MATLAB/LABVIEW
Lab Manual

II YEAR I SEM
EEE

By

Ms. Y. Satyavani
Assistant Professor, EEE

Ms. V. Usha
Assistant Professor, EEE

Department of Electrical and Electronics Engineering
Gokaraju Rangaraju Institute of Engineering & Technology
(Autonomous)
BACHUPALLY, MIYAPUR, HYDERABAD-500090
Gokaraju Rangaraju Institute of Engineering & Technology
(Autonomous)
Bachupally, Kukatpally
HYDERABAD 500090.

Matlab/Labview Lab Record

CERTIFICATE

This is to certify that it is a Bonafide Record of practical work done in the Matlab/Labview Laboratory in I sem of II Year during the year.............................

Name: ..................................................
Roll no: ..................................................
Course: B. Tech. ........... Year...... Semester.......... 
Branch: ..................................................

SIGNATURE OF STAFF MEMBER
# Contents

## Matlab:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Diode Characteristics</td>
<td>1</td>
</tr>
<tr>
<td>2. MOSFET Characteristics</td>
<td>6</td>
</tr>
<tr>
<td>3. IGBT Characteristics</td>
<td>12</td>
</tr>
<tr>
<td>4. Transient Analysis Of Linear Circuit</td>
<td>17</td>
</tr>
<tr>
<td>5. Single Phase Half wave Diode Rectifier</td>
<td>23</td>
</tr>
<tr>
<td>6. Single Phase Full Wave Diode Bridge Rectifier</td>
<td>27</td>
</tr>
<tr>
<td>7. Single Phase Full Wave Diode Bridge Rectifier With LC Filter</td>
<td>31</td>
</tr>
<tr>
<td>8. Three Phase Half wave Diode Rectifier</td>
<td>35</td>
</tr>
<tr>
<td>9. Starting Of A 5 HP 240V DC Motor With A Three-Step Resistance Starter</td>
<td>39</td>
</tr>
</tbody>
</table>

## Labview:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Simple Amplitude Measurement</td>
<td>42</td>
</tr>
<tr>
<td>2. Building Arrays Using For Loop And While Loop</td>
<td>46</td>
</tr>
<tr>
<td>3. Random Signal Generation</td>
<td>49</td>
</tr>
<tr>
<td>4. Waveform Minimum &amp; Maximum Value Display</td>
<td>55</td>
</tr>
<tr>
<td>5. Wave At Interface</td>
<td>58</td>
</tr>
<tr>
<td>6. Force Mass Spring Damper</td>
<td>60</td>
</tr>
<tr>
<td>7. Matrix Fundamentals</td>
<td>63</td>
</tr>
<tr>
<td>8. Simple Pendulum</td>
<td>66</td>
</tr>
</tbody>
</table>

## Scilab:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Single Phase Half wave Diode Rectifier</td>
<td>69</td>
</tr>
<tr>
<td>2. Creating the Vectors</td>
<td>71</td>
</tr>
</tbody>
</table>
COURSE OBJECTIVES

1. To provide students with a strong background on Matlab/Labview softwares.
2. To train the students how to approach for solving engineering problems.
3. To prepare the students to use Matlab/Labview in their project works.
4. To provide a foundation for use of these softwares in real time applications

COURSE OUTCOMES

1. An ability to express programming and simulation for engineering programs.
2. An ability to find importance of these softwares for lab experimentation.
3. Articulate importance of softwares in research by simulation work.
4. An in-depth knowledge of providing virtual instruments on Labview environment.
<table>
<thead>
<tr>
<th>S.No</th>
<th>Date</th>
<th>Topic</th>
<th>Page no</th>
<th>Signature of the Faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. DIODE CHARACTERISTICS

AIM: To draw the characteristic curves of diode.

