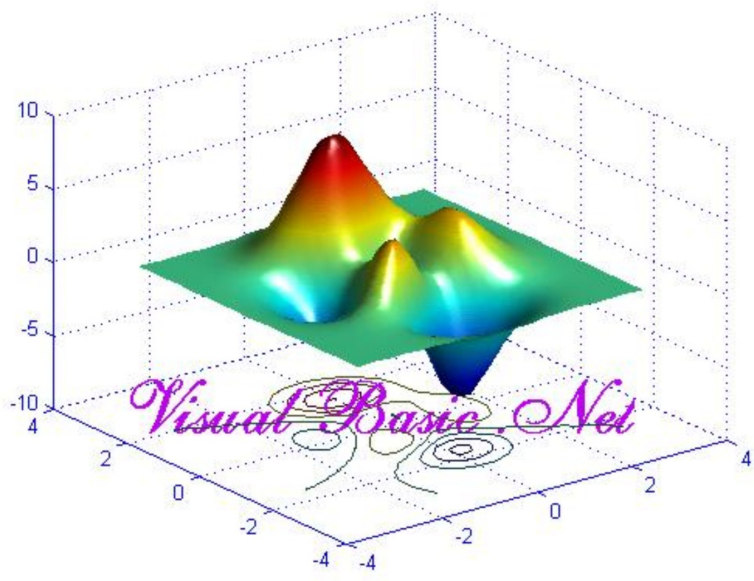
MATLAB® - Visual Basic .Net®

for Engineers

For MATLAB 7.9.0 (R2009b)
& MATLAB Compiler 4.11



This book is a great tutorial for Visual Basic .NET programmers who use MATLAB to develop applications and solutions

MATLAB® Visual Basic .NET® for Engineers

Published by
LePhan Publishing
Mountain View, CA 94043

Library of Congress Cataloging-in-Publication Data
Library of Congress Control Number (LCCN) 2010913990

ISBN 978-1-453-68135-0

Contents

Preface

MATLAB provides the toolboxes MATLAB Compiler and MATLAB Builder for .NET to handle technical problems between MATLAB and programming languages. The common task is to support MATLAB built-in functions for computer programmers in development. Microsoft Corporation has been developing VB programming language in Microsoft Visual .Net and this VB language is a power language, easy to develop, easy to maintain, and other advanced features, especially in the memory handling. VB is one of the programming languages can use classes created from MATLAB M-files. In addition, the MATLAB Builder for .NET toolbox provides a special feature that allows the user to create a Component Object Model (COM) from MATLAB M-files. The generated COM then can be used in the other programming languages that support COM applications. VB is one of the languages that supports COM, so VB programmers can use MATLAB M-files in the wrapper COM to develop applications as stand-alone applications. This book, ***MATLAB Visual Basic .NET for Engineers***, implements the combination of the advances of VB and MATLAB to solve the technical problems. The features of this book are designed to handle the following projects:

- VB functions use MATLAB built-in functions in the class created from MATLAB M-files to solve the mathematical problems
- VB Windows applications use MATLAB built-in functions
- VB functions plot figures from MATLAB Graphics
- VB functions use API functions (calling MATLAB Workspace in VB functions)
- VB functions use MATLAB Curve Fitting Toolbox functions
- VB functions use COMs generated from MATLAB M-files

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

Jack Phan
September 2010

Part I:

Creating and Using MATLAB Functions to Solve Mathematical Problems In VB .NET

Chapter 1

Introduction

1.1 Introduction

MATLAB is mathematical software that includes many toolboxes. `MATLAB Compiler` and `MATLAB Builder for .NET` are the most important toolboxes that supports Visual Basic .NET computer programmers to use MATLAB functions. We can use `MATLAB Builder for .NET` to create a class from MATLAB M-files and the generated functions in this class will then be called by VB functions.

In addition, the `MATLAB Builder for .NET` toolbox provides a special feature that the user can use to create Component Object Model (COM) from MATLAB M-files. The generated COM then can be used in the other programming languages that support COM applications. VB .NET is one of the languages that supports COM, so VB programmers can use MATLAB M-files in the wrapper COM to develop applications as stand-alone applications. This book, ***MATLAB VB .NET for Engineers***, implements the combination of advances of VB and MATLAB to solve common mathematical problems. The features of this book are designed to handle the following projects:

1. VB functions use MATLAB built-in functions in the class created from MATLAB M-files to solve the mathematical problems

2. VB Windows applications use MATLAB built-in functions

3. VB functions plot figures from MATLAB Graphics

4. VB functions use API functions (calling MATLAB Workspace in VB functions)
5. VB functions use MATLAB Curve Fitting Toolbox functions
6. VB functions use COMs generated from MATLAB M-files

1.2 Computer Software and Book Features

MATLAB Compiler has many versions and there are changes of their features in different versions. The example codes in this book are developed, compiled, and tested in Microsoft Visual .Net 2008, MATLAB 7.9 (R2009b), MATLAB Compiler 4.11, and MATLAB Builder for .NET 3.0.2 with Windows XP, Windows Vista, and Windows 7. These examples are intended to establish common work between VB 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 reuses the codes or writes his/her own codes by following the step-by-step procedure while solving the problems.

The most VB 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 all VB programming example codes and M-files in all chapters.

1.3 Reference Manuals

In working with `MATLAB Compiler` and `MATLAB Builder for .NET`, you may need more information to help your tasks. 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/compiler.pdf`
`http://www.mathworks.com/access/helpdesk/help/pdf_doc/compiler/example_guide.pdf`
`http://www.mathworks.com/access/helpdesk/help/pdf_doc/dotnetbuilder/dotnetbuilder.pdf`
`http://www.mathworks.com/access/helpdesk/help/pdf_doc/curvefit/curvefit.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/refbook3.pdf`

If you couldn't find these files at the time you are looking for, The MathWorks Inc. may change the URL of these files, but you can find them somewhere in The MathWorks website www.mathworks.com.

Chapter 2

Using Classes Created From MATLAB Compiler In Visual Basic .NET Applications

This chapter describes how to generate a class from MATLAB M-files by using `MATLAB Compiler` and `MATLAB Builder for .Net` and using it in Microsoft Visual VB .Net 2008. Based on the M-files, `MATLAB Compiler` and `MATLAB Builder for .NET` will generate functions in a class. These generated functions will then be called in VB functions to solve particular problems.

2.1 Choosing a Compiler in Using MATLAB Compiler

Before doing work with MATLAB Compiler, we need to run the command ***mbuild - setup*** to choose a compiler to work with MATLAB Compiler. The following shows process in a computer that chose the compiler of MSVS 2008.

```
>> mbuild -setup
Please choose your compiler for building standalone MATLAB applications:

Would you like mbuild to locate installed compilers [y]/n? y

Select a compiler:
```

```
[1] Lcc-win32 C 2.4.1 in C:\PROGRA~1\MATLAB\R2009b\sys\lcc
[2] Microsoft Visual C++ 2008 SP1 in c:\Program Files\Microsoft Visual Studio 9.0

[0] None

Compiler: 2

Please verify your choices:

Compiler: Microsoft Visual C++ 2008 SP1
Location: c:\Program Files\Microsoft Visual Studio 9.0

Are these correct [y]/n? y

********************************************************************************
  Warning: Applications/components generated using Microsoft Visual Studio
           2008 require that the Microsoft Visual Studio 2008 run-time
           libraries be available on the computer used for deployment.
           To redistribute your applications/components, be sure that the
           deployment machine has these run-time libraries.
********************************************************************************

Trying to update options file:
C:\Users\ComputerNha\AppData\Roaming\MathWorks\MATLAB\R2009b\compopts.bat
From template:
C:\PROGRA~1\MATLAB\R2009b\bin\win32\mbuildopts\msvc90compp.bat

Done . . .
```

2.2 Generating a Class from a MATLAB M-File

The following are the steps of the procedure to generate a class from a MATLAB M-file, `myplus.m`.

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

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

2. On the MATLAB Command Prompt, type the command `deploytool` (Fig. 2.1) to open a dialog ***Deployment Project*** to prepare generating a dll file.

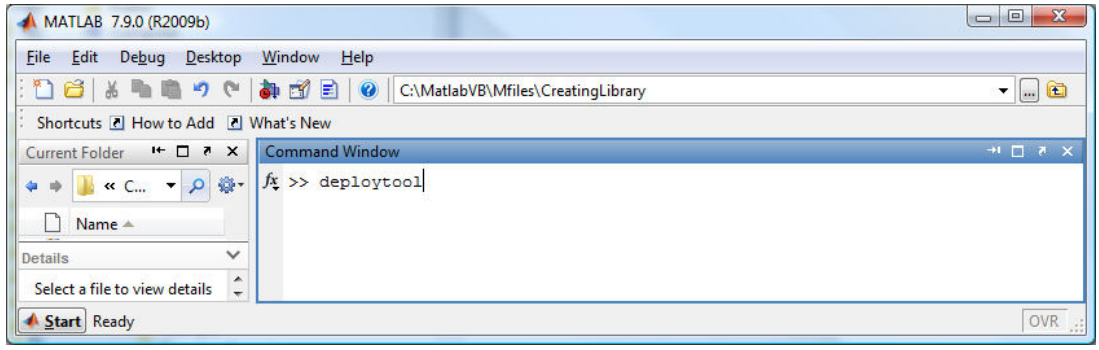


Figure 2.1: Command to start Deployment Project

3. In the dialog ***Deployment Project*** (see Fig. 2.2), name the project `PlusNameSpace` and in ***Location*** choose the directory that contains the file `myplus.m` above. Choose `.NET Assembly` for ***Target***.

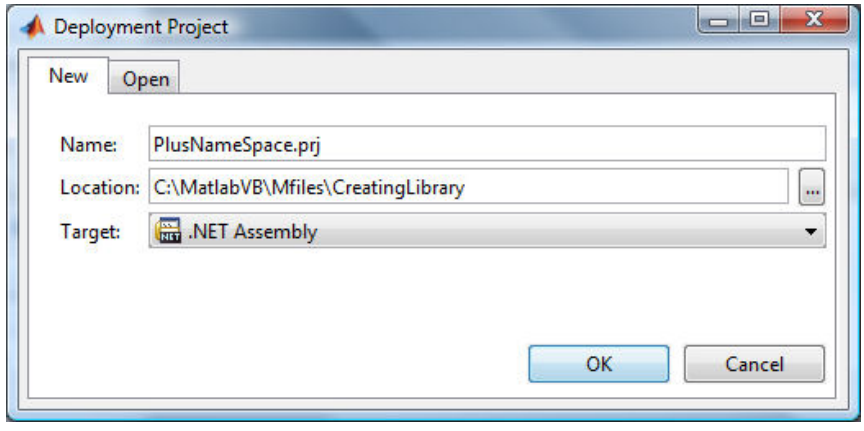


Figure 2.2: Deployment Project dialog

4. In the dialog ***Deployment Project***, click OK. It will pop up a dialog that shows process of the project (Fig. 2.3) and then shows another dialog to add name space, class, and M-files (Fig. 2.4).

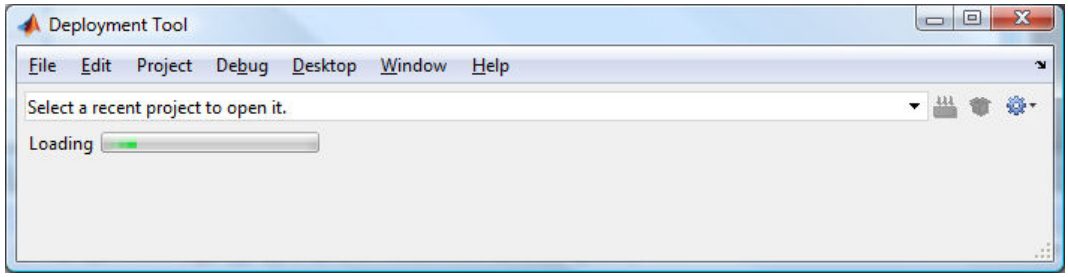


Figure 2.3: Dialog of processing a project

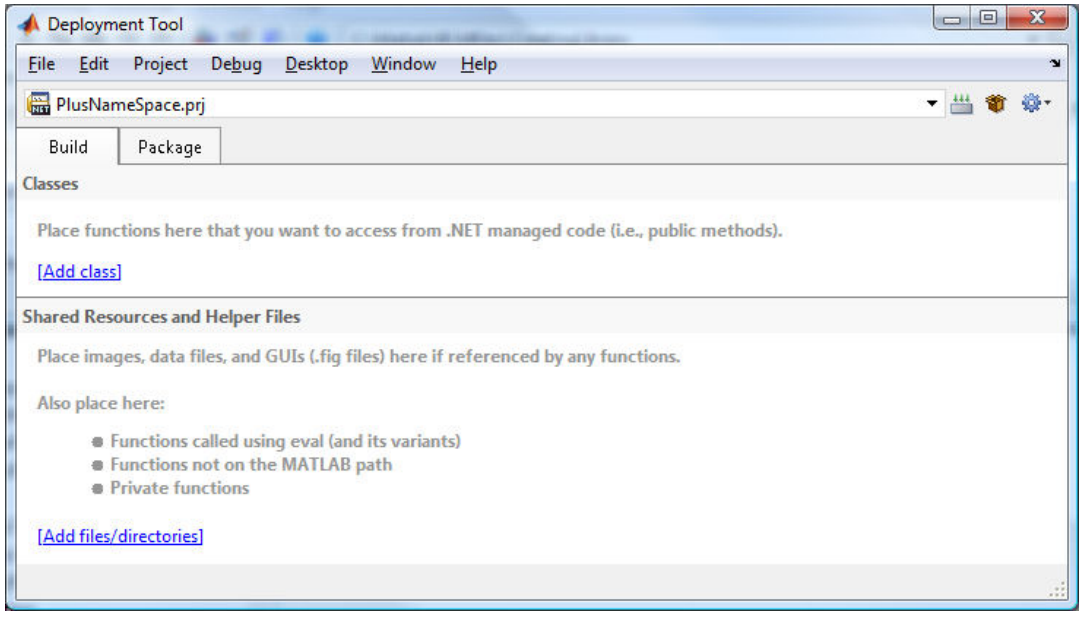


Figure 2.4: Dialog of adding classes

5. In the Fig. 2.4, click on ***Add class***, then add class name `PlusClass` (see Fig. 2.5).

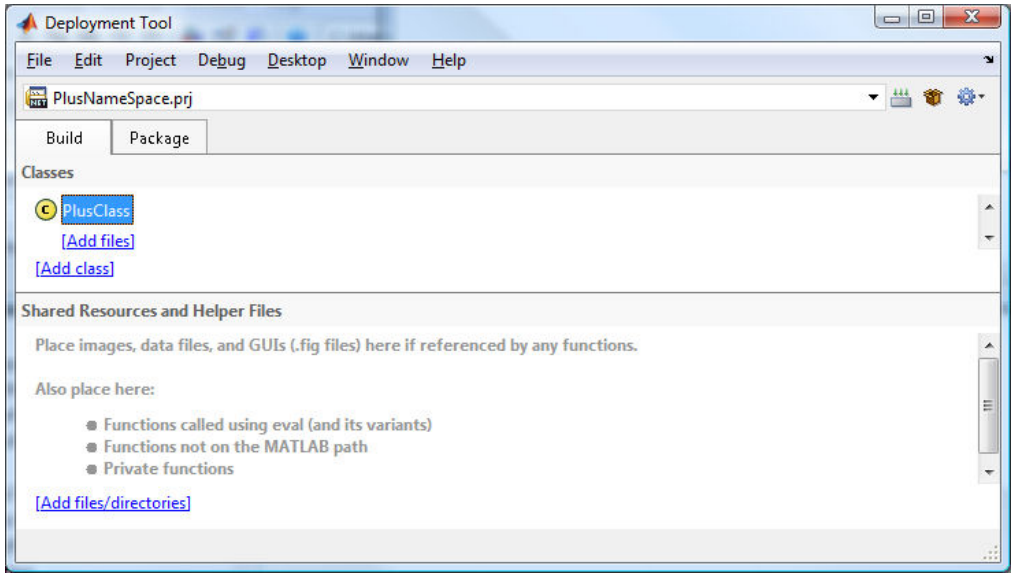


Figure 2.5: Adding a class name

6. In the Fig. 2.5, click on ***Add files***, then choose the file `myplus.m` above (see Fig. 2.6).

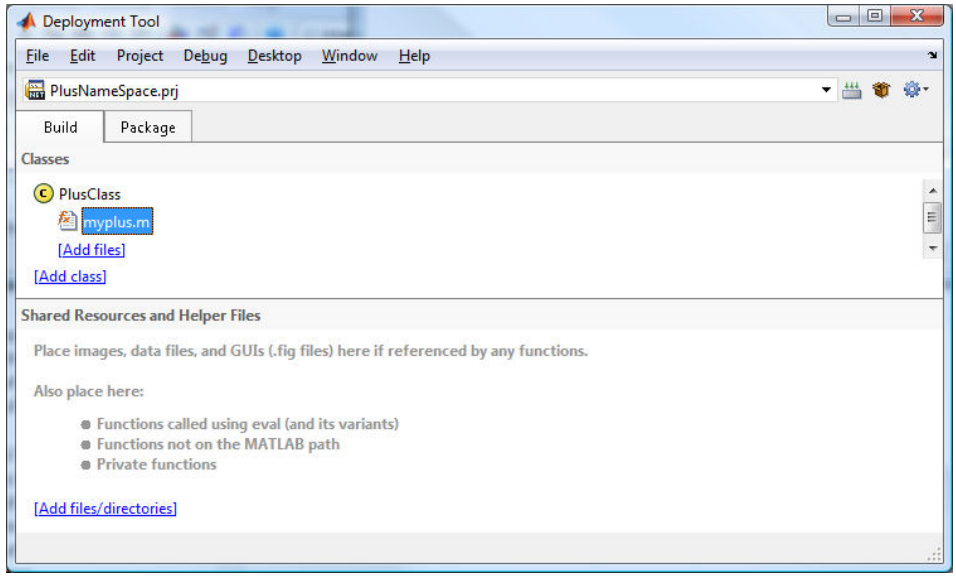


Figure 2.6: Adding a M-file

7. In the Fig. 2.6, click on ***Package*** tab to see the name of a dll file that will be generated.

Click on the Build icon to generate the dll file (see Fig 2.7).

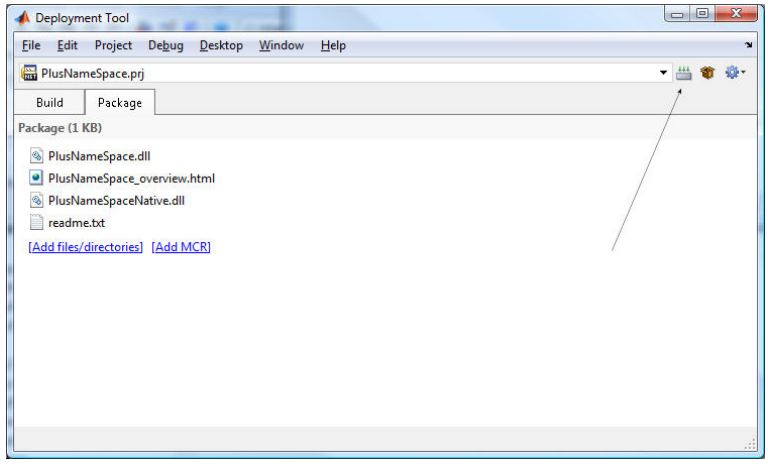


Figure 2.7: The dll file name

8. During process of generating a dll file, the Deployment project shows a dialog to indicate the process (see Fig. 2.8). After finished generated the dll file, there is a dialog to indicate it (see Fig 2.9). Look at the directory that contains the file `myplus.m`, you will see the dll file ***PlusNameSpace.dll*** (in the folder `PlusNameSpace`) (see Fig. 2.10).

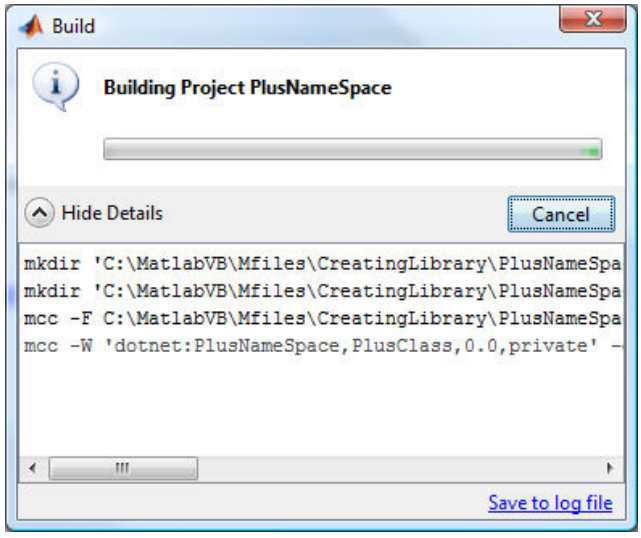


Figure 2.8: Dialog shows process of building the Deployment project

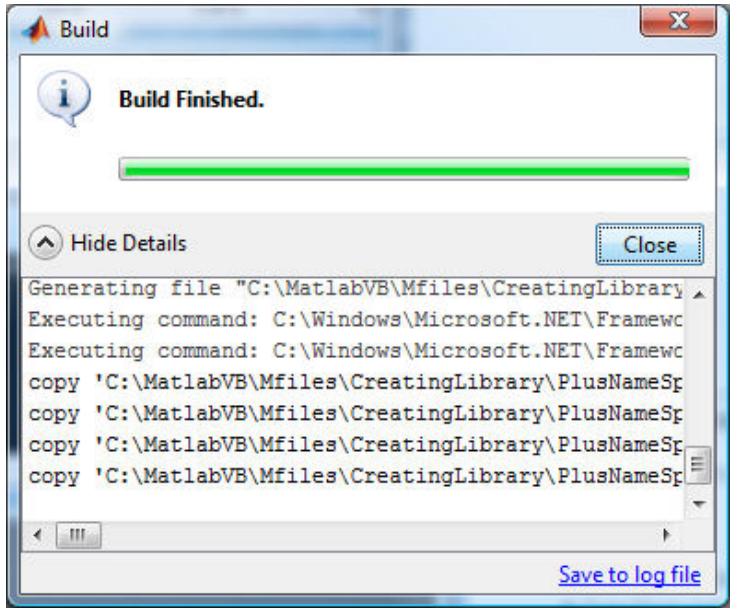


Figure 2.9: The dll file name

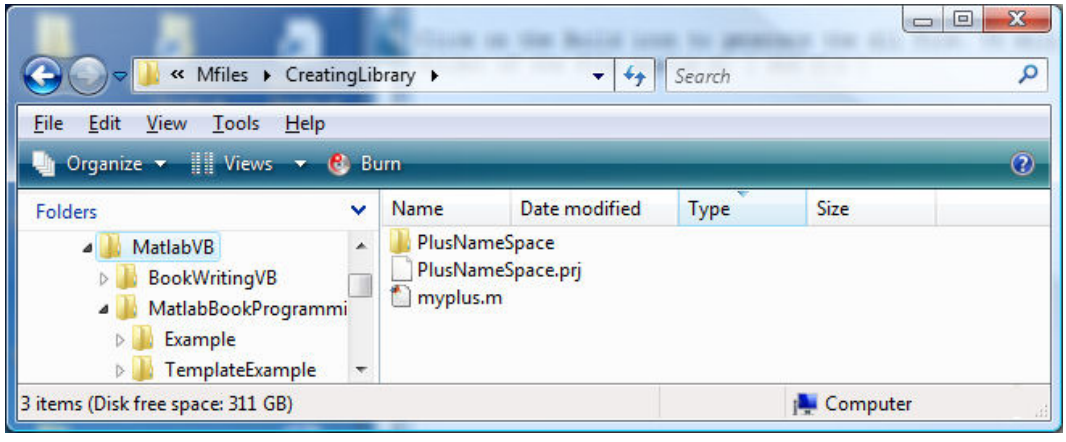


Figure 2.10: The directory **PlusNameSpace/distrib** contains the dll file

2.3 Using a Class Generated from a MATLAB M-File

This section shows steps how to use a class created from a MATLAB M-file.

1. Create a normal Console project in Visual Basic application (see Fig. 2.11).

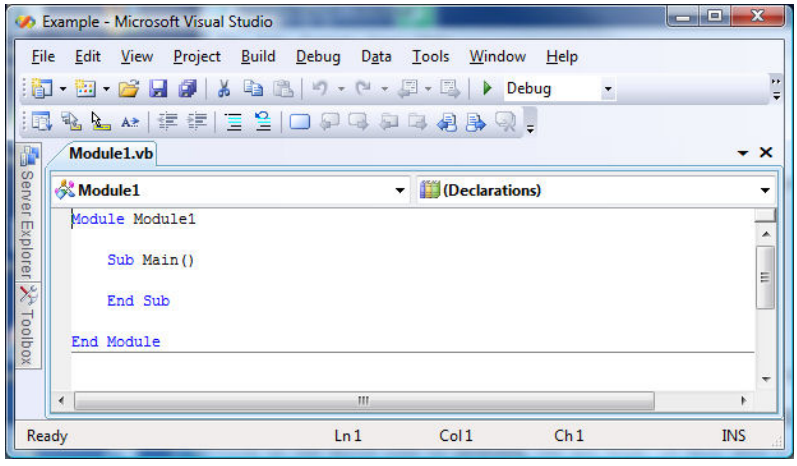


Figure 2.11: A normal Console project in Visual Basic

2. On ***Menu***, click ***Project***, ***Add Reference***. It will pop up a dialog ***Add Reference*** (see Fig. 2.12).

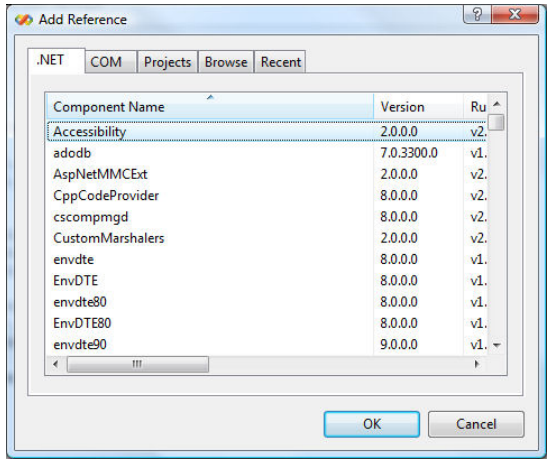


Figure 2.12: The Add Reference dialog

3. On the tab ***NET***, click on ***MathWorks, .NET MWArray API*** (see Fig. 2.13) then click OK to add this reference.

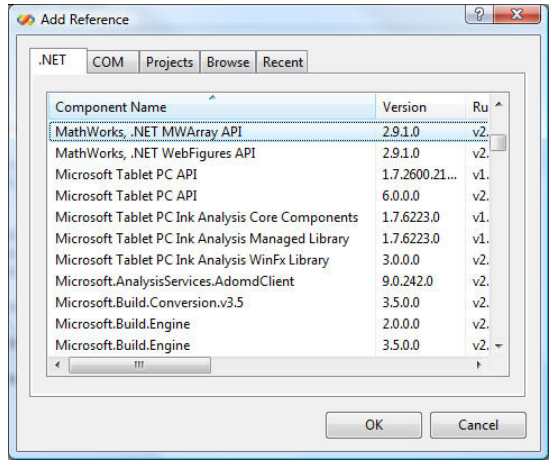


Figure 2.13: Adding the reference MATLAB MWArray API

4. On Menu, click again ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the generated folder above ***distrib*** to choose the file `PlusNameSpace.dll`, and then click OK (see Fig. 2.14).

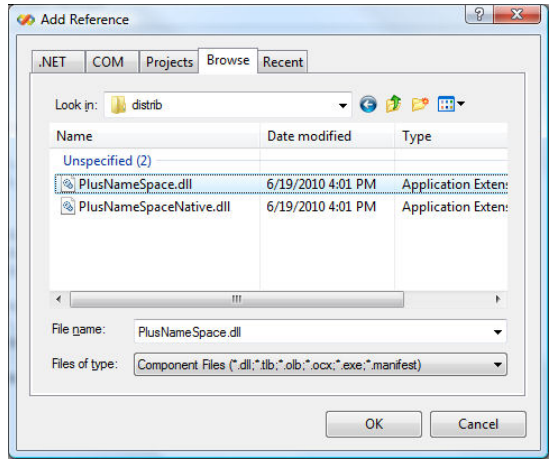


Figure 2.14: Adding the dll file

5. Open the file of VB code then add the following code to use the function `myplus.m` M-file

in the VB project to solve a simple addition problem.

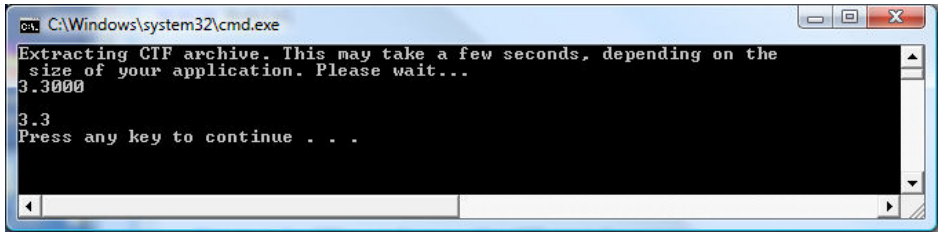


Figure 2.15: Calculation result of using the generated dll

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports PlusNameSpace

Module Module1

  Sub Main()

    Dim objVB As PlusClass = New PlusClass()

    Dim a As Double = 1.1
    Dim b As Double = 2.2
    Dim c As Double

    Dim mw_a As MWNumericArray = New MWNumericArray(a)
    Dim mw_b As MWNumericArray = New MWNumericArray(b)
    Dim mw_c As MWNumericArray = Nothing

    mw_c = objVB.myplus(mw_a, mw_b)

    c = mw_c.ToScalarDouble()

    Console.WriteLine(mw_c)
```

```
    mw_a.Dispose()
    mw_b.Dispose()
    mw_c.Dispose()

    Console.WriteLine()
    Console.WriteLine(c)

  End Sub

End Module
```

end code

In the project, click **Build**, then click **Build Solution** and we should have no error. Click **Debug**, then click **Start Without Debugging** we then have a result (see Fig. 2.15) without error.

To add more M-files in generating a class, we just regularly add as we did for the file `myplus.m` above. In the following chapters, we will create a class from multiple MATLAB M-files.

Chapter 3

Transfer of Values Between Visual Basic .NET Double and MATLAB MWNumericArray

In the version of MATLAB 7.9 (2009b) and MATLAB Compiler 4.11, MATLAB has utility classes that allow using MATLAB functions in a VB application easier. In order to use MATLAB functions in VB, we need to put inputs (scalar, vector, or matrix, etc.) to these functions. This chapter will show a way to transfer data between ***Double*** in VB and ***MWNumericArray*** in MATLAB. To begin with, create a simple project as show in Chapter 2 then you will use this project as an example to show how to transfer values between VB ***Double*** and MATLAB ***MWNumericArray***.

3.1 Transfer of real and complex scalar values between VB Double and MATLAB MWNumericArray

The following segment code shows the transfer of scalar values between VB ***Double*** and MATLAB ***MWNumericArray***.

Listing code

```
Public Sub TransferScalarValues()
```

```
'1a. transfer real value from double to MWNumericArray
Dim db_a As Double = 1.1
Dim mw_a As MWNumericArray = New MWNumericArray(db_a)

Dim db_b As Double = 2.2
Dim mw_b As MWNumericArray = New MWNumericArray(db_b)

Console.WriteLine("1a. double to MWNumericArray: mw_a = {0}", mw_a)
Console.WriteLine("    double to MWNumericArray: mw_b = {0}", mw_b)
Console.WriteLine()

'1b. transfer real value from MWNumericArray to double
Dim objVB As PlusClass = New PlusClass()
Dim mw_c As MWNumericArray = Nothing
mw_c = objVB.myplus(mw_a, mw_b)

Dim db_c As Double
db_c = mw_c.ToScalarDouble()
Console.WriteLine("1b. MWNumericArray to double: db_c = {0}", db_c)
Console.WriteLine()

'1c. transfer complex value from double to MWNumericArray
Dim dbReal_a As Double = 1.1
Dim dbImag_a As Double = 2.2
Dim mwComplex_a As MWNumericArray = New MWNumericArray(dbReal_a, dbImag_a)

Dim dbReal_b As Double = 3.3
Dim dbImag_b As Double = 4.4
Dim mwComplex_b As MWNumericArray = New MWNumericArray(dbReal_b, dbImag_b)

Console.WriteLine("1c. double to MWNumericArray Complex: mwComplex_a = {0}", mwComplex_a)
Console.WriteLine("    double to MWNumericArray Complex: mwComplex_b = {0}", mwComplex_b)
Console.WriteLine()

'1d. transfer complex value from MWNumericArray to double
```

```
  Dim mwComplex_c As MWNumericArray = Nothing
  mwComplex_c = objVB.myplus(mwComplex_a, mwComplex_b)


  Dim dbReal_c(1) As Double
  Dim dbImag_c(1) As Double

  dbReal_c = mwComplex_c.ToVector(MWArrayComponent.Real)
  ' In converting imaginary part to double,
  ' we'll get error if MWNumericArray mwComplex_c doesn't contain an imaginary part.

  If mwComplex_c.IsComplex Then
    dbImag_c = mwComplex_c.ToVector(MWArrayComponent.Imaginary)
  Else
    'dbImag_c will be assigned value of zero
  End If

  Console.WriteLine("1d. The real value dbReal_c = {0}: ", dbReal_c(0))
  Console.WriteLine("    The imag value dbImag_c = {0}: ", dbImag_c(0))
  Console.WriteLine()

  mw_a.Dispose()
  mw_b.Dispose()
  mw_c.Dispose()

  mwComplex_a.Dispose()
  mwComplex_b.Dispose()
  mwComplex_c.Dispose()

End Sub
```

end code

3.2 Transfer of real and complex vector values between VB Double and MATLAB MWNumericArray

The following segment code shows the transfer of vector values between VB ***Double*** and MATLAB ***MWNumericArray***.

Listing code

```
Public Sub TransferVectorValues()

  '2a. transfer real value of a vector from double to MWNumericArray
  Dim db_a() As Double = New Double() {1.1, 2.2, 3.3}
  Dim db_b() As Double = New Double() {4.4, 5.5, 6.6}

  Dim mw_a As MWNumericArray = New MWNumericArray(db_a)
  Dim mw_b As MWNumericArray = New MWNumericArray(db_b)

  Console.WriteLine("2a. double to MWNumericArray in a real vector mw_a: {0}", mw_a)
  Console.WriteLine("    double to MWNumericArray in a real vector mw_b: {0}", mw_b)
  Console.WriteLine()

  '2b. transfer real value of a vector from MWNumericArray to double
  Dim objVB As PlusClass = New PlusClass()
  Dim mw_c As MWNumericArray = Nothing
  mw_c = objVB.myplus(mw_a, mw_b)

  Dim db_c() As Double = mw_c.ToVector(MWArrayComponent.Real)

  Console.WriteLine("2b. MWNumericArray to double in a real vector:")
  PrintValues(db_c)
  Console.WriteLine()

  '2c. transfer complex value of a vector from double to MWNumericArray
  Dim dbReal_a() As Double = {1.1, 2.2, 3.3}
  Dim dbImag_a() As Double = {11.1, 22.2, 33.3}
  Dim mwComplex_a As MWNumericArray = New MWNumericArray(dbReal_a, dbImag_a)
```

```
Dim dbReal_b() As Double = {4.4, 5.5, 6.6}
Dim dbImag_b() As Double = {44.4, 55.5, 66.6}
Dim mwComplex_b As MWNumericArray = New MWNumericArray(dbReal_b, dbImag_b)

Console.WriteLine("2c. double to MWNumericArray in a complex vector:")
Console.WriteLine(ControlChars.Tab + "mwComplex_a = {0}", mwComplex_a)
Console.WriteLine(ControlChars.Tab + "mwComplex_b = {0}", mwComplex_b)
Console.WriteLine()

'2d. transfer complex value of a vector from MWNumericArray to double
Dim mwComplex_c As MWNumericArray = Nothing
mwComplex_c = objVB.myplus(mwComplex_a, mwComplex_b)

Dim vectorSize As Integer = mwComplex_c.ToArray(MWArrayComponent.Real).GetUpperBound(0) - _
                            mwComplex_c.ToArray(MWArrayComponent.Real).GetLowerBound(0) + 1

Dim dbReal_c(vectorSize) As Double
Dim dbImag_c(vectorSize) As Double

dbReal_c = mwComplex_c.ToVector(MWArrayComponent.Real)

Console.WriteLine("2d. MWNumericArray to double in a complex vector.")
Console.WriteLine(ControlChars.Tab + "mwComplex_c = {0}", mwComplex_c)

Console.WriteLine(ControlChars.Tab + "The real value dbReal_c :")
PrintValues(dbReal_c)

If mwComplex_c.IsComplex Then
  dbImag_c = mwComplex_c.ToVector(MWArrayComponent.Imaginary)
Else
  'nothing, dbImag_a2 will be assigned value of zero
End If

Console.WriteLine(ControlChars.Tab + "The imag value dbImag_c :")
PrintValues(dbImag_c)
```

```
  mw_a.Dispose()
  mw_b.Dispose()
  mw_c.Dispose()

  mwComplex_a.Dispose()
  mwComplex_b.Dispose()
  mwComplex_a.Dispose()

End Sub
```

end code

3.3 Transfer of real and complex matrix values between VB Double and MATLAB MWNumericArray

The following segment code shows the transfer of matrix values between VB ***Double*** and MATLAB ***MWNumericArray***.

Listing code

```
Public Sub TransferMatrixValues()

  '3a. transfer real value of a matrix from double to MWNumericArray
  Dim db_A(,) As Double = New Double(,) {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}, {7.7, 8.8, 9.9}}
  Dim db_B(,) As Double = New Double(,) {{101.1, 102.2, 103.3}, {104.4, 105.5, 106.6}, _
                                         {107.7, 108.8, 109.9}}

  Dim mw_A As MWNumericArray = New MWNumericArray(db_A)
  Dim mw_B As MWNumericArray = New MWNumericArray(db_B)

  Console.WriteLine("3a. double to MWNumericArray in a real matrix mw_A:")
  Console.WriteLine("{0}", mw_A)

  Console.WriteLine("    double to MWNumericArray in a real matrix mw_B:")
  Console.WriteLine("{0}", mw_B)
  Console.WriteLine()
```

```
'3b. transfer real value of a matrix from MWNumericArray to double
Dim objVB As PlusClass = New PlusClass()
Dim mw_C As MWNumericArray = Nothing
mw_C = objVB.myplus(mw_A, mw_B)

Dim db_C(,) As Double = mw_C.ToArray(MWArrayComponent.Real)
Console.WriteLine("3b. MWNumericArray to double in a real matrix mw_C:")
PrintValues(db_C)
Console.WriteLine()

'3c. transfer complex value of a matrix from double to MWNumericArray
Dim dbReal_A(,) As Double = New Double(,) {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}, _
                                           {7.7, 8.8, 9.9}}
Dim dbImag_A(,) As Double = New Double(,) {{11, 12, 13}, {14, 15, 16}, {17, 18, 19}}

Dim dbReal_B(,) As Double = New Double(,) {{10.1, 20.2, 30.3}, {40.4, 50.5, 60.6}, _
                                           {70.7, 80.8, 90.9}}
Dim dbImag_B(,) As Double = New Double(,) {{101, 102, 103}, {104, 105, 106}, _
                                           {107, 108, 109}}

Dim mwComplex_A As MWNumericArray = New MWNumericArray(dbReal_A, dbImag_A)
Dim mwComplex_B As MWNumericArray = New MWNumericArray(dbReal_B, dbImag_B)

Console.WriteLine("3c. double to MWNumericArray in a complex matrix mwComplex_A")
Console.WriteLine("{0}", mwComplex_A)
Console.WriteLine()

'3d. transfer complex value of a matrix from MWNumericArray to double
Dim mwComplex_C As MWNumericArray = Nothing
mwComplex_C = objVB.myplus(mwComplex_A, mwComplex_B)

Dim row As Integer = mwComplex_A.ToArray(MWArrayComponent.Real).GetUpperBound(0) - _
                     mwComplex_A.ToArray(MWArrayComponent.Real).GetLowerBound(0) + 1

Dim col As Integer = mwComplex_A.ToArray(MWArrayComponent.Real).GetUpperBound(1) - _
```

```
                    mwComplex_A.ToArray(MWArrayComponent.Real).GetLowerBound(1) + 1

  Dim dbReal_C(row, col) As Double
  Dim dbImag_C(row, col) As Double
  dbReal_C = mwComplex_C.ToArray(MWArrayComponent.Real)

  Console.WriteLine("3d. MWNumericArray to double in a complex matrix")
  Console.WriteLine(ControlChars.Tab + "mwComplex_C = {0}", mwComplex_C)

  Console.WriteLine(ControlChars.Tab + "The real value of the matrix dbReal_C :")
  PrintValues(dbReal_C)



  If mwComplex_C.IsComplex Then
    dbImag_C = mwComplex_C.ToArray(MWArrayComponent.Imaginary)
  Else
    'nothing, dbImag_C will be assigned value of zero

  End If

  Console.WriteLine(ControlChars.Tab + "The imag value of the matrix dbImag_C :")
  PrintValues(dbImag_C)

  mw_A.Dispose()
  mw_B.Dispose()
  mw_C.Dispose()

  mwComplex_A.Dispose()
  mwComplex_B.Dispose()
  mwComplex_C.Dispose()

End Sub
```

end code

The following is the code of the functions *PrintValues(..)* that used in the above subroutines.

Listing code

```
Public Sub PrintValues(ByVal myArr As Array)
  Dim myEnumerator As System.Collections.IEnumerator = _
     myArr.GetEnumerator()
  Dim i As Integer = 0
  Dim cols As Integer = myArr.GetLength(myArr.Rank - 1)

  'for row vector or column vector
  If myArr.Rank = 1 Then

    While myEnumerator.MoveNext()
      Console.WriteLine(ControlChars.Tab + "{0}", myEnumerator.Current)
    End While

  Else
    'for other
    While myEnumerator.MoveNext()
      If i < cols Then
        i += 1
      Else
        Console.WriteLine()
        i = 1
      End If
      Console.Write(ControlChars.Tab + "{0}", myEnumerator.Current)
    End While
  End If

  Console.WriteLine()
End Sub
```

end code

Chapter 4

Matrix Computations

In this chapter we will generate a class ***MatrixComputations*** from common M-files working on matrix computation problems. The generated functions of this class will be used in a VB .NET 2008 project to solve matrix computation problems.

We will write the M-files as shown below to generate the class ***MatrixComputations***,

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

mydet.m

```
function y = mydet(a)

y = det(a) ;
```

myinv.m

```
function y = myinv(a)

y = inv(a) ;
```

myminus.m

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

mymtimes.m

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

y = a*b ;
```

myplus.m

```
function y = (a, b)

y = a + b ;
```

mytranspose.m

```
function y = mytranspose( x )

y = x' ;
```

The following procedure to create the class ***MatrixComputations*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `MatrixComputationsNameSpace.dll` that contains the class ***MatrixComputations*** (see Fig.4.1 and Fig.4.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `MatrixComputationsNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `MatrixComputationsNameSpace.dll`, and then click OK (see Fig. 4.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

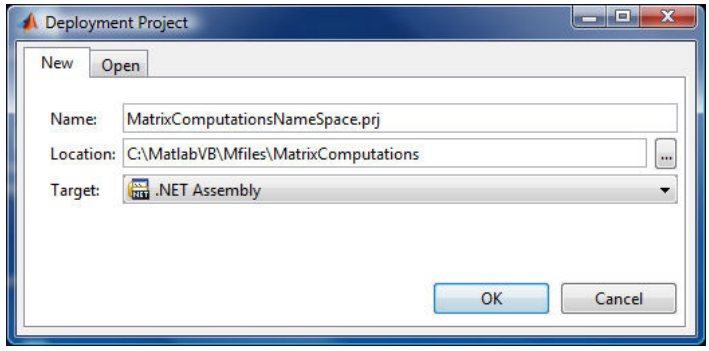


Figure 4.1: Deployment project for ***MatrixComputations***

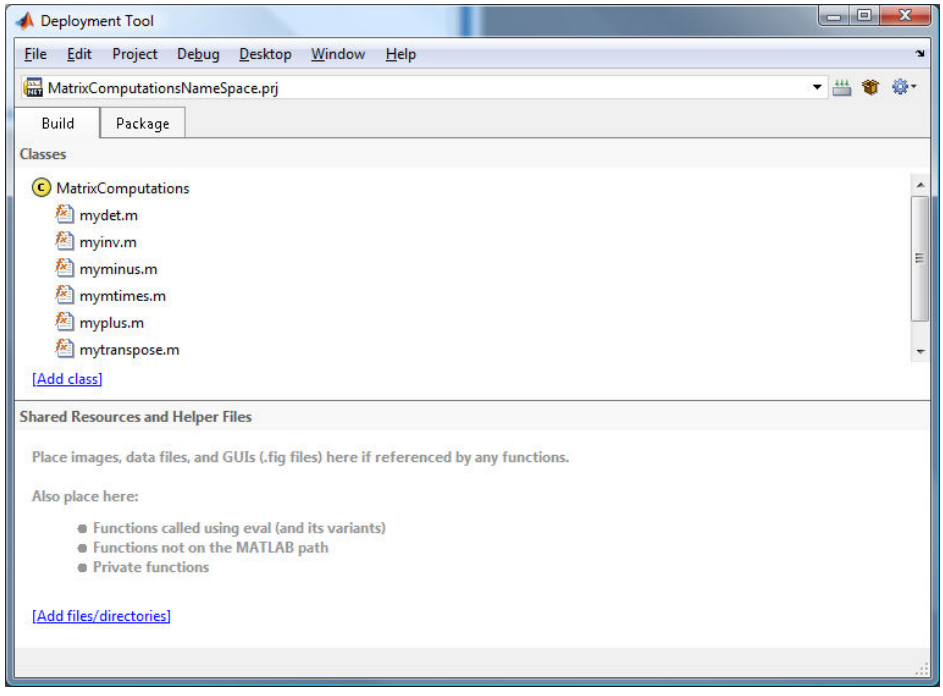


Figure 4.2: Adding M-files in deployment project for ***MatrixComputations***

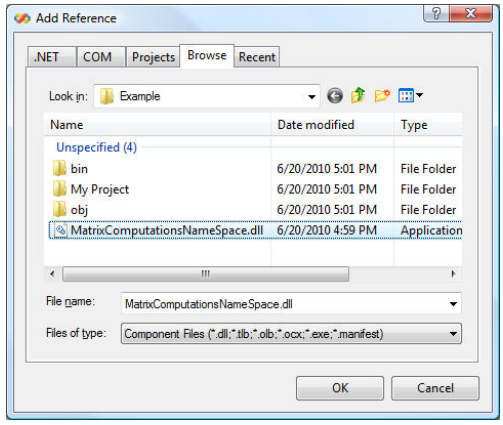


Figure 4.3: Add the reference `MatrixComputationsNameSpace.dll` in the VB project

The following sections show how to use the functions in the class ***MatrixComputations*** to solve common computation problems. The full code is at the end of this chapter.

4.1 Matrix Addition

In this section, we will use a MATLAB function in the class ***MatrixComputations*** to find a simple addition matrix.

Problem 1

input . Matrix $\mathbf{A}$ and $\mathbf{B}$

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

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

The following subroutine uses a function in the class ***MatrixComputations*** to solve Problem 1 .

Listing code

```
Public Sub AddMatrix()

  Dim A(,) As Double = New Double(,) {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}, {7.7, 8.8, 9.9}}
  Dim B(,) As Double = New Double(,) {{11, 12, 13}, {14, 15, 16}, {17, 18, 19}}

  ' declare variables
  Dim mw_A As MWNumericArray = New MWNumericArray(A)
  Dim mw_B As MWNumericArray = New MWNumericArray(B)
  Dim mw_C As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
         New MatrixComputationsNameSpace.MatrixComputations()

  mw_C = obj.myplus(mw_A, mw_B)

  ' convert back to double
  Dim C(,) As Double = mw_C.ToArray(MWArrayComponent.Real)

  'print out
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", mw_C)
  Console.WriteLine()

  ' or
  PrintValues(C)

  ' free memory
  mw_A.Dispose()
  mw_B.Dispose()
  mw_C.Dispose()

End Sub
```

end code

4.2 Matrix Subtraction

In this section, we will use a MATLAB function in the class ***MatrixComputations*** to find a simple subtraction matrix.

Problem 2

input . Matrix **A** and **B**

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

output . Finding the subtraction matrix $\mathbf{C} = \mathbf{A} - \mathbf{B}$

The following subroutine uses a function in the class ***MatrixComputations*** to solve Problem 2 .

