CHAPTER 7

Linear Algebra

This chapter introduces linear algebra. It discusses some of the essential approaches to solving systems of linear equations, as well as various matrix operations (matrix inverse, determinant, sum, subtraction, division, multiplication, power, exponential, elementwise and array-wise operations, and so forth). It covers eigen-value problems and matrix factorizations/decompositions, such as Cholesky, Schur, LU, QR, and singular value decomposition. It also includes built-in functions and scripts in MATLAB and Simulink models. Moreover, the chapter explains the standard matrix generator functions of MATLAB, how to create vector spaces, how to solve polynomials, and the logical indexing of matrices, all via examples in MATLAB and Simulink.

Introduction to Linear Algebra

Linear algebra is one of the more important branches of mathematics. It deals with vectors, vector spaces, linear spaces, matrices, and systems of linear equations. There is a wide range of linear algebra applications in engineering and scientific computing, including many fields of natural and social studies. Linear algebra starts with a system of linear equations for underdetermined, overdetermined, and well-defined systems.

If a given system is composed of m-linear equations with n-unknowns and $m \geq n$, that is solvable for unknowns. Consider the following linear system, formulated by the system of equations (Equation 7-1):

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + .. + a_{1n}x_n = b_1 \\ \vdots \quad + \quad \vdots \quad + \quad \vdots \quad = \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + .. + a_{mn}x_n = b_m \end{cases} \qquad \text{(Equation 7-1)}$$

The system of linear equations (Equation 7-1) is solvable directly for all cases when $m \geq n$. If $m < n$, there are more unknowns than the number of linearly independent equations, and such a system is called *underdetermined* and not solvable directly.

© Sulaymon Eshkabilov 2022
S. Eshkabilov, *Beginning MATLAB and Simulink*, https://doi.org/10.1007/978-1-4842-8748-4_7

If $m > n$, there are more linearly independent equations, and such a system is called *overdetermined* and is solvable directly.

For the sake of simplicity, let's take $m = n$ and rewrite Equation 7-1.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + .. + a_{1n}x_n = b_1 \\ : \quad + \quad : \quad + \quad : \quad = : \\ a_{n1}x_1 + a_{n2}x_2 + .. + a_{nn}x_n = b_n \end{cases} \qquad \text{(Equation 7-2)}$$

The given system of linear equations in Equation 7-2 can also be written in matrix notation form.

$$[A]*\{X\} = [B] \qquad \text{(Equation 7-3)}$$

Here, A and B are matrices and X is a vector of unknowns.

$$\text{Where } [A] = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ : & : & : \\ a_{n1} & \dots & a_{nn} \end{bmatrix}, \{X\} = \{x_1, x_2, \dots, x_n\}, [B] = \begin{bmatrix} b_1 \\ : \\ b_n \end{bmatrix}$$

Equation 7-3 can also be rewritten in the form of column matrices.

$$x_1 \begin{bmatrix} a_{11} \\ : \\ a_{n1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ : \\ a_{n2} \end{bmatrix} + \dots + x_n \begin{bmatrix} a_{1n} \\ : \\ a_{nn} \end{bmatrix} = \begin{bmatrix} b_1 \\ : \\ b_n \end{bmatrix} \qquad \text{(Equation 7-4)}$$

The system in Equation 7-3 or 7-4 can be solved for X (unknowns) with the next formulation:

$$\{X\} = [A]^{-1} * [B] \qquad \text{(Equation 7-5)}$$

Here, $[A]^{-1}$ is the inverse of the matrix [A].

Matrix Properties and Operators

Matrices have several important properties and operators, such as determinant, diagonal, transpose, inverse, singularity, rank, and so forth.

The *determinant* of a matrix can be computed only if the given matrix is a square. Here's an example:

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

The determinant of *M* will be computed with the following expression:

$$det\,(M) = aei + bfg + dhc - ceg - dbi - hfa$$

The MATLAB command for the determinant computation is `det()`. Here's an example:

```
 >> A=[   8   1   6;   3   5   7;   4   9   2]
A =

     8     1     6
     3     5     7
     4     9     2
>> det(A)
ans =
-360
```

The *diagonal* of a matrix is composed of its element along its diagonals. For example, in the previous example, the diagonals are *aei* and *ceg*.

The MATLAB command for diagonal separation is `diag()`. Here's an example:

```
>> A = [   8   1   6;   3   5   7;   4   9   2];
>> diag(A)
ans =
     8
     5
     2
```

The *transpose* of a matrix can be determined by the counterclockwise rotation of a matrix by 90^0 (degrees). The transpose properties are as follows:

$$\left(M^T\right)^T = M$$

$$\left(M + B\right)^T = M^T + B^T$$

$$\left(kM\right)^T = kM^T$$

$$\left(MB\right)^T = B^T M^T$$

$$\left(M^{-1}\right)^T = \left(M^T\right)^{-1}$$

Here, M and B are matrices of the same size, k is a scalar, and ^{T} and $^{-1}$ are the transpose and inverse operators.

The MATLAB command for the transpose operation is `transpose()`, or `'`.

Here's an example:

```
>> A =[   8   1   6;   3   5   7;   4   9   2];
>> transpose(A)
ans =

     8     3     4
     1     5     9
     6     7     2
>> A'
ans =

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

Simulink Blocks for Matrix Determinant, Diagonal Extraction, and Transpose

Simulink has blocks that you can use to compute the matrix determinant, extract the matrix diagonal elements, and obtain the matrix transpose. The determinant block (`[det(A) (3x3)]`) is present in Simulink's Aerospace Blockset/Utilities/Math Operations, and it has a constraint and can only compute the determinant of 3-by-3 matrices.

Note The block `[det(A) (3x3)]` from the aerospace blockset is limited; it can only compute the determinant of 3-by-3 matrices.

The block to extract the diagonal elements of a matrix is available in the DSP System Toolbox/MATH Functions/Matrices and Linear Algebra/Matrix Operations. The block to compute the matrix transpose is present in Simulink/Math Operations, and the block name is Math Function. It has a few math functions embedded in it, including exp (by default), log, 10^u, magnitude^2, square, pow, and transpose. Any of these math

functions in the Math Function block can be chosen. You simply click the Apply and OK buttons of the block, and the chosen math function becomes available. Figure 7-1 shows these three blocks.

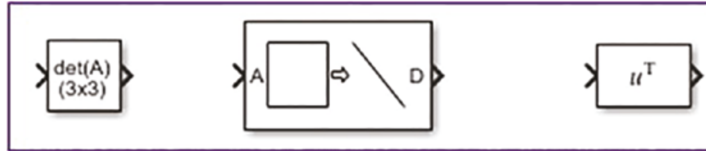


Figure 7-1. *Simulink blocks used for determinant calculation, diagonal extraction, and transpose operation, from left*

These blocks have one input and one output port. Therefore, you need to add two additional blocks, specifically, one Constant block for input entry and one Display block, to obtain/see the computation results. The Constant block can be taken from the Simulink Library Simulink/Sources or DSP System Toolbox/Sources. Similarly, the Display block can be taken from Simulink/Sinks or DSP System Toolbox/Sinks. Alternatively, with the latest versions of MATLAB starting from 2018a, you can obtain all the necessary blocks by double-clicking (with the left mouse button) and typing the block name in the search box. As discussed in the previous chapters, in any Simulink model one signal source can be used as many times as necessary. There is no need to generate that signal within one model to use it with other blocks as an input signal. Moreover, to optimize the Simulink model, it is strongly advised you build a Simulink model with fewer blocks to make your models more readable, comprehensive, and easy to edit. Therefore, this example uses one Constant block for input source [A]. Figure 7-2 shows the primary version of the Simulink model.

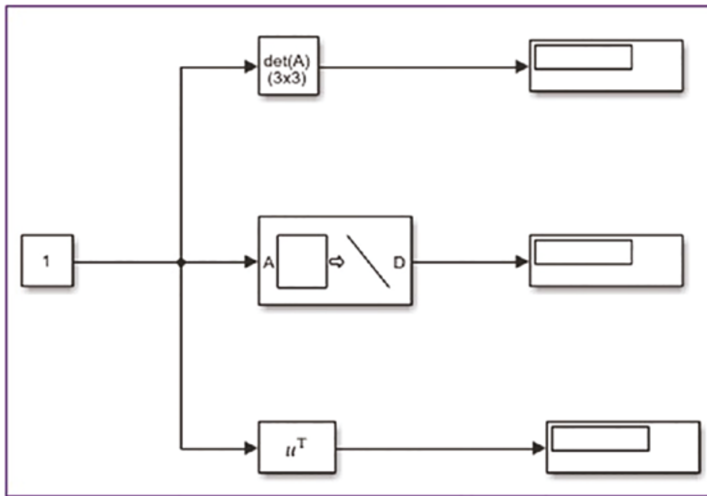


Figure 7-2. *Simulink model to compute the determinant of a matrix, extract diagonal elements of a matrix, and perform a transpose on a matrix*

Let's use example matrix [A] to demonstrate these three Simulink blocks. The elements of the matrices [A] can be entered in two different ways:

- By typing all elements in the Constant block's Constant Value box, as shown in Figure 7-3. Click the Apply and OK buttons.

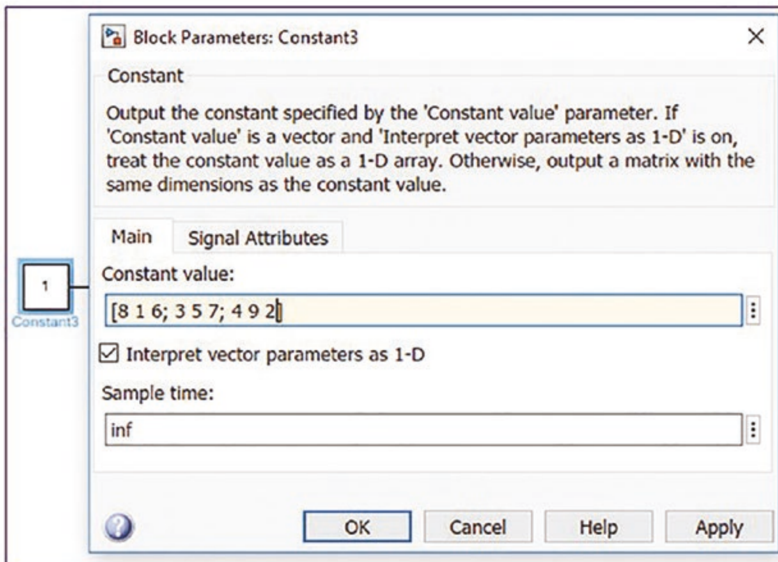


Figure 7-3. *Entering matrix elements in a Constant block*

- By defining [A] via MATLAB's Command window and workspace:

```
>> A =[  8  1  6;  3  5  7;  4  9  2];
```

Then provide the variable name A in the Constant block's Constant Value box for [A], as shown in Figure 7-4.

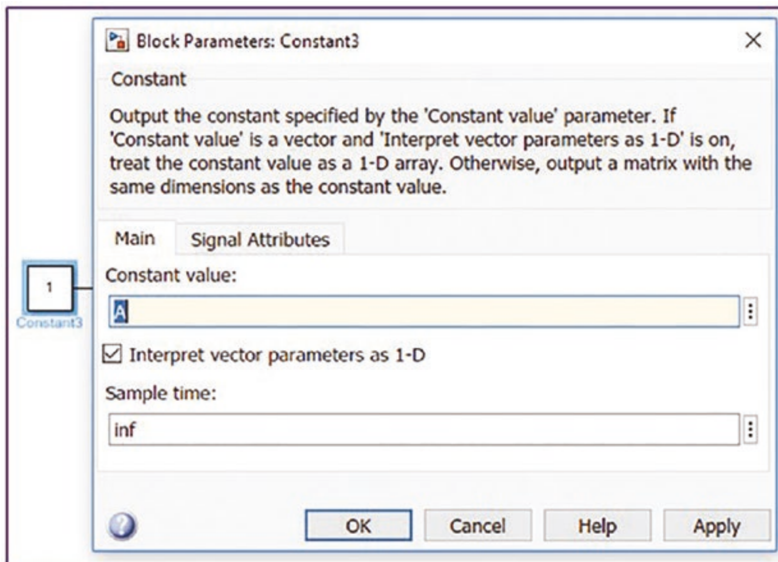
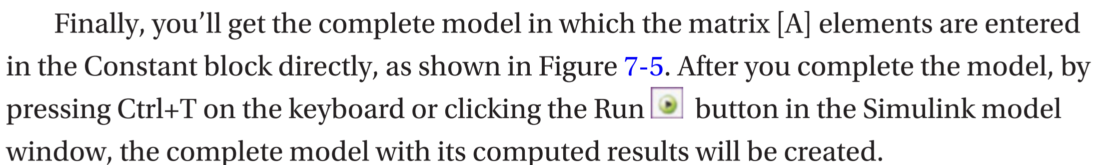


Figure 7-4. *Matrix [A], defined in the MATLAB workspace, called via the Constant block*

Then click the Apply and OK buttons. Note that we are not going to use the second method (see Figure 7-4) of defining matrix [A] elements in this example; it's just shown here for explanation purposes.

Finally, you'll get the complete model in which the matrix [A] elements are entered in the Constant block directly, as shown in Figure 7-5. After you complete the model, by pressing Ctrl+T on the keyboard or clicking the Run  button in the Simulink model window, the complete model with its computed results will be created.

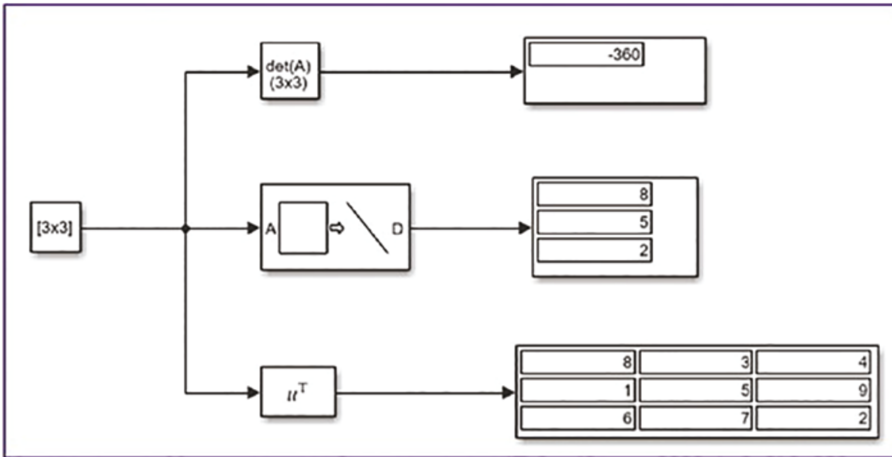


Figure 7-5. *Completed Simulink model that computes the determinant, extracts diagonal elements, and performs the transpose operation on the 3-by-3 matrix*

Note To see the simulation results in the Display block, it has to be resized/stretched. You left click it and then drag with the mouse while holding the button.

The simulation results of the Simulink models match the ones from the MATLAB commands, such as `det()`, `diag()`, and `transpose()`, or `'`.

Matrix Inverse or Inverse Matrix

The *inverse matrix* has the following important property:

$$[A]*[A]^{-1}=[I]$$

Here, $[I]$ is the identity matrix.

For example, $A=\begin{bmatrix}1 & 1\\ 3 & 4\end{bmatrix}$ has its inverse $A^{-1}=\begin{bmatrix}4 & -1\\ -3 & 1\end{bmatrix}$ that is computed from the following:

$$A^{-1}=\frac{1}{\det(A)}\,adjugate(A)=1/(4*3-(-1*-3)*\begin{bmatrix}4 & -1\\ -3 & 1\end{bmatrix}$$

The MATLAB command to compute the inverse of a matrix is `inv()`. Here's an example:

```
>> A =[   8    1    6;    3    5    7;    4    9    2];
>> inv(A)
ans =
    0.1472   -0.1444    0.0639
   -0.0611    0.0222    0.1056
   -0.0194    0.1889   -0.1028
```

A given matrix is *singular* if it is square, if it does not have an inverse, and if it has a determinant of 0.

Simulink Blocks for Inverse Matrix

The *matrix inverse* can also be calculated via several Simulink blocks with respect to a given matrix size, i.e., square matrix or rectangular. The inverse matrix or matrix inverse computing blocks are present in the DSP System and Aerospace Blockset Toolboxes of Simulink and can be accessed via the Simulink Library: the DSP System Toolbox/Math Functions/Matrices and Linear Algebra/Matrix Inverses, and the Aerospace Blockset/Utilities/Math Operations. Let's test the available blocks of this toolbox to compute the inverse of the matrix [A] shown in the previous example. Open a blank Simulink model and drag and drop the block from the libraries of the DSP System and Aerospace Blockset Toolboxes shown in Figure 7-6.

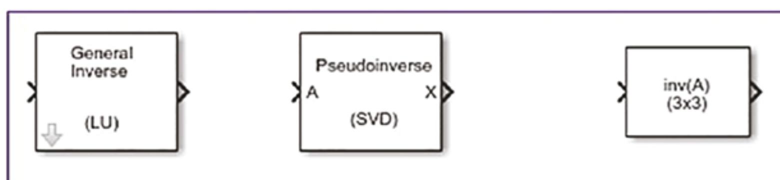


Figure 7-6. *Simulink blocks for computing the inverse matrix*

They are as indicated on the top of each block—General Inverse (LU), Pseudoinverse (SVD), and `inv(A)`—used to compute the matrix inverses based on LU factorization for square matrices, and pseudoinverse for rectangular matrices (i.e., `m>n`, or the number of rows is larger than the number of columns or vice versa). Theoretical aspects of the LU, SVD, and other matrix decomposition and transformation operations are highlighted in the "Matrix Decomposition" section.

The three blocks have one input port for the entry matrix and one output port for the computed inverse. Add two additional blocks—one Constant block and one Display—by following the procedures. Figure 7-7 shows the primary version of the Simulink model.

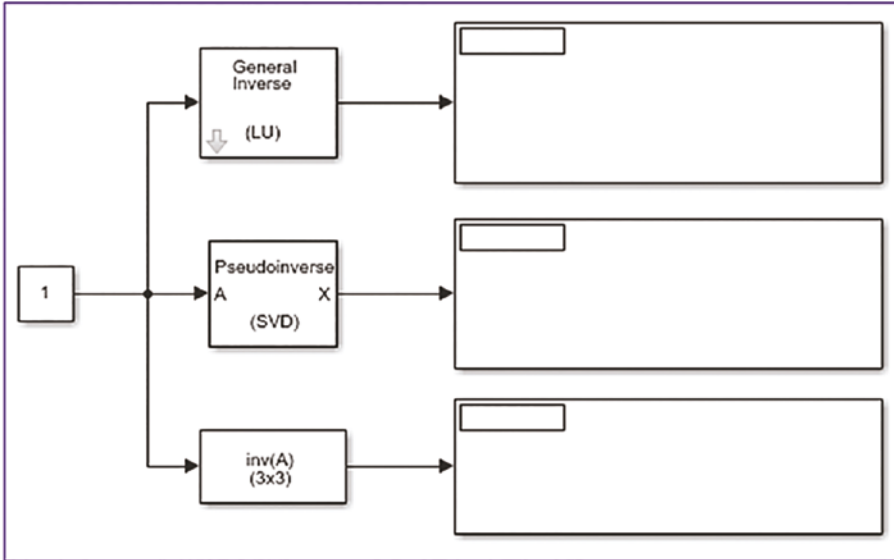
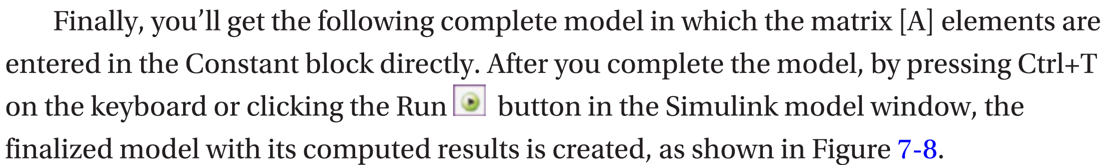


***Figure 7-7.** Simulink model to compute the inverse matrix via three different blocks*

The elements of the matrices [A] can be entered in two ways: (1) by typing all the elements in the Constant block's Constant Value box and then clicking the Apply and OK buttons; or (2) by defining [A] via the MATLAB's Command window and workspace.

Finally, you'll get the following complete model in which the matrix [A] elements are entered in the Constant block directly. After you complete the model, by pressing Ctrl+T on the keyboard or clicking the Run  button in the Simulink model window, the finalized model with its computed results is created, as shown in Figure 7-8.

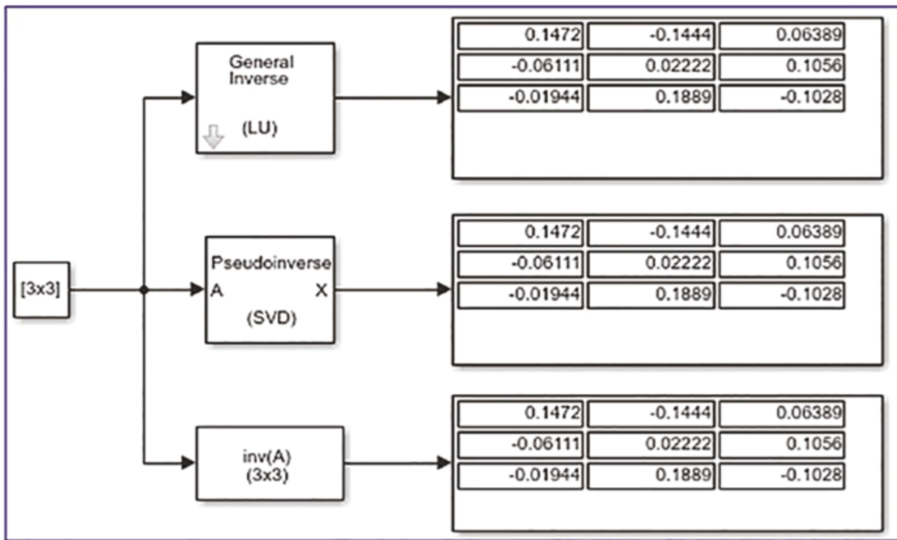


Figure 7-8. *The inverse matrix computed via three different blocks*

The computed inverse matrix (A^{-1}) values match the ones computed using MATLAB's `inv()` command, within four correct decimal places.

Another important operator of matrices is its *rank*. The *rank* of a matrix (e.g., [A]) is the maximum number of linearly independent row vectors of the matrix, which is the same as the maximum number of linearly independent column vectors. The [A] matrix is considered to have a full rank if its rank equals the largest possible for a matrix of the same dimensions. The [M] matrix is considered to be rank deficient if it does not have full rank. A matrix's rank determines how many linearly independent rows the system contains. The MATLAB command to compute the rank of a matrix is `rank()`. Here's an example:

