SHARED LIBRARY

Listing code

---

```
void Test::TwoDimensionsInterpolationInNonGrid() {

/* solving a problem in which experimental data are not in grids */
/* and we would like to get interpolation at a point (xi,yj) */

/* step 1 in the procedure */
/* from the experiment data in the table, fill the matrix */

double db_yoldmatrix[7][7] = {
{-2.92946,  -2.61131,  -2.74558,  -2.64490,  -2.90994,  -2.71622,  -2.76059 } ,\
{-2.18089,  -0.99629,  -1.64430,  -0.72109,  -2.69839,  -0.36805,  -1.07979 } ,\
{-1.80517,  -0.45902,  -0.40256,  -0.23343,  -1.17001,  -0.01013,  -0.52828 } ,\
{-1.29355,  -0.18465,  0.17894,  0.40697,  -0.50775,  0.86095,  -0.36045 } ,\
{-0.20766,  0.09307,  0.47884,  1.08507,  0.61721,  1.73317,  1.35979 } ,\
```

```
{ 0.96866, 0.49675, 0.84316, 1.69997, 1.60770, 2.82507, 1.46739 } ,\
{ 2.86339, 2.93001, 2.96219, 2.76526, 2.84620, 2.94050, 2.39232 } };
```

```
double db_zoldmatrix[7][7] = {
{ 0.00010, 0.00340, -0.02990, -0.24500, -0.11000, -0.00430, 0.00000 } ,\
{ 0.00070, 0.04680, -0.59210, -4.75960, -2.10240, -0.06160, 0.00040 } ,\
{ -0.00880, -0.13010, 1.85590, -0.72390, -0.27290, 0.49960, 0.01300 } ,\
{ -0.03650, -1.33270, -1.65230, 0.98100, 2.93690, 1.41220, 0.03310 } ,\
{ -0.01370, -0.48080, 0.22890, 3.68860, 2.43380, 0.58050, 0.01250 } ,\
{ 0.00080, 0.07970, 2.09670, 5.85910, 2.20990, 0.13280, 0.00130 } ,\
{ 0.00000, 0.00530, 0.10990, 0.29990, 0.11070, 0.00570, 0.00000 } };
```

```
/* step 2 in the procedure */
```

```
double db_oldvectorx[7] = {-2.19736, -1.75720, -1.24647, 0.77271, 1.09999, 2.03543, 2.60028};
double db_newcoly[11] = {-2.5, -2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.2 } ;
```

```
int row      = 7      ;
int col      = 7      ;
int newrow   = 11     ;
```

```
int i, j, k;
double db_a = 2.3 ;
double db_b = 0.7 ;
```

```
mxArray *mx_oldcoly      = NULL ;
mxArray *mx_oldcolz      = NULL ;
```

```
mxArray *mx_newcoly      = NULL ;
mxArray *mx_newcolz      = NULL ;
```

```
mxArray *mx_newmatrixx   = NULL ;
mxArray *mx_newmatrixy   = NULL ;
mxArray *mx_newmatrixz   = NULL ;
```

```
mxArray *mx_a = NULL ;
mxArray *mx_b = NULL ;
mxArray *mx_interp2z = NULL ;
```

```

mxArray *mx_method      ;

mx_oldcoly  = mxCreateDoubleMatrix(row, 1 , mxREAL) ;
mx_oldcolz  = mxCreateDoubleMatrix(row, 1 , mxREAL) ;

mx_newcoly  = mxCreateDoubleMatrix(newrow, 1 , mxREAL) ;
mx_newcolz  = mxCreateDoubleMatrix(newrow, 1 , mxREAL) ;

mx_newmatrixx  = mxCreateDoubleMatrix(newrow, col , mxREAL)      ;
mx_newmatrixy  = mxCreateDoubleMatrix(newrow, col , mxREAL)      ;
mx_newmatrixz  = mxCreateDoubleMatrix(newrow, col , mxREAL)      ;

mx_a = mxCreateDoubleMatrix(1, 1 , mxREAL) ;
mx_b = mxCreateDoubleMatrix(1, 1 , mxREAL) ;

mx_interp2z = mxCreateDoubleMatrix(1, 1 , mxREAL) ;

/* step 3 in the procedure */

/* ***** */
/* ***** */
/* ***** using 1D interpolation to establish a new matrix z *** */
/* run for 7(=col) columns */

double *db_oldcoly = new double [col] ;
double *db_oldcolz = new double [col] ;

// 11=newrow is the number of elements in the new column
double *db_newcolz = new double [11] ;

double **db_newmatrixz = new double* [newrow] ;
for (i=0; i<newrow; i++) {
    db_newmatrixz[i] = new double [col] ;
}

for (j=0; j<col; j++) {

    /* get a current column for calculation of 1D interpolation */

```

```

    for (i=0; i<row; i++) {
        db_oldcoly[i] = db_yoldmatrix[i][j] ;
        db_oldcolz[i] = db_zoldmatrix[i][j] ;
    }

    /* convert to use the function mlfMyinterp1(..) */
    double2mxArray_vectorReal(db_oldcoly, mx_oldcoly) ;
    double2mxArray_vectorReal(db_oldcolz, mx_oldcolz) ;
    double2mxArray_vectorReal(db_newcoly, mx_newcoly) ;

    /* calculate 1D inperpolation */
    mlfMyinterp1(1, &mx_newcolz, mx_oldcoly, mx_oldcolz, mx_newcoly);

    /* convert back to C/C++ double */
    mxArray2double_vectorReal(mx_newcolz, db_newcolz) ;

    /* fill the each column result to a new matrix z */
    /* each column includes 11(=newcol) elements */
    for(k=0; k<11; k++) {
        db_newmatrixz[k][j] = db_newcolz[k] ;
    }

}

/* ***** finished establishing the new matrix z *** */
/* ***** */
/* ***** */

cout << "New matrix z : " << endl ;
for (i=0; i<newrow; i++) {
    for (j=0; j<col; j++) {
        cout << db_newmatrixz[i][j] << "\t" ;
    }
    cout << endl ;
}

/* step 4 in the procedure */

```

```

/* create the matrix x from vectorx, all rows are same */
double **db_newmatrixx = new double* [newrow] ;
for (i=0; i<newrow; i++) {
    db_newmatrixx[i] = new double [col] ;
}

for (i=0; i<newrow; i++) {
    for (j=0; j<col; j++) {
        db_newmatrixx[i][j] = db_oldvectorx[j] ;
    }
}

cout << "New matrix x : " << endl ;
for (i=0; i<newrow; i++) {
    for (j=0; j<col; j++) {
        cout << db_newmatrixx[i][j] << "\t" ;
    }
    cout << endl ;
}

/* create the matrix y from db_oldcoly, all columns are same */
double **db_newmatrixy = new double* [newrow] ;
for (i=0; i<newrow; i++) {
    db_newmatrixy[i] = new double [col] ;
}

for (j=0; j<col; j++) {
    for (i=0; i<newrow; i++) {
        db_newmatrixy[i][j] = db_newcoly[i] ;
    }
}

cout << "New matrix y : " << endl ;
for (i=0; i<newrow; i++) {
    for (j=0; j<col; j++) {
        cout << db_newmatrixy[i][j] << "\t" ;
    }
    cout << endl ;
}

```

```

/* step 5 in the procedure */
const char  interp2method[] = "cubic"      ;
mx_method = mxCreateString(interp2method)  ;

double2mxArray_scalarReal(db_a, mx_a) ;
double2mxArray_scalarReal(db_b, mx_b) ;

double2mxArray_matrixReal(db_newmatrixx, mx_newmatrixx) ;
double2mxArray_matrixReal(db_newmatrixy, mx_newmatrixy) ;
double2mxArray_matrixReal(db_newmatrixz, mx_newmatrixz) ;

mlfMyinterp2(1, &mx_interp2z, mx_newmatrixx, mx_newmatrixy, mx_newmatrixz,
              mx_a, mx_b, mx_method);

double db_interp2z = mxArray2double_scalarReal(mx_interp2z) ;

cout << "Interpolation in two dimensions " ;
cout << "and experimental data are not in grids " << endl;
cout << " z = " << db_interp2z << endl ;

/* free memories */
mxDestroyArray(mx_oldcoly)  ;
mxDestroyArray(mx_oldcolz)  ;

mxDestroyArray(mx_newcoly)  ;
mxDestroyArray(mx_newcolz)  ;

mxDestroyArray(mx_newmatrixx) ;
mxDestroyArray(mx_newmatrixy) ;
mxDestroyArray(mx_newmatrixz) ;

mxDestroyArray(mx_a) ;
mxDestroyArray(mx_b) ;
mxDestroyArray(mx_interp2z) ;

mxDestroyArray(mx_method);

```

```

delete [] db_oldcoly ;
delete [] db_oldcolz ;

delete [] db_newcolz ;

delete [] db_newmatrixx ;
delete [] db_newmatrixy ;
delete [] db_newmatrixz ;

}

```

---

end code

**Remark:** We also add here the M-file to solve Problem 4 for reference.

```
function interp2z = Interp2InNonGrid(a, b)
```

```
%step 1
```

```

db_yoldmatrix = [
    -2.9295    -2.6113    -2.7456    -2.6449    -2.9099    -2.7162    -2.7606
    -2.1809    -0.9963    -1.6443    -0.7211    -2.6984    -0.3680    -1.0798
    -1.8052    -0.4590    -0.4026    -0.2334    -1.1700    -0.0101    -0.5283
    -1.2936    -0.1847     0.1789     0.4070    -0.5078     0.8609    -0.3604
    -0.2077     0.0931     0.4788     1.0851     0.6172     1.7332     1.3598
     0.9687     0.4968     0.8432     1.7000     1.6077     2.8251     1.4674
     2.8634     2.9300     2.9622     2.7653     2.8462     2.9405     2.3923 ] ;

```

```

db_zoldmatrix = [
    0.00010    0.00340   -0.02990   -0.24500   -0.11000   -0.00430    0.00000 ; ...
    0.00070    0.04680   -0.59210   -4.75960   -2.10240   -0.06160    0.00040 ; ...
   -0.00880   -0.13010    1.85590   -0.72390   -0.27290    0.49960    0.01300 ; ...
   -0.03650   -1.33270   -1.65230    0.98100    2.93690    1.41220    0.03310 ; ...
   -0.01370   -0.48080    0.22890    3.68860    2.43380    0.58050    0.01250 ; ...
    0.00080    0.07970    2.09670    5.85910    2.20990    0.13280    0.00130 ; ...
    0.00000    0.00530    0.10990    0.29990    0.11070    0.00570    0.00000 ] ;

```

```
%step 2
```

```

db_newcoly = [-2.5, -2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.2] ;
db_oldvectorx= [-2.19736, -1.75720, -1.24647, 0.77271, 1.09999, 2.03543, 2.60028] ;

```

```

%step 3
db_newmatrixz = zeros(11, 7) ;

for i = 1:7
db_newmatrixz(:,i) = interp1( db_yoldmatrix(:,i), db_zoldmatrix(:,i), db_newcoly ) ;
db_newmatrixxy(:,i) = db_newcoly ;
end

for i= 1:11 ;
db_newmatrixxx(i,:) = db_oldvectorx ;
end

%step 4
interp2z = interp2(db_newmatrixxx, db_newmatrixxy, db_newmatrixz, a, b, 'cubic') ;

_____ end code _____

```

## B. FOR C++ SHARED LIBRARY

Listing code

---

```

/* ***** */
void Test::TwoDimensionsInterpolationInNonGrid() {

/* solving a problem in which experimental data are not in grids */
/* and we would like to get interpolation at a point (xi,yj) */

/* step 1 in the procedure */
/* from the experiment data in the table, fill the matrix */

double db_yoldmatrix[7][7] = {
{-2.92946,  -2.61131,  -2.74558,   -2.64490,   -2.90994,  -2.71622,  -2.76059 } ,\
{-2.18089,  -0.99629,  -1.64430,   -0.72109,   -2.69839,  -0.36805,  -1.07979 } ,\
{-1.80517,  -0.45902,  -0.40256,   -0.23343,   -1.17001,  -0.01013,  -0.52828 } ,\
{-1.29355,  -0.18465,   0.17894,    0.40697,   -0.50775,   0.86095,  -0.36045 } ,\
{-0.20766,   0.09307,   0.47884,    1.08507,    0.61721,   1.73317,   1.35979 } ,\
{ 0.96866,   0.49675,   0.84316,    1.69997,    1.60770,   2.82507,   1.46739 } ,\
{ 2.86339,   2.93001,   2.96219,    2.76526,    2.84620,   2.94050,   2.39232 } };

```



```

double db_zoldmatrix[7][7] = {
{ 0.00010,  0.00340,  -0.02990, -0.24500,  -0.11000,  -0.00430,  0.00000 } ,\
{ 0.00070,  0.04680,  -0.59210, -4.75960,  -2.10240,  -0.06160,  0.00040 } ,\
{ -0.00880, -0.13010,  1.85590, -0.72390,  -0.27290,  0.49960,  0.01300 } ,\
{ -0.03650, -1.33270,  -1.65230,  0.98100,  2.93690,  1.41220,  0.03310 } ,\
{ -0.01370, -0.48080,  0.22890,  3.68860,  2.43380,  0.58050,  0.01250 } ,\
{ 0.00080,  0.07970,  2.09670,  5.85910,  2.20990,  0.13280,  0.00130 } ,\
{ 0.00000,  0.00530,  0.10990,  0.29990,  0.11070,  0.00570,  0.00000 } };

/* step 2 in the procedure */
double db_oldvectorx[7] = {-2.19736, -1.75720, -1.24647, 0.77271, 1.09999, 2.03543, 2.60028};
double db_newcoly[11] = {-2.5, -2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0, 2.2 } ;

int row      = 7      ;
int col      = 7      ;
int newrow   = 11     ;

int i, j, k;
double db_a = 2.3 ;
double db_b = 0.7 ;

mwArray mw_a(1, 1, mxDOUBLE_CLASS) ;
mw_a(1,1) = db_a ;

mwArray mw_b(1, 1, mxDOUBLE_CLASS) ;
mw_b(1,1) = db_b ;

/* step 3 in the procedure */

/* ***** */
/* ***** */
/* ***** using 1D interpolation to establish a new matrix z *** */
/* run for 7(=col) columns */

double *db_oldcoly = new double [col] ;
double *db_oldcolz = new double [col] ;

// 11=newrow is the number of elements in the new column

```

```

double *db_newcolz = new double [11] ;

double **db_newmatrixz = new double* [newrow] ;
for (i=0; i<newrow; i++) {
    db_newmatrixz[i] = new double [col] ;
}

for (j=0; j<col; j++) {

    /* get a current column for calculation of 1D inperpolation */
    for (i=0; i<row; i++) {
        db_oldcoly[i] = db_yoldmatrix[i][j] ;
        db_oldcolz[i] = db_zoldmatrix[i][j] ;
    }

    /* convert to use the function myinterp1(..) */
    mxArray mw_oldcoly(row, 1, mxDOUBLE_CLASS) ;
    mw_oldcoly.SetData(db_oldcoly, row) ;

    mxArray mw_oldcolz(row, 1, mxDOUBLE_CLASS) ;
    mw_oldcolz.SetData(db_oldcolz, row) ;

    mxArray mw_newcoly(newrow, 1, mxDOUBLE_CLASS) ;
    mw_newcoly.SetData(db_newcoly, newrow) ;

    /* calculate 1D inperpolation */
    mxArray mw_newcolz(newrow, 1, mxDOUBLE_CLASS) ;
    myinterp1(1, mw_newcolz, mw_oldcoly, mw_oldcolz, mw_newcoly);

    /* convert back to C/C++ double */
    mxArray2double_vectorReal(mw_newcolz, db_newcolz) ;

    /* fill the each column result to a new matrix z */
    /* each column includes 11(=newcol) elements */
    for(k=0; k<11; k++) {
        db_newmatrixz[k][j] = db_newcolz[k] ;
    }
}