APPARATUS: SOFTWARE REQUIRED: MAT LAB SIMULINK

THEORY:

Diodes are active devices constructed to allow current to flow in one direction. The diode consists of N-type and P-type materials (see diagram shown below). Each of these materials originally consisted of pure silicon doped to obtain the type of characteristic desired. Doping is the process of adding impurities to the pure semiconductor material. N-type material is formed when the impurities with five electrons in the outer most shell (pentavalent) are added to the pure semiconductor material. Pentavalent materials are elements such as antimony, arsenic, and phosphorus. The same procedure is then performed for the P-type material using an atom containing only three electrons in its outer shell (trivalent). Trivalent materials are elements such as boron, gallium, and indium. A diode is formed when a piece of pure material is doped half as N-type and half as P-type material. It is not constructed by fusing the N and P type materials together. The N-type material is called the cathode and the P-type material is called the anode. A junction, called the depletion region, is formed where the two materials meet. Some of the free electrons begin diffusing across the junction and fill the holes in the P-type material. When this occurs the atom that the electron joined becomes a negative ion. When an electron leaves N-type material, it leaves a hole creating a positive ion. Eventually the diffusion of the free electrons and holes in the junction of the two materials will decrease to a point where it will stop. The area of positive and negative ions created around the junction is called the depletion layer. The free electrons in the N-type material are blocked from diffusing to the P-type material by the negative ions in the P material.
Holes from the P material are blocked by the positive ions in the N material. For charge carriers to flow through the layer, a small voltage potential of approximately 0.7V (silicon) or .3V (germanium) is required to break it down. This is called the barrier potential. There are two types of biasing that can be applied to a diode. For a diode to be forward biased, a power supply is connected with the positive terminal to the P-type material (anode) and the negative terminal to the N-type material (cathode). As the potential across the diode approaches the barrier, the diode begins conducting. For a diode to be reversed biased, the power supply leads are set up with the negative terminal attached to the P-type material and the positive terminal attached to the N-type material. With this configuration, the electrons in the N-type material and the holes in the P-type material are drawn away from the depletion layer increasing the width of the layer. Even though the majority current has stopped flowing, minority current is still flowing due to thermal energy. Minority current is also called saturation current and can only be increased by an increase in temperature. There is also a second minute current that flows through the resistive paths created by surface impurities. This type of current is called surface-leakage current. The sum of saturation current and the surface-leakage current is called the reverse current.

A diode, such as the silicon diode, will conduct in the reverse bias direction with a significant amount of voltage applied. At the point of conduction, the diode breaks down resulting in the diode being damaged. The value of voltage at which the diode breaks down is called the breakdown voltage. Other types of diodes, such as zener diodes, function in the reverse bias region. The effect of a zener diode breaking down in the reverse region allows it to be used as a voltage regulator. The graph of a silicon diode shows the response curve including the forward and reverse bias regions (See Figure 1 next page). The graph of the silicon diode shows the point in the forward bias region where the barrier potential is exceeded and the diode begins conduction. The voltage at which the diode begins conducting is called the knee voltage. In the reverse bias region, the point at which the diode begins breaking down is also shown. The voltage level at the moment of breakdown is called the breakdown voltage as mentioned previously.
Circuit diagram

![Circuit Diagram](image)

**VI characteristics:**

![VI Characteristics Graph](image)

**FIGURE 1**
Calculations:
Graph:

Result:
2. MOSFET CHARACTERISTICS

AIM: To draw the characteristic curves for an N-channel MOSFET

APPARATUS: SOFTWARE REQUIRED: MATLAB SIMULINK

Theory: The metal–oxide–semiconductor field-effect transistor (MOSFET) is a transistor used for amplifying or switching electronic signals. In MOSFETs, a voltage on the oxide-insulated gate electrode can induce a conducting channel between the two other contacts called source and drain. The channel can be of n-type or p-type, and is accordingly called an nMOSFET or a pMOSFET. Figure 1 shows the schematic diagram of the structure of an nMOS device before and after channel formation.