```
>> A =[8,   1,   6;   3,   5,   7;   4,   9,   2];   % Full rank matrix
>> rank(A)
ans = 3
>> M  =[8   0   6;   -3,   0,   7;   0   0   2]   % Rank deficient matrix
M =
     8     0     6
    -3     0     7
     0     0     2
>> rank(M)

ans =

     2
```

Based on the rank, the systems (system matrices) can be full rank, overdetermined, and underdetermined.

Example 1: Solving a System of Linear Equations

The following example shows you how to solve a linear equation by using these formulations:

$$\begin{cases} 2x+3y+5z=1 \\ -3x-2y+5z=2 \\ 4x-7y+6z=3 \end{cases}$$

To solve this problem for unknowns, such as x, y, z, you apply Equations 7-3, 7-4, and 7-5 directly and then use the following operations:

$$\begin{bmatrix} 2 & 3 & 5 \\ -3 & -2 & 5 \\ 4 & -7 & 6 \end{bmatrix} * \begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

That can be written as follows:

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{bmatrix} 2 & 3 & 5 \\ -3 & -2 & 5 \\ 4 & -7 & 6 \end{bmatrix}^{-1} * \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} \cong \begin{bmatrix} 0.0754 & -0.1738 & 0.0820 \\ 0.1246 & -0.0262 & -0.0820 \\ 0.0951 & 0.0852 & 0.0164 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \cong \begin{bmatrix} -0.0262 \\ -0.1738 \\ 0.3148 \end{bmatrix}$$

$$Solution: \begin{Bmatrix} x \\ y \\ z \end{Bmatrix} \cong \begin{bmatrix} -0.0262 \\ -0.1738 \\ 0.3148 \end{bmatrix}$$

Let's solve this exercise using the reduced row echelon method in MATLAB.

```
% Step 1. Write an augmented matrix: AU = [A, b]
A = [2 3 5; -3 -2 5; 4 -7 6;]; b = [1;2;3];
AU=[A, b];
% Step 2. Row1 = Row1 - Row2
```

```
AU(1,:)=AU(1,:)-AU(2,:);
% Step 3. Row3 = Row3-4*Row1/5
AU(3,:)= AU(3,:)-4*AU(1,:)/5;
% Step 4. Row2 = Row2+3*Row1/5
AU(2,:)= AU(2,:)+3*AU(1,:)/5;
% Step 5. Row3 = Row3+11*Row2
AU(3,:)= AU(3,:)+11*AU(2,:);
% Step 6. Row2 = Row2+5*Row3/61
AU(2,:)= AU(2,:)-5*AU(3,:)/61;
% Step 7. Row1 = Row1/5-Row2
AU(1,:)= AU(1,:)/5-AU(2,:);
% Step 8. Row3 = Row3/61
AU(3,:)= AU(3,:)/61;
% Step 9. Solution:
x= AU(:, end)
x =
       -0.0262295081967213
        -0.173770491803279
         0.314754098360656
```

Alternative ways of solving this example include Gauss elimination and graphical methods. There are a number of operators and built-in functions in MATLAB that can be used to solve a linear system of equations. They are as follows:

- `inv()`, which computes the inverse of a given matrix or the pseudo-inverse of the given system (used for overdetermined systems).
- `\`, the backslash operator, which solves the system of linear equations directly. It's based on the Gaussian elimination method. This is one of the most powerful MATLAB operators (tools) for handling matrices.
- `mldivide()`, which is a built-in function similar to the `\` backslash operator.
- `linsolve()`, which is a built-in function similar to the `\` backslash operator.
- `lsqr()`, which is a built-in function based on the least squares method.

- `lu()`, which is a built-in function based on the Gauss elimination method.
- `rref()`, which is a built-in function based on the reduced row echelon method.
- `svd()`, which is a built-in function based on the singular value decomposition.
- `chol()`, which is a built-in function based on the Cholesky decomposition.
- `qr()`, which is a built-in function based on the orthogonal triangular decomposition.
- `decomposition()`, which is a built-in function that automatically choses the decomposition method.
- `bicg()`, `cgs()`, `gmres()`, `pcg()`, `symmlq()`, and `gmr()`, which are built-in functions that are based on gradient methods.
- `solve()`, which is a built-in function from the Symbolic MATH toolbox.

Note Among these listed functions/commands and operators, some of them use the same computing algorithm and are alternatives to each other. For example, the \ backslash operator is an alternative to `mldivide()`.

First, denote the given system with the following notations:

$$A = \begin{bmatrix} 2 & 3 & 5 \\ -3 & -2 & 5 \\ 4 & -7 & 6 \end{bmatrix},\ B = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

The entries of $[A]$ matrix (coefficients of the unknowns x, y, z) are defined, and the elements of $[B]$ matrix are defined in the Command window.

```
>> A = [2 3, 5; -3, -2, 5; 4, -7, 6]
A =
     2     3     5
    -3    -2     5
     4    -7     6
```

```
>> B = [1;2;3]
B =
     1
     2
     3
```

Using inv() and (*), we can compute the solutions of the system.

```
>>Ai=inv(A)   %   [B] matrix is an  inverse  matrix  of  [A]  matrix.
Ai =0.0754   -0.1738   0.0820
0.1246   -0.0262  -0.0820
0.0951    0.0852   0.0164
>> XYZ1=Ai*B   %   Solutions of the problem
Ai =-0.0262
-0.1738
0.3148
```

The next example uses the backslash \ operator based on the Gaussian elimination method. This approach is quite simple and efficient in terms of computation time.

```
>> XYZ2=A\B
Ai=-0.0262
-0.1738
0.3148
```

Using mldivide():

```
>>XYZ3=mldivide(A,B)
-0.0262
-0.1738
0.3148
```

Using linsolve():

```
>>XYZ4=linsolve(A,B)
-0.0262
-0.1738
0.3148
```

Using lsqr():

```
>>XYZ5=lsqr(A,B)
lsqr converged at iteration 3 to  a  solution  with  relative
residual  6.6e-17.
-0.0262
-0.1738
0.3148
```

Using lu():

```
>>[L, U, P] = lu(A); %L-lower; U-upper triangular; P-Permutation matrix
>> y = L\(P*B);
>> XYZ6 =  U\y
XYZ6 =
-0.0262
-0.1738
0.3148
```

Using rref():

```
>>   MA   = [A,  B];   % Augmented matrix
>> xyz = rref(MA);
>> XYZ7= xyz(:,end)
-0.0262
-0.1738
0.3148
```

Using svd() and inv():

```
>> [U, S, V]= svd(A);
>> XYZ8 = V*inv(S)*U'*B
-0.0262
-0.1738
0.3148
```

Using chol():

```
>>  [U,  L]  =  chol(A);   % A has to be Hermitian positive definite
>>   XYZ9 =  U\(U'\B)   % U'*U = A
-0.0262
-0.1738
0.3148
```

Using qr():

```
>> [Q, R] = qr(A);
>>  XYZ10   = R\Q.'*B
-0.0262
-0.1738
0.3148
```

Using decomposition():

```
>> XYZ11 = decomposition(A)\B
-0.0262
-0.1738
0.3148
```

Using bicg() gradient methods:

```
>> XYZ12 = bicg(A, B)
bicg converged at iteration 3 to a solution with relative  residual  3.1e-14.
-0.0262
-0.1738
0.3148
```

Using solve(), which is a Symbolic Math Toolbox function:

```
>> syms x y z
>> sol=solve(2*x+3*y+5*z-1, -3*x-2*y+5*z-2, 4*x-7*y+6*z-3);
>> XYZ13=[sol.x; sol.y; sol.z]
-8/305
-53/305 96/305
```

```
>> XYZ13=double([sol.x; sol.y; sol.z])
-0.0262
-0.1738
0.3148
```

All of the computed solutions are accurate within four decimal places of the employed operators and functions. In fact, the accuracy of the solutions and the computation time of each operator or function will differ. For instance, the inverse matrix calculation is not only costly in terms of computation time but is also less accurate. Moreover, among the studied methods, the last function of the Symbolic Math Toolbox, `solve()`, is the slowest and least efficient method.

Note The `decomposition()` function is available in the recent versions of MATLAB starting from MATLAB 2018b.

Simulink Modeling

In addition to the MATLAB commands demonstrated, Simulink has several blocks by which the linear system of equations, such as [A]{*x*} = [B], can be solved. All of the solver blocks are present in the DSP System Toolbox and can be accessed via the Simulink Library: the DSP System Toolbox/Math Functions/Matrices and Linear Algebra/Linear System Solvers. Let's test some of the blocks here to solve the previous example, called Example 1. Open a blank Simulink model and drag and drop the block from the DSP System Toolbox library, as shown in Figure 7-9.

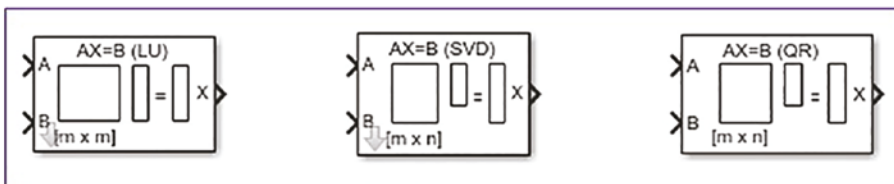


Figure 7-9. *Simulink blocks used to solve a system of linear equations*

They are as indicated on the top of each block—LU, SVD, QR factorization and decomposition operation-based solvers. All of them have two input ports for [A] and [B] and one output port for a solution, {x}. Therefore, you need to add three additional

blocks—two Constant and one Display block—which you add as explained previously in building Simulink models to compute determinant, transpose, and inverse of matrices. Figure 7-10 shows the primary version of the Simulink model.

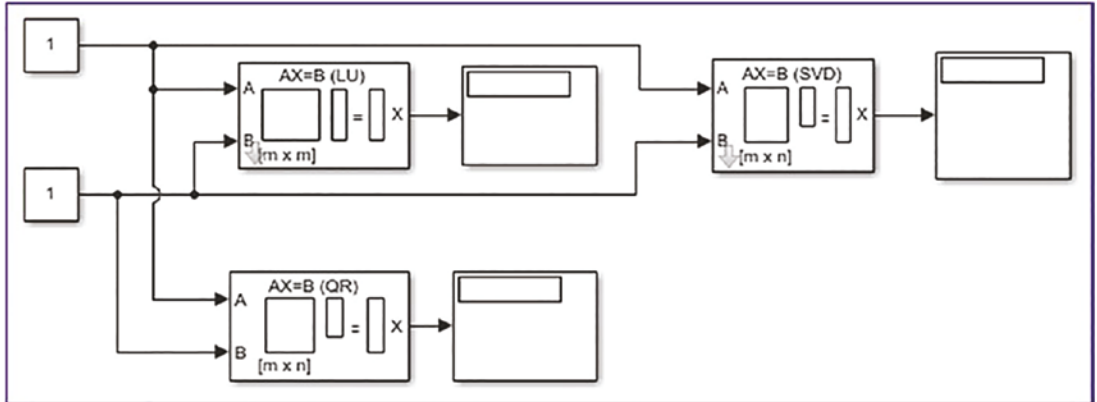


Figure 7-10. *Simulink model to solve a system of linear equations*

The elements of the matrices [A] and [B] can be inserted, as shown in Figure 7-3, directly in the Constant block's Constant Value window. Or you can define the elements of [A] and [B] via MATLAB's Command window and workspace.

```
>> A=[2, 3, 5; -3, -2, 5; 4, -7, 6]
>> B=[1; 2; 3];
```

The variable names A and B are then entered in the first and second Constant block's Constant Value box for [A] and [B], respectively, as shown in Figure 7-11. Click Apply and OK to complete the model.

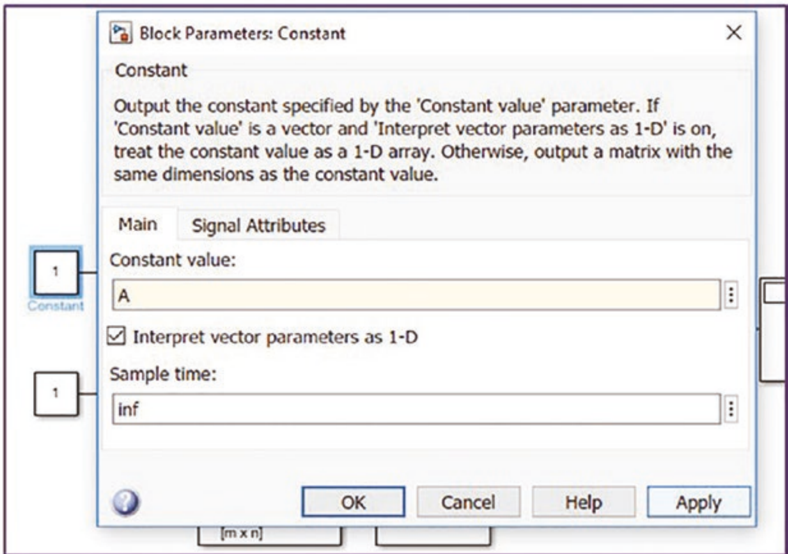
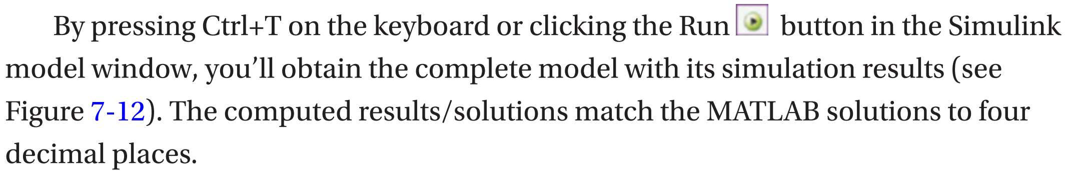


Figure 7-11. *The variable names defined in the Constant block*

By pressing Ctrl+T on the keyboard or clicking the Run  button in the Simulink model window, you'll obtain the complete model with its simulation results (see Figure 7-12). The computed results/solutions match the MATLAB solutions to four decimal places.

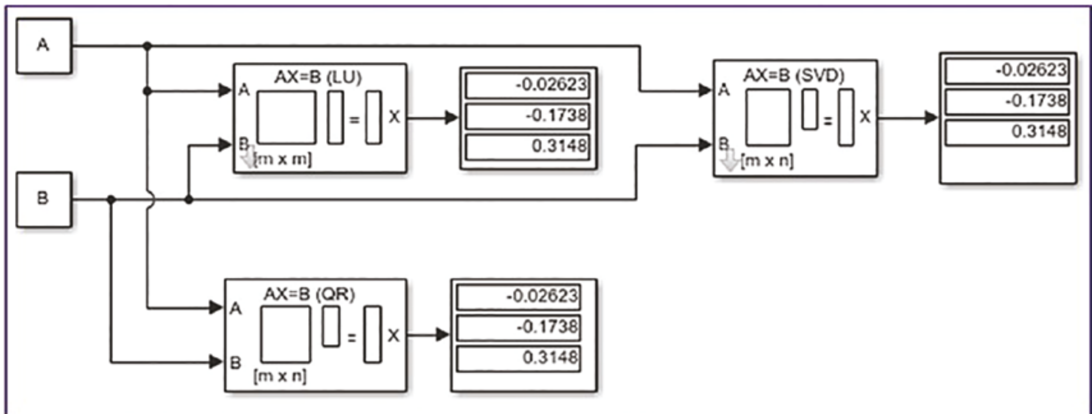


Figure 7-12. *Complete model with computed results*

Note that the variables (matrices) A and B are defined via MATLAB's Command window.

To obtain more decimal places of the computed results with the Display block, the block parameters (Format Type) need to be tuned by selecting `long_e`, as shown in Figure 7-13.

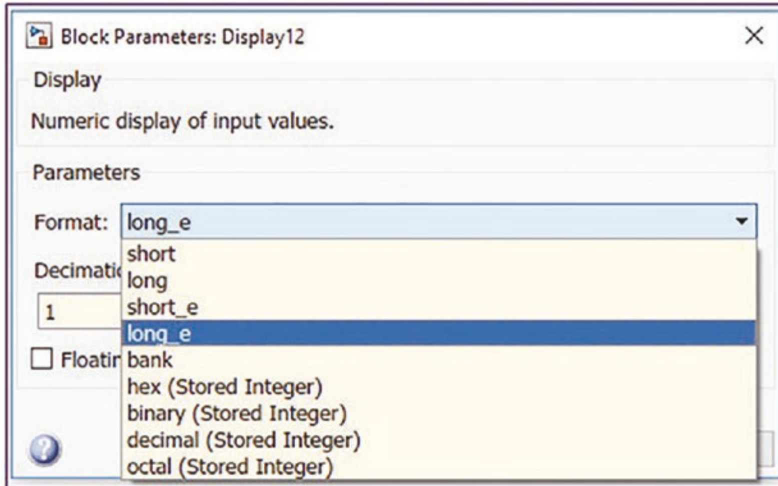


Figure 7-13. *Adjusting the Display block's Format parameter*

Example 2: Embedding a MATLAB Function Block to Compute the Determinant and Solve Linear Equations

All of the aforementioned MATLAB functions/commands used for computing matrix determinants, matrix inverses, or solutions of linear systems can be embedded in Simulink via the MATLAB Function block . Let's take two MATLAB functions/commands used for computing a determinant of a matrix of any size with `det()` and solving with `linsolve()` and embed them into a Simulink model. Here's an example:

$$A1 = \begin{bmatrix} 16 & 2 & -3 & 13 \\ -5 & 11 & 10 & -8 \\ 9 & 7 & -6 & 12 \\ -4 & 14 & 15 & 1 \end{bmatrix}, B1 = \begin{bmatrix} 3 \\ 2 \\ 4 \\ 5 \end{bmatrix}$$

Here are completed Simulink models. Figure 7-14 is built with three Constant, two MATLAB Function, and two Display blocks.

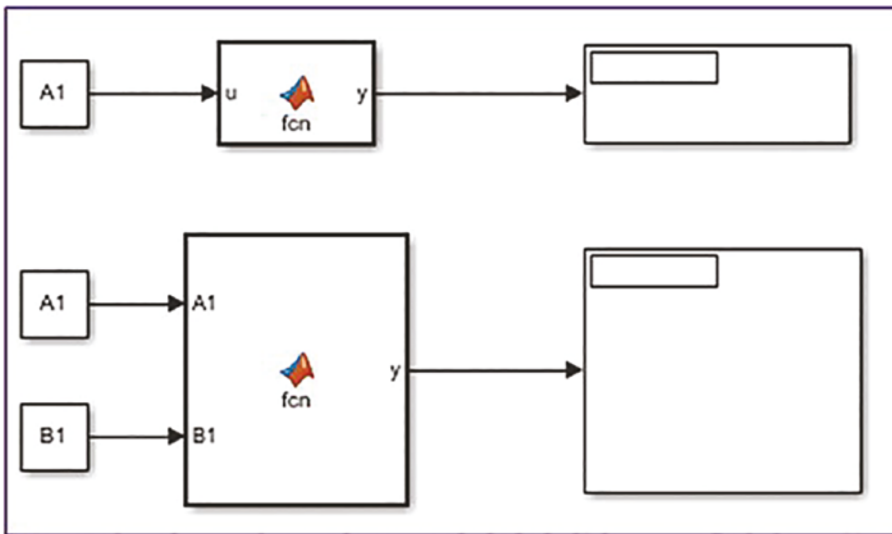


***Figure 7-14.** Simulink models with MATLAB Function blocks to compute the determinant and solve a linear system of equations*

The input variables/entries for A1 and B1 are defined via the Command window and MATLAB workspace in this model. To edit and type in the necessary script, you have to open the MATLAB Function block. It can be opened by double-clicking it, which opens the MATLAB editor window. The following function file scripts for the MATLAB Function blocks are typed in the MATLAB editor for the upper MATLAB Function block (with one input) and the lower one (with two inputs A1 and B1) models, respectively. After editing the codes of the blocks, save them. They will be saved under the created Simulink model and not as a separate MATLAB function file.