```

```

}

/* ***** finished establishing the new matrix z *** */

/* ***** */
/* ***** */

cout << "New matrix z : " << endl ;
for (i=0; i<newrow; i++) {
    for (j=0; j<col; j++) {
        cout << db_newmatrixz[i][j] << "\t" ;
    }
    cout << endl ;
}

/* step 4 in the procedure */

/* create the matrix x from vectorx, all rows are same */
double **db_newmatrixx = new double* [newrow] ;
for (i=0; i<newrow; i++) {
    db_newmatrixx[i] = new double [col] ;
}

for (i=0; i<newrow; i++) {
    for (j=0; j<col; j++) {
        db_newmatrixx[i][j] = db_oldvectorx[j] ;
    }
}

cout << "New matrix x : " << endl ;
for (i=0; i<newrow; i++) {
    for (j=0; j<col; j++) {
        cout << db_newmatrixx[i][j] << "\t" ;
    }
    cout << endl ;
}

/* create the matrix y from db_oldcoly, all columns are same */
double **db_newmatrixy = new double* [newrow] ;
for (i=0; i<newrow; i++) {

```

```

        db_newmatrixy[i] = new double [col] ;
    }

    for (j=0; j<col; j++) {
        for (i=0; i<newrow; i++) {
            db_newmatrixy[i][j] = db_newcoly[i] ;
        }
    }

    cout << "New matrix y : " << endl ;
    for (i=0; i<newrow; i++) {
        for (j=0; j<col; j++) {
            cout << db_newmatrixy[i][j] << "\t" ;
        }
        cout << endl ;
    }

    /* step 5 in the procedure */
    mwArray mw_newmatrixx = double2mwArray_matrixReal(db_newmatrixx, newrow, col) ;
    mwArray mw_newmatrixy = double2mwArray_matrixReal(db_newmatrixy, newrow, col) ;
    mwArray mw_newmatrixz = double2mwArray_matrixReal(db_newmatrixz, newrow, col) ;

    mwArray mw_interp2z(1, 1, mxDOUBLE_CLASS) ;
    mwArray mw_method("cubic") ;

    myinterp2(1, mw_interp2z, mw_newmatrixx, mw_newmatrixy, mw_newmatrixz,
              mw_a, mw_b, mw_method);

    double db_interp2z = (double)mw_interp2z(1,1) ;

    cout << "Interpolation in two dimensions " ;
    cout << "and experimental data are not in grids " << endl;
    cout << " z = " << db_interp2z << endl ;

    /* free memories */
    delete [] db_oldcoly ;
    delete [] db_oldcolz ;

    delete [] db_newcolz ;

```

```
delete [] db_newmatrixx ;  
delete [] db_newmatrixy ;  
delete [] db_newmatrixz ;  
  
}
```

---

end code



## Chapter 13

# Roots of Equations

In this chapter we'll generate a C shared library *rootslib* and a C++ shared library *cpprootslib* from common M-files to find the roots of a polynomial function and a nonlinear function. The generated functions of these library will be used in a MSVC .Net project to find the roots of functions.

Following are steps to create a C shared library rootslib.dll and a C++ shared library cproot-slib.dll which will be used in the next sections.

We will write the M-files as shown below. These functions will be used to generate the C and C++ shared libraries.

myfzero.m and myroots.m

---

```
function r = myroots(c)
r = roots(c) ;
```

---

```
function x = myfzero(strfunc, x0)
```

```
F = inline(strfunc) ;
x = fzero(F, x0) ;
```

## A. FOR C SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C shared library

***rootslib*** :

```
mcc -B csharedlib:rootslib myfzero.m myroots.m
```

2. MATLAB Compiler 4 will create eight files for this C shared library:

```
rootslib.c      rootslib.ctf      rootslib.dll
rootslib.exp    rootslib.exports    rootslib.h
rootslib.lib    rootslib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the problems(open the file rootslib.h to see the names of these functions):

```
void mlfMyroots(int nargout, mxArray** r, mxArray* c);
void mlfMyfzero(int nargout, mxArray** x, mxArray* strfunc, mxArray* x0);
```

## B. FOR C++ SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C shared library

***cpprootslib*** :

```
mcc -W cpplib:cpprootslib -T link:lib myfzero.m myroots.m
```

2. MATLAB Compiler 4 will create eight files for this C++ shared library:

```
cpprootslib.cpp  cpprootslib.ctf      cpprootslib.dll
cpprootslib.exp  cpprootslib.exports  cpprootslib.h
cpprootslib.lib  cpprootslib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the problems(open the file rootslib.h to see the names of these functions):

```
void myfzero(int nargout, mxArray& x
              , const mxArray& strfunc, const mxArray& x0);

void myroots(int nargout, mxArray& r, const mxArray& c);
```



## 13.1 Roots of Polynomials

This section describe how to use the functions in the generated libraries to find the roots of a polynomial function.

### Problem 1

Find the root of the polynomial function:

$$f(x) = -x^3 + 7.2x^2 - 21x - 5$$

In calculation, the values of these coefficients will be assigned to an array `c[ ]` in the function (pay attention to its order):

$$f(x) = c_1x^3 + c_2x^2 + c_3x + c_4$$

where:

$$c_1 = -1, \quad c_2 = 7.2, \quad c_3 = -21, \text{ and } c_4 = -5$$

### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 1 by using the function `mlfMyroots(..)`. See the MATLAB manual [4] for more information on the MATLAB function `roots(..)`.

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Roots of a polynormial function." << endl ;
    Test obj ;
    obj.FindingRootsPolynormial() ;

    return 0 ;
}
```

---

```

/* Example.h */

#include <iostream.h>
#include "rootslib.h"
#include "mxUtilityCompilerVer4.h"

class Test {

public:
void FindingRootsPolynormal() ;

    Test () {
        mclInitializeApplication(NULL,0);
        rootslibInitialize();
    }

    ~Test () {
        rootslibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */

void Test::FindingRootsPolynormal() {

/*
Find the solutions of polynomial function:

$$f(x) = -x^3 + 7.2x^2 - 21x - 5$$

*/
int order = 3 ;
double db_coefs[] = { -1, 7.2, -21, -5 } ;

/* step 1 : declare mxArray variables */
mxArray *mx_coefs = NULL ;
mxArray *mx_x      = NULL ;

/* step 2 : assign memory */
int vectorSize = order + 1 ;

```

```

mx_coefs    = mxCreateDoubleMatrix(vectorSize, 1, mxREAL)      ;
mx_x        = mxCreateDoubleMatrix( order    , 1, mxCOMPLEX)  ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_vectorReal (db_coefs, mx_coefs) ;

/* step 4 : call an implemental function */
mlfMyroots(1, &mx_x, mx_coefs);

/* step 5 : convert back to C/C++ double */
double *db_xReal = new double [order] ;
double *db_xImag = new double [order] ;

mxArray2double_vectorComplex(mx_x, db_xReal, db_xImag) ;

/* print out */
cout << "Solutions of the polynomial function : " << endl ;
int i ;
for (i=0; i<3; i++) {
    cout << db_xReal[i] << " + " ;
    cout << db_xImag[i] << "i" << endl ;
}

/* step 6 : free memories */
mxDestroyArray(mx_coefs)      ;
mxDestroyArray(mx_x)         ;

delete [] db_xReal      ;
delete [] db_xImag      ;

}

```

---

end code

### Note

The roots of a polynomial function may have imaginary terms. Therefore we need to use the function *mxArray2double\_vectorComplex(..)* in converting to *double* type.

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 1 by using the function *myroots(..)*. See the MATLAB manual [4] for more information on the MATLAB function *roots(..)*.

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Roots of functions." << endl ;
    Test obj ;
    obj.FindingRootsPolynormal() ;

    return 0 ;
}
```

---

```
#include <iostream.h>
#include "cpprootslib.h"
#include "mwUtilityCompilerVer4.h"

class Test {

public:
    void FindingRootsPolynormal() ;

    Test () {
        mclInitializeApplication(NULL,0);
        cpprootslibInitialize();
    }

    ~Test () {
        cpprootslibTerminate();
        mclTerminateApplication();
    }
} ;
```

```

/* ***** */
void Test::FindingRootsPolynomial() {
/*
Find the solutions of polynomial function:

$$f(x) = -x^3 + 7.2x^2 - 21x - 5$$

*/
int order = 3 ;
double db_coefs[] = { -1, 7.2, -21, -5 } ;
int vectorSize = order + 1 ;

/* declare mxArray variables */
mxArray mw_coefs(vectorSize, 1, mxDOUBLE_CLASS) ;
mw_coefs.SetData(db_coefs, vectorSize) ;

/* call an implemental function */
mxArray mw_x(order, 1, mxDOUBLE_CLASS, mxCOMPLEX) ;
myroots(1, mw_x, mw_coefs);

/* convert back to C/C++ double */
double *db_xReal = new double [order] ;
double *db_xImag = new double [order] ;

mwArray2double_vectorComplex(mw_x, db_xReal, db_xImag) ;

/* print out */
cout << "Solutions of the polynomial function : " << endl ;
int i ;
for (i=0; i<3; i++) {
    cout << db_xReal[i] << " + " ;
    cout << db_xImag[i] << "i" << endl ;
}

/* step 6 : free memories */
delete [] db_xReal ;
delete [] db_xImag ;

}

```

---

end code

## 13.2 The Root of a Nonlinear-Equation

This section describe how to use the function `mlfMyfzero(..)` in the generated *rootslib* library or `myfzero(..)` in the generated *cpprootslib* library to find a root of a nonlinear function. These functions use a MATLAB function `fzero(..)` which returns ONLY ONE SOLUTION near the initial guess value. For more information of the `fzero(..)` function, refer to MATLAB manual [6].

### Problem 2

Find the root of the function:

$$f(x) = \sin(2x) + \cos(x) + 1$$

### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 2 by using the function `mlfMyfzero(..)`.

Listing code

---

```
void Test::FindingZeroFunction ( )
{
/* Find the solution of the function f(x) = sin(2*x) + cos(x) + 1
   fzero(..) returns ONLY ONE SOLUTION near a initial guess value
   If your problem is complicated, please look at functions
   in Optimization Tool Box
*/
double db_initialGuess = 0.9 ;
const char  strfunc[] = "sin(2*x) + cos(x) + 1" ;

/* step 1 : declare mxArray variables */
mxArray *mx_initialGuess = NULL ;
mxArray *mx_x            = NULL ;
mxArray *mx_strfunc ;

/* step 2 : assign memory */
mx_initialGuess = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_x            = mxCreateDoubleMatrix(1, 1, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal (db_initialGuess, mx_initialGuess) ;
```

```

mx_strfunc = mxCreateString(strfunc)      ;

/* step 4 : call an implemental function */
mlfMyfzero(1, &mx_x, mx_strfunc, mx_initialGuess);

/* step 5 : convert back to C/C++ double */
double db_xReal      ;
db_xReal = mxArray2double_scalarReal(mx_x) ;

/* print out */
cout << "Solutions of the given function : " << endl ;
cout << db_xReal << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_initialGuess)      ;
mxDestroyArray(mx_x)                  ;
mxDestroyArray(mx_strfunc)            ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 2 by using the function myfzero(..).

Listing code

---

```

void Test::FindingZeroFunction () {

/* Find the solution of the function f(x) = sin(2*x) + cos(x) + 1
   fzero(..) returns ONLY ONE SOLUTION near a initial guess value
   If your problem is complicated, please look at functions
   in Optimization Tool Box
*/

mwArray mw_initialGuess(1, 1, mxDOUBLE_CLASS) ;
mw_initialGuess(1,1) = 0.9 ;
mwArray mw_strfunc("sin(2*x) + cos(x) + 1") ;

```

```

/* call an implemental function */
mwArray mw_x(1, 1, mxDOUBLE_CLASS) ;
myfzero(1, mw_x, mw_strfunc, mw_initialGuess);

/* convert back to C/C++ double */
double db_xReal = (double) mw_x(1,1) ;

/* print out */
cout << "Solutions of the given function : " << endl ;
cout << db_xReal << endl ;

}

```

---

end code

### Note

The generated function myfzero(..) or myfzero(..) is a function-function and has an argument as an expression string. The form of this expression string follows the rule of a MATLAB expression string.



## Chapter 14

# Fast Fourier Transform

Fourier analysis is very useful for data analysis in applications. The Fourier transform divides a function into constituent sinusoids of different frequencies.

The Fourier transform of a function  $f(x)$  is defined as:

$$F(s) = \int_{-\infty}^{+\infty} f(x) e^{-i(2\pi s)x} dx \quad (14.1a)$$

and its inverse

$$f(x) = \int_{-\infty}^{+\infty} F(s) e^{i(2\pi x)s} ds \quad (14.1b)$$

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the discrete Fourier transform [1]. MATLAB provides functions for working on the Fast Fourier Transform and its inverse. These functions will be used to generate shared libraries in the following sections.

In this chapter we'll generate a C shared library ***fftlib*** and a C++ shared library ***cppfftlib*** from common M-files working on FFT problems. The generated functions of these libraries will be used in a MSVC .Net project to solve the integral problems.

Following are steps to create a C shared library `fftlib.dll` and a C++ shared library `cppfftlib.dll` which will be used to solve FFT problems in the next sections.

We will write the M-files as shown below. These functions will be used to generate the C and C++ shared libraries.

`myfft.m`, `myifft.m`, `myfft2.m`, and `myifft2.m`

---

```
function Y = myfft(X)
```

```
Y = fft(X) ;
```

---

```
function Y = myifft(X)
```

```
Y = ifft(X) ;
```

---

```
function Y = myfft2(X)
```

```
Y = fft2(X) ;
```

---

```
function Y = myifft2(X)
```

```
Y = ifft2(X) ;
```

---

## A. FOR C SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C shared library

*fttlib* :

```
mcc -B csharedlib:fttlib myfft.m myifft.m myfft2.m myifft2.m
```

2. MATLAB Compiler 4.0 will create eight files for this C shared library:

```
fftlib.c      fftlib.ctf      fftlib.dll
fftlib.exp    fftlib.exports  fftlib.h
fftlib.lib    fftlib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the common problems in the fast Fourier transform (open the file `fftlib.h` to see the names of these functions):

```
void mlfMyfft  (int nargout, mxArray** Y, mxArray* X);
void mlfMyifft (int nargout, mxArray** Y, mxArray* X);
void mlfMyfft2 (int nargout, mxArray** Y, mxArray* X);
void mlfMyifft2(int nargout, mxArray** Y, mxArray* X);
```

## B. FOR C++ SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C++ shared library *cppfftlib* :

```
mcc -W cpplib:cppfftlib -T link:lib myfft.m myifft.m myfft2.m myifft2.m
```

2. MATLAB Compiler 4 will create eight files for this C++ shared library:

```
cppfftlib.cpp    cppfftlib.ctf        cppfftlib.dll
cppfftlib.exp    cppfftlib.exports      cppfftlib.h
cppfftlib.lib    cppfftlib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the common problems in the fast Fourier transform (open the file *cppfftlib.h* to see the names of these functions):