Listing code

```
Public Sub SubtractMatrix()

  Dim A(,) As Double = New Double(,) {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}, {7.7, 8.8, 9.9}}
  Dim B(,) As Double = New Double(,) {{11, 12, 13}, {14, 15, 16}, {17, 18, 19}}

  ' declare variables
  Dim mw_A As MWNumericArray = New MWNumericArray(A)
  Dim mw_B As MWNumericArray = New MWNumericArray(B)
  Dim mw_C As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
         New MatrixComputationsNameSpace.MatrixComputations()

  mw_C = obj.myminus(mw_A, mw_B)
```

```
    ' convert back to double  '
    Dim C(,) As Double = mw_C.ToArray(MWArrayComponent.Real)

    ' print out
    Console.WriteLine(ControlChars.Tab + "result=")
    Console.WriteLine("{0}", mw_C)
    Console.WriteLine()

    ' or
    PrintValues(C)

    ' free memory
    mw_A.Dispose()
    mw_B.Dispose()
    mw_C.Dispose()

End Sub
```

end code

4.3 Matrix Multiplication

In this section, we will use a MATLAB function in the class ***MatrixComputations*** to find a multiplication matrix.

Problem 3

input . Matrix **A** and **B**

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

output . Finding the product matrix $\mathbf{C} = \mathbf{A} * \mathbf{B}$

The following subroutine uses a function in the class ***MatrixComputations*** to solve Problem 3 .

Listing code

```
Public Sub MultipleMatrix()

  Dim A(,) As Double = New Double(,) {{1.1, 2.2, 3.3, 4.4}, _
                              {5.5, 6.6, 7.7, 8.8}, _
                              {9.9, 10.1, 11.11, 12.12}}

  Dim B(,) As Double = New Double(,) {{10, 11}, {12, 13}, {14, 15}, {16, 17}}

  ' declare variables
  Dim mw_A As MWNumericArray = New MWNumericArray(A)
  Dim mw_B As MWNumericArray = New MWNumericArray(B)
  Dim mw_C As MWNumericArray = Nothing



  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
         New MatrixComputationsNameSpace.MatrixComputations()

  mw_C = obj.mymtimes(mw_A, mw_B)

  ' convert back to double
  Dim C(,) As Double = mw_C.ToArray(MWArrayComponent.Real)

  ' print out
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", mw_C)
  Console.WriteLine()

  ' or
  PrintValues(C)

  ' free memory
  mw_A.Dispose()
  mw_B.Dispose()
```

```
  mw_C.Dispose()

End Sub
```

end code

4.4 Matrix Determinant

In this section, we will use a MATLAB function in the class ***MatrixComputations*** to find determinant of a matrix.

Problem 4

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**

The following subroutine uses a function in the class ***MatrixComputations*** to solve Problem 4.

Listing code

```
Public Sub DeterminantMatrix()

  Dim A(,) As Double = New Double(,) {{1.1, 2.2, 3.3}, {7.7, 4.4, 9.9}, {4.4, 5.5, 8.8}}

  ' declare variables */
  Dim mw_A As MWNumericArray = New MWNumericArray(A)
  Dim mw_detA As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
         New MatrixComputationsNameSpace.MatrixComputations()

  mw_detA = obj.mydet(mw_A)
```

```
  ' convert back to double
  Dim detA As Double = mw_detA.ToScalarDouble()

  ' print out
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", mw_detA)
  Console.WriteLine()

  ' or
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", detA)
  Console.WriteLine()

  ' free memory
  mw_A.Dispose()
  mw_detA.Dispose()

End Sub
```

end code

4.5 Inverse Matrix

In this section, we will use a MATLAB function in the class ***MatrixComputations*** to find inverse of a matrix.

Problem 5

input . Matrix **A**

$$\mathbf{A} = \begin{bmatrix} -1 & 1 & 2 \\ 3 & -1 & 1 \\ -1 & 3 & 4 \end{bmatrix}$$

output . Finding the inverse of this matrix **A**

The following code describes how to use the function `myinv(..)` in the generated class ***MatrixComputations*** to find a matrix inversion.

The following subroutine uses a function in the class ***MatrixComputations*** to solve Problem 5 .

Listing code

```
Public Sub InverseMatrix()

  Dim A(,) As Double = New Double(,) {{-1, 1, 2}, {3, -1, 1}, {-1, 3, 4}}

  ' declare variables
  Dim mw_A As MWNumericArray = New MWNumericArray(A)
  Dim mw_inverseA As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
         New MatrixComputationsNameSpace.MatrixComputations()

  mw_inverseA = obj.myinv(mw_A)

  ' convert back to double
  Dim inverseA(,) As Double = mw_inverseA.ToArray(MWArrayComponent.Real)

  ' print out
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", mw_inverseA)
  Console.WriteLine()

  ' or
  PrintValues(inverseA)

  ' free memory
  mw_A.Dispose()
```

```
  mw_inverseA.Dispose()

End Sub
```

end code

4.6 Transpose Matrix

In this section, we will use a MATLAB function in the class ***MatrixComputations*** to find a transpose matrix.

Problem 6

input . Matrix **A**

$$\mathbf{A} = \begin{bmatrix} -1 & 1 & 2 \\ 3 & -1 & 1 \\ -1 & 3 & 4 \end{bmatrix}$$

output . Finding the transpose of this matrix **A**

The following subroutine uses a function in the class ***MatrixComputations*** to solve Problem 6.

Listing code

```
Public Sub TransposeMatrix()

  Dim A(,) As Double = New Double(,) {{-1, 1, 2}, {3, -1, 1}, {-1, 3, 4}}

  ' declare variables
  Dim mw_A As MWNumericArray = New MWNumericArray(A)
  Dim mw_transposeA As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
         New MatrixComputationsNameSpace.MatrixComputations()

  mw_transposeA = obj.mytranspose(mw_A)
```

```
  ' convert back to double
  Dim transposeA(,) As Double = mw_transposeA.ToArray(MWArrayComponent.Real)

  ' print out
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", mw_transposeA)
  Console.WriteLine()

  ' or
  PrintValues(transposeA)

  ' free memory
  mw_A.Dispose()
  mw_transposeA.Dispose()

End Sub
```

end code

The following is the full code for this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports MatrixComputationsNameSpace
Module Module1

  Sub Main()

    Dim objVB As Example = New Example()

    Console.WriteLine("Matrix Computations")
    Console.WriteLine("Matrix addition")

    objVB.addMatrix()
```

```
    Console.WriteLine("Matrix subtraction")
    objVB.subtractMatrix()

    Console.WriteLine("Matrix multiplication")
    objVB.multipleMatrix()

    Console.WriteLine("Matrix determinant")
    objVB.determinantMatrix()

    Console.WriteLine("Inverse matrix")
    objVB.inverseMatrix()

    Console.WriteLine("Transpose matrix")
    objVB.transposeMatrix()


  End Sub

  Public Class Example

    ' ****************************************
    Public Sub AddMatrix()

      Dim A(,) As Double = New Double(,) {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}, {7.7, 8.8, 9.9}}
      Dim B(,) As Double = New Double(,) {{11, 12, 13}, {14, 15, 16}, {17, 18, 19}}

      ' declare variables
      Dim mw_A As MWNumericArray = New MWNumericArray(A)
      Dim mw_B As MWNumericArray = New MWNumericArray(B)
      Dim mw_C As MWNumericArray = Nothing

      ' call an implemental function
      Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
             New MatrixComputationsNameSpace.MatrixComputations()
```

```
  mw_C = obj.myplus(mw_A, mw_B)

  ' convert back to double
  Dim C(,) As Double = mw_C.ToArray(MWArrayComponent.Real)

  'print out
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", mw_C)
  Console.WriteLine()

  ' or
  PrintValues(C)

  ' free memory
  mw_A.Dispose()
  mw_B.Dispose()
  mw_C.Dispose()

End Sub

' ****************************************
Public Sub SubtractMatrix()

  Dim A(,) As Double = New Double(,) {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}, {7.7, 8.8, 9.9}}
  Dim B(,) As Double = New Double(,) {{11, 12, 13}, {14, 15, 16}, {17, 18, 19}}

  ' declare variables
  Dim mw_A As MWNumericArray = New MWNumericArray(A)
  Dim mw_B As MWNumericArray = New MWNumericArray(B)
  Dim mw_C As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
         New MatrixComputationsNameSpace.MatrixComputations()

  mw_C = obj.myminus(mw_A, mw_B)
```

```
    ' convert back to double  '
    Dim C(,) As Double = mw_C.ToArray(MWArrayComponent.Real)

    ' print out
    Console.WriteLine(ControlChars.Tab + "result=")
    Console.WriteLine("{0}", mw_C)
    Console.WriteLine()

    ' or
    PrintValues(C)

    ' free memory
    mw_A.Dispose()
    mw_B.Dispose()
    mw_C.Dispose()

End Sub

' ****************************************
Public Sub MultipleMatrix()

    Dim A(,) As Double = New Double(,) {{1.1, 2.2, 3.3, 4.4}, _
                                {5.5, 6.6, 7.7, 8.8}, _
                                {9.9, 10.1, 11.11, 12.12}}

    Dim B(,) As Double = New Double(,) {{10, 11}, {12, 13}, {14, 15}, {16, 17}}

    ' declare variables
    Dim mw_A As MWNumericArray = New MWNumericArray(A)
    Dim mw_B As MWNumericArray = New MWNumericArray(B)
    Dim mw_C As MWNumericArray = Nothing


    ' call an implemental function
    Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
```

```
        New MatrixComputationsNameSpace.MatrixComputations()

  mw_C = obj.mymtimes(mw_A, mw_B)

  ' convert back to double
  Dim C(,) As Double = mw_C.ToArray(MWArrayComponent.Real)

  ' print out
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", mw_C)
  Console.WriteLine()

  ' or
  PrintValues(C)

  ' free memory
  mw_A.Dispose()
  mw_B.Dispose()
  mw_C.Dispose()

End Sub

' ****************************************
Public Sub DeterminantMatrix()

  Dim A(,) As Double = New Double(,) {{1.1, 2.2, 3.3}, {7.7, 4.4, 9.9}, {4.4, 5.5, 8.8}}

  ' declare variables */
  Dim mw_A As MWNumericArray = New MWNumericArray(A)
  Dim mw_detA As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
        New MatrixComputationsNameSpace.MatrixComputations()

  mw_detA = obj.mydet(mw_A)
```

```
    ' convert back to double
    Dim detA As Double = mw_detA.ToScalarDouble()

    ' print out
    Console.WriteLine(ControlChars.Tab + "result=")
    Console.WriteLine("{0}", mw_detA)
    Console.WriteLine()

    ' or
    Console.WriteLine(ControlChars.Tab + "result=")
    Console.WriteLine("{0}", detA)
    Console.WriteLine()

    ' free memory
    mw_A.Dispose()
    mw_detA.Dispose()

  End Sub

  ' ****************************************
  Public Sub InverseMatrix()

    Dim A(,) As Double = New Double(,) {{-1, 1, 2}, {3, -1, 1}, {-1, 3, 4}}

    ' declare variables
    Dim mw_A As MWNumericArray = New MWNumericArray(A)
    Dim mw_inverseA As MWNumericArray = Nothing

    ' call an implemental function
    Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
           New MatrixComputationsNameSpace.MatrixComputations()

    mw_inverseA = obj.myinv(mw_A)

    ' convert back to double
```

```
  Dim inverseA(,) As Double = mw_inverseA.ToArray(MWArrayComponent.Real)

  ' print out
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", mw_inverseA)
  Console.WriteLine()

  ' or
  PrintValues(inverseA)

  ' free memory
  mw_A.Dispose()
  mw_inverseA.Dispose()

End Sub

' ****************************************
Public Sub TransposeMatrix()

  Dim A(,) As Double = New Double(,) {{-1, 1, 2}, {3, -1, 1}, {-1, 3, 4}}

  ' declare variables
  Dim mw_A As MWNumericArray = New MWNumericArray(A)
  Dim mw_transposeA As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
         New MatrixComputationsNameSpace.MatrixComputations()

  mw_transposeA = obj.mytranspose(mw_A)

  ' convert back to double
  Dim transposeA(,) As Double = mw_transposeA.ToArray(MWArrayComponent.Real)

  ' print out
  Console.WriteLine(ControlChars.Tab + "result=")
```

```
    Console.WriteLine("{0}", mw_transposeA)
    Console.WriteLine()

    ' or
    PrintValues(transposeA)

    ' free memory
    mw_A.Dispose()
    mw_transposeA.Dispose()

  End Sub

  ' ********************************
  Public Sub PrintValues(ByVal myArr As Array)
    Dim myEnumerator As System.Collections.IEnumerator = _
       myArr.GetEnumerator()
    Dim i As Integer = 0
    Dim cols As Integer = myArr.GetLength(myArr.Rank - 1)

    'for row vector or column vector
    If myArr.Rank = 1 Then

      While myEnumerator.MoveNext()
        Console.WriteLine(ControlChars.Tab + "{0}", myEnumerator.Current)
      End While

    Else
      'for other
      While myEnumerator.MoveNext()
        If i < cols Then
          i += 1
        Else
          Console.WriteLine()
          i = 1
        End If
        Console.Write(ControlChars.Tab + "{0}", myEnumerator.Current)
```

```
        End While
      End If

      Console.WriteLine()
    End Sub


  End Class

End Module
```

end code

Chapter 5

Linear System Equations

In this chapter we will generate a class ***LinearSystem*** from common M-files working on problems of linear system equations. The generated functions of this class will be used in a VB .Net 2008 project to solve problems of linear system equations.

We will write the M-files as shown below to generate the class ***LinearSystem***.

`mydiag.m, myextractmatrix.m, myfull.m, mylu.m, mymldivide.m, mymrdivide.m, mysparse.m, myspdiags.m, and mytranspose.m`

mydiag.m

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

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

myextractmatrix.m

```
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
```

myfull.m

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

mylu.m

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

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

mymldivide.m

```
function x = mymldivide(A, b)
%solve equation Ax = b
x = A\b ;
```

mymrdivide.m

```
function x = mymrdivide(A, b)
%solve equation xA = b ==>
x = A/b ;
```

mysparse.m

```
function  S = mysparse(A)

S = sparse(A) ;
```

myspdiags.m

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

mytranspose.m

```
function y = ( x )

y = x' ;
```

The following procedure to create the class ***LinearSystem*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `LinearSystemNameSpace.dll` that contains the class ***LinearSystem*** (see Fig.5.1 and Fig.5.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `LinearSystemNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `LinearSystemNameSpace.dll`, and then click OK (see Fig. 5.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

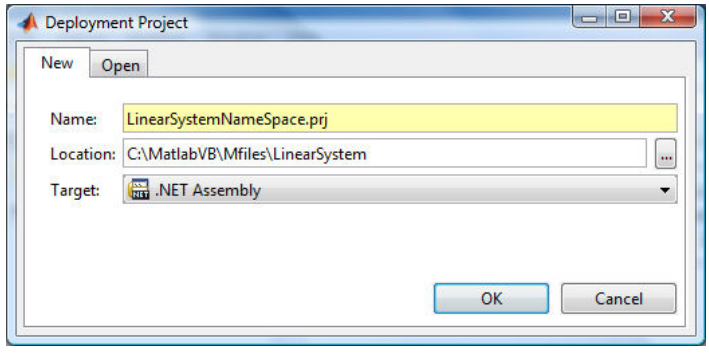


Figure 5.1: Deployment project for ***LinearSystem***

The following sections show how to use the functions in the class ***LinearSystem*** to solve the common linear system problems. The full code is at the end of this chapter.

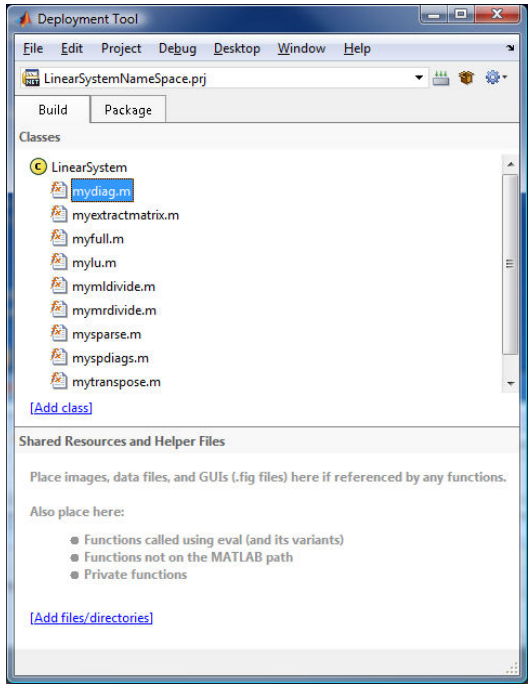


Figure 5.2: Adding M-files in deployment project for ***LinearSystem***

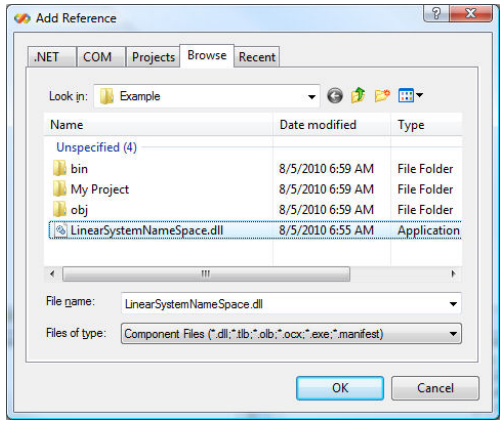


Figure 5.3: Add the reference `LinearSystemNameSpace.dll` in the VB project

5.1 Linear System Equations

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

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ a_{31}x_1 + a_{32}x_2 + \cdots + a_{3n}x_n &= b_3 \\ \cdots \qquad\qquad & \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{5.1}$$

In this section, we will use a MATLAB function in the class ***LinearSystem*** to solve linear system problems.

Problem 1

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

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

output . Finding the solution **x** of linear system equations, $\mathbf{Ax} = \mathbf{b}$

. Finding the lower **L** and upper **U** of the matrix **A** in the decompression method

The following subroutine uses the functions *LinearSystemEquations()* and *LU_ decompression()* in the class ***LinearSystem*** to solve Problem 1.

Listing code

```
Public Sub LinearSystemEquations()

  ' Solve general linear system equations Ax = b
  Dim A(,) As Double = New Double(,) {{1.1, 5.6 , 3.3}, _
```

```
                                        {4.4, 12.3, 6.6}, _
                                        {7.7, 8.8 , 9.9}}

  Dim vectorb() As Double = New Double() { 12.5, 32.2 , 45.6 }

  ' declare variables
  Dim mw_A        As MWNumericArray = New MWNumericArray(A)
  Dim mw_vectorb  As MWNumericArray = New MWNumericArray(vectorb)
  Dim mw_x        As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As LinearSystem = New LinearSystem()

  mw_vectorb  = obj.mytranspose(mw_vectorb)
  mw_x        = obj.mymldivide(mw_A, mw_vectorb)

  ' convert back to double
  Dim x() As Double = mw_x.ToVector(MWArrayComponent.Real)

  ' print out
  Console.WriteLine(ControlChars.Tab + "result=")
  Console.WriteLine("{0}", mw_x)

  ' or
  Console.WriteLine(ControlChars.Tab + "result=")
  PrintValues(x)

  ' free memory
  mw_A.Dispose()
  mw_vectorb.Dispose()
  mw_x.Dispose()

End Sub

' ****************************
Public Sub LU_decompression()
```

```
' Find lower and upper matrixes
Dim A(,) As Double = New Double (,) { {1.1,  5.6, 3.3}  , _
                                      {4.4, 12.3, 6.6}  , _
                                      {7.7,  8.8, 9.9}  }

' Declare variables
Dim mw_A As MWNumericArray = new MWNumericArray(A)

Dim mw_ArrayOut() As MWArray = Nothing

Dim mw_L As MWNumericArray = Nothing
Dim mw_U As MWNumericArray = Nothing
Dim mw_P As MWNumericArray = Nothing

' call an implemental function
' value of 3 is three of ouput in mylu.m
Dim obj As LinearSystem = New LinearSystem()
mw_ArrayOut = obj.mylu(3, mw_A)

'LinearSystem objLinear = new LinearSystem() ;
mw_L = mw_ArrayOut(0)
mw_U = mw_ArrayOut(1)
mw_P = mw_ArrayOut(2)

' convert back to double
Dim L(,) As Double = mw_L.ToArray(MWArrayComponent.Real)
Dim U(,) As Double = mw_U.ToArray(MWArrayComponent.Real)
Dim P(,) As Double = mw_P.ToArray(MWArrayComponent.Real)

' print out
Console.WriteLine("The lower matrix L = ")
Console.WriteLine("{0}", mw_L)

Console.WriteLine("The upper matrix U = ")
Console.WriteLine("{0}", mw_U)
```

```
  Console.WriteLine("P = ")
  Console.WriteLine("{0}", mw_P)

  ' or
  Console.WriteLine("Printing matrix L")
  PrintValues(L)

  Console.WriteLine("Printing matrix U")
  PrintValues(U)

  Console.WriteLine("Printing matrix P")
  PrintValues(P)

  ' free memory
  mw_A.Dispose()
  MWNumericArray.DisposeArray(mw_ArrayOut)
  mw_L.Dispose()
  mw_U.Dispose()
  mw_P.Dispose()

End Sub
```

end code

5.2 Sparse Linear System

The sparse linear system is a common system created to solve a particular 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.

In this section, we will use MATLAB functions to solve the following problems.

Problem 2

input . Sparse matrix **A** and vector **b**

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1.1 \\ 0 & 2.2 & 0 & 0 & 0 \\ 3.3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6.6 & 0 \\ 0 & 0 & 5.5 & 0 & 0 \end{bmatrix} \quad , \quad \mathbf{b} = \begin{bmatrix} 11.1 \\ 0 \\ 22.2 \\ 0 \\ 33.3 \end{bmatrix}$$

output . Finding the solution **x** of the sparse system equations $\mathbf{Ax} = \mathbf{b}$

The common steps to solve a sparse linear system using functions in the class ***LinearSytem*** are:

1. Establishing the spare matrix by using the function `mysparse(..)`.
2. Solving the sparse system by using the function `mymldivide(..)`.

The following is the code to solve Problem 2 by using the functions, `mysparse(..)` and `mymldivide(..)` in the class ***LinearSytem***.

Listing code

```
' ****************************
Public Sub 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
Dim A(,) As Double = { {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   } }

Dim vectorb() As Double =  {11.1,   0  ,    22.2,  0  ,    33.3}

' declare variables
Dim mw_A        As MWNumericArray = New MWNumericArray(A)
Dim mw_vectorb  As MWNumericArray = New MWNumericArray(vectorb)
Dim mw_x        As MWNumericArray = Nothing

' call an implemental function
Dim obj As LinearSystem = New LinearSystem()

mw_vectorb = obj.mytranspose(mw_vectorb)

mw_A       = obj.mysparse(mw_A)
mw_vectorb = obj.mysparse(mw_vectorb)

mw_x       = obj.mymldivide(mw_A, mw_vectorb)
mw_x       = obj.myfull(mw_x)

'convert back to double
Dim x() As Double = mw_x.ToVector(MWArrayComponent.Real)

' print out
Console.WriteLine("Solution x :")
Console.WriteLine("{0}", mw_x)

'or
Console.WriteLine("Printing vector x:")
PrintValues(x)
```

```
  ' free memory
  mw_A.Dispose()
  mw_vectorb.Dispose()
  mw_x.Dispose()

End Sub
```

end code

5.3 Tridiagonal System Equations

This section focuses on finding a solution of the tridiagonal linear system equations $\mathbf{Ax} = \mathbf{d}$:

$$\begin{bmatrix} a_1 & b_1 & 0 & & \cdots & 0 \\ c_2 & a_2 & b_2 & & \cdots & 0 \\ 0 & c_3 & a_3 & b_3 & \cdots & 0 \\ \cdots & 0 & \cdots & & \cdots & \cdots \\ 0 & \cdots & 0 & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \cdots & 0 & 0 & c_n & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \cdots \\ d_{n-1} \\ d_n \end{bmatrix} \tag{5.2}$$

Problem 3

input . Matrix **B** includes column vectors **c**, **a**, and **b**

. Vector right-hand side **d**

$$\mathbf{B} = \begin{bmatrix} c_1 & a_1 & b_1 \\ c_2 & a_2 & b_2 \\ c_3 & a_3 & b_3 \\ c_4 & a_4 & b_4 \\ c_5 & a_5 & b_5 \\ c_6 & a_6 & b_6 \end{bmatrix} = \begin{bmatrix} 1.1 & 4.1 & 2.1 \\ 1.2 & 4.2 & 2.2 \\ 1.3 & 4.3 & 2.3 \\ 1.4 & 4.4 & 2.4 \\ 1.5 & 4.5 & 2.5 \\ 1.6 & 4.6 & 2.6 \end{bmatrix} \quad , \quad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \end{bmatrix} = \begin{bmatrix} 1.2 \\ 4.5 \\ 5.6 \\ 12.4 \\ 7.8 \\ 6.8 \end{bmatrix} \tag{5.3}$$

output . Finding the solution **x** of tridiagonal system equations in the form of Eq. 5.2

The steps to solve Problem 3 are:

1. Establish a buffer matrix **bufferA** (in Eq. 5.4) from the given matrix **B** (in Eq. 5.3) by using the following function in the class ***LinearSytem***. The function `myspdiags(..)` creates an m-by-n sparse matrix **bufferA** by taking the columns of B and placing them along the diagonals shown in the following.

$$\mathbf{bufferA} = \begin{bmatrix} c_1 & a_1 & b_1 & 0 & & \cdots & 0 & 0 \\ 0 & c_2 & a_2 & b_2 & & \cdots & 0 & 0 \\ 0 & 0 & c_3 & a_3 & b_3 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & c_{n-1} & a_{n-1} & b_{n-1} & 0 \\ 0 & 0 & \cdots & 0 & 0 & c_n & a_n & b_n \end{bmatrix} \tag{5.4}$$

2. Obtain the matrix A as in Eq. 5.2 by extracting from the matrix **bufferA**

3. Use the functions in the class ***LinearSytem*** to solve the tridiagonal linear system equations.

The following is the code to solve Problem 3 by using the functions in the library ***LinearSytem***.

Listing code

```
Public Sub TridiagonalSystem()

  Dim B(5,2) As Double

  ' 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

Dim db_vectord(5) As Double
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

' The size of the matrix A
Dim row As Integer = 6
Dim col AS Integer = 6

Dim band As Integer = 3  ' tridiagnal, band width

' Establish the row size=m, and column size=n
' of the bufferA

Dim m As Integer = row
Dim n As Integer = col + (band - 1)

Dim d() As Double = { 0, 1, 2 } ' for band=3, start from 0
```

```
Dim rowB As Integer = row
Dim colB As Integer = band

' declare mxArray variables '
Dim mw_B As MWNumericArray       = New MWNumericArray(B)
Dim mw_bufferA As MWNumericArray = Nothing
Dim mw_A As MWNumericArray       = Nothing

Dim mw_d As MWNumericArray = New MWNumericArray(d)

Dim mw_vectord As MWNumericArray = New MWNumericArray(db_vectord)
Dim mw_x As MWNumericArray       = Nothing

' call an implemental function
Dim obj As LinearSystemNameSpace.LinearSystem =     _
       New LinearSystemNameSpace.LinearSystem()

' create a sparse matrix mw_bufferA from column-matrix B
mw_bufferA = obj.myspdiags(mw_B, mw_d, m, n)
mw_bufferA = obj.myfull(mw_bufferA)

' plot to see
Console.WriteLine("The buffer matrix A:")
Console.WriteLine(mw_bufferA)

 'extract the need-matrix from the buffter matrix,
 ' (start at 1 following MATLAB in m.file )
 '  from row 1 to row 6(=m) and from column 2(=start+1) to column 7(=n-1)
mw_A = obj.myextractmatrix(mw_bufferA, 1, row, 2, 7)

' plot to see
Console.WriteLine("The need-matrix A:")
Console.WriteLine(mw_A)

' solve the tridiagnal system equations
```

```
  mw_A = obj.mysparse(mw_A)

  mw_vectord = obj.mytranspose(mw_vectord)
  mw_vectord = obj.mysparse(mw_vectord)

  mw_x = obj.mymldivide(mw_A, mw_vectord)
  mw_x = obj.myfull(mw_x)

  ' convert back to double
  Dim x() As Double = mw_x.ToVector(MWArrayComponent.Real)

  'print out
  Console.WriteLine("Tridiagnal system solution:")
  Console.WriteLine(mw_x)

  ' or
  PrintValues(x)

  ' free memory
  mw_B.Dispose()
  mw_bufferA.Dispose()
  mw_A.Dispose()

  mw_d.Dispose()

  mw_vectord.Dispose()
  mw_x.Dispose()

End Sub
```

end code

5.4 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:

$$\begin{bmatrix} a_1 & b_1 & e_1 & 0 & & \cdots & 0 \\ c_2 & a_2 & b_2 & e_2 & 0 & \cdots & 0 \\ 0 & c_3 & a_3 & b_3 & e_3 & \cdots & 0 \\ \cdots & \cdots & \cdots & & & \cdots & \cdots \\ 0 & \cdots & 0 & c_{n-2} & a_{n-2} & b_{n-2} & e_{n-2} \\ 0 & \cdots & 0 & 0 & c_{n-1} & a_{n-1} & b_{n-1} \\ 0 & \cdots & 0 & 0 & 0 & c_n & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \cdots \\ d_{n-2} \\ d_{n-1} \\ d_n \end{bmatrix} \tag{5.5}$$

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**

$$\mathbf{B} = \begin{bmatrix} c_1 & a_1 & b_1 & e_1 \\ c_2 & a_2 & b_2 & e_2 \\ c_3 & a_3 & b_3 & e_3 \\ c_4 & a_4 & b_4 & e_4 \\ c_5 & a_5 & b_5 & e_5 \\ c_6 & a_6 & b_6 & e_6 \end{bmatrix} = \begin{bmatrix} 1.1 & 4.1 & 2.1 & 7.1 \\ 1.2 & 4.2 & 2.2 & 7.2 \\ 1.3 & 4.3 & 2.3 & 7.3 \\ 1.4 & 4.4 & 2.4 & 7.4 \\ 1.5 & 4.5 & 2.5 & 7.5 \\ 1.6 & 4.6 & 2.6 & 7.6 \end{bmatrix}, \quad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \end{bmatrix} = \begin{bmatrix} 1.2 \\ 4.5 \\ 5.6 \\ 12.4 \\ 7.8 \\ 6.8 \end{bmatrix} \tag{5.6}$$

output . Finding the solution **x** of the band system in the form of Eq. 5.5.

The steps to solve this problem are similar to the steps in the tridiagonal problem. These steps are:

1. Establish a buffer matrix **bufferA** (in Eq. 5.7) from given matrix **B** (in Eq. 5.6) by using

a function in the class ***LinearSytem***.

$$\mathbf{bufferA} = \begin{bmatrix} c_1 & a_1 & b_1 & e_1 & 0 & & \cdots & 0 & 0 & 0 \\ 0 & c_2 & a_2 & b_2 & e_2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & c_3 & a_3 & b_3 & e_3 & \cdots & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & & & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 0 & c_{n-2} & a_{n-2} & b_{n-2} & e_{n-2} & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & c_{n-1} & a_{n-1} & b_{n-1} & e_{n-1} & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & c_n & a_n & b_n & e_n \end{bmatrix} \tag{5.7}$$

2. Obtain the matrix A as in Eq. 5.5 by extracting from the matrix **bufferA**.

3. Use the functions in the class to solve the band system diagonal equations.

The following is the code to solve Problem 4 by using the functions in the library ***LinearSytem***.

Listing code

```
Public Sub BandMatrixSystem()


  Dim B(5,3) As Double


  ' 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

Dim db_vectord(5) As Double
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

' The size of the matrix A
Dim row As Double = 6
Dim col As Double = 6

Dim band As Integer = 4  ' band width

' Establish the row size=m, and column size=n
' of the bufferA

Dim m As Integer = row
Dim n As Integer = col + (band - 1)
```

```
Dim d() As Double = New Double() { 0, 1, 2, 3 } ' for band = 4 start from 0

Dim rowB As Double = row
Dim colB As Double = band

'declare mxArray variables
Dim mw_B As MWNumericArray       = New MWNumericArray(B)
Dim mw_bufferA As MWNumericArray = Nothing
Dim mw_A As MWNumericArray       = Nothing

Dim mw_d As MWNumericArray= new MWNumericArray(d)

Dim  mw_vectord As MWNumericArray= New MWNumericArray(db_vectord)
Dim  mw_x As MWNumericArray= Nothing

' call an implemental function
' create a sparse matrix, size mxn, from column-matrix B

Dim obj As LinearSystemNameSpace.LinearSystem =     _
       New LinearSystemNameSpace.LinearSystem()

mw_bufferA = obj.myspdiags(mw_B, mw_d, m, n)
mw_bufferA = obj.myfull(mw_bufferA)

' plot to see
Console.WriteLine("The buffer band-matrix A:")
Console.WriteLine(mw_bufferA)

' extract the need-matrix A from the buffter matrix,
'    from row 1 to row 6 and from column 2 to column 7 */

' extract the need-matrix from the buffter matrix,
'  (start at 1 following MATLAB in m.file )
'  from row 1 to row 6(=m) and from column 2(=start+1) to column 7(=n-1) */
mw_A = obj.myextractmatrix(mw_bufferA, 1, row, 2, 7)
```

```
  ' plot to see
  Console.WriteLine("The band-need-matrix A:")
  Console.WriteLine(mw_A)

  ' solve the system equations
  mw_A = obj.mysparse(mw_A)

  mw_vectord = obj.mytranspose(mw_vectord)
  mw_vectord = obj.mysparse(mw_vectord)

  mw_x = obj.mymldivide(mw_A, mw_vectord)
  mw_x = obj.myfull(mw_x)

  ' convert back to double
  Dim solution_x() As Double = mw_x.ToVector(MWArrayComponent.Real)

  ' print out
  ' convert back to double
  Console.WriteLine("Band matrix system solution:")
  Console.WriteLine(mw_x)

  ' or
  PrintValues(solution_x)

  ' free memory
  mw_B.Dispose()
  mw_bufferA.Dispose()
  mw_A.Dispose()

  mw_d.Dispose()

  mw_vectord.Dispose()
  mw_x.Dispose()

End Sub
```

end code

The following is the full code for this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports LinearSystemNameSpace

Module Module1

  Sub Main()
    Dim objVB As Example = New Example()

    Console.WriteLine(" Linear System Equations")

    objVB.LinearSystemEquations()

    Console.WriteLine("ControlChars.NewLine + LU_decompression")
    objVB.LU_decompression()

    Console.WriteLine(ControlChars.NewLine + "Spare System")
    objVB.SparseSystem()

    Console.WriteLine()
    objVB.TridiagonalSystem()

    Console.WriteLine()
    objVB.BandMatrixSystem()

  End Sub

  Public Class Example
    Public Sub LinearSystemEquations()
```

```
    ' Solve general linear system equations Ax = b
    Dim A(,) As Double = New Double(,) {{1.1, 5.6 , 3.3}, _
                                        {4.4, 12.3, 6.6}, _
                                        {7.7, 8.8 , 9.9}}

    Dim vectorb() As Double = New Double() { 12.5, 32.2 , 45.6 }

    ' declare variables
    Dim mw_A        As MWNumericArray = New MWNumericArray(A)
    Dim mw_vectorb  As MWNumericArray = New MWNumericArray(vectorb)
    Dim mw_x        As MWNumericArray = Nothing

    ' call an implemental function
    Dim obj As LinearSystem = New LinearSystem()

    mw_vectorb  = obj.mytranspose(mw_vectorb)
    mw_x        = obj.mymldivide(mw_A, mw_vectorb)

    ' convert back to double
    Dim x() As Double = mw_x.ToVector(MWArrayComponent.Real)

    ' print out
    Console.WriteLine(ControlChars.Tab + "result=")
    Console.WriteLine("{0}", mw_x)

    ' or
    Console.WriteLine(ControlChars.Tab + "result=")
    PrintValues(x)

    ' free memory
    mw_A.Dispose()
    mw_vectorb.Dispose()
    mw_x.Dispose()

End Sub
```

```
' ****************************
Public Sub LU_decompression()

  ' Find lower and upper matrixes
  Dim A(,) As Double = New Double (,) { {1.1,  5.6, 3.3}  , _
                                        {4.4, 12.3, 6.6}  , _
                                        {7.7,  8.8, 9.9}  }

  ' Declare variables
  Dim mw_A As MWNumericArray = new MWNumericArray(A)

  Dim mw_ArrayOut() As MWArray = Nothing

  Dim mw_L As MWNumericArray = Nothing
  Dim mw_U As MWNumericArray = Nothing
  Dim mw_P As MWNumericArray = Nothing

  ' call an implemental function
  ' value of 3 is three of ouput in mylu.m
  Dim obj As LinearSystem = New LinearSystem()
  mw_ArrayOut = obj.mylu(3, mw_A)

  'LinearSystem objLinear = new LinearSystem() ;
  mw_L = mw_ArrayOut(0)
  mw_U = mw_ArrayOut(1)
  mw_P = mw_ArrayOut(2)

  ' convert back to double
  Dim L(,) As Double = mw_L.ToArray(MWArrayComponent.Real)
  Dim U(,) As Double = mw_U.ToArray(MWArrayComponent.Real)
  Dim P(,) As Double = mw_P.ToArray(MWArrayComponent.Real)

  ' print out
  Console.WriteLine("The lower matrix L = ")
  Console.WriteLine("{0}", mw_L)
```

```
    Console.WriteLine("The upper matrix U = ")
    Console.WriteLine("{0}", mw_U)

    Console.WriteLine("P = ")
    Console.WriteLine("{0}", mw_P)

    ' or
    Console.WriteLine("Printing matrix L")
    PrintValues(L)

    Console.WriteLine("Printing matrix U")
    PrintValues(U)

    Console.WriteLine("Printing matrix P")
    PrintValues(P)

    ' free memory
    mw_A.Dispose()
    MWNumericArray.DisposeArray(mw_ArrayOut)
    mw_L.Dispose()
    mw_U.Dispose()
    mw_P.Dispose()

  End Sub

  ' ****************************
  Public Sub 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
Dim A(,) As Double = { {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   } }

Dim vectorb() As Double =  {11.1,   0  ,    22.2,  0  ,    33.3}

' declare variables
Dim mw_A        As MWNumericArray = New MWNumericArray(A)
Dim mw_vectorb  As MWNumericArray = New MWNumericArray(vectorb)
Dim mw_x        As MWNumericArray = Nothing

' call an implemental function
Dim obj As LinearSystem = New LinearSystem()

mw_vectorb = obj.mytranspose(mw_vectorb)

mw_A       = obj.mysparse(mw_A)
mw_vectorb = obj.mysparse(mw_vectorb)

mw_x       = obj.mymldivide(mw_A, mw_vectorb)
mw_x       = obj.myfull(mw_x)

'convert back to double
Dim x() As Double = mw_x.ToVector(MWArrayComponent.Real)

' print out
Console.WriteLine("Solution x :")
Console.WriteLine("{0}", mw_x)

'or
Console.WriteLine("Printing vector x:")
PrintValues(x)
```

```
  ' free memory
  mw_A.Dispose()
  mw_vectorb.Dispose()
  mw_x.Dispose()

End Sub

' ************************** */
Public Sub TridiagonalSystem()

  Dim B(5,2) As Double

  ' 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
```

```
Dim db_vectord(5) As Double
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

' The size of the matrix A
Dim row As Integer = 6
Dim col AS Integer = 6

Dim band As Integer = 3  ' tridiagnal, band width

' Establish the row size=m, and column size=n
' of the bufferA

Dim m As Integer = row
Dim n As Integer = col + (band - 1)

Dim d() As Double = { 0, 1, 2 } ' for band=3, start from 0

Dim rowB As Integer = row
Dim colB As Integer = band

' declare mxArray variables '
Dim mw_B As MWNumericArray       = New MWNumericArray(B)
Dim mw_bufferA As MWNumericArray = Nothing
Dim mw_A As MWNumericArray       = Nothing

Dim mw_d As MWNumericArray = New MWNumericArray(d)

Dim mw_vectord As MWNumericArray = New MWNumericArray(db_vectord)
Dim mw_x As MWNumericArray       = Nothing
```

```
' call an implemental function
Dim obj As LinearSystemNameSpace.LinearSystem =     _
       New LinearSystemNameSpace.LinearSystem()

' create a sparse matrix mw_bufferA from column-matrix B
mw_bufferA = obj.myspdiags(mw_B, mw_d, m, n)
mw_bufferA = obj.myfull(mw_bufferA)

' plot to see
Console.WriteLine("The buffer matrix A:")
Console.WriteLine(mw_bufferA)

 'extract the need-matrix from the buffter matrix,
 ' (start at 1 following MATLAB in m.file )
 '  from row 1 to row 6(=m) and from column 2(=start+1) to column 7(=n-1)
mw_A = obj.myextractmatrix(mw_bufferA, 1, row, 2, 7)

' plot to see
Console.WriteLine("The need-matrix A:")
Console.WriteLine(mw_A)

' solve the tridiagnal system equations
mw_A = obj.mysparse(mw_A)

mw_vectord = obj.mytranspose(mw_vectord)
mw_vectord = obj.mysparse(mw_vectord)

mw_x = obj.mymldivide(mw_A, mw_vectord)
mw_x = obj.myfull(mw_x)

' convert back to double
Dim x() As Double = mw_x.ToVector(MWArrayComponent.Real)

'print out
Console.WriteLine("Tridiagnal system solution:")
Console.WriteLine(mw_x)
```

```
  ' or
  PrintValues(x)

  ' free memory
  mw_B.Dispose()
  mw_bufferA.Dispose()
  mw_A.Dispose()

  mw_d.Dispose()

  mw_vectord.Dispose()
  mw_x.Dispose()

End Sub

' ****************************
Public Sub BandMatrixSystem()

  Dim B(5,3) As Double

  ' 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

Dim db_vectord(5) As Double
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

' The size of the matrix A
Dim row As Double = 6
Dim col As Double = 6

Dim band As Integer = 4  ' band width

' Establish the row size=m, and column size=n
' of the bufferA

Dim m As Integer = row
Dim n As Integer = col + (band - 1)
```

```
Dim d() As Double = New Double() { 0, 1, 2, 3 } ' for band = 4 start from 0

Dim rowB As Double = row
Dim colB As Double = band

'declare mxArray variables
Dim mw_B As MWNumericArray       = New MWNumericArray(B)
Dim mw_bufferA As MWNumericArray = Nothing
Dim mw_A As MWNumericArray       = Nothing

Dim mw_d As MWNumericArray= new MWNumericArray(d)

Dim  mw_vectord As MWNumericArray= New MWNumericArray(db_vectord)
Dim  mw_x As MWNumericArray= Nothing

' call an implemental function
' create a sparse matrix, size mxn, from column-matrix B

Dim obj As LinearSystemNameSpace.LinearSystem =     _
       New LinearSystemNameSpace.LinearSystem()

mw_bufferA = obj.myspdiags(mw_B, mw_d, m, n)
mw_bufferA = obj.myfull(mw_bufferA)

' plot to see
Console.WriteLine("The buffer band-matrix A:")
Console.WriteLine(mw_bufferA)

' extract the need-matrix A from the buffter matrix,
'    from row 1 to row 6 and from column 2 to column 7 */

' extract the need-matrix from the buffter matrix,
'  (start at 1 following MATLAB in m.file )
'  from row 1 to row 6(=m) and from column 2(=start+1) to column 7(=n-1) */
mw_A = obj.myextractmatrix(mw_bufferA, 1, row, 2, 7)
```

```
    ' plot to see
    Console.WriteLine("The band-need-matrix A:")
    Console.WriteLine(mw_A)

    ' solve the system equations
    mw_A = obj.mysparse(mw_A)

    mw_vectord = obj.mytranspose(mw_vectord)
    mw_vectord = obj.mysparse(mw_vectord)

    mw_x = obj.mymldivide(mw_A, mw_vectord)
    mw_x = obj.myfull(mw_x)

    ' convert back to double
    Dim solution_x() As Double = mw_x.ToVector(MWArrayComponent.Real)

    ' print out
    ' convert back to double
    Console.WriteLine("Band matrix system solution:")
    Console.WriteLine(mw_x)

    ' or
    PrintValues(solution_x)

    ' free memory
    mw_B.Dispose()
    mw_bufferA.Dispose()
    mw_A.Dispose()

    mw_d.Dispose()

    mw_vectord.Dispose()
    mw_x.Dispose()

End Sub
```

```
    ' ******************************
    Public Sub PrintValues(ByVal myArr As Array)
      Dim myEnumerator As System.Collections.IEnumerator = _
         myArr.GetEnumerator()
      Dim i As Integer = 0
      Dim cols As Integer = myArr.GetLength(myArr.Rank - 1)

      'for row vector or column vector
      If myArr.Rank = 1 Then

        While myEnumerator.MoveNext()
          Console.WriteLine(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While

      Else
        'for other
        While myEnumerator.MoveNext()
          If i < cols Then
            i += 1
          Else
            Console.WriteLine()
            i = 1
          End If
          Console.Write(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While
      End If

      Console.WriteLine()
    End Sub

  End Class

End Module
```

end code

Chapter 6

Ordinary Differential Equations

In this chapter we will generate a class ***ODE*** from common M-files working on problems of ordinary differential equations (ODE). The generated functions of this class will be used in VB .Net 2008 project to solve common ODE problems.

The MATLAB function of M-files that are used to generate the library to solve ODE problems in this chapter is `ode45(..)`. There are other functions, `ode23(..)`, `ode113(..)`, `ode15s(..)`, and `ode23s(..)`, that can be used to solve ODE problems. Therefore, we can choose a function with options that satisfies problem requirements. For more information on these functions, refer to the manual [5].

We will write the M-files as shown below. These functions will be used to generate the class ***ODE*** . The procedure to create the class ***ODE*** is the same as the procedure in Chapter 2.

```
myode45firstorder.m, yourfunc.m
myode45secondorder.m, yoursecondfunc.m
```

myode45firstorder.m

```
function [t, y] = myode45firstorder(strfunc, tspan, y0)

[t, y] = ode45(@yourfunc, tspan, y0, [], strfunc) ;
```

yourfunc.m

```
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) ;
```

myode45secondorder.m

```
function [t, y] = myode45secondorder(strfunc, tspan, y0)

[t,y] = ode45(@yoursecondfunc, tspan, y0, [], strfunc) ;
```

yoursecondfunc.m

```
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) ) ;
```

The following procedure to create the class ***ODE*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `ODENameSpace.dll` that contains the class ***ODE*** (see Fig.6.1 and Fig.6.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `ODENameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `ODENameSpace.dll`, and then click OK (see Fig. 6.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

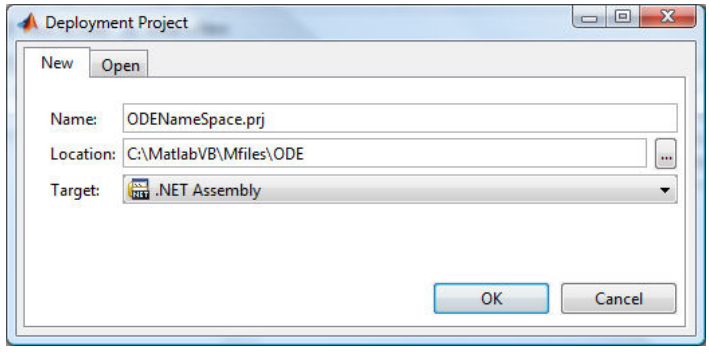


Figure 6.1: Deployment project for ***ODE***

The following sections show how to use the functions in the class ***ODE*** to solve the common computation problems. The full code is at the end of this chapter.

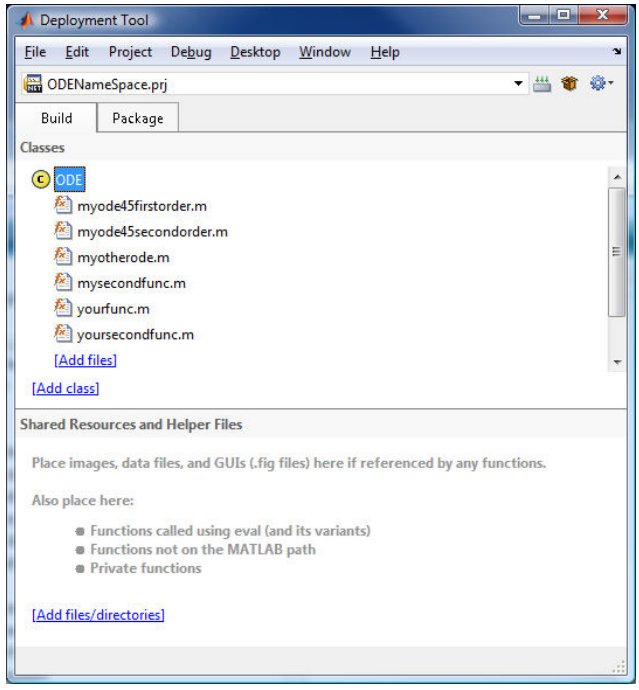


Figure 6.2: Adding M-files in deployment project for ***ODE***

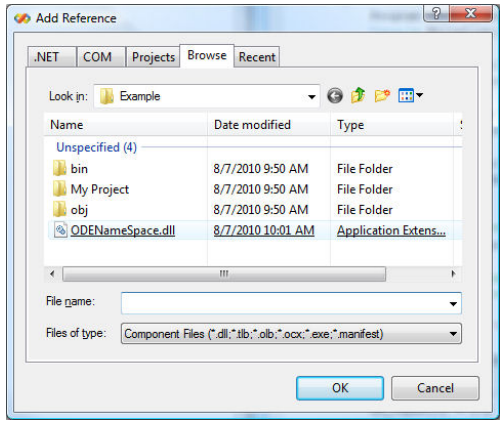


Figure 6.3: Add the reference `ODENameSpace.dll` in the VB project

6.1 First Order ODE

In this section, we will use MATLAB functions to solve the following problems.