<table>
<thead>
<tr>
<th>Drain (D)</th>
<th>Drain (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gate (G)</td>
<td>Gate (G)</td>
</tr>
<tr>
<td>Source (S)</td>
<td>Source (S)</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
</tr>
</tbody>
</table>

Fig. (1a): nMOSFET before channel formation

Fig. (1b): nMOSFET structure after channel formation
Output Characteristics

MOSFET output characteristics plot $I_D$ versus $V_{DS}$ for several values of $V_{GS}$.

The characteristics of an nMOS transistor can be explained as follows. As the voltage on the top electrode increases further, electrons are attracted to the surface. At a particular voltage level, which we will shortly define as the threshold voltage, the electron density at the surface exceeds the hole density. At this voltage, the surface has inverted from the p-type polarity of the original substrate to an n-type inversion layer, or inversion region, directly underneath the top plate as indicated in Fig. 1(b). This inversion region is an extremely shallow layer, existing as a charge sheet directly below the gate. In the MOS capacitor, the high density of electrons in the inversion layer is supplied by the electron–hole generation process within the depletion layer. The positive charge on the gate is balanced by the combination of negative charge in the inversion layer plus negative ionic acceptor charge in the depletion layer. The voltage at which the surface inversion layer just forms plays an extremely important role in field-effect transistors and is called the threshold voltage $V_{th}$. The region of output characteristics where $V_{GS} < V_{th}$ and no current flows is called the cut-off region. When the channel forms in the nMOS (pMOS) transistor, a positive (negative) drain voltage with respect to the source creates a horizontal electric field moving the electrons (holes) toward the drain forming a positive (negative) drain current coming into the transistor. The positive current convention is used for electron and hole current, but in both cases electrons are the actual charge carriers. If the channel horizontal electric field is of the same order or smaller than the vertical thin oxide field, then the inversion channel remains almost uniform along the device length. This continuous carrier profile from drain to source puts the transistor in a bias state that is equivalently called either the non saturated, linear, or ohmic bias state. The drain and source are effectively short-circuited. This
happens when $V_{GS} > V_{DS} + V_{tn}$ for nMOS transistor and $V_{GS} < V_{DS} + V_{tp}$ for pMOS transistor. Drain current is linearly related to drain-source voltage over small intervals in the linear bias state.

But if the nMOS drain voltage increases beyond the limit, so that $V_{GS} < V_{DS} + V_{tn}$, then the horizontal electric field becomes stronger than the vertical field at the drain end, creating an asymmetry of the channel carrier inversion distribution.

If the drain voltage rises while the gate voltage remains the same, then $V_{GD}$ can go below the threshold voltage in the drain region. There can be no carrier inversion at the drain-gate oxide region, so the inverted portion of the channel retracts from the drain, and no longer “touches” this terminal.

The pinched-off portion of the channel forms a depletion region with a high electric field. The n-drain and p-bulk form a pn junction. When this happens the inversion channel is said to be “pinched-off” and the device is in the **saturation region**. The characteristics can be loosely modelled by the following equations.

\[
I_D = \begin{cases} 
0, & \text{Cut-off state} \\
\frac{\mu_n \varepsilon_{ox} W}{T_{ox}} \frac{W}{L} \left( (V_{GS} - V_{tn}) V_{DS} - \frac{V_{DS}^2}{2} \right), & \text{Ohmic state} \\
\frac{\mu_p \varepsilon_{ox}}{2T_{ox}} \frac{W}{L} (V_{GS} - V_{tn})^2, & \text{Saturation state}
\end{cases}
\]
Circuit Diagram:

MOSFET Characteristics

This model generates the characteristic curves for an N-channel MOSFET. Define the vector of gate voltages and minimum and maximum drain-source voltages by double clicking on the block labelled 'Define $V_g$ and $V_d$'. Then double click on the block labelled 'Generate Characteristics'.