```
void myfft (int nargout, mxArray& Y, const mxArray& X);
void myifft (int nargout, mxArray& Y, const mxArray& X);
void myfft2 (int nargout, mxArray& Y, const mxArray& X);
void myifft2(int nargout, mxArray& Y, const mxArray& X);
```

## 14.1 One-Dimensional Fast Fourier Transform

The MATLAB functions implement FFT and its inverse for a vector  $\mathbf{X}$  with length  $N$  as follows:

$$\mathbf{Y}_k = \sum_{n=1}^N \mathbf{X}_n e^{(-i2\pi)\left(\frac{n-1}{N}\right)(k-1)} \quad 1 \leq k \leq N \quad (14.2a)$$

and its inverse

$$\mathbf{X}_n = \frac{1}{N} \sum_{k=1}^N \mathbf{Y}_k e^{(i2\pi)\left(\frac{k-1}{N}\right)(n-1)} \quad 1 \leq n \leq N \quad (14.2b)$$

### Remarks

1. The vectors  $\mathbf{X}$  and  $\mathbf{Y}$  in equation 14.2 are represented by functions  $f(x)$  and  $F(s)$  in equation 14.1, respectively.
2. The first index of a vector in MATLAB starts at 1, therefore in the equation 14.2 we have term  $(n-1)$  and  $(k-1)$ . This produces identical results as the traditional Fourier equations from 0 to  $(N-1)$ .

**Problem 1**

**input**      Vector  $\mathbf{X}$ ,  
 $\mathbf{X} = \{ 6, 3, 7, -9, 0, 3, -2, 1 \}$

**output**    Finding the FFT vector  $\mathbf{Y}$  in Eq. 14.2a

**A. FOR C SHARED LIBRARY**

The following is the code to solve Problem 1 by using the function `mlfMyfft(..)`.

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Fast Fourier Transform." << endl ;
    Test obj ;
    obj.FastFourierTrans1D() ;

    return 0 ;
}
```

---

```
/* Example.h */

#include <iostream.h>
#include "fftlib.h"
#include "mxUtilityCompilerVer4.h"

class Test {

public:
    void FastFourierTrans1D() ;
```

```

Test () {
    mclInitializeApplication(NULL,0);
    fftlibInitialize();
}

~Test () {
    fftlibTerminate();
    mclTerminateApplication();
}

} ;

/* ***** */

void Test::FastFourierTrans1D() {

double  db_X [8] = { 6, 3, 7, -9, 0, 3, -2, 1 } ;
int vectorSize = 8 ;

/* step 1 : declare mxArray variables */
mxArray *mx_X = NULL ;
mxArray *mx_Y = NULL ;

/* step 2 : assign memory */
mx_X      = mxCreateDoubleMatrix(vectorSize, 1, mxREAL)      ;
mx_Y      = mxCreateDoubleMatrix(vectorSize, 1, mxCOMPLEX)   ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_vectorReal (db_X, mx_X) ;

/* step 4 : call an implemental function */
mlfMyfft (1, &mx_Y, mx_X);

/* step 5 : convert back to C/C++ double */
double *db_YReal = new double [vectorSize] ;
double *db_YImag = new double [vectorSize] ;

mxArray2double_vectorComplex(mx_Y, db_YReal, db_YImag) ;

/* print out */

```

```

cout << "Fast Fourier Transform of X : " << endl ;
int i ;
for (i=0; i<vectorSize; i++) {
    cout << db_YReal[i] << " + " ;
    cout << db_YImag[i] << "i" << endl ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_X)    ;
mxDestroyArray(mx_Y)    ;

delete [] db_YReal  ;
delete [] db_YImag  ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 1 by using the function myfft(..).

Listing code

---

```

/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Fast Fourier Transform." << endl ;
    Test obj ;
    obj.FastFourierTrans1D() ;

    return 0 ;

}

```

---

```

/* Example.h */

#include <iostream.h>
#include "cppffftlib.h"
#include "mwUtilityCompilerVer4.h"

class Test {

public:
void FastFourierTrans1D() ;

    Test () {
        mclInitializeApplication(NULL,0);
        cppffftlibInitialize();
    }

    ~Test () {
        cppffftlibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */
void Test::FastFourierTrans1D() {

double  db_X [8] = { 6, 3, 7, -9, 0, 3, -2, 1 } ;
int vectorSize = 8 ;

/* declare mwArray variables */
mwArray mw_X(vectorSize, 1, mxDOUBLE_CLASS) ;
mw_X.SetData(db_X, vectorSize) ;

/* call an implemental function */
mwArray mw_Y(vectorSize, 1, mxDOUBLE_CLASS) ;
myfft (1, mw_Y, mw_X);

/* convert back to C/C++ double */
double *db_YReal = new double [vectorSize] ;
double *db_YImag = new double [vectorSize] ;

```

```

mwArray2double_vectorComplex(mw_Y, db_YReal, db_YImag) ;

/* print out */
cout << "Fast Fourier Transform of X : " << endl ;
int i ;
for (i=0; i<vectorSize; i++) {
    cout << db_YReal[i] << " + " ;
    cout << db_YImag[i] << "i" << endl ;
}
cout << endl ;

delete [] db_YReal ;
delete [] db_YImag ;

}

```

---

end code

### **Problem 1B**

**input**      a vector **Y**,

**Y** real numbers = {9.00, 13.0711, 1.0, -1.0711, 13.0, -1.0711 , 1.0 , 13.0711 }

**Y** imaginary numbers = {0, -1.9289, -14.0, 16.0711, 0, -16.0711, 14.0, 1.9289 }

**output**    Finding an inverse FFT vector **X** in Eq. 14.2b

### **A. FOR C SHARED LIBRARY**

The following is the code to solve Problem 1B by using the function `mlfMyifft(..)`.

Listing code

---

```

void Test::InverseFastFourierTrans1D() {

double db_YReal[8] = { 9.00, 13.0711, 1.00, -1.0711, 13.00, -1.0711 , 1.00 , 13.0711 } ;
double db_YImag[8] = { 0 , -1.9289, -14.00, 16.0711, 0 , -16.0711, 14.00, 1.9289 } ;

int vectorSize = 8 ;

/* step 1 : declare mxArray variables */
mxArray *mx_X = NULL ;

```



```

mxArray *mx_Y = NULL ;

/* step 2 : assign memory */
mx_Y      = mxCreateDoubleMatrix(vectorSize, 1, mxCOMPLEX) ;
mx_X      = mxCreateDoubleMatrix(vectorSize, 1, mxCOMPLEX) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_vectorComplex (db_YReal, db_YImag, mx_Y) ;

/* step 4 : call an implemental function */
mlfMyifft  (1, &mx_X, mx_Y);

/* step 5 : convert back to C/C++ double */
double *db_XReal = new double [vectorSize] ;
double *db_XImag = new double [vectorSize] ;

mxArray2double_vectorComplex(mx_X, db_XReal, db_XImag) ;

/* print out */
cout << "Inverse Fast Fourier Transform of Y : " << endl ;
int i ;
for (i=0; i<vectorSize; i++) {
    cout << db_XReal[i] << " + " ;
    cout << db_XImag[i] << "i" << endl ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_X) ;
mxDestroyArray(mx_Y) ;

delete [] db_XReal ;
delete [] db_XImag ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 1B by using the function `myifft(..)`.

```

void Test::InverseFastFourierTrans1D() {

double  db_YReal[8] = { 9.00, 13.0711,   1.00, -1.0711, 13.00, -1.0711 , 1.00 , 13.0711 } ;
double  db_YImag[8] = { 0    , -1.9289, -14.00, 16.0711, 0    , -16.0711, 14.00, 1.9289  } ;

int vectorSize = 8 ;

/* declare mxArray variables */
mwArray mw_Y(vectorSize, 1, mxDOUBLE_CLASS, mxCOMPLEX) ;
mw_Y.Real().SetData(db_YReal, vectorSize) ;
mw_Y.Imag().SetData(db_YImag, vectorSize) ;

/* call an implemental function */
mwArray mw_X(vectorSize, 1, mxDOUBLE_CLASS, mxCOMPLEX) ;
myifft(1, mw_X, mw_Y);

std::cout << mw_X << std::endl ;

/* convert back to C/C++ double */
double *db_XReal = new double [vectorSize] ;
double *db_XImag = new double [vectorSize] ;

mwArray2double_vectorComplex(mw_X, db_XReal, db_XImag) ;

/* print out */
cout << "Inverse Fast Fourier Transform of Y : " << endl ;
int i ;
for (i=0; i<vectorSize; i++) {
    cout << db_XReal[i] << " + " ;
    cout << db_XImag[i] << "i" << endl ;
}
cout << endl ;

/* free memories */
delete [] db_XReal ;
delete [] db_XImag ;
}

```

## 14.2 Two-Dimensional Fast Fourier Transform

The MATLAB function `fft2(..)` ( $\mathbf{Y} = \text{fft2}(\mathbf{X})$ ) computes the one-dimensional FFT of each column of a matrix  $\mathbf{X}$ , and the size of the result matrix  $\mathbf{Y}$  is the same size of  $\mathbf{X}$ . If you want to get a different size, use the function  $\mathbf{Y} = \text{fft2}(\mathbf{X}, m, n)$ , refer to the MATLAB manual [6].

### Problem 2

**input**     Matrix  $\mathbf{X}$ ,

$$\mathbf{X} = \begin{bmatrix} 4 & 3.2 & 6.8 & 9.1 \\ -4 & 1.2 & 4.3 & 5.4 \\ 2.2 & -6.7 & 8 & 12 \end{bmatrix}$$

**output**     Finding the matrix  $\mathbf{Y}$ , which is a FFT matrix of  $\mathbf{X}$ .

### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 2 by using the function `mlfMyfft2(..)` in the generated *fftlib*.

Listing code

---

```
void Test::FastFourierTrans2D() {

double X[3][4] = { {4 , 3.2, 6.8, 9.1 } ,\
                   {-4 , 1.2, 4.3, 5.4 } ,\
                   {2.2, -6.7, 8 , 12.2 } } ;

int row = 3 ;
int col = 4 ;
int i, j ;

/* assign values for a buffer */
double **db_X = new double*[row] ;
for (i=0; i<row; i++) {
    db_X[i] = new double [col] ;
}

for(i=0; i<row; i++) {
    db_X[i] = & X[i][0] ;
}
```

```

/* step 1 : declare mxArray variables */
mxArray *mx_X = NULL ;
mxArray *mx_Y = NULL ;

/* step 2 : assign memory */
mx_X    = mxCreateDoubleMatrix(row, col, mxREAL)      ;
mx_Y    = mxCreateDoubleMatrix(row, col, mxCOMPLEX) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_matrixReal (db_X, mx_X) ;

/* step 4 : call an implemental function */
mlfMyfft2 (1, &mx_Y, mx_X);

/* step 5 : convert back to C/C++ double */
double **db_YReal = new double* [row] ;
double **db_YImag = new double* [col] ;

for (i=0; i<row; i++) {
    db_YReal[i] = new double [col] ;
    db_YImag[i] = new double [col] ;
}

mxArray2double_matrixComplex(mx_Y, db_YReal, db_YImag) ;

/* print out */
cout << "2-D Fast Fourier Transform of X : " << endl ;

for (i=0; i<row; i++) {
    for (j=0; j<col; j++ ) {
        cout << db_YReal[i][j] << " + " ;
        cout << db_YImag[i][j] << "i" << "\t\t" ;
    }
    cout << endl ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_X) ;

```

```

mxDestroyArray(mx_Y)      ;

delete [] db_X            ;
delete [] db_YReal        ;
delete [] db_YImag        ;

}

```

---

end code

---

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 2 by using the function `myfft2(..)` in the generated *cppfftlb*.

Listing code

---

```

void Test::FastFourierTrans2D() {

double X[3][4] = { {4 , 3.2, 6.8, 9.1 } ,\
                   {-4 , 1.2, 4.3, 5.4 } ,\
                   {2.2, -6.7, 8 , 12.2 } } ;

int row = 3 ;
int col = 4 ;
int i, j ;

/* step 1 : declare mxArray variables */
mxArray *mx_X = NULL ;
mxArray *mx_Y = NULL ;

/* step 2 : assign memory */
mx_X = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_Y = mxCreateDoubleMatrix(row, col, mxCOMPLEX) ;

/* step 3 : convert C/C++ double to mxArray */
mwArray mw_X = double2mwArray_matrixReal(&X[0][0], row, col) ;

/* step 4 : call an implemental function */
mwArray mw_Y(row, col, mxDOUBLE_CLASS, mxCOMPLEX) ;
myfft2 (1, mw_Y, mw_X);

```

```

/* step 5 : convert back to C/C++ double */
double **db_YReal = new double* [row] ;
double **db_YImag = new double* [col] ;

for (i=0; i<row; i++) {
    db_YReal[i] = new double [col] ;
    db_YImag[i] = new double [col] ;
}

mwArray2double_matrixComplex(mw_Y, db_YReal, db_YImag) ;

/* print out */
cout << "2-D Fast Fourier Transform of X : " << endl ;

for (i=0; i<row; i++) {
    for (j=0; j<col; j++ ) {
        cout << db_YReal[i][j] << " + " ;
        cout << db_YImag[i][j] << "i" << "\t\t" ;
    }
    cout << endl ;
}
cout << endl ;

/* step 6 : free memories */
delete [] db_YReal ;
delete [] db_YImag ;
}

```

---

end code

### **Problem 2B**

**input**     Matrix **Y**,

$$\mathbf{Y} = \begin{bmatrix} (45.70 + 0i) & (-16.9000 + 29.0000i) & (-3.10 + 0i) & (-16.9000 - 29.0000i) \\ (11.80 + 7.621i) & (-8.4806 - 3.4849i) & (-0.70 + 9.5263i) & (16.9806 + 7.8151i) \\ (11.80 - 7.621i) & (16.9806 - 7.8151i) & (-0.70 - 9.5263i) & (-8.4806 + 3.4849i) \end{bmatrix}$$

**output**     Finding the matrix **X** which is an inverse FFT matrix of **Y**.

## A. FOR C SHARED LIBRARY

The following is the code to solve Problem 2B using the function `mlfMyifft2(..)`.

Listing code

---

```
void Test::InverseFastFourierTrans2D() {

double YReal[3][4] = {{ 45.7000,  -16.9000,  -3.1000,  -16.9000 } ,\
                      { 11.8000,  -8.4806,  -0.7000,   16.9806 } ,\
                      { 11.8000,   16.9806,  -0.7000,  -8.4806 }  };

double YImag[3][4] =  {{ 0,          29.0000,   0,        -29.0000 } ,\
                      { 7.6210,  -3.4849,   9.5263,   7.8151 } ,\
                      {-7.6210,  -7.8151,  -9.5263,   3.4849 }  };

int row = 3 ;
int col = 4 ;
int i, j ;

/* assign values for a buffer */
double** db_YReal = new double* [row] ;
double** db_YImag = new double* [row] ;

for (i=0; i<row; i++) {
    db_YReal[i] = new double [col] ;
    db_YImag[i] = new double [col] ;
}

/* assign value for dbBufferMatrixY_Real */
for (i=0; i<row; i++) {
    db_YReal[i] = &YReal[i][0] ;
    db_YImag[i] = &YImag[i][0] ;
}

/* step 1 : declare mxArray variables */
mxArray *mx_X = NULL ;
mxArray *mx_Y = NULL ;

/* step 2 : assign memory */
```

```

mx_Y      = mxCreateDoubleMatrix(row, col, mxCOMPLEX) ;
mx_X      = mxCreateDoubleMatrix(row, col, mxCOMPLEX) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_matrixComplex(&YReal[0][0], &YImag[0][0], mx_Y) ;

/* step 4 : call an implemental function */
mlfMyifft2 (1, &mx_X, mx_Y);

/* step 5 : convert back to C/C++ double */
double **db_XReal = new double* [row] ;
double **db_XImag = new double* [row] ;

for(i=0; i<row; i++) {
    db_XReal[i] = new double [col] ;
    db_XImag[i] = new double [col] ;
}

mxArray2double_matrixComplex(mx_X, db_XReal, db_XImag) ;

/* print out */
cout << "Inverse 2-D Fast Fourier Transform of Y : " << endl ;

for (i=0; i<row; i++) {
    for (j=0; j<col; j++) {
        cout << db_XReal[i][j] << " + " ;
        cout << db_XImag[i][j] << "i" << "\t\t" ;
    }
    cout << endl ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_X)      ;
mxDestroyArray(mx_Y)      ;

delete [] db_XReal ;
delete [] db_XImag ;
}

```

---

end code



## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 2B using the function myifft2(..).