Problem 1 Find the function, $y(t)$, from the ODE function:

$$\frac{dy}{dt} = \cos(t)$$

with initial condition :

$$y_0 = 2.2 \qquad \text{at} \quad t_0 = 0.2$$

Note:

1. When 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, $a = t_o$. The end of the interval, b, is chosen by users to show the time range in the problem.
2. The time step is set to default if we do not provide a time step.
3. The time step can be set by providing `tspan`=$[t_0, t_2, \ldots, t_n]$ 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`.
4. The ODE function is passed as an expression string to the generated function `myode45firstorder(..)` that has an argument as an expression string. The form of this expression string follows the rule of a MATLAB expression string.

The following is the code to solve Problem 1 by using the function `myode45firstorder(..)` in the generated ***ODE*** class with the time step set to default.

Listing code

```
Public Sub FirstOrder()

  ' Calculating first order ODE
  ' Dim strfunc As String = "2+y"

  Dim strfunc As String = "cos(t)"

  Dim db_y0 As Double = 2.2    ' initial condition at t0

  Dim db_tspan(1) As Double
  db_tspan(0) = 0.2    ' begin interval t0 = 0.2
  db_tspan(1) = 6.5    ' end interval, we choose this

  ' declare mxArray variables
  Dim mw_strfunc As MWCharArray  = New MWCharArray(strfunc)
  Dim mw_tspan As MWNumericArray = New MWNumericArray(db_tspan)

  Dim mw_ArrayOut() As MWArray = Nothing

  Dim mw_t As MWNumericArray = Nothing
  Dim mw_y As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As ODENameSpace.ODE = New ODENameSpace.ODE()

  ' mx_tspan is a column vector when using in the following function
  ' call an implemental function
  mw_ArrayOut = obj.myode45firstorder(2, mw_strfunc, mw_tspan, db_y0)
  mw_t        = mw_ArrayOut(0)
  mw_y        = mw_ArrayOut(1)

  ' convert back to double
  Dim db_t() As Double = mw_t.ToVector(MWArrayComponent.Real)
  Dim db_y() As Double = mw_y.ToVector(MWArrayComponent.Real)
```

```
  Console.WriteLine("The column of time")
  Console.WriteLine(mw_t)

  ' or
  PrintValues(db_t)

    Console.WriteLine("The column of the function values y")
  Console.WriteLine(mw_y)

  ' or
  PrintValues(db_y)

  ' free memory
  mw_strfunc.Dispose()
  mw_tspan.Dispose()
  mw_t.Dispose()
  mw_y.Dispose()

  MWNumericArray.DisposeArray(mw_ArrayOut)

End Sub
```

end code

Problem 2 Find the function, $y(t)$, from the ODE function:

$$\frac{dy}{dt} = 6.4t^2 - 3.8ty$$

with initial condition :

$$y_0 = 1.24 \qquad \text{at} \quad t_0 = 0.15$$

Problem 2 is solved similarly to Problem 1. In the code we just change the expression string:

```
String 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.

Listing code

```
Public Sub FirstOrderGetParticularValues()

  ' Calculating first order ODE
  Dim strfunc As String = "6.4*t.^2 - 3.8*t*y"
  Dim db_y0 As Double = 1.24   ' initial condition at t0

  Dim db_tspan(3) As Double
  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

  Dim mw_strfunc As MWCharArray    = New MWCharArray(strfunc)
  Dim mw_tspan   As MWNumericArray = New MWNumericArray(db_tspan)

  Dim mw_ArrayOut() As MWArray = Nothing
  Dim  mw_t As MWNumericArray  = Nothing
  Dim  mw_y As MWNumericArray  = Nothing

  ' mx_tspan is a column vector when using in the following function
  ' convert double to mxArray

  ' call an implemental function

  ' call an implemental function
  Dim obj As ODENameSpace.ODE = New ODENameSpace.ODE()
  mw_ArrayOut = obj.myode45firstorder(2, mw_strfunc, mw_tspan, db_y0)
  mw_t        = mw_ArrayOut(0)
```

```
  mw_y        = mw_ArrayOut(1)

  ' convert back to double
  Dim db_t() As Double = mw_t.ToVector(MWArrayComponent.Real)
  Dim db_y() As Double = mw_y.ToVector(MWArrayComponent.Real)

  Console.WriteLine("The column of time")
  Console.WriteLine(mw_t)

  ' or
  PrintValues(db_t)

  Console.WriteLine("The column of the function values y" )
  Console.WriteLine(mw_y)

  ' or
  PrintValues(db_y)

  ' free memory
  mw_strfunc.Dispose()
  mw_tspan.Dispose()
  mw_t.Dispose()
  mw_y.Dispose()

  MWNumericArray.DisposeArray(mw_ArrayOut)

End Sub
```

end code

6.2 Second Order ODE

In this section, we will use MATLAB functions to solve the following problems.

Problem 3 Find the function $y(t)$, and its derivative $y'(t)$ from the ODE function,

$$y'' = \cos(3t) + 2y' + 6y \tag{6.1}$$

with initial conditions :

$$\text{at} \quad t_0 = 0.12, \quad y_0 = 0.2 \quad \text{and} \quad y'_0 = 1.1$$

6.2.1 Analysis of second order ODE

1. To solve Problem 3 by writing M-files in MATLAB, we can write an M-file `mysecondfunc.m` as follows:

mysecondfunc.m

```
function dy = mysecondfunc(t, y)

dy = [y(2) ; cos(3*t) + 2*y(2) + 6*y(1)] ;
% y(1) is y
% y(2) is y'
```

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 expressions from the provided equation (6.1) :

$$y = y_1$$
$$y' = y_2$$
$$y'' = y'_2$$

Problem 3 then becomes:

$$\begin{aligned} y &= y_1 \\ y' &= y_2 \\ y'' &= \cos(3t) + 2y' + 6y \\ &= \cos(3t) + 2y_2 + 6y_1 \end{aligned}$$

This is the second expression in the above 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 to create a class ***ODE*** 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) ) ;
     ```

6.2.2 Using a second order ODE function

As explained in above, we have an easy way to use the generated function `myode45secondorder(..)` in the class ***ODE*** to solve the second order ODE problems by following the steps:

1. Write the ode-function with second derivative in the left-hand-side, for example:
   $y'' = \cos(3t) + 2y' + 6y$

2. Rewrite your ode-function as a 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:
   `String strfunc = "cos(3*t) + 2*yprime + 6*y" ;`

The following is the code to solve Problem 3 by using the function `myode45secondorder(..)` in the generated ***ODE*** library .

Listing code

```
Public Sub SecondOrder()

  ' Calculating second order ODE
  Dim strfunc As String= "cos(3*t) + 2*yprime + 6*y"

  Dim db_ybc(1) As Double         ' y boundary conditions
  db_ybc(0)  = 0.2                ' initial condition of y  at t0
  db_ybc(1)  = 1.1                ' initial condition of y' at t0

  Dim db_tspan(1) As Double
  db_tspan(0) = 1.2    ' begin interval t0 = 1.2
  db_tspan(1) = 2.5    ' end interval, we choose this

  ' declare mxArray variables
  Dim mw_strfunc As MWCharArray  = New MWCharArray(strfunc)
  Dim mw_tspan As MWNumericArray = New MWNumericArray(db_tspan)
  Dim mw_ybc As MWNumericArray   = New MWNumericArray(db_ybc)

  Dim mw_ArrayOut() As MWArray = Nothing

  Dim mw_t As MWNumericArray = Nothing
  Dim mw_y As MWNumericArray = Nothing
```

```
' call an implemental function
Dim obj As ODENameSpace.ODE = New ODENameSpace.ODE()

mw_ArrayOut = obj.myode45secondorder(2, mw_strfunc, mw_tspan, mw_ybc)
mw_t        = mw_ArrayOut(0)
mw_y        = mw_ArrayOut(1)

' convert back to double
Dim db_t()  As Double = mw_t.ToVector(MWArrayComponent.Real)
Dim db_y(,) As Double = mw_y.ToArray(MWArrayComponent.Real)

Console.WriteLine("The column of time")
Console.WriteLine(mw_t)

' or
 PrintValues(db_t)

Console.WriteLine("The column of the function values y :")
' first column of the matrix y
Dim i As Integer
Dim aLength As Integer = db_t.Length

For i=0 to (aLength-1)

  Console.Write("{0} ", db_y.GetValue(i,0).ToString() )
  Console.WriteLine()

Next

Console.WriteLine("The column of the first derivative y' :")
' second column of the matrix y
For i=0 to (aLength-1)

  Console.Write("{0} ", db_y.GetValue(i,1).ToString() )
  Console.WriteLine()
```

```
  Next

  ' free memory
  mw_strfunc.Dispose()
  mw_tspan.Dispose()
  mw_ybc.Dispose()
  mw_t.Dispose()
  mw_y.Dispose()

  MWNumericArray.DisposeArray(mw_ArrayOut)

End Sub
```

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 ode-functions to the VB 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` below will directly call the M-file `mysecondfunc.m`:

mysecondfunc.m

```
function dy = mysecondfunc(t, y)
dy = [y(2) ; cos(3*t) + 2*y(2) + 6*y(1)] ;
```

myotherode.m

```
function [t, y] = myotherode(tspan, y0)

[t,y] = ode45(@mysecondfunc, tspan, y0) ;
```

The following is full code for this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports ODENameSpace

Module Module1

  Sub Main()
    Dim objVB As Example = New Example()
    Console.WriteLine("ODE problems.")

    Console.WriteLine("First order ODE.")
    objVB.FirstOrder()

    Console.WriteLine("First order ODE. Get particular values")
    objVB.FirstOrderGetParticularValues()

    Console.WriteLine("Second order ODE.")
    objVB.SecondOrder()

  End Sub

  Public Class Example

    ' ****************************
    Public Sub FirstOrder()

      ' Calculating first order ODE
      ' Dim strfunc As String = "2+y"

      Dim strfunc As String = "cos(t)"
```

```
Dim db_y0 As Double = 2.2   ' initial condition at t0

Dim db_tspan(1) As Double
db_tspan(0) = 0.2   ' begin interval t0 = 0.2
db_tspan(1) = 6.5   ' end interval, we choose this

' declare mxArray variables
Dim mw_strfunc As MWCharArray  = New MWCharArray(strfunc)
Dim mw_tspan As MWNumericArray = New MWNumericArray(db_tspan)

Dim mw_ArrayOut() As MWArray = Nothing

Dim mw_t As MWNumericArray = Nothing
Dim mw_y As MWNumericArray = Nothing

' call an implemental function
Dim obj As ODENameSpace.ODE = New ODENameSpace.ODE()

' mx_tspan is a column vector when using in the following function
' call an implemental function
mw_ArrayOut = obj.myode45firstorder(2, mw_strfunc, mw_tspan, db_y0)
mw_t        = mw_ArrayOut(0)
mw_y        = mw_ArrayOut(1)

' convert back to double
Dim db_t() As Double = mw_t.ToVector(MWArrayComponent.Real)
Dim db_y() As Double = mw_y.ToVector(MWArrayComponent.Real)

Console.WriteLine("The column of time")
Console.WriteLine(mw_t)

' or
PrintValues(db_t)

Console.WriteLine("The column of the function values y")
```

```
  Console.WriteLine(mw_y)

  ' or
  PrintValues(db_y)

  ' free memory
  mw_strfunc.Dispose()
  mw_tspan.Dispose()
  mw_t.Dispose()
  mw_y.Dispose()

  MWNumericArray.DisposeArray(mw_ArrayOut)

End Sub

' ****************************
Public Sub FirstOrderGetParticularValues()

  ' Calculating first order ODE
  Dim strfunc As String = "6.4*t.^2 - 3.8*t*y"
  Dim db_y0 As Double = 1.24   ' initial condition at t0

  Dim db_tspan(3) As Double
  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

  Dim mw_strfunc As MWCharArray    = New MWCharArray(strfunc)
  Dim mw_tspan   As MWNumericArray = New MWNumericArray(db_tspan)

  Dim mw_ArrayOut() As MWArray = Nothing
  Dim  mw_t As MWNumericArray  = Nothing
  Dim  mw_y As MWNumericArray  = Nothing

  ' mx_tspan is a column vector when using in the following function
```

```
    ' convert double to mxArray

    ' call an implemental function

    Dim obj As ODENameSpace.ODE = New ODENameSpace.ODE()
    mw_ArrayOut = obj.myode45firstorder(2, mw_strfunc, mw_tspan, db_y0)
    mw_t        = mw_ArrayOut(0)
    mw_y        = mw_ArrayOut(1)

    ' convert back to double
    Dim db_t() As Double = mw_t.ToVector(MWArrayComponent.Real)
    Dim db_y() As Double = mw_y.ToVector(MWArrayComponent.Real)

    Console.WriteLine("The column of time")
    Console.WriteLine(mw_t)

    ' or
    PrintValues(db_t)

    Console.WriteLine("The column of the function values y" )
    Console.WriteLine(mw_y)

    ' or
    PrintValues(db_y)

    ' free memory
    mw_strfunc.Dispose()
    mw_tspan.Dispose()
    mw_t.Dispose()
    mw_y.Dispose()

    MWNumericArray.DisposeArray(mw_ArrayOut)

  End Sub

  ' ****************************
```

```
Public Sub SecondOrder()

  ' Calculating second order ODE
  Dim strfunc As String= "cos(3*t) + 2*yprime + 6*y"

  Dim db_ybc(1) As Double          ' y boundary conditions
  db_ybc(0)  = 0.2              ' initial condition of y  at t0
  db_ybc(1)  = 1.1              ' initial condition of y' at t0

  Dim db_tspan(1) As Double
  db_tspan(0) = 1.2   ' begin interval t0 = 1.2
  db_tspan(1) = 2.5   ' end interval, we choose this

  ' declare mxArray variables
  Dim mw_strfunc As MWCharArray  = New MWCharArray(strfunc)
  Dim mw_tspan As MWNumericArray = New MWNumericArray(db_tspan)
  Dim mw_ybc As MWNumericArray   = New MWNumericArray(db_ybc)

  Dim mw_ArrayOut() As MWArray = Nothing

  Dim mw_t As MWNumericArray = Nothing
  Dim mw_y As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As ODENameSpace.ODE = New ODENameSpace.ODE()

  mw_ArrayOut = obj.myode45secondorder(2, mw_strfunc, mw_tspan, mw_ybc)
  mw_t        = mw_ArrayOut(0)
  mw_y        = mw_ArrayOut(1)

  ' convert back to double
  Dim db_t()  As Double = mw_t.ToVector(MWArrayComponent.Real)
  Dim db_y(,) As Double = mw_y.ToArray(MWArrayComponent.Real)

  Console.WriteLine("The column of time")
  Console.WriteLine(mw_t)
```

```
    ' or
    PrintValues(db_t)

    Console.WriteLine("The column of the function values y :")
    ' first column of the matrix y
    Dim i As Integer
    Dim aLength As Integer = db_t.Length

    For i=0 to (aLength-1)

      Console.Write("{0} ", db_y.GetValue(i,0).ToString() )
      Console.WriteLine()

    Next

    Console.WriteLine("The column of the first derivative y' :")
    ' second column of the matrix y
    For i=0 to (aLength-1)

      Console.Write("{0} ", db_y.GetValue(i,1).ToString() )
      Console.WriteLine()

    Next

    ' free memory
    mw_strfunc.Dispose()
    mw_tspan.Dispose()
    mw_ybc.Dispose()
    mw_t.Dispose()
    mw_y.Dispose()

    MWNumericArray.DisposeArray(mw_ArrayOut)

  End Sub
```

```
    ' ********************************
    Public Sub PrintValues(ByVal myArr As Array)
      Dim myEnumerator As System.Collections.IEnumerator = _
         myArr.GetEnumerator()
      Dim i As Integer = 0
      Dim cols As Integer = myArr.GetLength(myArr.Rank - 1)

      'for row vector or column vector
      If myArr.Rank = 1 Then

        While myEnumerator.MoveNext()
          Console.WriteLine(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While

      Else
        'for other
        While myEnumerator.MoveNext()
          If i < cols Then
            i += 1
          Else
            Console.WriteLine()
            i = 1
          End If
          Console.Write(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While
      End If

      Console.WriteLine()
    End Sub

  End Class

End Module
```

end code

Chapter 7

Integration

In this chapter we will generate a class ***Integration*** from common M-files working on problems of single and double integrations. The generated functions of this library will be used in a VB .Net 2008 project to solve the integral problems.
The procedure to create the class ***Integration*** is the same as the procedure in Chapter 2.

We will write the M-files `myquad.m` and `mydblquad.m`. These functions will be used to generate the class ***Integration***.

myquad.m

```
function y = myquad(strfunc, a, b)

F = inline(strfunc) ;
y = quad(F, a, b) ;
```

mydblquad.m

```
function dbint = mydblquad(strfunc, x1, x2, y1, y2)

F = inline(strfunc)  ;
dbint = dblquad(F, x1, x2, y1, y2) ;
```

The following procedure to create the class ***Integration*** is the same as the procedure in Chapter

2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `IntegrationNameSpace.dll` that contains the class ***Integration*** (see Fig.7.1 and Fig.7.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `IntegrationNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `IntegrationNameSpace.dll`, and then click OK (see Fig. 7.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

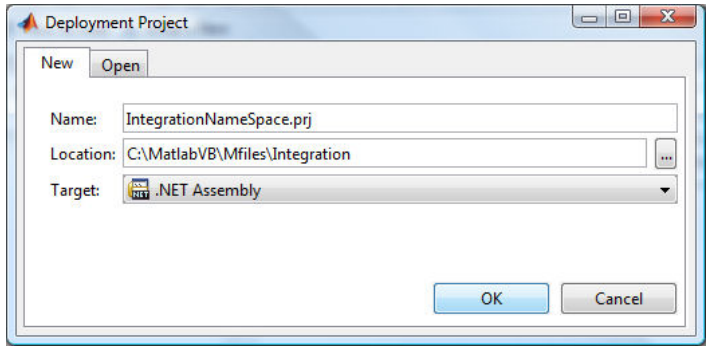


Figure 7.1: Deployment project for ***Integration***

The following sections show how to use the functions in the class ***Integration*** to solve the common computation problems. The full code is at the end of this chapter.

7.1 Single Integration

In this section, we will use MATLAB functions to solve the following problems.

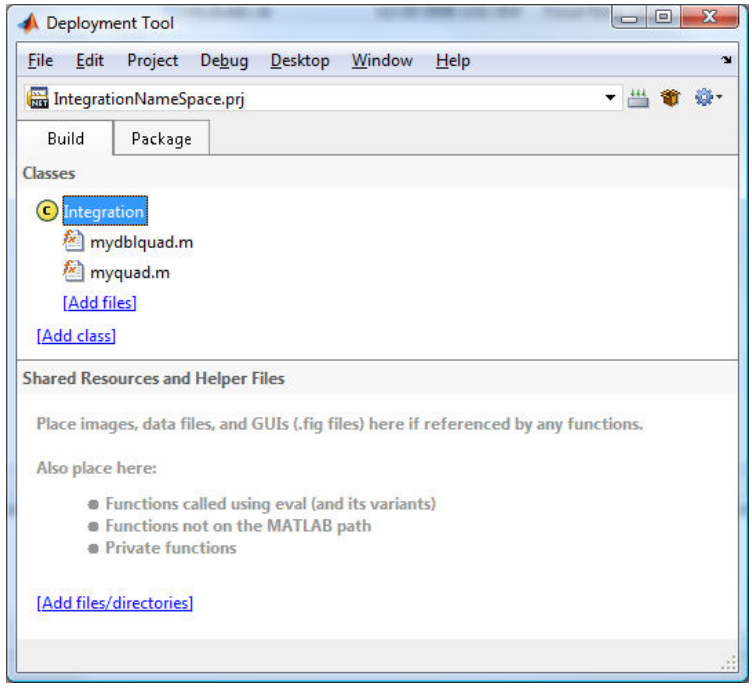


Figure 7.2: Adding M-files in deployment project for ***Integration***

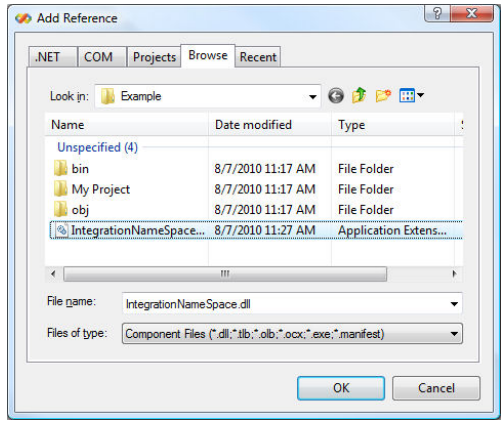


Figure 7.3: Add the reference `IntegrationNameSpace.dll` in the VB project

Problem 1 Calculate the integration :

$$I = \int_0^{3\pi} \Big(\sin(x) + x^2 \Big) dx$$

The following is the code to solve Problem 1 by using the functions `myquad(..)` in the class ***Integration***. The function `myquad(..)` uses a MATLAB function `quad(..)` with default as shown in the M-file `myquad.m`. To use more options see this function `quad(..)` refer to [5].

Listing code

```
Public Sub SingleIntegration()

  Dim strfunc As String = "sin(x) + x.^2"

  Dim db_beginInterval As Double = 0
  Dim db_endInterval As Double = 3*Math.PI  ' using the value pi in .NET

  ' declare mxArray variables */
  Dim mw_strfunc As MWCharArray = New MWCharArray(strfunc)
  Dim mw_y As MWNumericArray    = Nothing

  ' call an implemental function */
  Dim obj As IntegrationNameSpace.Integration = New IntegrationNameSpace.Integration()

  mw_y = obj.myquad(mw_strfunc, db_beginInterval, db_endInterval)

  ' print out
  Console.WriteLine(" I = {0}", mw_y )

  ' or
  Dim db_y As Double = mw_y.ToScalarDouble()
  Console.WriteLine(" I = {0}", db_y )

  ' free memory */
  mw_strfunc.Dispose()
```

```
  mw_y.Dispose()

End Sub
```

end code

Note

1. The generated function `myquad(..)` is a function that 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.

7.2 Double-Integration

In this section, we will use MATLAB functions to solve the following problems.

Problem 2 Calculate the double-integration:

$$I = \int_0^{\frac{\pi}{2}} \int_0^{\pi} \Big(\sin(x) + x^2 + y^3 \Big) dx\, dy$$

The following is the code to solve Problem 2 using the functions `mydblquad(..)` in the class ***Integration***. 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 [5].

Listing code

```
Public Sub DoubleIntegration()

  Dim strfunc As String= "sin(x) + x.^2 + y.^3"

  Dim db_x1 As Double = 0
  Dim db_x2 As Double = 3*Math.PI  ' using the value pi in NET
```

```
  Dim db_y1 As Double = 0
  Dim db_y2 As Double = Math.PI

  ' declare mxArray variables */
  Dim mw_strfunc As MWCharArray = New MWCharArray(strfunc)
  Dim mw_ArrayOut() As MWArray  = Nothing
  Dim mw_II As MWNumericArray   = Nothing

  ' call an implemental function */
  Dim obj As IntegrationNameSpace.Integration = New IntegrationNameSpace.Integration()

  mw_II = obj.mydblquad(mw_strfunc, db_x1, db_x2, db_y1, db_y2)

  ' print out
  Console.WriteLine(" II = {0}", mw_II )

  ' or
  Dim db_II As Double = mw_II.ToScalarDouble()
  Console.WriteLine(" II = {0}", db_II )

  ' free memory */
  mw_strfunc.Dispose()
  mw_II.Dispose()

End Sub
```

end code

The following is the full code for this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports IntegrationNameSpace

Module Module1
```

```
Sub Main()

Console.WriteLine("Single integration:")

Dim objVB As Example = New Example()

objVB.SingleIntegration()

Console.WriteLine("Double-integration:")
objVB.DoubleIntegration()

End Sub

Public Class Example

  ' ****************************************** */
  Public Sub SingleIntegration()

    Dim strfunc As String = "sin(x) + x.^2"

    Dim db_beginInterval As Double = 0
    Dim db_endInterval As Double = 3*Math.PI  ' using the value pi in .NET

    ' declare mxArray variables */
    Dim mw_strfunc As MWCharArray = New MWCharArray(strfunc)
    Dim mw_y As MWNumericArray    = Nothing

    ' call an implemental function */
    Dim obj As IntegrationNameSpace.Integration = New IntegrationNameSpace.Integration()

    mw_y = obj.myquad(mw_strfunc, db_beginInterval, db_endInterval)

    ' print out
    Console.WriteLine(" I = {0}", mw_y )
```

```
    ' or
    Dim db_y As Double = mw_y.ToScalarDouble()
    Console.WriteLine(" I = {0}", db_y )

    ' free memory */
    mw_strfunc.Dispose()
    mw_y.Dispose()

  End Sub

  ' ****************************************** */
  Public Sub DoubleIntegration()

    Dim strfunc As String= "sin(x) + x.^2 + y.^3"

    Dim db_x1 As Double = 0
    Dim db_x2 As Double = 3*Math.PI  ' using the value pi in NET

    Dim db_y1 As Double = 0
    Dim db_y2 As Double = Math.PI

    ' declare mxArray variables */
    Dim mw_strfunc As MWCharArray = New MWCharArray(strfunc)
    Dim mw_ArrayOut() As MWArray  = Nothing
    Dim mw_II As MWNumericArray   = Nothing

    ' call an implemental function */
    Dim obj As IntegrationNameSpace.Integration = New IntegrationNameSpace.Integration()

    mw_II = obj.mydblquad(mw_strfunc, db_x1, db_x2, db_y1, db_y2)

    ' print out
    Console.WriteLine(" II = {0}", mw_II )

    ' or
    Dim db_II As Double = mw_II.ToScalarDouble()
```

```
      Console.WriteLine(" II = {0}", db_II )

      ' free memory */
      mw_strfunc.Dispose()
      mw_II.Dispose()

    End Sub

  End Class

End Module
```

end code

Chapter 8

Polynomial Fitting and Interpolations

In this chapter we will generate a class ***PolyInterpolation*** from common M-files working on interpolation curve fitting problems. The generated functions of these libraries will be used in VB .Net 2008 project to solve common polynormial fitting problems. The procedure to create the class ***PolyInterpolation*** is the same as the procedure in Chapter 2.

We will write the M-files as shown below. These functions will be used to generate the class ***PolyInterpolation***.

```
myinterp1.m, myinterp2.m, mypolyfit.m, mypolyval.m,
mymeshgrid.m, mygriddata.m, and myfinemeshgrid.m
```

myinterp1.m

```
function yi = myinterp1(x,y,xi)

yi = interp1(x,y,xi) ;
```

myinterp2.m

```
function ZI = myinterp2(X,Y,Z,XI,YI,method)

ZI = interp2(X,Y,Z,XI,YI,method) ;
```

mypolyfit.m

```
function p = mypolyfit(x,y,n)

p = polyfit(x,y,n) ;
```

mypolyval.m

```
function y = mypolyval(p,x)

y = polyval(p,x) ;
```

mymeshgrid.m

```
function [X,Y] = mymeshgrid(vectorstepx, vectorstepy)

%

[X,Y] = meshgrid( vectorstepx(1):vectorstepx(2):vectorstepx(3), ...
                  vectorstepy(1):vectorstepy(2):vectorstepy(3) ) ;
```

mygriddata.m

```
function ZI = mygriddata(x,y,z,XI,YI)

ZI = griddata(x,y,z,XI,YI) ;
```

myfinemeshgrid.m

```
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) ;
```

The following procedure to create the class ***PolyInterpolation*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `PolyInterpolationNameSpace.dll` that contains the class ***PolyInterpolation*** (see Fig.8.1 and Fig.8.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `PolyInterpolationNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `PolyInterpolationNameSpace.dll`, and then click OK (see Fig. 8.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

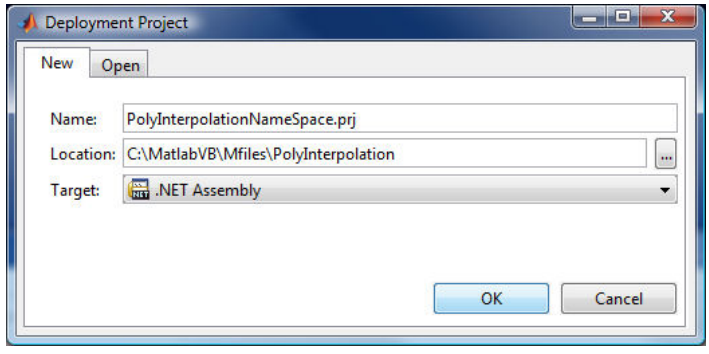


Figure 8.1: Deployment project for ***PolyInterpolation***

8.1 Polynomial Curve Fitting

This section describes how to use the functions in the generated library to to find the coefficients of a polynomial function that fits a set of data in the least-squares sense. An array **c** representing

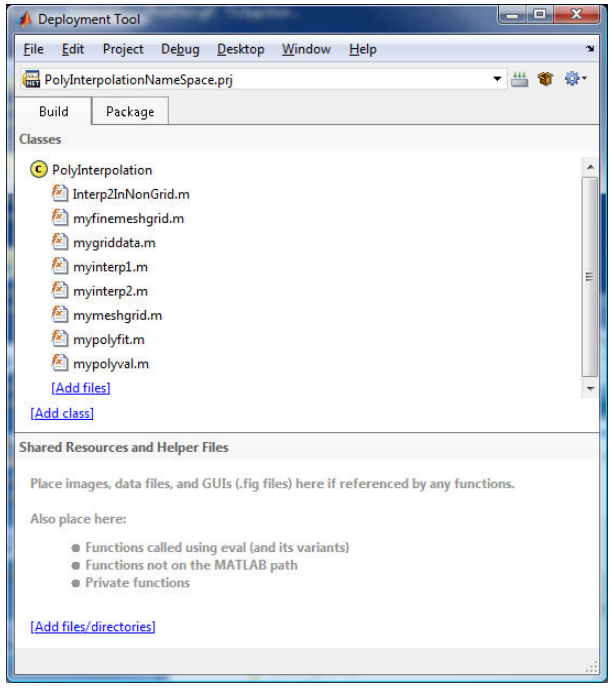


Figure 8.2: Adding M-files in deployment project for ***PolyInterpolation***

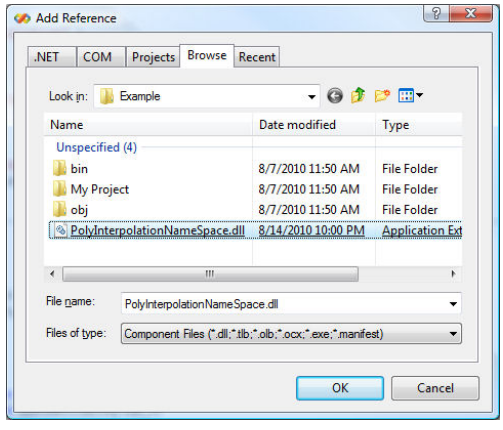


Figure 8.3: Add the reference `PolyInterpolationNameSpace.dll` in the VB project

these coefficients is in the polynomial form:

$$f(x) = c_1 x^n + c_2 x^{n-1} + c_3 x^{n-2} + \cdots + c_n x + c_{n+1} \tag{8.1}$$

In this section, we will use MATLAB functions to solve the following problems.

Problem 1

input . There are two arrays **X** and **Y** which have a relationship via a function, $y = f(x)$, $x \in \mathbf{X}$ and $y \in \mathbf{Y}$.

Array $\mathbf{X}$ = { 1, 2, 3, 4, 5, 6 }

Array $\mathbf{Y}$ = { 6.8, 50.2, 140.8, 280.5, 321.4, 428.6 }

output . Finding an array **c** of the polynomial function in Eq. 8.1 with degree n=3; since degree n=3, the function $y = f(x)$ has the form:

$$y = c_1 x^3 + c_2 x^2 + c_3 x + c_4$$

. Calculating the interpolation value of the function $y(x)$, at $x = 2.2$

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

```
Public Sub PolynomialFittingCurve()

  Dim db_X() As Double = { 1, 2, 3, 4, 5, 6 }
  Dim db_Y() As Double = { 6.8, 50.2, 140.8, 280.5, 321.4, 428.6 }

  Dim db_oneValue As Double = 2.2

  ' declare mwArray variables
  Dim mw_X As MWNumericArray          = New MWNumericArray(db_X)
  Dim mw_Y As MWNumericArray          = New MWNumericArray(db_Y)
  Dim mw_coefs As MWNumericArray      = Nothing
  Dim mw_funcValue As MWNumericArray  = Nothing
```

```
' call an implemental function
' mw_coefs is a matrix with row = 1
Dim obj As PolyInterpolationNameSpace.PolyInterpolation = _
       New PolyInterpolationNameSpace.PolyInterpolation()
mw_coefs = obj.mypolyfit(mw_X, mw_Y, 3)

' print out
Console.WriteLine("The coefficient value :")
Console.WriteLine("{0}", mw_coefs)

' convert back to double
Dim db_coefs() As Double = mw_coefs.ToVector(MWArrayComponent.Real)

' print out
Console.WriteLine("The polynomial:" )
Console.Write( " {0}x^3 + ",  db_coefs(0).ToString() )
Console.Write( " {0}x^2 + ",  db_coefs(1).ToString() )
Console.Write( " {0}x + "  ,  db_coefs(2).ToString() )
Console.Write(                db_coefs(3).ToString() )
Console.WriteLine()

' calculate the function value at oneValue
mw_funcValue = obj.mypolyval(mw_coefs, db_oneValue)

Dim db_funcValue As Double = mw_funcValue.ToScalarDouble()

Console.WriteLine("The function value at 2.2 is:")
Console.WriteLine("{0}", mw_funcValue )

Console.WriteLine("The function value at 2.2 is")
Console.WriteLine("{0}", db_funcValue)

' free memory
mw_X.Dispose()
mw_Y.Dispose()
mw_coefs.Dispose()
```

```
    mw_funcValue.Dispose()

End Sub
```

end code

8.2 One-Dimensional Polynomial Interpolation

This section describes how to use the functions in the generated class to solve an one-dimensional interpolation problem. This function uses polynomial techniques to evaluate the value of a function at a desired interpolation point.

Problem 2

input . The two two arrays $\mathbf{X}$ and $\mathbf{Y}$ have a relationship via a function, $y = f(x)$, $x \in \mathbf{X}$ and $y \in \mathbf{Y}$.

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 `myinterp1(..)` with the default method (liner method) to solve the problem. To learn more about other possible methods, see the MATLAB function `inter1(..)` in [4].

Listing code

```
Public Sub OneDimensionInterpolation()

  Dim db_X() As Double = { 1, 2, 3, 4, 5, 6, 7, 8 }
  Dim db_Y() As Double = { 6.8, 24.6, 50.2, 74, 140.8, 280.5, 321.4, 428.6 }

  Dim db_oneValue As Double  = 2.1

  ' declare mwArray variables
  Dim mw_X As MWNumericArray = New MWNumericArray(db_X)
  Dim mw_Y As MWNumericArray = New MWNumericArray(db_Y)
```

```
    Dim mw_funcValue As MWNumericArray = Nothing

    ' call an implemental function
    Dim obj As PolyInterpolationNameSpace.PolyInterpolation = _
           New PolyInterpolationNameSpace.PolyInterpolation()
    mw_funcValue = obj.myinterp1(mw_X, mw_Y, db_oneValue)

    ' print out
    Console.WriteLine( "The function value at 2.1 is:")
    Console.WriteLine( "{0}", mw_funcValue )

    ' convert back to double
    Dim db_funcValue As Double = mw_funcValue.ToScalarDouble()
    Console.WriteLine( "The function value at 2.1 is:")
    Console.WriteLine("{0}", db_funcValue.ToString() )

    ' free memory
    mw_X.Dispose()
    mw_Y.Dispose()
    mw_funcValue.Dispose()

End Sub
```

end code

8.3 Two-Dimensional Polynomial Interpolation for Grid Points

This section describes how to use the functions in the generated class to solve a two-dimensional interpolation problem. This function uses polynomial techniques to evaluate the value of a function at a desired interpolation point.

Problem 3

input . 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 $z \in \mathbf{z}$

. Representation of the grid points (x,y) is two matrixes:

matrix **x** contains the x-direction values of all grid points

matrix **y** contains the y-direction values of all grid points

. Matrix **z** contains the function values $z(x, y)$ of all grid points

(the values of matrixes **x**, **y**, and **z** are shown below)

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 we have grid points as in Fig. 8.4.

The matrix **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
```

The matrix **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
```

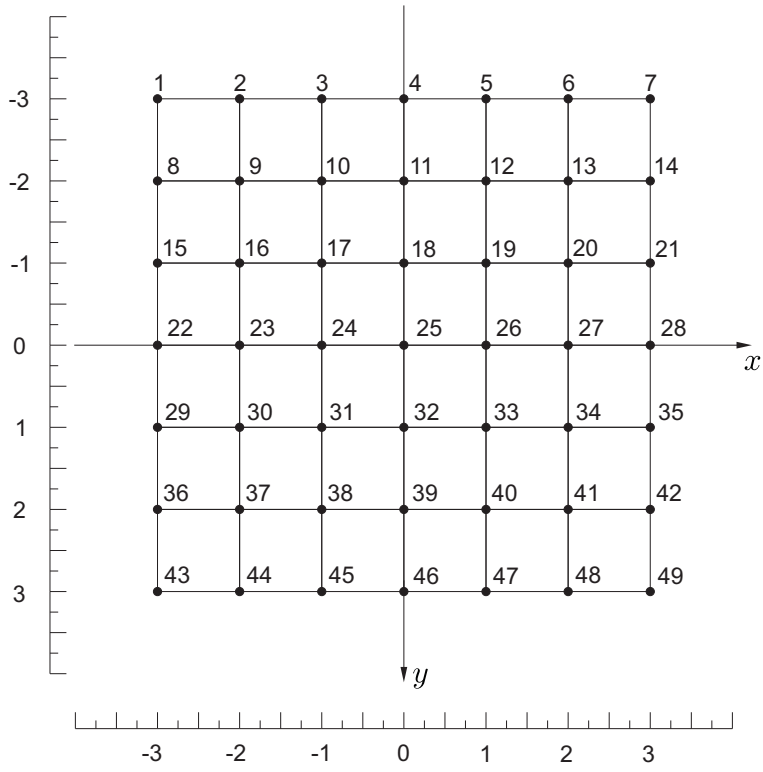


Figure 8.4: Grid points

The matrix **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 code is used to solve Problem 3. In the code, you will use the generated function `myinterp2(..)` with the *cubic* method to solve the problem. To learn more about other possible methods see the MATLAB function `inter2(..)` in [4]. The procedure of this code is:

1. Assign values for matrix **x** and matrix **y**
2. Assign values for matrix **z**
3. Use the two-dimensional interpolation function to evaluate the value at the point (a, b)

Listing code

```
Public Sub TwoDimensionsInterpolation()


  ' function values z at 47 points, (x_i, y_j) are :
  ' matrix z values
Dim z(,) As Double = { _
{  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  }  }
```

```
Dim db_vectorstep() As Double = {-3, 1, 3}
Dim db_a As Double = 2.3
Dim db_b As Double = 0.7

' declare mwArray variables
Dim interp2method As String = "cubic"

Dim mw_interp2z As MWNumericArray = Nothing
Dim mw_z As MWNumericArray = New MWNumericArray(z)
Dim mw_x As MWNumericArray = Nothing
Dim mw_y As MWNumericArray = Nothing

' same for two step-vectors
Dim mw_vectorstepx As MWNumericArray = New MWNumericArray(db_vectorstep)
Dim mw_vectorstepy As MWNumericArray = New MWNumericArray(db_vectorstep)

Dim mw_method As MWCharArray = New MWCharArray(interp2method)
Dim mw_ArrayGrid() As MWArray = Nothing

' call implemental functions
Dim obj As PolyInterpolationNameSpace.PolyInterpolation = _
       New PolyInterpolationNameSpace.PolyInterpolation()

' create values for the matrix x and matrix y
mw_ArrayGrid = obj.mymeshgrid(2, mw_vectorstepx, mw_vectorstepy)

mw_x =  mw_ArrayGrid(0)
mw_y =  mw_ArrayGrid(1)

mw_interp2z = obj.myinterp2(mw_x, mw_y, mw_z, db_a, db_b, mw_method)

'convert back to double
Dim db_x(,) As Double = mw_x.ToArray(MWArrayComponent.Real)
Dim db_y(,) As Double = mw_y.ToArray(MWArrayComponent.Real)

' print out
```

```
  Console.WriteLine("Matrix x")
  Console.WriteLine("{0}", mw_x )

  ' or
  Console.WriteLine("Matrix x")
  PrintValues(db_x)

  Console.WriteLine("Matrix y :")
  Console.WriteLine("{0}", mw_y )

  ' or
  Console.WriteLine("Matrix y :")
  PrintValues(db_y)

  Console.WriteLine("Interpolation in two dimensions with cubic method at 2.3 and 0.7  " )
  Console.WriteLine(" z = " )
  Console.WriteLine("{0}", mw_interp2z )

  'or
  Console.WriteLine("Interpolation in two dimensions with cubic method at 2.3 and 0.7  " )
  Dim db_interp2z As Double = mw_interp2z.ToScalarDouble()
  Console.WriteLine(" z = ")
  Console.WriteLine("{0}", db_interp2z.ToString() )

  ' free memory
  mw_interp2z.Dispose()
  mw_z.Dispose()
  mw_x.Dispose()
  mw_y.Dispose()
  mw_vectorstepx.Dispose()
  mw_vectorstepy.Dispose()
  mw_method.Dispose()

  MWNumericArray.DisposeArray(mw_ArrayGrid)