![Graph of MOSFET Characteristics](image-url)
Calculations:
Graph:

Result:
3. IGBT CHARACTERISTICS

AIM: To draw the characteristic curves for IGBT

APPARATUS: MatLab Simulink

Theory:
The Insulated Gate Bipolar Transistor, (IGBT) uses the insulated gate (hence the first part of its name) technology of the MOSFET with the output performance characteristics of a conventional bipolar transistor, (hence the second part of its name). The result of this hybrid combination is that the “IGBT Transistor” has the output switching and conduction characteristics of a bipolar transistor but is voltage-controlled like a MOSFET.

IGBTs are mainly used in power electronics applications, such as inverters, converters and power supplies, were the demands of the solid state switching device are not fully met by power bipolars and power MOSFETs. High-current and high-voltage bipolars are available, but their switching speeds are slow, while power MOSFETs may have high switching speeds, but high-voltage and high-current devices are expensive and hard to achieve.

The advantage gained by the insulated gate bipolar transistor device over a BJT or MOSFET is that it offers greater power gain than the bipolar type together with the higher voltage operation and lower input losses of the MOSFET. In effect it is an FET integrated with a bipolar transistor in a form of Darlington configuration as shown.

Insulated Gate Bipolar Transistor

We can see that the insulated gate bipolar transistor is a three terminal, transconductance device that combines an insulated gate N-channel MOSFET input with a PNP bipolar transistor output connected in a type of Darlington configuration. As a result the terminals are labelled
as: **Collector**, **Emitter** and **Gate**. Two of its terminals (C-E) are associated with a conductance path and the third terminal (G) associated with its control.

The amount of amplification achieved by the insulated gate bipolar transistor is a ratio between its output signal and its input signal. For a conventional bipolar junction transistor, (BJT) the amount of gain is approximately equal to the ratio of the output current to the input current, called Beta.

For a metal oxide semiconductor field effect transistor or MOSFET, there is no input current as the gate is isolated from the main current carrying channel. Therefore, an FET’s gain is equal to the ratio of output current change to input voltage change, making it a transconductance device and this is also true of the IGBT. Then we can treat the IGBT as a power BJT whose base current is provided by a MOSFET.

The **Insulated Gate Bipolar Transistor** can be used in small signal amplifier circuits in much the same way as the BJT or MOSFET type transistors. But as the IGBT combines the low conduction loss of a BJT with the high switching speed of a power MOSFET an optimal solid state switch exists which is ideal for use in power electronics applications.

Also, the IGBT has a much lower “on-state” resistance, $R_{ON}$ than an equivalent MOSFET. This means that the $I^2R$ drop across the bipolar output structure for a given switching current is much lower. The forward blocking operation of the IGBT transistor is identical to a power MOSFET.

When used as static controlled switch, the insulated gate bipolar transistor has voltage and current ratings similar to that of the bipolar transistor. However, the presence of an isolated gate in an IGBT makes it a lot simpler to drive than the BJT as much less drive power is needed.

An insulated gate bipolar transistor is simply turned “ON” or “OFF” by activating and deactivating its Gate terminal. A constant positive voltage input signal across the Gate and the Emitter will keep the device in its “ON” state, while removal of the input signal will cause it to turn “OFF” in much the same way as a bipolar transistor or MOSFET.

**IGBT Characteristics**

![IGBT Circuit Diagram and Characteristics Graph](image-url)
Because the IGBT is a voltage-controlled device, it only requires a small voltage on the Gate to maintain conduction through the device unlike BJT’s which require that the Base current is continuously supplied in a sufficient enough quantity to maintain saturation.

Also the IGBT is a unidirectional device, meaning it can only switch current in the “forward direction”, that is from Collector to Emitter unlike MOSFET’s which have bi-directional current switching capabilities (controlled in the forward direction and uncontrolled in the reverse direction).