Listing code

---

```
void Test::InverseFastFourierTrans2D() {

double YReal[3][4] = {{ 45.7000,  -16.9000,  -3.1000,  -16.9000 } ,\
                      { 11.8000,  -8.4806,  -0.7000,   16.9806 } ,\
                      { 11.8000,   16.9806,  -0.7000,  -8.4806 } };

double YImag[3][4] = {{ 0,          29.0000,   0,        -29.0000 } ,\
                      { 7.6210,  -3.4849,   9.5263,   7.8151 } ,\
                      {-7.6210,  -7.8151,  -9.5263,   3.4849 } };

int row = 3 ;
int col = 4 ;
int i, j ;

/* convert C/C++ double to mxArray */
mxArray mw_Y = double2mxArray_matrixComplex (&YReal[0][0], &YImag[0][0], row, col) ;

/* call an implemental function */
mxArray mw_X(row, col, mxDOUBLE_CLASS, mxCOMPLEX) ;
myifft2 (1, mw_X, mw_Y);

/* convert back to C/C++ double */
double **db_XReal = new double* [row] ;
double **db_XImag = new double* [row] ;

for(i=0; i<row; i++) {
    db_XReal[i] = new double [col] ;
    db_XImag[i] = new double [col] ;
}

mxArray2double_matrixComplex(mw_X, db_XReal, db_XImag) ;

/* print out */
cout << "Inverse 2-D Fast Fourier Transform of Y : " << endl ;
```

```
for (i=0; i<row; i++) {  
    for (j=0; j<col; j++ ) {  
        cout << db_XReal[i][j] << " + " ;  
        cout << db_XImag[i][j] << "i" << "\\t\\t" ;  
    }  
    cout << endl ;  
}  
cout << endl ;  
  
/* free memories */  
delete [] db_XReal ;  
delete [] db_XImag ;  
  
}
```

---

end code

## Chapter 15

# Eigenvalues and Eigenvectors

In this chapter we'll generate a C shared library *eigenlib* and a C++ shared library *cppeigenlib* from common M-files working on problems of eigenvectors and eigenvalues. The generated functions of these libraries will be used in a MSVC .Net project to solve the eigen problems. This chapter focus on using the MATLAB function `eig(..)` to find the eigenvalues and eigenvectors of a square matrix. For more information of this function, refer to the MATLAB manual [5].

Following are steps to create a C shared library `eigenlib.dll` and a C++ shared library `cppeigenlib.dll` which will be used to solve problems in the next sections.

We will write the M-file `myeig.m` as shown below. These functions will be used to generate the C and C++ shared libraries.

```
function [V,D] = myeig(A)
```

```
[V,D] = eig(A) ;
```

### A. FOR C SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C shared library *eigenlib* :

```
mcc -B csharedlib:eigenlib myeig.m
```

2. MATLAB Compiler 4 will create eight files for this C shared library:

<code>eigenlib.c</code>	<code>eigenlib.ctf</code>	<code>eigenlib.dll</code>
<code>eigenlib.exp</code>	<code>eigenlib.exports</code>	<code>eigenlib.h</code>
<code>eigenlib.lib</code>	<code>eigenlib_mcc_component_data.c</code>	

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the next sections, we'll use the following implemental function in this library to solve the common eigen-problems (open the file `eigenlib.h` to see the name of this function):

```
void mlfMyeig(int nargout, mxArray** V, mxArray** D, mxArray* A);
```

## B. FOR C++ SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C++ shared library *cppeigenlib* :

```
mcc -W cpplib:cppeigenlib -T link:lib myeig.m
```

2. MATLAB Compiler 4 will create eight files for this C++ shared library:

```
cppeigenlib.cpp    cppeigenlib.ctf        cppeigenlib.dll
cppeigenlib.exp    cppeigenlib.exports    cppeigenlib.h
cppeigenlib.lib    cppeigenlib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the next sections, we'll use the following implemental function in this library to solve the common eigen-problems (open the file `cppeigenlib.h` to see the name of this function):

```
void myeig(int nargout, mxArray& V, mxArray& D, const mxArray& A);
```

## 15.1 Eigenvalues and Eigenvectors

### Problem 1

**input**     a square matrix **A**

$$\mathbf{A} = \begin{bmatrix} 0 & -6 & -1 \\ 6 & 2 & -16 \\ -5 & 20 & -10 \end{bmatrix}$$

**output**     Finding eigenvalues and eigenvectors of **A**

## A. FOR C SHARED LIBRARY

The following is the code to solve Problem 1 by using function `mlfMyeig(..)` in the generated *eigenlib* library to find eigenvalues and eigenvectors of a square matrix.

Listing code

---

```

/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Eigenvalues and eigenvectors" << endl ;
    Test obj ;
    obj.EigValueVector() ;

    return 0 ;
}



---


/* Example.h */

#include <iostream.h>
#include "eigenlib.h"
#include "mxUtilityCompilerVer4.h"

class Test {
public:
    void EigValueVector() ;

    Test () {
        mclInitializeApplication(NULL,0);
        eigenlibInitialize();
    }

    ~Test () {
        eigenlibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */
void Test::EigValueVector() {
    double A[3][3] = {{ 0, -6, -1} , {6, 2, -16} , {-5, 20, -10} } ;

```

```

int row = 3 ;
int col = 3 ;
int i, j ;

/* step 1 : declare mxArray variables */
mxArray *mx_A          = NULL ;
mxArray *mx_eigenvectors = NULL ;
mxArray *mx_eigenvalues  = NULL ;

/* step 2 : assign memory */
mx_A          = mxCreateDoubleMatrix(row, col, mxREAL) ;
mx_eigenvectors = mxCreateDoubleMatrix(row, col, mxCOMPLEX) ;
mx_eigenvalues  = mxCreateDoubleMatrix(row, col, mxCOMPLEX) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_matrixReal(&A[0][0], mx_A) ;

/* step 4 : call an implemental function */
mlfMyeig(2, &mx_eigenvectors, &mx_eigenvalues, mx_A);

/* step 5 : convert back to C/C++ double */
double **db_eigenvectorsReal = new double* [col] ;
double **db_eigenvectorsImag = new double* [col] ;

double **db_eigenvaluesReal = new double* [row] ;
double **db_eigenvaluesImag = new double* [row] ;

for (i=0; i<row; i++) {
    db_eigenvectorsReal[i] = new double [col] ;
    db_eigenvectorsImag[i] = new double [col] ;

    db_eigenvaluesReal[i] = new double [col] ;
    db_eigenvaluesImag[i] = new double [col] ;
}

mxArray2double_matrixComplex(mx_eigenvectors, db_eigenvectorsReal, db_eigenvectorsImag) ;
mxArray2double_matrixComplex(mx_eigenvalues , db_eigenvaluesReal , db_eigenvaluesImag) ;

/* print out */

```

```

cout << "Eigenvalues of the matrix A : " << endl ;

for (i=0; i<row; i++) {
    cout << db_eigenvaluesReal[i][i] << " + " ;
    cout << db_eigenvaluesImag[i][i] << "i" << endl ;
}

cout << endl ;

cout << "Eigenvectors of the matrix A : " << endl ;

for (j=0; j<col; j++ ) {
    cout << "Eigenvector " << (j+1) << " is : " << endl ;

    for (i=0; i<row; i++) {
        cout << db_eigenvectorsReal[i][j] << " + " ;
        cout << db_eigenvectorsImag[i][j] << "i" << endl ;
    }

    cout << endl ;
}

cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_A)      ;
mxDestroyArray(mx_eigenvalues ) ;
mxDestroyArray(mx_eigenvectors) ;

delete [] db_eigenvaluesReal ;
delete [] db_eigenvaluesImag ;

delete [] db_eigenvectorsReal ;
delete [] db_eigenvectorsImag ;
}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 1 by using function `myeig(..)` in the generated *cppeigenlib* library to find eigenvalues and eigenvectors of a square matrix.

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Eigenvalues and eigenvectors" << endl ;
    Test obj ;
    obj.EigValueVector() ;

    return 0 ;
}
```

---

```
/* Example.h */

#include <iostream.h>
#include "cppeigenlib.h"
#include "mwUtilityCompilerVer4.h"

class Test {

public:
    void EigValueVector() ;

    Test () {
        mclInitializeApplication(NULL,0);
        cppeigenlibInitialize();
    }

    ~Test () {
        cppeigenlibTerminate();
        mclTerminateApplication();
    }
}
```



```

    }

} ;

/* ***** */
void Test::EigValueVector() {

double A[3][3] = {{ 0, -6, -1} , {6, 2, -16} , {-5, 20, -10} } ;

int row = 3 ;
int col = 3 ;
int i, j ;

/* convert C/C++ double to mxArray */
mwArray mw_A = double2mwArray_matrixReal(&A[0][0], row, col) ;

/* call an implemental function */
mwArray mw_eigenvectors(row, col, mxDOUBLE_CLASS, mxCOMPLEX) ;
mwArray mw_eigenvalues (row, col, mxDOUBLE_CLASS, mxCOMPLEX) ;

myeig(2, mw_eigenvectors, mw_eigenvalues, mw_A);

/* convert back to C/C++ double */
double **db_eigenvectorsReal = new double* [col] ;
double **db_eigenvectorsImag = new double* [col] ;

double **db_eigenvaluesReal = new double* [row] ;
double **db_eigenvaluesImag = new double* [row] ;

for (i=0; i<row; i++) {
    db_eigenvectorsReal[i] = new double [col] ;
    db_eigenvectorsImag[i] = new double [col] ;

    db_eigenvaluesReal[i] = new double [col] ;
    db_eigenvaluesImag[i] = new double [col] ;
}

mwArray2double_matrixComplex(mw_eigenvectors, db_eigenvectorsReal, db_eigenvectorsImag) ;
mwArray2double_matrixComplex(mw_eigenvalues , db_eigenvaluesReal , db_eigenvaluesImag ) ;

```

```

/* print out */
cout << "Eigenvalues of the matrix A : " << endl ;

for (i=0; i<row; i++) {
    cout << db_eigenvaluesReal[i][i] << " + " ;
    cout << db_eigenvaluesImag[i][i] << "i" << endl ;
}
cout << endl ;

cout << "Eigenvectors of the matrix A : " << endl ;

for (j=0; j<col; j++ ) {
    cout << "Eigenvector " << (j+1) << " is : " << endl ;

    for (i=0; i<row; i++) {

        cout << db_eigenvectorsReal[i][j] << " + " ;
        cout << db_eigenvectorsImag[i][j] << "i" << endl ;
    }

    cout << endl ;
}
cout << endl ;

/* step 6 : free memories */
delete [] db_eigenvaluesReal ;
delete [] db_eigenvaluesImag ;

delete [] db_eigenvectorsReal ;
delete [] db_eigenvectorsImag ;

}

```

---

end code

See the code of the files `mxUtilityCompilerVer4.h` and `mwUtilityCompilerVer4.h` in Chapter 7.

### Remarks

1. The MATLAB function `[V,D] = eig(A)` assigns eigenvalues D as a diagonal matrix, in which the eigenvalues are diagonal terms. The above programming gives the eigenvalues in

the matrix form as follow:

$$eigenvalues = \begin{bmatrix} -0.30710 & 0.00000 & 0.00000 \\ 0.00000 & -0.24645 + 1.76008i & 0.00000 \\ 0.00000 & 0.00000 & -0.24645 - 1.76008i \end{bmatrix}$$

then the eigenvalues of the matrix  $\mathbf{A}$  are:

$$eigenvalue_1 = -0.30710$$

$$eigenvalue_2 = -0.24645 + 1.76008i$$

$$eigenvalue_3 = -0.24645 - 1.76008i$$

2. The MATLAB function  $[\mathbf{V}, \mathbf{D}] = \mathbf{eig}(\mathbf{A})$  also assigns eigenvectors  $\mathbf{V}$  as a matrix, in which the eigenvectors are matrix columns. The above programming gives the value as follows:

$$eigenvectors = \begin{bmatrix} -0.83261 & 0.20027 - 0.13936i & 0.20027 + 0.13936i \\ -0.35534 & -0.21104 - 0.64472i & -0.21104 + 0.64472i \\ -0.42485 & -0.69301 & -0.69301 \end{bmatrix}$$

then the eigenvectors of the matrix  $\mathbf{A}$  are:

$$eigenvector_1 = \begin{bmatrix} -0.83261 \\ -0.35534 \\ -0.42485 \end{bmatrix}, \quad eigenvector_2 = \begin{bmatrix} 0.20027 - 0.13936i \\ -0.21104 - 0.64472i \\ -0.69301 \end{bmatrix}$$

$$, \text{ and } eigenvector_3 = \begin{bmatrix} 0.20027 + 0.13936i \\ -0.21104 + 0.64472i \\ -0.69301 \end{bmatrix}$$



## Chapter 16

# Random Numbers

In this chapter we'll generate a C shared library ***randomlib*** and a C++ shared library ***cpprandomlib*** from common M-files working on random problems. The generated functions of these libraries will be used in a MSVC .Net project to solve the typical random problems.

Following are steps to create a C shared library randomlib.dll and a C++ shared library cpprandomlib.dll which will be used to solve problems in the next sections.

We will write the M-files, myrand.m and myrandn.m, as shown below. These functions will be used to generate the C and C++ shared libraries.

---

```
function Y = myrand(m,n)
```

```
Y = rand(m,n) ;
```

---

```
function Y = myrandn(m,n)
```

```
Y = randn(m,n) ;
```

```
%
```

### A. FOR C SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C shared library ***randomlib***:

```
mcc -B csharedlib:randomlib myrand.m myrandn.m
```

2. MATLAB Compiler 4 will create eight files for this C shared library:

```
randomlib.c      randomlib.ctf      randomlib.dll
randomlib.exp    randomlib.exports    randomlib.h
randomlib.lib    randomlib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the common random problems (open the file `randomlib.h` to see the names of these functions):

```
void mlfMyrand (int nargout, mxArray** Y, mxArray* m, mxArray* n);
void mlfMyrandn(int nargout, mxArray** Y, mxArray* m, mxArray* n);
```

## B. FOR C++ SHARED LIBRARY

1. Write the command in Windows Command Prompt as follows to create a C++ shared library *cpprandomlib*:

```
mcc -W cpplib:cpprandomlib -T link:lib myrand.m myrandn.m
```

2. MATLAB Compiler 4 will create eight files for this C++ shared library:

```
cpprandomlib.cpp  cpprandomlib.ctf      cpprandomlib.dll
cpprandomlib.exp  cpprandomlib.exports  cpprandomlib.h
cpprandomlib.lib  cpprandomlib_mcc_component_data.c
```

Add and set these files to the MSVC .Net project as described in Chapter 6.

3. In the following sections, we'll use the following implemental functions in this library to solve the common random problems (open the file `cpprandomlib.h` to see the names of these functions):

```
void myrand(int nargout, mxArray& Y, const mxArray& m, const mxArray& n);
void myrandn(int nargout, mxArray& Y, const mxArray& m, const mxArray& n);
```

## 16.1 Uniform Random Numbers

This section describe how to use the function `mlfMyrand(..)` in the generated *randomlib* library or `myrand(..)` in the generated *cpprandomlib* library to find uniform random numbers. Uniform random numbers are random numbers that lie within a specified range. This section describe

how to use the function `mlfMyrand(..)` or `myrand(..)` in the generated libraries to solve uniform random number problems. These functions use the MATLAB function `rand(m,n)` to generate a matrix (size  $m \times n$ ) of random numbers. These random numbers are uniformly distributed in the interval (0,1). For more information of this function `rand(m,n)`, refer to the MATLAB manual [7].