End Sub
```

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. 8.4, the y axis direction is from top to bottom, therefore pay attention when assigning your data into the matrix **y**.

2. The matrix **z** values will be assigned according to the same rules (left to right, top to bottom) as matrixes **x** and **y** (Fig. 8.5). Therefore to avoid mistakes and to have the convenience of visibility, you can set up your matrix data as in Fig. 8.4 (**y axis direction from top to bottom**).

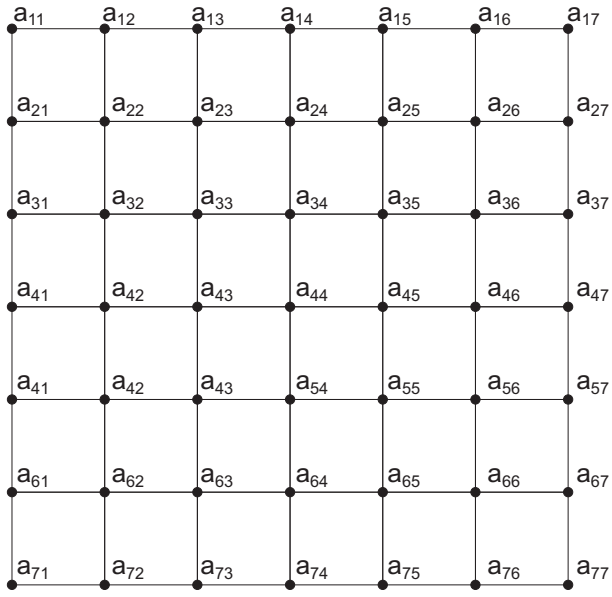


Figure 8.5: A matrix form for grid points in two-dimensional interpolation

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 sur-

face 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 [4]. The function `TwoDimensionsInterpolationFineSolution()` in the end of this chapter finds the fine solution of Problem 3 by using the function `mygriddata(..)`.

The following is the full code for this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports PolyInterpolationNameSpace

Module Module1

  Sub Main()
    Dim objVB As Example = New Example()
    Console.WriteLine("Curve Fitting")

    Console.WriteLine()
    Console.WriteLine("1. Polynomial")
    objVB.PolynomialFittingCurve()

    Console.WriteLine()
    Console.WriteLine("2. One-dimensional interplation" )
    objVB.OneDimensionInterpolation()

    Console.WriteLine()
    Console.WriteLine("3. Two-dimensional interpolation")
    objVB.TwoDimensionsInterpolation()

    Console.WriteLine()
    Console.WriteLine("4. Two-dimensional interpolation with fine grid")
    objVB.TwoDimensionsInterpolationFineSolution()
```

```
End Sub
Public Class Example
  ' ********************
  Public Sub PolynomialFittingCurve()

    Dim db_X() As Double = { 1, 2, 3, 4, 5, 6 }
    Dim db_Y() As Double = { 6.8, 50.2, 140.8, 280.5, 321.4, 428.6 }

    Dim db_oneValue As Double = 2.2

    ' declare mwArray variables
    Dim mw_X As MWNumericArray          = New MWNumericArray(db_X)
    Dim mw_Y As MWNumericArray          = New MWNumericArray(db_Y)
    Dim mw_coefs As MWNumericArray      = Nothing
    Dim mw_funcValue As MWNumericArray  = Nothing

    ' call an implemental function
    ' mw_coefs is a matrix with row = 1
    Dim obj As PolyInterpolationNameSpace.PolyInterpolation = _
           New PolyInterpolationNameSpace.PolyInterpolation()
    mw_coefs = obj.mypolyfit(mw_X, mw_Y, 3)

    ' print out
    Console.WriteLine("The coefficient value :")
    Console.WriteLine("{0}", mw_coefs)

    ' convert back to double
    Dim db_coefs() As Double = mw_coefs.ToVector(MWArrayComponent.Real)

    ' print out
    Console.WriteLine("The polynomial:" )
    Console.Write( " {0}x^3 + ",  db_coefs(0).ToString() )
    Console.Write( " {0}x^2 + ",  db_coefs(1).ToString() )
    Console.Write( " {0}x + "  ,  db_coefs(2).ToString() )
    Console.Write(                db_coefs(3).ToString() )
    Console.WriteLine()
```

```
  ' calculate the function value at oneValue
  mw_funcValue = obj.mypolyval(mw_coefs, db_oneValue)

  Dim db_funcValue As Double = mw_funcValue.ToScalarDouble()

  Console.WriteLine("The function value at 2.2 is:")
  Console.WriteLine("{0}", mw_funcValue )

  Console.WriteLine("The function value at 2.2 is")
  Console.WriteLine("{0}", db_funcValue)

  ' free memory
  mw_X.Dispose()
  mw_Y.Dispose()
  mw_coefs.Dispose()
  mw_funcValue.Dispose()

End Sub

' ***************************
Public Sub OneDimensionInterpolation()

  Dim db_X() As Double = { 1, 2, 3, 4, 5, 6, 7, 8 }
  Dim db_Y() As Double = { 6.8, 24.6, 50.2, 74, 140.8, 280.5, 321.4, 428.6 }

  Dim db_oneValue As Double  = 2.1

  ' declare mwArray variables
  Dim mw_X As MWNumericArray = New MWNumericArray(db_X)
  Dim mw_Y As MWNumericArray = New MWNumericArray(db_Y)
  Dim mw_funcValue As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As PolyInterpolationNameSpace.PolyInterpolation = _
         New PolyInterpolationNameSpace.PolyInterpolation()
```

```
    mw_funcValue = obj.myinterp1(mw_X, mw_Y, db_oneValue)

    ' print out
    Console.WriteLine( "The function value at 2.1 is:")
    Console.WriteLine( "{0}", mw_funcValue )

    ' convert back to double
    Dim db_funcValue As Double = mw_funcValue.ToScalarDouble()
    Console.WriteLine( "The function value at 2.1 is:")
    Console.WriteLine("{0}", db_funcValue.ToString() )

    ' free memory
    mw_X.Dispose()
    mw_Y.Dispose()
    mw_funcValue.Dispose()

End Sub

' ****************************
Public Sub TwoDimensionsInterpolation()

    ' function values z at 47 points, (x_i, y_j) are :
    ' matrix z values
Dim z(,) As Double = { _
{  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  }  }

    Dim db_vectorstep() As Double = {-3, 1, 3}
    Dim db_a As Double = 2.3
    Dim db_b As Double = 0.7
```

```
' declare mwArray variables
Dim interp2method As String = "cubic"

Dim mw_interp2z As MWNumericArray = Nothing
Dim mw_z As MWNumericArray = New MWNumericArray(z)
Dim mw_x As MWNumericArray = Nothing
Dim mw_y As MWNumericArray = Nothing

' same for two step-vectors
Dim mw_vectorstepx As MWNumericArray = New MWNumericArray(db_vectorstep)
Dim mw_vectorstepy As MWNumericArray = New MWNumericArray(db_vectorstep)

Dim mw_method As MWCharArray = New MWCharArray(interp2method)
Dim mw_ArrayGrid() As MWArray = Nothing

' call implemental functions
Dim obj As PolyInterpolationNameSpace.PolyInterpolation = _
       New PolyInterpolationNameSpace.PolyInterpolation()

' create values for the matrix x and matrix y
mw_ArrayGrid = obj.mymeshgrid(2, mw_vectorstepx, mw_vectorstepy)

mw_x =  mw_ArrayGrid(0)
mw_y =  mw_ArrayGrid(1)

mw_interp2z = obj.myinterp2(mw_x, mw_y, mw_z, db_a, db_b, mw_method)

'convert back to double
Dim db_x(,) As Double = mw_x.ToArray(MWArrayComponent.Real)
Dim db_y(,) As Double = mw_y.ToArray(MWArrayComponent.Real)

' print out
Console.WriteLine("Matrix x")
Console.WriteLine("{0}", mw_x )

' or
```

```
    Console.WriteLine("Matrix x")
    PrintValues(db_x)

    Console.WriteLine("Matrix y :")
    Console.WriteLine("{0}", mw_y )

    ' or
    Console.WriteLine("Matrix y :")
    PrintValues(db_y)

    Console.WriteLine("Interpolation in two dimensions with cubic method at 2.3 and 0.7  " )
    Console.WriteLine(" z = " )
    Console.WriteLine("{0}", mw_interp2z )

    'or
    Console.WriteLine("Interpolation in two dimensions with cubic method at 2.3 and 0.7  " )
    Dim db_interp2z As Double = mw_interp2z.ToScalarDouble()
    Console.WriteLine(" z = ")
    Console.WriteLine("{0}", db_interp2z.ToString() )

    ' free memory
    mw_interp2z.Dispose()
    mw_z.Dispose()
    mw_x.Dispose()
    mw_y.Dispose()
    mw_vectorstepx.Dispose()
    mw_vectorstepy.Dispose()
    mw_method.Dispose()

    MWNumericArray.DisposeArray(mw_ArrayGrid)

End Sub

'* ***************************
Public Sub TwoDimensionsInterpolationFineSolution()
```

```
    ' function values z, (x_i, y_j) are :
    ' matrix z values

    Dim z(,) As Double = { _
{  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  }  }

    Dim db_vectorstep() As Double     = {-3,  1 , 3} ' interval = 1
    Dim db_finevectorstep() As Double = {-3, 0.2, 3} ' interval = 0.2

    Dim db_a As Double = 2.3
    Dim db_b As Double = 0.7
    Dim interp2method As String = "cubic"

    ' declare mwArray variables
    Dim mw_interp2z  As MWNumericArray = Nothing
    Dim mw_z As MWNumericArray = New MWNumericArray(z)
    Dim mw_finez As MWNumericArray = New MWNumericArray(z)

    Dim mw_x As MWNumericArray = Nothing
    Dim mw_y As MWNumericArray = Nothing

    Dim mw_finex As MWNumericArray = Nothing
    Dim mw_finey As MWNumericArray = Nothing

    ' same for two step-vectors
    Dim mw_vectorstepx As MWNumericArray = New MWNumericArray(db_vectorstep)
    Dim mw_vectorstepy As MWNumericArray = new MWNumericArray(db_vectorstep)

    Dim mw_method As MWCharArray = New MWCharArray(interp2method)
```

```
Dim mw_finevectorstepx As MWNumericArray = New MWNumericArray(db_finevectorstep)
Dim mw_finevectorstepy As MWNumericArray = New MWNumericArray(db_finevectorstep)

Dim mw_ArrayGridXY() As MWArray = Nothing

Dim mw_ArrayMatrixXY() As MWArray = Nothing
Dim mw_ArrayMatrixXY_fine() As MWArray = Nothing

Dim mw_ArrayGridXY_fine() As MWArray = Nothing
Dim mw_ArrayFineVector() As MWArray = Nothing

' call implemental functions
' get size for fine matrixes
Dim obj As PolyInterpolationNameSpace.PolyInterpolation = _
       New PolyInterpolationNameSpace.PolyInterpolation()

' create values for the matrix x and matrix y
mw_ArrayMatrixXY = obj.mymeshgrid(2, mw_vectorstepx, mw_vectorstepy)
mw_x = mw_ArrayMatrixXY(0)
mw_y = mw_ArrayMatrixXY(1)

'Console.WriteLine(" see normal x = ",  mw_x ) ;

' create values for the fine matrix x and fine matrix y
mw_ArrayMatrixXY_fine = obj.mymeshgrid(2, mw_finevectorstepx, mw_finevectorstepy)
mw_finex = mw_ArrayMatrixXY_fine(0)
mw_finey = mw_ArrayMatrixXY_fine(1)

'Console.WriteLine(" see fine x = ",  mw_finex )

' get a fine matrix mx_finez from mx_z
mw_finez = obj.mygriddata(mw_x, mw_y, mw_z, mw_finex, mw_finey)

Console.WriteLine(" see fine z = {0}",  mw_finez )

mw_interp2z = obj.myinterp2(mw_finex, mw_finey, mw_finez, db_a, db_b, mw_method)
```

```
  ' convert back to double
  Dim db_interp2finez As Double = mw_interp2z.ToScalarDouble()

  Console.Write("Interpolation in two dimensions with cubic method ")
  Console.WriteLine("and a fine grid")
  Console.WriteLine(" z = {0}",  mw_interp2z )
  Console.WriteLine(" z = {0}",  db_interp2finez.ToString() )

  ' free memory
  mw_interp2z.Dispose()
  mw_z.Dispose()
  mw_finez.Dispose()

  mw_x.Dispose()
  mw_y.Dispose()

  mw_vectorstepx.Dispose()
  mw_vectorstepy.Dispose()
  mw_method.Dispose()

  mw_finevectorstepx.Dispose()
  mw_finevectorstepy.Dispose()

  MWNumericArray.DisposeArray(mw_ArrayGridXY)
  MWNumericArray.DisposeArray(mw_ArrayGridXY_fine)
  MWNumericArray.DisposeArray(mw_ArrayFineVector)

End Sub

' ********************************
Public Sub PrintValues(ByVal myArr As Array)
  Dim myEnumerator As System.Collections.IEnumerator = _
     myArr.GetEnumerator()
  Dim i As Integer = 0
  Dim cols As Integer = myArr.GetLength(myArr.Rank - 1)
```

```
      'for row vector or column vector
      If myArr.Rank = 1 Then

        While myEnumerator.MoveNext()
          Console.WriteLine(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While

      Else
        'for other
        While myEnumerator.MoveNext()
          If i < cols Then
            i += 1
          Else
            Console.WriteLine()
            i = 1
          End If
          Console.Write(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While
      End If

      Console.WriteLine()
    End Sub

  End Class

End Module
```

end code

Chapter 9

Using MATLAB Curve Fitting Toolbox In VB .NET Functions

In this chapter we will generate a class ***CurveFittings*** from M-files to use functions of Curve Fitting Toolbox in VB functions. The generated functions of this class will be used in a VB .Net 2008 project to plot curves and show information of a curve fitting.

Note that, in order to run applications in this chapter you need to have MATLAB Curve Fitting Toolbox in your computer.

We will write the M-files as shown below to generate that class.

`CurveFitting.m, mytextread.m, mytranspose.m, CurveFittingWithPlots.m`

CurveFittings.m

```
function varargout = CurveFittings(x,y,cftLib)

[cft_data,cft_info] = fit(x,y,cftLib);
formula_a    = formula(cft_data)     ;
varargout{1} = formula_a ;

coeff_names  = coeffnames(cft_data)  ;
varargout{2} = coeff_names ;
```

```
coeff_values = coeffvalues(cft_data) ;
varargout{3} = coeff_values ;

conf_int     = confint(cft_data)     ;
varargout{4} = conf_int ;

varargout{5} = cft_info ;
```

CurveFittingWithPlots.m

```
function  varargout = CurveFittingWithPlots(x,y,cftLib)

%%%%%%% Curve fit data %%%%%%%%%%
[cft_data,cft_info] = fit(x,y,cftLib);
formula_a    = formula(cft_data)     ;
varargout{1} = formula_a ;

coeff_names  = coeffnames(cft_data)  ;
varargout{2} = coeff_names ;

coeff_values = coeffvalues(cft_data) ;
varargout{3} = coeff_values ;

conf_int     = confint(cft_data)     ;
varargout{4} = conf_int ;

varargout{5} = cft_info ;
%%%%%%%%%%%%%%%%%

close all ;
cftResult = fit(x,y, cftLib);

Hcfit = plot(cftResult) ;
hold on ; % require here: after plot result
```

```
set(Hcfit,'Color','red','Marker','.','LineStyle','-') ;

Hcfit_legend = 'Fitting curve';

Hdata = plot(x,y) ;
set(Hdata, 'Color', 'blue','Marker','*','LineStyle','none') ;
Hdata_legend = 'data' ;

Hlegend = legend(Hcfit_legend, Hdata_legend) ;
set(Hlegend, 'FontWeight', 'bold') ;

hold off ;

grid on ;
```

CurveFittingWithPlotsAdvance.m

```
function  varargout = CurveFittingWithPlotsAdvance(x,y,cftLib, ...
                                          graTitle, xlab, ylab, ...
                       cftColor, dataColor, cftLegend, dataLengend)

%%%%%%% Curve fit data %%%%%%%%%%
[cft_data,cft_info] = fit(x,y,cftLib);
formula_a    = formula(cft_data)     ;
varargout{1} = formula_a ;

coeff_names  = coeffnames(cft_data)  ;
varargout{2} = coeff_names ;

coeff_values = coeffvalues(cft_data) ;
varargout{3} = coeff_values ;

conf_int     = confint(cft_data)     ;
varargout{4} = conf_int ;
```

```
varargout{5} = cft_info ;
%%%%%%%%%%%%%%%%%

close all ;
cftResult = fit(x,y, cftLib);

Hcfit = plot(cftResult) ;
hold on ; % require here: after plot result

set(Hcfit,'Color',cftColor,'Marker','.','LineStyle','-') ;

Hcfit_legend = cftLegend ;

Hdata = plot(x,y) ;
set(Hdata,'Color',dataColor,'Marker','*','LineStyle','none') ;
Hdata_legend = dataLengend ;

Hlegend = legend(Hcfit_legend, Hdata_legend) ;
set(Hlegend, 'FontWeight', 'bold') ;

title(graTitle) ;
xlabel(xlab) ;
ylabel(ylab) ;



hold off ;

grid on ;
```

mytextread.m

```
function varargout = mytextread(fileName, colNum, mydelimiter)

  switch colNum
```

```
    case 2
    [A1, A2] = textread(fileName, '%f%f','delimiter', mydelimiter) ;
    varargout{1} = A1 ; varargout{2} = A2 ;

    case 3
    [A1, A2, A3] = textread(fileName, '%f%f%f', 'delimiter', mydelimiter) ;
    varargout{1} = A1 ; varargout{2} = A2 ; varargout{3} = A3 ;

    case 4
    [A1, A2, A3, A4] = textread(fileName, '%f%f%f%f', 'delimiter', mydelimiter) ;
    varargout{1} = A1 ; varargout{2} = A2 ; varargout{3} = A3 ;
    varargout{4} = A4 ;

    case 5
    [A1, A2, A3, A4, A5] = textread(fileName, '%f%f%f%f%f', 'delimiter', mydelimiter) ;
    varargout{1} = A1 ; varargout{2} = A2 ; varargout{3} = A3 ;
    varargout{4} = A4 ; varargout{5} = A5 ;

    otherwise
        disp('This function reads files with max colums = 5.');

  end
```

mytranspose.m

```
function y = mytranspose( x )

y = x' ;
```

The following procedure to create the class ***CurveFitting*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file

`CurveFittingNameSpace.dll` that contains the class ***CurveFitting*** (see Fig.9.1 and Fig.9.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `CurveFittingNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `CurveFittingNameSpace.dll`, and then click OK (see Fig. 9.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

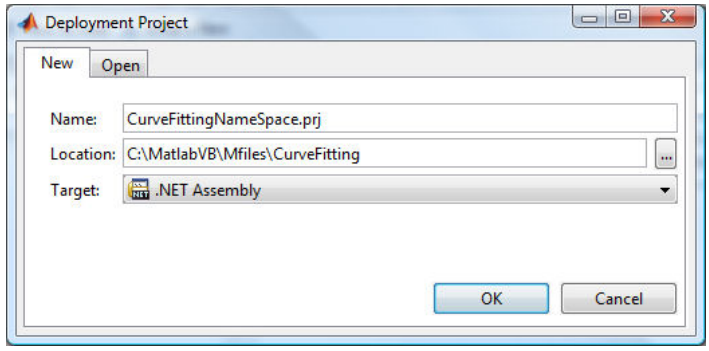


Figure 9.1: Deployment project for ***CurveFitting***

The following sections show how to use the functions in the class ***CurveFitting*** to solve common Curve Fitting problems. The full code is at the end of this chapter.

9.1 Using Curve Fitting Toolbox functions in VB .NET

Problem 1

input . There are two data arrays:

```
X = {0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600} ;
```

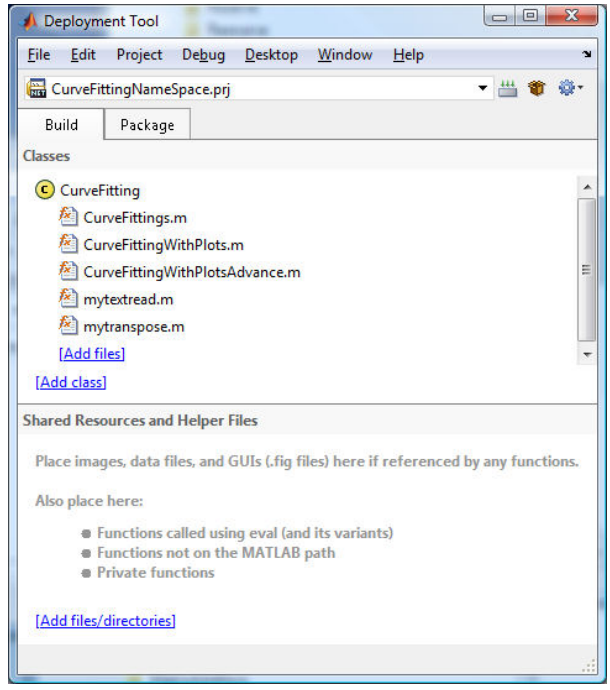


Figure 9.2: Adding M-files in deployment project for ***CurveFitting***

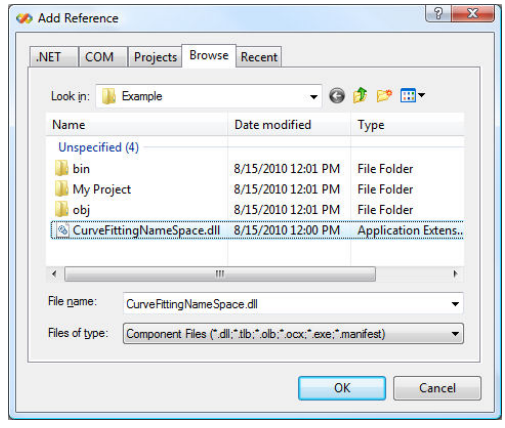


Figure 9.3: Add the reference `CurveFittingNameSpace.dll` in the VB project

```
Y = {0.4000, 0.2426, 0.1472, 0.0893, 0.0541, 0.0328, 0.0199,
                0.0121, 0.0073, 0.0044, 0.0027, 0.0016,  0.0010 } ;
```

output . Find a Curve Fitting exponent function and its information, with the formula of exponent function:

$$a * exp(b * x) + c * exp(d * x)$$

The code below of the subroutine `CurveFittingInfo()` is for this problem. The result output is:

```
Curve fitting formula :
a*exp(b*x) + c*exp(d*x)

Coefficient names:
    'a'
    'b'
    'c'
    'd'

Coefficient values:
-3.04017687611051E-06
0.0031266697523144
0.400008836352447
-0.00999952268115858

Confidence interval:
-4.84852291943982E-05 -0.0248717641096469 0.399931839884673  -0.0100046533755882
4.24048754421772E-05   0.0311251036142757 0.400085832820221  -0.00999439198672901

Error info:
           sse: 1.0092e-008
       rsquare: 1.0000
           dfe: 9
    adjrsquare: 1.0000
```

```
rmse: 3.3486e-005
```

The output has five variables as described in the following paragraphs.

1. The first variable is a string of function formula:

$$a * exp(b * x) + c * exp(d * x)$$

2. The second variable is a string of coefficient names. In this function, the coefficient names are a, b, c, and d.

3. The third variable is the value of the coefficient names above:

$$a = -3.04017687611051E - 06$$
$$b = 0.0031266697523144$$
$$c = 0.400008836352447$$
$$d = -0.00999952268115858$$

4. The fourth variable is value of confidence bounds at the confidence level specified by level (level must be between 0 and 1). In this work, we choose the default value of level as 0.95 (95%).

   In this work, the confidence bounds is a matrix in which each column is bound of each coefficient value. The first column is bound for the coefficient a. Second column is bound for coefficient b and so on.

5. The fifth variable is error information of this curve fitting function for the original data.

```
sse          sum of squares due to error
rsquare      coefficient of determination or R^2
dfe          error Degrees of Freedom.
  The DFE is equal to the number of data points minus the number
  of fitted coefficients.
adjrsquare   degree of freedom adjusted R^2
rmse         fit standard error or root mean square error
```

If we solve this problem in MATLAB Command Prompt we will have the following solution.

```
General model Exp2:
    a*exp(b*x) + c*exp(d*x)
Coefficients (with 95% confidence bounds):
  a =  -3.04e-006  (-4.849e-005, 4.24e-005)
  b =    0.003127  (-0.02487, 0.03113)
  c =         0.4  (0.3999, 0.4001)
  d =       -0.01  (-0.01, -0.009994)
```

The following code is used to solve Problem 1 by using the function in MATLAB Curve Fitting Toolbox.

Listing code

```
Public Sub CurveFittingInfo()
  Dim X() As Double = {0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600}

  Dim Y() As Double = {0.4000, 0.2426, 0.1472, 0.0893, 0.0541, 0.0328, 0.0199,  _
        0.0121, 0.0073, 0.0044, 0.0027, 0.0016,  0.0010 }

  Dim mw_X As MWNumericArray = New MWNumericArray(X)
  Dim mw_Y As MWNumericArray = New MWNumericArray(Y)

  Dim mw_ArrayOut()         As MWArray        = Nothing
  Dim mwChar_formula        As MWCharArray    = Nothing
  Dim mwCell_coeffnames     As MWCellArray    = Nothing
  Dim mwDouble_coeffvalues  As MWNumericArray = Nothing
  Dim mwDouble_confint      As MWNumericArray = Nothing
  Dim mwStructure_info      As MWStructArray  = Nothing
  Dim curveFitLib           As String         = "exp2"

  Dim objMatlab As CurveFittingNameSpace.CurveFitting = _
              New CurveFittingNameSpace.CurveFitting()

  mw_X = objMatlab.mytranspose(mw_X)
```

```
mw_Y = objMatlab.mytranspose(mw_Y)

mw_ArrayOut = objMatlab.CurveFittings(5,mw_X, mw_Y, curveFitLib)

mwChar_formula       = CType( mw_ArrayOut(0), MWCharArray   )
mwCell_coeffnames    = CType( mw_ArrayOut(1), MWCellArray   )
mwDouble_coeffvalues = CType( mw_ArrayOut(2), MWNumericArray)
mwDouble_coeffvalues = CType( mw_ArrayOut(2), MWNumericArray)
mwDouble_confint     = CType( mw_ArrayOut(3), MWNumericArray)
mwStructure_info     = CType( mw_ArrayOut(4), MWStructArray )

' This output works as well
'Console.WriteLine(mwChar_formula      )
'Console.WriteLine(mwCell_coeffnames   )
'Console.WriteLine(mwDouble_coeffvalues)
'Console.WriteLine(mwDouble_confint    )
'Console.WriteLine(mwStructure_info    )

Dim str_formula As String  = mwChar_formula.ToString()
Console.WriteLine("Curve fitting formula :")
Console.WriteLine(str_formula)
Console.WriteLine()

Dim str_coeffnames As String = mwCell_coeffnames.ToString()
Console.WriteLine("Coefficient names:")
Console.WriteLine(str_coeffnames)
Console.WriteLine()

Dim db_coeffvalues() As Double = mwDouble_coeffvalues.ToVector(MWArrayComponent.Real)
Console.WriteLine("Coefficient values:")
PrintValues(db_coeffvalues)
Console.WriteLine()

Dim confidentInterval(,) As Double = mwDouble_confint.ToArray(MWArrayComponent.Real)
Console.WriteLine("Confidence interval:")
PrintValues(confidentInterval)
```

```
    Console.WriteLine()

    Console.WriteLine("Error info:")
    Dim str_info As String = mwStructure_info.ToString()
    Console.WriteLine(str_info)

    ' Free memory
    mw_X.Dispose()
    mw_Y.Dispose()

    MWNumericArray.DisposeArray(mw_ArrayOut)

    mwChar_formula.Dispose()
    mwCell_coeffnames.Dispose()
    mwDouble_coeffvalues.Dispose()
    mwDouble_confint.Dispose()

    mwStructure_info.Dispose()
End Sub
```

end code

9.2 Plot Graphics from Curve Fitting Toolbox in VB .NET

Problem 2

input

. There are two data arrays:

```
X = {0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600} ;
Y = {0.4000, 0.2426, 0.1472, 0.0893, 0.0541, 0.0328, 0.0199,
                0.0121, 0.0073, 0.0044, 0.0027, 0.0016,  0.0010 } ;
```

output :

. Find a Curve Fitting polynomial function and its information, with the formula:

$$p1 * x^3 + p2 * x^2 + p3 * x + p4$$

. Plot curves of (X,Y) and curve fitting data

The code of the subroutine, `CurveFittingWithPlots()`, at the end of this chapter is used for this problem. The result output is:

```
Curve fitting formula :
p1*x^3 + p2*x^2 + p3*x + p4

Coefficient names:
    'p1'
    'p2'
    'p3'
    'p4'

Coefficient values:
-5.66993006993008E-09
7.0736063936064E-06
-0.00285847252747253
0.383108791208791

Confidence interval:
-7.23782835968991E-09 5.64027438515616E-06 -0.00321949974326486 0.359112207949607
-4.10203178017025E-09 8.50693840205665E-06 -0.0024974453116802  0.407105374467976

Error info:
           sse: 0.0014
       rsquare: 0.9920
           dfe: 9
    adjrsquare: 0.9893
          rmse: 0.0124
```

The subroutine `CurveFittingWithPlots()` generates two curves. One is a curve of (X,Y) and other is a curve of fitting data shown in the following.

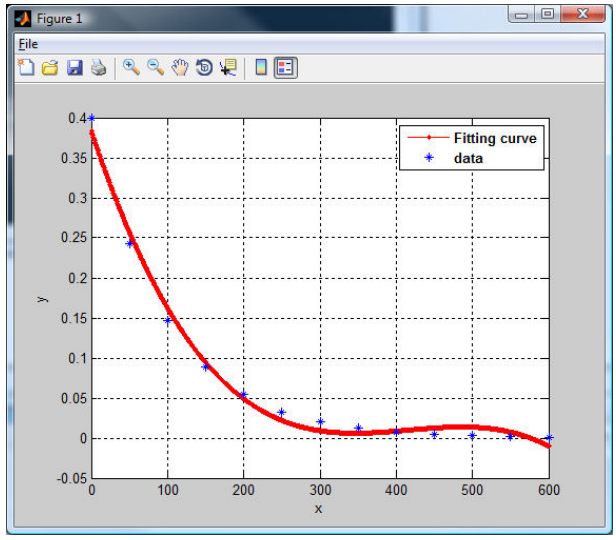


Figure 9.4: Graphics of curve fitting

9.3 Choosing functions in MATLAB Curve Fitting Toolbox in VB .NET functions

In the previous sections, the code we chose were the two functions ***exp2*** for $a * exp(b * x) + c * exp(d * x)$ and ***poly3*** for $p1 * x^3 + p2 * x^2 + p3 * x + p4$. The following are common models of library curves in Curve Fitting Toolbox (these get from command ***cflibhelp*** in MATLAB Commnd Prompt).

1. Polynomial models

```
poly1     Y = p1*x+p2
poly2     Y = p1*x^2+p2*x+p3
poly3     Y = p1*x^3+p2*x^2+...+p4
...
poly9     Y = p1*x^9+p2*x^8+...+p10
```

2. Exponential models

```
exp1      Y = a*exp(b*x)
exp2      Y = a*exp(b*x)+c*exp(d*x)
```

3. Sinusoidal models

```
sin1     Y = a1*sin(b1*x+c1)
sin2     Y = a1*sin(b1*x+c1)+a2*sin(b2*x+c2)
sin3     Y = a1*sin(b1*x+c1)+...+a3*sin(b3*x+c3)
...
sin8     Y = a1*sin(b1*x+c1)+...+a8*sin(b8*x+c8)
```

4. Power models

```
power1    Y = a*x^b
power2    Y = a*x^b+c
```

5. Rational models

```
rat02      Y = (p1)/(x^2+q1*x+q2)
rat21      Y = (p1*x^2+p2*x+p3)/(x+q1)
rat55      Y = (p1*x^5+...+p6)/(x^5+...+q5)
```

6. Gaussian models

```
gauss1     Y = a1*exp(-((x-b1)/c1)^2)
gauss2     Y = a1*exp(-((x-b1)/c1)^2)+a2*exp(-((x-b2)/c2)^2)
gauss3     Y = a1*exp(-((x-b1)/c1)^2)+...+a3*exp(-((x-b3)/c3)^2)
...
gauss8     Y = a1*exp(-((x-b1)/c1)^2)+...+a8*exp(-((x-b8)/c8)^2)
```

7. Fourier models

```
fourier1   Y = a0+a1*cos(x*p)+b1*sin(x*p)
fourier2   Y = a0+a1*cos(x*p)+b1*sin(x*p)+a2*cos(2*x*p)+b2*sin(2*x*p)
fourier3   Y = a0+a1*cos(x*p)+b1*sin(x*p)+...+a3*cos(3*x*p)+b3*sin(3*x*p)
 ...
fourier8   Y = a0+a1*cos(x*p)+b1*sin(x*p)+...+a8*cos(8*x*p)+b8*sin(8*x*p)

where p = 2*pi/(max(xdata)-min(xdata)).
```

9.4 Advance of Setting Plot Graphics from Curve Fitting Toolbox in VB .NET functions

In the code of this chapter, there is a subroutine `CurveFittingPlotsAdvance()` that allows users to set curve properties of title, xlabel, ylabel, curve colors, and legends. Based on this setting, users can write similar functions M-files to create their own way of using graphics in Curve Fitting Toolbox. To know more information of properties of MATLAB , see the MATLAB graphics function `Plot`.

These are some syntax characters for colors of MATLAB graphics that we will use in the following code:

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

The following is the full code of this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports CurveFittingNameSpace

Module Module1

  Sub Main()
    Dim objVB As Example = New Example()

    Console.WriteLine(" Matlab VB .NET for Curve Fititng Toolbox. ")
```

```
  'objVB.CurveFittingInfo()
  'objVB.CurveFittingPlots()
  objVB.CurveFittingPlotsAdvance()

End Sub

Public Class Example

  ' ****************************
  Public Sub CurveFittingInfo()

    Dim X() As Double = {0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600}

    Dim Y() As Double = {0.4000, 0.2426, 0.1472, 0.0893, 0.0541, 0.0328, 0.0199,  _
          0.0121, 0.0073, 0.0044, 0.0027, 0.0016,  0.0010 }

    Dim mw_X As MWNumericArray = New MWNumericArray(X)
    Dim mw_Y As MWNumericArray = New MWNumericArray(Y)

    Dim mw_ArrayOut()         As MWArray        = Nothing
    Dim mwChar_formula        As MWCharArray    = Nothing
    Dim mwCell_coeffnames     As MWCellArray    = Nothing
    Dim mwDouble_coeffvalues  As MWNumericArray = Nothing
    Dim mwDouble_confint      As MWNumericArray = Nothing
    Dim mwStructure_info      As MWStructArray  = Nothing
    Dim curveFitLib           As String         = "exp2"

    Dim objMatlab As CurveFittingNameSpace.CurveFitting = _
                 New CurveFittingNameSpace.CurveFitting()

    mw_X = objMatlab.mytranspose(mw_X)
    mw_Y = objMatlab.mytranspose(mw_Y)

    mw_ArrayOut = objMatlab.CurveFittings(5,mw_X, mw_Y, curveFitLib)
```

```
mwChar_formula       = CType( mw_ArrayOut(0), MWCharArray   )
mwCell_coeffnames    = CType( mw_ArrayOut(1), MWCellArray   )
mwDouble_coeffvalues = CType( mw_ArrayOut(2), MWNumericArray)
mwDouble_coeffvalues = CType( mw_ArrayOut(2), MWNumericArray)
mwDouble_confint     = CType( mw_ArrayOut(3), MWNumericArray)
mwStructure_info     = CType( mw_ArrayOut(4), MWStructArray )

' This output works as well
'Console.WriteLine(mwChar_formula      )
'Console.WriteLine(mwCell_coeffnames   )
'Console.WriteLine(mwDouble_coeffvalues)
'Console.WriteLine(mwDouble_confint    )
'Console.WriteLine(mwStructure_info    )

Dim str_formula As String  = mwChar_formula.ToString()
Console.WriteLine("Curve fitting formula :")
Console.WriteLine(str_formula)
Console.WriteLine()

Dim str_coeffnames As String = mwCell_coeffnames.ToString()
Console.WriteLine("Coefficient names:")
Console.WriteLine(str_coeffnames)
Console.WriteLine()

Dim db_coeffvalues() As Double = mwDouble_coeffvalues.ToVector(MWArrayComponent.Real)
Console.WriteLine("Coefficient values:")
PrintValues(db_coeffvalues)
Console.WriteLine()

Dim confidentInterval(,) As Double = mwDouble_confint.ToArray(MWArrayComponent.Real)
Console.WriteLine("Confidence interval:")
PrintValues(confidentInterval)
Console.WriteLine()

Console.WriteLine("Error info:")
Dim str_info As String = mwStructure_info.ToString()
```

```
  Console.WriteLine(str_info)

  ' Free memory
  mw_X.Dispose()
  mw_Y.Dispose()

  MWNumericArray.DisposeArray(mw_ArrayOut)

  mwChar_formula.Dispose()
  mwCell_coeffnames.Dispose()
  mwDouble_coeffvalues.Dispose()
  mwDouble_confint.Dispose()

  mwStructure_info.Dispose()
End Sub

' ***************************
Public Sub CurveFittingPlots()

  Dim X() As Double = {0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600}

  Dim Y() As Double = {0.4000, 0.2426, 0.1472, 0.0893, 0.0541, 0.0328, 0.0199, _
        0.0121, 0.0073, 0.0044, 0.0027, 0.0016,  0.0010 }

  Dim mw_X As MWNumericArray = New MWNumericArray(X)
  Dim mw_Y As MWNumericArray = New MWNumericArray(Y)

  Dim mw_ArrayOut()         As MWArray          = Nothing
  Dim mwChar_formula        As MWCharArray      = Nothing
  Dim mwCell_coeffnames     As MWCellArray      = Nothing
  Dim mwDouble_coeffvalues  As MWNumericArray   = Nothing
  Dim mwDouble_confint      As MWNumericArray   = Nothing
  Dim mwStructure_info      As MWStructArray    = Nothing
  Dim curveFitLib           As String           = "poly3"

  Dim objMatlab As CurveFittingNameSpace.CurveFitting = _
```

```
            New CurveFittingNameSpace.CurveFitting()

mw_X = objMatlab.mytranspose(mw_X)
mw_Y = objMatlab.mytranspose(mw_Y)

mw_ArrayOut = objMatlab.CurveFittingWithPlots(5,mw_X, mw_Y, curveFitLib)

mwChar_formula       = CType( mw_ArrayOut(0), MWCharArray    )
mwCell_coeffnames    = CType( mw_ArrayOut(1), MWCellArray    )
mwDouble_coeffvalues = CType( mw_ArrayOut(2), MWNumericArray )
mwDouble_confint     = CType( mw_ArrayOut(3), MWNumericArray )
mwStructure_info     = CType( mw_ArrayOut(4), MWStructArray  )

' This output works as well
'
'Console.WriteLine(mwChar_formula      )
'Console.WriteLine(mwCell_coeffnames   )
'Console.WriteLine(mwDouble_coeffvalues)
'Console.WriteLine(mwDouble_confint    )
'Console.WriteLine(mwStructure_info    )


Dim str_formula As String = mwChar_formula.ToString()
Console.WriteLine("Curve fitting formula :")
Console.WriteLine(str_formula)
Console.WriteLine()

Dim str_coeffnames As String = mwCell_coeffnames.ToString()
Console.WriteLine("Coefficient names:")
Console.WriteLine(str_coeffnames)
Console.WriteLine()

Dim db_coeffvalues() As Double = mwDouble_coeffvalues.ToVector(MWArrayComponent.Real)
Console.WriteLine("Coefficient values:")
PrintValues(db_coeffvalues)
Console.WriteLine()
```

```
  Dim confidentInterval(,) As Double = mwDouble_confint.ToArray(MWArrayComponent.Real)
  Console.WriteLine("Confidence interval:")
  PrintValues(confidentInterval)
  Console.WriteLine()

  Console.WriteLine("Error info:")
  Dim str_info As String = mwStructure_info.ToString()
  Console.WriteLine(str_info)

  objMatlab.WaitForFiguresToDie()

  ' Free memory
  mw_X.Dispose()
  mw_Y.Dispose()

  MWNumericArray.DisposeArray(mw_ArrayOut)

  mwChar_formula.Dispose()
  mwCell_coeffnames.Dispose()
  mwDouble_coeffvalues.Dispose()
  mwDouble_confint.Dispose()

  mwStructure_info.Dispose()

End Sub
' ***************************

Public Sub CurveFittingPlotsAdvance()

  Dim X() As Double = {0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600}

  Dim Y() As Double = {0.4000, 0.2426, 0.1472, 0.0893, 0.0541, 0.0328, 0.0199, _
        0.0121, 0.0073, 0.0044, 0.0027, 0.0016,  0.0010 }

  Dim mw_X As MWNumericArray = New MWNumericArray(X)
```

```
Dim mw_Y As MWNumericArray = new MWNumericArray(Y)

Dim mw_ArrayOut()         As MWArray        = Nothing
Dim mwChar_formula        As MWCharArray    = Nothing
Dim mwCell_coeffnames     As MWCellArray    = Nothing
Dim mwDouble_coeffvalues  As MWNumericArray = Nothing
Dim mwDouble_confint      As MWNumericArray = Nothing
Dim mwStructure_info      As MWStructArray  = Nothing
Dim curveFitLib           As String         = "poly3"

Dim objMatlab As CurveFittingNameSpace.CurveFitting = _
             New CurveFittingNameSpace.CurveFitting()

mw_X = objMatlab.mytranspose(mw_X)
mw_Y = objMatlab.mytranspose(mw_Y)

Dim graphicTitle As String = "Curve Fitting Graphics"
Dim xlabel       As String = "Force"
Dim ylabel       As String = "Voltage"

Dim curveFittingColor As String = "r"
Dim curveDataColor    As String = "g"

Dim curveFittingLegend As String = "My curve fitting"
Dim dataLegend         As String = "My data"

mw_ArrayOut = objMatlab.CurveFittingWithPlotsAdvance(5,mw_X, mw_Y, curveFitLib, _
        graphicTitle,xlabel,ylabel, _
        curveFittingColor,curveDataColor,curveFittingLegend,dataLegend)

mwChar_formula       = CType( mw_ArrayOut(0), MWCharArray     )
mwCell_coeffnames    = CType( mw_ArrayOut(1), MWCellArray     )
mwDouble_coeffvalues = CType( mw_ArrayOut(2), MWNumericArray  )
mwDouble_confint     = CType( mw_ArrayOut(3), MWNumericArray  )
mwStructure_info     = CType( mw_ArrayOut(4), MWStructArray   )
```

```
' This output works as well

'Console.WriteLine(mwChar_formula      )
'Console.WriteLine(mwCell_coeffnames   )
'Console.WriteLine(mwDouble_coeffvalues)
'Console.WriteLine(mwDouble_confint    )
'Console.WriteLine(mwStructure_info    )

Dim str_formula As String = mwChar_formula.ToString()
Console.WriteLine("Curve fitting formula :")
Console.WriteLine(str_formula)
Console.WriteLine()

Dim str_coeffnames As String = mwCell_coeffnames.ToString()
Console.WriteLine("Coefficient names:")
Console.WriteLine(str_coeffnames)
Console.WriteLine()

Dim db_coeffvalues() As Double = mwDouble_coeffvalues.ToVector( _
                                              MWArrayComponent.Real)
Console.WriteLine("Coefficient values:")
PrintValues(db_coeffvalues)
Console.WriteLine()

Dim confidentInterval(,) As Double = mwDouble_confint.ToArray( _
                                              MWArrayComponent.Real)
Console.WriteLine("Confidence interval:")
PrintValues(confidentInterval)
Console.WriteLine()

Console.WriteLine("Error info:")
Dim str_info As String = mwStructure_info.ToString()
Console.WriteLine(str_info)
objMatlab.WaitForFiguresToDie()

' Free memory
```

```
    mw_X.Dispose()
    mw_Y.Dispose()

    MWNumericArray.DisposeArray(mw_ArrayOut)

    mwChar_formula.Dispose()
    mwCell_coeffnames.Dispose()
    mwDouble_coeffvalues.Dispose()
    mwDouble_confint.Dispose()

    mwStructure_info.Dispose()

  End Sub

  ' ******************************
  Public Sub PrintValues(ByVal myArr As Array)
    Dim myEnumerator As System.Collections.IEnumerator = _
       myArr.GetEnumerator()
    Dim i As Integer = 0
    Dim cols As Integer = myArr.GetLength(myArr.Rank - 1)

    'for row vector or column vector
    If myArr.Rank = 1 Then

      While myEnumerator.MoveNext()
        Console.WriteLine(ControlChars.Tab + "{0}", myEnumerator.Current)
      End While

    Else
      'for other
      While myEnumerator.MoveNext()
        If i < cols Then
          i += 1
        Else
          Console.WriteLine()
          i = 1
```

```
          End If
          Console.Write(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While
      End If

      Console.WriteLine()
    End Sub

  End Class

End Module
```

end code

Chapter 10

Roots of Equations

In this chapter we will generate a class ***FindingRoots*** from common M-files to find roots of a polynomial function and a nonlinear function. The generated functions of this library will be used in a VB .Net 2008 project to find the roots of functions. The procedure to create the class ***FindingRoots*** is the same as the procedure in Chapter 2.

We will write the M-files as shown below. These functions will be used to generate the class ***FindingRoots***.

`myfzero.m and myroots.m`

myroots.m

```
function r = (c)
r = roots(c) ;
```

myfzero.m

```
function x = myfzero(strfunc, x0)

F = inline(strfunc) ;
x = fzero(F, x0) ;
```

The following procedure to create the class ***FindingRoots*** is the same as the procedure in

Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `FindingRootsNameSpace.dll` that contains the class ***FindingRoots*** (see Fig.10.1 and Fig.10.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `FindingRootsNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `FindingRootsNameSpace.dll`, and then click OK (see Fig. 10.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

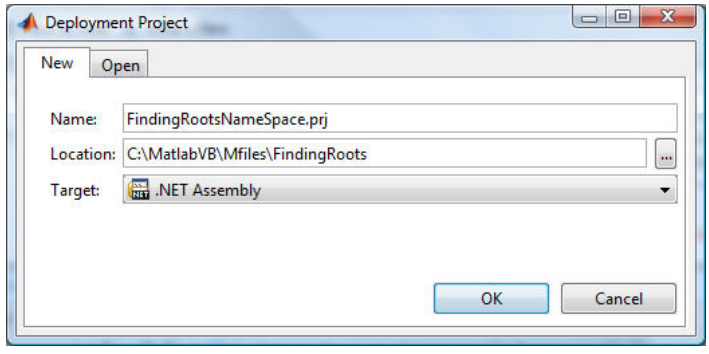


Figure 10.1: Deployment project for ***FindingRoots***

The following sections show how to use the functions in the class ***FindingRoots*** to solve common computation problems. The full code is at the end of this chapter.

10.1 Roots of Polynomials

This section describes how to use the functions in the generated class to find the roots of a polynomial function.

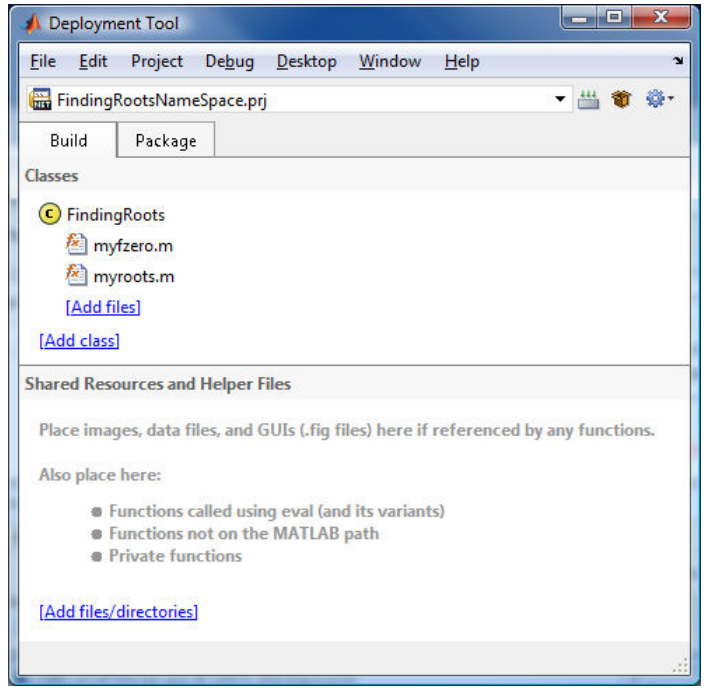


Figure 10.2: Adding M-files in deployment project for ***FindingRoots***

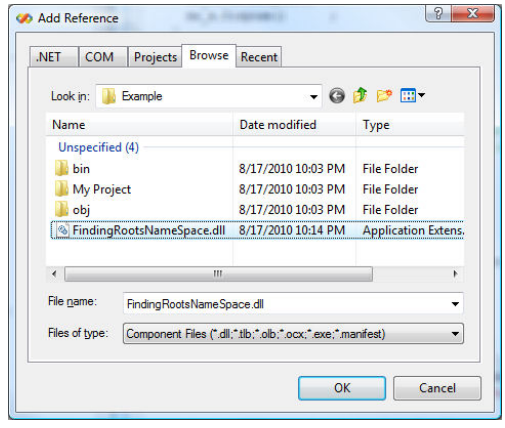


Figure 10.3: Add the reference `FindingRootsNameSpace.dll` in the VB project

In this section, we will use MATLAB functions to solve the following problems.

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:

$$c_1 = -1\ , \quad c_2 = 7.2\ , \quad c_3 = -21\ , \text{and} \quad c_4 = -5$$

The following is the code to solve Problem 1 by using the function `myroots(..)`. See the MATLAB manual [2] for more information on the MATLAB function `roots(..)`.

Listing code

```
Public Sub FindingRootsPolynormial()
  '
  'Find the solutions of polynomial function:
  '   f(x) = -x^3 + 7.2x^2 -21x -5
  '
  Dim order As Integer = 3
  Dim db_coefs() As Double = New Double(){ -1, 7.2, -21, -5 }

  ' declare mxArray variables
  Dim mw_coefs As MWNumericArray = New MWNumericArray(db_coefs)
  Dim mw_x     As MWNumericArray = Nothing

  ' call an implemental function
```

```
Dim obj As FindingRootsNameSpace.FindingRoots = _
              New FindingRootsNameSpace.FindingRoots()

mw_x = obj.myroots(mw_coefs)

Console.WriteLine("Solutions of the polynomial function:")
Console.WriteLine("{0}", mw_x)
Console.WriteLine()

' convert back to double
Dim db_xReal(order-1) As Double
Dim db_xImag(order-1) As Double

db_xReal = mw_x.ToVector(MWArrayComponent.Real)

Try
  db_xImag = mw_x.ToVector(MWArrayComponent.Imaginary)

Catch
  'nothing, db_xImag will be assigned value of zero
End Try

' or
Console.WriteLine("Solutions of the polynomial function : ")
Dim i As Integer
for i=0 to (order-1)

Console.Write( db_xReal(i).ToString() +  " + " )
Console.Write( db_xImag(i).ToString() +  "i" + ControlChars.Tab)
Console.WriteLine()
Next

' free memory */
mw_coefs.Dispose()
mw_x.Dispose()
```

```
    End Sub
```

end code

Note

The roots of a polynomial function may have imaginary terms. Therefore we need to use another variable (say db_xImag) to receive the imaginary value of a MWNumericArray variable. Or we can use the property `IsComplex` (for example `mw_x.IsComplex`) in this case.

10.2 The Root of a Nonlinear-Equation

This section describes how to use the function `myfzero(..)` in the generated ***FindingRoot*** class 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 [4].

Problem 2

Find the root of the function:

$$f(x) = \sin(2x) + \cos(x) + 1$$

The following is the code to solve Problem 2 by using the function `myfzero(..)`.

Listing code

```
    Public Sub 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

      Dim db_initialGuess As Double = 0.9
      Dim strfunc         As String = "sin(2*x) + cos(x) + 1"

      ' declare mxArray variables
```

```
    Dim mw_strfunc As MWCharArray = New MWCharArray(strfunc)
    Dim mw_x As MWNumericArray = Nothing

    ' call an implemental function
    Dim obj As FindingRootsNameSpace.FindingRoots = _
                  New FindingRootsNameSpace.FindingRoots()

    mw_x = obj.myfzero(mw_strfunc, db_initialGuess)

    Console.WriteLine("Solutions of the given function :")
    Console.WriteLine("{0}", mw_x )

    ' or
    ' convert back to double
    Dim db_xReal As Double = mw_x.ToScalarDouble()

    Console.WriteLine("Solutions of the given function : " )
    Console.WriteLine( db_xReal.ToString() )

    ' free memory */
    mw_strfunc.Dispose()
    mw_x.Dispose()

End Sub
```

end code

Note

The generated function `myfzero(..)` is a function that has an argument as an expression string. The form of this expression string follows the rule of a MATLAB expression string.

The following is full code for this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports FindingRootsNameSpace
```

```
Module Module1

  Sub Main()

    Dim objVB As Example = New Example()
    Console.WriteLine("Roots of functions.")
    objVB.FindingRootsPolynormial()
    Console.WriteLine()
    Console.WriteLine()
    objVB.FindingZeroFunction()

  End Sub

  Public Class Example

  ' ************************** */
  Public Sub FindingRootsPolynormial()
   '
   'Find the solutions of polynomial function:
   ' f(x) = -x^3 + 7.2x^2 -21x -5
   '
   Dim order As Integer = 3
   Dim db_coefs() As Double = New Double(){ -1, 7.2, -21, -5 }

   ' declare mxArray variables
   Dim mw_coefs As MWNumericArray = New MWNumericArray(db_coefs)
   Dim mw_x     As MWNumericArray = Nothing

   ' call an implemental function
   Dim obj As FindingRootsNameSpace.FindingRoots = _
           New FindingRootsNameSpace.FindingRoots()

   mw_x = obj.myroots(mw_coefs)

   Console.WriteLine("Solutions of the polynomial function:")
```

```
 Console.WriteLine("{0}", mw_x)
 Console.WriteLine()
 ' convert back to double
 Dim db_xReal(order-1) As Double
 Dim db_xImag(order-1) As Double

 db_xReal = mw_x.ToVector(MWArrayComponent.Real)

 Try
   db_xImag = mw_x.ToVector(MWArrayComponent.Imaginary)

 Catch
   'nothing, db_xImag will be assigned value of zero
 End Try

 ' or
 Console.WriteLine("Solutions of the polynomial function : ")
 Dim i As Integer
 for i=0 to (order-1)
      Console.Write( db_xReal(i).ToString() +  " + " )
      Console.Write( db_xImag(i).ToString() +  "i" + ControlChars.Tab)
      Console.WriteLine()
    Next

    ' free memory */
 mw_coefs.Dispose()
 mw_x.Dispose()

End Sub

  ' ****************************
Public Sub 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

    Dim db_initialGuess As Double = 0.9
    Dim strfunc      As String = "sin(2*x) + cos(x) + 1"

    ' declare mxArray variables
    Dim mw_strfunc As MWCharArray = New MWCharArray(strfunc)
    Dim mw_x As MWNumericArray = Nothing

    ' call an implemental function
    Dim obj As FindingRootsNameSpace.FindingRoots = _
            New FindingRootsNameSpace.FindingRoots()

    mw_x = obj.myfzero(mw_strfunc, db_initialGuess)

    Console.WriteLine("Solutions of the given function :")
    Console.WriteLine("{0}", mw_x )

    ' or convert back to double
    Dim db_xReal As Double = mw_x.ToScalarDouble()

    Console.WriteLine("Solutions of the given function : " )
    Console.WriteLine( db_xReal.ToString() )

    ' free memory */
    mw_strfunc.Dispose()
    mw_x.Dispose()

  End Sub

  End Class

End Module
```

end code

Chapter 11

Fast Fourier Transform

The 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:

$$F(s) = \int_{-\infty}^{+\infty} f(x) e^{-i(2\pi s)x} dx \tag{11.1a}$$

and its inverse

$$f(x) = \int_{-\infty}^{+\infty} F(s) e^{\ i(2\pi x)s} ds \tag{11.1b}$$

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 a class in the following sections.