```
function y  =  fcn(u)
y = det(u);
end
```

```
function y =  fcn(A1,  B1)
y = linsolve(A1, B1);
```

The model is then completed, and the finalized model is executed. Figure 7-15 shows the completed model with its computed results in the Display blocks. The upper Display block shows the determinant, and the lower one shows the solution of the given system.

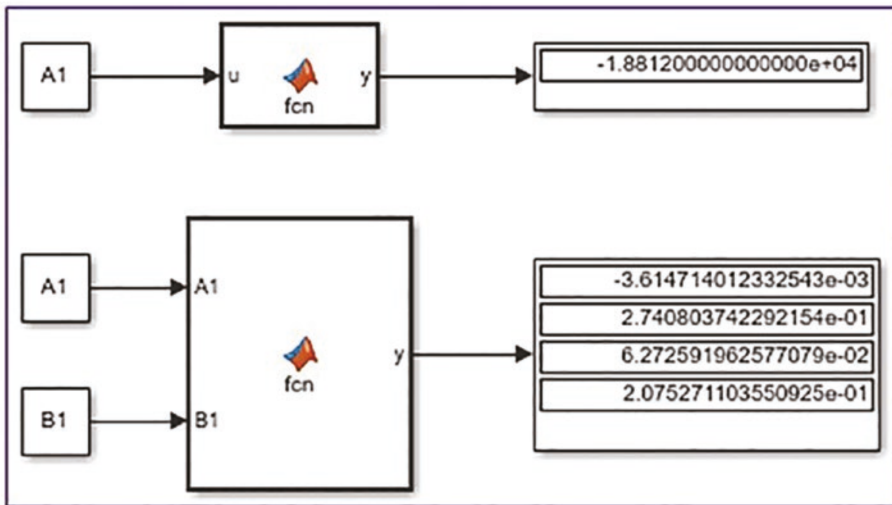


Figure 7-15. *Completed models with computed results*

The computed results of the Simulink model can be compared with MATLAB.

```
>> A1=[16 2 -3 13 ; -5 11 10 -8; 9 7 -6 12; -4 14 15 1 ];
>> B1 = [3; 2; 4; 5];
>> det(A1)
ans =
-18812

>> linsolve(A1, B1)
-3.614714012332536e-03 2.740803742292154e-01
6.272591962577077e-02
2.075271103550925e-01
```

The computed results from the determinant calculation and linear MATLAB solver match the Simulink model's results to 13 decimal places.

Example 3: Accuracy of Solver Functions of Linear Equations

Let's find out which one of the functions/tools (methods) highlighted in Example 1 is more accurate in computing the solutions. For this exercise, you'll take the following 13-by-13 [A] and 13-by-1 [B] matrices generated by the `magic()` and `randi()` (random

integer) matrix generator functions of MATLAB. Moreover, the norm() function is used to compute the norm of the given linear system with its computed solutions. LA_Ex3.m is the complete solution script.

```
%% Given 13-by-13 system of linear equations
A    =   magic(13);
B = randi([-169,169], 13,1); % Elements of B vary within [-169, 169]
%% 1-Way: inv()   or  pinv()   %% INVERSE matrix method
x1a = inv(A)*B; Err_INV  =  norm(A*x1a-B)/norm(B)     %#ok: ERROR checking
x1a = inv(A)*B; Err_PINV  =  norm(A*x1b-B)/norm(B)   %#ok: ERROR checking
%% 2-Way: \   %% backslash
x1a = inv(A)*B; Err_BACKSLASH = norm(A*x2-B)/norm(B)     %#ok: ERROR checking
%%  3-Way: mldivide()   %% Left divide function
x1a = inv(A)*B; Err_MLDIVIDE  =  norm(A*x3-B)/norm(B)   %#ok: ERROR checking
%% 4-Way: Using linsolve();
x1a = inv(A)*B; Err_LINSOLVE  =  norm(A*x4-B)/norm(B)      %#ok: ERROR
checking
%% 5-Way: Using lsqr()
x1a = inv(A)*B; Err_LSQR  =  norm(A*x5-B)/norm(B)         %#ok: ERROR
checking
%% 6-Way: Using lu()
x1a = inv(A)*B; y = L\(P*B); x6 = U\y;
Err_LU  =  norm(A*x6-B)/norm(B)   %#ok: ERROR checking
%% 7 - Way: Using rref()
x1a = inv(A)*B; xyz = rref(MA); x7= xyz(:,end);
Err_RREF  =  norm(A*x7-B)/norm(B)   %#ok: ERROR checking
%%   8 - Way: Using svd()
x1a = inv(A)*B; x8 = V*inv(S)*U'*B;
Err_SVD  =  norm(A*x8-B)/norm(B)   %#ok: ERROR checking
%% 9 - Way: Using chol()
x1a = inv(A)*B; x9 = U\(U'\B);
Err_CHOL  =  norm(A*x9-B)/norm(B)   %#ok: ERROR checking
%% 10 - Way: Using qr()
x1a = inv(A)*B; x10   =  R\Q.'*B;
Err_QR  =  norm(A*x10-B)/norm(B)   %#ok: ERROR checking
%% 11 - Way: Using decomposition()
```

```
x1a = inv(A)*B; Err_DECOMPOSITION = norm(A*x11-B)/norm(B) %#ok: ERROR checking
%% 12 - Way: Using bicg()
x1a = inv(A)*B; Err_BICG  =  norm(A*x12-B)/norm(B)   %#ok: ERROR checking
%% 13-Way: solve()   %% SOLVE() symbolic math method
x = sym('x', [1, 13]); x=x.'; Eqn = A*(x); Eqn = Eqn - B;
Solution = solve(Eqn); SOLs = struct2array(Solution); SOLs = double(SOLs);
 x13 = SOLs';
Err_SOLVE  =  norm(A*x13-B)/norm(B)   %#ok: ERROR checking
```

Here are the errors that were made while computing the solutions of the system with the employed methods:

```
Err_INV =
5.8087e-16
Err_PINV =
3.7982e-15 Err_BACKSLASH = 3.0569e-16 Err_MLDIVIDE = 3.0569e-16 Err_
LINSOLVE = 3.0569e-16
```

lsqr converged at iteration 7 to a solution with relative residual 3.5e-07. Err_LSQR =

```
3.4959e-07
Err_LU =
3.0569e-16
Err_RREF =
1.1576e-05
Err_SVD =
3.7982e-15
Err_CHOL  =
2.2400
Err_QR =
7.0615e-16
Err_DECOMPOSITION =
3.0569e-16
```

`bicg` stopped at iteration 13 without converging to the desired tolerance 1e-06 because the maximum number of iterations was reached.

The iterate returned (number 13) has relative residual 9.6e-06.

```
Err_BICG = 9.5856e-06
Err_SOLVE = 1.5109e-16
```

From the computed errors, it is clear that the RREF(), BICG(), and LSQR() functions make errors within the margin of $10^{-5}...10^{-7}$ and all other methods make errors within the margin of $10^{-15}...10^{-16}$ while computing the solutions of this given system.

Example 4: Efficiency of Solver Functions of Linear Equations

This example demonstrates which one of the shown ways is more efficient in terms of computation time. For this demonstration, you'll consider two large matrices of 1000-by-1000 and 1000-by-1, generated by the random integer number generator function randi() to generate the elements of matrices [A] and [B]. In addition, to record the elapsed time of each computation method, the [tic, toc] functions are used. Here is the complete solution script, called LA_Ex4.m:

```
clearvars
A=randi([-100,100],1000); B=randi([-100,   100],   1000,   1);
%% 1) inv() or pinv()
tic; Ai = inv(A); xyz1=Ai*B; T_inv=toc
%% 2) bacslash operator: \
clearvars
A=randi([-100,100],1000); B=randi([-100,   100],   1000,   1);
tic; xyz2 = A\B; T_backslash = toc
%% 3) mldivide()
clearvars
A=randi([-100,100],1000); B=randi([-100,   100],   1000,   1);
tic; xyz3= mldivide(A, B); T_mld = toc
%% 4) linsolve()
clearvars
A=randi([-100,100],1000); B=randi([-100,   100],   1000,   1);
tic; xyz4 = linsolve(A, B); T_linsolve =   toc
%% 5) lsqr()
clearvars
A=randi([-100,100],1000); B=randi([-100,   100],   1000,   1);
tic; xyz5 = lsqr(A, B); T_lsqr = toc
%% 6) lu()
clearvars
A=randi([-100,100],1000); B=randi([-100, 100], 1000, 1);
tic; [L, U, P]=lu(A); y=L\(P*B); xys6=U\y; T_lu=toc
```

```
%% 7) rref()
clearvars; A=randi([-100,100],1000); B=randi([-100, 100], 1000, 1);
tic; MA = [A, B];xyz7 = rref(MA); XYZ7=xyz7(:, end); T_rref=toc
%% 8) svd()
clearvars; A=randi([-100,100],1000); B=randi([-100, 100], 1000, 1);
tic; [U S V] = svd(A); xyz8 = V*inv(S)*U'*B; T_svd=toc
%% 9) chol()
clearvars; A=randi([-100,100],1000); B=randi([-100, 100], 1000, 1);
tic; [U L]= chol(A); xyz9 = U\(U'\B); T_chol=toc
%% 10) qr()
clearvars
A=randi([-100,100],1000); B=randi([-100, 100], 1000, 1);
tic; [Q R] = qr(A); xyz10 = R\Q.'*B ; T_qr=toc
%% 11) decomposition()
clearvars; A=randi([-100,100],1000); B=randi([-100, 100], 1000, 1);
tic; xyz11 = decomposition(A)\B; T_decom = toc
%%  12)  bicg()   Gradient methods
clearvars; A=randi([-100,100], 1000); B=randi([-100, 100], 1000, 1);
tic; xyz12 = bicg(A, B); T_bicg=toc
%% 13) solve()
A=randi([-100,100],100); B=randi([-100, 100], 100, 1);
tic;
x = sym('x', [1, 100]); x=x.';
Eqn = A*(x); Eqn = Eqn - B;
Solution = solve(Eqn); SOLs = struct2array(Solution); SOLs = double(SOLs);
x13 = SOLs';
T_solve=toc
```

Here are the elapsed computation time values from the simulations:

```
T_inv =
0.0390
T_backslash = 0.0173
T_mld =
0.0171
T_linsolve =
0.0171
```

lsqr stopped at iteration 20 without converging to the desired tolerance 1e-06 because the maximum number of iterations was reached.

The iterate returned (number 20) has relative residual 0.24.

```
T_lsqr = 0.0236
T_lu =
0.0235
T_rref =
10.1406
T_svd =
0.4263
T_chol =
0.0330
T_qr =
0.1045
T_decom =
0.0459
```

bicg stopped at iteration 20 without converging to the desired tolerance 1e-06 because the maximum number of iterations was reached.

The iterate returned (number 0) has a relative residual of 1.

```
T_bicg =
0.0195
T_solve =
14.6306
```

From these computations, it is clear that `linsolve()`, `mldivide`, and `\` (the backslash operator) (Gaussian elimination method) are the fastest among all the tested methods. The slowest and computationally costliest one is the `solve()` operator of the Symbolic MATH even when the size of the system was 10 times smaller. It is worth noting that the reduced row echelon method called `rref()` is the next slowest, after the `solve()` operator.

Let's consider another example to solve these four different methods, which are `\`, `linsolve()`, `inv()`, and `solve()`, discussed previously.

Example 5: Solving Linear Equations ([A]{x} = [b]) by Changing Values of [b]

This exercise is composed of two parts:

[1]. Solve the given linear system for unknowns *a*, *b*, and *c*.

$$\begin{cases} -0.072a - c = -12 \\ 0.12b - c = -9 \\ a + b = 50 \end{cases}$$

[2]. Solve the given system for unknowns *a*, *b*, and *c*. The third equation's value changes in the range of 50…250.

$$\begin{cases} -0.072a - c = -12 \\ 0.12b - c = -9 \\ a + b = 50\ldots250 \end{cases}$$

The system is rewritten in a matrix form as $[A]\{x\} = [B]$ and then solved directly for unknowns *a*, *b*, and *c*. Here is the solution script (`LA_Ex4.m`):

```
% PART 1.
% The given system is written from the Ax=B as [A]*[abc]=[B]
A=[.072, 0, -1; 0, .12, -1; 1 1 0];
B=[-12, -9, 50];

abc1=A\B'                    %#ok   % BACKSLASH \
abc2  =   linsolve(A,B')     %#ok   % LINSOLVE()
abc3  =  inv(A)*B'           %#ok   % INV
% SOLVE() in symbolic MATH
syms a b c; abc4=solve(0.072*a-c+12, 0.12*b-c+9, a+b-50);
abc4=double([abc4.a;   abc4.b;  abc4.c])  %#ok
% SOLVE()
%% Part II. %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% BACKSLASH \ ; LINSOLVE(); INV
tic; Bk=50:250;
a=zeros(numel(Bk),1); b=zeros(numel(Bk),1);
c=zeros(numel(Bk),1); A=[.072, 0, -1; 0, .12, -1; 1 1 0];
```

```
for ii=1:numel(Bk)
B=[-12; -9; Bk(ii)];
abc=A\B;
a(ii)=abc(1,:);
b(ii)=abc(2,:);
c(ii)=abc(3,:);
end Time1=toc;
fprintf('Computation time with  BACKSLASH: %3.3f   \n', Time1); clearvars
tic; Bk=50:250;
a=zeros(numel(Bk),1); b=zeros(numel(Bk),1);
c=zeros(numel(Bk),1); A=[.072, 0, -1; 0, .12, -1; 1 1 0];
for ii=1:numel(Bk)
B=[-12; -9; Bk(ii)];
abc=linsolve(A,B);
a(ii)=abc(1,:);
b(ii)=abc(2,:);
c(ii)=abc(3,:);
end
Time2=toc;
fprintf('Computation  time with LINSOLVE:   %3.3f   \n', Time2) clearvars
tic Bk=50:250;
a=zeros(numel(Bk),1); b=zeros(numel(Bk),1); c=zeros(numel(Bk),1); A=[.072,
0, -1; 0, .12, -1; 1 1 0];
for ii=1:numel(Bk)
   B=[-12; -9; Bk(ii)];
   abc=inv(A)*B;
   a(ii)=abc(1,:); b(ii)=abc(2,:); c(ii)=abc(3,:);
end
Time3=toc;
fprintf('Computation  time with  INV:   %3.3f   \n', Time3)
%% SOLVE() from symbolic math
clearvars; tic;
Bk=50:250;
a1=zeros(numel(Bk),1);b1=zeros(numel(Bk),1); c1=zeros(numel(Bk),1);
syms a b c
for ii=1:numel(Bk)
```

```
   abc=solve(0.072*a-c+12,0.12*b-c+9,a+b-Bk(ii));
   a1(ii)=double(abc.a);
   b1(ii)=double(abc.b);
   c1(ii)=double(abc.c);
end
Time4=toc;
fprintf('Computation  time with SOLVE:   %3.3f   \n', Time4)
```

Here are the results of the calculations from Part 1:

```
abc1 =
   15.6250
   34.3750
   13.1250
abc2 =
15.6250
34.3750
13.1250
abc3 =
15.6250
34.3750
13.1250
abc4 =
15.6250
34.3750
13.1250
```

Here are the results of the script from Part 2:

```
Computation time with BACKSLASH: 0.002
Computation time with LINSOLVE:  0.002
Computation time with INV:       0.002
Computation time  with  SOLVE:   22.066
```

From the computation time spent to compute solutions of the given linear system with three variables and 201 possible cases using four ways, it is clear that the least efficient way of solving linear equations is using the Symbolic Math toolbox's `solve()`

function. The backslash operator (\) and `linsolve()` and `inv()` methods all performed similarly. The solver `linsolve()`, \, and `inv()` methods are more than 11,033 times more efficient and faster than the `solve()` function.

Example 6: Linear Equations ([A]{*x*} = [b]) Applied for the Least Squares Method

This exercise demonstrates how to apply the principles of solving linear equations in the form of [A]{*x*} = [b] to solve the least squares problem to find best-fit model coefficients. In this exercise, we introduce the Vandermonde matrix expression to determine the polynomial fit models.

Here is the N-th order polynomial:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_3 x^3 + a_2 x^2 + ax + a_0$$

To compute the fit model $f(x_i)$, we set it equal to the measured data y_i: $f(x_i) = [y_i]$.

$$a_n x_1^n + a_{n-1} x_1^{n-1} + \ldots + a_3 x_1^3 + a_2 x_1^2 + a_1 x_1 + a_0 = y_1$$

$$a_n x_2^n + a_{n-1} x_2^{n-1} + \ldots + a_3 x_2^3 + a_2 x_2^2 + a_1 x_2 + a_0 = y_2$$

$$a_n x_3^n + a_{n-1} x_3^{n-1} + \ldots + a_3 x_3^3 + a_2 x_3^2 + a_1 x_3 + a_0 = y_3$$

$$: \quad : \quad \ldots \quad : \quad : \quad : \quad : = :$$

$$a_n x_m^n + a_{n-1} x_m^{n-1} + \ldots + a_3 x_m^3 + a_2 x_m^2 + a_1 x_m + a_0 = y_m$$

These expressions can be written as follows:

$$[V]\{a_i\} = [y_i]$$

Here, $[V]$ is the Vandermonde matrix, $\{a_i\}$ is the coefficients of the n-th order polynomial, and $[y_i]$ is the measured data points.

$$V = \begin{bmatrix} 1 & x_1 & x_1 & \cdots & x_1 \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ : & : & : & : & : \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix}; \; a_i = \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ : \\ a_n \end{Bmatrix}; \; y_i = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ : \\ y_m \end{bmatrix}$$

Or

$$V = \begin{bmatrix} x_1^n \ldots & x_1^2 & x_1 & 1 \\ x_2^n \ldots & x_2^2 & x_2 & 1 \\ & \vdots & & \\ x_m^n \ldots & x_m^2 & x_m & 1 \end{bmatrix}; \quad a_i = \begin{Bmatrix} a_n \\ \vdots \\ a_1 \\ a_0 \end{Bmatrix}; \quad y_i = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix}$$

Here, x_i and y_i are known, and a_i polynomial fit coefficient values are needed to be computed. Therefore, we can compute a_i from the next expression:

$$\{a_i\} = [V]^{-1} * [y_i]$$

Let's consider the following example.

Given test data:

Test #	Test1	Test2	Test3	Test4	Test5	Test6	Test7
Applied Load, [N]	10	20	30	40	50	60	70
Deflection, δ[m]	0.145	0.435	0.505	0.765	1.025	1.199	1.430

The task is to compute the fit model using Hooke's law formulation for linear elastic materials. The Hooke's law formulation is $F = k\delta$, where F is applied force in [N] and δ is a dependent variable, which is the deflection of an elastic material when F force is applied. And k is the stiffness coefficient of a material. Thus, the unknown variable here is k that will be computed using the least squares criterion.

First, we express the test data with respect to the system of linear equations [A]{x} = [b]. Here the applied force is the dependent variable [b], and the independent variable {x} corresponds to the resulted deflection δ. Therefore, in this exercise, the unknown variable is k, which is stiffness of the material. In this exercise, a first tricky point is how to compute the values of [A]. To compute the elements of [A], we use the Vandermonde matrix approach. According to Hooke's law, it is a first-order polynomial, i.e., $F(\delta) = k\delta$, that can be also written as $k = F(\delta)/\delta$. Using the given data in this exercise, we can define the Vandermonde matrix and load matrix.

$$V = \begin{bmatrix} \delta_1 & 0 \\ \delta_1 & 0 \\ \vdots & \vdots \\ \delta_n & 0 \end{bmatrix} = \begin{bmatrix} 0.145 & 0 \\ 0.435 & 0 \\ \vdots & \vdots \\ 1.430 & 0 \end{bmatrix} F = \begin{bmatrix} 10 \\ 20 \\ \vdots \\ 70 \end{bmatrix}$$

Here, V is the Vandermonde matrix. Note the size of the Vandermonde matrix is 7-by-2 and the size of the applied load is 7-by-1. Therefore, the size of the stiffness matrix will be 1-by-2. The reason of having zeros in the second column of [V] is that according to Hooke's law, the linear relationship between the applied load and deflection of a linear elastic material is in the form of $f(x) = a_1 * x + a_0$ and $a_0 = 0$. Therefore, the unknown stiffness is found from the following:

$$k = [V]^{-1} * [F]$$

Note that to compute the values of $[k]$ in a more efficient and exactly, we employ the backslash (\) operator. An alternative solution function to the backslash operator is `linsolve()` or `mldivide()`.

The final solution script (`LA_Ex6.m`) is shown here:

```
% LA_Ex6.m
% Part 1. Vandermonde matrix
clc; clear variables
F = (10:10:70).';                                         % Applied Load
d = [0.145  0.435 0.505 0.765 1.025 1.199 1.430].';  % Deflection
scatter(F, d, 'filled')
ylim([0, max(d)+.2]),shg
A = [F zeros(size(F))];
FM =A\d;
FM_values = FM(1)*F;
hold on
plot(F, FM_values, 'k-', 'linewidth', 2)
gtext(['Fit model: F = '  num2str(FM(1)) '*\delta'])
gtext(['Stifness is: '  num2str(FM(1))])
grid on
xlabel('Applied Load, F [N]')
ylabel('Deflection, \delta [m]')
```

Figure 7-16 shows the resulted plot of the calculations from the script.

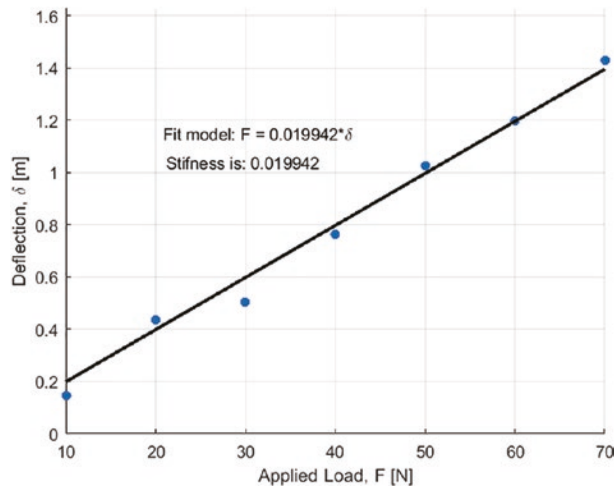


Figure 7-16. *Fit model is computed using the least squares method*

There are a few functions (polyfit, fitlm, fit) in Curve Fitting and Statistics and Machine Learning Toolboxes, which can be used easily to compute approximation polynomials. Let's look at the previous example of how to employ these functions:

```
% Part 2. Polynomial Approximation Fcn: Curve Fitting Toolbox
FM2 = polyfit(F,d, 1);
fprintf('CFTOOL Fit Model: F(d) = %f*d \n', FM2(1));
% Part 3. Polynomial Approximation Fcn: Stats and ML Toolbox
FM3 = fitlm(F,d, 'linear');
fprintf('Stats and ML Fit Model: F(d) = %f*d \n', FM3.Coefficients.
Estimate(2));
```

Parts 2 and 3 of the code (LA_Ex6.m) produce close approximation coefficients of the first-order polynomial. The following results will be displayed in the Command window:

```
CFTOOL Fit Model: F(d) = 0.021082*d
Stats and ML Fit Model: F(d) = 0.021082*d
```

Note that there is a small difference between the Vandermonde approach and polyfit() and fitlm() functions. The reason for the difference is the intercept value is set equal to "0" with the Vandermonde matrix, and with the other two functions, the intercept is considered.

Example 7: Linear Equations ([A]{*x*} = [b]) Applied for the Least Squares Method

The following data table gives the stopping distance y as a function of initial speed v, for certain car model. Find the quadratic polynomial coefficients that fit the data.

v(*km*/*h*)	**20**	**30**	**40**	**50**	**60**	**70**
y(*m*)	45	80	130	185	250	330

The Vandermonde matrix of this exercise for the quadratic fit model is computed from the following:

$$V = \begin{bmatrix} v_1^0 & v_1 & v_1^2 \\ v_2^0 & v_2 & v_2^2 \\ : & : & : \\ v_n^0 & v_n & v_n^2 \end{bmatrix}$$

Note that $v_1^0, v_2^0, \ldots v_n^0 = 1$ corresponds to a_0. Therefore, V can be also expressed as follows:

$$V = \begin{bmatrix} 1 & v_1 & v_1^2 \\ 1 & v_2 & v_2^2 \\ : & : & : \\ 1 & v_n & v_n^2 \end{bmatrix}$$

Note that V can be also expressed as follows:

$$V = \begin{bmatrix} v_1^2 & v_1 & 1 \\ v_2^2 & v_2 & 1 \\ : & : & : \\ v_n^2 & v_n & 1 \end{bmatrix}$$

The Vandermonde matrix of the data from this exercise is equal to the following:

$$V = \begin{bmatrix} 1 & 20 & 20^2 \\ 1 & 30 & 30^2 \\ : & : & : \\ 1 & 70 & 70^2 \end{bmatrix} \text{ or } V = \begin{bmatrix} 20^2 & 20 & 1 \\ 30^2 & 30 & 1 \\ : & : & : \\ 70^2 & 70 & 1 \end{bmatrix}$$

The measured data points in this exercise are as follows:

$$y_i = \begin{bmatrix} 45 \\ 80 \\ : \\ 330 \end{bmatrix}$$

The unknown coefficient of the quadratic polynomial is found from the following, depending on which way [V] is defined:

$$a = [a_0, a_1, a_2] \text{ or } a = [a_2, a_1, a_0]$$

$$a = V^{-1} * [y_i]$$

Note that in this exercise, the size of the Vandermonde matrix is 6-by-3.

The complete code of this exercise is `LA_Ex7.m`.

```
% LA_Ex7.m
clc; clear variables; close
% Part 1. Vandermonde matrix
v = (20:10:70).';                     % Velocity, [km/h]
y = [45  80 130 185 250 330].';       % Braking distance, [m]
scatter(v, y, 'filled')
ylim([0, max(y)+.2])
A = [v.^2, v,  ones(size(v))];
FM =A\y;
FM_values = FM(1)*v.^2+FM(2)*v+FM(3);
hold on
plot(v, FM_values, 'k-', 'linewidth', 2)
gtext(['Fit model: s(v) = '  num2str(FM(1))  'v^2 +' num2str(FM(2)) '*v +',
num2str(FM(3))])
grid on
```

```
xlabel('\it Velocity, v [km/h]')
ylabel('\it Braking Distance, s [m]')
% Part 2. Polynomial Approximation Fcns: Curve Fitting Toolbox
FM2 = polyfit(v,s, 2);
fprintf('CFTOOL Fit Model: s(v) = %f*v.^2 + %f*v + %f \n', FM2);
% Part 3. Polynomial Approximation Fcn: Stats and ML Toolbox
FM3 = fitlm(v, s, 'poly2');
fprintf('Stats and ML Fit Model: s(v) = %f*v.^2 + %f*v + %f \n', flip(FM3.
Coefficients.Estimate));
```

Figure 7-17 shows the simulation results of `LA_Ex7.m`.

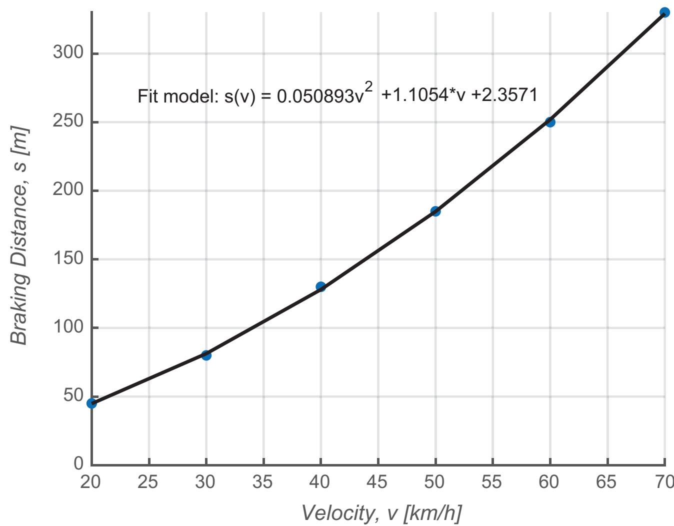


Figure 7-17. *Quadratic fit model is computed using the least squares method*

Also, in the Command window, the following outputs will be displayed after executing the script: `LA_Ex7.m`:

```
FMM =

    0.0509
    1.1054
    2.3571
CFTOOL Fit Model: s(v) = 0.050893*v.^2 + 1.105357*v + 2.357143
Stats and ML Fit Model: s(v) = 0.050893*v.^2 + 1.105357*v + 2.357143
```

The results from the three approaches are identical, which proves that the Vandermonde approach is well correlated with the functions of the two toolboxes.

Example 8: Linear Equations ([A]{x} = [b]) Applied for the Least Squares Method Using Simulink Modeling

The following data table gives the stopping distance y as a function of initial speed v, for a certain car model. Find the quadratic polynomial coefficients that fit the data.

$v(km/h)$	20	30	40	50	60	70
$y(m)$	45	80	130	185	250	330

Let's build a Simulink model to solve this exercise and apply the least squares polynomial solver block. A Simulink model of this exercise is relatively simple and composed of three blocks: Constant, Least Squares Polynomial Fit, and Display blocks, as shown in Figure 7-18.

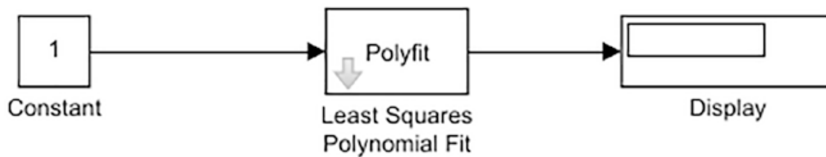


Figure 7-18. *Simulink model, the least squares method*

The Simulink model shown in Figure 7-18 is not complete yet. There are two more adjustments to be made in the Constant and Least Squares Polynomial Fit blocks. The Constant should be opened by double-clicking it, and the data for y, i.e., [45 80 130 185 250 330].' should be entered. Note the data has to be a column vector. Then the next block parameters should be adjusted, as shown in Figure 7-19. Note that Control Parameter (X) values are v values in a column vector form, and Polynomial order (N) is 2 because we are looking for a quadratic polynomial fit.

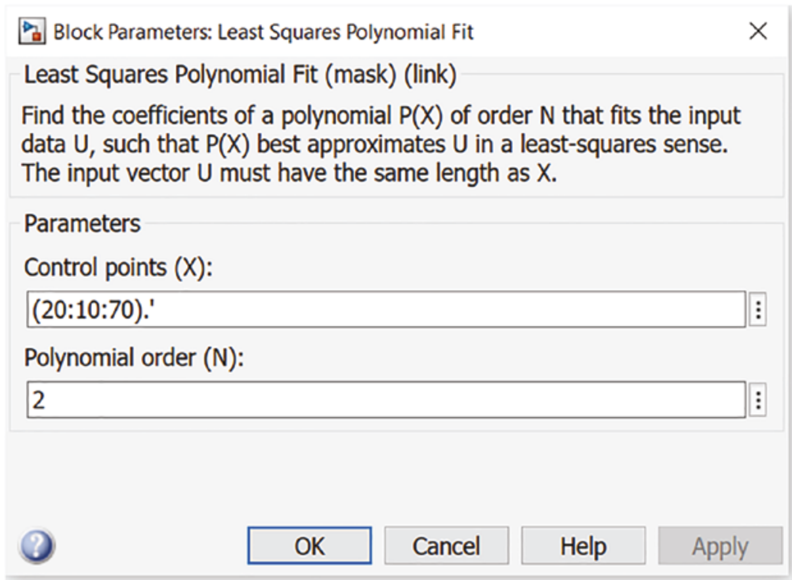


Figure 7-19. Least squares Polynomial Fit block parameters adjustment

Once all adjustments are made and values are entered, the model is ready to simulate. The completed model (LA_Ex8.slx) with simulation results after resizing the Display block to see all results is shown in Figure 7-20.

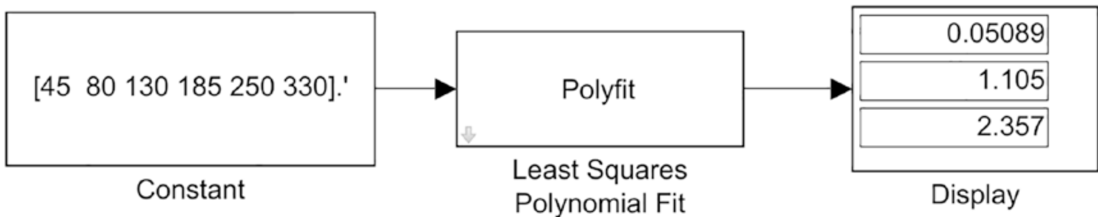


Figure 7-20. Simulink model, LA_Ex8.slx

Note that the found results from the Simulink model `LA_Ex8.slx` match perfectly well with the ones found using the Vandermonde matrix, `polyfit()` and `fitlm()`.

Matrix Operations

This section covers general mathematical operations and computations of matrices, vectors, and eigen-vectors. Many numerical examples are used to explain the matrix operations. Table 7-1 lists the matrix operations their command syntax.

Table 7-1. Matrix Operators in Two Equivalent Formulations

Operation Name	MATLAB First Way	MATLAB Second Way
Matrix multiplication	A*B	mtimes(A,B)
Array-wise multiplication	A.*B	times(A,B)
Matrix right division	A/B	mrdivide(A,B)
Array-wise right division	A./B	rdivide(A,B)
Matrix left division	A\B	mldivide(A,B)
Array-wise left division	A.\B	ldivide(A,B)
Matrix power	A^B	mpower(A,B)
Array-wise power	A.^B	power(A,B)
Complex transpose	A'	ctranspose(A)
Matrix transpose	A.'	transpose(A)
Binary addition	A+B	plus(A,B)
Unary plus	+A	uplus(A)
Binary subtraction	A-B	minus(A,B)
Unary minus	-A	uminus(A)
Determinant	det(A)	det(A)
Rotate by 90^0	rot90(A)	rot90(A)
Replicate and tile an array n times	repmat(A, n)	repmat(A, n)
Flip matrix left/right	fliplr(A)	fliplr(A)
Flip matrix in up/down	flipud(A)	flipud(A)

Basic MATLAB unit data is in the array type format. Matrices and vectors can be employed in many cases to define input and output, local data, and function inputs and outputs. Moreover, they can be used to combine separate scalars into one signal and process multidimensional input and output signals. An array is defined by a single name and a collection of data arranged by rows and columns, as shown here.

Row # 1 ⇨	A_{11}	A_{12}	A_{13}
Row # 2 ⇨	A_{21}	A_{22}	A_{23}
Row # 3 ⇨	A_{31}	A_{22}	A_{33}
Row # 4 ⇨	A_{41}	A_{42}	A_{43}
	⇧ Column # 1	⇧ Column # 2	⇧ Column # 3

Let's look at some numerical examples. They perform matrix operations with scalars, such as addition, subtraction, power, multiplication, and division, including array-wise (elementwise) operations in the Command window.

```
>>   A=[8,1,6;   3,5,7;   4,9,2]   % Matrix 3-by-3
A =
     8     1     6
     3     5     7
     4     9     2
>> a = 2; b = 2+3i; c = 5j;
>> B=A^a  % Note the difference between ^ and .^
B =
    91    67    67
    67    91    67
    67    67    91

>> C=A.^a  % Elementwise. Note the difference between ^ and .^
C =
    64     1    36
     9    25    49
    16    81     4
```

```
>> D = A*a+B/b
D =
  30.0000 -21.0000i  12.3077 -15.4615i  22.3077 -15.4615i
  16.3077 -15.4615i  24.0000 -21.0000i  24.3077 -15.4615i
  18.3077 -15.4615i  28.3077 -15.4615i  18.0000 -21.0000i
>> E =  C./c
E =
   0.0000 -12.8000i   0.0000 - 0.2000i   0.0000 - 7.2000i
   0.0000 - 1.8000i   0.0000 - 5.0000i   0.0000 - 9.8000i
   0.0000 - 3.2000i   0.0000 -16.2000i   0.0000 - 0.8000i
>> F = C/c
F =
   0.0000 -12.8000i   0.0000 - 0.2000i   0.0000 - 7.2000i
   0.0000 - 1.8000i   0.0000 - 5.0000i   0.0000 - 9.8000i
   0.0000 - 3.2000i   0.0000 -16.2000i   0.0000 - 0.8000i
```

Example: Performing Matrix Operations

Given six arrays: A (4 – by – 3), B(3 – by – 4), C(4 – by – 4), D(4 – by – 3), $E(3 - by - 3)$, and $F(3 - by - 3)$.

Let's perform several matrix operations—such as summation, subtraction, multiplication, power, scalar multiplication, square root, mean, round, standard deviations, and replicate/rotate/flip matrix—from the Command window.

```
>> A=[2 -3 1; 3 2 5; 1 3 4; -3 -2 3] ;
>> B=[3,4,-2 1;2,5,4,-6;4,-3, 1,2] ;
>> C=[16,2,3,13;5,11,10,8;9 4 7 14;6 15 12 1] ;
>> D=[1 2 3; 2 3 4; 4 3 1; -2 -3 1] ;
>> E=[8, 1, 6; 3, 5, 7; 4, 9, 2];
>> F=[3 7 3; 3 2 8; 9 2 1];
>> M_AB = A*B
M_AB =
     4   -10   -15    22
    33     7     7     1
    25     7    14    -9
    -1   -31     1    15
```

```
>> M_BA = B*A
M_BA =
    13    -9    18
    41    28    25
    -6   -19    -1

>> M_S = M_AB-C
M_S =
   -12   -12   -18     9
    28    -4    -3    -7
    16     3     7   -23
    -7   -46   -11    14
>> M_S= M_BA-C
```

Matrix dimensions must agree.