### 16.1.1 Generating Uniform Random Numbers in Range [0,1]

#### Problem 1

**input**      A number,  $N = 5$

**output**    Generating  $N$  uniform random numbers in range  $[0, 1]$

#### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 1 by using the function `mlfMyrand(..)`

Listing code

---

```
/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Random Number" << endl ;
    Test obj ;
    obj.uniformRandom_vector() ;

    return 0 ;
}
```

---

```
/* Example.h */

#include <iostream.h>
#include <math.h>
```

```

#include "randomlib.h"
#include "mxUtilityCompilerVer4.h"

class Test {

public:
void uniformRandom_vector () ;

    Test () {
        mclInitializeApplication(NULL,0);
        randomlibInitialize();
    }

    ~Test () {
        randomlibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */
void Test::uniformRandom_vector () {

    int row = 1 ;
    int col = 5 ;
    int i ;

    /* step 1 : declare mxArray variables */
    mxArray *mx_row = NULL ;
    mxArray *mx_col = NULL ;
    mxArray *mx_uniformRandVector  = NULL ;

    /* step 2 : assign memory */
    mx_row  = mxCreateDoubleMatrix(1, 1, mxREAL)    ;
    mx_col  = mxCreateDoubleMatrix(1, 1, mxREAL)    ;
    mx_uniformRandVector = mxCreateDoubleMatrix(row, col, mxREAL)    ;

    /* step 3 : convert C/C++ double to mxArray */
    double2mxArray_scalarReal (row, mx_row) ;
    double2mxArray_scalarReal (col, mx_col) ;

```



```

/* step 4 : call an implemental function */
mlfMyrand(1, &mx_uniformRandVector, mx_row, mx_col);

/* step 5 : convert back to C/C++ double */
double * db_uniformRandVector = new double [col] ;
mxArray2double_vectorReal(mx_uniformRandVector, db_uniformRandVector) ;

/* print out */
cout << endl ;
cout << "Uniform random numbers from 0 to 1 : " << endl ;
for (i=0; i<col; i++) {
    cout << db_uniformRandVector[i] << "\t" ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_row) ;
mxDestroyArray(mx_col) ;
mxDestroyArray(mx_uniformRandVector) ;

delete [] db_uniformRandVector ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 1 by using the function myrand(..)

Listing code

---

```

/* Example.cpp */

#pragma warning(disable : 4995)
#include "Example.h"

int main() {

    cout << "Random Number" << endl ;

```

```

    Test obj ;

    obj.uniformRandom_vector () ;

    return 0 ;
}



---


/* Example.h */

#include <iostream.h>
#include <math.h>
#include "cpprandomlib.h"
#include "mwUtilityCompilerVer4.h"

class Test {

public:
    void uniformRandom_vector () ;

    Test () {
        mclInitializeApplication(NULL,0);
        cpprandomlibInitialize();
    }

    ~Test () {
        cpprandomlibTerminate();
        mclTerminateApplication();
    }

} ;

/* ***** */
void Test::uniformRandom_vector () {

    int row = 1 ;
    int col = 5 ;
    int i ;

    /* declare mwArray variables */
    mwArray mw_row(1, 1, mxDOUBLE_CLASS) ;
    mw_row(1,1) = row ;

```

```

mwArray mw_col(1, 1, mxDOUBLE_CLASS) ;
mw_col(1,1) = col ;

/* call an implemental function */
mwArray mw_uniformRandVector(row, col, mxDOUBLE_CLASS) ;
myrand(1, mw_uniformRandVector, mw_row, mw_col);

/* convert back to C/C++ double */
double * db_uniformRandVector = new double [col] ;
mwArray2double_vectorReal(mw_uniformRandVector, db_uniformRandVector) ;

/* print out */
cout << endl ;
cout << "Uniform random numbers from 0 to 1 :" << endl ;
for (i=0; i<col; i++) {
    cout << db_uniformRandVector[i] << "\t" ;
}
cout << endl ;

/* free memories */
delete [] db_uniformRandVector ;

}

```

---

end code

### 16.1.2 Generating Uniform Random Numbers in Range $[a,b]$

#### Problem 2

**input**     A number,  $N = 6$   
               A range  $[a, b]$ , where  $a=2$  and  $b=18$

**output**    Generating  $N$  uniform random numbers in the range  $[a, b]$

## A. FOR C SHARED LIBRARY

The following is the code to solve Problem 2 using the function `mlfMyrand(..)`.

Listing code

---

```
void Test::uniformRandom_vector2 () {

double a = 2.0 ;
double b = 18.0 ;

int row = 1 ;
int col = 6 ;
int i ;

/* step 1 : declare mxArray variables */
mxArray *mx_row = NULL ;
mxArray *mx_col = NULL ;
mxArray *mx_uniformRandVector = NULL ;

/* step 2 : assign memory */
mx_row = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_col = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_uniformRandVector = mxCreateDoubleMatrix(row, col, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal (row, mx_row) ;
double2mxArray_scalarReal (col, mx_col) ;

/* step 4 : call an implemental function */
/* random numbers in [0, 1] */
mlfMyrand(1, &mx_uniformRandVector, mx_row, mx_col);

/* step 5 : convert back to C/C++ double */
double * db_uniformRandVector = new double [col] ;
mxArray2double_vectorReal(mx_uniformRandVector, db_uniformRandVector) ;

/* print out */
cout << endl ;
cout << "Uniform random numbers from a=2 to b=18 : " << endl ;
for (i=0; i<col; i++) {
    cout << a + (b-a)*db_uniformRandVector[i] << "\t" ;
```

```

}

cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_row) ;
mxDestroyArray(mx_col) ;
mxDestroyArray(mx_uniformRandVector) ;

delete [] db_uniformRandVector ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 2 using the function `myrand(..)`.

Listing code

---

```

void Test::uniformRandom_vector2 () {

double a = 2.0 ;
double b = 18.0 ;

int row = 1 ;
int col = 6 ;
int i ;

/* declare mxArray variables */
mxArray mw_row(1, 1, mxDOUBLE_CLASS) ;
mw_row(1,1) = row ;

mxArray mw_col(1, 1, mxDOUBLE_CLASS) ;
mw_col(1,1) = col ;

/* call an implemental function */
/* random numbers in [0, 1] */
mxArray mw_uniformRandVector(row, col, mxDOUBLE_CLASS) ;
myrand(1, mw_uniformRandVector, mw_row, mw_col);
}

```

```

/* convert back to C/C++ double */
double * db_uniformRandVector = new double [col] ;
mwArray2double_vectorReal(mw_uniformRandVector, db_uniformRandVector) ;

/* print out */
cout << endl ;
cout << "Uniform random numbers from a=2 to b=18 : " << endl ;
for (i=0; i<col; i++) {
    cout << a + (b-a)*db_uniformRandVector[i] << "\t" ;
}
cout << endl ;

/* free memories */
delete [] db_uniformRandVector ;

}

```

---

end code

### 16.1.3 Generating a Matrix of Uniform Random Numbers in Range [0,1]

#### Problem 3

**input** A row number  $m = 8$  and a column number  $n = 5$

**output** Generating a matrix (size  $m \times n$ ) of uniform random numbers in the range [0,1]

#### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 3 by using the function `mlfMyrand(..)`.

Listing code

---

```

void Test::uniformRandom_matrix () {

int row = 8 ;
int col = 5 ;
int i, j ;

/* step 1 : declare mxArray variables */

```

```

mxArray *mx_row = NULL ;
mxArray *mx_col = NULL ;
mxArray *mx_uniformRandMatrix = NULL ;

/* step 2 : assign memory */
mx_row = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_col = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_uniformRandMatrix = mxCreateDoubleMatrix(row, col, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal (row, mx_row) ;
double2mxArray_scalarReal (col, mx_col) ;

/* step 4 : call an implemental function */
mlfMyrand(1, &mx_uniformRandMatrix, mx_row, mx_col);

/* step 5 : convert back to C/C++ double */
double ** db_uniformRandMatrix = new double* [row] ;
for (i=0; i<row; i++) {
    db_uniformRandMatrix[i] = new double [col] ;
}

mxArray2double_matrixReal(mx_uniformRandMatrix, db_uniformRandMatrix) ;

/* print out */
cout << endl ;
cout << "The matrix of uniform random numbers from 0 to 1 :" << endl ;
for (i=0; i<row; i++) {
    for (j=0; j<col; j++) {
        cout << db_uniformRandMatrix[i][j] << "\t" ;
    }
    cout << endl ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_row) ;
mxDestroyArray(mx_col) ;
mxDestroyArray(mx_uniformRandMatrix) ;

```

```

delete [] db_uniformRandMatrix ;

}

```

---

end code

---

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 3 by using the function myrand(..).

### Listing code

---

```

void Test::uniformRandom_matrix () {

    int row = 8 ;
    int col = 5 ;
    int i;

    /* declare mxArray variables */
    mxArray mw_row(1, 1, mxDOUBLE_CLASS) ;
    mw_row(1,1) = row ;

    mxArray mw_col(1, 1, mxDOUBLE_CLASS) ;
    mw_col(1,1) = col ;

    /* call an implemental function */
    mxArray mw_uniformRandMatrix(row, col, mxDOUBLE_CLASS) ;
    myrand(1, mw_uniformRandMatrix, mw_row, mw_col);

    /* convert back to C/C++ double */
    double ** db_uniformRandMatrix = new double* [row] ;
    for (i=0; i<row; i++) {
        db_uniformRandMatrix[i] = new double [col] ;
    }

    mxArray2double_matrixReal(mw_uniformRandMatrix, db_uniformRandMatrix) ;

    /* print out */
    cout << endl ;
    cout << "The matrix of uniform random numbers from 0 to 1 :" << endl ;
    printMatrix(db_uniformRandMatrix, row, col) ;
    cout << endl ;
}

```



```
delete [] db_uniformRandMatrix ;
}
```

---

end code

---

### 16.1.4 Generating a Matrix of Uniform Random Numbers in Range $[a,b]$

#### Problem 4

**input** A row number  $m = 8$  and a column number  $n = 5$   
 Range  $[a,b]$ , where  $a=4$  and  $b=17$

**output** Generating a matrix (size  $m \times n$ ) of uniform random numbers in the range  $[a,b]$

**A. FOR C SHARED LIBRARY** The following is the code to solve Problem 4 by using the function `mlfMyrand(..)`.

Listing code

---

```
void Test::uniformRandom_matrix () {

int row = 8 ;
int col = 5 ;
int i, j ;

/* step 1 : declare mxArray variables */
mxArray *mx_row = NULL ;
mxArray *mx_col = NULL ;
mxArray *mx_uniformRandMatrix = NULL ;

/* step 2 : assign memory */
mx_row = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_col = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_uniformRandMatrix = mxCreateDoubleMatrix(row, col, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal (row, mx_row) ;
```

```

double2mxArray_scalarReal (col, mx_col) ;

/* step 4 : call an implemental function */
mlfMyrand(1, &mx_uniformRandMatrix, mx_row, mx_col);

/* step 5 : convert back to C/C++ double */
double ** db_uniformRandMatrix = new double* [row] ;
for (i=0; i<row; i++) {
    db_uniformRandMatrix[i] = new double [col] ;
}

mxArray2double_matrixReal(mx_uniformRandMatrix, db_uniformRandMatrix) ;

/* print out */
cout << endl ;
cout << "The matrix of uniform random numbers from 0 to 1 : " << endl ;
for (i=0; i<row; i++) {
    for (j=0; j<col; j++) {
        cout << db_uniformRandMatrix[i][j] << "\t" ;
    }
    cout << endl ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_row) ;
mxDestroyArray(mx_col) ;
mxDestroyArray(mx_uniformRandMatrix) ;

delete [] db_uniformRandMatrix ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 4 by using the function myrand(..).

Listing code

---

```

void Test::uniformRandom_matrix2 () {

    int row = 8 ;
    int col = 5 ;
    int i, j ;

    double a = 4.0 ;
    double b = 17.0 ;

    /* declare mxArray variables */
    mxArray mw_row(1, 1, mxDOUBLE_CLASS) ;
    mw_row(1,1) = row ;

    mxArray mw_col(1, 1, mxDOUBLE_CLASS) ;
    mw_col(1,1) = col ;

    /* call an implemental function */
    mxArray mw_uniformRandMatrix(row, col, mxDOUBLE_CLASS) ;
    myrand(1, mw_uniformRandMatrix, mw_row, mw_col);

    /* convert back to C/C++ double */
    double ** db_uniformRandMatrix = new double* [row] ;
    for (i=0; i<row; i++) {
        db_uniformRandMatrix[i] = new double [col] ;
    }

    mxArray2double_matrixReal(mw_uniformRandMatrix, db_uniformRandMatrix) ;

    /* print out */
    cout << endl ;
    cout << "The matrix of uniform random numbers from a to b :" << endl ;
    for (i=0; i<row; i++) {
        for (j=0; j<col; j++) {
            cout << a + (b-a)*db_uniformRandMatrix[i][j] << "\t" ;
        }
        cout << endl ;
    }
    cout << endl ;
}

```

```

/* free memories */
delete [] db_uniformRandMatrix ;

}

```

---

end code

---

## 16.2 Normal Random Numbers

Normal random numbers are random numbers that establish the normal distribution (Gaussian distribution). This section describe how to use the function `mlfMyrandn(..)` in the generated ***randomlib*** library or `myrandn(..)` in the generated ***randomlib*** to solve normal random number problems. These functions use the MATLAB function `randn(m,n)` to generate a matrix (size  $n$  by  $m$ ) of random numbers. These random numbers are normally distributed with specified properties,  $\mu = 0$ , variance  $\sigma^2 = 1$ .

### 16.2.1 Generating Normal Random Numbers with mean=0 and variance=1

#### Problem 5

**input**     A number,  $N = 5$

**output**    Generating ( $N$ ) normal random numbers with :

mean  $\mu = 0$

variance  $\sigma^2 = 1$

#### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 5 by using the function `mlfMyrandn(..)`.

Listing code

---

```

void Test::normalRandom_vector () {

int row = 1 ;
int col = 5 ;
int i ;

```

```

/* step 1 : declare mxArray variables */
mxArray *mx_row = NULL ;
mxArray *mx_col = NULL ;
mxArray *mx_normalRandVector = NULL ;

/* step 2 : assign memory */
mx_row = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_col = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_normalRandVector = mxCreateDoubleMatrix(row, col, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal (row, mx_row) ;
double2mxArray_scalarReal (col, mx_col) ;

/* step 4 : call an implemental function */
mlfMyrandn(1, &mx_normalRandVector, mx_row, mx_col);

/* step 5 : convert back to C/C++ double */
double * db_normalRandVector = new double [col] ;
mxArray2double_vectorReal(mx_normalRandVector, db_normalRandVector) ;

/* print out */
cout << endl ;
cout << "Normal random numbers :" << endl ;
for (i=0; i<col; i++) {
    cout << db_normalRandVector[i] << "\t" ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_row) ;
mxDestroyArray(mx_col) ;
mxDestroyArray(mx_normalRandVector) ;

delete [] db_normalRandVector ;
}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 5 by using the function myrandn(..).