In this chapter we will generate a class ***FFT*** from common M-files working on FFT problems. The generated functions of this class will be used in a VB .Net 2008 project to solve the FFT problems.

We will write the M-files as shown below. These functions will be used to generate the class ***FFT***.

`myfft.m, myifft.m, myfft2.m, and myifft2.m`

myfft.m

```
function Y = myfft(X)

Y = fft(X) ;
```

myifft.m

```
function Y = myifft(X)

Y = ifft(X) ;
```

myfft2.m

```
function Y = myfft2(X)

Y = fft2(X) ;
```

myifft2.m

```
function Y = myifft2(X)

Y = ifft2(X) ;
```

The following procedure to create the class ***FFT*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `FFTNameSpace.dll` that contains the class ***FFT*** (see Fig.11.1 and Fig.11.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `FFTNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `FFTNameSpace.dll`, and then click OK (see Fig. 11.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

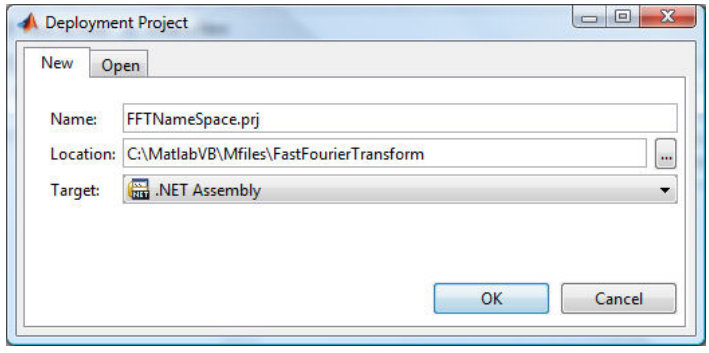


Figure 11.1: Deployment project for ***FFT***

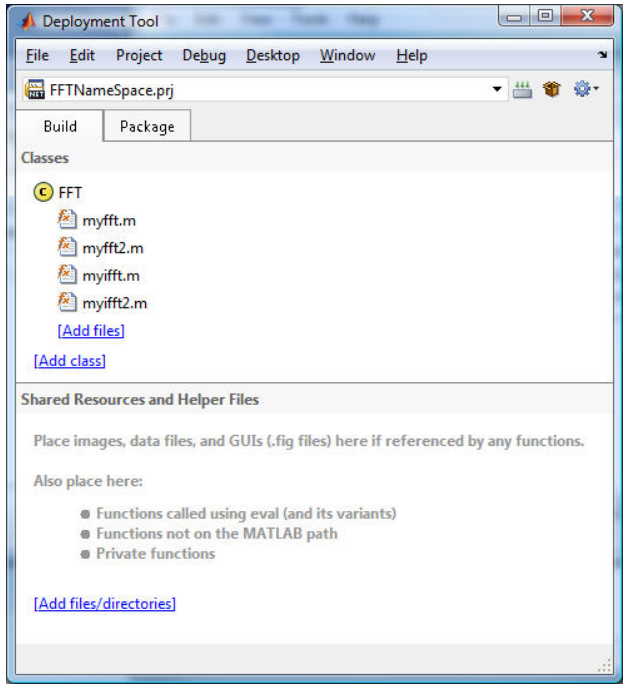


Figure 11.2: Adding M-files in deployment project for ***FFT***

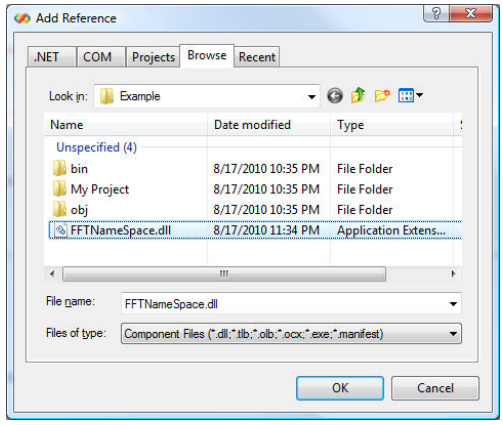


Figure 11.3: Add the reference `FFTNameSpace.dll` in the VB project

The following sections show how to use the functions in the class ***FFT*** to solve common computation problems. The full code is at the end of this chapter.

11.1 One-Dimensional Fast Fourier Transform

The MATLAB functions implement FFT and its inverse for a vector $\mathbf{X}$ with length N as follows:

$$\mathbf{Y}_k = \sum_{n=1}^{N} \mathbf{X}_n \, e^{(-i2\pi)\left(\frac{n-1}{N}\right)(k-1)} \qquad 1 \le k \le N \tag{11.2a}$$

and its inverse

$$\mathbf{X}_n = \frac{1}{N} \sum_{k=1}^{N} \mathbf{Y}_k \, e^{(i2\pi)\left(\frac{k-1}{N}\right)(n-1)} \qquad 1 \le n \le N \tag{11.2b}$$

Remarks

1. The vectors $\mathbf{X}$ and $\mathbf{Y}$ in equation 11.2 are represented by functions $f(x)$ and $F(s)$ in equation 11.1, respectively.

2. The first index of a vector in MATLAB starts at 1, therefore in the equation 11.2 we have term $(n-1)$ and $(k-1)$. This produces identical results as the traditional Fourier equations from 0 to $(N-1)$.

In this section, we will use MATLAB functions to solve the following problems.

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. 11.2a

The following is the code to solve Problem 1 by using the function `myfft(..)`.

Listing code

```
Public Sub FastFourierTrans1D()

  Dim db_X() As Double = New Double() { 6, 3, 7, -9, 0, 3, -2, 1 }
  Dim vectorSize As Integer = 8

  ' declare mxArray variables

  Dim mw_X As MWNumericArray = New MWNumericArray(db_X)
  Dim mw_Y As MWNumericArray = Nothing

   ' call an implemental function
  Dim obj As FFTNameSpace.FFT = New FFTNameSpace.FFT()

  mw_Y = obj.myfft(mw_X)
  Console.WriteLine("Fast Fourier Transform of X :")
  Console.WriteLine("{0}", mw_Y)

  ' convert back to double
  Dim db_YReal(vectorSize-1) As Double
  Dim db_YImag(vectorSize-1) As Double

  db_YReal = mw_Y.ToVector(MWArrayComponent.Real)

  Try
    db_YImag = mw_Y.ToVector(MWArrayComponent.Imaginary)
```

```
  Catch
    ' nothing, db_YImag() will be assigned value of zero
  End Try

  'or print out
  Console.WriteLine("Fast Fourier Transform of X " + ControlChars.NewLine)
  Dim i As Integer

  for i=0 to (vectorSize-1)

    Console.Write( db_YReal(i).ToString() +  " + " )
    Console.Write( db_YImag(i).ToString() +  "i"  + ControlChars.NewLine )

  Next

  Console.WriteLine()

  mw_X.Dispose()
  mw_Y.Dispose()

End Sub
```

end code

Problem 1B

input . A vector $\mathbf{Y}$,

$\mathbf{Y}$ real numbers = {9.00, 13.0711, 1.0, -1.0711, 13.0, -1.0711 , 1.0 , 13.0711 }

$\mathbf{Y}$ imaginary numbers = {0, -1.9289, -14.0, 16.0711, 0, -16.0711, 14.0, 1.9289 }

output . Finding an inverse FFT vector $\mathbf{X}$ in Eq. 11.2b

The following is the code to solve Problem 1B by using the function `myifft(..)`.

Listing code

```
Public Sub InverseFastFourierTrans1D()
```

```
Dim db_YReal() As Double = New Double() _
          { 9.00, 13.0711,   1.00, -1.0711, 13.00, -1.0711 , 1.00 , 13.0711 }
Dim db_YImag() As Double = New Double() _
          { 0   , -1.9289, -14.00, 16.0711, 0    , -16.0711, 14.00, 1.9289  }

Dim vectorSize As Integer = 8

' declare mxArray variables
Dim mw_Y As MWNumericArray = New MWNumericArray(db_YReal, db_YImag)
Dim mw_X As MWNumericArray = New MWNumericArray (MWArrayComplexity.Complex, _
                                                  MWNumericType.Double, 7)

' call an implemental function
Dim obj As FFTNameSpace.FFT = New FFTNameSpace.FFT()

mw_X = obj.myifft(mw_Y)
Console.WriteLine("Inverse Fast Fourier Transform of Y :")
Console.WriteLine("{0}", mw_X )
Console.WriteLine()

' convert back to double
Dim db_XReal() As Double = mw_X.ToVector(MWArrayComponent.Real)
Dim db_XImag(vectorSize - 1) As Double

Try
  db_XImag = mw_X.ToVector(MWArrayComponent.Imaginary)
Catch
  ' nothing, db_XImag() will be assigned value of zero
End Try

'or, print out
Console.WriteLine("Inverse Fast Fourier Transform of Y : " )

Dim i As Integer
for i=0 to (vectorSize - 1)
```

```
      Console.Write( db_XReal(i).ToString() +  " + " )
      Console.Write( db_XImag(i).ToString() +  "i" + ControlChars.NewLine )
  Next

  Console.WriteLine()

  mw_X.Dispose()
  mw_Y.Dispose()

End Sub
```

end code

11.2 Two-Dimensional Fast Fourier Transform

The MATLAB function `fft2(..)` ($\mathbf{Y} = fft2(\mathbf{X})$) 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 $\mathbf{Y} = fft2(\mathbf{X}, m, n)$, refer to the MATLAB manual [4].

In this section, we will use MATLAB functions to solve the following problems.

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 $\mathbf{X}$

The following is the code to solve Problem 2 by using the function `myfft2(..)` in the generated class ***FFT***.

Listing code

```
Public Sub FastFourierTrans2D()
```

```
Dim X(,) As Double = New Double(,)  _
        { {4  ,  3.2,  6.8,  9.1  }  , _
          {-4 ,  1.2,  4.3,  5.4  }  , _
          {2.2, -6.7,  8  ,  12.2 }  }
Dim row As Integer = 3
Dim col As Integer = 4

Dim i, j As Integer

' declare mxArray variables
Dim mw_X As MWNumericArray = New MWNumericArray(X)
Dim mw_Y As MWNumericArray = Nothing

' call an implemental function
Dim obj As FFTNameSpace.FFT = New FFTNameSpace.FFT()

mw_Y = obj.myfft2(mw_X)
  Console.WriteLine("2-D Fast Fourier Transform of X:" )
Console.WriteLine("{0}", mw_Y)
Console.WriteLine()

' convert back to double
Dim db_YReal(row-1,col-1) As Double
Dim db_YImag(row-1,col-1) As Double

db_YReal = mw_Y.ToArray(MWArrayComponent.Real)

Try
  db_YImag = mw_Y.ToArray(MWArrayComponent.Imaginary)
Catch
    ' nothing, db_YImag will be assigned value of zero
End Try

' or, print out
Console.WriteLine("2-D Fast Fourier Transform of X : ")
```

```
  for i=0 to (row-1)

    for j=0 to (col-1)

      Console.Write( db_YReal.GetValue(i,j).ToString() + " + " )
      Console.Write( db_YImag.GetValue(i,j).ToString() + "i"   )
      Console.Write(ControlChars.Tab + ControlChars.Tab)

    Next

    Console.WriteLine()

  Next
  Console.WriteLine()

  ' free memory
  mw_X.Dispose()
  mw_Y.Dispose()

End Sub
```

end code

Problem 2B

input . Matrix $\mathbf{Y}$,

$$\mathbf{Y} = \begin{bmatrix} (45.70 + 0i) & (-16.9000 + 29.0000i) & (-3.10 + 0i) & (-16.9000 - 29.0000i) \\ (11.80 + 7.621i) & (-8.4806 - 3.4849i) & (-0.70 + 9.5263i) & (16.9806 + 7.8151i) \\ (11.80 - 7.621i) & (16.9806 - 7.8151i) & (-0.70 - 9.5263i) & (-8.4806 + 3.4849i) \end{bmatrix}$$

output . Finding the matrix $\mathbf{X}$ which is an inverse FFT matrix of $\mathbf{Y}$.

The following is the code to solve Problem 2B by using the function `myifft2(..)`.

Listing code

```
Public Sub InverseFastFourierTrans2D()

  Dim YReal(,) As Double = New Double (,) _
          { { 45.7000,  -16.9000,  -3.1000,  -16.9000 } , _
            { 11.8000,   -8.4806,  -0.7000,   16.9806 } , _
            { 11.8000,   16.9806,  -0.7000,  -8.4806  } }

  Dim YImag(,) As Double = New Double(,) _
           {{ 0,       29.0000,   0     , -29.0000 }    , _
            { 7.6210,  -3.4849,   9.5263,   7.8151 }    , _
            {-7.6210,  -7.8151,  -9.5263,   3.4849 }    }

  Dim row As Integer = 3
  Dim col As Integer = 4

  Dim i, j As Integer

  ' declare mxArray variables
  Dim mw_X As MWNumericArray = Nothing
  Dim mw_Y As MWNumericArray = New MWNumericArray(YReal, YImag)

  ' call an implemental function
  Dim obj As FFTNameSpace.FFT = New FFTNameSpace.FFT()

  mw_X = obj.myifft2(mw_Y)
  Console.WriteLine("Inverse 2-D Fast Fourier Transform of Y :")
  Console.WriteLine("{0}", mw_X )

  ' convert back to double
  Dim db_XReal(row-1, col-1) As Double
  Dim db_XImag(row-1, col-1) As Double

  db_XReal = mw_X.ToArray(MWArrayComponent.Real)

  Try
    db_XImag = mw_X.ToArray(MWArrayComponent.Imaginary)
```

```
  Catch
    ' nothing, db_XImag will be assigned value of zero
  End Try

 'or print out
  Console.WriteLine()
  Console.WriteLine("Inverse 2-D Fast Fourier Transform of Y : " )

  for i=0 to (row-1)

    for j=0 to (col-1)

      Console.Write( db_XReal.GetValue(i,j).ToString() +  " + " )
      Console.Write( db_XImag.GetValue(i,j).ToString() +  "i"   )
      Console.Write(ControlChars.Tab + ControlChars.Tab)
    Next
    Console.WriteLine()

  Next
  Console.WriteLine()

  ' free memory
  mw_X.Dispose()
  mw_Y.Dispose()

End Sub
```

end code

The following is full code for this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports FFTNameSpace
```

```
Module Module1

  Sub Main()

    Dim objVB As Example = New Example()

    Console.WriteLine("Fast Fourier Transform.")
    Console.WriteLine("Fast Fourier Transform in 1D.")

    objVB.FastFourierTrans1D()
    Console.WriteLine()

    Console.WriteLine("Fast Fourier Transform in inverse of 1D.")
    objVB.InverseFastFourierTrans1D()
    Console.WriteLine()

    Console.WriteLine("Fast Fourier Transform in 2D.")
    objVB.FastFourierTrans2D()
    Console.WriteLine()

    Console.WriteLine("Fast Fourier Transform in inverse of 2D.")
    objVB.InverseFastFourierTrans2D()

  End Sub

  Public Class Example

   ' ****************************
  Public Sub FastFourierTrans1D()

      Dim db_X() As Double = New Double() { 6, 3, 7, -9, 0, 3, -2, 1 }
      Dim vectorSize As Integer = 8

      ' declare mxArray variables
```

```
Dim mw_X As MWNumericArray = New MWNumericArray(db_X)
Dim mw_Y As MWNumericArray = Nothing

' call an implemental function
Dim obj As FFTNameSpace.FFT = New FFTNameSpace.FFT()

mw_Y = obj.myfft(mw_X)
Console.WriteLine("Fast Fourier Transform of X :")
Console.WriteLine("{0}", mw_Y)

' convert back to double
Dim db_YReal(vectorSize-1) As Double
Dim db_YImag(vectorSize-1) As Double

db_YReal = mw_Y.ToVector(MWArrayComponent.Real)

Try
  db_YImag = mw_Y.ToVector(MWArrayComponent.Imaginary)
Catch
  ' nothing, db_YImag() will be assigned value of zero
End Try

'or print out
Console.WriteLine("Fast Fourier Transform of X " + ControlChars.NewLine)
Dim i As Integer

for i=0 to (vectorSize-1)
  Console.Write( db_YReal(i).ToString() +  " + " )
  Console.Write( db_YImag(i).ToString() +  "i"  + ControlChars.NewLine )
Next

Console.WriteLine()

mw_X.Dispose()
mw_Y.Dispose()
```

```
End Sub

'***************************
Public Sub InverseFastFourierTrans1D()

  Dim db_YReal() As Double = New Double() _
          { 9.00, 13.0711,   1.00, -1.0711, 13.00, -1.0711 , 1.00 , 13.0711 }
  Dim db_YImag() As Double = New Double() _
          { 0   , -1.9289, -14.00, 16.0711, 0    , -16.0711, 14.00, 1.9289  }

  Dim vectorSize As Integer = 8

  ' declare mxArray variables
  Dim mw_Y As MWNumericArray = New MWNumericArray(db_YReal, db_YImag)
  Dim mw_X As MWNumericArray = New MWNumericArray (MWArrayComplexity.Complex, _
                                                   MWNumericType.Double, 7)

  ' call an implemental function
  Dim obj As FFTNameSpace.FFT = New FFTNameSpace.FFT()

  mw_X = obj.myifft(mw_Y)
  Console.WriteLine("Inverse Fast Fourier Transform of Y :")
  Console.WriteLine("{0}", mw_X )
  Console.WriteLine()

  ' convert back to double
  Dim db_XReal() As Double = mw_X.ToVector(MWArrayComponent.Real)
  Dim db_XImag(vectorSize - 1) As Double

  Try
    db_XImag = mw_X.ToVector(MWArrayComponent.Imaginary)
  Catch
    ' nothing, db_XImag() will be assigned value of zero
  End Try
```

```
    'or, print out
    Console.WriteLine("Inverse Fast Fourier Transform of Y : " )

    Dim i As Integer
    for i=0 to (vectorSize - 1)
      Console.Write( db_XReal(i).ToString() +  " + " )
      Console.Write( db_XImag(i).ToString() +  "i" + ControlChars.NewLine )
    Next

    Console.WriteLine()

    mw_X.Dispose()
    mw_Y.Dispose()

  End Sub

  ' ***************************
  Public Sub FastFourierTrans2D()

    Dim X(,) As Double = New Double(,)  _
            { {4  ,  3.2,  6.8,  9.1  }  , _
            {-4 ,  1.2,  4.3,  5.4  }  , _
            {2.2, -6.7,  8  ,  12.2 }  }
    Dim row As Integer = 3
    Dim col As Integer = 4

    Dim i, j As Integer

    ' declare mxArray variables
    Dim mw_X As MWNumericArray = New MWNumericArray(X)
    Dim mw_Y As MWNumericArray = Nothing

    ' call an implemental function
    Dim obj As FFTNameSpace.FFT = New FFTNameSpace.FFT()

    mw_Y = obj.myfft2(mw_X)
```

```
Console.WriteLine("2-D Fast Fourier Transform of X:" )
Console.WriteLine("{0}", mw_Y)
Console.WriteLine()

' convert back to double
Dim db_YReal(row-1,col-1) As Double
Dim db_YImag(row-1,col-1) As Double

db_YReal = mw_Y.ToArray(MWArrayComponent.Real)

Try
  db_YImag = mw_Y.ToArray(MWArrayComponent.Imaginary)
Catch
  ' nothing, db_YImag will be assigned value of zero
End Try

' or, print out
Console.WriteLine("2-D Fast Fourier Transform of X : ")

for i=0 to (row-1)

  for j=0 to (col-1)

    Console.Write( db_YReal.GetValue(i,j).ToString() + " + " )
    Console.Write( db_YImag.GetValue(i,j).ToString() + "i"   )
    Console.Write(ControlChars.Tab + ControlChars.Tab)

  Next

  Console.WriteLine()

Next
  Console.WriteLine()

  ' free memory
  mw_X.Dispose()
```

```
        mw_Y.Dispose()

    End Sub

  ' ****************************
    Public Sub InverseFastFourierTrans2D()

      Dim YReal(,) As Double = New Double (,) _
              { { 45.7000,  -16.9000,  -3.1000,  -16.9000 } , _
              { 11.8000,   -8.4806,  -0.7000,   16.9806 } , _
              { 11.8000,   16.9806,  -0.7000,  -8.4806  } }

      Dim YImag(,) As Double = New Double(,) _
               {{ 0,       29.0000,   0  , -29.0000 } , _
              { 7.6210,  -3.4849,   9.5263,   7.8151 } , _
              {-7.6210,  -7.8151,  -9.5263,   3.4849 } }

      Dim row As Integer = 3
      Dim col As Integer = 4

      Dim i, j As Integer

      ' declare mxArray variables
      Dim mw_X As MWNumericArray = Nothing
      Dim mw_Y As MWNumericArray = New MWNumericArray(YReal, YImag)

      ' call an implemental function
      Dim obj As FFTNameSpace.FFT = New FFTNameSpace.FFT()

      mw_X = obj.myifft2(mw_Y)
      Console.WriteLine("Inverse 2-D Fast Fourier Transform of Y :")
      Console.WriteLine("{0}", mw_X )

      ' convert back to double
      Dim db_XReal(row-1, col-1) As Double
      Dim db_XImag(row-1, col-1) As Double
```

```
      db_XReal = mw_X.ToArray(MWArrayComponent.Real)

      Try
        db_XImag = mw_X.ToArray(MWArrayComponent.Imaginary)
      Catch
        ' nothing, db_XImag will be assigned value of zero
      End Try

      'or print out
      Console.WriteLine()
      Console.WriteLine("Inverse 2-D Fast Fourier Transform of Y : " )

      for i=0 to (row-1)

        for j=0 to (col-1)

          Console.Write( db_XReal.GetValue(i,j).ToString() +  " + " )
          Console.Write( db_XImag.GetValue(i,j).ToString() +  "i"   )
          Console.Write(ControlChars.Tab + ControlChars.Tab)
        Next
      Console.WriteLine()

      Next
      Console.WriteLine()

      ' free memory
      mw_X.Dispose()
      mw_Y.Dispose()

    End Sub

  End Class

End Module
```

end code

Chapter 12

Eigenvalues and Eigenvectors

In this chapter we will generate a class ***Eigen*** from common M-files working on problems of eigenvectors and eigenvalues. The generated functions of this class will be used in a VB .Net 2008 project to solve the eigen problems. The procedure to create the class ***Eigen*** is the same as the procedure in Chapter 2.

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 [3].

We will write the M-file myeig.m as shown below. This function will be used to generate the class ***Eigen***.

myeig.m

```
function [V,D] = myeig(A)

[V,D] = eig(A) ;
```

The following procedure to create the class ***Eigensystem*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `EigensystemNameSpace.dll` that contains the class ***Eigensystem*** (see Fig.12.1 and Fig.12.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `EigensystemNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `EigensystemNameSpace.dll`, and then click OK (see Fig. 12.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

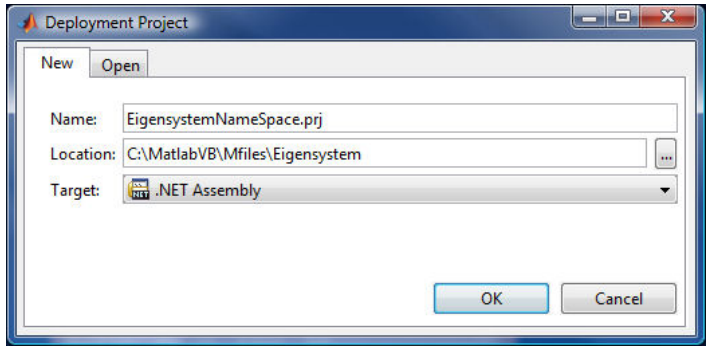


Figure 12.1: Deployment project for ***Eigensystem***

The following sections show how to use the functions in the class ***Eigensystem*** to solve common computation problems. The full code is at the end of this chapter.

12.1 Eigenvalues and Eigenvectors

In this section, we will use MATLAB functions to solve the following problems.

Problem 1

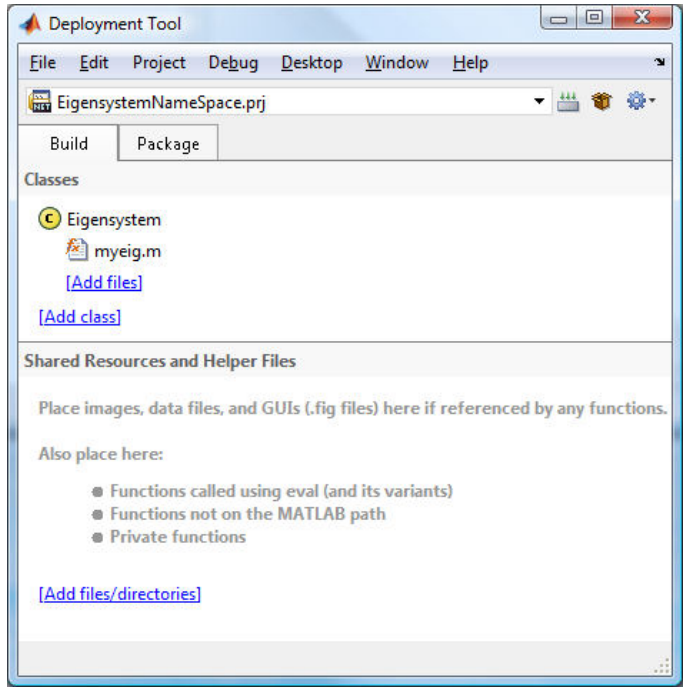


Figure 12.2: Adding M-files in deployment project for ***Eigensystem***

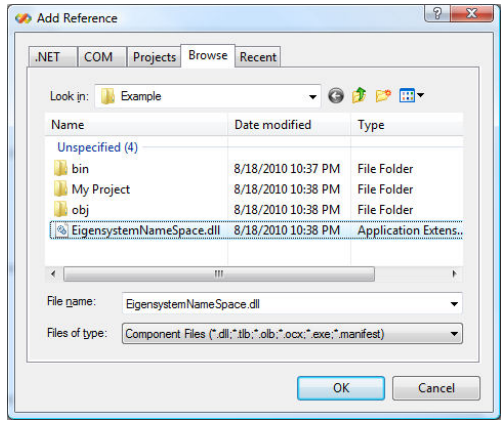


Figure 12.3: Add the reference `EigensystemNameSpace.dll` in the VB project

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**

The following is the code to solve Problem 1 by using function `myeig(..)` in the generated ***Eigen*** class to find eigenvalues and eigenvectors of a square matrix.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays



Module Module1

    Sub Main()

   Dim objVB As Example = New Example()
   Console.WriteLine()

   Console.WriteLine("Eigenvalues and eigenvectors" )
   objVB.EigValueVector()

    End Sub

  Public Class Example

  ' ***************************
  Public Sub EigValueVector()

   Dim A(,) As Double = New Double(,) _
        {{ 0, -6, -1} , {6, 2, -16} ,  {-5, 20, -10} }
```

```
Dim row As Integer = 3
Dim col As Integer = 3
Dim i, j As Integer

' declare mx_Array variables '
Dim mw_A       As MWNumericArray = New MWNumericArray(A)
Dim mw_ArrayOut() As MWArray     = Nothing

Dim mw_eigenvectors As MWNumericArray = Nothing
Dim mw_eigenvalues  As MWNumericArray = Nothing

' call an implemental function
Dim obj  As EigensystemNameSpace.Eigensystem = _
        New EigensystemNameSpace.Eigensystem()

mw_ArrayOut = obj.myeig(2, mw_A)
Console.WriteLine("Eigen of the matrix A : ")
Console.WriteLine("{0}", mw_ArrayOut)

mw_eigenvectors = mw_ArrayOut(0)
mw_eigenvalues  = mw_ArrayOut(1)

' convert back to double
Dim db_eigenvectorsReal(row-1, col-1) As Double
Dim db_eigenvectorsImag(row-1, col-1) As Double

Dim db_eigenvaluesReal(row-1, col-1) As Double
Dim db_eigenvaluesImag(row-1, col-1) As Double

db_eigenvectorsReal = mw_eigenvectors.ToArray(MWArrayComponent.Real)
db_eigenvaluesReal  = mw_eigenvalues.ToArray(MWArrayComponent.Real)

Try
  db_eigenvectorsImag = mw_eigenvectors.ToArray(MWArrayComponent.Imaginary)
  db_eigenvaluesImag  = mw_eigenvalues.ToArray (MWArrayComponent.Imaginary)
```

```
Catch
  ' nothing, db_eigenvectorsImag and db_eigenvaluesImag will be assigned value of zero
End Try

' print out
Console.WriteLine("Eigenvalues of the matrix A : " )

for i=0 to (row-1)

  Console.Write( db_eigenvaluesReal.GetValue(i,i).ToString() +  " + "  )
  Console.Write( db_eigenvaluesImag.GetValue(i,i).ToString() +  "i"    )
  Console.WriteLine()

Next
Console.WriteLine()
' or
Console.WriteLine("Eigenvectors of the matrix A : " )

for j=0 to (col-1)

  Console.WriteLine("Eigenvector {0} is : ", (j+1).ToString()  )

  for i=0 to (row-1)

    Console.Write( db_eigenvectorsReal.GetValue(i,j).ToString() + " + "  )
    Console.Write( db_eigenvectorsImag.GetValue(i,j).ToString() + "i"    )
    Console.WriteLine()
  Next

  Console.WriteLine()
Next

Console.WriteLine()

' free memory
```

```
    mw_A.Dispose()
    MWNumericArray.DisposeArray(mw_ArrayOut)

    mw_eigenvectors.Dispose()
    mw_eigenvalues.Dispose()

  End Sub


  End Class

End Module
```

end code

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 **A** are:

$$eigenvalue_1 = -0.30710$$
$$eigenvalue_2 = -0.24645 + 1.76008i$$
$$eigenvalue_3 = -0.24645 - 1.76008i$$

2. The MATLAB function `[V,D] = eig(A)` also assigns eigenvectors `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:

$$eigenvector_1 = \begin{bmatrix} -0.83261 \\ -0.35534 \\ -0.42485 \end{bmatrix}$$

$$eigenvector_2 = \begin{bmatrix} 0.20027 - 0.13936i \\ -0.21104 - 0.64472i \\ -0.69301 \end{bmatrix}$$

$$eigenvector_3 = \begin{bmatrix} 0.20027 + 0.13936i \\ -0.21104 + 0.64472i \\ -0.69301 \end{bmatrix}$$

Chapter 13

Random Numbers

In this chapter we will generate a class ***RandonNumber*** from common M-files working on matrix computation problems. The generated functions of this class will be used in a VB .Net 2008 project to solve random number problems.

We will write two M-files, `myrand.m` and `myrandn.m` as shown below to generate the class ***RandonNumber***.

myrand.m

```
function Y = myrand(m,n)

Y = rand(m,n) ;
```

myrandn.m

```
function Y = myrandn(m,n)

Y = randn(m,n) ;
%
```

The following procedure to create the class ***RandomNumber*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `RandomNumberNameSpace.dll` that contains the class ***RandomNumber*** (see Fig.13.1 and Fig.13.2).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `RandomNumberNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `RandomNumberNameSpace.dll`, and then click OK (see Fig. 13.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

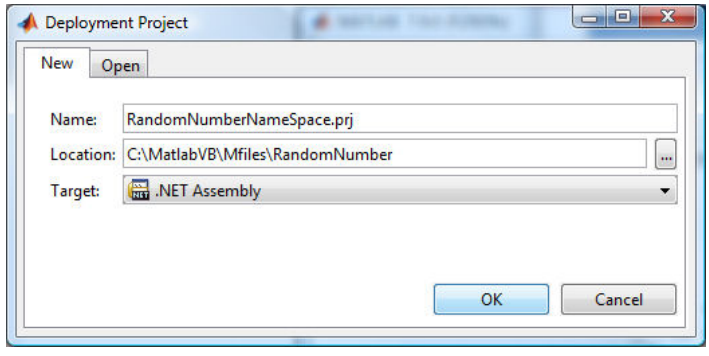


Figure 13.1: Deployment project for ***RandomNumber***

The following sections show how to use the functions in the class ***RandomNumber*** to solve common computation problems. The full code is at the end of this chapter.

13.1 Uniform Random Numbers

Uniform random numbers are random numbers that lie within a specific range. This section describes how to use the MATLAB function `myrand(..)` in the generated class to solve uniform random number problems. The functions of this class uses the MATLAB function `rand(m,n)`

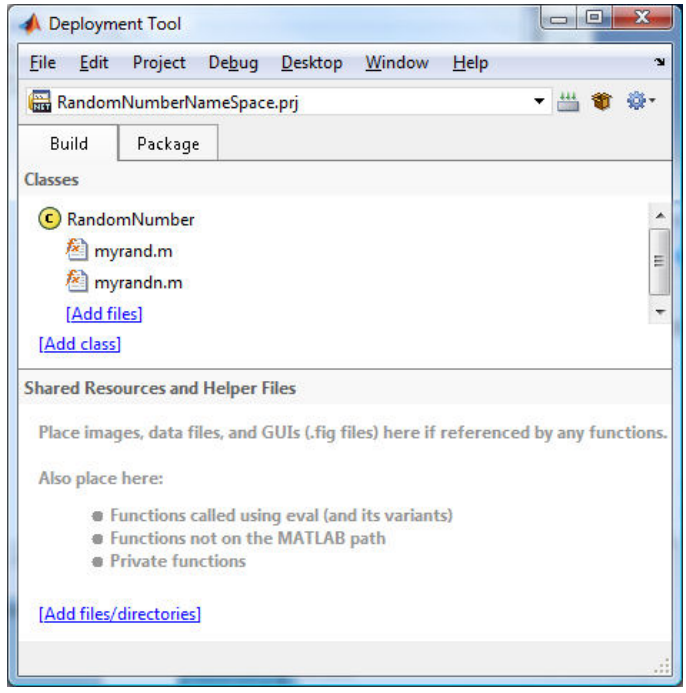


Figure 13.2: Adding M-files in deployment project for ***RandomNumber***

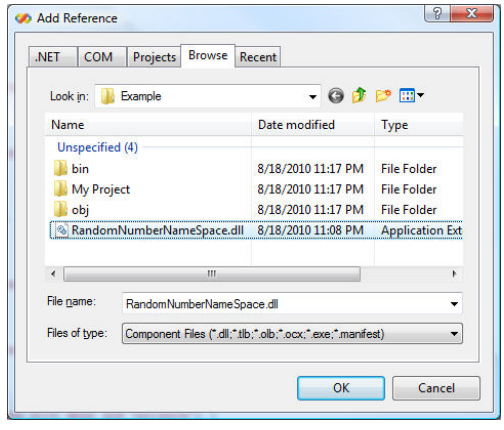


Figure 13.3: Add the reference `RandomNumberNameSpace.dll` in the VB project

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 [5].

13.1.1 Generating Uniform Random Numbers in Range [0,1]

Problem 1

input . A number, $N = 5$

output . Generating N uniform random numbers in range $[0, 1]$

The following subroutine uses a function in the class ***RandomNumber*** to solve Problem 1.

Listing code

```
Public Sub UniformRandom_vector()
  Dim row As Integer = 1
  Dim col As Integer = 5

  Dim uniformRandVector As MWNumericArray = Nothing
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = new MWNumericArray(col)

  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
               New RandomNumberNameSpace.RandomNumber()

  uniformRandVector = objMatlab.myrand(mw_row, mw_col)

  Console.WriteLine("Uniform random numbers from 0 to 1:")
  Console.WriteLine(uniformRandVector)

  ' Or convert back to double
  Dim db_uniformRandVector() As Double =
     uniformRandVector.ToVector(MWArrayComponent.Real)
  Console.WriteLine()
```

```
    PrintValues(db_uniformRandVector)

    uniformRandVector.Dispose()

End Sub
```

end code

13.1.2 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 $[a, b]$

The following subroutine uses a function in the class ***RandomNumber*** to solve Problem 2.

Listing code

```
Public Sub UniformRandom_vector2()

  Dim a As Double = 2.0
  Dim b As Double = 18.0

  Dim row As Integer = 1
  Dim col As Integer = 6
  Dim i As Integer

  ' Get random numbers in [0, 1]
  Dim uniformRandVector As MWNumericArray = Nothing

  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = New MWNumericArray(col)

  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                New RandomNumberNameSpace.RandomNumber()
```

```
    uniformRandVector = objMatlab.myrand(mw_row, mw_col)

    ' Convert back to double
    Dim db_uniformRandVector() As Double = _
      uniformRandVector.ToVector(MWArrayComponent.Real)

    Console.WriteLine()
    Console.WriteLine("Uniform random numbers from a=2 to b=18 : ")

    For i=0 to (db_uniformRandVector.Length -1)
      Dim aVal As Double = a + (b-a)*db_uniformRandVector(i)
      Console.WriteLine( aVal.ToString() )
    Next

    ' Free memories
    uniformRandVector.Dispose()

  End Sub
```

end code

13.1.3 Generating a Matrix of Uniform Random Numbers in Range [0,1]

In this section, we will use MATLAB functions to solve the following problems.

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]

The following subroutine uses a function in the class ***RandomNumber*** to solve Problem 3.

Listing code

```
  Public Sub UniformRandom_matrix()
```

```
    Dim row As Integer = 8
    Dim col As Integer = 5

    Dim mw_uniformRandMatrix As MWNumericArray = Nothing
    Dim mw_row As MWNumericArray = New MWNumericArray(row)
    Dim mw_col As MWNumericArray = New MWNumericArray(col)

    Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                  New RandomNumberNameSpace.RandomNumber()

    mw_uniformRandMatrix = objMatlab.myrand(mw_row, mw_col)

    Console.WriteLine("The matrix of uniform random numbers from 0 to 1 :" )
    Console.WriteLine(mw_uniformRandMatrix)

    ' Or convert back to  double
    Dim db_uniformRandMatrix(,) As Double = _
       mw_uniformRandMatrix.ToArray(MWArrayComponent.Real)
    Console.WriteLine()
    PrintValues(db_uniformRandMatrix)

    mw_uniformRandMatrix.Dispose()

End Sub
```

end code

13.1.4 Generating a Matrix of Uniform Random Numbers in Range [a,b]

In this section, we will use MATLAB functions to solve the following problems.

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 $[a, b]$

The following subroutine uses a function in the class ***RandomNumber*** to solve Problem 4.

Listing code

```
Public Sub UniformRandom_matrix2()

  Dim row As Integer = 8
  Dim col As Integer = 5
  Dim i, j As Integer

  Dim a As Double = 4.0
  Dim b As Double = 17.0

  Dim mw_uniformRandMatrix As MWNumericArray = Nothing
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = New MWNumericArray(col)

  ' Get matrix of random number from 0 to 1
  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                New RandomNumberNameSpace.RandomNumber()
  mw_uniformRandMatrix = objMatlab.myrand(mw_row, mw_col)

  '' Convert to double
  Dim db_uniformRandMatrix(,) As Double = _
    mw_uniformRandMatrix.ToArray(MWArrayComponent.Real)

  ' Print out
  Console.WriteLine()
  Console.WriteLine("The matrix of uniform random numbers from a=4.0 to b=17.0:")

  For i=0 to (row - 1)
    For j=0 to (col-1)
      Console.WriteLine("{0}", a + (b-a)*db_uniformRandMatrix(i,j)  )
```

```
      Next
    Console.WriteLine()
    Next

    Console.WriteLine()

    ' Free memories
    mw_uniformRandMatrix.Dispose()

  End Sub
```

end code

13.2 Normal Random Numbers

Normal random numbers are random numbers that establish the normal distribution (Gaussian distribution). This section describe how to use the function in the generated class ***RandomNumber*** 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 $\sigma^2 = 1$.

13.2.1 Generating Normal Random Numbers with mean=0 and variance=1

In this section, we will use MATLAB functions to solve the following problems.

Problem 5

input . A number, $N = 5$

output . Generating (N) normal random numbers with :

mean $\mu = 0$

variance $\sigma^2 = 1$

The following subroutine uses a function in the class ***RandomNumber*** to solve Problem 5 .

Listing code

```
Public Sub NormalRandom_vector()

  Dim row As Integer = 1
  Dim col As Integer = 5

  Dim mw_normalRandVector As MWNumericArray = Nothing
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = New MWNumericArray(col)

  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
              New RandomNumberNameSpace.RandomNumber()
  mw_normalRandVector = objMatlab.myrandn(mw_row, mw_col)

  Console.WriteLine("Normal random numbers :")
  Console.WriteLine(mw_normalRandVector)

  ' Or convert back to double
  Dim db_normalRandVector() As Double = _
    mw_normalRandVector.ToVector(MWArrayComponent.Real)
  Console.WriteLine()
  PrintValues(db_normalRandVector)

  ' Free memories
  mw_normalRandVector.Dispose()

End Sub
```

end code

13.2.2 Generating Normal Random Numbers with mean=a and variance=b

Problem 6

input . A number, $N = 5$

output . Generating N normal random numbers with specified properties:

mean $\mu = 0.56$

variance $\sigma^2 = 0.12$

The following subroutine uses a function in the class ***RandomNumber*** to solve Problem 6.

Listing code

```
Public Sub NormalRandom_vector2()

  ' Generate a vector of normal random numbers at
  ' particular mean and variance
  Dim row As Integer = 1
  Dim col As Integer = 5

  Dim mean_mu As Double = 0.56
  Dim variance As Double = 0.12
  Dim i As Integer

  Dim mw_normalRandVector As MWNumericArray = Nothing
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = new MWNumericArray(col)

  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
               New RandomNumberNameSpace.RandomNumber()

  mw_normalRandVector = objMatlab.myrandn(mw_row, mw_col)

  ' Convert to double
  Dim db_normalRandVector() As Double = _
    mw_normalRandVector.ToVector(MWArrayComponent.Real)

  Dim standard_deviation As Double = Math.Sqrt(variance)
```

```
    ' Print out
    Console.WriteLine("Normal random numbers with mean = 0.56 and variance = 0.12 :")

    For i=0 to (col-1)
      Dim aVal As Double = mean_mu + standard_deviation*db_normalRandVector(i)
      Console.WriteLine(aVal.ToString() + ControlChars.Tab)
    Next

    Console.WriteLine()

    ' Free memories
    mw_normalRandVector.Dispose()

  End Sub
```

end code

13.2.3 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 $\sigma^2 = 1$

The following subroutine uses a function in the class ***RandomNumber*** to solve Problem 7.

Listing code

```
  Public Sub NormalRandom_matrix()

    Dim row As Integer = 8
```

```
    Dim col As Integer = 5

    Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                  New RandomNumberNameSpace.RandomNumber()

    Dim mw_normalRandMatrix As MWNumericArray = Nothing
    Dim mw_row As MWNumericArray = New MWNumericArray(row)
    Dim mw_col As MWNumericArray = New MWNumericArray(col)

    ' Get matrix of random number from 0 to 1
    mw_normalRandMatrix = objMatlab.myrandn(mw_row, mw_col)

    Console.WriteLine(mw_normalRandMatrix)

    ' Or convert to double
    Dim db_normalRandMatrix(,) As Double = _
      mw_normalRandMatrix.ToArray(MWArrayComponent.Real)
    Console.WriteLine()
    PrintValues(db_normalRandMatrix)

    ' Free memories
    mw_normalRandMatrix.Dispose()

  End Sub
```

end code

13.2.4 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 $\sigma^2 = 0.12$

The following subroutine uses a function in the class ***RandomNumber*** to solve Problem 8.