```
>> CM=C*M_S   % Not equivalent to M_S*C
CM =
  -179  -789  -416   243
   352  -442  -141  -150
    18  -747  -279    88
   533  -142   -80  -313
>> CM1=M_S*C   % Not equivalent to C*M_S
CM1 =
  -360   -93  -174  -495
   359  -105   -61   283
   196  -252  -149   307
  -357  -354  -390  -599
>> CM2=M_S.*C   % Elementwise  operation:  NOT  equivalent  to  M_S*C
CM2 =
  -192   -24   -54   117
   140   -44   -30   -56
   144    12    49  -322
   -42  -690  -132    14
```

```
>> MDE=M_S./C   % Elementwise operation:  NOT  equivalent  to  M_S/C
MDE =
   -0.7500   -6.0000   -6.0000    0.6923
    5.6000   -0.3636   -0.3000   -0.8750
    1.7778    0.7500    1.0000   -1.6429
   -1.1667   -3.0667   -0.9167   14.0000

>> MD=M_S/C   % Not equivalent to M_S./C
MD =
    1.9275    8.5704   -5.6271   -5.8414
    1.4496   -6.4076    1.5420    3.8277
   -1.0389  -10.8246    5.0116    6.9401
   -4.9118  -12.9118   12.6765    3.6765
>> M_AD =A.*D % Elementwise operation: matrix multiplication
M_AD =
     2    -6     3
     6     6    20
     4     9     4
     6     6     3
>> MM_AD= A*D    % Error due to size mismatch of [A] and [D]
```

Error using * Incorrect dimensions for matrix multiplication. Check that the number of columns in the first matrix matches the number of rows in the second matrix. To perform elementwise multiplication, use '.*'. Related documentation

```
>> M_EF=E.*F   % Elementwise multiplication of square matrices
M_EF =
    24     7    18
     9    10    56
    36    18     2
>> MM_EF=E*F   % Square matrices can be multiplied matrix-wise
MM_EF =
    81    70    38
    87    45    56
    57    50    86
```

```
>> Csqrt=sqrt(C) % Not equivalent to sqrtm(C)
Csqrt =
    4.0000    1.4142    1.7321    3.6056
    2.2361    3.3166    3.1623    2.8284
    3.0000    2.0000    2.6458    3.7417
    2.4495    3.8730    3.4641    1.0000
>> Csqrt=sqrtm(C) % Not equivalent to sqrt(C)
Csqrt =
   3.8335 - 0.0167i   0.0738 + 0.7839i   0.1262 + 0.3666i   1.7975 - 1.1337i
   0.3251 + 0.0011i   2.6850 - 0.0526i   1.6850 - 0.0246i   1.1359 + 0.0761i
   1.3123 - 0.0237i   0.7322 + 1.1107i   1.9687 + 0.5194i   1.8178 - 1.6064i
   0.5925 + 0.0373i   2.0922 - 1.7477i   1.7997 - 0.8172i   1.3466 + 2.5276i
>>  C_E1 = expm(C)   % Matrix exponential not equal to  exp(C)
C_E1 =
   1.0e+14 *
    1.5718    1.3711    1.3622    1.5295
    1.5718    1.3711    1.3622    1.5295
    1.5718    1.3711    1.3622    1.5295
    1.5718    1.3711    1.3622    1.5295
>> C_E2 = exp(C) %  Exponential  of  a  matrix:  not  equal  to  expm(C)
C_E2 =
   1.0e+06 *
    8.8861    0.0000    0.0000    0.4424
    0.0001    0.0599    0.0220    0.0030
    0.0081    0.0001    0.0011    1.2026
    0.0004    3.2690    0.1628    0.0000
>> S=[A(1,1:3); B(2,1:3);C(3,2:4)]; % Created from the existed
>> Y=[A(1), 1.3];                   % Created from the existed
>> Arot90=rot90(A)                  % Matrix rotate
```

```
Arot90 =
     1     5     4     3
    -3     2     3    -2
     2     3     1    -3
>> Crep=repmat(C,  2,1)   % Matrix replication/copy
Crep =
    16     2     3    13
     5    11    10     8
     9     4     7    14
     6    15    12     1
    16     2     3    13
     5    11    10     8
     9     4     7    14
     6    15    12     1
>> Bflip=fliplr(B)   % Matrix flip
Bflip =
     1    -2     4     3
    -6     4     5     2
     2     1    -3     4
Cud=flipud(Crep)   % Matrix flip up or down
Cud =
     6    15    12     1
     9     4     7    14
     5    11    10     8
    16     2     3    13
     6    15    12     1
     9     4     7    14
     5    11    10     8
    16     2     3    13
```

Many of these matrix operations can also be performed in the Simulink environment. Let's use the previous examples to demonstrate how and what Simulink uses for matrix operations and manipulations.

The Simulink Library contains the blocks for sum, multiplication/division, power, exponent, and concatenation, as shown in Figure 7-21.

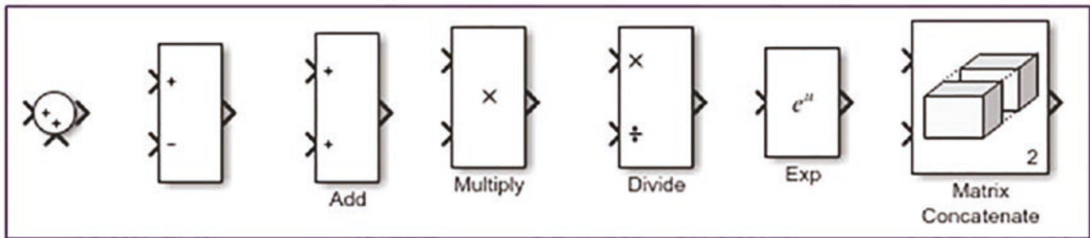


Figure 7-21. Matrix operation blocks in the Simulink Library

First define the [A] and [D] matrices in the Command window.

```
>> A=[2 -3 1; 3 2 5; 1 3 4; -3 -2 3] ;
>> D=[1 2 3; 2 3 4; 4 3 1; -2 -3 1] ;
```

Now compute the sum and subtraction of matrices [A] and [D], as shown in Figure 7-22.

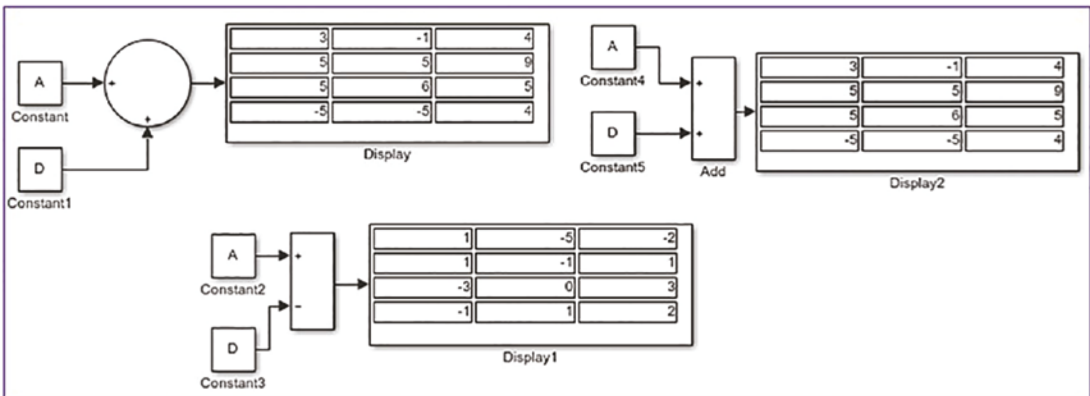


Figure 7-22. Matrix sum and subtraction operations in Simulink

Note that matrices [A] and [D] are defined via the Command window and workspace. The computed sums match the ones calculated using MATLAB's Command window.

Here are the results of multiplication (see Figure 7-23), exponent, and square (see Figure 7-24) of the matrices.

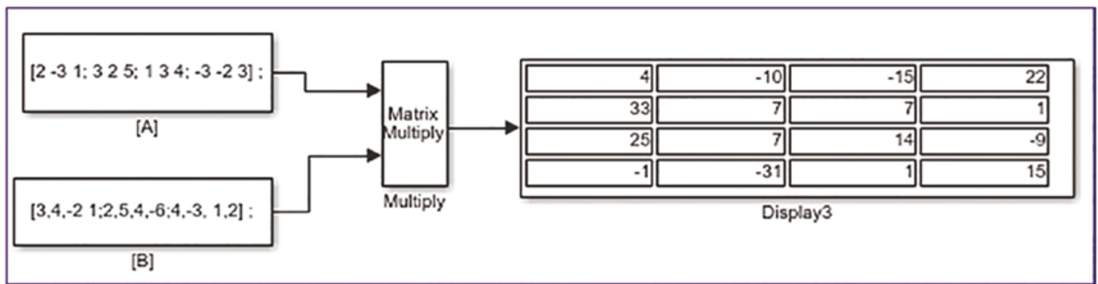


Figure 7-23. Matrix multiplication in Simulink

Note that for the matrix multiplication operation shown in Figure 7-23, the Multiply block changes from element-wise (.*) multiplication to matrix (*) multiplication, as shown in Figure 7-24.

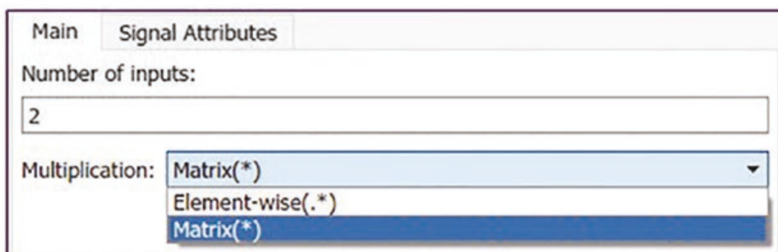


Figure 7-24. Setting up the Matrix Multiply block for matrix multiplication (*) or element-wise multiplication (.)*

Otherwise, the multiplication operation will not be performed due to the mismatched sizes of [A] and [B]. Again, the computed results match the ones from MATLAB.

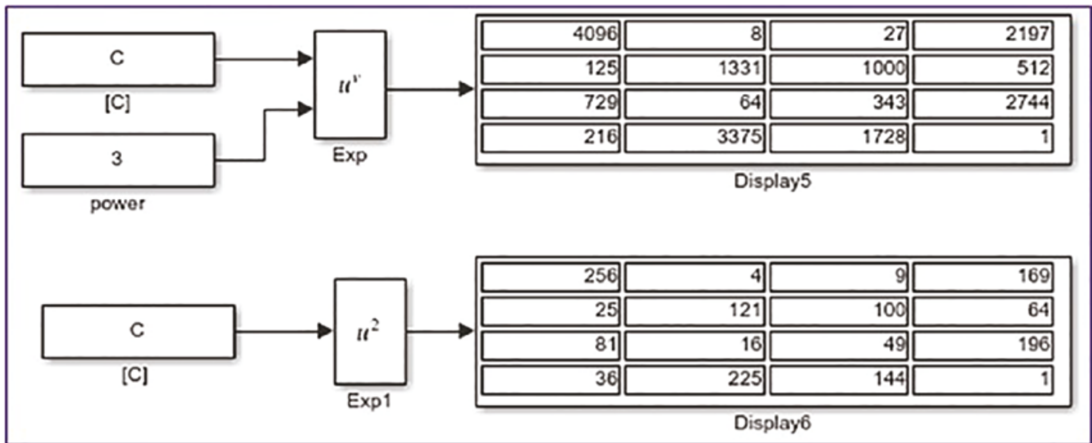


Figure 7-25. *Matrix exponential and square operation blocks*

Note that in the operations in Figure 7-25, the exponential and power operations are performed with one block (one Math Function block), by choosing its Function type [pow] in u^v and [square] in u^2 (see Figure 7-26).

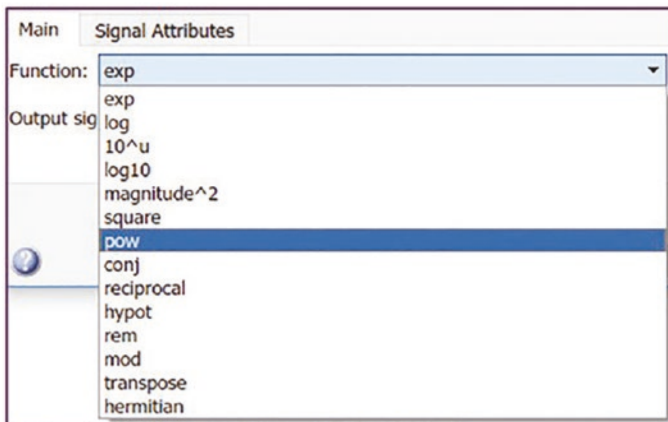


Figure 7-26. *How to set the Math Function block for matrix operations*

Now by using the Matrix Concatenate block, we create a new matrix (4-by-10) from the computed the matrix sum (4-by-3), square (4-by-4), and matrix division (4-by-3). See Figure 7-27.

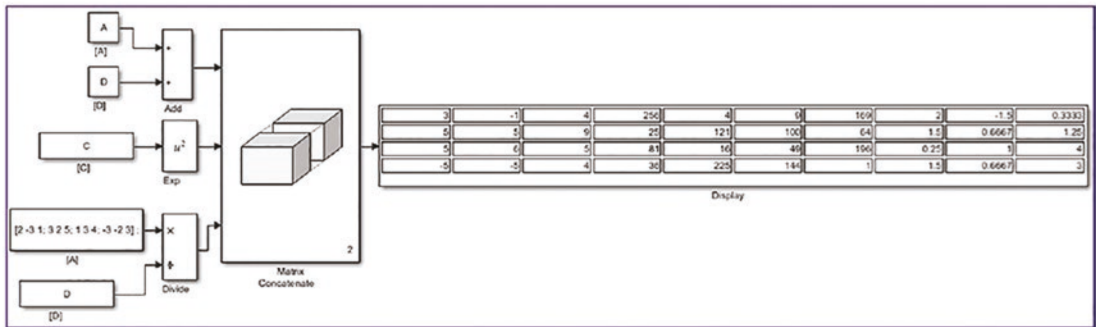


Figure 7-27. *The Matrix Concatenate block performs matrix concatenation.*

As demonstrated, Simulink blocks perform various matrix operations, much like MATLAB functions. However, there are computationally costly simulations with matrix and array operations in which Simulink models might be slower than MATLAB scripts. For example, when computing discrete Fourier transforms, Simulink models are much slower than MATLAB. For some matrix and array operations, the MATLAB Fcn block or the Interpreted MATLAB Fcn block can be used in Simulink modeling.

In addition to these matrix operations, there are a few other operations by which you can create new matrices. For instance, you can take out diagonals of existing matrices with `diag(A)` or take out selected elements of matrices and create a new matrix.

```
>> E=[8, 1, 6; 3, 5, 7; 4, 9, 2];
>> F=[3 7 3; 3 2 8; 9 2 1];
>> EF = [diag(E),  diag(F)]
EF =
8    3
5    2
2    1
```

Standard Matrix Generators

MATLAB has numerous standard array and matrix generators, which can be used to generate a wide range of matrices. For instance, `eye(n)`, `eye(k, m)`, `ones(m)`, `ones(m, k)`, `zeros(l)`, `zeros(l,k)`, `magic(k)`, `pascal(k)`, `pascal(k, m)`, `rand(m)`, `rand(k, m)`, `randi(n,m,k)`, `repmat(A, r, c)`, `blkdiag(A, B, C)`, `sparse(m,n)`, and many more. Here's an example:

```
>> eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
>> magic(5)  %   Magic   matrix in a size of 5 by 5
ans =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
>>  A=pascal(4)   % Pascal  matrix  in  a  size  of  4  by  4
A =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
>> A=pascal(4,2)   % Pascal matrix in a size of 4 by 4
A =
    -1    -1    -1    -1
     3     2     1     0
    -3    -1     0     0
     1     0     0     0
>> zeros(3)  % Zero matrix 3-by-3
ans =
     0     0     0
     0     0     0
     0     0     0
>> zeros(2,3)  % Zero matrix 2-by-3
ans =
     0     0     0
     0     0     0
```

```
>> ones(3)  % Ones matrix 3-by-3
ans =
     1     1     1
     1     1     1
     1     1     1
>> ones(2,3)  % Ones matrix 2-by-3
ans =
     1     1     1
     1     1     1
>> eye(3,4)   % Unit diagonal matrix of size  3  - by - 4
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
>> eye(4,5)   % Unit diagonal matrix of size 4 - by - 5
ans =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
>> rand(2)    % Uniform random matrix 2-by-2
ans =
    0.8147    0.1270
    0.9058    0.9134
>> rand(2, 4)    % Uniform random matrix 2-by-4
ans =
    0.6324    0.2785    0.9575    0.1576
    0.0975    0.5469    0.9649    0.9706
>> randn(3)    % Normally distributed random matrix 3-by-3
ans =
    0.7254   -0.2050    1.4090
   -0.0631   -0.1241    1.4172
    0.7147    1.4897    0.6715
```

```
>> A = round(randn(3))   % Round up to the nearest 0
A =
    -1     0     0
     1     1     0
     2     1    -1
>> A_rep=repmat(A, 2, 3) % replicating the matrix A by making its
% replication 2 times of rows and 3 times of columns
A_rep =
    -1     0     0    -1     0     0    -1     0     0
     1     1     0     1     1     0     1     1     0
     2     1    -1     2     1    -1     2     1    -1
    -1     0     0    -1     0     0    -1     0     0
     1     1     0     1     1     0     1     1     0
     2     1    -1     2     1    -1     2     1    -1

>> C=eye(2); B=magic(3); A=ones(4);
>> D=blkdiag(A,B,C)   % combine matrices in diagonal directions to
% create a block diagonal matrix.
D =
     1     1     1     1     0     0     0     0     0
     1     1     1     1     0     0     0     0     0
     1     1     1     1     0     0     0     0     0
     1     1     1     1     0     0     0     0     0
     0     0     0     0     8     1     6     0     0
     0     0     0     0     3     5     7     0     0
     0     0     0     0     4     9     2     0     0
     0     0     0     0     0     0     0     1     0
     0     0     0     0     0     0     0     0     1
>> randi([-13, 13], 5) % Random  integers within [-13, 13]
ans =
     0     7    12     9    -4
    12    -7     1    -7    -8
    -4     0   -10     8    -7
     2     5    -9    -7     3
    -7    11    -7    12    -1
```

```
>> K=reshape(randperm(9), 3,3) % Change the size (reshape)
of array % to make 3 by 3 matrix by random permutation
K =
     6     4     7
     1     3     2
     9     8     5
```

In addition, there are a few dozen matrix generation functions. They are the gallery of test matrices, such as `binomial`, `cauchy`, `clement`, `invol`, `house`, `krylov`, `leslie`, `lesp`, `neumann`, `poisson`, `ris`, `rando`, `smoke`, `wilk`, and many more. In general, the command syntax of these matrices is as follows:

```
[A, B, C,...] = gallery(matname,P1,P2,...);
[A, B, C,...] = gallery(matname,P1,P2,..., classname);
A=gallery(3);
B=gallery(5);
```

To get more information about the gallery of matrices, type this in the Command window:

```
>> help gallery
>> doc gallery
```

Here are several examples of how to employ gallery matrices:

```
>> S=[3 2 7]; X=[2 2];
% This is the 3-by-3 Leslie population matrix taken from the model with
average birth numbers S(1:n) and survival rates X(1:n-1)
>> L=gallery('leslie', S, X)
L =
     3     2     7
     2     0     0
     0     2     0
% Chebyshev spectral differentiation matrix of order 3
>> C = gallery('chebspec', 3,1)
C =
   -0.3333   -1.0000    0.3333
    1.0000    0.3333   -1.0000
   -1.3333    4.0000   -3.1667
```

```
% Cauchy matrix 3-by-3, C(I, j) = 1/(S(i)+Y(j)). The arguments S and Y are
vectors of length 3.
% If you pass in scalars for S and Y, they are interpreted as vectors 1:S
and 1:Y.
>> S = [3 2 6]; Y = [1 3 2];
>> C = gallery('cauchy', S, Y)
C =
    0.2500    0.1667    0.2000
    0.3333    0.2000    0.2500
    0.1429    0.1111    0.1250
>> % Krylov matrix of size 5-by-5.
>>  B  =  gallery('krylov',  randn(5))
B =
    1.0000    2.4392    3.9250   26.5823   24.9976
    1.0000    1.2031    7.8039    6.5275   61.9487
    1.0000   -1.3094   -7.4622   11.6113  -14.5418
    1.0000    0.3038   -3.8311  -16.0811  -10.1830
    1.0000   -3.7454    0.3824    5.7186  -65.2352
>> % House-holder matrix of size 3-by-1.
>> A = [3;2;5];    % Must be a column matrix
>> H =   gallery('house', A)
H =
    9.1644
    2.0000
    5.0000
>> % Hankel matrix of size 5-by-5 with elements H(I, j)=0.5/(n-i-j+1.5).
>> B  =  gallery('ris',5)
B =
    0.1111    0.1429    0.2000    0.3333    1.0000
    0.1429    0.2000    0.3333    1.0000   -1.0000
    0.2000    0.3333    1.0000   -1.0000   -0.3333
    0.3333    1.0000   -1.0000   -0.3333   -0.2000
    1.0000   -1.0000   -0.3333   -0.2000   -0.1429
```

```
>> % Smoke matrix  of size 3-by-3 - complex, with "smoke ring" pseudo-
spectrum.
>> SM=gallery('smoke', 3)
SM =
  -0.5000 + 0.8660i   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i  -0.5000 - 0.8660i   1.0000 + 0.0000i
   1.0000 + 0.0000i   0.0000 + 0.0000i   1.0000 + 0.0000i
```

These standard and gallery matrices have special properties that can be of great use in various numerical simulations and analysis problems. For instance, these standard matrices—`ones()`, `eye()`, `zeros()`, `rand()`, `randn()`—are used often for signal processing, data analysis, and memory allocation in large computations.

Vector Spaces

In signal processing, numerical analyses, and building computer simulation models, vector spaces are very important. For instance, the logarithmic space is used for digital signal processing when frequencies go over a unit circle. There are several straightforward ways by which vectors, vector spaces, and arrays with equal spaces between their elements can be created. Let's suppose that we need to create a vector W that begins with a value $w1$ and ends with $w2$, as shown in Figure 7-28.

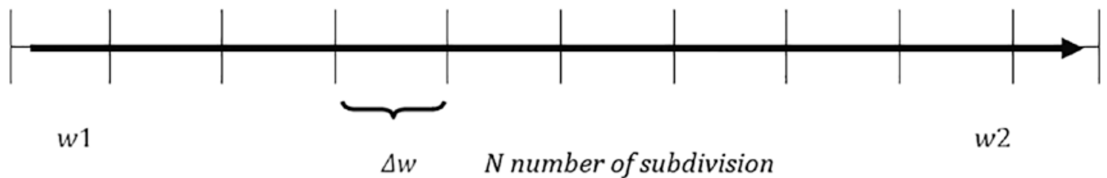


Figure 7-28. *Vector space*

If the size Δw is known, then the space can be expressed by $W = w1{:}\ \Delta w{:}\ w2$. For instance, the whole space can be defined in terms of $w1 = 1$, $w2 = 13$, $\Delta w = 0.1$ with the following:

```
>> w=1:0.1:13;
```

Moreover, if *N* number of points between the start and end boundaries of a space are known, the linear space function `linspace()` can be used.

```
>> % This creates a linear space of w array with equally spaced k number of
elements
>> w=linspace(1, 13, N);
```

Note If *N* is not specified in the `linspace()` command, its default value is 100.

This simplex example generates sound waves with a sine function.