Listing code

---

```
void Test::normalRandom_vector () {

    int row = 1 ;
    int col = 5 ;
    int i ;

    /* declare mxArray variables */
    mxArray mw_row(1, 1, mxDOUBLE_CLASS) ;
    mw_row(1,1) = row ;

    mxArray mw_col(1, 1, mxDOUBLE_CLASS) ;
    mw_col(1,1) = col ;

    /* call an implemental function */
    mxArray mw_normalRandVector(row, col, mxDOUBLE_CLASS) ;
    myrandn(1, mw_normalRandVector, mw_row, mw_col);

    /* convert back to C/C++ double */
    double * db_normalRandVector = new double [col] ;
    mxArray2double_vectorReal(mw_normalRandVector, db_normalRandVector) ;

    /* print out */
    cout << endl ;
    cout << "Normal random numbers :" << endl ;
    for (i=0; i<col; i++) {
        cout << db_normalRandVector[i] << "\t" ;
    }
    cout << endl ;

    /* free memories */
    delete [] db_normalRandVector ;

}
```

---

end code

## 16.2.2 Generating Normal Random Numbers with mean= $\mu$ and variance= $\sigma^2$

### Problem 6

**input**     A number,  $N = 5$

**output**    Generating  $N$  normal random numbers with specified properties:

mean  $\mu = 0.56$

variance  $\sigma^2 = 0.12$

### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 6 by using the function `mlfMyrandn(..)`.

#### Listing code

---

```
void Test::normalRandom_vector2 () {

/* Generate a vector of normal random numbers at
particular mean and variance */

int row = 1 ;
int col = 5 ;

double mean_mu = 0.56 ;
double variance = 0.12 ;
int i ;

/* step 1 : declare mxArray variables */
mxArray *mx_row = NULL ;
mxArray *mx_col = NULL ;
mxArray *mx_normalRandVector = NULL ;

/* step 2 : assign memory */
mx_row = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_col = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_normalRandVector = mxCreateDoubleMatrix(row, col, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal (row, mx_row) ;
```

```

double2mxArray_scalarReal (col, mx_col) ;

/* step 4 : call an implemental function */
mlfMyrandn(1, &mx_normalRandVector, mx_row, mx_col);

/* step 5 : convert back to C/C++ double */
double * db_normalRandVector = new double [col] ;
mxArray2double_vectorReal(mx_normalRandVector, db_normalRandVector) ;

/* print out */
cout << endl ;
double standard_deviation = sqrt(variance) ;

cout << "Normal random numbers :" << endl ;
for (i=0; i<col; i++) {
    cout << mean_mu + standard_deviation*db_normalRandVector[i] << "\t" ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_row) ;
mxDestroyArray(mx_col) ;
mxDestroyArray(mx_normalRandVector) ;

delete [] db_normalRandVector ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 6 by using the function myrandn(..).

Listing code

---

```

void Test::normalRandom_vector2 () {

/* Generate a vector of normal random numbers at
particular mean and variance */

```



```

int row = 1 ;
int col = 5 ;

double mean_mu = 0.56 ;
double variance = 0.12 ;
int i ;

/* declare mxArray variables */
mxArray mw_row(1, 1, mxDOUBLE_CLASS) ;
mw_row(1,1) = row ;

mxArray mw_col(1, 1, mxDOUBLE_CLASS) ;
mw_col(1,1) = col ;

/* call an implemental function */
mxArray mw_normalRandVector (row, col, mxDOUBLE_CLASS) ;
myrandn(1, mw_normalRandVector, mw_row, mw_col);

/* convert back to C/C++ double */
double * db_normalRandVector = new double [col] ;
mxArray2double_vectorReal(mw_normalRandVector, db_normalRandVector) ;

/* print out */
cout << endl ;
double standard_deviation = sqrt(variance) ;

cout << "Normal random numbers :" << endl ;
for (i=0; i<col; i++) {
    cout << mean_mu + standard_deviation*db_normalRandVector[i] << "\t" ;
}
cout << endl ;

/* free memories */
delete [] db_normalRandVector ;

}

```

---

end code

### 16.2.3 Generating a Matrix of Normal Random Numbers with mean=0 and variance=1

#### Problem 7

**input**     A row number  $m = 8$  and a column number  $n = 5$

**output**    Generating a matrix (size  $m \times n$ ) of normal random numbers  
with specified properties:

mean  $\mu = 0$

variance  $\sigma^2 = 1$

#### A. FOR C SHARED LIBRARY

The following is the code to solve Problem 7 by using the function `mlfMyrandn(..)`.

Listing code

---

```
void Test::normalRandom_matrix () {

int row = 8 ;
int col = 5 ;
int i, j ;

/* step 1 : declare mxArray variables */
mxArray *mx_row = NULL ;
mxArray *mx_col = NULL ;
mxArray *mx_normalRandMatrix = NULL ;

/* step 2 : assign memory */
mx_row = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_col = mxCreateDoubleMatrix(1, 1, mxREAL) ;
mx_normalRandMatrix = mxCreateDoubleMatrix(row, col, mxREAL) ;

/* step 3 : convert C/C++ double to mxArray */
double2mxArray_scalarReal (row, mx_row) ;
double2mxArray_scalarReal (col, mx_col) ;

/* step 4 : call an implemental function */
mlfMyrandn(1, &mx_normalRandMatrix, mx_row, mx_col);
```

```

/* step 5 : convert back to C/C++ double */
double ** db_normalRandMatrix = new double* [row] ;
for (i=0; i<row; i++) {
    db_normalRandMatrix[i] = new double [col] ;
}

mxArray2double_matrixReal(mx_normalRandMatrix, db_normalRandMatrix) ;

/* print out */
cout << endl ;
cout << "The matrix of normal random numbers from 0 to 1 : " << endl ;
printMatrix(db_normalRandMatrix, row, col) ;
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_row) ;
mxDestroyArray(mx_col) ;
mxDestroyArray(mx_normalRandMatrix) ;

delete [] db_normalRandMatrix ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 7 by using the function `myrandn(..)`.

Listing code

---

```

void Test::normalRandom_matrix () {

    int row = 8 ;
    int col = 5 ;
    int i, j ;

    /* declare mxArray variables */
    mxArray mw_row(1, 1, mxDOUBLE_CLASS) ;
    mw_row(1,1) = row ;

```

```

mwArray mw_col(1, 1, mxDOUBLE_CLASS) ;
mw_col(1,1) = col ;

/* call an implemental function */
mwArray mw_normalRandMatrix(row, col, mxDOUBLE_CLASS) ;
myrandn(1, mw_normalRandMatrix, mw_row, mw_col);

/* convert back to C/C++ double */
double ** db_normalRandMatrix = new double* [row] ;
for (i=0; i<row; i++) {
    db_normalRandMatrix[i] = new double [col] ;
}

mwArray2double_matrixReal(mw_normalRandMatrix, db_normalRandMatrix) ;

/* print out */
cout << endl ;
cout << "The matrix of normal random numbers from 0 to 1 : " << endl ;
printMatrix(db_normalRandMatrix, row, col) ;
cout << endl ;

/* free memories */
delete [] db_normalRandMatrix ;

}

```

---

end code

#### 16.2.4 Generating a Matrix of Normal Random Numbers with mean= $a$ and variance= $b$

##### Problem 8

**input**     A row number  $m = 8$  and a column number  $n = 5$

**output**    Generating a matrix (size  $m \times n$ ) of normal random numbers  
with specified properties:

mean  $\mu = 0.56$

variance  $\sigma^2 = 0.12$

## A. FOR C SHARED LIBRARY

The following is the code to solve Problem 8 by using the function `mlfMyrandn(..)`.

Listing code

---

```
void Test::normalRandom_matrix2 () {

    int row = 8 ;
    int col = 5 ;
    int i, j ;

    double mean_mu = 0.56 ;
    double variance = 0.12 ;

    double standard_deviation = sqrt (variance) ;

    /* step 1 : declare mxArray variables */
    mxArray *mx_row = NULL ;
    mxArray *mx_col = NULL ;
    mxArray *mx_normalRandMatrix = NULL ;

    /* step 2 : assign memory */
    mx_row = mxCreateDoubleMatrix(1, 1, mxREAL) ;
    mx_col = mxCreateDoubleMatrix(1, 1, mxREAL) ;
    mx_normalRandMatrix = mxCreateDoubleMatrix(row, col, mxREAL) ;

    /* step 3 : convert C/C++ double to mxArray */
    double2mxArray_scalarReal (row, mx_row) ;
    double2mxArray_scalarReal (col, mx_col) ;

    /* step 4 : call an implemental function */
    mlfMyrandn(1, &mx_normalRandMatrix, mx_row, mx_col);

    /* step 5 : convert back to C/C++ double */
    double ** db_normalRandMatrix = new double* [row] ;
    for (i=0; i<row; i++) {
        db_normalRandMatrix[i] = new double [col] ;
    }
}
```

```

mxArray2double_matrixReal(mx_normalRandMatrix, db_normalRandMatrix) ;

/* print out */
cout << endl ;
cout << "The matrix of normal random numbers " ;
cout << "at specified mean and variance" << endl ;
for (i=0; i<row; i++) {
    for (j=0; j<col; j++) {
cout << mean_mu + standard_deviation*db_normalRandMatrix[i][j] << "\t" ;
    }
    cout << endl ;
}
cout << endl ;

/* step 6 : free memories */
mxDestroyArray(mx_row) ;
mxDestroyArray(mx_col) ;
mxDestroyArray(mx_normalRandMatrix) ;

delete [] db_normalRandMatrix ;

}

```

---

end code

## B. FOR C++ SHARED LIBRARY

The following is the code to solve Problem 8 by using the function `myrandn(..)`.

Listing code

---

```

void Test::normalRandom_matrix2 () {

int row = 8 ;
int col = 5 ;
int i, j ;

double mean_mu = 0.56 ;
double variance = 0.12 ;

```

```

double standard_deviation = sqrt (variance) ;

/* declare mxArray variables */
mxArray mw_row(1, 1, mxDOUBLE_CLASS) ;
mw_row(1,1) = row ;

mxArray mw_col(1, 1, mxDOUBLE_CLASS) ;
mw_col(1,1) = col ;

/* call an implemental function */
mxArray mw_normalRandMatrix(row, col, mxDOUBLE_CLASS) ;
myrandn(1, mw_normalRandMatrix, mw_row, mw_col);

/* convert back to C/C++ double */
double ** db_normalRandMatrix = new double* [row] ;
for (i=0; i<row; i++) {
    db_normalRandMatrix[i] = new double [col] ;
}

mxArray2double_matrixReal(mw_normalRandMatrix, db_normalRandMatrix) ;

/* print out */
cout << endl ;
cout << "The matrix of normal random numbers " ;
cout << "at specified mean and variance" << endl ;
for (i=0; i<row; i++) {
    for (j=0; j<col; j++) {
cout << mean_mu + standard_deviation*db_normalRandMatrix[i][j] << "\t" ;
    }
    cout << endl ;
}
cout << endl ;

/* step 6 : free memories */
delete [] db_normalRandMatrix ;

}

```

---

end code





**Part III:**

**MATLAB Engine:**

**Calling MATLAB Workspace in**

**C/C<sup>++</sup> Functions**

**MEX-File: Calling C Functions in**

**MATLAB Workspace**

**Generating Stand-Alone**

**Applications from MATLAB**

**M-Files**



## Chapter 17

# Calling MATLAB Workspace in C/C++ Functions

This chapter describes how to call the MATLAB workspace to perform particular tasks from a C/C++ function. When performing tasks in the MATLAB workspace, we need to transfer inputs from a C/C++ function to the MATLAB workspace and then transfer outputs from the MATLAB workspace back to the C/C++ function. In this chapter, scalars, vectors, and matrixes are used as the function inputs/outputs in the example codes.

### 17.1 Calling MATLAB Workspace with Input/Output as a Scalar

#### Problem 1

**input** Two given numbers  $a = 1.2$  and  $b = 2.5$

#### **output**

- . Call the MATLAB workspace to perform the task,  $c = a + b$ , in a C++ function
- . Get the result  $c$  in the MATLAB workspace and transfer back to the C++ function

The following is the code to solve Problem 1 by using MATLAB Engine in MATLAB Compiler 4.

#### Listing code

---

```
/* Example.cpp */  
/*  
showing the purpose of calling the MATLAB workspace to perform  
a particular task.
```

Procedure:

1. convert C/C++ double to mxArray
2. convert an mxArray name to a new name for using in the MATLAB workspace
3. ask the MATLAB workspace performing the particular tasks
4. get results in the MATLAB workspace then convert back to mxArray
5. convert the results mxArray to C++ double

\*/

```
#pragma warning(disable : 4995)
```

```
#include <iostream.h>
```

```
#include "Example.h"
```

```
int main() {
```

```
    cout << "Using MATLAB Engine to call the MATLAB workspace " << endl ;
```

```
    cout << "performing the particular tasks " << endl  ;
```

```
    Test obj ;
```

```
    obj.SimplePlus() ;
```

```
    obj.LinearSolve() ;
```

```
    return 0 ;
```

```
}
```

---

```
/* Example.h */
```

```
#include "engine.h"
```

```
#include "mxUtilityCompilerVer4.h"
```

```
class Test {
```

```
public:
```

```
void SimplePlus () ;
```

```

Engine *ep ;

Test () {

    if (!(ep = engOpen(NULL))) {
        cout<<"Can't start MATLAB engine, check again!";
        exit(-1);
    }

}

~Test () {

    engClose(ep) ;

}

} ;

/* ***** */
void Test::SimplePlus() {

    double db_a = 1.2 ;
    double db_b = 2.5 ;

    // step1: declare mxArray variables to use
    mxArray *mx_a = NULL ;
    mxArray *mx_b = NULL ;
    mxArray *mx_c = NULL ;

    // step2: create mxArray variables
    mx_a = mxCreateDoubleMatrix(1, 1, mxREAL) ;
    mx_b = mxCreateDoubleMatrix(1, 1, mxREAL) ;
    mx_c = mxCreateDoubleMatrix(1, 1, mxREAL) ;

    // step3: convert C++ double to mxArray
    double2mxArray_scalarReal(db_a, mx_a) ;
    double2mxArray_scalarReal(db_b, mx_b) ;

```

```

// step4: convert name mx_a to ml_a to use in the MATLAB workspace
engPutVariable(ep, "ml_a", mx_a)      ;
engPutVariable(ep, "ml_b", mx_b)      ;

// step5: perform a task in MATLAB workspace
engEvalString( ep, "ml_c = ml_a + ml_b ;" ) ;

// step6: get result in Matlab workspace then convert to mxArray
mx_c = engGetVariable(ep, "ml_c" )    ;

// step7: convert mxArray to C++ double
double db_c = mxArray2double_scalarReal(mx_c) ;

// step8: close Matlab workspace
engEvalString(ep, "close") ;

// step9: print out
cout << "The result from the simple plus:"<< endl ;
cout << db_c << endl ;

}

```

---

end code

See the code of the file `mxUtilityCompilerVer4.h` in Chapter 7.

## 17.2 Calling MATLAB Workspace with Input/Output as a Vector and a Matrix

### Problem 2

**input**      a matrix **A** and a vector **b**,

$$\mathbf{A} = \begin{bmatrix} 1.1 & 5.6 & 3.3 \\ 4.4 & 12.3 & 6.6 \\ 7.7 & 8.8 & 9.9 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 12.5 \\ 32.2 \\ 45.6 \end{bmatrix}$$

**output**

. Call the MATLAB workspace to perform the tasks as following in a C++ function:

- a) Finding the solution  $\mathbf{x}$  of linear system equations  $\mathbf{Ax} = \mathbf{b}$
- b) Calculating the upper matrix U in the LU decomposition method
- c) Getting results in the MATLAB workspace and converting to C++ double

The following is the code to solve Problem 2 by using MATLAB Engine in MATLAB Compiler 4.