Listing code

```
Public Sub NormalRandom_matrix2()

  Dim row As Integer = 8
  Dim col As Integer = 5
  Dim i, j As Integer

  Dim mean_mu As Double = 0.56
  Dim variance As Double = 0.12

  Dim standard_deviation As Double = Math.Sqrt(variance)

  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
               New RandomNumberNameSpace.RandomNumber()

  Dim mw_normalRandMatrix As MWNumericArray = Nothing
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = New MWNumericArray(col)

  ' Get matrix of normal random number
  mw_normalRandMatrix = objMatlab.myrandn(mw_row, mw_col)

  ' Convert to double
  Dim db_normalRandMatrix(,) As Double = _
    mw_normalRandMatrix.ToArray(MWArrayComponent.Real)

  ' Print out
  Console.WriteLine()
  Console.WriteLine("The matrix of normal random numbers ")
```

```
      Console.WriteLine("at specified mean and variance" )

      For i=0 to (row-1)
        For j=0 to (col-1)

          Dim aVal As Double = mean_mu + standard_deviation*db_normalRandMatrix(i,j)
          Console.WriteLine( aVal.ToString() )
        Next
      Console.WriteLine()
      Next

      Console.WriteLine()

      ' Free memories
      mw_normalRandMatrix.Dispose()

    End Sub
```

end code

The following is the full code for this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports RandomNumberNameSpace

Module Module1

  Sub Main()

    Console.WriteLine("Random Number")

    Dim objProgram As Example = New Example()
    Console.WriteLine()
```

```
    Console.WriteLine("1. Vector of uniform random number from 0 to 1")
    objProgram.UniformRandom_vector()

    Console.WriteLine()
    Console.WriteLine("2. Vector of uniform random numbers from a to b ")
    objProgram.UniformRandom_vector2()

    Console.WriteLine()
    Console.WriteLine("3. Matrix of random numbers")
    objProgram.UniformRandom_matrix()

    Console.WriteLine()
    Console.WriteLine("4. Matrix of random numbers from a to b")
    objProgram.UniformRandom_matrix2()

    Console.WriteLine()
    Console.WriteLine("5. Vector of normal random number")
    objProgram.NormalRandom_vector()

    Console.WriteLine()
    Console.WriteLine("6. Vector of normal random number with mean and variance")
    objProgram.NormalRandom_vector2()

    Console.WriteLine()
    Console.WriteLine("7. Matrix of normal random number")
    objProgram.NormalRandom_matrix()

    Console.WriteLine()
    Console.WriteLine("8. Matrix of normal random number with mean and variance")
    objProgram.NormalRandom_matrix2()

End Sub

Public Class Example
```

```
' ****************************
Public Sub UniformRandom_vector()
  Dim row As Integer = 1
  Dim col As Integer = 5

  Dim uniformRandVector As MWNumericArray = Nothing
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = new MWNumericArray(col)

  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                New RandomNumberNameSpace.RandomNumber()

  uniformRandVector = objMatlab.myrand(mw_row, mw_col)

  Console.WriteLine("Uniform random numbers from 0 to 1:")
  Console.WriteLine(uniformRandVector)

  ' Or convert back to double
  Dim db_uniformRandVector() As Double = _
    uniformRandVector.ToVector(MWArrayComponent.Real)
  Console.WriteLine()
  PrintValues(db_uniformRandVector)

  uniformRandVector.Dispose()

End Sub

'  *********************************
Public Sub UniformRandom_vector2()

  Dim a As Double = 2.0
  Dim b As Double = 18.0

  Dim row As Integer = 1
  Dim col As Integer = 6
  Dim i As Integer
```

```
    ' Get random numbers in [0, 1]
    Dim uniformRandVector As MWNumericArray = Nothing

    Dim mw_row As MWNumericArray = New MWNumericArray(row)
    Dim mw_col As MWNumericArray = New MWNumericArray(col)

    Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                  New RandomNumberNameSpace.RandomNumber()

    uniformRandVector = objMatlab.myrand(mw_row, mw_col)

    ' Convert back to double
    Dim db_uniformRandVector() As Double = _
     uniformRandVector.ToVector(MWArrayComponent.Real)

    Console.WriteLine()
    Console.WriteLine("Uniform random numbers from a=2 to b=18 : ")

    For i=0 to (db_uniformRandVector.Length -1)
      Dim aVal As Double = a + (b-a)*db_uniformRandVector(i)
      Console.WriteLine( aVal.ToString() )
    Next

    ' Free memories
    uniformRandVector.Dispose()

  End Sub

  ' ************************************
  Public Sub UniformRandom_matrix()

    Dim row As Integer = 8
    Dim col As Integer = 5

    Dim mw_uniformRandMatrix As MWNumericArray = Nothing
```

```
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = New MWNumericArray(col)

  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                New RandomNumberNameSpace.RandomNumber()

  mw_uniformRandMatrix = objMatlab.myrand(mw_row, mw_col)

  Console.WriteLine("The matrix of uniform random numbers from 0 to 1 :" )
  Console.WriteLine(mw_uniformRandMatrix)

  ' Or convert back to  double
  Dim db_uniformRandMatrix(,) As Double = _
    mw_uniformRandMatrix.ToArray(MWArrayComponent.Real)
  Console.WriteLine()
  PrintValues(db_uniformRandMatrix)

  mw_uniformRandMatrix.Dispose()

End Sub

' ********************************
Public Sub UniformRandom_matrix2()

  Dim row As Integer = 8
  Dim col As Integer = 5
  Dim i, j As Integer

  Dim a As Double = 4.0
  Dim b As Double = 17.0

  Dim mw_uniformRandMatrix As MWNumericArray = Nothing
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = New MWNumericArray(col)

  ' Get matrix of random number from 0 to 1
```

```
    Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                  New RandomNumberNameSpace.RandomNumber()
    mw_uniformRandMatrix = objMatlab.myrand(mw_row, mw_col)

    '' Convert to double
    Dim db_uniformRandMatrix(,) As Double = _
      mw_uniformRandMatrix.ToArray(MWArrayComponent.Real)

    ' Print out
    Console.WriteLine()
    Console.WriteLine("The matrix of uniform random numbers from a=4.0 to b=17.0:")

    For i=0 to (row - 1)
      For j=0 to (col-1)
        Console.WriteLine("{0}", a + (b-a)*db_uniformRandMatrix(i,j)  )
      Next
    Console.WriteLine()
    Next

    Console.WriteLine()

    ' Free memories
    mw_uniformRandMatrix.Dispose()

  End Sub

  ' ******************************** */
  Public Sub NormalRandom_vector()

    Dim row As Integer = 1
    Dim col As Integer = 5

    Dim mw_normalRandVector As MWNumericArray = Nothing
    Dim mw_row As MWNumericArray = New MWNumericArray(row)
    Dim mw_col As MWNumericArray = New MWNumericArray(col)
```

```
  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                New RandomNumberNameSpace.RandomNumber()
  mw_normalRandVector = objMatlab.myrandn(mw_row, mw_col)

  Console.WriteLine("Normal random numbers :")
  Console.WriteLine(mw_normalRandVector)

  ' Or convert back to double
  Dim db_normalRandVector() As Double = _
    mw_normalRandVector.ToVector(MWArrayComponent.Real)
  Console.WriteLine()
  PrintValues(db_normalRandVector)

  ' Free memories
  mw_normalRandVector.Dispose()

End Sub

' ****************************
Public Sub NormalRandom_vector2()

  ' Generate a vector of normal random numbers at
  ' particular mean and variance
  Dim row As Integer = 1
  Dim col As Integer = 5

  Dim mean_mu As Double = 0.56
  Dim variance As Double = 0.12
  Dim i As Integer

  Dim mw_normalRandVector As MWNumericArray = Nothing
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = new MWNumericArray(col)

  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                New RandomNumberNameSpace.RandomNumber()
```

```
    mw_normalRandVector = objMatlab.myrandn(mw_row, mw_col)

    ' Convert to double
    Dim db_normalRandVector() As Double = _
      mw_normalRandVector.ToVector(MWArrayComponent.Real)

    Dim standard_deviation As Double = Math.Sqrt(variance)

    ' Print out
    Console.WriteLine("Normal random numbers with mean = 0.56 and variance = 0.12 :")

    For i=0 to (col-1)
      Dim aVal As Double = mean_mu + standard_deviation*db_normalRandVector(i)
      Console.WriteLine(aVal.ToString() + ControlChars.Tab)
    Next

    Console.WriteLine()

    ' Free memories
    mw_normalRandVector.Dispose()

  End Sub

' ********************************** *
  Public Sub NormalRandom_matrix()

    Dim row As Integer = 8
    Dim col As Integer = 5

    Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                  New RandomNumberNameSpace.RandomNumber()

    Dim mw_normalRandMatrix As MWNumericArray = Nothing
    Dim mw_row As MWNumericArray = New MWNumericArray(row)
    Dim mw_col As MWNumericArray = New MWNumericArray(col)
```

```
  ' Get matrix of random number from 0 to 1
  mw_normalRandMatrix = objMatlab.myrandn(mw_row, mw_col)

  Console.WriteLine(mw_normalRandMatrix)

  ' Or convert to double
  Dim db_normalRandMatrix(,) As Double = _
    mw_normalRandMatrix.ToArray(MWArrayComponent.Real)
  Console.WriteLine()
  PrintValues(db_normalRandMatrix)

  ' Free memories
  mw_normalRandMatrix.Dispose()

End Sub

' ************************** *
Public Sub NormalRandom_matrix2()

  Dim row As Integer = 8
  Dim col As Integer = 5
  Dim i, j As Integer

  Dim mean_mu As Double = 0.56
  Dim variance As Double = 0.12

  Dim standard_deviation As Double = Math.Sqrt(variance)

  Dim objMatlab  As RandomNumberNameSpace.RandomNumber = _
                New RandomNumberNameSpace.RandomNumber()

  Dim mw_normalRandMatrix As MWNumericArray = Nothing
  Dim mw_row As MWNumericArray = New MWNumericArray(row)
  Dim mw_col As MWNumericArray = New MWNumericArray(col)
```

```
    ' Get matrix of normal random number
    mw_normalRandMatrix = objMatlab.myrandn(mw_row, mw_col)

    ' Convert to double
    Dim db_normalRandMatrix(,) As Double = _
      mw_normalRandMatrix.ToArray(MWArrayComponent.Real)

    ' Print out
    Console.WriteLine()
    Console.WriteLine("The matrix of normal random numbers ")
    Console.WriteLine("at specified mean and variance" )

    For i=0 to (row-1)
      For j=0 to (col-1)

        Dim aVal As Double = mean_mu + standard_deviation*db_normalRandMatrix(i,j)
        Console.WriteLine( aVal.ToString() )
      Next
    Console.WriteLine()
    Next

    Console.WriteLine()

    ' Free memories
    mw_normalRandMatrix.Dispose()

  End Sub

  ' ********************************
  Public Sub PrintValues(ByVal myArr As Array)
    Dim myEnumerator As System.Collections.IEnumerator = _
       myArr.GetEnumerator()
    Dim i As Integer = 0
    Dim cols As Integer = myArr.GetLength(myArr.Rank - 1)

    'for row vector or column vector
```

```
      If myArr.Rank = 1 Then

        While myEnumerator.MoveNext()
          Console.WriteLine(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While

      Else
        'for other
        While myEnumerator.MoveNext()
          If i < cols Then
            i += 1
          Else
            Console.WriteLine()
            i = 1
          End If
          Console.Write(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While
      End If

      Console.WriteLine()
    End Sub


  End Class

End Module
```

end code

Part II:
Using MATLAB functions in VB. NET Windows Form Applications

Chapter 14

Using MATLAB Functions In VB .NET Windows Forms Applications

14.1 Using MATLAB Built-in Functions to Calculate a Addition Matrix

This chapter describes how to use MATLAB functions in a VB Windows Form application. The steps of this application are:

1. Get input data from a Windows Form application.
2. Assign input data to variables in VB *Double* type.
3. Use MATLAB built-in functions in the generated class to perform the task.
4. Transfer back result values to the VB *Double* type.
5. Put the result to the Windows Form application.

An example of the Windows Forms application in this section is a simple application of the matrix addition.

We will write a simple M-file `myplus.m` as shown below to generate a class for a Windows Form application .

myplus.m

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

y = a + b ;
```

The following procedure to create the class ***Plus*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `PlusNameSpace.dll` that contains the class ***Plus*** (see Fig.14.1 and Fig.14.2).
2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.
3. Copy the file `PlusNameSpace.dll` and put it in the same folder `Module1.vb`
4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `PlusNameSpace.dll`, and then click OK (see Fig. 14.3).
5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.
6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

The form of this Windows Form application shows in Fig.14.4
The following is the code of this example for the Windows Form application.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports PlusNameSpace
```

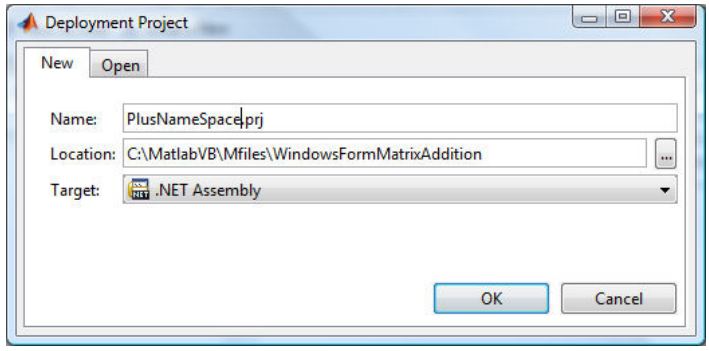


Figure 14.1: Deployment project for ***Plus***

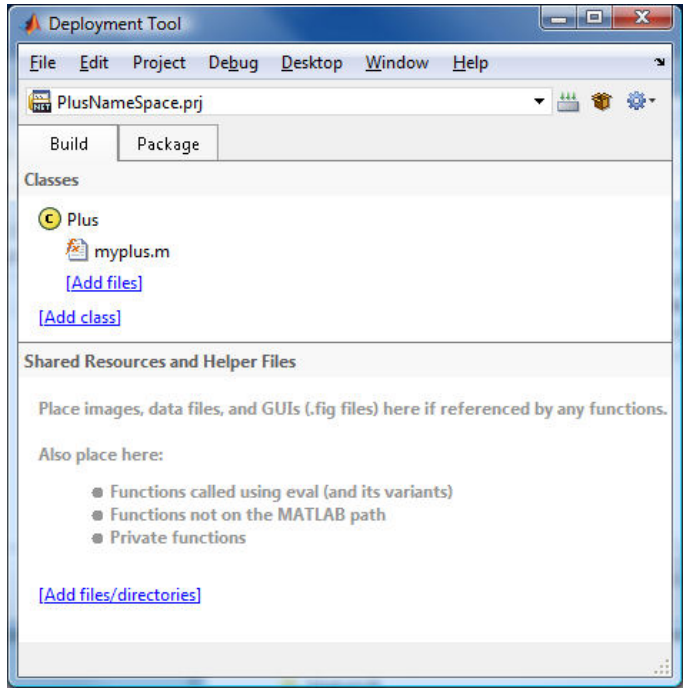


Figure 14.2: Adding M-files in deployment project for ***Plus***

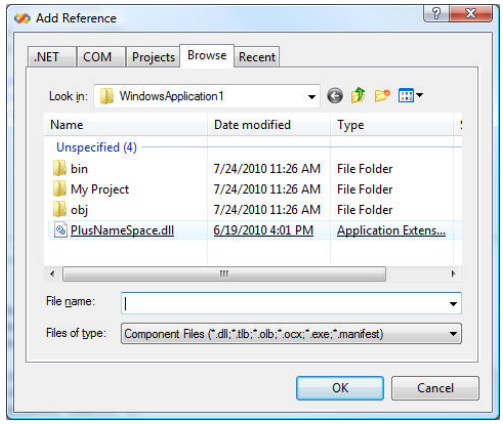


Figure 14.3: Add the reference `PlusNameSpace.dll` in the VB project

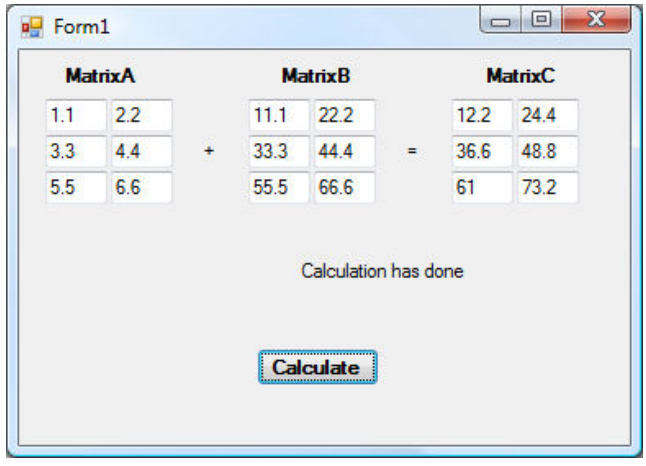


Figure 14.4: Matrix Addition in Windows Forms Application

```
Public Class Form1

  Public TextBoxMatrixA(,) As TextBox
  Public TextBoxMatrixB(,) As TextBox
  Public TextBoxMatrixC(,) As TextBox

  Public StartPoint As Point = New Point()
  Public TextboxLength As Integer
  Public TextboxHeight As Integer

  Public NumRow As Integer = 3
  Public NumCol As Integer = 2

  Public MatrixA_data(,) As Double
  Public MatrixB_data(,) As Double
  Public MatrixC_data(,) As Double

  Public Sub New()
    InitializeComponent()
    InitMatrixes()

  End Sub

  ' ***************************
  Public Sub InitMatrixes()

    TextBoxMatrixA = New TextBox(NumRow, NumCol) {}
    TextBoxMatrixB = New TextBox(NumRow, NumCol) {}
    TextBoxMatrixC = New TextBox(NumRow, NumCol) {}

    MatrixA_data = New Double(NumRow, NumCol) {}
    MatrixB_data = New Double(NumRow, NumCol) {}
    MatrixC_data = New Double(NumRow, NumCol) {}

    StartPoint.X = 16
```

```
    StartPoint.Y = 30

    TextboxLength = 38
    TextboxHeight = 20

    SetTextBoxPositionMatrixA()
    SetTextBoxPositionMatrixB()

    labelPlusSign.Location = New System.Drawing.Point(StartPoint.X - 30, _
                                                      (StartPoint.Y + 24))

    SetTextBoxPositionMatrixC()
    labelEqualSign.Location = New System.Drawing.Point(StartPoint.X - 30, _
                                                       (StartPoint.Y + 24))
  End Sub

  ' ***************************
  Public Sub SetTextBoxPositionMatrixA()

    Dim i, j As Integer

    For i = 0 To NumRow - 1
      For j = 0 To NumCol - 1

        TextBoxMatrixA(i, j) = New System.Windows.Forms.TextBox()
        TextBoxMatrixA(i, j).Location = New System.Drawing.Point( _
               StartPoint.X + (TextboxLength + 2) * j, StartPoint.Y + (TextboxHeight + 2) * i)
        TextBoxMatrixA(i, j).Name = "textBox1"
        TextBoxMatrixA(i, j).Size = New System.Drawing.Size(TextboxLength, TextboxHeight)
        TextBoxMatrixA(i, j).TabIndex = i
        TextBoxMatrixA(i, j).Text = "0.0"
        Me.Controls.Add(TextBoxMatrixA(i, j))

      Next
    Next
```

```
End Sub

' ************************** */
Public Sub SetTextBoxPositionMatrixB()

  Dim i, j As Integer

  StartPoint.X = StartPoint.X + (NumRow * TextboxLength + 10)

  For i = 0 To NumRow - 1
    For j = 0 To NumCol - 1

      TextBoxMatrixB(i, j) = New System.Windows.Forms.TextBox()
      TextBoxMatrixB(i, j).Location = New System.Drawing.Point( _
             StartPoint.X + (TextboxLength + 2) * j, StartPoint.Y + (TextboxHeight + 2) * i)
      TextBoxMatrixB(i, j).Name = "textBox1"
      TextBoxMatrixB(i, j).Size = New System.Drawing.Size(TextboxLength, TextboxHeight)
      TextBoxMatrixB(i, j).TabIndex = i
      TextBoxMatrixB(i, j).Text = "0.0"
      Me.Controls.Add(TextBoxMatrixB(i, j))

    Next
  Next

End Sub

' ************************** */
Public Sub SetTextBoxPositionMatrixC()

  Dim i, j As Integer

  StartPoint.X = StartPoint.X + (NumRow * TextboxLength + 10)

  For i = 0 To NumRow - 1
    For j = 0 To NumCol - 1
```

```
        TextBoxMatrixC(i, j) = New System.Windows.Forms.TextBox()
        TextBoxMatrixC(i, j).Location = New System.Drawing.Point( _
         StartPoint.X + (TextboxLength + 2) * j, StartPoint.Y + (TextboxHeight + 2) * i)
        TextBoxMatrixC(i, j).Name = "textBox1"
        TextBoxMatrixC(i, j).Size = New System.Drawing.Size(TextboxLength, TextboxHeight)
        TextBoxMatrixC(i, j).TabIndex = i
        TextBoxMatrixC(i, j).Text = "0.0"
        Me.Controls.Add(TextBoxMatrixC(i, j))

      Next
    Next

  End Sub

  ' ************************** */
  Private Sub ButtonMatrixAddition_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles ButtonMatrixAddition.Click

    '/////////// get matrix A /////////////
    Try
      GetMatrixA()
    Catch
      MessageBox.Show("Data in matrix A is not valid")
      Return
    End Try

    '/////////// get matrix B /////////////
    Try
      GetMatrixB()
    Catch
      MessageBox.Show("Data in matrix B is not valid")
      Return
    End Try

    '///////////
```

```
  Me.labelCalcInfo.Text = "Calculation is in process. Please wait."
  Me.Invalidate()
  Me.Refresh()

  '  /////////////// calculate matrix C ////////////
  GetMatrixC()

End Sub

' ************************** */
Public Sub GetMatrixA()

  Dim i, j As Integer

  For i = 0 To (NumRow - 1)
    For j = 0 To (NumCol - 1)

      MatrixA_data(i, j) = Convert.ToDouble(TextBoxMatrixA(i, j).Text)

    Next
  Next

End Sub

' **************************
Public Sub GetMatrixB()

  Dim i, j As Integer

  For i = 0 To (NumRow - 1)
    For j = 0 To (NumCol - 1)

      MatrixB_data(i, j) = Convert.ToDouble(TextBoxMatrixB(i, j).Text)

    Next
  Next
```

```
End Sub

' ***************************
Public Sub GetMatrixC()

  'calculate matrix multiplication
  Dim mw_A As MWNumericArray = New MWNumericArray(MatrixA_data)
  Dim mw_B As MWNumericArray = New MWNumericArray(MatrixB_data)
  Dim mw_C As MWNumericArray = Nothing

  Dim objMatlab As PlusNameSpace.Plus = New PlusNameSpace.Plus()

  mw_C = objMatlab.myplus(mw_A, mw_B)

  ' convert back to double
  MatrixC_data = mw_C.ToArray(MWArrayComponent.Real)

  ' display to GUI
  Dim i, j As Integer
  For i = 0 To (NumRow - 1)
    For j = 0 To (NumCol - 1)

      TextBoxMatrixC(i, j).Text = MatrixC_data(i, j).ToString()

    Next
  Next

  'Show message

  labelCalcInfo.Text = "Calculation has done"

  mw_A.Dispose()
  mw_B.Dispose()
  mw_C.Dispose()

End Sub
```

```
  ' ***************************
  ' ***************************
  ' ***************************
End Class
```

end code

14.2 Using MATLAB Built-in Functions to Calculate a Multiplication Matrix

This section describes how to use MATLAB functions in a VB Windows Form application. The tasks of this application are:

1. Get input data from the Windows Form application.
2. Assign input data to variables in the VB *Double* type.
3. Use MATLAB built-in functions in the generated class to perform the task.
4. Transfer back result values to VB *Double* type.
5. Put the result to the Windows Form application.

An example of the Windows Forms application in this section is a simple application of the matrix multiplication.
We will write the M-file `mymtimes.m` as shown below to generate a class for a Windows Form application.

mymtimes.m

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

y = a*b ;
```

The following procedure to create the class ***MatrixComputations*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `MatrixComputationsNameSpace.dll` that contains the class ***MatrixComputations*** (see Fig.14.5 and Fig.14.6).

2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.

3. Copy the file `MatrixComputationsNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `MatrixComputationsNameSpace.dll`, and then click OK (see Fig. 14.7).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

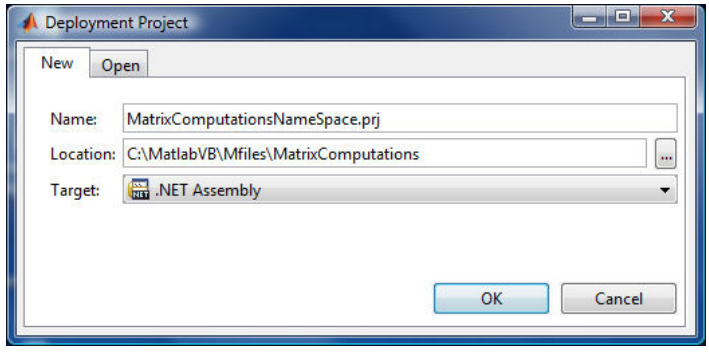


Figure 14.5: Deployment project for ***MatrixComputations***

The form of this Windows Form application shows in Fig.14.8

The following is code of this example for Windows Form application.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Arrays
Imports MathWorks.MATLAB.NET.Utility
Imports MatrixComputationsNameSpace
```

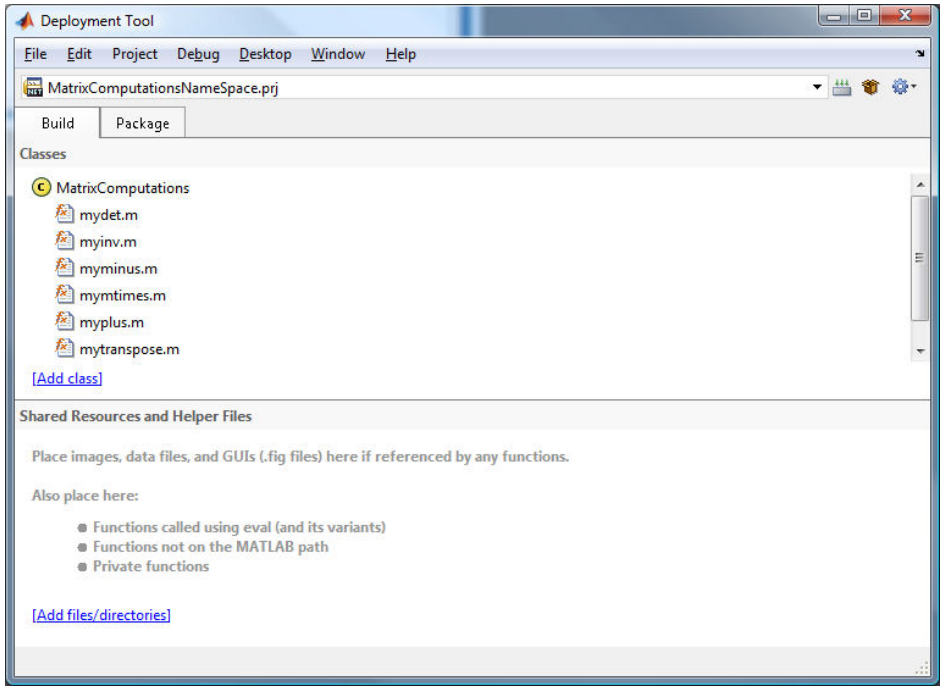


Figure 14.6: Adding M-files in deployment project for ***MatrixComputations***

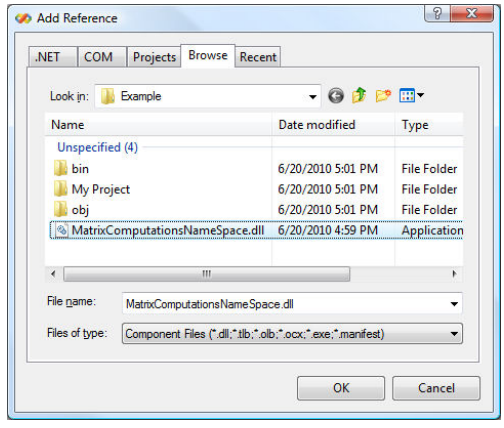


Figure 14.7: Add the reference `MatrixComputationsNameSpace.dll` in the VB project

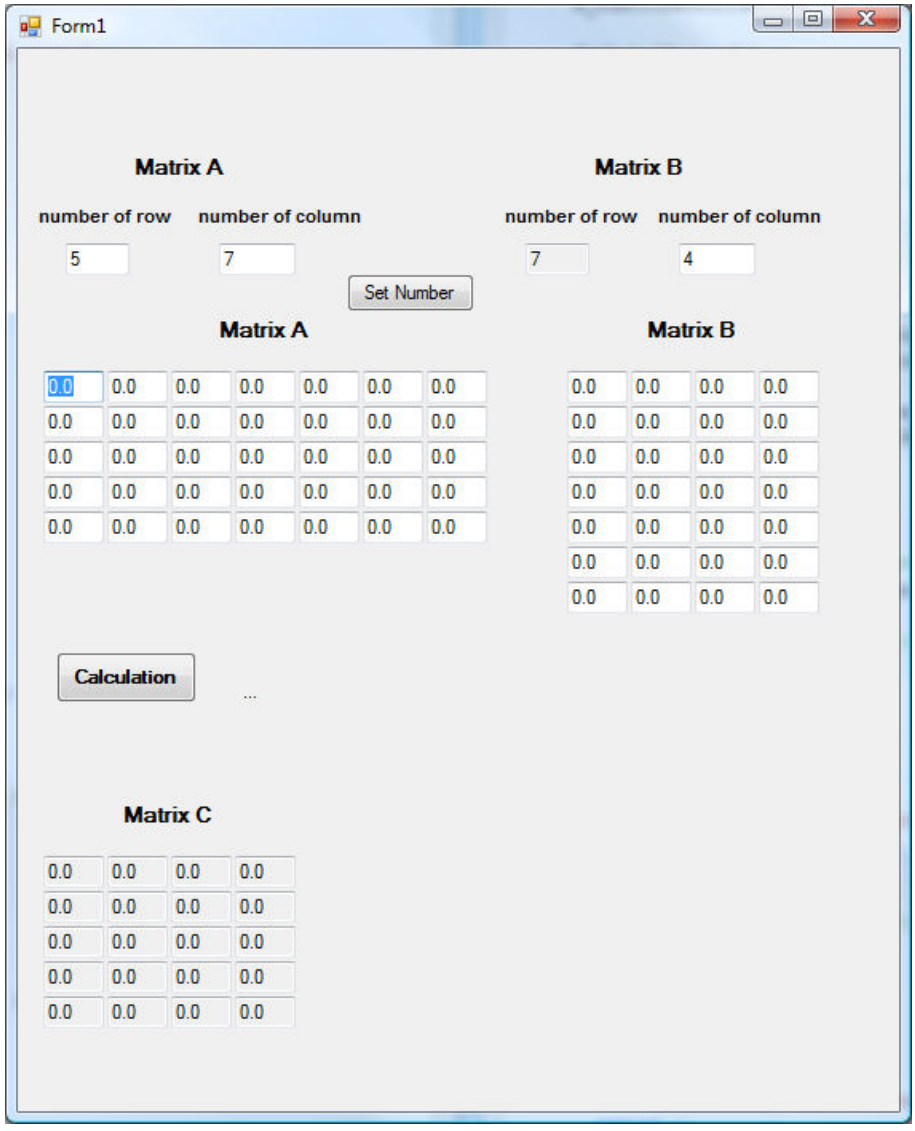


Figure 14.8: Matrix Multiplication in Windows Forms Application

```
Public Class Form1
  Public arrTextBoxMatrixA(,) As TextBox
  Public db_MatrixA(,) As Double

  Public arrTextBoxMatrixB(,) As TextBox
  Public db_MatrixB(,) As Double

  Public arrTextBoxMatrixC(,) As TextBox
  Public db_MatrixC(,) As Double

  Public m_numRowMatricATextBox As Integer
  Public m_numColMatricATextBox As Integer

  Public m_tbLength As Integer
  Public m_tbHeight As Integer

  Public m_startPoint As Point = New Point()
  Public m_startPointMatrixB As Point = New Point()
  Public m_startPointMatrixC As Point = New Point()

  Public m_numRowMatrixBTextBox As Integer
  Public m_numColMatrixBTextBox As Integer

  Public m_FormLength As Integer
  Public m_FormHeight As Integer

  Public m_numRowMatriCTextBox As Integer
  Public m_numColMatriCTextBox As Integer

  Public Sub New()
    InitializeComponent()

    'Add any initialization after the InitializeComponent() call
    m_startPoint.X = 16
    m_startPoint.Y = 24 + 28 + 120 + 30
```

```
    m_numRowMatricATextBox = 5
    m_numColMatricATextBox = 7

    m_numRowMatrixBTextBox = m_numColMatricATextBox
    m_numColMatrixBTextBox = 4

    m_numRowMatriCTextBox = m_numRowMatricATextBox
    m_numColMatriCTextBox = m_numColMatrixBTextBox

    m_tbLength = 38
    m_tbHeight = 20

    Dim adistance As Integer = 50
    ' = length of A + a distance
    m_startPointMatrixB.X = m_startPoint.X + (m_numColMatricATextBox) * _
                                             (m_tbLength + 2) - 2 + adistance
    m_startPointMatrixB.Y = m_startPoint.Y

    Dim lbMatrixBlength As Integer = (m_numColMatrixBTextBox) * (m_tbLength + 2) - 2
    m_FormLength = m_startPointMatrixB.X + lbMatrixBlength + adistance
    m_FormHeight = 500

    arrTextBoxMatrixA = New TextBox(m_numRowMatricATextBox, m_numColMatricATextBox) {}
    db_MatrixA = New Double(m_numRowMatricATextBox, m_numColMatricATextBox) {}

    arrTextBoxMatrixB = New TextBox(m_numRowMatrixBTextBox, m_numColMatrixBTextBox) {}
    db_MatrixB = New Double(m_numRowMatrixBTextBox, m_numColMatrixBTextBox) {}

    arrTextBoxMatrixC = New TextBox(m_numRowMatriCTextBox, m_numColMatriCTextBox) {}
    db_MatrixC = New Double(m_numRowMatriCTextBox, m_numColMatriCTextBox) {}

    SetProperty()

  End Sub

  Private Sub btSetNumber_Click(ByVal sender As System.Object, _
```

```
          ByVal e As System.EventArgs) Handles btSetNumber.Click

Dim rowA As Integer
Dim colA As Integer

Dim rowB As Integer
Dim colB As Integer

Try
  rowA = Convert.ToInt32(tbMatrixA_row.Text)
  colA = Convert.ToInt32(tbMatrixA_col.Text)

  rowB = Convert.ToInt32(tbMatrixB_row.Text)
  colB = Convert.ToInt32(tbMatrixB_col.Text)

Catch

  MessageBox.Show("Please enter the number")
  Return
End Try

' update row and col of A & B
m_numRowMatricATextBox = rowA
m_numColMatricATextBox = colA

m_numRowMatrixBTextBox = m_numColMatricATextBox
m_numColMatrixBTextBox = colB

m_numRowMatriCTextBox = m_numRowMatricATextBox
m_numColMatriCTextBox = m_numColMatrixBTextBox

'resize of arrays
ReDim arrTextBoxMatrixA(rowA, colA)
ReDim arrTextBoxMatrixB(rowB, colB)
ReDim arrTextBoxMatrixC(rowA, colB)
```

```
    SetProperty()

  End Sub

  Private Sub btCalculation_Click(ByVal sender As System.Object, _
                                  ByVal e As System.EventArgs) Handles btCalculation.Click

    '////////// get matrix A /////////////
    Try
      GetMatrixA()
    Catch

      MessageBox.Show("We have wrong data")
      Return
    End Try

    '////////// get matrix B /////////////
    Try
      GetMatrixB()
    Catch

      MessageBox.Show("We have wrong data")
      Return
    End Try

    '/////////////// calculate matrix C ////////////
    GetMatrixC()

  End Sub

  Public Sub GetMatrixA()

    Dim rowA As Integer
    Dim colA As Integer

    '//////////////////
```

```
  Try
    rowA = Convert.ToInt32(tbMatrixA_row.Text)
    colA = Convert.ToInt32(tbMatrixA_col.Text)

  Catch
    MessageBox.Show("Please enter the number")
    Return

  End Try

  '//////////////////
  Dim i, j As Integer

  Try
    For i = 0 To (m_numRowMatricATextBox - 1)
      For j = 0 To (m_numColMatricATextBox - 1)

        db_MatrixA(i, j) = Convert.ToDouble(arrTextBoxMatrixA(i, j).Text)

      Next
    Next

  Catch
    MessageBox.Show("Please enter the number in Matrix A")
    Return
  End Try

End Sub

Public Sub GetMatrixB()

  Dim rowB As Integer
  Dim colB As Integer

  '//////////////////
  Try
```

```
      rowB = Convert.ToInt32(tbMatrixB_row.Text)
      colB = Convert.ToInt32(tbMatrixB_col.Text)

    Catch
      MessageBox.Show("Please enter the number")
      Return
    End Try

    '//////////////////

    Dim i As Integer
    Dim j As Integer

    Try
      For i = 0 To (m_numRowMatrixBTextBox - 1)

        For j = 0 To (m_numColMatrixBTextBox - 1)

          db_MatrixB(i, j) = Convert.ToDouble(arrTextBoxMatrixB(i, j).Text)
        Next
      Next

    Catch
      MessageBox.Show("Please enter the number in Matrix B")
      Return

    End Try

  End Sub

  Public Sub GetMatrixC()

    ' calculate matrix multiplication

    Dim mw_A As MWNumericArray = New MWNumericArray(db_MatrixA)
```

```
  Dim mw_B As MWNumericArray = New MWNumericArray(db_MatrixB)
  Dim mw_C As MWNumericArray = Nothing

  ' call an implemental function
  Dim obj As MatrixComputationsNameSpace.MatrixComputations = _
          New MatrixComputationsNameSpace.MatrixComputations()

  mw_C = obj.mymtimes(mw_A, mw_B)

  ' convert back to double
  db_MatrixC = mw_C.ToArray(MWArrayComponent.Real)

  ' display to GUI
  Dim i, j As Integer
  For i = 0 To (m_numRowMatriCTextBox - 1)
    For j = 0 To (m_numColMatriCTextBox - 1)

      arrTextBoxMatrixC(i, j).Text = db_MatrixC(i, j).ToString()

    Next
  Next

  MessageBox.Show("It has done")

  mw_A.Dispose()
  mw_B.Dispose()
  mw_C.Dispose()

End Sub

Public Sub SetProperty()

  Me.Controls.Clear()

  m_tbLength = 38
  m_tbHeight = 20
```

```
Dim adistance As Integer = 50
' = length of A + a distance
m_startPointMatrixB.X = m_startPoint.X + (m_numColMatricATextBox) * _
                                         (m_tbLength + 2) - 2 + adistance
m_startPointMatrixB.Y = m_startPoint.Y

Dim lbMatrixBlength As Integer = (m_numColMatrixBTextBox) * (m_tbLength + 2) - 2

m_startPointMatrixC.X = m_startPoint.X
Dim aheight_A As Integer = m_startPoint.Y + (m_numRowMatricATextBox) * _
                                            (m_tbHeight + 2) + 3 * adistance
Dim aheight_B As Integer = m_startPoint.Y + (m_numRowMatrixBTextBox) * _
                                            (m_tbHeight + 2) + 3 * adistance

Dim aheight As Integer = Math.Max(aheight_A, aheight_B)
m_startPointMatrixC.Y = aheight

' reset form size
InitializeComponent()
tbMatrixA_row.Text = m_numRowMatricATextBox.ToString()
tbMatrixA_col.Text = m_numColMatricATextBox.ToString()

tbMatrixB_row.Text = m_numRowMatrixBTextBox.ToString()
tbMatrixB_col.Text = m_numColMatrixBTextBox.ToString()

m_numRowMatriCTextBox = m_numRowMatricATextBox
m_numColMatriCTextBox = m_numColMatrixBTextBox

SetLabelMatrixA()
SetLabelMatrixB()
SetLabelMatrixC()

' set calculation button
Me.btCalculation.Location = New System.Drawing.Point(m_startPoint.X, aheight - 100)
```

```
  SetTextBoxMatrixA()
  SetTextBoxMatrixB()
  SetTextBoxMatrixC()

  m_FormLength = m_startPointMatrixB.X + lbMatrixBlength + adistance
  m_FormHeight = m_startPointMatrixC.Y + m_numRowMatriCTextBox * (m_tbHeight + 2) + adistance

  m_FormLength = Math.Max(544, m_FormLength)
  Me.ClientSize = New System.Drawing.Size(m_FormLength, m_FormHeight)

  Me.Refresh()

End Sub

Public Sub SetLabelMatrixA()

  Dim lbMatrixA As Label = New System.Windows.Forms.Label()

  Dim lbMatrixAlength As Integer = (m_numColMatricATextBox) * (m_tbLength + 2) - 2
  lbMatrixA.Font = New System.Drawing.Font("Microsoft Sans Serif", 9.75F, _
          System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, CByte(0))
  lbMatrixA.Location = New System.Drawing.Point(m_startPoint.X, m_startPoint.Y - 40)
  lbMatrixA.Size = New System.Drawing.Size(lbMatrixAlength, 28)
  lbMatrixA.Name = "lbMatrixA"
  lbMatrixA.Text = "Matrix A"
  lbMatrixA.TextAlign = System.Drawing.ContentAlignment.MiddleCenter

  Me.Controls.Add(lbMatrixA)

End Sub

Public Sub SetLabelMatrixB()

  Dim lbMatrixB As Label = New System.Windows.Forms.Label()
  lbMatrixB.Font = New System.Drawing.Font("Microsoft Sans Serif", 9.75F, _
          System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, CByte(0))
```

```
    ' row B = col A
    Dim lbMatrixBlength As Integer = (m_numColMatrixBTextBox) * (m_tbLength + 2) - 2
    lbMatrixB.Location = New System.Drawing.Point( _
                                    m_startPointMatrixB.X, m_startPoint.Y - 40)
    lbMatrixB.Size = New System.Drawing.Size(lbMatrixBlength, 28)
    lbMatrixB.Name = "lbMatrixB"
    lbMatrixB.Text = "Matrix B"

    lbMatrixB.TextAlign = System.Drawing.ContentAlignment.MiddleCenter
    Me.Controls.Add(lbMatrixB)

  End Sub

  Public Sub SetLabelMatrixC()

    Dim lbMatrixC As Label = New System.Windows.Forms.Label()
    lbMatrixC.Font = New System.Drawing.Font("Microsoft Sans Serif", 9.75F, _
            System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, CByte(0))

    ' row B = col A
    Dim lbMatrixClength As Integer = (m_numColMatriCTextBox) * (m_tbLength + 2) - 2
    lbMatrixC.Location = New System.Drawing.Point( _
                                m_startPointMatrixC.X, m_startPointMatrixC.Y - 40)
    lbMatrixC.Size = New System.Drawing.Size(lbMatrixClength, 28)
    lbMatrixC.Name = "lbMatrixC"
    lbMatrixC.Text = "Matrix C"

    lbMatrixC.TextAlign = System.Drawing.ContentAlignment.MiddleCenter
    Me.Controls.Add(lbMatrixC)

  End Sub

  Public Sub SetTextBoxMatrixA()

    Dim i As Integer
```

```
  Dim j As Integer

  For i = 0 To (m_numRowMatricATextBox - 1)
    For j = 0 To (m_numColMatricATextBox - 1)

      arrTextBoxMatrixA(i, j) = New System.Windows.Forms.TextBox()
      arrTextBoxMatrixA(i, j).Location = New System.Drawing.Point( _
          m_startPoint.X + (m_tbLength + 2) * j, m_startPoint.Y + (m_tbHeight + 2) * i)
      arrTextBoxMatrixA(i, j).Name = "textBox1"
      arrTextBoxMatrixA(i, j).Size = New System.Drawing.Size(m_tbLength, m_tbHeight)
      arrTextBoxMatrixA(i, j).TabIndex = i
      arrTextBoxMatrixA(i, j).Text = "0.0"
      Me.Controls.Add(arrTextBoxMatrixA(i, j))

    Next
  Next

End Sub

Public Sub SetTextBoxMatrixB()

  Dim i As Integer
  Dim j As Integer

  For i = 0 To (m_numRowMatrixBTextBox - 1)
    For j = 0 To (m_numColMatrixBTextBox - 1)

      arrTextBoxMatrixB(i, j) = New System.Windows.Forms.TextBox()
      arrTextBoxMatrixB(i, j).Location = New System.Drawing.Point( _
         m_startPointMatrixB.X + (m_tbLength + 2) * j, m_startPointMatrixB.Y + _
                                                        (m_tbHeight + 2) * i)
      arrTextBoxMatrixB(i, j).Name = "textBox2"
      arrTextBoxMatrixB(i, j).Size = New System.Drawing.Size(m_tbLength, m_tbHeight)
      arrTextBoxMatrixB(i, j).TabIndex = i
      arrTextBoxMatrixB(i, j).Text = "0.0"
      Me.Controls.Add(arrTextBoxMatrixB(i, j))
```

```
    Next
  Next

End Sub

Public Sub SetTextBoxMatrixC()

  Dim i As Integer
  Dim j As Integer

  For i = 0 To (m_numRowMatriCTextBox - 1)
    For j = 0 To (m_numColMatriCTextBox - 1)

      arrTextBoxMatrixC(i, j) = New System.Windows.Forms.TextBox()
      arrTextBoxMatrixC(i, j).Location = New System.Drawing.Point( _
        m_startPointMatrixC.X + (m_tbLength + 2) * j, m_startPointMatrixC.Y + _
                                                    (m_tbHeight + 2) * i)
      arrTextBoxMatrixC(i, j).Name = "textBox2"
      arrTextBoxMatrixC(i, j).Size = New System.Drawing.Size(m_tbLength, m_tbHeight)
      arrTextBoxMatrixC(i, j).TabIndex = i
      arrTextBoxMatrixC(i, j).Text = "0.0"
      arrTextBoxMatrixC(i, j).ReadOnly = True
      Me.Controls.Add(arrTextBoxMatrixC(i, j))

    Next
  Next

End Sub

Private Sub tbMatrixA_col_TextChanged(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles tbMatrixA_col.TextChanged

  tbMatrixB_row.Text = tbMatrixA_col.Text

End Sub
```

```
  '////////////////////
  '////////////////////
  '////////////////////
End Class
```

end code

Part III:
Plotting MATLAB Graphics Figures In VB .NET

Chapter 15

Using MATLAB Graphics In VB Functions

In this chapter we will generate a class ***GeneratingGraphics*** from three common M-files to plot MATLAB 2D Graphics figures. The generated functions of this class will be used in VB .Net version 2008 project to plot 2D figures.

The procedure to create the class ***GeneratingGraphics*** is the same as the procedure in Chapter 2. We will write the M-files as shown below to generate that class.

simplePlot.m

```
function simplePlot(x, y, strColor)

plot(x, y, strColor) ;

grid on ;
```

multiPlotForArrays.m

```
function multiPlotForArrays(t, y1, y2, y3, y4, ...
                                strTitle, strxlabel, strylabel )

hold on ;
```

```
Hplot1 =  plot (t, y1) ;
Hplot2 =  plot (t, y2) ;
Hplot3 =  plot (t, y3) ;
Hplot4 =  plot (t, y4) ;

set(Hplot1, 'Color', 'r', 'Marker', '*', 'LineStyle', '-' )  ;
set(Hplot2, 'Color', 'b', 'Marker', '^', 'LineStyle', ':' )  ;
set(Hplot3, 'Color', 'k', 'Marker', 'o', 'LineStyle', '--')  ;
set(Hplot4, 'Color', 'g', 'Marker', 'o', 'LineStyle', '--')  ;

string1 = 'plot 1' ;
string2 = 'plot 2' ;
string3 = 'plot 3' ;
string4 = 'plot 4' ;

pos = 'SouthEast' ;

Hlegend = legend(string1, string2, string3, string4, 'Location', pos) ;
set(Hlegend, 'FontWeight', 'bold') ;
legend('boxon') ;

hold off ;

%title('Figure Legends') ;
 title(strTitle) ;

%xlabel('x') ;
 xlabel(strxlabel) ;

%ylabel('y') ;
 ylabel(strylabel) ;
```

```
grid on ;
```

multiPlotForFuncs.m

```
function multiPlotForFuncs(strPlot1, colorPlot1, markerPlot1, ...
                           strPlot2, colorPlot2, markerPlot2, ...
                           strPlot3, colorPlot3, markerPlot3, ...
                           strTitle, strxlabel , strylabel )

t = linspace(0, 2*pi, 20) ;

% y1 = cos(t)           ;
% y2 = cos(t + pi/3)    ;
% y3 = cos(t + 2*pi/3)  ;

f1 = inline(strPlot1)   ;
y1 = feval( f1, t )     ;

f2 = inline(strPlot2)   ;
y2 = feval( f2, t )     ;

f3 = inline(strPlot3)   ;
y3 =  feval( f3, t )    ;

hold on ;

Hplot1 =  plot (t, y1) ;
Hplot2 =  plot (t, y2) ;
Hplot3 =  plot (t, y3) ;

% set(Hplot1, 'Color', 'r', 'Marker', '*', 'LineStyle', '-')    ;
% set(Hplot2, 'Color', 'b', 'Marker', '^', 'LineStyle', 'none') ;
% set(Hplot3, 'Color', 'k', 'Marker', 'o', 'LineStyle', '--')   ;
%
```

```
 set(Hplot1, 'Color', colorPlot1, 'Marker', markerPlot1, 'LineStyle', '-')      ;
 set(Hplot2, 'Color', colorPlot2, 'Marker', markerPlot2, 'LineStyle', 'none')   ;
 set(Hplot3, 'Color', colorPlot3, 'Marker', markerPlot3, 'LineStyle', '--')     ;

string1 = 'plot 1' ;
string2 = 'plot 2' ;
string3 = 'plot 3' ;

pos = 'SouthEast' ;

Hlegend = legend(string1, string2, string3, 'Location', pos) ;
set(Hlegend, 'FontWeight', 'bold') ;
legend('boxon') ;

hold off ;

%title('Figure Legends') ;
 title(strTitle) ;

%xlabel('x') ;
 xlabel(strxlabel) ;

%ylabel('y') ;
 ylabel(strylabel) ;

grid on ;
```

mytextread.m

```
function varargout = mytextread(fileName, colNum, mydelimiter)

  switch colNum
```

```
    case 2
    [A1, A2] = textread(fileName, '%f%f','delimiter', mydelimiter) ;
    varargout{1} = A1 ; varargout{2} = A2 ;

    case 3
    [A1, A2, A3] = textread(fileName, '%f%f%f', 'delimiter', mydelimiter) ;
    varargout{1} = A1 ; varargout{2} = A2 ; varargout{3} = A3 ;

    case 4
    [A1, A2, A3, A4] = textread(fileName, '%f%f%f%f', 'delimiter', mydelimiter) ;
    varargout{1} = A1 ; varargout{2} = A2 ; varargout{3} = A3 ;
    varargout{4} = A4 ;

    case 5
    [A1, A2, A3, A4, A5] = textread(fileName, '%f%f%f%f%f', 'delimiter', mydelimiter) ;
    varargout{1} = A1 ; varargout{2} = A2 ; varargout{3} = A3 ;
    varargout{4} = A4 ; varargout{5} = A5 ;

    otherwise
        disp('This function reads files with max colums = 5.');

end
```

The following procedure to create the class ***GeneratingGraphics*** is the same as the procedure in Chapter 2 as follows:

1. Write the command *deploytool* in MATLAB `Command Prompt` to generate a dll file `GeneratingGraphicsNameSpace.dll` that contains the class ***GeneratingGraphics*** (see Fig.15.1 and Fig.15.2).
2. Create a regular VB .NET project in Console Application of Microsoft Visual Studio 2008.
3. Copy the file `GeneratingGraphicsNameSpace.dll` and put it in the same folder `Module1.vb`

4. On Menu, click ***Project***, ***Add Reference***. On the dialog ***Add Reference***, click the tab ***Browse***, then go to the ***distrib*** folder to choose the file `GeneratingGraphicsNameSpace.dll`, and then click OK (see Fig. 15.3).

5. In the project menu, click again **Project**, **Add Reference**, the project will pop up a dialog to choose a reference.

6. Click on tab **.Net**, **MathWorks, .NET MWArray API**. Then the project will add MATLAB array wrapper classes for .NET, MWArray into the VB project.

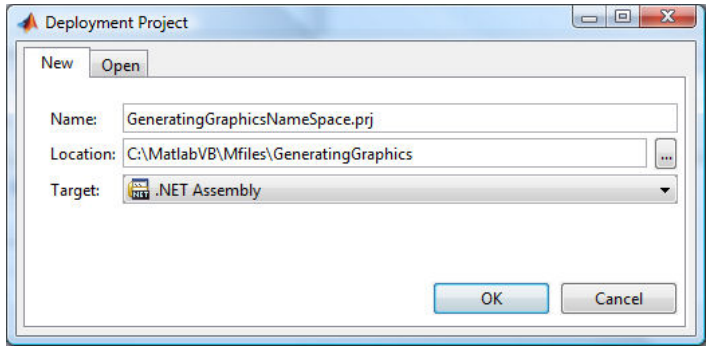


Figure 15.1: Deployment project for ***GeneratingGraphics***

15.1 Using MATLAB Graphics to Plot a Simple 2D Figure

Problem 1

Plot a figure from two arrays with black color :

```
X =   11.1,     22.2,       33.3  ;
Y =   110.1,    220.2,      330.3 ;
```

The code shown to solve Problem 1 is using the function `simplePlot.m`.

Listing code

```
Public Sub SimplePlot()

  Dim db_vectorX() As Double = New Double() {11.1, 22.2, 33.3}
  Dim db_vectorY() As Double = New Double() {110.1, 220.2, 330.3}
```

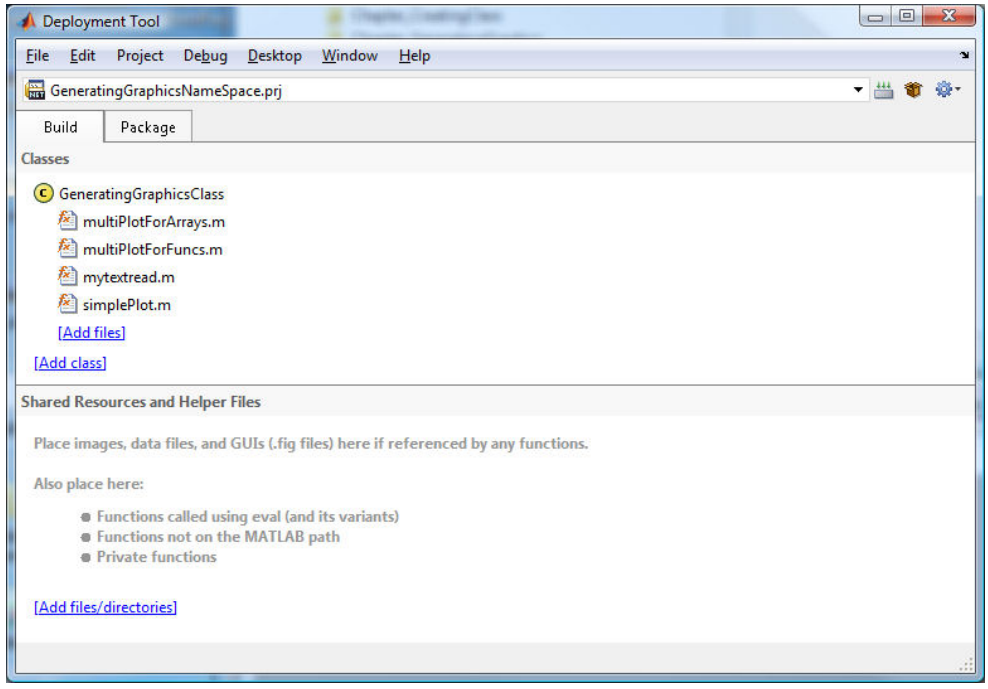


Figure 15.2: Adding M-files in deployment project for ***GeneratingGraphics***

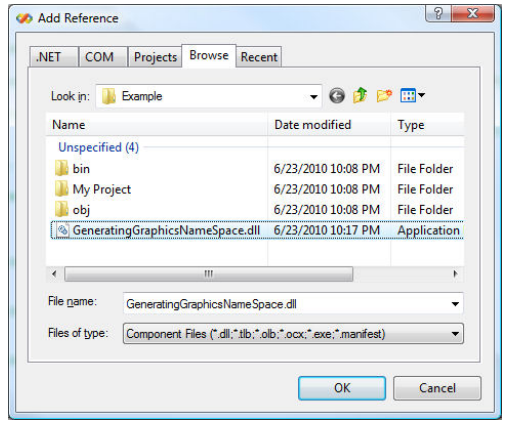


Figure 15.3: Add the reference `GeneratingGraphicsNameSpace.dll` in the VB project

```
  Dim strColor As String = "k" ' k: black, b: blue, r: red

  ' declare mwArray variables
  Dim mw_X As MWNumericArray = New MWNumericArray(db_vectorX)
  Dim mw_Y As MWNumericArray = New MWNumericArray(db_vectorY)

  ' set color, see color symbols in MATLAB Graphics
  Dim mw_strColor As MWCharArray = New MWCharArray(strColor)

  ' call the implemental function
  Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()
  objMatlab.simplePlot(mw_X, mw_Y, mw_strColor)

  objMatlab.WaitForFiguresToDie()

  'Typically you use WaitForFiguresToDie when:
  '. There are one or more figures open that were created
  '  by a .NET component created by the builder.
  '. The method that displays the graphics requires user input before continuing.
  '. The method that calls the figures was called from main() in a console program.