```
fs=3e4;      % Sampling frequency
% Different signal frequencies:
f1=100; f2=200; f3=300; f4=400; f5=500; f6=600;
t=0:1/fs:5;  % Time
% Signal: sum of sine waves
x=sin(2*pi*t*f1)+sin(2*pi*t*f2)+sin(2*pi*t*f3)+
sin(2*pi*t*f4)+sin(2*pi*t*f5);
[m,  n]=size(x);            % Gets the size of the created vector space
sound(x,  fs)               % Plays a created sound & hear from sound cards
```

The `linspace()` command creates linearly spaced vector spaces/arrays. In MATLAB, there is another similar function, called `logspace()`, that creates logarithmic scaled vector spaces. For example, you use the following command to create a logarithmic space of the x array containing 130 logarithmically spaced elements (here, $N = 130$) between boundary points 0 and 13:

```
>> x=logspace(1,13,130);
```

Likewise, use this command to create 50 logarithmic spaced points between 0 and π:

```
>> s=logspace(0, pi);
```

Note If *N* is not specified in `logspace()`, then its default value is 50.

Polynomials Represented by Vectors

For numerical simulations in MATLAB, polynomials are represented via vectors using coefficients of polynomials in descending order. For instance, a fifth-order polynomial is given as follows:

$$12x^5 + 13x^4 - 15x^2 + 17x - 13$$

That is defined as a vector space in the following manner:

```
>>  f = [12, 13, 0, -15, 17, -13];
```

Note, that MATLAB reads vector entries as a vector of length *n+1* as an *n*-th order polynomial. Thus, if any of the given polynomial misses any coefficients, zero has to be entered for its coefficient. For instance, in the previous example, 0 is entered for the coefficient of x^3.

There are several functions that can be used to compute the roots of polynomials. They are as follows:

- Using the `roots()` MATLAB function
- Using the `zero()` Control System Toolbox function
- Using the `solve()` Symbolic MATH Toolbox function

You find roots of the given polynomial using the base MATLAB function, `roots()`.

```
>> x_sols=roots(f)
x_sols =
  -1.2403 + 0.9412i
  -1.2403 - 0.9412i
   0.7941 + 0.0000i
   0.3015 + 0.6869i
   0.3015 - 0.6869i
```

Note that the given polynomial has only one real value root and four complex valued roots.

The roots are computed by using the `solve()` function of MATLAB to find symbolic solutions of the polynomial, and then solutions are converted (note that conversion may be not necessary) to obtain a shorter number of decimal point numeric data using the `double()` function with the following entries in the Command window:

```
>> syms x
>> syms x
>> Sol=solve(12*x^5+13*x^4-15*x^2+17*x-13)
Sol =
root(z^5 + (13*z^4)/12 - (5*z^2)/4 + (17*z)/12 - 13/12, z, 1)
root(z^5 + (13*z^4)/12 - (5*z^2)/4 + (17*z)/12 - 13/12, z, 2)
root(z^5 + (13*z^4)/12 - (5*z^2)/4 + (17*z)/12 - 13/12, z, 3)
root(z^5 + (13*z^4)/12 - (5*z^2)/4 + (17*z)/12 - 13/12, z, 4)
root(z^5 + (13*z^4)/12 - (5*z^2)/4 + (17*z)/12 - 13/12, z, 5)
>> double(Sol)
ans =
   0.3015 - 0.6869i
   0.3015 + 0.6869i
   0.7941 + 0.0000i
  -1.2403 - 0.9412i
  -1.2403 + 0.9412i
```

Roots can be computed by using zero(), which is a function of the Control Toolbox of MATLAB:

```
>> F_tf = tf(f, 1)
F_tf =
   12 s^5 + 13 s^4 - 15 s^2 + 17 s - 13
Continuous-time transfer function.

>> x_sols = zero(F_tf)
x_sols =
  -1.2403 + 0.9412i
  -1.2403 - 0.9412i
   0.7941 + 0.0000i
   0.3015 + 0.6869i
   0.3015 - 0.6869i
```

Note in this case, a transfer function (ratio of two polynomials) with a denominator of 1 in the “s” domain is created first. Then the roots of s are computed, which would make the polynomial equal to zero.

The values of polynomials at specific input argument values can be computed using MATLAB's built-in function `polyval()`. Here is an example how to use this function:

```
>> f = [12, 13, 0, -15, 17, -13];       % Given polynomial
>> x = linspace(-10, 10, 500);
>> f_val = polyval(f,x);                % Computed polynomial values
```

Simulink Model-Based Solution of Polynomials

To solve polynomials via Simulink modeling, use the MATLAB Fcn block, the Constant block to input the polynomial coefficients, and the Display block to see the computed roots. Figure 7-29 shows the complete model saved as `Polynomial_Solver.slx`.

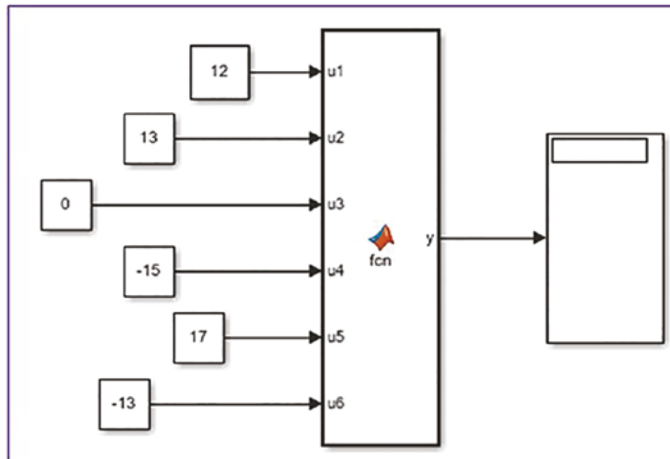


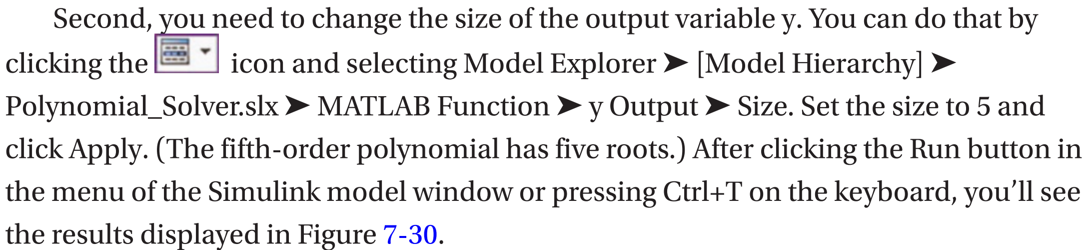
Figure 7-29. *Simulink model to solve the polynomial* $12x^5 + 13x^4 - 15x^2 + 17x - 13 = 0$

The MATLAB Function block has the following command syntax embedded in it:

```
function y  =  fcn(u1,  u2,  u3,  u4,  u5,  u6)
y = roots([u1, u2, u3, u4, u5, u6]);
```

The MATLAB Fcn block calls the MATLAB function `roots()` and computes the roots of the polynomial with respect to its coefficients given by the input variables u1, u2, … u6 since we are solving a fifth-order polynomial. As it is, this model does not run, and there are two more issues related to the size of the variables and solver type. First, the *solver type* has to be a fixed-step size type. That can be adjusted via Simulation ➤ Model

Configuration Parameters ➤ Solver Selection ➤ Fixed Step Solver. By default, the solver is a variable type.

Second, you need to change the size of the output variable y. You can do that by clicking the  icon and selecting Model Explorer ➤ [Model Hierarchy] ➤ Polynomial_Solver.slx ➤ MATLAB Function ➤ y Output ➤ Size. Set the size to 5 and click Apply. (The fifth-order polynomial has five roots.) After clicking the Run button in the menu of the Simulink model window or pressing Ctrl+T on the keyboard, you'll see the results displayed in Figure 7-30.

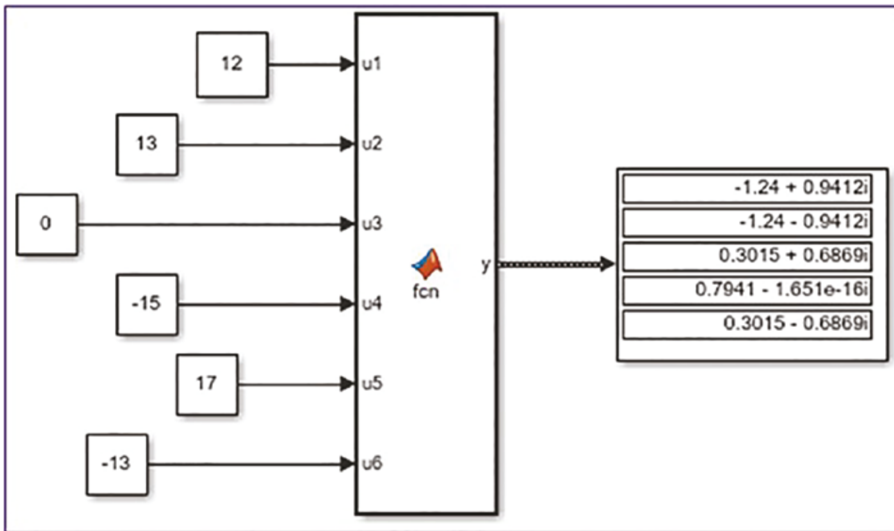


Figure 7-30. *Complete model with computed roots of the polynomial* $12x^5 + 13x^4 - 15x^2 + 17x - 13 = 0$

The computed roots of the given polynomial match the ones computed by the MATLAB commands `roots()` and `zero()` to four decimal places.

Eigen-Values and Eigen-Vectors

Eigen-values and eigen-vectors have broad applications, not only in linear algebra but also in many engineering problems. For instance, they are used with vibrations, modal analysis, control applications, robotics, and so forth.

Definition 1. An *eigen-value* and *eigen-vector* of a square matrix *A* are, respectively, a scalar λ and a nonzero vector v that satisfy the following:

$$Av = \lambda v \qquad \text{(Equation 7-6)}$$

Definition 2. Given a linear transformation *A* (a square matrix), a nonzero vector v is defined to be an *eigen-vector* of the transformation if it satisfies the following *eigen-value* equation for some scalar λ:

$$[A]\{v\} = \lambda\{v\} \qquad \text{(Equation 7-7)}$$

In this case, the scalar λ is called an eigen-value of A corresponding to the eigen-vector $\{v\}$.

$$\underbrace{\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}}_{[A]} * \underbrace{\begin{Bmatrix} x \\ y \\ z \end{Bmatrix}}_{[X]} = \lambda * \underbrace{\begin{Bmatrix} x \\ y \\ z \end{Bmatrix}}_{[X]}$$

$$[A]*[\boldsymbol{X}] - \lambda*[I]*[\boldsymbol{X}] = 0 \qquad \text{(Equation 7-8)}$$

Here, $[I]$ is the identity matrix. Now by rearranging, the next formulation can be written as follows:

$$\left([A]*[X] - [\lambda]*[I]\right)*[X] = 0 \qquad \text{(Equation 7-9)}$$

Let's assume that there is an inverse matrix of the coefficient of $[X]$, *i.e.*, $([A] - [\lambda] * [I])$.

$$\left([A] - [\lambda]*[I]\right)^{-1} = 0 \qquad \text{(Equation 7-10)}$$

There can be other solutions apart from a trivial solution $[X] = 0$. So, this means $([A] - [\lambda] * [I]) = 0$ is obtained via determinant of this matrix equal to 0.

$$det\{[A] - [\lambda]*[I]\} = 0 \qquad \text{(Equation 7-11)}$$

The left side of Equation 7-11 is called a *characteristic polynomial*. So, when this equation is expanded, it will lead to a polynomial equation of λ. Use the following example to compute eigen-values and eigen-vectors:

$$\begin{cases} 2.3x_1 + 3.4x_2 + 5x_3 = 0 \\ 3x_1 + 2.4x_2 - 1.5x_3 = 0 \\ 2x_1 - 0.4x_2 - 7.2x_3 = 0 \end{cases}$$

Now, the given system's equations are written in matrix form.

$$\begin{bmatrix} 2.3 & 3.4 & 5 \\ 3 & 2.4 & -1.5 \\ 2 & -0.4 & -7.2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

Eigen-values of this transformation matrix are defined to be:

$$\det \begin{bmatrix} 2.3-\lambda & 3.4 & 5 \\ 3 & 2.4-\lambda & -1.5 \\ 2 & -0.4 & -7.2-\lambda \end{bmatrix} = 0$$

$$-7.884 + 49.12\,\lambda - 2.5\,\lambda^2 - \lambda^3 = 0$$

Solutions of this characteristic polynomial equation are as follows:

$$\lambda_1 = -8.434; \lambda_2 = 0.162; \lambda_3 = 5.772$$

Further, three eigen-vectors are computed by plugging in each eigen-value one by one into the equation. Hand calculations of eigen-values and eigen-vectors for larger systems are tedious and time-consuming. For very large systems of linear equations, it is infeasible to compute eigen-values and eigen-vectors with hand calculations. All of these computations can be performed with a single built-in function of MATLAB, called `eig(A)`:

```
>> A = [2.3 3.4, 5; 3, 2.4, -1.5; 2, -0.4, -7.2]
A =
    2.3000    3.4000    5.0000
    3.0000    2.4000   -1.5000
    2.0000   -0.4000   -7.2000

>> [v, lambda]=eig(A)
v =
   -0.7649   -0.6510   -0.4725
   -0.6366    0.7276    0.2479
   -0.0983   -0.2164    0.8458
lambda =
    5.7726         0         0
         0    0.1619         0
         0         0   -8.4345
```

```
>> A*v - v*lambda   % Verify:  eigen-vectors and eigen-values;
ans =
   1.0e-14 *
   -0.0888   -0.0638   -0.1776
   -0.1776   -0.0763         0
    0.0555   -0.1783   -0.1776
```

Note that there are several different syntax forms of the eig() function to compute eigen-values and eigen-vectors of square arrays, and there is another command, called eigs(A), to compute eigen-values and eigen-vectors.

```
d = eig(A)
d = eig(A,B)
[V,D] = eig(A)
[V,D] = eig(A,'nobalance')
[V,D] = eig(A,B)
[V,D] = eig(A,B,flag)
```

To evaluate the largest eigen-values and eigen-vectors, use this:

```
d   = eigs(A)
[V,D]  = eigs(A)
[V,D,flag] = eigs(A); eigs(A,B)
eigs(A,k)
eigs(A,B,k)
eigs(A,k,sigma); eigs(A,B,k,sigma); eigs(A,K,sigma,opts);
eigs(A,B,k,sigma,opts)
```

Matrix Decomposition

The matrix decompositions have broad and valuable applications in many areas of linear algebra and engineering problem solving, for instance, solving linear equations, linear least squares, nonlinear optimization, Monte-Carlo simulation, experimental data analysis, modal analysis, circuit design, filter design, and many more. There are a few types of matrix transformations and decompositions, including QR, LU, LQ, Cholesky, Schur, singular value decomposition, and so forth. We very briefly already discussed

the command syntaxes of QR, LU, LQ, `chol()` Cholesky, and `svd()` singular value decompositions while solving the systems of linear equations. This section explains how to compute matrix decompostions by using MATLAB's built-in functions.

QR Decomposition

QR decomposition is also called *orthogonal-triangular decomposition*. It's the process of factoring out a given matrix as a product of two matrices. They are traditionally called the Q and R matrices, and they are the orthogonal matrix Q and the upper triangular matrix R.

$$A = QR \qquad (\text{Equation } 7-12)$$

$$Q^T Q = I \qquad (\text{Equation } 7-13)$$

Here, Q is an orthogonal matrix, Q^T is a transpose of Q, R is an upper triangular matrix, and I is an identity matrix. The QR decomposition is based on the Gram-Schmidt method. More details of the Gram-Schmidt method can be found on Wikipedia [1]. In MATLAB for the QR decomposition computation, there is a function called `qr()`. It has a few different syntax methods that evaluate Q, R, and other relevant matrices.

```
[Q,R] = qr(A)  %Produces  upper  triangular  matrix  R  &  unit  matrix  Q
[Q,R] = qr(A,0)     %Produces the economy-size decomposition
[Q,R,E] = qr(A)     %Produces Q, R and permutation matrix E =>A*E = Q*R
[Q,R,E] =  qr(A,0)  %Produces  economy-size  decomposition:  A(:,E) =  Q*R
X  =  qr(A)         %Produces  matrix  X.  triu(X) is  upper  triangular
factor  R
X = qr(A,0)         % The same as X = qr(A);
R = qr(A)           % Used when A is a sparse matrix and computes a Q-less
% QR decomposition and returns R.
```

Example: Computing QR Decomposition of a 5-by-5 Matrix

Let's take matrix [A] of size 5x5 generated from a normally distributed random number generator, called `randn()`. Compute the QR decompositions of the [A] matrix.

```
>> format short
>> A = randn(5)
A =
    0.3335   -0.4762   -0.3349    0.6601    0.0230
    0.3914    0.8620    0.5528   -0.0679    0.0513
    0.4517   -1.3617    1.0391   -0.1952    0.8261
   -0.1303    0.4550   -1.1176   -0.2176    1.5270
    0.1837   -0.8487    1.2607   -0.3031    0.4669
>> [Q, R]=qr(A)
Q =
   -0.4629    0.0336    0.6635    0.4906    0.3219
   -0.5432   -0.7902   -0.2551   -0.1236    0.0155
   -0.6269    0.4640    0.0710   -0.4160   -0.4622
    0.1808   -0.1702    0.5228   -0.7440    0.3339
   -0.2549    0.3609   -0.4650   -0.1322    0.7557
R =
   -0.7205    0.9045   -1.3201   -0.1084   -0.3993
         0   -1.7127    0.6793   -0.0872    0.2522
         0         0   -1.4600    0.4687    0.6421
         0         0         0    0.6154   -1.5365
         0         0         0         0    0.4891
>> [Q,  R]=qr(A,  0)
Q =
   -0.4629    0.0336    0.6635    0.4906    0.3219
   -0.5432   -0.7902   -0.2551   -0.1236    0.0155
   -0.6269    0.4640    0.0710   -0.4160   -0.4622
    0.1808   -0.1702    0.5228   -0.7440    0.3339
   -0.2549    0.3609   -0.4650   -0.1322    0.7557
R =
   -0.7205    0.9045   -1.3201   -0.1084   -0.3993
         0   -1.7127    0.6793   -0.0872    0.2522
         0         0   -1.4600    0.4687    0.6421
         0         0         0    0.6154   -1.5365
         0         0         0         0    0.4891
```

```
>> [Q,R,E]=qr(A)
Q =
   -0.1608   -0.0026   -0.4424    0.8652    0.1726
    0.2655   -0.0455    0.7987    0.4982   -0.2035
    0.4990   -0.4921   -0.3661    0.0262   -0.6116
   -0.5367   -0.8164    0.1799   -0.0321    0.1099
    0.6054   -0.2988   -0.0065   -0.0386    0.7366
R =
    2.0823   -0.1148   -1.1322   -0.2883    0.4568
         0   -1.7950    0.5142    0.3657   -0.1895
         0         0    1.4850   -0.3119   -0.0250
         0         0         0    0.5510    0.4924
         0         0         0         0   -0.1773
E =
     0     0     0     0     1
     0     0     1     0     0
     1     0     0     0     0
     0     0     0     1     0
     0     1     0     0     0
>> A*E
ans =
   -0.3349    0.0230   -0.4762    0.6601    0.3335
    0.5528    0.0513    0.8620   -0.0679    0.3914
    1.0391    0.8261   -1.3617   -0.1952    0.4517
   -1.1176    1.5270    0.4550   -0.2176   -0.1303
    1.2607    0.4669   -0.8487   -0.3031    0.1837
>> Q*R
ans =
   -0.3349    0.0230   -0.4762    0.6601    0.3335
    0.5528    0.0513    0.8620   -0.0679    0.3914
    1.0391    0.8261   -1.3617   -0.1952    0.4517
   -1.1176    1.5270    0.4550   -0.2176   -0.1303
    1.2607    0.4669   -0.8487   -0.3031    0.1837
```

LU Decomposition

The LU decomposition or factorization is also called a modified form of the Gauss elimination method and was introduced by Alan Turing [2]. It is defined as follows:

$$A = LU \qquad \text{(Equation 7-14)}$$

Here, A is a rectangular matrix, and L and U are the lower and upper triangular matrices, respectively.

For example, a 3-by-3 matrix can be LU factorized with the following expressions:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} * \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

In MATLAB, the LU decomposition is evaluated using the following syntax of the built-in function `lu()`:

```
Y  =  lu(A)       %Produces matrix Y, for sparse  A.  Y  contains  only
                  L  [L,U]  =  lu(A)   %Produces U and L
[L,U,P] = lu(A)   %Produces U & L with a unit diagonal & permutation
                  matrix P
[L,U,P,Q]   =   lu(A)   % Produces U, L, and row permutation matrix P
                        % and column reordering matrix Q, so that
                          P*A*Q = L*U
[L,U,P,Q,R]  =  lu(A)   % Produces U,L, & permutation matrices P and Q,
                        % diagonal scaling matrix R so that  P*(R\
                           A)*Q  =  L*U
                        % for sparse non-empty A.
[...] = lu(A,'vector')  %Produces the permutation information in two %row
                        vectors p and q. A user can specify from 1 to 5
                        outputs.
[...] = lu(A,thresh)
[...] = lu(A,thresh,'vector')
```

Example: Computing LU Composition of a 3-by-3 Pascal Matrix

Let's compute L, U, and other (P, Q, R) matrices from any given rectangular matrix. For this task, you write a small script called `LU_decomposition.m` with MATLAB's built-in function `lu()`. The script takes one user entry (input), which has to be a rectangular matrix. You'll employ in this script another built-in function of MATLAB, called `issparse()`. It identifies whether the user-entered matrix is a sparse matrix or not.