#### Listing code

---

```
void Test::LinearSolve() {

    int i ;

    double A[3][3] = { {1.1, 5.6, 3.3}, {4.4, 12.3, 6.6} , { 7.7, 8.8, 9.9} } ;
    double b[3] = { 12.5, 32.2 , 45.6 } ;

    int row = 3 ;
    int col = 3 ;

    double **db_matrixU      ;
    double  *db_vectorX      ;

    db_matrixU = new double*[row] ;
    db_vectorX = new double [row] ;

    for(i=0; i<row; i++) {
        db_matrixU[i] = new double [col] ;
    }

    /* declare mxArray variables */
    mxArray *mx_A      = NULL ;
    mxArray *mx_b       = NULL ;
    mxArray *mx_vectorX = NULL ;
    mxArray *mx_U       = NULL ;

    /* assign memory */
    mx_A      = mxCreateDoubleMatrix(row, col, mxREAL)      ;
    mx_U      = mxCreateDoubleMatrix(row, col, mxREAL)      ;
    mx_b      = mxCreateDoubleMatrix(row, 1 , mxREAL)       ;
    mx_vectorX = mxCreateDoubleMatrix(row, 1 , mxREAL)       ;
```

```

/* convert C/C++ double to mxArray */
double2mxArray_vectorReal(b, mx_b) ;
double2mxArray_matrixReal(&A[0][0], mx_A) ;

/* begin performing tasks in the Matlab workspace

Matlab workspace tasks : Solve  $Ax = b$ , and get
a vector  $x$  and an upper matrix  $U$ .
The problem  $Ax = b$  is solved here as the purpose
of showing the use of the MATLAB workspace in a C/C++ function. */

engPutVariable(ep, "ml_A", mx_A)      ;
engPutVariable(ep, "ml_b", mx_b)      ;

engEvalString(ep, " ml_vectorX = mldivide(ml_A, ml_b) ; " ) ;
engEvalString(ep, " [ml_L, ml_U, ml_P] = lu( ml_A ) ; " ) ;

/* end tasks in the Matlab workspace */

/* get results in the Matlab workspace then convert to mxArray */
mx_vectorX = engGetVariable(ep, "ml_vectorX" ) ;
mx_U        = engGetVariable(ep, "ml_U" )      ;

/* convert mxArray to C/C++ double */
mxArray2double_vectorReal(mx_vectorX, db_vectorX) ;
mxArray2double_matrixReal(mx_U, db_matrixU) ;

/* close Matlab workspace */
engEvalString(ep, "close") ;

/* print out */
cout << "Solution of the equation  $Ax = b$  is: " << endl ;

    for (i=0; i<row; i++) {
        cout<< *( db_vectorX + i) << endl ;
    }

cout<< "The upper matrix U : " << endl;
printMatrix(db_matrixU, row, col) ;

```



```

/* free memory */
mxDestroyArray(mx_A) ;
mxDestroyArray(mx_b) ;
mxDestroyArray(mx_vectorX) ;
mxDestroyArray(mx_U) ;

delete [] db_matrixU ;
delete [] db_vectorX ;

}

```

---

end code

---

### 17.3 Generating a MATLAB Graphic from a C/C++ Function

In this section we will directly create a MATLAB graphic by performing a plotting task in the MATLAB workspace. The steps are:

1. Set up a project in Microsoft Visual C++ .Net (MSVC.Net) for working with MATLAB Compiler 4 as described in Chapter 5.
2. Write a code to call the MATLAB workspace to perform the plotting task in a C++ function.

Following is the code to create the graphic. The figure is created as shown in Fig. 17.1.

Listing code

---

```

void Test:: MatlabEnginePlot() {

double X[3] = { 12.5, 32.2 , 45.6 } ;
double Y[3] = { 12.5, 32.2 , 45.6 } ;

int vectorSize = 3 ;

/* step1: declare mxArray variables */
mxArray *mx_X      = NULL ;
mxArray *mx_Y      = NULL ;

/* step2: create mxArray vectors and mxArray matrixes */
mx_X      = mxCreateDoubleMatrix(vectorSize, 1 , mxREAL)      ;
mx_Y      = mxCreateDoubleMatrix(vectorSize, 1 , mxREAL)      ;

```

```

// step3: Convert C/C++ double to mxArray
double2mxArray_vectorReal(X, mx_X) ;
double2mxArray_vectorReal(Y, mx_Y) ;

/* step4: transfer name mx_X to the name, ml_X */
engPutVariable(ep, "ml_X" , mx_X ) ;
engPutVariable(ep, "ml_Y" , mx_Y ) ;

/* step5: perform tasks in the Matlab workspace */
engEvalString(ep, " plot(ml_X, ml_Y , 'r'); " ) ;

/* keep the figure */
cout << "Hit return to close the figure and continue" ;
cout << endl ;
fgetc(stdin);

/* step6: close the Matlab workspace */
engEvalString(ep, "close") ;

/* step7: free memory */
mxDestroyArray(mx_X) ;
mxDestroyArray(mx_Y) ;

}

```

---

end code

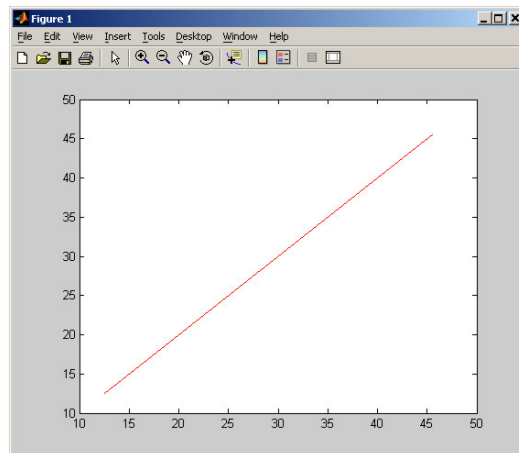


Figure 17.1: The figure is created by using the MATLAB workspace

**Remark**

- With directly creating the graphic by MATLAB Engine, you can use the graphic features as Insert, Tool, etc.
- You need to hit Enter key to close the figure as shown in the line of the code **fgetc(stdin)**.



## Chapter 18

# MEX-Files, Calling a C Function in MATLAB Workspace

In working with the MATLAB workspace, we can call an existing C function by writing a MEX-function for it. This MEX-function then will be called into the MATLAB workspace. The procedure to call a C function from the MATLAB workspace is:

1. Write the MEX-function for the existing C function.
2. Write the command in the MATLAB space to generate an dll-file that contains the MEX-function.
3. Call the generated MEX-function from the MATLAB workspace.

This chapter describes how to write a MEX-file for an existing C function and use this function in MATLAB. The following sections are for working on the C functions that have input/output as scalars, vectors, or matrixes.

### 18.1 MEX-File with Input/Output as Scalars

Suppose that there is an existing C function `mysquare(..)` in the file `mysquare.c` (see the code below and note that the function name and the file name are the same). To call this function in the MATLAB workspace, we write the command, `mex mysquare.c ;`, MATLAB will generate a file `mysquare.dll` that contains the function `mysquare(..)`. This function `mysquare(..)` will be called into the MATLAB workspace to calculate the square value. The example code in the MATLAB workspace is:

```
>> mex mysquare.c ;  
>> x = 1.2 ;  
>> square = mysquare(x) ;
```

```
>> square
```

```
square =
```

```
1.4400
```

```
>>
```

The following is the MEX-function code of the function *double mysquare(double x)*.

#### Listing code

---

```
/* mysquare.c */

#include "mex.h"
double mysquare(double x)
{
    double y = x*x ;
    return y ;
}

/* ***** */
void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

    /* declare double variables */
    double *db_py ;
    double db_x ;

    /* assign value for input */
    db_x = mxGetScalar( prhs[0] ) ;

    /* assign memory for return value */
    plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL) ;

    /* assign memory for output */
    db_py = mxGetPr( plhs[0] ) ;

    /* assign value for output */
    *db_py = mysquare( db_x ) ;
}
```

---

end code

## 18.2 MEX-File with Input/Output as Vectors

In this section, we will show how to write a MEX-function for an existing C function that has the input/output as vectors. This function will then be called into the MATLAB workspace.

Suppose that there is an existing C function `vectortimes(..)` in the `vectortimes.c` file (note that the C function name and the C file name are the same). To call this function in the MATLAB workspace we write the command, `mex vectortimes.c ;`, MATLAB will generate a file `vectortimes.dll` that contains the function `vectortimes(..)`. This function `vectortimes(..)` will be called into the MATLAB workspace to calculate the value. The example code in the MATLAB workspace is:

```
>> mex vectortimes.c ;
>> X = [1.1 2.2 3.3 4.4 ] ;
>> factor = 3.4 ;
>> Y = vectortimes(X, factor) ;
>> Y

Y =

    3.7400    7.4800   11.2200   14.9600
```

The following is the MEX-function code of the function `void vectortimes(..)`.

Listing code

---

```
/* vectortimes.c */
#include "mex.h"

void vectortimes( double *x, int vectorSize, double factor, double *y ) {

    /* calculate y = k*x */
    int i ;

    for (i=0; i<vectorSize; i++) {
        y[i] = x[i]*factor ;
    }
}
```

```

/* ***** */
void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{

/* step 1 : declare double variables */
    double *db_vectorY ;
    double *db_vectorX ;

    double db_factor ;
    int vectorSize ;

/* step 2 : assign value for input */
    db_vectorX = mxGetPr( prhs[0] ) ;
    db_factor = mxGetScalar( prhs[1] ) ;
    vectorSize = mxGetN( prhs[0] ) ;

/* step 3 : assign memory for return value */
    plhs[0] = mxCreateDoubleMatrix(1, vectorSize, mxREAL) ;

/* step 4 : assign memory for output */
    db_vectorY = mxGetPr( plhs[0] ) ;

/* step 5 : assign value for output */
    vectortimes( db_vectorX, vectorSize, db_factor, db_vectorY ) ;

}

```

---

end code

## 18.3 MEX-File with Input/Output as Matrixes

In this section, we will show how to write a MEX-function for a C function that has the input/output as matrixes. This function then will be called into MATLAB workspace.

Suppose that there is an existing C function, *matrixtimes(..)*, in the file *matrixtimes.c* (note that C function name and C file name are the same). To call this function in the MATLAB workspace we write the command, `mex matrixtimes.c ;`, MATLAB will generate a file *matrixtimes.dll* that contains the function *matrixtimes(..)*. This function *matrixtimes(..)* will be called from the MATLAB workspace to calculate the value. The example code in the MATLAB



workspace is:

```
>> mex matrixtimes.c ;
>>
>> A = [ 1.1 2.2 3.3 ; 4.4 5.5 6.6 ]
```

A =

```
    1.1000    2.2000    3.3000
    4.4000    5.5000    6.6000
```

```
>>
```

```
>> k = 5.5 ;
```

```
>>
```

```
>> B = matrixtimes(A, k)
```

B =

```
    6.0500   12.1000   18.1500
   24.2000   30.2500   36.3000
```

```
>>
```

The following is the MEX-function code of the function *void matrixtimes(..)*.

Listing code

---

```
/* matrixtimes.c */

#include <stdlib.h>
#include "mex.h"

/*
purpose : Writing the MEX function
input   : . scalar
          . matrix

output  : . matrix
*/
```

```

void DoubleMatrixToDoubleVector(double **db_matrix, int row, int col, double *db_vector)
{
    int i, j, index ;
        for(j=0; j<col; j++) {
            for(i=0; i<row; i++) {
                index = j*row + i ;
                db_vector[index] = db_matrix [i][j] ;

            }
        }
}

/* ***** */
void DoubleVectorToDoubleMatrix(double **db_matrix, int row, int col, double *db_vector)
{
    int i, j, index ;

        for(j=0; j<col; j++) {
            for(i=0; i<row; i++) {
                index = j*row + i ;
                db_matrix [i][j] = db_vector[index] ;

            }
        }
}

/* ***** */
/* ***** */

void matrixtimes( double **x, int row, int col, double factor, double **y )
{
    /* calculate y = x*k */
    int i, j ;
        for (i=0; i<row; i++) {
            for (j=0; j<col; j++) {
                y[i][j] = x[i][j]*factor ;

            }
        }
}

/* ***** */

```

```

void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
/* procedure :
input prhs[0] --> vectorX --> matrixX --> call matrixtimes(..) get matrixY
               --> matrixY --> vectorY --> output plhs[0]   */

int i ;
int    row ;
int    col ;

/* step 1 : declare double variables */
double **db_matrixY ;
double **db_matrixX ;
double  *db_vectorX ;
double  *db_vectorY ;

double  db_factor ;

/* step 2 : assign value for input */
db_vectorX = mxGetPr( prhs[0] ) ;

row = mxGetM( prhs[0] ) ;
col = mxGetN( prhs[0] ) ;

db_factor = mxGetScalar( prhs[1] ) ;

/* step 3 : assign memory for matrix */
db_matrixX = (double **)malloc( row * sizeof(double * ) );
db_matrixY = (double **)malloc( row * sizeof(double * ) );

for(i = 0; i < row; i++) {
db_matrixX[i] = (double *)malloc( col * sizeof(double) );
db_matrixY[i] = (double *)malloc( col * sizeof(double) );
}

/* step 4 : transfer C vector to C matrix. The matrix is used as
a buffer input in calling function matrixtimes */

DoubleVectorToDoubleMatrix(db_matrixX, row, col, db_vectorX) ;

```

```

/* step 5 : assign memory for return value */
    plhs[0] = mxCreateDoubleMatrix(row, col, mxREAL)    ;

/* step 6 : assign memory for output */
    db_vectorY = mxGetPr( plhs[0] ) ;

/* step 7 : Calculate the value of output */

    matrixtimes( db_matrixX, row, col, db_factor, db_matrixY ) ;

/* step 8: transfer back C matrix to C vector,
    for db_vectorY=result in mex-file */

    DoubleMatrixToDoubleVector( db_matrixY, row, col, db_vectorY ) ;

/* free memory */

    for(i = 0; i < row; i++) {
        free(db_matrixX[i]);
        free(db_matrixY[i]);
    }

    free(db_matrixX);
    free(db_matrixY);

/*
Note: Matrixes db_matrixX and db_matrixY are used as the buffer variables.
    Therefore their memories are need to free after using.
*/

}

```

---

end code

## 18.4 MEX-Function Analysis

Going through three above examples will give us a general understanding of writing a MEX-function. This section is a more in-depth analysis of the MEX-function. The MEX-function features are:

1. The MEX-function has the form,

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]);
```

2. The arguments in this *void mexFunction(..)* are:

**nlhs**        is number of the left hand side representing number of outputs  
**\*plhs[]**    is pointer to the left hand side representing the outputs,  
                     plhs[0] is first output  
                     plhs[1] is second output  
                     ...  
                     plhs[n] is  $(n + 1)^{th}$  output  
**nrhs**        is number of the right hand side representing number of inputs  
**\*prhs[]**    is pointer to the right hand side representing the inputs,  
                     prhs[0] is first input  
                     prhs[1] is second input  
                     ...  
                     prhs[n] is  $(n + 1)^{th}$  input



Figure 18.1: The MEX-Function in the mathematical form

3. The matrix input  $prhs[i]$  and matrix output  $plhs[i]$  of *mexFunction(..)* has the relationship with C programming in a vector form (*double\**), **not** in a matrix form (*double\*\**). Therefore, we need to transfer the matrix form to the vector form before assigning the matrix to the input or output of *mexFunction(..)*. For example in Section 18.3, the code to perform this task is:

```
/* step 2 : assign value for input */
db_vectorX = mxGetPr( prhs[0] ) ;
...
```

```

/* step 6 : assign memory for output */
db_vectorY = mxGetPr( plhs[0] ) ;

```

4. The function,