  'When WaitForFiguresToDie is called, execution of the calling program is blocked
  'if any figures created by the calling object remain open.

  mw_strColor.Dispose()
  mw_X.Dispose()
  mw_Y.Dispose()

End Sub
```

end code

The program generates a figure as shown in Fig. 15.4.

The following code shows another simple plot that uses the function `simplePlot.m`.

Listing code

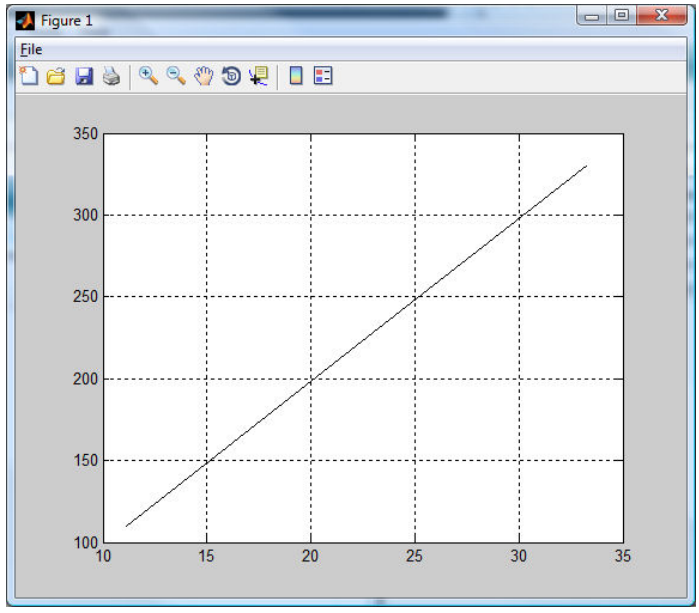


Figure 15.4: A simple plot

```
Public Sub AnotherSimplePlot(ByVal arrayX() As Double, ByVal arrayY() As Double)

  Dim strColor As String = "b" ' k: black, b: blue, r: red

  ' declare mwArray variables
  Dim mw_X As MWNumericArray = New MWNumericArray(arrayX)
  Dim mw_Y As MWNumericArray = New MWNumericArray(arrayY)

  ' set color, see color symbols in MATLAB Graphics
  Dim mw_strColor As MWCharArray = New MWCharArray(strColor)

  ' call the implemental function
  Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()

  objMatlab.simplePlot(mw_X, mw_Y, mw_strColor)
  objMatlab.WaitForFiguresToDie()

  mw_strColor.Dispose()
```

```
  mw_X.Dispose()
  mw_Y.Dispose()

End Sub
```

end code

15.2 Using MATLAB Graphics to Plot a 2D Figure with Data From a File

Problem 2

Plot a figure with red color from two columns of a data file **PlottingFile.txt** as shown below.
PlottingFile.txt:

```
1.1   100.1
2.2   200.2
3.3   300.3
4.4   400.4
5.5   500.5
6.6   600.6
7.7   700.7
8.8   800.8
9.9   900.9
```

The following shows the code to solve Problem 2 by using the MATLAB M-files functions `mytextread.m` and `simplePlot.m` (shown at the beginning of this chapter).

The program generates a figure as shown in Fig. 15.5.

Listing code

```
Public Sub PloPlotDataFromFile()

  ' Read data from a file
  Dim fileName As String = "PlottingFile.txt"
  Dim mw_ArrayOut() As MWArray = Nothing
```

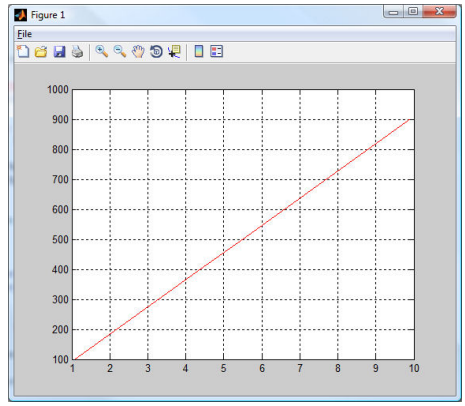


Figure 15.5: Figure from a data file

```
Dim mw_X As MWNumericArray = Nothing
Dim mw_Y As MWNumericArray = Nothing

Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()
Dim delimiter As String = " " ' blank
Dim numCol As Integer = 2

'Value of the first 2 is two of ouput of varargout in mytextread.m
mw_ArrayOut = objMatlab.mytextread(2, fileName, numCol, delimiter)

mw_X = mw_ArrayOut(0)
mw_Y = mw_ArrayOut(1)

'End of Reading data from a file

'Set color, see color symbols in MATLAB Graphics
'Plot graphic
Dim strColor As String = "r"  'k: black, b: blue, r: red
Dim mw_strColor As MWCharArray = New MWCharArray(strColor)
objMatlab.simplePlot(mw_X, mw_Y, mw_strColor)

objMatlab.WaitForFiguresToDie()
```

```
  MWNumericArray.DisposeArray(mw_ArrayOut)
  mw_strColor.Dispose()
  mw_X.Dispose()
  mw_Y.Dispose()

End Sub
```

end code

15.3 Using MATLAB Graphics 2D to Plot Multiple Figures from Mathematical Functions

Similar to the previous section, in this section we will generate multiple figures with data from mathematical functions by using the MATLAB M-file `multiPlotForFuncs.m` (shown at the beginning of this chapter). In this function, we particularly plot three curves. If you would like to plot with another number of curves you could modify this function.

Problem 3

Plot figures from the following functions and their properties:

```
Function 1 = cos(t)
Function 2 = cos(t + pi/3)
Function 3 = cos(t + 2*pi/3)

marker of Plot 1 = *
marker of Plot 2 = ^
marker of Plot 3 = o

color of Plot 1 =  red
color of Plot 2 =  blue
color of Plot 3 =  black

Title   = Figure Legends
x label = My x label
```

```
y label = My y label
```

The following shows the code to solve Problem 3 by using the function `myplotmultiple2D(..)`. This program creates a figure as shown in Fig. 15.6

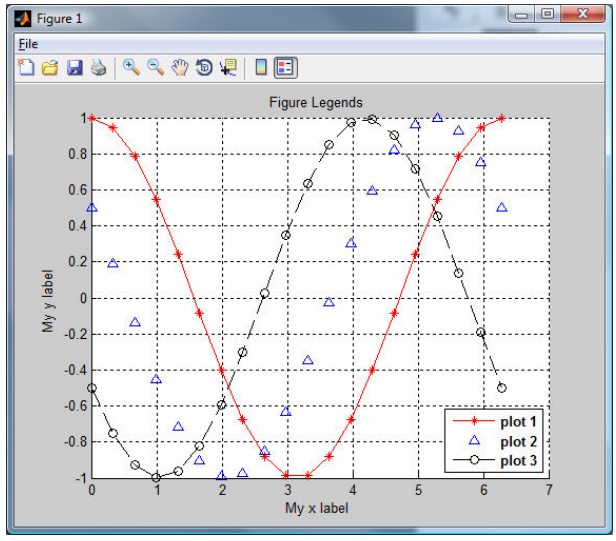


Figure 15.6: Multiple figures from math functions

Listing code

```
Public Sub PlotFromFunctions()

  Dim strfunc1 As String = "cos(t)"
  Dim strfunc2 As String = "cos(t + pi/3)"
  Dim strfunc3 As String = "cos(t + 2*pi/3)"

  Dim strPlot1 As MWCharArray = New MWCharArray(strfunc1)
  Dim strPlot2 As MWCharArray = New MWCharArray(strfunc2)
  Dim strPlot3 As MWCharArray = New MWCharArray(strfunc3)

  Dim markerPlot1 As MWCharArray = New MWCharArray("*")
  Dim markerPlot2 As MWCharArray = New MWCharArray("^")
  Dim markerPlot3 As MWCharArray = New MWCharArray("o")
```

```
  Dim colorPlot1 As MWCharArray = New MWCharArray("r")    ' red
  Dim colorPlot2 As MWCharArray = New MWCharArray("b")    ' blue
  Dim colorPlot3 As MWCharArray = New MWCharArray("k")    ' black

  Dim strTitle As MWCharArray = New MWCharArray("Figure Legends ")
  Dim strxlabel As MWCharArray = New MWCharArray("My x label ")
  Dim strylabel As MWCharArray = New MWCharArray("My y label ")

  ' call the implemental function
  Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()

  'Call the implemental function
  objMatlab.multiPlotForFuncs(strPlot1, colorPlot1, markerPlot1, _
                              strPlot2, colorPlot2, markerPlot2, _
                              strPlot3, colorPlot3, markerPlot3, _
                              strTitle, strxlabel, strylabel)

  objMatlab.WaitForFiguresToDie() 'program will continue after close this figure

  strPlot1.Dispose()
  strPlot2.Dispose()
  strPlot3.Dispose()

  markerPlot1.Dispose()
  markerPlot2.Dispose()
  markerPlot3.Dispose()

  colorPlot1.Dispose()
  colorPlot2.Dispose()
  colorPlot3.Dispose()

  strTitle.Dispose()
  strxlabel.Dispose()
  strylabel.Dispose()

End Sub
```

end code

15.4 Using MATLAB Graphics to Plot Multiple Figures with Data from Arrays

Similar to the previous section, in this section we will generate multiple figures with data from arrays.

Problem 4

Plot multiple figures from the following arrays and their properties:

```
Dim db_vectort() As Double = New Double() {0.1, 0.2, 0.3}

Dim db_vectorY1() As Double = New Double() {11.1, 22.2, 33.3}
Dim db_vectorY2() As Double = New Double() {40.1, 50.2, 60.3}
Dim db_vectorY3() As Double = New Double() {70.1, 80.2, 90.3}
Dim db_vectorY4() As Double = New Double() {100.1, 110.2, 120.3}

Title   = The Title
x label = time
y label = voltage
```

The following shows the code to solve Problem 3 by using the function `myplotmultiple2DFromArray(..)`. The program generates a figure as shown in Fig. 15.7.

Listing code

```
Public Sub MultiPlotFromArrays()

  Dim db_vectort() As Double = New Double() {0.1, 0.2, 0.3}

  Dim db_vectorY1() As Double = New Double() {11.1, 22.2, 33.3}
  Dim db_vectorY2() As Double = New Double() {40.1, 50.2, 60.3}
  Dim db_vectorY3() As Double = New Double() {70.1, 80.2, 90.3}
```

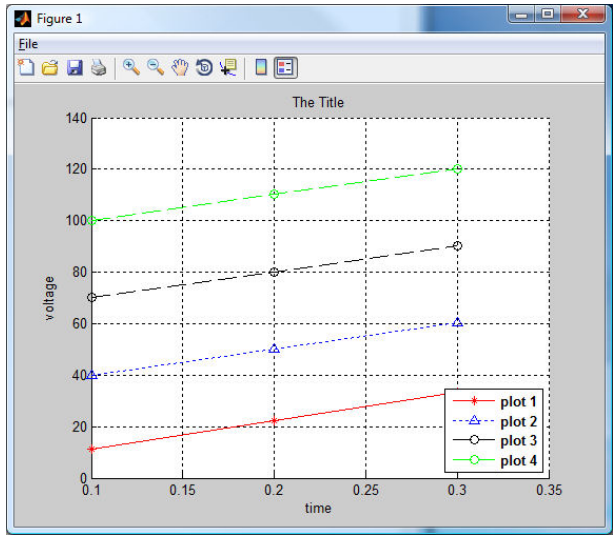


Figure 15.7: Multiple figures from arrays

```
Dim db_vectorY4() As Double = New Double() {100.1, 110.2, 120.3}

' declare mwArray variables
Dim mw_t As MWNumericArray = New MWNumericArray(db_vectort)

Dim mw_y1 As MWNumericArray = New MWNumericArray(db_vectorY1)
Dim mw_y2 As MWNumericArray = New MWNumericArray(db_vectorY2)
Dim mw_y3 As MWNumericArray = New MWNumericArray(db_vectorY3)
Dim mw_y4 As MWNumericArray = New MWNumericArray(db_vectorY4)

' set properties for MATLAB Graphics
Dim mw_Title As MWCharArray = New MWCharArray("The Title")
Dim mw_xlabel As MWCharArray = New MWCharArray("time")
Dim mw_ylabel As MWCharArray = New MWCharArray("voltage")

' call the implemental function
Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()
objMatlab.multiPlotForArrays(mw_t, mw_y1, mw_y2, mw_y3, mw_y4, _
                             mw_Title, mw_xlabel, mw_ylabel)
```

```
  objMatlab.WaitForFiguresToDie() 'program will continue after close this figure

  mw_t.Dispose()

  mw_y1.Dispose()
  mw_y2.Dispose()
  mw_y3.Dispose()
  mw_y4.Dispose()

  mw_Title.Dispose()
  mw_xlabel.Dispose()
  mw_ylabel.Dispose()

End Sub
```

end code

15.5 Using MATLAB Graphics to Plot Multiple Figures with Data from a File

Problem 5

Plot multiple figures from a data file MultiArrays.csv and their properties as follows:

```
0.1,    1,  2,  4,      8
0.2,    2,  4,  8,      16
0.3,    4,  8,  16,     32
0.4,    8,  16, 32,     64
0.5,    16, 32, 64,     28
0.6,    32, 64, 128,    256
```

The following shows the code to solve Problem 5 by using the function `MultiPlotFromFile(..)`. The program generates a figure as shown in Fig. 15.8.

Listing code

```
Public Sub MultiPlotFromFile()
```

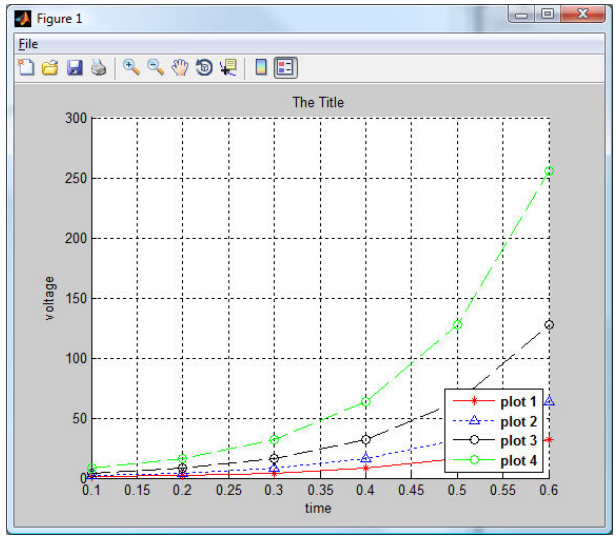


Figure 15.8: Multiple figures from a data file

```
' Read data from a file
Dim fileName As String = "MultiArrays.csv"

Dim mw_ArrayOut() As MWArray = Nothing

Dim mw_Col0 As MWNumericArray = Nothing
Dim mw_Col1 As MWNumericArray = Nothing
Dim mw_Col2 As MWNumericArray = Nothing
Dim mw_Col3 As MWNumericArray = Nothing
Dim mw_Col4 As MWNumericArray = Nothing

Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()

Dim delimiter As String = ","  ' comma
Dim numCol As Integer = 5

' Value of the first 5 is five of ouput of varargout in mytextread.m
mw_ArrayOut = objMatlab.mytextread(5, fileName, numCol, delimiter)
```

```
  mw_Col0 = mw_ArrayOut(0)
  mw_Col1 = mw_ArrayOut(1)
  mw_Col2 = mw_ArrayOut(2)
  mw_Col3 = mw_ArrayOut(3)
  mw_Col4 = mw_ArrayOut(4)

  ' End of Reading data from a file

  ' set properties for MATLAB Graphics
  Dim mw_Title As MWCharArray = New MWCharArray("The Title")
  Dim mw_xlabel As MWCharArray = New MWCharArray("time")
  Dim mw_ylabel As MWCharArray = New MWCharArray("voltage")

  ' Call the implemental function
  objMatlab.multiPlotForArrays(mw_Col0, mw_Col1, mw_Col2, mw_Col3, mw_Col4, _
                               mw_Title, mw_xlabel, mw_ylabel)

  objMatlab.WaitForFiguresToDie() ' program will continue after close this figure

  MWNumericArray.DisposeArray(mw_ArrayOut)
  mw_Col0.Dispose()
  mw_Col1.Dispose()
  mw_Col2.Dispose()
  mw_Col3.Dispose()
  mw_Col4.Dispose()

  mw_Title.Dispose()
  mw_xlabel.Dispose()
  mw_ylabel.Dispose()

End Sub
```

end code

The following is the full code for this chapter.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports GeneratingGraphicsNameSpace

Module Module1

  Sub Main()

    Dim objVB As Example = New Example()

    Console.WriteLine(" Using the plot function. Please wait. ")

    ' Prolem 1: Plot a simple 2D figure
    'objVB.SimplePlot()

    ' Prolem : Plot a 2D figure by passing arrays

    'Dim db_vectorX() As Double = New Double() {0, 50, 100, 150, 200, 250, _
    '                                    300, 350, 400, 450, 500, 550, 600}

    'Dim db_vectorY() As Double = New Double() {0.4, 0.2426, 0.1472, 0.0893, _
    '    0.0541, 0.0328, 0.0199, 0.0121, 0.0073, 0.0044, 0.0027, 0.0016, 0.001}

    'objVB.AnotherSimplePlot(db_vectorX, db_vectorY)

    ' Prolem 2: Plot a 2D figure with data from a file
    ' objVB.PloPlotDataFromFile()


    'Problem 3: Plot multiple figures with data from math functions
    'objVB.PlotFromFunctions()

    'Problem 4: Plot multiple figures with data from arrays
    'objVB.MultiPlotFromArrays()
```

```
  'Problem 5: Plot multiple figures with data from a file
  objVB.MultiPlotFromFile()

End Sub

Public Class Example

  ' ****************************
  Public Sub SimplePlot()

    Dim db_vectorX() As Double = New Double() {11.1, 22.2, 33.3}
    Dim db_vectorY() As Double = New Double() {110.1, 220.2, 330.3}

    Dim strColor As String = "k" ' k: black, b: blue, r: red

    ' declare mwArray variables
    Dim mw_X As MWNumericArray = New MWNumericArray(db_vectorX)
    Dim mw_Y As MWNumericArray = New MWNumericArray(db_vectorY)

    ' set color, see color symbols in MATLAB Graphics
    Dim mw_strColor As MWCharArray = New MWCharArray(strColor)

    ' call the implemental function
    Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()
    objMatlab.simplePlot(mw_X, mw_Y, mw_strColor)

    objMatlab.WaitForFiguresToDie()

    'Typically you use WaitForFiguresToDie when:
    '. There are one or more figures open that were created
    '  by a .NET component created by the builder.
    '. The method that displays the graphics requires user input before continuing.
    '. The method that calls the figures was called from main() in a console program.

    'When WaitForFiguresToDie is called, execution of the calling program is blocked
```

```
    'if any figures created by the calling object remain open.

    mw_strColor.Dispose()
    mw_X.Dispose()
    mw_Y.Dispose()

  End Sub

  ' ****************************
  Public Sub AnotherSimplePlot(ByVal arrayX() As Double, ByVal arrayY() As Double)

    Dim strColor As String = "b" ' k: black, b: blue, r: red

    ' declare mwArray variables
    Dim mw_X As MWNumericArray = New MWNumericArray(arrayX)
    Dim mw_Y As MWNumericArray = New MWNumericArray(arrayY)

    ' set color, see color symbols in MATLAB Graphics
    Dim mw_strColor As MWCharArray = New MWCharArray(strColor)

    ' call the implemental function
    Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()

    objMatlab.simplePlot(mw_X, mw_Y, mw_strColor)
    objMatlab.WaitForFiguresToDie()

    mw_strColor.Dispose()
    mw_X.Dispose()
    mw_Y.Dispose()

  End Sub

  ' ****************************
  Public Sub PloPlotDataFromFile()

    ' Read data from a file
```

```
  Dim fileName As String = "PlottingFile.txt"
  Dim mw_ArrayOut() As MWArray = Nothing
  Dim mw_X As MWNumericArray = Nothing
  Dim mw_Y As MWNumericArray = Nothing

  Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()
  Dim delimiter As String = " " ' blank
  Dim numCol As Integer = 2

  'Value of the first 2 is two of ouput of varargout in mytextread.m
  mw_ArrayOut = objMatlab.mytextread(2, fileName, numCol, delimiter)

  mw_X = mw_ArrayOut(0)
  mw_Y = mw_ArrayOut(1)

  'End of Reading data from a file

  'Set color, see color symbols in MATLAB Graphics
  'Plot graphic
  Dim strColor As String = "r"  'k: black, b: blue, r: red
  Dim mw_strColor As MWCharArray = New MWCharArray(strColor)
  objMatlab.simplePlot(mw_X, mw_Y, mw_strColor)

  objMatlab.WaitForFiguresToDie()

  MWNumericArray.DisposeArray(mw_ArrayOut)
  mw_strColor.Dispose()
  mw_X.Dispose()
  mw_Y.Dispose()

End Sub

' ***************************
Public Sub PlotFromFunctions()

  Dim strfunc1 As String = "cos(t)"
```

```
Dim strfunc2 As String = "cos(t + pi/3)"
Dim strfunc3 As String = "cos(t + 2*pi/3)"

Dim strPlot1 As MWCharArray = New MWCharArray(strfunc1)
Dim strPlot2 As MWCharArray = New MWCharArray(strfunc2)
Dim strPlot3 As MWCharArray = New MWCharArray(strfunc3)

Dim markerPlot1 As MWCharArray = New MWCharArray("*")
Dim markerPlot2 As MWCharArray = New MWCharArray("^")
Dim markerPlot3 As MWCharArray = New MWCharArray("o")

Dim colorPlot1 As MWCharArray = New MWCharArray("r")    ' red
Dim colorPlot2 As MWCharArray = New MWCharArray("b")    ' blue
Dim colorPlot3 As MWCharArray = New MWCharArray("k")    ' black

Dim strTitle As MWCharArray = New MWCharArray("Figure Legends ")
Dim strxlabel As MWCharArray = New MWCharArray("My x label ")
Dim strylabel As MWCharArray = New MWCharArray("My y label ")

' call the implemental function
Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()

'Call the implemental function
objMatlab.multiPlotForFuncs(strPlot1, colorPlot1, markerPlot1, _
                            strPlot2, colorPlot2, markerPlot2, _
                            strPlot3, colorPlot3, markerPlot3, _
                            strTitle, strxlabel, strylabel)

objMatlab.WaitForFiguresToDie() 'program will continue after close this figure

strPlot1.Dispose()
strPlot2.Dispose()
strPlot3.Dispose()

markerPlot1.Dispose()
markerPlot2.Dispose()
```

```
  markerPlot3.Dispose()

  colorPlot1.Dispose()
  colorPlot2.Dispose()
  colorPlot3.Dispose()

  strTitle.Dispose()
  strxlabel.Dispose()
  strylabel.Dispose()

End Sub

'    ' ****************************
Public Sub MultiPlotFromArrays()

  Dim db_vectort() As Double = New Double() {0.1, 0.2, 0.3}

  Dim db_vectorY1() As Double = New Double() {11.1, 22.2, 33.3}
  Dim db_vectorY2() As Double = New Double() {40.1, 50.2, 60.3}
  Dim db_vectorY3() As Double = New Double() {70.1, 80.2, 90.3}
  Dim db_vectorY4() As Double = New Double() {100.1, 110.2, 120.3}

  ' declare mwArray variables
  Dim mw_t As MWNumericArray = New MWNumericArray(db_vectort)

  Dim mw_y1 As MWNumericArray = New MWNumericArray(db_vectorY1)
  Dim mw_y2 As MWNumericArray = New MWNumericArray(db_vectorY2)
  Dim mw_y3 As MWNumericArray = New MWNumericArray(db_vectorY3)
  Dim mw_y4 As MWNumericArray = New MWNumericArray(db_vectorY4)

  ' set properties for MATLAB Graphics
  Dim mw_Title As MWCharArray = New MWCharArray("The Title")
  Dim mw_xlabel As MWCharArray = New MWCharArray("time")
  Dim mw_ylabel As MWCharArray = New MWCharArray("voltage")

  ' call the implemental function
```

```
    Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()
    objMatlab.multiPlotForArrays(mw_t, mw_y1, mw_y2, mw_y3, mw_y4, _
                                 mw_Title, mw_xlabel, mw_ylabel)

    objMatlab.WaitForFiguresToDie() 'program will continue after close this figure

    mw_t.Dispose()

    mw_y1.Dispose()
    mw_y2.Dispose()
    mw_y3.Dispose()
    mw_y4.Dispose()

    mw_Title.Dispose()
    mw_xlabel.Dispose()
    mw_ylabel.Dispose()

  End Sub

  ''    ' ****************************
  Public Sub MultiPlotFromFile()

    ' Read data from a file
    Dim fileName As String = "MultiArrays.csv"

    Dim mw_ArrayOut() As MWArray = Nothing

    Dim mw_Col0 As MWNumericArray = Nothing
    Dim mw_Col1 As MWNumericArray = Nothing
    Dim mw_Col2 As MWNumericArray = Nothing
    Dim mw_Col3 As MWNumericArray = Nothing
    Dim mw_Col4 As MWNumericArray = Nothing

    Dim objMatlab As GeneratingGraphicsClass = New GeneratingGraphicsClass()

    Dim delimiter As String = ","  ' comma
```

```
    Dim numCol As Integer = 5

    ' Value of the first 5 is five of ouput of varargout in mytextread.m
    mw_ArrayOut = objMatlab.mytextread(5, fileName, numCol, delimiter)

    mw_Col0 = mw_ArrayOut(0)
    mw_Col1 = mw_ArrayOut(1)
    mw_Col2 = mw_ArrayOut(2)
    mw_Col3 = mw_ArrayOut(3)
    mw_Col4 = mw_ArrayOut(4)

    ' End of Reading data from a file

    ' set properties for MATLAB Graphics
    Dim mw_Title As MWCharArray = New MWCharArray("The Title")
    Dim mw_xlabel As MWCharArray = New MWCharArray("time")
    Dim mw_ylabel As MWCharArray = New MWCharArray("voltage")

    ' Call the implemental function
    objMatlab.multiPlotForArrays(mw_Col0, mw_Col1, mw_Col2, mw_Col3, mw_Col4, _
                                 mw_Title, mw_xlabel, mw_ylabel)

    objMatlab.WaitForFiguresToDie() ' program will continue after close this figure

    MWNumericArray.DisposeArray(mw_ArrayOut)
    mw_Col0.Dispose()
    mw_Col1.Dispose()
    mw_Col2.Dispose()
    mw_Col3.Dispose()
    mw_Col4.Dispose()

    mw_Title.Dispose()
    mw_xlabel.Dispose()
    mw_ylabel.Dispose()

End Sub
```

```
  End Class

End Module
```

end code

Part IV:
Creating and Using COM From MATLAB Builder for .NET In VB .NET

Chapter 16

Using COM Created From MATLAB Builder for .NET In VB .NET

This chapter describes how to use COM created from MATLAB Builder in a simple VB .NET Console Application.

In Windows Vista and Windows 7, there was an error when I was creating a COM. The versions of MATLAB used in this book are MATLAB 2009(b), MATLAB Compiler 4.11, MATLAB Builder NE 3.0.2, with Microsoft Visual 2008. With these versions, MATLAB Compiler supports only Microsoft .Net Framework SDK 2.0, see

http://www.mathworks.com/support/compilers/R2009b/

The .NET Framework version 2.0 is .NET Framework which is included in .NET Framework 3.5 SP1. Also, Windows Vista and Windows 7 won't let user install .Net Framework 2.0. This causes errors when creating a COM from these MATLAB versions, see

http://social.msdn.microsoft.com/forums/en-US/vssetup/

`thread/60424309-bd78-4ca2-b618-03c4a16123b6`

All COMs in this chapter and the next chapter are created in Windows XP including .NET framework 2.0.

The following are steps to create a COM from an M-file and use it in VB functions:

1. Write an M-file, say `myplus.m`
2. Use MATLAB Builder for .Net to create COM from the M-file `myplus.m` (the steps shown how to do this are in Section 16.1)
3. Use COM in VB functions

16.1 Creating COM From MATLAB Builder for .NET

To create COM from MATLAB:

1. Write a MATLAB M-file, for example `myplus.m`, as follows:

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


y = a + b ;
```

2. Run the following command at the MATLAB prompt: `deploytool` (see Fig. 16.1). The command initializes a dialog to choose a project name as shown in Fig.16.2.

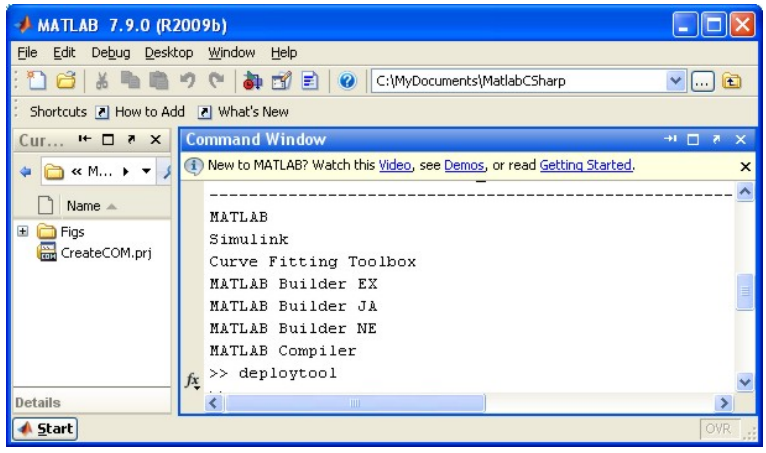


Figure 16.1: Initialization for creating COM

3. Set up the project name, say **CreateCOM.prj**, and target **Generic COM Component** as shown in Fig.16.2. Note that the project name will be the name of **namespace** in the code.

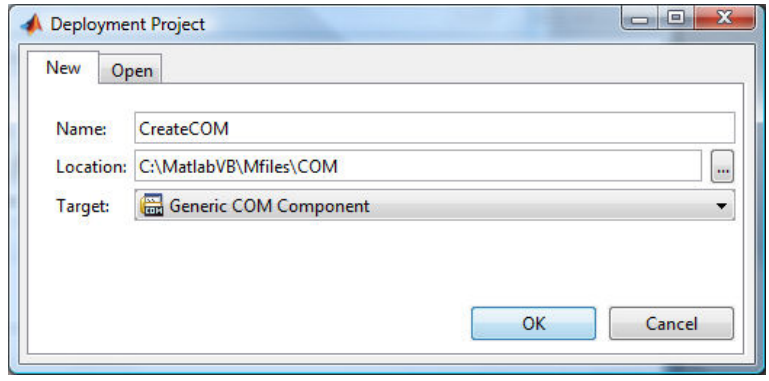


Figure 16.2: Choosing a creating COM project

4. In this dialog (Fig. 16.2) click OK. It should pop up another dialog to set up files and classes as shown in Fig.16.3.

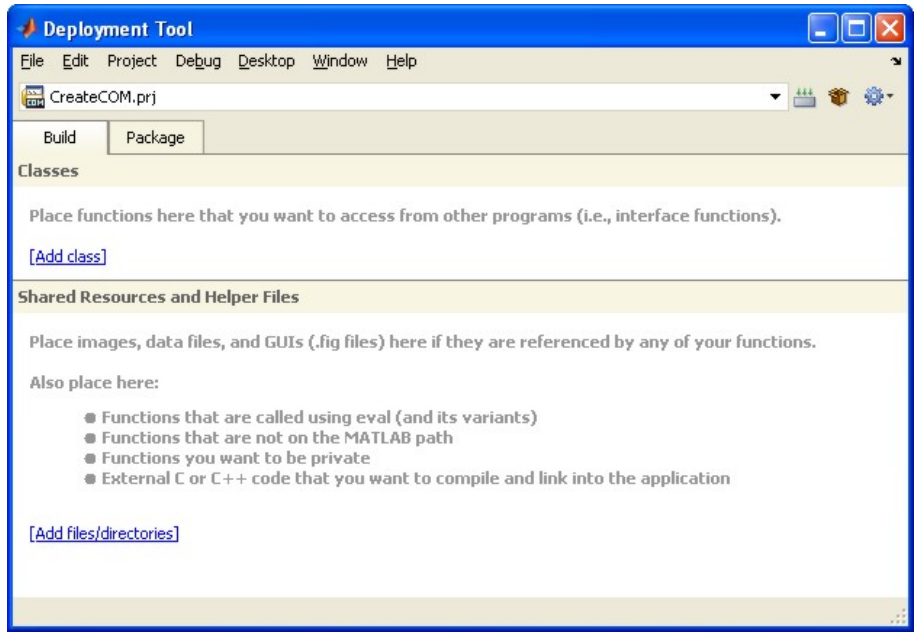


Figure 16.3: Creating COM (continue)

5. In this dialog (Fig.16.3), click on **Add Class** (see Fig.16.4) to add a class name, say **MyCOM**. Note that, the class name that MATLAB will create is **MyCOMClass** (MATLAB adds the suffix **Class** at the end).

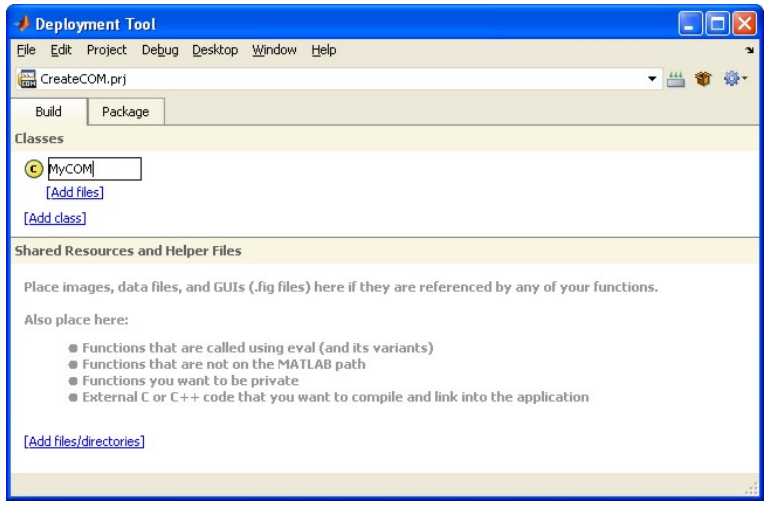


Figure 16.4: Creating COM – Setting a class

6. Click on **Add files** (below **MyCOM**) to choose the M-file `myplus.m` (see Fig.16.5). You can see the file `myplus.m` added to the project as shown in Fig.16.6.

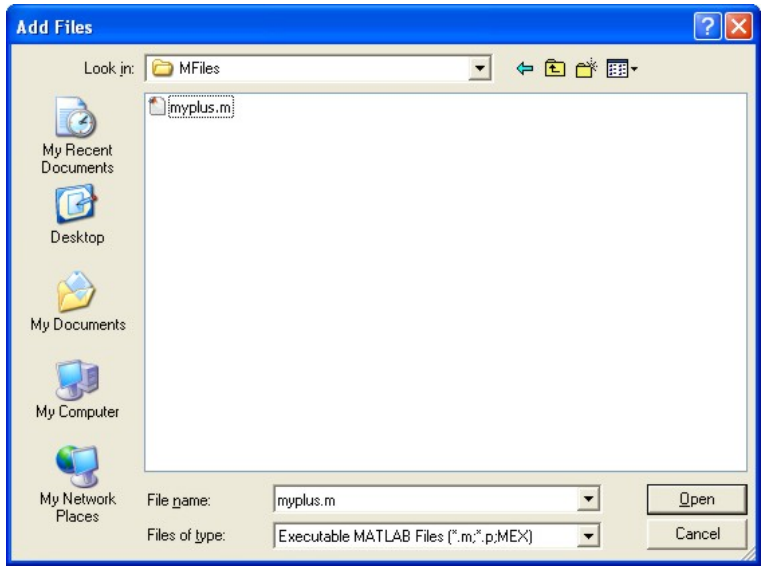


Figure 16.5: Creating COM – Setting a M-file

7. On Fig16.6, click on **Package** to see the *dll* file name, `CreateCOM_1_0.dll` (see Fig.16.7). We'll choose this file to add as reference later in coding.

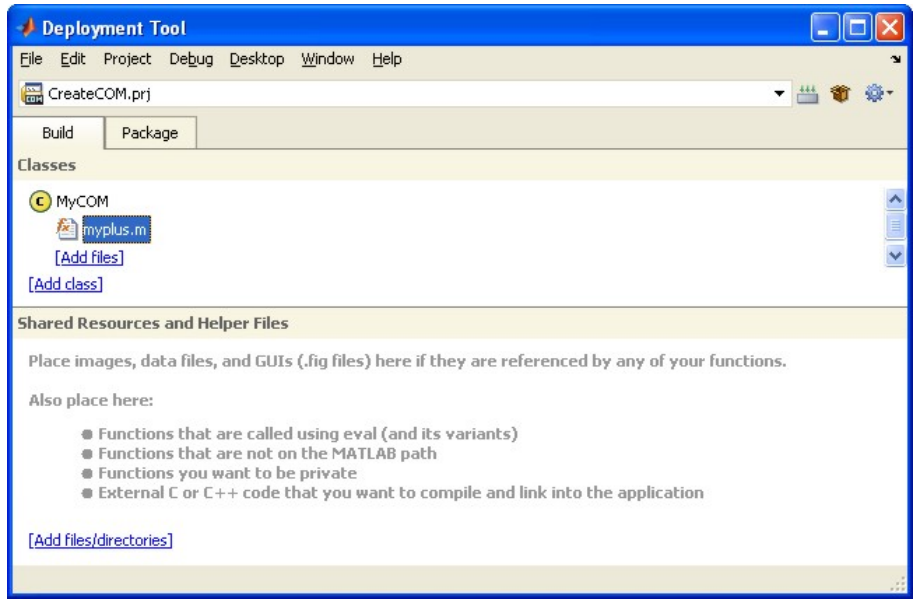


Figure 16.6: Creating COM – Setting a M-file (continue)

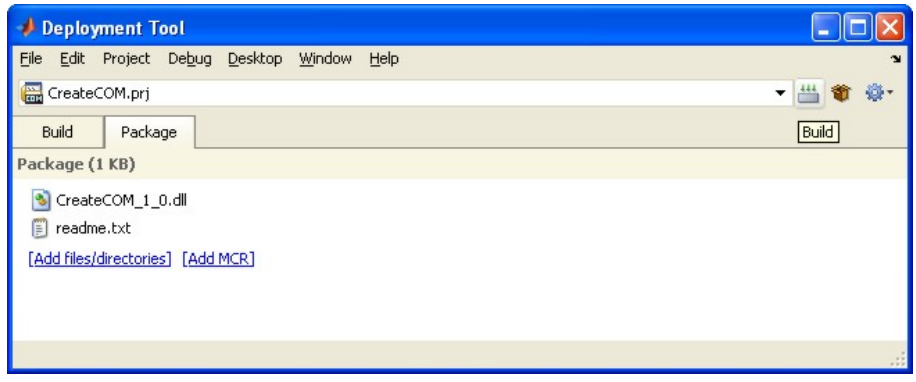


Figure 16.7: Creating COM – The *dll* file name

8. In Fig16.7, click on the **Build** icon (or click on **Project, Build**) to generate the COM. The project will generate a COM based on M-files we add to the project, see Fig.16.8.

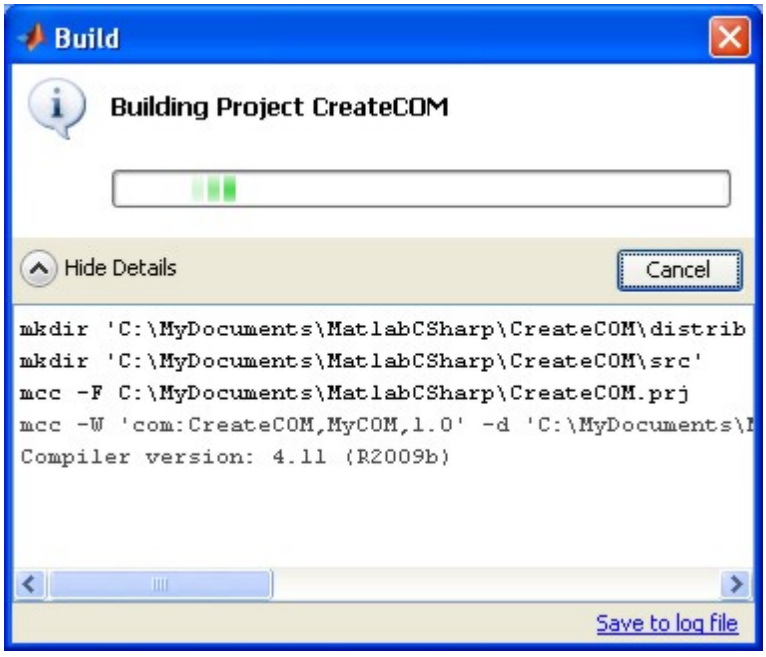


Figure 16.8: Creating COM – Process of building COM

9. The project will create the COM **CreateCOM_1_0.dll** in the directory **...\CreateCOM\distrib** (see Fig.16.9).

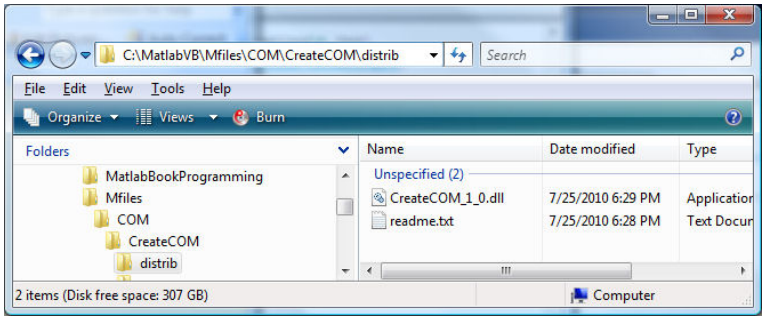


Figure 16.9: Creating COM – Directory of the *dll* file

16.2 Using COM in VB .NET functions

In this section we'll describe the steps to use the created COM, `CreateCOM_1_0.dll`, in a target machine (without MATLAB software) through an example. The steps are:

1. Copy and install the file `MCRInstaller.exe` to your target machine. The contained folder of `MCRInstaller.exe` is shown in the file `readme.txt` in the created directory **...\CreateCOM\distrib**. In my computer, this file is located at **C:\Program Files\MATLAB\R2009b\toolbox\compiler\deploy win32\MCRInstaller.exe**.

2. Create a normal VB. NET Console Application, say **Example**.

3. In the project **Example**, right click on **References** then click **Add Reference**. The Add Reference dialog appears (Fig. 16.10) and then choose **CreateCOM_1_0 Type Library**. Click OK to choose this COM.

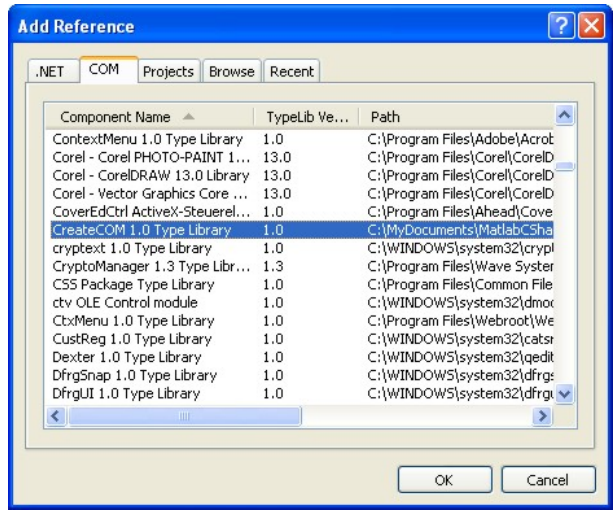


Figure 16.10: Adding COM to References

Note: Before going to the code, we want to mention something here:

1. When we add the COM `CreateCOM_1_0.dll` to **References**, a class is created and named **MyCOMClass**. This name is added a suffix **'Class'** to our predefined class name **MyCOM**.

2. The created function from the M-function `myplus(a, b)` is
   `void myplus(int, ref object, object, object)`.

The following code demonstrates how to call the function `myplus(..)` in a VB function.

Listing code

```
Imports System
Imports CreateCOM

Module Module1

  Sub Main()
    Dim objPro As Example = New Example()

    Dim result As Double = objPro.CalculatePlus(1.2, 3.4)
    Console.WriteLine("Result :{0}", result.ToString())

  End Sub

  Public Class Example

    Public Function CalculatePlus(ByVal db_a As Double, _
                      ByVal db_b As Double) As Double

      Dim objMatlab As MyCOM = New MyCOM()

      Dim obj_A As Object
      obj_A = CObj(db_a)

      Dim obj_B As Object
      obj_B = CObj(db_b)

      Dim obj_C As Object
      obj_C = CObj(0)

      ' call the implemental function
```

```
      objMatlab.myplus(1, obj_C, obj_A, obj_B)

      Dim db_c As Double = CDbl(obj_C)

      Return db_c

    End Function

  End Class

End Module
```

end code

Chapter 17

Creating and Using COM From Multiple M-files In VB .NET

This chapter describes how to use COM created from multiple MATLAB M-files and use them in VB Console Application.
In Windows Vista and Windows 7, there was an error when I was creating a COM. The versions of MATLAB used in this book are MATLAB 2009(b), MATLAB Compiler 4.11, MATLAB Builder NE 3.0.2, with Microsoft Visual 2008. With these versions, MATLAB Compiler supports only Microsoft .Net Framework SDK 2.0, see

http://www.mathworks.com/support/compilers/R2009b/

The .NET Framework version 2.0 is .NET Framework which is included in .NET Framework 3.5 SP1. Also, Windows Vista and Windows 7 won't let user install .Net Framework 2.0, so this causes errors when creating a COM from these MATLAB versions, see

http://social.msdn.microsoft.com/forums/en-US/vssetup/
thread/60424309-bd78-4ca2-b618-03c4a16123b6

The COM in this chapter was created by Windows XP including .NET framework 2.0.

17.1 Creating COM From Multiple MATLAB M-files

We will write the following M-files to create a COM ***MATLABCOMLinear***.

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

mymtimes.m

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

y = a*b ;
```

mytranspose.m

```
function y = mytranspose( x )

y = x' ;
```

mylu.m

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

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

mymldivide.m

```
function x = mymldivide(A, b)
%solve equation Ax = b
x = A\b ;
```

From these M-files we will use MATLAB Builder for .Net to create a COM, named **MATLAB-COMLinear**. The procedure to create this COM and add it to **References** of the VB .NET project is the same as described in the previous chapter, Chapter 16.