```
% LU_decomposition.m
A=input('Enter rectangular matrix: ');
if   issparse(A)
     Y  =  lu(A)                     %#ok
     [L,U,P,Q]   =   lu(A)           %#ok
     disp('   oops more   ')
     [L,U,P,Q,R]  =  lu(A)           %#ok
     [L, U, P, Q, R] = lu(A,'vector') %#ok
else
     [L,U]  =  lu(A)   %#ok
     [L,U,P]   =   lu(A)   %#ok
     % Check evaluation results:
     ERROR=P*A-L*U   %#ok
     [L,U,P]  =  lu(A,  'vector')     %#ok
end
```

Run the script `LU_decomposition.m` and enter a standard matrix, called `pascal(3)`, as an input matrix.

```
Enter rectangular matrix: pascal(3)
L =
    1.0000         0         0
    1.0000    0.5000    1.0000
    1.0000    1.0000         0
U =
    1.0000    1.0000    1.0000
         0    2.0000    5.0000
         0         0   -0.5000
```

```
L =
    1.0000         0         0
    1.0000    1.0000         0
    1.0000    0.5000    1.0000
U =
    1.0000    1.0000    1.0000
         0    2.0000    5.0000
         0         0   -0.5000
P =
     1     0     0
     0     0     1
     0     1     0
ERROR =
     0     0     0
     0     0     0
     0     0     0
L =
    1.0000         0         0
    1.0000    1.0000         0
    1.0000    0.5000    1.0000
U =
    1.0000    1.0000    1.0000
         0    2.0000    5.0000
         0         0   -0.5000
P =
     1     3     2
```

Rerun the script and use a sparse matrix of size 3-by-3 as input.

```
Enter rectangular matrix: sparse(3)
Y =
   (1,1)        3
   oops more
L =
   (1,1)        1
```

```
U =
   (1,1)        1
P =
   (1,1)        1
Q =
   (1,1)        1
R =
   (1,1)        3
L =
   (1,1)        1
U =
   (1,1)        1
P =
     1
Q =
     1
R =
   (1,1)        3
```

Example: Solving [A]{x}=[b] Using LU Composition

LU composition can be employed to solve the [A]{x} = [b] system of linear equations using the MATLAB's `mldivide()` or backslash (\) operator.

$$[A]\{x\} = [b] \rightarrow [A] = [P]' * [L] * [U]$$
$$[y] = [L]([P] * [b]) \rightarrow \{x\} = [U] \backslash [y]$$

Let's take the following example:

$$\begin{cases} 3x - \frac{2}{3}y + z = 1 \\ 2x + y - \frac{1}{2}z = 2 \\ \frac{3}{4}x - y - z = 3 \end{cases}$$

The solution of this example is as follows:

```
A = [3 -2/3 1; 2 1 -1/2; 3/4 -1 -1];
b = [1;2;3];
[L, U, P]=lu(A);
y=mldivide(L,(P*b));
x = U\y
x =
                          0.82
                        -0.555
                         -1.83
```

Cholesky Decomposition

The Cholesky decomposition is particularly important for Monte Carlo simulations and Kalman filter designs. This type of matrix factorization is applicable only to square matrices and to Cholesky triangles, which are decompositions of positive and definite matrixes that is decomposed into a product of a lower triangular matrix and its transpose. The Cholesky decomposition [3, 4] can be expressed via the following formulation:

$$A = U^T U \quad \text{(Equation 7-15)}$$

Here, A is a square matrix, and U and U^T are an upper triangular matrix and its transpose, respectively. This formulation can be written with lower triangular matrix (L) and its transpose (L^T) as well.

$$A = LL^T \quad \text{(Equation 7-16)}$$

In MATLAB, the Cholesky decompositions are evaluated using the following syntax options of the MATLAB's built-in function, `chol()`:

```
R = chol(A) % Produces an upper triangular matrix R satisfying: R'*R=A
L = chol(A,'lower') % Produces a lower triangular matrix R satisfying:
                    % L*L'=A
[R,p] = chol(A) %Produces  an  upper  triangular  matrix  R  and  p  is  0
[L,p] = chol(A,'lower') %Produces lower triangular matrix R&p  is  0
[R,p,S] = chol(A) % When A is a sparse matrix, produces a permutation
                  % matrices S and R, and p that can be zero or non-zero
```

```
[R,p,s] = chol(A,'vector') % Produces the permutation information %as a
vector 's'
[L,p,s] = chol(A,'lower','vector') % Produces a lower triangular matrix
                                   % L and a permutation vector 's'
```

Note Using `chol` (the Cholesky decomposition operator) is preferable over the `eig` (eigen-value and eigen-vector) operator for determining positive definiteness.

To evaluate the Cholesky decompositions of any given matrix (a user-entered matrix), you write the next script, called `Chol_decoposition.m`, by considering the requirements and properties of the Cholesky decompositions to compute decompositions of any matrix with respect to the formulations in Equations 7-15 and 7-16. It takes one input, which is a user entry matrix. Note that in this script, we used `disp()`, `size()`, `det()`, `run()`, and a pop-up dialog box command, `warndlg()`.

```
% Chol_decomposition.m
clearvars; clc
disp('Note your matrix must be square & positive definite!!!')
disp('NB: Positive means all determinants must be positive.')
disp('You can enter as matrix elements ')
disp('or define your matrix 1st, ')
disp('and then just enter your matrix name')
disp('      ')
A=input('Enter a given Matrix: ');
[rows, cols]=size(A);
for k=1:rows
    % Determinants are computed
    Det_A(k)=det(A(1:k, 1:k));
end
if rows==cols
  if Det_A>0
     if issparse(A)
            [R,p,S] = chol(A) %#ok
            [R,p,s] = chol(A,'vector');
            [L,p,s] = chol(A,'lower','vector');
```

```
else
            R = chol(A) %#ok % Upper triangular matrix R: R'*R=A
            L = chol(A,'lower') %#ok % Lower triangular matrix R.
            [R,p] = chol(A);
% Verify:
Error_up  = A-R'*R;
Error_low = A-L*L';
disp('Error is with upper triangular matrix: ')
disp(Error_up)
disp('Error is with lower triangular matrix:')
disp(Error_low)
       end
else
warndlg('Sorry your matrix is not positive and definite!')
warndlg('Try again!!!')
run('Chol_decomposition')
    end
end
```

You can test the script with different input entries (matrices). Let's use a 4-by-4 standard matrix generated with `pascal()`.

```
Note your matrix must be square & positive definite!!!
NB: Positive means all determinants must be positive.
You can enter as matrix elements
or define your matrix 1st,
and then just enter your matrix name

Enter a given Matrix: pascal(4)
R =
     1     1     1     1
     0     1     2     3
     0     0     1     3
     0     0     0     1
```

```
L =
     1     0     0     0
     1     1     0     0
     1     2     1     0
     1     3     3     1
Error is with upper triangular matrix:
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
Error is with lower triangular matrix:
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
Now, consider a magic matrix of size 3-by-3.

>> run('Chol_decomposition')
Note your matrix must be square & positive definite!!!
NB: Positive means all determinants must be positive.
You can enter as matrix elements
or define your matrix 1st,
and then just enter your matrix name

Enter a given Matrix: magic(3)
```

After running the script with an input entry of a magic square matrix of size 3-by-3, the warning dialog boxes shown in Figure 7-31 appear.

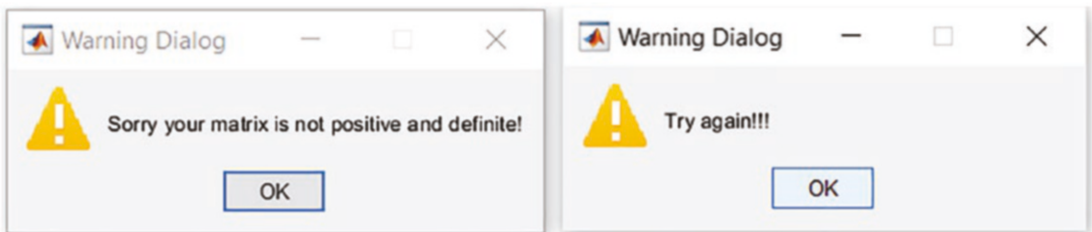


Figure 7-31. *Warnings showing that the input matrix is not positive and definite and so cannot compute the Cholesky decompositions*

Besides these two warning message boxes shown in Figure 7.31, the code keeps asking to enter a matrix. The `Chol_decoposition.m` script identifies the Cholesky decomposition properties and computes the Cholesky decomposition of a user-entered matrix. It detects a matrix type and works for given square and positive definite matrices with the MATLAB built-in function `chol()`.

Schur Decomposition

The Schur decomposition has many applications in numerical analyses, including image-processing areas in combination with other matrix decompositions or factorization tools. The Schur decomposition of a complex square matrix [A] is defined as a matrix decomposition [5]:

$$Q^H\ AQ = T = D + N \qquad \text{(Equation 7-17)}$$

Here, Q is a unitary matrix, Q^H is a conjugate transpose of Q, and T is an upper triangular matrix that's equal to sum of a matrix $D = diag\,(\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_n)$ a diagonal matrix consisting of eigen-values λ_i of A,and strictly upper triangular matrix N. The Schur decomposition can be computed via the MATLAB's built-in function, `schur()`.

```
T  =  schur(A)   % Produces the Schur matrix of A
T  =  schur(A, flag) % Produces the Schur matrix for two cases.
%{
```

for real matrix A, returns a Schur matrix T in one of two forms depending on the value of flag:

```
'complex'   T is triangular and is complex if A has complex eigenvalues.
'real'   T has the real eigen-values on the diagonal and the complex eigen-
values in 2-by-2 blocks on the diagonal. 'real' is the default.

%}
[U,T] = schur(A,...)
```

Let's look at several examples of standard matrices and compute their Schur decompositions:

```
>> A=magic(5); B=pascal(3); C=round(randn(5,5)*10);
>> SA=schur(A)
SA =
   65.0000    0.0000   -0.0000    0.0000   -0.0000
         0  -21.2768   -2.5888    2.1871   -3.4893
         0         0  -13.1263   -3.3845   -2.8239
         0         0         0   21.2768    2.6287
         0         0         0         0   13.1263
>> SB=schur(B)
SB =
    0.1270         0         0
         0    1.0000         0
         0         0    7.8730
>> SC=schur(C)
SC =
   20.7072    7.3851   -0.2741    9.7514    1.9523
         0   -6.1453   17.4134   -5.0801  -14.3751
         0  -10.3437   -6.1453   14.7269    9.9502
         0         0         0    4.8687    2.1653
         0         0         0         0   -9.2853

>> [T,  U]=schur(A,  'complex')
T =
   -0.4472    0.0976   -0.6331    0.6145   -0.1095
   -0.4472    0.3525    0.7305    0.3760    0.0273
   -0.4472    0.5501   -0.2361   -0.6085    0.2673
   -0.4472   -0.3223    0.0793   -0.3285   -0.7628
   -0.4472   -0.6780    0.0594   -0.0535    0.5778
U =
   65.0000    0.0000   -0.0000    0.0000   -0.0000
         0  -21.2768   -2.5888    2.1871   -3.4893
         0         0  -13.1263   -3.3845   -2.8239
         0         0         0   21.2768    2.6287
         0         0         0         0   13.1263
```

```
>> [TA,  UA]=schur(B,  'real')
TA =
   -0.5438   -0.8165    0.1938
    0.7812   -0.4082    0.4722
   -0.3065    0.4082    0.8599
UA =
    0.1270         0         0
         0    1.0000         0
         0         0    7.8730
>> [T, U]=rsf2csf(U,T) %  Convert real Schur form to complex Schur form
T =
  -61.5539   20.8834   -0.0000   -0.0000    0.0000
    6.2354   18.3788    9.2270   -1.2464    3.6069
    2.0845    6.1442   -9.8039    6.8268    2.6290
    0.9854    2.9044   -4.6344  -20.6565   -1.4269
    0.0340    0.1003   -0.1601   -0.7135  -13.1055
U =
   -0.5636    0.1041    0.6400    0.4570   -0.0505
         0    0.8710   -0.2777    0.0336    0.2993
   -0.4472         0   -0.4787    0.7335   -0.3928
   -0.4472   -0.3223         0   -0.5309   -0.6067
   -0.4472   -0.6780    0.0594         0    0.6209
```

Singular Value Decomposition

The singular value decomposition (SVD) has many applications in signal processing, statistics, and image processing areas. It is formulated as a product of three matrices, which are an orthogonal matrix (U_{ij}), a diagonal matrix (D_{ij}), and the transpose of an orthogonal matrix (V_{jj}), if a given matrix A_{ij} is an i by j sized real matrix with $i > j$.

$$A = U_{ii} D_{ij} V_{jj}^T \quad \text{(Equation 7-18)}$$

Here, $U_{ii}^T U_{ii} = I, V_{jj}^T V_{jj} = I$. Diagonal entries of D_{ij} are known as singular values of A_{ij}.

Moreover, there are a few other important properties of the SVD.

- Left-singular vectors of *Aij* are eigen-vectors of $A_{ij}A_{ij}*$.
- Right-singular vectors of *Aij* are eigen-vectors of $A_{ij}*A_{ij}$.
- Nonzero singular values (on the diagonal entries of D_{ij}) of A_{ij} are square roots of the nonzero eigen-values of both $A_{ij}*A_{ij}$ and $A_{ij}*$.

There are a few ways to evaluate the SVD, singular values, and vectors of any given matrix. You use `svd()` and `svds()`, which are MATLAB built-in functions.

```
s = svd(A)       %Produces a vector of singular values
[U,D,V] = svd(A) %Produces a diagonal matrix D of the same dimension
%as A, with nonnegative diagonal elements in decreasing order, and
% unitary matrices U and V so that X = U*D*V'.
[U,D,V] = svd(A,0) % Produces the "economy size" decomposition. If A
%  is   m-by-n with m > n, then SVD computes only the first n columns of
%U and  D  is n-by-n. s = svds(A)
s = svds(A,k)
s = svds(A,k,sigma) s = svds(A,k,'L')
s = svds(A,k,sigma,options) [U,D,V] = svds(A,...)
[U,D,V,flag] = svds(A,...)
```

Now, take two matrices (of size 2-by-3 and 3-by-3) and evaluate their SVDs.

```
>> A=ceil(randn(2,3)*10); B=pascal(3);
>> A=ceil(randn(2,3)*10); B=pascal(3);
>> A
A =
    -2    -4     3
   -15    -1    -2
>> B
B =
     1     1     1
     1     2     3
     1     3     6
```

```
>> svd(A)
ans =
   15.2914
    5.0172
>> [U,V,D]=svd(A)
U =
   -0.1354   -0.9908
   -0.9908    0.1354
V =
   15.2914         0         0
         0    5.0172         0
D =
    0.9896   -0.0100   -0.1434
    0.1002    0.7629    0.6387
    0.1030   -0.6464    0.7560
>> [U,V,D]=svd(A, 0)
U =
   -0.1354   -0.9908
   -0.9908    0.1354
V =
   15.2914         0         0
         0    5.0172         0
D =
    0.9896   -0.0100   -0.1434
    0.1002    0.7629    0.6387
    0.1030   -0.6464    0.7560
>> SA = svds(A)
SA =
   15.2914
    5.0172
>> SB = svds(B)
SB =
    7.8730
    1.0000
    0.1270
```

```
>> SA = svds(A, 2)
SA =
   15.2914
    5.0172
>> SB = svds(B, 2)
SB =
    7.8730
    1.0000
>> SB = svds(B, 3)
SB =
    7.8730
    1.0000
    0.1270
```

Logic Operators, Indexes, and Conversions

MATLAB uses logic 1 and logic 0 for system variables to denote logic values for true and false, respectively. Variables of logical values are distinguished by a logical data type.

Table 7-2 is a list of logic operators and their operational functions used in MATLAB.

Table 7-2. *Logical Expressions and Operators in MATLAB*

Operator	Operation
`true`, `false`	Setting logical value
& `(and)`, | `(or)`, ~ `(not)`, `xor`, `any`, `all`	Logical operations
&&, ||	Short-circuits operations
`bitand`, `bitcmp`, `bitor`, `bitmax`, `bitxor`, `bitset`, `bitget`, `bitshift`	Bitwise operations
`==(eq)`, `~=(ne)`, `<(lt)`, `>(gt)`, `<=(le)`, `>=(ge)`	Relational operations
`strcmp`, `strncmp`, `strcmpi`, `strncmpi`	String comparisons

Note To get a complete list of relational operators, their functions, and how to use them, type `>> help relop` in the Command window.

Logical Indexing

Logic operators are one of the most central and essential keys to any programming language. Logic operators introduce another method for accessing data in MATLAB variables. For instance, given a magic matrix [A] of size 5-by-5, say you need to separate out the elements of [A] that are equal to or less than 13.

```
>> A=magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
>> Index = A>15 | A<5   % Show which element is greater than 15 or
less than 5
Index =
  5×5 logical array
   1   1   1   0   0
   1   0   0   0   1
   1   0   0   1   1
   0   0   1   1   1
   0   1   1   1   0
>> A(Index)
ans =
    17
    23
     4
    24
    18
     1
```

```
    19
    25
    20
    21
     2
    16
    22
     3
>> A(A>15 | A<5);  % Or in a direct way
```

Let's explore the logical indexing properties further via examples to select matrix elements.

```
>> E = eye(5)  % Identity matrix
E =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

Array indices must be positive integers or logical values.

```
>>EL=loogical(E)
EL =
  5×5 logical array
   1   0   0   0   0
   0   1   0   0   0
   0   0   1   0   0
   0   0   0   1   0
   0   0   0   0   1
>> A(EL)    % Compare Ih A(E)
ans =
    17
     5
    13
```

```
    21
     9
A(EL) -  shows all diagonal elements of A matrix.
Note
```

Note The previous example demonstrates the identity matrix [e] (whose elements are 1s and 0s), which is not equivalent to the logic matrix [e_L] (whose elements are also 1s and 0s).

Moreover, there are a number of functions/commands (e.g., the `is*()` command) that can be used to find out whether the input is of a specified type of variable, contains any elements of a particular type, or whether such a variable or file exists, and so forth. All of these functions can be used for logical indexing. Let's look at a few simple examples:

```
>>  x=13;  isnumeric(x)     % whether x is a numeric data or not?
ans = 1
>>  x=13; islogical(x)      % whether x is a logical data Or not?
ans = 0
>> x=13; islogical(x>110)   % whether the operation
(x>110)  is  logic  or  not
ans =1
>>  x  =  13;  isempty(x)   % whether x is an empty or not
ans =1
>> x = [ ]; isempty(x)      % whether x is an empty or not
ans = logical 1
>>  x  =  [1  , 2;  3, 4]; iscell(x)   % whether x is a cell array or not
ans = logical 0
>> x = [1, 0; 0, 4];
>>   X_x   =  x/0;  isnan(X_x)   % whether any elements of X_x
are  not-a-number
ans = 2×2 logical array
0   1
1   0
```

```
>>'  e'ist''X_'',  'var')   % whether the variable called
X_x  exists  or  not
ans =1
```

In the previous example, zero divided by zero (0/0) is defined to be NaN (i.e., not-a-number) in MATLAB.

Note The logical indexing operations have particular importance in matrix/array operations, programming, data analysis, and processing since they can be used to sort out, locate, or change particular elements of matrices/arrays/data sets.

Example: Logical Indexing to Locate and Substitute Elements of [A] Matrix

Given: 3-by-3 matrix [A] with some elements equal to infinity $A = \begin{bmatrix} 17 & \infty & -6 \\ 5 & -3 & 11 \\ \infty & 13 & \infty \end{bmatrix}$

How do you substitute the elements equal to inf with 1000 and all negative-valued elements with 0? This task can be solved easily using logical indexing operations.

```
>> A = [17, Inf,   -6  ; 5 -3, 11; Inf, 13, Inf] % [A] is entered
A =
    17   Inf    -6
     5    -3    11
   Inf    13   Inf
>>   Index_inf  =  (A==1/0)   % Find out which elements of A
are  equal  to  inf
Index_inf =
  3×3 logical array
   0   1   0
   0   0   0
   1   0   1
```

```
>> A(Index_inf) =1000       % Set inf elements equal to 1000
A =
          17        1000          -6
           5          -3          11
        1000          13        1000
>> Index_neg = A<0          % Find out which elements  of  A  are  negative
Index_neg =
  3×3 logical array
   0   0   1
   0   1   0
   0   0   0
>> A(Index_neg)=0           % Set all negative elements equal to "0"
A =
          17        1000           0
           5           0          11
        1000          13        1000
```

Note The division of any value by 0 gives the value of `Inf` in MATLAB.