```

void matrixtimes( double **x, int row, int col, double factor, double **y )

```

, in the file `matrixtimes.c` is different from the function that we called in the MATLAB workspace, `matrixtimes(A, k)`. The difference is in the arguments. And note that, the function that we call in the MATLAB workspace, `matrixtimes(A, k)`, is established from `mexFunction(..)`. In `mexFunction(..)`, we determine the inputs and outputs for the function, `B = matrixtimes(A, k)`, that is called in the MATLAB workspace.

## Chapter 19

# Stand-Alone Applications

MATLAB Compiler 4 provides the method to create stand-alone applications from either entirely of M-files or some combination of M-files, MEX-files, and C/C++ source code files. These stand-alone applications then can be used in the target machine which doesn't have the MATLAB software. This chapter describes how to generated and use the stand-alone applications. These applications are executable files and generated from the M-files.

### 19.1 Installing MATLAB Component Runtime to a target machine

To use the stand-alone applications, we need to install MATLAB Component Runtime (MCR) to the target machine. The steps of installing are:

1. Copy the file MCRInstaller.exe to your target machine (MCRInstaller.exe is located in the directory `..\MATLAB7\toolbox\compiler\deploy\win32\`). Double-click it to install.

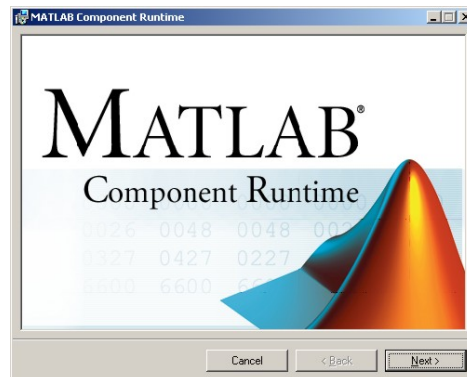


Figure 19.1: MCR installation

2. Click Next, Fig. 19.2 appears.



Figure 19.2: MCR installation (continued)

3. Click Next, Fig. 19.3 appears. You can choose any directory to install. In here we choose a default directory.

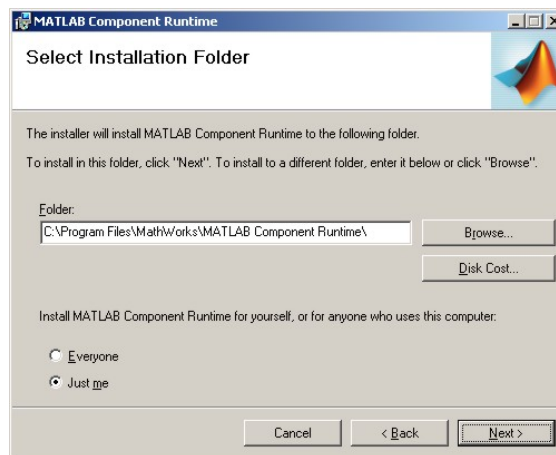


Figure 19.3: MCR installation (continued)

4. Click Next to install.



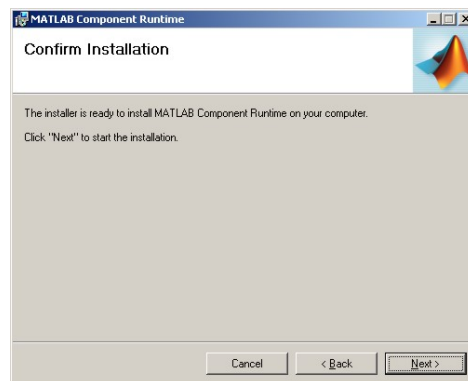


Figure 19.4: MCR installation (continued)

In Fig. 19.3 choosing **Everyone** or **Just me** is not effective on your stand-alone application. The following sections we'll write several examples of M-files for stand-alone applications. We'll compile its to get the executable files, then use in Command Prompt. There are some limitations for M-files in generating stand-alone applications, we refer to page 1-19 of [2].

## 19.2 Stand-Alone Application for an Addition Operator

In this section we'll write a stand-alone application for a simple addition operator. The steps of this procedure are:

1. Writing the M-file myaddition.m to generate the executable file myaddition.exe as follows:

```
function myaddition(a, b)

if (ischar(a))
    a = str2num(a);
end

if (ischar(b))
    b = str2num(b);
end

c = a + b
```

2. Write the command in Command Prompt (see Fig. 19.5):

```
mcc -m myaddition.m
```

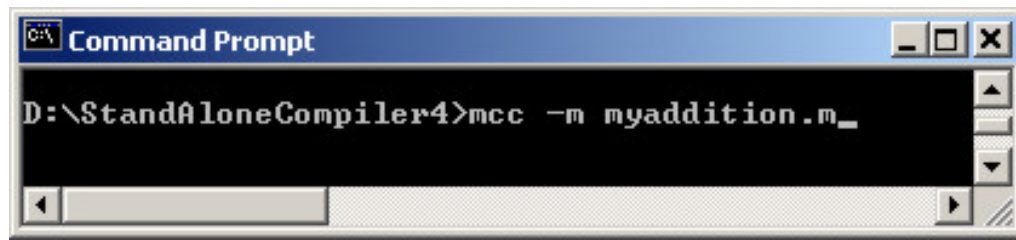


Figure 19.5: Stand-alone compile command

3. MATLAB Compiler will create five files:

```
myaddition.ctf
myaddition.exe
myaddition.m
myaddition_main.c
myaddition_mcc_component_data.c
```

4. Copy two files myaddition.ctf and myaddition.exe to your target machine.
5. Execute this stand-alone application by write the command in Command Prompt (see Fig. 19.6): myaddition.exe 1.2 3.4

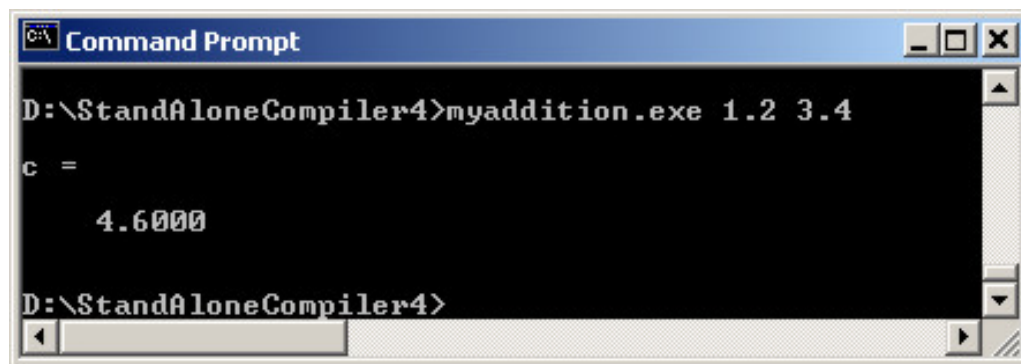


Figure 19.6: Execute command

#### Note

- You will receive a warning as following, but this is a bug and MathWorks Inc. will fix it. This warning does not affect the compilation or execution of the program. See this warning at:  
[www.mathworks.com/support/solutions/data/1-ON0NN.html?solution=1-ON0NN](http://www.mathworks.com/support/solutions/data/1-ON0NN.html?solution=1-ON0NN).  
The warning is:

```
Warning: C:\Program Files\MathWorks\MATLAB Component Runtime\v70\toolbox\
local\pathdef.m not found.
Toolbox Path Cache is not being used. Type 'help toolbox_path_cache'
for more info.
```

- The inputs of the executable file are separated by the space(s).
- The inputs of the executable file are recognized as characters **not a number**. Therefore in the M-file myaddition.m we wrote lines of code to convert these characters to a number before using.

The procedure for examples in the next sections is similar in this section.

## 19.3 Stand-Alone Application for Linear Equations

In this subsection, we'll write a stand-alone application for the linear equations to solve the solution  $\mathbf{x}$  from  $\mathbf{Ax} = \mathbf{b}$ . The input values of a vector  $\mathbf{b}$  and a matrix  $\mathbf{A}$  are stored in the files `vectorb.dat` and `matrixa.dat`, respectively. The output solution  $\mathbf{x}$  is written to the file `mysolution.dat`.

In this example, we have the input files as follows.

`vectorb.dat`

12.5

32.2

45.6

`matrixa.dat`

1.1      5.6      3.3

4.4      12.3      6.6

7.7      8.8      9.9

The steps to create this stand-alone application are:

1. Writing the M-file mylinear.m to generate the executable file mylinear.exe as follows:

```
function mylinear(matrixsize)

if (ischar(matrixsize))
    matrixsize = str2num(matrixsize) ;
end
```

```

vectorfile = fopen('vectorb.dat', 'r') ;
b = fscanf(vectorfile, '%f', matrixsize) ;
fclose(vectorfile) ;

matrixfile = fopen('matrixa.dat', 'r') ;
A = fscanf(matrixfile, '%f', [matrixsize matrixsize]) ;
fclose(matrixfile) ;

% MATLAB reads a matrix in column major
% we need A as a matrix in row major (C/C++ format)
A = A' ;
x = A\b ;

fid = fopen('mysolution.dat','w');
fprintf(fid, '%f\t', x);
fclose(fid) ;

```

2. Write the command in Command Prompt to generated files of the stand-alone application:  
`mcc -m mylinear.m`
3. Copy two files `mylinear.ctf` and `mylinear.exe` to the target machine.
4. Execute this stand-alone application by write the command in Command Prompt:  
`mylinear.exe 3`
5. MATLAB Component Runtime in the target machine will execute the file `mylinear.exe` and create the file `mysolution.dat` which includes the solution `x`.

## 19.4 Stand-Alone Application for Using Matlab Plots

In this subsection, we'll write a stand-alone application for using MATLAB plots. The input values of `x` and `y` are stored in files `xdata.dat` and `ydata.dat`, respectively. In this example, we have the input files as follows.

`xdata.dat`

```

1
2
3
4

```

ydata.dat

```
21.1    41.1
22.2    42.2
23.3    43.3
24.4    44.4
```

The steps to create this stand-alone application are:

1. Writing the M-file myplot.m to generate the executable file myplot.exe.

```
function myplot(xdatafile, ydatafile)

x = textread(xdatafile, '%f') ;
[y1, y2] = textread(ydatafile, '%f %f') ;

hold on ;

plot(x,y1) ;
plot(x,y2) ;

title('Figure in Stand-Alone Application') ;
xlabel('x axis') ;
ylabel('y axis') ;

hold off
```

2. Write the command in Command Prompt to generate files for the stand-alone application:  
mcc -m myplot.m
3. Copy two files myplot.ctf and myplot.exe to the target machine.
4. Execute this stand-alone application by write the command in Command Prompt:  
myplot.exe xdata.dat ydata.dat
5. MATLAB Component Runtime in the target machine will execute the file myplot.exe and create the figure as shown in Fig. 19.7.

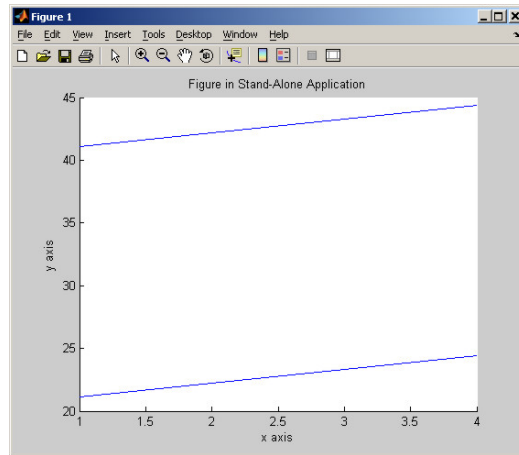


Figure 19.7: Plots in a stand-alone application

## 19.5 Stand-Alone Application for Calculating an Integration

In this subsection, we'll write a stand-alone application for calculating an integration. The steps to create this stand-alone application are:

1. Writing the M-file myquad.m to generate the executable file myquad.exe.

```
function myquad(strfunc, a, b)
F = inline(strfunc) ;
if (ischar(a))
    a=str2num(a);
end

if (ischar(b))
    b=str2num(b);
end

y = quad(F, a, b)
```

2. Write the command in Command Prompt to generated files of the stand-alone application:

```
mcc -m myquad.m
```

3. Copy two files myquad.ctf and myquad.exe to the target machine.

4. Suppose you want to calculate an integration  $I$  in the target machine:

$$I = \int_{0.1}^{1.2} \cos(2x) + x^2 + 1.2x - 0.24$$

To calculate  $I$ , execute the stand-alone application by write the command in Command Prompt:

```
myquad.exe cos(2*x)+power(x,2)+1.2*x-0.24 0.1 1.2
```

MATLAB Component Runtime in the target machine will execute the file `myquad.exe` (Fig. 19.8) to calculate  $I$ .

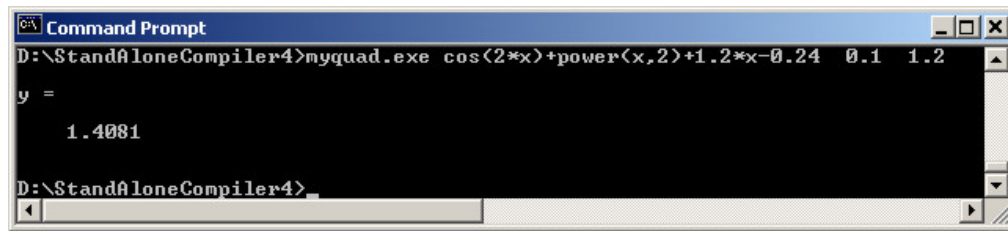


Figure 19.8: Calculating an integration in a stand-alone application

#### Note

- First input variable of `myquad.exe` is an expression string. This expression string **does not** include the space, because the executable file recognizes the space as the symbol of separation of input variables.
- In the expression string, try to use the function name instead of a special character. For example, use the function name `power(..)` instead of the character `^`.





# Bibliography

- [1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992.
- [2] Inc. The MathWork. External Interfaces References. Version 7 The Language of Technical Computing. URL address:  
[www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/apiref.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiref.pdf).
- [3] Inc. The MathWork. MATLAB compiler version 4, user guide. The Language of Technical Computing. URL address:  
[www.mathworks.com/access/helpdesk/help/pdf\\_doc/compiler/compiler4.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/compiler/compiler4.pdf).
- [4] Inc. The MathWork. MATLAB C++ math library version 2.1, The Language of Technical Computing.
- [5] Inc. The MathWork. MATLAB Function Reference. Volume 1, The Language of Technical Computing. URL address:  
[www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/refbook.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/refbook.pdf).
- [6] Inc. The MathWork. MATLAB Function Reference. Volume 2, The Language of Technical Computing. URL address:  
[www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/refbook2.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/refbook2.pdf).
- [7] Inc. The MathWork. MATLAB Function Reference. Volume 3, The Language of Technical Computing. URL address:  
[www.mathworks.com/access/helpdesk/help/pdf\\_doc/matlab/refbook3.pdf](http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/refbook3.pdf).



# Index

- Band diagonal matrix, 140
- call
  - MATLAB workspace, 273
- curve fitting, 161
- differences between C/C++ and MATLAB
  - C/C++, 5
- eigenvalues, 233
- eigenvectors, 233
- experimental data, 188
- Fast Fourier Transform, 215
- features, 3
- function-function, 157, 214
- fzero, 212
- Gaussian distribution, 258
- generate
  - a C function, 4
  - C functions, 47
  - dll-file, 48
- inline function, 151
- integrations, 151
- interpolation, 170, 173, 188
- least-squares, 165
- linear system equations, 115, 119, 123, 277
- LU
  - decompression method, 277
- manuals, 5
- MATLAB Engine, 273
- matrix
  - addition, 93, 96
  - determinant, 102, 103
  - inversion, 104, 105
  - multiplication, 99
  - subtraction, 98
  - transpose, 106
- MEX-file, 283
- MEX-function, 4
- Microsoft Visual C++ 6.0, 21
- mwArray, 61
  - to C++ double type, 69
- mwUtilityCompilerVer4.h, 83
- mxArray, 61
- mxUtilityCompilerVer4.h, 72
- normal random numbers, 258
- projects, 3
- random, 243
- random numbers, 258
- roots
  - of a nonlinear, 212
  - of a polynomial, 205, 207
- set up, 17, 33
- sparse matrix, 126
- stand-alone application, 293
- tested, 4
- testing the project setting, 20
- tridiagonal, 131
- uniform random numbers, 244