1. Create a COM named **MATLABCOMLinear**
2. Adding all M-files in above to the COM
3. MATLAB Builder for .NET will generate a dll file MATLABCOMLinear_1_0.dll
4. Add this COM `MATLABCOMLinear` to **References** of the VB .NET project

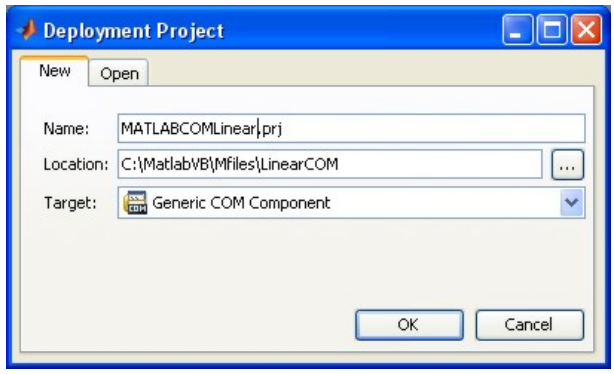


Figure 17.1: Creating COM from Multiple M-files

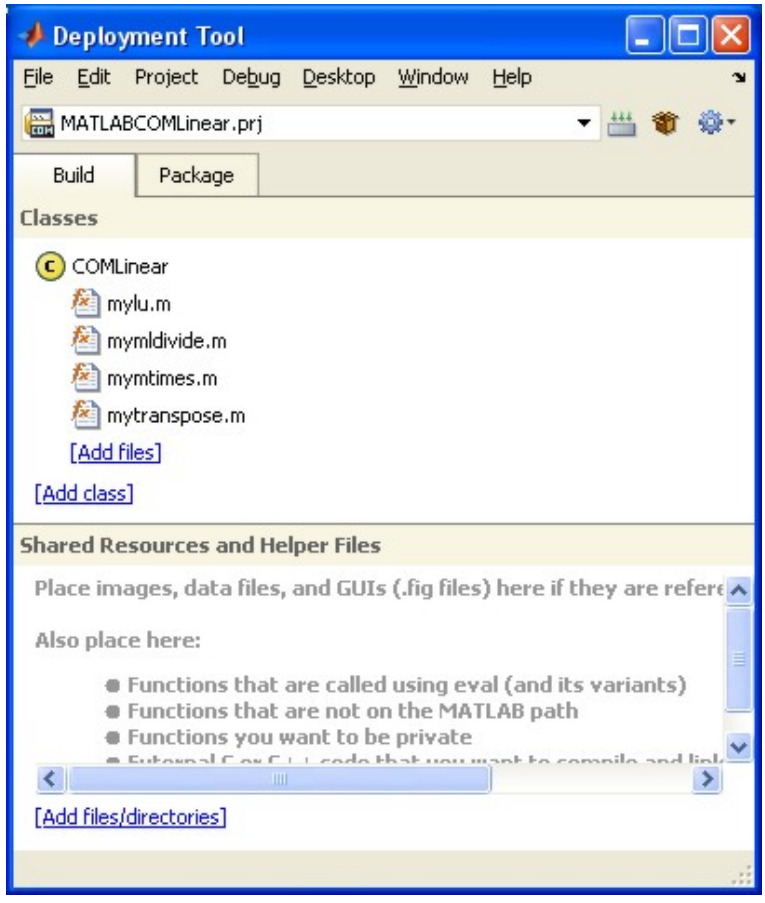


Figure 17.2: Adding M-files to COM

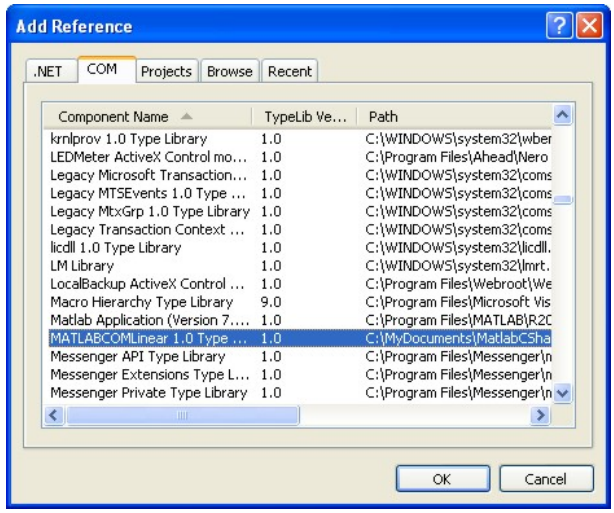


Figure 17.3: Adding M-files to COM (cont.)

17.2 Using COM From MATLAB Builder for .NET To Calculate Matrix Multiplication

Problem 1

input Matrix **A** and **B**

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

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

This following code describes how to use the created COM **MATLABCOMLinear** to calculate the matrix multiplication in Problem 1.

Listing code

```
Public Sub MatrixMultiplication()

  Dim objMatlab As MATLABCOMLinear.COMLinear = _
              New MATLABCOMLinear.COMLinear()

  Dim db_A(,) As Double = {{1.1, 2.2, 3.3, 4.4}, _
                           {5.5, 6.6, 7.7, 8.8}, _
                           {9.9, 10.1, 11.11, 12.12}}

  Dim db_B(,) As Double = {{10, 11}, {12, 13}, {14, 15}, {16, 17}}

  Dim obj_A As Object
  obj_A = CObj(db_A)

  Dim obj_B As Object
  obj_B = CObj(db_B)

  Dim obj_C As Object
  obj_C = CObj(0)

  ' call the implemental function
  objMatlab.mymtimes(1, obj_C, obj_A, obj_B)

  'convert back to Double
  Dim C(,) As Double = New Double(,) {}

  C = MatlabVBNetObj.CopyToMatrixDouble(obj_C)

  'print out
  MatlabVBNetObj.PrintValues(C)

End Sub
```

end code

The code of the subroutine `CopyToMatrixDouble(..)` is at the end of this chapter.

17.3 Using COM from MATLAB COM Builder to Solve Linear System Equations

Problem 2

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

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

output . Finding the solution **x** of linear system equations, $\mathbf{Ax} = \mathbf{b}$

. Finding the lower **L** and upper **U** of the matrix **A**

This following code describes how to use the created COM to solve Problem 2.

Listing code

```
Imports System
Imports MATLABCOMLinear

Module Module1

  Sub Main()

    Dim objVB As Example = New Example()

    Console.WriteLine("MATLAB Builder for .NET")
    Console.WriteLine("Example of Matrix Computations and Linear Equations")
    Console.WriteLine()

    Console.WriteLine("Matrix Multiplication")
    objVB.MatrixMultiplication()
```

```
  Console.WriteLine()

  Console.WriteLine("Linear System Equation")
  objVB.LinearSystemEquations()

  Console.WriteLine()
  Console.WriteLine("LU decompression")
  objVB.LU_decompression()

End Sub

Public Class Example

  Dim MatlabVBNetObj As MatlabVBNet

  Public Sub New()
    MatlabVBNetObj = New MatlabVBNet
  End Sub

  ' ********************************
  Public Sub MatrixMultiplication()

    Dim objMatlab As MATLABCOMLinear.COMLinear = _
                 New MATLABCOMLinear.COMLinear()

    Dim db_A(,) As Double = {{1.1, 2.2, 3.3, 4.4}, _
                             {5.5, 6.6, 7.7, 8.8}, _
                             {9.9, 10.1, 11.11, 12.12}}

    Dim db_B(,) As Double = {{10, 11}, {12, 13}, {14, 15}, {16, 17}}

    Dim obj_A As Object
    obj_A = CObj(db_A)

    Dim obj_B As Object
    obj_B = CObj(db_B)
```

```
    Dim obj_C As Object
    obj_C = CObj(0)

    ' call the implemental function
    objMatlab.mymtimes(1, obj_C, obj_A, obj_B)

    'convert back to Double
    Dim C(,) As Double = New Double(,) {}

    C = MatlabVBNetObj.CopyToMatrixDouble(obj_C)

    'print out
    MatlabVBNetObj.PrintValues(C)

  End Sub

  ' ********************************
  Public Sub LinearSystemEquations()

    Dim obj_Matrix As MATLABCOMLinear.COMLinear = _
                New MATLABCOMLinear.COMLinear()

    ' Solve general linear system equations Ax = b
    Dim db_A(,) As Double = New Double(,) _
                            {{1.1, 5.6, 3.3}, _
                             {4.4, 12.3, 6.6}, _
                             {7.7, 8.8, 9.9}}

    Dim db_vectorb() As Double = New Double() {12.5, 32.2, 45.6}

    Dim obj_A As Object = CObj(db_A)
    Dim obj_b As Object = CObj(db_vectorb)
    Dim obj_Transb As Object = CObj(0)

    ' to get a comlumn vector b
```

```
  ' note: type of obj_Transb is Double(,)
  obj_Matrix.mytranspose(1, obj_Transb, obj_b)

  Dim obj_x As Object = CObj(0)
  obj_Matrix.myldivide(1, obj_x, obj_A, obj_Transb)

  ' note: type of obj_x is a matrix with column = 1
  Dim aLength As Integer = db_vectorb.Length
  Dim db_x(,) As Double = New Double(,) {}

  db_x = MatlabVBNetObj.CopyToMatrixDouble(obj_x)

  'print out
  MatlabVBNetObj.PrintValues(db_x)

End Sub

' ****************************
Public Sub LU_decompression()

  ' find lower and upper matrixes
  Dim db_A(,) As Double = New Double(,) _
                      {{1.1, 5.6, 3.3}, _
     {4.4, 12.3, 6.6}, _
     {7.7, 8.8, 9.9}}

  Dim obj_A As Object = CObj(db_A)
  Dim obj_L As Object = CObj(0)
  Dim obj_U As Object = CObj(0)
  Dim obj_P As Object = CObj(0)

  ' call an implemental function
  Dim obj_Linear As MATLABCOMLinear.COMLinear = _
               New MATLABCOMLinear.COMLinear()

  obj_Linear.mylu(3, obj_L, obj_U, obj_P, obj_A)
```

```
      Dim db_L(,) As Double = New Double(,) {}
      Dim db_U(,) As Double = New Double(,) {}

      db_L = MatlabVBNetObj.CopyToMatrixDouble(obj_L)
      db_U = MatlabVBNetObj.CopyToMatrixDouble(obj_U)

      ' print out
      Console.WriteLine()
      Console.WriteLine("The lower matrix")
      MatlabVBNetObj.PrintValues(db_L)

      Console.WriteLine()
      Console.WriteLine("The upper matrix")
      MatlabVBNetObj.PrintValues(db_U)

    End Sub

  End Class

End Module

Public Class MatlabVBNet

  Public Sub PrintValues(ByVal myArr As Array)
    Dim myEnumerator As System.Collections.IEnumerator = _
       myArr.GetEnumerator()
    Dim i As Integer = 0
    Dim cols As Integer = myArr.GetLength(myArr.Rank - 1)

    'for row vector or column vector
    If myArr.Rank = 1 Then

      While myEnumerator.MoveNext()
        Console.WriteLine(ControlChars.Tab + "{0}", myEnumerator.Current)
      End While
```

```
  Else
    'for other
    While myEnumerator.MoveNext()
      If i < cols Then
        i += 1
      Else
        Console.WriteLine()
        i = 1
      End If
      Console.Write(ControlChars.Tab + "{0}", myEnumerator.Current)
    End While
  End If

  Console.WriteLine()
End Sub

' ******************************
Public Function CopyToMatrixDouble(ByVal myArr As Object) As Double(,)
  Dim myEnumerator As System.Collections.IEnumerator = _
     myArr.GetEnumerator()

  Dim i As Integer = 0
  Dim j As Integer = 0

  Dim col As Integer = myArr.GetLength(myArr.Rank - 1)
  Dim row As Integer = myArr.GetLength(myArr.Rank - 2)

  'Dim m As Integer
  Dim db_matrix(row - 1, col - 1) As Double

  For i = 0 To row - 1
    For j = 0 To col - 1
      myEnumerator.MoveNext()
      db_matrix(i, j) = myEnumerator.Current
    Next
  Next
```

```
    Return db_matrix
  End Function

  ' ********************************
  Public Function CopyToVectorDouble(ByVal myArr As Object) As Double()
    Dim myEnumerator As System.Collections.IEnumerator = _
       myArr.GetEnumerator()

    Dim i As Integer = 0
    Dim row = myArr.Length
    Dim db_vector(row - 1) As Double

    For i = 0 To row - 1
      myEnumerator.MoveNext()
      db_vector(i) = myEnumerator.Current

    Next

    Return db_vector

  End Function

  ' ********************************
  Public Sub ConvertComplexObjToObjs(ByRef obj_Real As Object, _
              ByRef obj_Imag As Object, ByVal objML As Object)

    Dim check As Boolean = False
    check = objML.GetType().IsCOMObject()

    If (check = True) Then
      'for complex number
      obj_Real = objML.Real()
      obj_Imag = objML.Imag()

    Else
```

```
    'for real number
    ConvertRealObjToDouble(obj_Real, obj_Imag, objML)

  End If

End Sub

' ******************************
Public Sub ConvertRealObjToDouble(ByRef obj_Real As Object, _
           ByRef obj_Imag As Object, ByVal objML As Object)

  'assign values for real terms
  obj_Real = objML

  'assign zeros for all imaginary terms
  'for a vector or a scalar
  If (objML.Rank = 1) Then

    Dim row = objML.Length

    'for a scalar
    If (row <= 1) Then
      Dim db_scalar As Double
      db_scalar = 0
      obj_Imag = CObj(db_scalar)

      'for a vector
    Else
      Dim db_vector(row - 1) As Double
      Dim i As Integer

      For i = 0 To row - 1
        db_vector(i) = 0
      Next

      obj_Imag = CObj(db_vector)
```

```
      End If

    End If

    'for a matrix
    If (objML.Rank = 2) Then

      Dim i As Integer = 0
      Dim j As Integer = 0

      Dim col As Integer = objML.GetLength(objML.Rank - 1)
      Dim row As Integer = objML.GetLength(objML.Rank - 2)

      'Dim m As Integer
      Dim db_matrix(row - 1, col - 1) As Double

      For i = 0 To row - 1
        For j = 0 To col - 1

          db_matrix(i, j) = 0
        Next
      Next

      obj_Imag = CObj(db_matrix)
    End If


  End Sub

End Class
```

end code

Part V:
Using MATLAB API Functions in VB .NET –
Using MATLAB Workspace in VB .NET

Chapter 18

Using MATLAB Workspace in VB .NET Functions

This chapter describes how to use the MATLAB workspace to perform particular tasks from a VB .NET function. When performing tasks in the MATLAB workspace, we need to put inputs from a VB function to the MATLAB workspace and then get outputs from the MATLAB workspace back to the VB function. In this chapter, scalars, vectors, and matrixes are used as the function inputs/outputs in the example codes. It also shows how to put a string to MATLAB workspace and plot graphic from MATLAB workspace.

18.1 Setting Up a Project To Use MATLAB Workspace

In this section we built a VB project to use MATLAB workspace into the project. First of all, we need to set up a VB project following below steps.

1. Create a regular VB project in Console Application
2. Add MATLAB MWArray reference as follows:
   - On project menu, click **Project**, **Add Reference** .
   - The project should pop up a following window, see Fig. 18.1, then click on *.NET* tab, **MathWorks, .NET MWArray API**.
3. Add the other reference MATLAB MLApp as follow:

- On project menu, click **Project**, **Add Reference** .
- The project should pop up a following window, see Fig. 18.2, then click on COM tab, **Matlab Application Type Library**.

4. Add the class **MatlabVBAPI** to the project. This class **MatlabVBAPI** is built based-on the MATLAB class **MLAppClass** and the code of this class **MatlabVBAPI** is at the end of this chapter. Do the following steps:

   - Put the file `MatlabVBAPIVer4p11.vb` (at the end of this chapter) in the same directory of the file `Module1.vb`, see Fig.18.3
   - Add this file to the project by click **Project** on the project menu, then click **Add Existing Item**. The project will pop up a window then choose the file `MatlabVBAPIVer4p11.vb`.

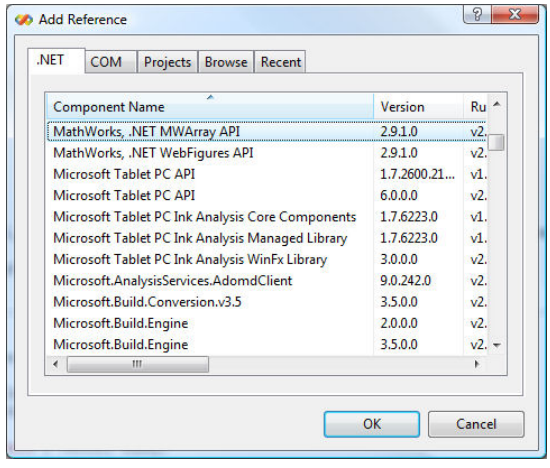


Figure 18.1: Adding MATLAB MWArray reference in VB

The following subroutines will show how to use MATLAB workspace in a VB function to solve problems. The rest of code is at the end of this chapter.

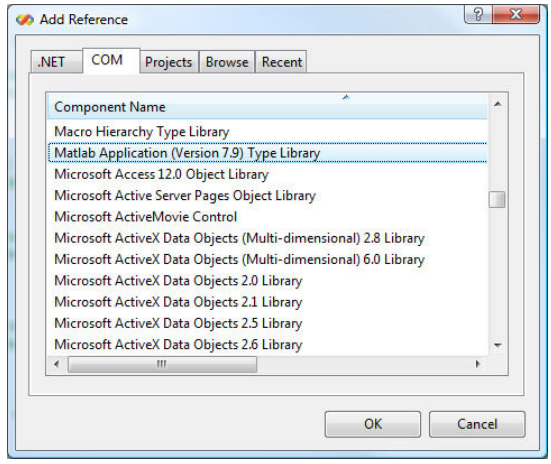


Figure 18.2: Adding MLApp reference in VB

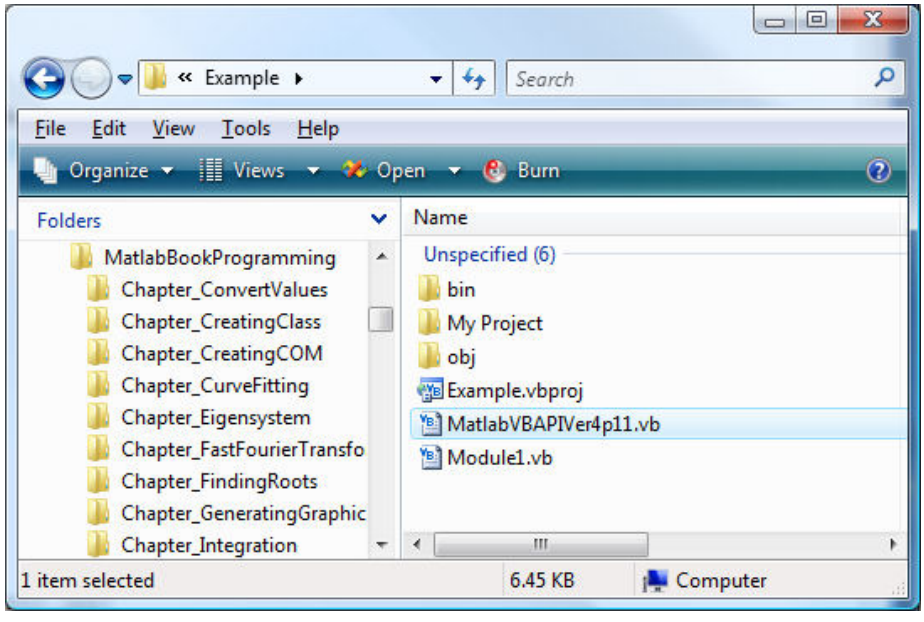


Figure 18.3: Utility file in a VB .NET project

18.2 Calling MATLAB Workspace with Input/Output as a Scalar

In this section, we will use MATLAB functions to solve the following problems.

Problem 1 Put and get numbers between VB and MATLAB workspace.

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 VB function

. Get the result c in the MATLAB workspace and put back to the VB function

The following subroutine is used to solve Problem 1.

The full of code is at the end of this chapter.

Listing code

```
Public Sub SimplePlus()

  Dim db_a As Double = 1.1
  Dim db_b As Double = 2.2

  ObjVBAPI.PutScalarReal("ml_a", db_a)
  ObjVBAPI.PutScalarReal("ml_b", db_b)

  ObjVBAPI.Execute(" ml_c = ml_a + ml_b ; ")

  Dim db_c As Double = 0
  ObjVBAPI.GetScalarReal("ml_c", db_c)

  '''' print out
  Console.WriteLine("The result from simple addition: ")
  Console.WriteLine(db_c.ToString())
End Sub
```

end code

Problem 2 Put and get complex numbers between VB and MATLAB workspace.

input . Two given complex numbers a and b:

real of a = 1.1, imaginary of a = 101.1

real of b = 2.2, imaginary of b = 202.2

output

. Call the MATLAB workspace to perform the task, $c = a + b$, in a VB function

. Get the result of the complex number c in the MATLAB workspace and put back to the VB function

The following subroutine is used to solve Problem 2.

Listing code

```
Public Sub PlusComplexNumbers()

  ' 1. Set number
  Dim aReal As Double = 1.1
  Dim aImag As Double = 101.1

  Dim bReal As Double = 2.2
  Dim bImag As Double = 202.2

  ' 2. Perform tasks in Matlab Workspace
  matlab.PutWorkspaceData("ml_aReal", "base", aReal)
  matlab.PutWorkspaceData("ml_aImag", "base", aImag)
  matlab.Execute(" ml_aComp = complex(ml_aReal, ml_aImag) ; ")

  matlab.PutWorkspaceData("ml_bReal", "base", bReal)
  matlab.PutWorkspaceData("ml_bImag", "base", bImag)
  matlab.Execute(" ml_bComp = complex(ml_bReal, ml_bImag) ; ")

  matlab.Execute(" ml_cComp = ml_aComp + ml_bComp ; " )

  matlab.Execute(" ml_cReal = real(ml_cComp) ; ")
  matlab.Execute(" ml_cImag = imag(ml_cComp) ; ")
```

```
  ' 3. Get data from Matlab Workspace
  Dim obj_cReal As Object = Nothing
  Dim obj_cImag As Object = Nothing
  matlab.GetWorkspaceData("ml_cReal", "base", obj_cReal)
  matlab.GetWorkspaceData("ml_cImag", "base", obj_cImag)

  '4. Print out
  Dim cReal As Double = 0
  Dim cImag As Double = 0

  cReal = CDbl(obj_cReal)
  cImag = CDbl(obj_cImag)

  Console.WriteLine("Result of a complex number :")
  Console.WriteLine("Real Part: {0}", cReal.ToString())
  Console.WriteLine("Imag Part: {0}", cImag.ToString())

End Sub
```

end code

18.3 Calling MATLAB Workspace with Input/Output as a Vector and a Matrix

In this section, we will use MATLAB functions to solve the following problems.

Problem 3

input . A matrix $\mathbf{A}$ and a vector $\mathbf{b}$,

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

output

. Call the MATLAB workspace to perform the tasks as following in a VB function:

a) Finding the solution $\mathbf{x}$ of linear system equations $\mathbf{Ax} = \mathbf{b}$

b) Calculating the upper matrix U with the LU decompression method

c) Getting results of matrixes $\mathbf{U}$ and $\mathbf{L}$ in the MATLAB workspace and converting to VB Double

The following subroutine is used to solve Problem 3.

Listing code

```
Public Sub LinearSolve()

  Dim A(,) As Double = New Double(,) {{1.1, 5.6, 3.3}, {4.4, 12.3, 6.6}, {7.7, 8.8, 9.9}}
  Dim b() As Double = {12.5, 32.2, 45.6}

  ' Put values to MATLAB variables
  ObjVBAPI.PutMatrixReal("ml_A", A)
  ObjVBAPI.PutVectorReal("ml_b", b)

  '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 VB function.
  ObjVBAPI.Execute(" ml_vectorX = mldivide(ml_A, ml_b) ; ")
  ObjVBAPI.Execute(" [ml_L, ml_U, ml_P] = lu( ml_A )   ; ")

  'Get results from MATLAB workspace
  Dim vectorX As Array = Array.CreateInstance(GetType(Double), b.Length)
  ObjVBAPI.GetVectorReal("ml_vectorX", vectorX)

  Dim row As Integer
  Dim col As Integer

  row = A.GetUpperBound(0) - A.GetLowerBound(0) + 1
  col = A.GetUpperBound(1) - A.GetLowerBound(1) + 1

  Dim matrixU As Array = Array.CreateInstance(GetType(Double), row, col)
```

```
  ObjVBAPI.GetMatrixReal("ml_U", matrixU)

  ' Print out
  Console.WriteLine("Linear system: result of the vector solution")
  ObjVBAPI.PrintValues(vectorX)

  Console.WriteLine()
  Console.WriteLine("Linear system: result of the U matrix")
  ObjVBAPI.PrintValues(matrixU)

End Sub
```

end code

Problem 4 Put and get complex vectors (one-dimensional array) between VB and MATLAB workspace.

input

. Two given complex vectors **a** and **b**:

Real values of vector **a** are: 0.1, 1.1, 11.1

Imaginary values of vector **a** are: 100.1, 101.1, 111.1

Real values of vector **b** are: 0.2, 2.2, 22.2

Imaginary values of vector **b** are: 200.2, 202.2, 222.2

output

. Call the MATLAB workspace to perform the task, $\mathbf{c} = \mathbf{a} + \mathbf{b}$, in a VB function

. Get the result of the complex vector **c** in the MATLAB workspace and put back to the VB function

The following subroutine is used to solve Problem 4.

Listing code

```
Public Sub PlusComplexVectors()

  Dim vecAReal() As Double = {0.1, 1.1, 11.1}
  Dim vecAImag() As Double = {100.1, 101.1, 111.1}
```

```
  Dim vecBReal() As Double = {0.2, 2.2, 22.2}
  Dim vecBImag() As Double = {200.2, 202.2, 222.2}

  ObjVBAPI.PutVectorComplex("ml_vecAComp", vecAReal, vecAImag)
  ObjVBAPI.PutVectorComplex("ml_vecBComp", vecBReal, vecBImag)

  ObjVBAPI.Execute(" ml_vecCComp = ml_vecAComp + ml_vecBComp ; ")

  Dim vectorC_Real As Array = Array.CreateInstance(GetType(Double), vecAReal.Length)
  Dim vectorC_Imag As Array = Array.CreateInstance(GetType(Double), vecAImag.Length)

  ObjVBAPI.GetVectorComplex("ml_vecCComp", vectorC_Real, vectorC_Imag)

  ' Print out
  Console.WriteLine("Result of a real vector")
  ObjVBAPI.PrintValues(vectorC_Real)

  Console.WriteLine("Result of an imag vector")
  ObjVBAPI.PrintValues(vectorC_Imag)

End Sub
```

end code

18.4 Generating a MATLAB Graphic from a VB .NET Function

In this section we will directly create a MATLAB graphic by performing a plotting task in the MATLAB workspace .

Problem 5 Plot graphic from two arrays:

```
X = 0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600
Y = 0.4000, 0.2426, 0.1472, 0.0893, 0.0541, 0.0328, 0.0199,
            0.0121, 0.0073, 0.0044, 0.0027, 0.0016,  0.0010
```

Following is the code to create the graphic. The created figure is shown in Fig. 18.4 .

Listing code

```
Public Sub API_Plot()

  Dim X() As Double = {0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600}

  Dim Y() As Double = {0.4, 0.2426, 0.1472, 0.0893, 0.0541, 0.0328, 0.0199, _
         0.0121, 0.0073, 0.0044, 0.0027, 0.0016, 0.001}

  Dim myColor As String = "blue"

  ObjVBAPI.PutVectorReal("ml_X", X)
  ObjVBAPI.PutVectorReal("ml_Y", Y)

  ObjVBAPI.PutString("ml_plotColor", myColor)

  ObjVBAPI.Execute("plot(ml_X, ml_Y, ml_plotColor); ")
  ObjVBAPI.Execute("grid on; ")

  Console.WriteLine("Close the figure and hit Enter key to continue.")
  Console.ReadLine()  ' Keep figure staying

End Sub
```

end code

Remark

- With directly creating the graphic by MATLAB workspace, 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 **System.Console.Read() ;**.

The following is the full code for this chapter.

Listing code

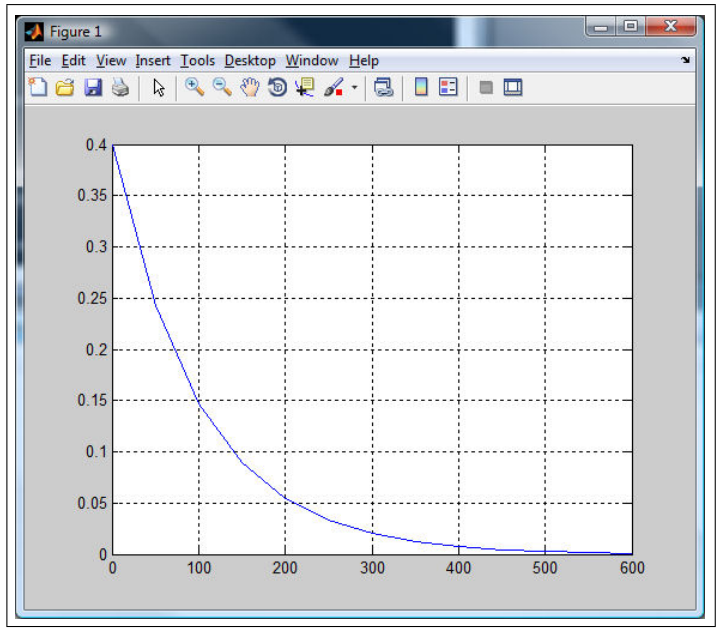


Figure 18.4: The figure is created by using the MATLAB workspace

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Module Module1

  Dim ObjVBAPI As UtilityMatlabVBAPIVer4p11.MatlabVBAPI = _
                                  New UtilityMatlabVBAPIVer4p11.MatlabVBAPI()

  Sub Main()
    Dim objVB As Example = New Example()

    ' I . Scalar
    ' 1. Put scalars in MATLAB Workspace
    ' 2. Get a scalar from MATLAB Workspace
    Console.WriteLine()
    objVB.SimplePlus()
```

```
    Console.WriteLine()
    objVB.PlusComplexNumbers()

    ' II. Vector & Matrix
    ' 1. Put a vector in MATLAB API
    ' 2. Get a vecotr from MATLAB API
    Console.WriteLine()
    objVB.PlusRealVectors()

    Console.WriteLine()
    objVB.PlusComplexVectors()

    ' III. Matrix
    ' 1. Put a matrix in MATLAB  Workspace
    ' 2. Get a matrix from MATLAB Workspace
    Console.WriteLine()
    objVB.PlusRealMatrixes()

    Console.WriteLine()
    objVB.PlusComplexMatrixes()

    Console.WriteLine()
    objVB.LinearSolve()

    'IV. Plot
    ' 1. Put a string in MATLAB Workspace
    ' 2. Plot graphic from MATLAB Workspace
    Console.WriteLine()
    objVB.API_Plot()

  End Sub

  ' ********************************
  ' ********************************
  ' ********************************
  Public Class Example
```

```
' ********************************
Public Sub SimplePlus()

  Dim db_a As Double = 1.1
  Dim db_b As Double = 2.2

  ObjVBAPI.PutScalarReal("ml_a", db_a)
  ObjVBAPI.PutScalarReal("ml_b", db_b)

  ObjVBAPI.Execute(" ml_c = ml_a + ml_b ; ")

  Dim db_c As Double = 0
  ObjVBAPI.GetScalarReal("ml_c", db_c)

  ' print out
  Console.WriteLine("The result from simple addition: ")
  Console.WriteLine(db_c.ToString())

End Sub

' ****************************
Public Sub PlusComplexNumbers()

  ObjVBAPI.PlusComplexNumbers()

End Sub

' ****************************
Public Sub PlusRealVectors()

  Dim vecA() As Double = {0.1, 1.1, 11.1}
  Dim vecB() As Double = {0.2, 2.2, 22.2}

  ObjVBAPI.PutVectorReal("ml_vecA", vecA)
  ObjVBAPI.PutVectorReal("ml_vecB", vecB)
```

```
    ObjVBAPI.Execute(" ml_vecC = ml_vecA + ml_vecB ; ")

    Dim vectorC As Array = Array.CreateInstance(GetType(Double), vecA.Length)

    ObjVBAPI.GetVectorReal("ml_vecC", vectorC)

    ' Print out
    Console.WriteLine("Result of a real vector")
    ObjVBAPI.PrintValues(vectorC)

End Sub

' ****************************
Public Sub PlusComplexVectors()

    Dim vecAReal() As Double = {0.1, 1.1, 11.1}
    Dim vecAImag() As Double = {100.1, 101.1, 111.1}

    Dim vecBReal() As Double = {0.2, 2.2, 22.2}
    Dim vecBImag() As Double = {200.2, 202.2, 222.2}

    ObjVBAPI.PutVectorComplex("ml_vecAComp", vecAReal, vecAImag)
    ObjVBAPI.PutVectorComplex("ml_vecBComp", vecBReal, vecBImag)

    ObjVBAPI.Execute(" ml_vecCComp = ml_vecAComp + ml_vecBComp ; ")

    Dim vectorC_Real As Array = Array.CreateInstance(GetType(Double), vecAReal.Length)
    Dim vectorC_Imag As Array = Array.CreateInstance(GetType(Double), vecAImag.Length)

    ObjVBAPI.GetVectorComplex("ml_vecCComp", vectorC_Real, vectorC_Imag)

    ' Print out
    Console.WriteLine("Result of a real vector")
    ObjVBAPI.PrintValues(vectorC_Real)
```

```
  Console.WriteLine("Result of an imag vector")
  ObjVBAPI.PrintValues(vectorC_Imag)

End Sub

' ****************************
Public Sub PlusRealMatrixes()

  Dim A(,) As Double = New Double(,) {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}, _
                                                        {7.7, 8.8, 9.9}}
  Dim B(,) As Double = New Double(,) {{11, 12, 13}, {14, 15, 16}, {17, 18, 19}}

  ObjVBAPI.PutMatrixReal("ml_matrixA", A)
  ObjVBAPI.PutMatrixReal("ml_matrixB", B)

  ObjVBAPI.Execute(" ml_matrixC = ml_matrixA + ml_matrixB ; ")

  Dim row As Integer
  Dim col As Integer

  row = A.GetUpperBound(0) - A.GetLowerBound(0) + 1
  col = A.GetUpperBound(1) - A.GetLowerBound(1) + 1

  Dim matrixC As Array = Array.CreateInstance(GetType(Double), row, col)
  ObjVBAPI.GetMatrixReal("ml_matrixC", matrixC)

  ' Printer out
  Console.WriteLine("Result of a real matrix")
  ObjVBAPI.PrintValues(matrixC)

End Sub

' ****************************
Public Sub PlusComplexMatrixes()

  Dim matrixA_real(,) As Double = New Double(,) _
```

```
                                {{1.1, 2.2, 3.3}, {4.4, 5.5, 6.6}, {7.7, 8.8, 9.9}}
    Dim matrixA_imag(,) As Double = New Double(,) _
                {{100.1, 200.2, 300.3}, {400.4, 500.5, 600.6}, {700.7, 800.8, 900.9}}


    Dim matrixB_real(,) As Double = New Double(,) _
                                {{11, 12, 13}, {14, 15, 16}, {17, 18, 19}}
    Dim matrixB_imag(,) As Double = New Double(,) _
                                {{111, 112, 113}, {114, 115, 16}, {117, 118, 119}}

    ObjVBAPI.PutMatrixComplex("ml_matrixA", matrixA_real, matrixA_imag)
    ObjVBAPI.PutMatrixComplex("ml_matrixB", matrixB_real, matrixB_imag)
    ObjVBAPI.Execute(" ml_matrixC = ml_matrixA + ml_matrixB ; ")

    Dim row As Integer
    Dim col As Integer

    row = matrixA_real.GetUpperBound(0) - matrixA_real.GetLowerBound(0) + 1
    col = matrixA_real.GetUpperBound(1) - matrixA_real.GetLowerBound(1) + 1

    Dim matrixC_real As Array = Array.CreateInstance(GetType(Double), row, col)
    Dim matrixC_imag As Array = Array.CreateInstance(GetType(Double), row, col)

    ObjVBAPI.GetMatrixComplex("ml_matrixC", matrixC_real, matrixC_imag)

    ' Print out
    Console.WriteLine("Result of a real matrix")
    ObjVBAPI.PrintValues(matrixC_real)

    Console.WriteLine("Result of an imag matrix")
    ObjVBAPI.PrintValues(matrixC_imag)

End Sub

' ****************************
Public Sub LinearSolve()
```

```
Dim A(,) As Double = New Double(,) {{1.1, 5.6, 3.3}, {4.4, 12.3, 6.6}, _
                                                    {7.7, 8.8, 9.9}}
Dim b() As Double = {12.5, 32.2, 45.6}

' Put values to MATLAB variables
ObjVBAPI.PutMatrixReal("ml_A", A)
ObjVBAPI.PutVectorReal("ml_b", b)

'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 VB function.
ObjVBAPI.Execute(" ml_vectorX = mldivide(ml_A, ml_b) ; ")
ObjVBAPI.Execute(" [ml_L, ml_U, ml_P] = lu( ml_A )   ; ")

'Get results from MATLAB workspace
Dim vectorX As Array = Array.CreateInstance(GetType(Double), b.Length)
ObjVBAPI.GetVectorReal("ml_vectorX", vectorX)

Dim row As Integer
Dim col As Integer

row = A.GetUpperBound(0) - A.GetLowerBound(0) + 1
col = A.GetUpperBound(1) - A.GetLowerBound(1) + 1

Dim matrixU As Array = Array.CreateInstance(GetType(Double), row, col)
ObjVBAPI.GetMatrixReal("ml_U", matrixU)

' Print out
Console.WriteLine("Linear system: result of the vector solution")
ObjVBAPI.PrintValues(vectorX)

Console.WriteLine()
Console.WriteLine("Linear system: result of the U matrix")
ObjVBAPI.PrintValues(matrixU)
```

```
    End Sub

    ' ******************************
    Public Sub API_Plot()

      Dim X() As Double = {0, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600}

      Dim Y() As Double = {0.4, 0.2426, 0.1472, 0.0893, 0.0541, 0.0328, 0.0199, _
             0.0121, 0.0073, 0.0044, 0.0027, 0.0016, 0.001}

      Dim myColor As String = "blue"

      ObjVBAPI.PutVectorReal("ml_X", X)
      ObjVBAPI.PutVectorReal("ml_Y", Y)

      ObjVBAPI.PutString("ml_plotColor", myColor)

      ObjVBAPI.Execute("plot(ml_X, ml_Y, ml_plotColor); ")
      ObjVBAPI.Execute("grid on; ")

      Console.WriteLine("Close the figure and hit Enter key to continue.")
      Console.ReadLine()  ' Keep figure staying

    End Sub

  End Class

End Module
```

end code

Utility file ***MatlabCSharpAPIVer4p11.cs***.

Listing code

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
```

```
Imports MLApp

' For MATLAB Version 7.9 (2009b) and MATLAB Compiler 4.11
Namespace UtilityMatlabVBAPIVer4p11
  Class MatlabVBAPI

    Dim matlab As MLApp.MLApp = New MLApp.MLApp()

    'Instantiate MATLAB Engine Interface through COM MLApp

    ''* ****************************
    ''* ********** I. Scalar ********
    ''* ****************************
    Public Sub PutScalarReal(ByVal ml_a As String, ByVal a As Double)
      Dim obj_a As Object = CObj(a)

      matlab.PutWorkspaceData(ml_a, "base", obj_a)
    End Sub

    ' ****************************
    Public Sub GetScalarReal(ByVal ml_a As String, ByRef a As Double)
      Dim obj_a As Object = Nothing
      matlab.GetWorkspaceData(ml_a, "base", obj_a)
      a = CDbl(obj_a)
    End Sub

    ' ****************************
    Public Sub PlusComplexNumbers()

      ' 1. Set number
      Dim aReal As Double = 1.1
      Dim aImag As Double = 101.1

      Dim bReal As Double = 2.2
      Dim bImag As Double = 202.2
```

```
    ' 2. Perform tasks in Matlab Workspace
    matlab.PutWorkspaceData("ml_aReal", "base", aReal)
    matlab.PutWorkspaceData("ml_aImag", "base", aImag)
    matlab.Execute(" ml_aComp = complex(ml_aReal, ml_aImag) ; ")

    matlab.PutWorkspaceData("ml_bReal", "base", bReal)
    matlab.PutWorkspaceData("ml_bImag", "base", bImag)
    matlab.Execute(" ml_bComp = complex(ml_bReal, ml_bImag) ; ")

    matlab.Execute(" ml_cComp = ml_aComp + ml_bComp ; " )

    matlab.Execute(" ml_cReal = real(ml_cComp) ; ")
    matlab.Execute(" ml_cImag = imag(ml_cComp) ; ")

    ' 3. Get data from Matlab Workspace
    Dim obj_cReal As Object = Nothing
    Dim obj_cImag As Object = Nothing
    matlab.GetWorkspaceData("ml_cReal", "base", obj_cReal)
    matlab.GetWorkspaceData("ml_cImag", "base", obj_cImag)

    '4. Print out
    Dim cReal As Double = 0
    Dim cImag As Double = 0

    cReal = CDbl(obj_cReal)
    cImag = CDbl(obj_cImag)

    Console.WriteLine("Result of a complex number :")
    Console.WriteLine("Real Part: {0}", cReal.ToString())
    Console.WriteLine("Imag Part: {0}", cImag.ToString())

End Sub

' ********************************
' *************II. Vector ********
' ********************************
```

```
Public Sub PutVectorReal(ByVal ml_vector As String, ByVal vector As Array)

  Dim dummy As Array = Array.CreateInstance(GetType(Double), vector.Length)
  Array.Clear(dummy, 0, vector.Length)

  matlab.PutFullMatrix(ml_vector, "base", vector, dummy)

End Sub

' ********************************
Public Sub PutVectorComplex(ByVal ml_vectorComplex As String, _
                            ByVal vectorReal As Array, ByVal vectorImag As Array)

  matlab.PutFullMatrix(ml_vectorComplex, "base", vectorReal, vectorImag)

End Sub

' ********************************
Public Sub GetVectorReal(ByVal ml_vector As String, ByRef vector As Array)

  Dim dummy As Array = Array.CreateInstance(GetType(Double), vector.Length)
  Array.Clear(dummy, 0, vector.Length)

  matlab.GetFullMatrix(ml_vector, "base", vector, dummy)

End Sub

' ********************************
Public Sub GetVectorComplex(ByVal ml_vectorComplex As String, _
                            ByRef vectorReal As Array, ByRef vectorImag As Array)

  matlab.GetFullMatrix(ml_vectorComplex, "base", vectorReal, vectorImag)

End Sub

'********************************
```

```
'********** III. Matrix *********
'*******************************
Public Sub PutMatrixReal(ByVal ml_matrix As String, ByRef matrix As Array)

  Dim row As Integer
  Dim col As Integer

  row = matrix.GetUpperBound(0) - matrix.GetLowerBound(0) + 1
  col = matrix.GetUpperBound(1) - matrix.GetLowerBound(1) + 1

  Dim dummy As Array = Array.CreateInstance(GetType(Double), row, col)
  Array.Clear(dummy, 0, row * col)

  matlab.PutFullMatrix(ml_matrix, "base", matrix, dummy)

End Sub

' *******************************
Public Sub GetMatrixReal(ByVal ml_matrix As String, ByRef matrix As Array)

  Dim row As Integer
  Dim col As Integer

  row = matrix.GetUpperBound(0) - matrix.GetLowerBound(0) + 1
  col = matrix.GetUpperBound(1) - matrix.GetLowerBound(1) + 1

  Dim dummy As Array = Array.CreateInstance(GetType(Double), row, col)
  Array.Clear(dummy, 0, row * col)

  matlab.GetFullMatrix(ml_matrix, "base", matrix, dummy)

End Sub

' *******************************
Public Sub PutMatrixComplex(ByVal ml_matrixComplex As String, _
                            ByRef matrixReal As Array, ByRef matrixImag As Array)
```

```
  matlab.PutFullMatrix(ml_matrixComplex, "base", matrixReal, matrixImag)

End Sub

' ********************************
Public Sub GetMatrixComplex(ByVal ml_matrixComplex As String, _
                            ByRef matrixReal As Array, ByRef matrixImag As Array)

  matlab.GetFullMatrix(ml_matrixComplex, "base", matrixReal, matrixImag)

End Sub

' ********************************
' ************* Print out *******
' ********************************
' ********************************
Public Sub PrintValues(ByVal myArr As Array)
  Dim myEnumerator As System.Collections.IEnumerator = _
     myArr.GetEnumerator()
  Dim i As Integer = 0
  Dim cols As Integer = myArr.GetLength(myArr.Rank - 1)

  'for row vector or column vector
  If myArr.Rank = 1 Then

    While myEnumerator.MoveNext()
      Console.WriteLine(ControlChars.Tab + "{0}", myEnumerator.Current)
    End While

  Else
    'for other
    While myEnumerator.MoveNext()
      If i < cols Then
        i += 1
      Else
```

```
            Console.WriteLine()
            i = 1
          End If
          Console.Write(ControlChars.Tab + "{0}", myEnumerator.Current)
        End While
      End If

      Console.WriteLine()
    End Sub

    '    /* ******************************** */
    Public Sub Execute(ByVal commandName As String)

      matlab.Execute(commandName)
    End Sub

    ' ******************************** */
    Public Sub PutString(ByVal ml_str As String, ByVal str As String)

      matlab.PutCharArray(ml_str, "base", str)

    End Sub

    '********************************
    '********************************
    '********************************

  End Class

End Namespace
```

end code

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. MATLAB compiler version 4, user guide. The Language of Technical Computing. URL address: www.mathworks.com/access/helpdesk/help/pdf_doc/compiler/compiler.pdf.

[3] 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.

[4] 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.

[5] 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.

Index

MATLAB - Visual Basic .NET for Engineers

CD-Rom available at www.LePhanPublishing.com

The CD-Rom of **MATLAB - Visual Basic .NET for Engineers** includes:

1. All the text files of this book **MATLAB - Visual Basic .NET for Engineers** in the pdf format
2. All codes of M-files are used in the book
3. The completed Visual Basic .NET codes for every chapter

The Visual Basic .NET examples are working on scalars, vectors, and matrixes, which are inputs/outputs of functions for every application. In addition, the example codes are portable and presented in the step-by-step method, therefore you can easily reuse the codes or write your own codes following the step-by-step procedure.