Let's look at another example. Given a matrix [A] of size 4-by-5 with `NaN` (not a number) and `inf` (infinity) elements, how do you substitute `NaN` elements with 0 and `inf` with 100? This task can be solved easily with logical indexing similar to the previously demonstrated example.

```
>> A  =  [2, -3, -2, -3, 1; Inf, -2,   3, -1, NaN; -3, 0, Inf, 3, 2; 3,
NaN, 0, 2, Inf]
A =
     2    -3    -2    -3     1
   Inf    -2     3    -1   NaN
    -3     0   Inf     3     2
     3   NaN     0     2   Inf
>> Index_nan = isnan(A)   % Find out which elements are NaN
```

```
Index_nan =
  4×5 logical array
   0   0   0   0   0
   0   0   0   0   1
   0   0   0   0   0
   0   1   0   0   0
>> A(Index_nan)=0  % Set all NaN elements equal to "0"
A =
     2    -3    -2    -3     1
   Inf    -2     3    -1     0
    -3     0   Inf     3     2
     3     0     0     2   Inf
```

Note that this section contains rather simple and small examples to demonstrate how easily you can substitute specific (valued) elements of a matrix using logical indexing operations. This technique (logical indexing or relational operators) can be applied to matrices, arrays, and data sets of any size. Therefore, the logical indexing is particularly useful in analysis and processing of large data sets. It is fast and efficient and does not require any additional effort to program with loop (`for ... end`, `while ... end`) and conditional (`if ditio. endlyit`) operators.

Conversions

There are many examples in signal processing where you need to convert something. Analog to digital converters and vice versa, data processing and analysis, and programming when analog signal data format or type needs to be converted into digital or vice versa. For instance, to resolve memory issues in image processing, you might need to convert decimal (double) formatted data into binary numbers. That can be easily accomplished in MATLAB using `DEC2BIN()`. Conversely, `BIN2DEC()` is used to convert binary strings into decimal (double) type of data. `DEC2BIN(D)` returns the binary representation of *D* as a string. *D* must be a non-negative integer smaller than 2^{52}.

`DEC2BIN(D,N)` produces a binary representation with at least *N* bits.

Another conversion example is character conversion. You need to convert numbers into character strings and vice versa. MATLAB uses the `CHAR()` command to convert numbers into ASCII/ANSI formatted characters, `DOUBLE()` to convert characters and symbolic representations of numbers into double precision format, `STR2NUM()` to convert

strings into binary numbers, and NUM2STR() to convert any number into a string. Let's consider several examples of employing these conversion commands:

```
>> dec2bin(11) % Converts  decimal  (integer) into  a  binary  string
ans =
    '1011'
>> dec2bin(23)
ans =
'10111'
>> dec2bin(22) ans =
'10110'
>> x=13.125/5.5;
>> dec2bin(x)
ans =
    '10'
>> dec2bin(11.11)
ans =
    '1011'
>> dec2bin(11)
ans =
    '1011'
>> bin2dec('1101')  %  Converts  a  binary  number  into  decimal  one
ans =
13
>> bin2dec('10110')
ans =
22
>> dec2bin(64)
ans =
10000000
>> char(bin2dec('10000000'))
ans =
@
>>   G='MatLab' G =
MatLab
```

```
>> GO=G+0 GO =
77   97   116   76   97   98
>> d2bGO=dec2bin(GO) d2bGO =
1001101
1100001
1110100
1001100
1100001
1100010
>> b2dGO=bin2dec(d2bGO) b2dGO =
77
97
116
76
97
98
>> char(b2dGO)' ans =
MatLab
>> num2str(123) ans =
'123'
>> num2str('matlab') ans =
'matlab'
>> ans+0
109   97   116   108   97   98
```

Example: Creating Character Strings with char()

Create the following letters in a progressive format by writing a script that has one input argument that has to be an integer. All the other letters need to be generated programmatically.

```
a
b c
d e f
g h i j
k l m n o
```

These characters can be generated in several ways. First, you need to determine the ASCII/ANSI numeric representation of a. Then you can generate all the other letters.

```
>> format short
>> double('a')
   97
>> char(97)
   'a'
>> double('b')
   98
```

The letter as numeric representation in ASCII/ANSI is 97, b is represented by 98, and so forth. Based on these, you can generate linear space of integers starting at 97 and convert them to character strings one row at a time. In other words, you display one character on the first row, two characters on the second, three in the third row, etc. Here is the complete script (`print_character.m`), which prints the letters in progressive order:

```
% print_character.m
% Part 1.
Start = 97;
for ii = 1:5
    for jj = 1:ii
        fprintf(char(Start));
        Start = Start+1;
    end
    fprintf('\n')
end
```

Here is the result of the script:

```
a
bc
def
ghij
klmno
```

Let's consider the following example, which prints a series of uppercase characters:

```
ABCDEF
GHIJK
```

```
LMNO
PQR
ST
U
```

This example is similar to the previous example with a few small differences—it requires uppercase characters, starts with six letters, and reduces in the following rows.

Again, you can determine the numerical representation of A in ANSI/ASCII with the following commands:

```
>> double('A')
ans =
   65
>> 'A' + 0    % An alternative way:
ans =
   65
```

So now you know that the numerical representation of `A` is 65. You can then edit the script (`print_characters.m`) by introducing two small changes and then write this script:

```
%% Part 2. Upper cases
Start = 65;
for ii = 1:2:9
    for jj = 1:ii
        fprintf(char(Start));
        Start = Start+1;
    end
    fprintf('\n')
end
```

When you execute this script, you obtain the following output in the Command window:

```
ABCDEF
GHIJK
```

```
LMNO
PQR
ST
U
```

Via a few examples, this section discussed logic operators, conversions, and indexing issues briefly. Applications of the issues of conversions are demonstrated via more extended examples in other chapters.

Summary

This chapter introduced linear algebra, matrix operations, vector spaces, polynomials, methods of solving linear systems of equations, and matrix decompositions and conversions. Via examples, you learned how to use MATLAB's built-in functions and commands, how to develop Simulink blocks in association with the MATLAB Command window, and how to use functions and the MATLAB Fcn block. The following MATLAB functions were discussed and explained in examples:

- Matrix operations `+`,`-`, `*`, and `/`
- Elementwise operations `.*`, `.^`, and `./`
- Backslash operator (`\`) and `mldivide()`
- Solving linear equations with `linsolve()`
- Matrix inverse operators `inv()` and `pinv()`
- Eigen-values and eigen-vectors `eig()`
- Polynomial solvers `roots()`, `solve()`, and `zero()`
- Symbolic math equation solver `solve()`
- Standard matrices and gallery matrices, `magic()`, `gallery()`, and `sparse()`
- Vector spaces `linspace()` and `logspace()`
- Matrix operations and factorization methods, such as QR, LU, Cholesky, SVD, Schur: `qr()`, `lu()`, `chol()`, `svd()`, `schur()`, and `decomposition()`

- Logical operators (`<=, ~=, >=, |, &...`, `is*()`) and indexing options
- Conversion tools and operators (`bin2dec`, `dec2bin`, `double`, and `char`)

References

[1]. Wikipedia, http://en.wikipedia.org/wiki/Gram-Schmidt_process, viewed on September 19, 2013.

[2]. Bunch, James R.; Hopcroft, John (1974), "Triangular Factorization and Inversion by Fast Matrix Multiplication," *Mathematics of Computation* 28: 231–236, ISSN 0025-5718.

[3]. Gentle, J. E. "Cholesky Factorization." §3.2.2 in *Numerical Linear Algebra for Applications in Statistics.* Berlin: Springer-Verlag, pp. 93-95, 1998.

[4]. Nash, J. C. "The Choleski Decomposition." Ch. 7 in *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation, 2nd ed.* Bristol, England: Adam Hilger, pp. 84-93, 1990.

[5]. Mathworld , http://mathworld.wolfram.com/SchurDecomposition.html, viewed on September 20, 2013.

Exercises for Self-Testing

Exercise 1

Solve the following equations for variables *x*, *y*, and *z*:

$$\begin{cases} 3x+5y+4z=-2 \\ -2x+3y-2z=2 \\ x+6\left(y-\frac{z}{2}\right)=0 \end{cases}$$

1. Use the backslash (\) operator or `mldivide()` to solve the given system of equations.
2. Use the inverse matrix method `inv()` to solve the given system of equations.
3. Use the `linsolve()` function to solve the given system of equations.
4. Use the `solve()` function to solve the given system of equations.
5. Use `chol()` to solve the given system of equations.
6. Use Simulink blocks to solve the given system of equations.
7. Compute errors by computing norms for each of the methods.

Exercise 2

Solve the following equations, using the matrix inverse:

$$\begin{cases} 2q_1 + 9q_2 + 3q_3 = 15 \\ 13q_1 + 2q_2 - 5q_3 = 11 \\ q_1 - 2q_2 + 2q_3 = 9 \end{cases}$$

1. Use the inverse matrix method `inv()`.
2. Use the least squares method `lsqr()`.
3. Use the Gauss Elimination method with the `lu()`.
4. Use `rref()`.
5. Use the `solve()`.
6. Use Simulink blocks.
7. Compare the accuracy (to eight decimal places) of each solution.

Exercise 3

Solve the following equations:

$$\begin{cases} 2.5x_1 - x_2 + 3.3x_3 = 5 \\ -2.2x_1 + 2x_2 - 5x_3 = -2 \\ x_1 - 2x_2 + 2.5x_3 = 3 \end{cases}$$

1. Use the inverse matrix method `qr()` to solve the given system of equations.
2. Use the reduced row echelon method step-by-step by multiplying rows by scalars and adding or subtracting from each other (don't use `rref()`).
3. Use the reduced row echelon method `rref()` to solve the given system of equations.
4. Use the `decomposition()` function to solve the given system of equations.
5. Use the `solve()` function to solve the given system of equations.
6. Use Simulink blocks to solve the given system of equations.
7. Compare the accuracy (to 10 decimal places) of these four methods.

Exercise 4

Solve the following equations:

$$\begin{cases} 2.5x_1 - x_2 + 3.3x_3 - 0.3x_5 = 5 \\ -1.2x_1 + 2.5x_2 - 2x_3 - 2.2x_4 + 5.2x_5 = -3 \\ x_1 + 3x_2 - 2.5x_3 - x_5 = 1 \\ 2x_1 + x_2 - 5x_3 - 3x_4 - 4.3x_5 = -6 \\ 3x_1 - 2.4x_2 + 1.75x_5 = 13 \end{cases}$$

1. Use the inverse matrix method `inv()` to solve the given system of equations.
2. Use the singular decomposition `svd()` method to solve the given system of equations.
3. Use the `linsolve()` function to solve the given system of equations.
4. Use the `solve()` function to solve the given system of equations.
5. Use Simulink blocks to solve the given system of equations.
6. Compare the accuracy (to 13 decimal places) of these methods.

Exercise 5

Given:

$$3x + 6y - cz = 0$$

$$2x + 4y - 6z = 0$$

$$x + 2y - 3z = 0$$

- Find for which values of c the set of equations has a trivial solution.
- Find for which values of c the set of equations has an infinite number of solutions.
- Find relations between x, y, and z.

Exercise 6

Find the inverse of the given matrix:

$$A = \begin{bmatrix} 3 & 6 & -12 \\ 2 & 4 & -6 \\ 1 & 2 & -3 \end{bmatrix}$$

- Explain why the given matrix does not have an inverse.
- Compute the determinant of the matrix.
- Find eigen-values and eigen-vectors of the given system by using `eig()`.
- Find eigen-values by using `roots()`.

Exercise 7

Find the inverse of the given matrix:

$$A = \begin{bmatrix} 3 & 6 & -2 \\ 1 & 2 & -4 \\ 0 & 1 & -3 \end{bmatrix}$$

- Compute determinant of the matrix.
- Find eigen-values and eigen-vectors of the given system using `eig()`.
- Find eigen-values using `roots()`.

Exercise 8

Find a solution to the following set of equations representing an underdetermined system, using the left division (\ backslash) method and the pseudo-inverse method (`pinv`). Compare your obtained results and discuss the differences.

$$2.5x_1 - x_2 + 3.3x_3 + 1.3x_4 - 0.3x_5 = 11$$

$$-1.2x_1 + 2x_2 - 5x_3 - 2.1x_4 + 5.2x_5 = -2$$

$$x_1 - 2x_2 + 2.5x_3 = 3$$

Exercise 9

Solve the following set of equations using the backslash (`\`) operator, as well as the `linsolve()`, `inv()`, `lsqr()`, and `solve()` functions:

$$2x - 3y = 5$$

$$6x + 10y = 70$$

$$10x - 4y = 53$$

Exercise 10

Show why there is no solution to the following set of equations:

$$-2x - 3y = 2$$
$$-3x - 5y = 7$$
$$5x - 2y = -4$$

Exercise 11

Solve the following equations:

$$\begin{cases} 2.5x_1 - x_2 + 3.3x_3 - 0.3x_5 = 5 \\ -1.2x_1 + 2.5x_2 - 2x_3 - 2.2x_4 + 5.2x_5 = -3 \\ x_1 + 3x_2 - 2.5x_3 - x_5 = 1 \\ 2x_1 + x_2 - 5x_3 - 3x_4 - 4.3x_5 = -6 \\ 3x_1 - 2.4x_2 + 1.75x_5 = v \end{cases}$$

$$v = [-10, -9, -8, \ldots 9, 10]$$

1. Use the inverse matrix method `mldivide()` to solve the given system of equations.
2. Use the singular decomposition `svd()` method to solve the given system of equations.
3. Use the `linsolve()` function to solve the given system of equations.
4. Use the `solve()` function to solve the given system of equations.

Exercise 12

Compute the eigen-values and vectors of the following set of equations:

$$\begin{cases} 3x-2y+5z+2u-w=0 \\ x+y+z-w=0 \\ -2x+3y+4z-5u-7w=0 \\ -5y-\frac{3}{4}z+\frac{7}{13}u+9w=0 \\ 10x-11y+8u-8w=0 \end{cases}$$

Exercise 13

Create the matrix [*C*] from the given two [*A*] and [*B*] matrices by using logic operators. Explain why some of the elements of new array are zeros.

$$C=\begin{bmatrix} 0 & 0 & -3 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & -2 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, A=\begin{bmatrix} 3 & 6 & -3 \\ 1 & 2 & -2 \\ 0 & 1 & 1 \\ 1 & 3 & -2 \\ 2 & 1 & -1 \\ 1 & 0 & 1 \end{bmatrix}, B=\begin{bmatrix} 1 & 2 & -3 \\ 1 & 1 & -3 \\ -1 & -2 & 1 \\ 1 & 1 & -1 \\ 2 & 2 & -2 \\ -2 & 0 & 1 \end{bmatrix}$$

Hints Use logic operators (<, =) and element-wise matrix multiplication.

Exercise 14

The useful life of a machine bearing depends on its operating temperature, as the following data shows. Obtain a functional description (linear, square, and cubic polynomials) of this data. Plot the found fit functions and the data on the same plot. Estimate a bearing's life if it operates at 52.5^{0}C.

Temperature (^{0}C)	40	45	50	55	60	65	70
Bearing life (hours x 10^3)	28	21	15	11	8	6	4

Exercise 15

The following represents pressure samples, in MPa, taken in a fuel line once every second for 10 sec:

Time (Sec)	Pressure (MPa)	Time (Sec)	Pressure (MPa)
1	2.61	6	3.06
2	2.70	7	3.11
3	2.82	8	3.13
4	2.90	9	3.10
5	2.98	10	3.05

a. Fit a *first – degree* polynomial, a *second – degree* polynomial, and a *third – degree* polynomial to this data. Plot the curve fits along with the data points.

b. Use the results from part *a* to predict the pressure at $t = 11$ *sec*.

Exercise 16

The distance a spring stretches from its "free length" is a function of how much tension force is applied to it. The following table gives the spring length y that the given applied force F produced in a particular spring. The spring's free length is 4.7 *m*. Find functional relation between F and x, the extension from the free length ($x = y - 4.7$).

Force $F(kN)$	Spring Length y (m)
0	4.7
0.47	7.2
1.15	10.6
1.64	12.9

Also, plot experimental data (*F* versus *x*) and functional relation based fit (F_linear *vs*. *x*) in the same plot. Use the appropriate plot maker type, color, size, etc., options.

Exercise 17

Perform the following:

- Obtain an eye matrix of the size 5-by-5 from the magic matrix of the size of 5-by-5.
- Create a square eye matrix of the size 10-by-10 from the random square matrix of the size 10-by-10.
- Obtain a replicated square matrix of size 3-by-9 from the gallery matrix `pascal()` of size 3-by-3.

Exercise 18

Solve the following equations and discuss the solutions for two cases: $a = 13$ and $a = 29$.

$$\begin{cases} q_1 + q_2 = 1 \\ 13q_1 + 23q_2 = a \\ q_1 - 2q_2 = 9 \end{cases}$$

Write a script with logic and loop operators (`if`, `break`, `for`, and `end`) to find such value of *a* that gives real solutions to these equations. Consider that *a* has an integer value that lies within 1 to 50.

Hints Use the `rank()` function and backslash (`\`) operators.

Exercise 19

Solve the following polynomials with `roots()`, `solve()`, `zero()`, and the Simulink model.

$$2u^9 - 3u^8 + 5u^4 - 13u^2 - 131 = 0$$

$$y^7 + 5y^5 - \frac{13y^4}{4} - 11y^3 + 9y + 3 = 0$$

$$5x^5 + \frac{4x^4}{11} - \frac{3x^3}{13} + x^2 - 269x = 13$$

Exercise 20

Create a logarithmic spaced array (a row vector) *B* of numbers starting with 10 and ending with 100, and create *BB* column vector from a row vector *B*.

Exercise 21

Play a sound that is defined in the next expression:

$$S(t) = cos(2\pi tf_1) + cos(2\pi tf_2) + cot(2\pi tf_3) + tan(2\pi tf_4) + tan(2\pi tf_5)$$

Here, f_s = 10000 Hz (sampling frequency); t = 13 sec. (time length); f_1 = 100 Hz (1st signal); f_2 = 200 Hz (2nd signal); f_3 = 300 Hz (3rd signal); f_4 = 600 Hz (4th signal); f_5 = 700 Hz (5th signal).

Exercise 22

Answer the following questions using MATLAB:

- What are the binary representations of decimal numbers 123, 123.123, 321, 321.123, 223, 322, 333, and 333.3?
- Why are the binary representations of 123 vs. 123.123, 321 vs. 321.123, and 333 vs. 333.3 the same?

Exercise 23

Answer the following questions using MATLAB:

- What are the decimal representations of the binary numbers 1001, 01010, 111100, 0101011?
- What are character representations of the binary numbers 1001, 01010, 111100, 0101011?

Exercise 24

Write a script that takes one input number (an integer) and prints out the following characters in the order in the Command window:

```
A
BCD
EFGHI
JKLMNOP
QRSTUVWXY
```

Exercise 25

Use numeric values of matrices [A] and [B] from Exercise 11 to evaluate the QR, LU, LQ, Cholesky, Schur, and singular value decompositions. Explain why some of the decompositions (matrix factorizations) of [A] and [B] cannot be computed.

Exercise 26

Create the Hilbert matrix of size 5-by-5 using gallery matrix functions and compute Cholesky decomposition using the `Chol_decoposition.m` script. Edit the script (`Chol_decoposition.m`) in order to make it compute only the lower triangular matrix of Cholesky decomposition.

Exercise 27

Create the Riemann matrix of size 3-by-3 using gallery matrix functions and compute its QR, LU, LQ, Cholesky, Schur, and singular value decompositions.

Exercise 28

Perform the following:

- Create the 4-by-4 random matrix with normalized columns and specified singular values using gallery matrix functions. Hint: Use `randcolu`.
- Compute the QR, LU, LQ, and decompositions of the matrix you just created.

Exercise 29

Perform the following:

- Create one 5-by-5 random matrix with random integer elements varying in the range of 1 to 13 and name it `A_mat`.
- Create one 5-by-5 Krylov matrix using a matrix gallery of Krylov and name it `K_mat`.
- Create logic valued 5-by-5 matrix called `Logic_A` by using logic operation ($A_mat \geq K_mat$) and elementwise matrix multiplication from `A_mat` and `K_mat`

Exercise 30

Create the following 10-by-10 matrix:

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

Hint Use `magic()` and `repmat()`.