The principal of operation and Gate drive circuits for the insulated gate bipolar transistor are very similar to that of the N-channel power MOSFET. The basic difference is that the resistance offered by the main conducting channel when current flows through the device in its “ON” state is very much smaller in the IGBT. Because of this, the current ratings are much higher when compared with an equivalent power MOSFET.

The main advantages of using the Insulated Gate Bipolar Transistor over other types of transistor devices are its high voltage capability, low ON-resistance, ease of drive, relatively fast switching speeds and combined with zero gate drive current makes it a good choice for moderate speed, high voltage applications such as in pulse-width modulated (PWM), variable speed control, switch-mode power supplies or solar powered DC-AC inverter and frequency converter applications operating in the hundreds of kilohertz range.

Circuit diagram:
Output characteristics:

![Graph showing output characteristics with various Vge values.]

Calculations:
Graph:

Result:
4. **Transient analysis of linear circuit**

**Aim:** To observe the transient response of a linear circuit

**Apparatus:** Matlab simulink

**Theory:**

Series RLC circuit:

The circuit shown on Figure 1 is called the series RLC circuit. We will analyze this circuit in order to determine its transient characteristics once the switch S is closed.

![Series RLC Circuit Diagram](image)

The equation that describes the response of the system is obtained by applying KVL around the mesh:

\[ v_R + v_L + v_C = V_S \]  \hspace{1cm} (1.1)

The current flowing in the circuit is

\[ i = C \frac{dv_C}{dt} \]  \hspace{1cm} (1.2)

And thus the voltages \( v_R \) and \( v_L \) are given by

\[ v_R = iR = RC \frac{dv_C}{dt} \]  \hspace{1cm} (1.3)

\[ v_L = L \frac{di}{dt} = LC \frac{d^2v_C}{dt^2} \]  \hspace{1cm} (1.4)

Substituting Equations (1.3) and (1.4) into Equation (1.1) we obtain The solution to equation (1.5) is the linear combination of the homogeneous and the particular solution \( v_C = v_{C_p} + v_{C_h} \)

The particular solution is
\[
\frac{d^2v_c}{dt^2} + \frac{R}{L} \frac{dv_c}{dt} + \frac{1}{LC} v_c = \frac{1}{LC} V_s
\]  
(1.5)

Assuming a homogeneous solution is of the form \(As^\alpha\) and by substituting into Equation (1.7) we obtain the characteristic equation

\[
s^2 + \frac{R}{L} s + \frac{1}{LC} = 0
\]  
(1.8)

By defining

\[
\alpha = \frac{R}{2L} : \text{Damping rate}
\]  
(1.9)

And

\[
\omega_o = \frac{1}{\sqrt{LC}} : \text{Natural frequency}
\]  
(1.10)

The characteristic equation becomes

\[
s^2 + 2\alpha s + \omega_o^2 = 0
\]  
(1.11)

The roots of the characteristic equation are

\[
s1 = -\alpha + \sqrt{\alpha^2 - \omega_o^2}
\]  
(1.12)

\[
s2 = -\alpha - \sqrt{\alpha^2 - \omega_o^2}
\]  
(1.13)

And the homogeneous solution becomes

\[
v_c_h = A_1 e^{s_1 t} + A_2 e^{s_2 t}
\]  
(1.14)

The total solution now becomes

\[
v_c = V_s + A_1 e^{s_1 t} + A_2 e^{s_2 t}
\]  
(1.15)

The parameters \(A1\) and \(A2\) are constants and can be determined by the application of the initial conditions of the system \(v_c(t = 0)\) and \(\frac{dv_c(t = 0)}{dt}\).
The value of the term determines the behavior of the response. Three types of responses are possible:

1. \( \alpha = \omega_o \) then \( s1 \) and \( s2 \) are equal and real numbers: no oscillatory behavior
   
   **Critically Damped System**

2. \( \alpha > \omega_o \). Here \( s1 \) and \( s2 \) are real numbers but are unequal: no oscillatory behavior

   **Over Damped System**
   
   \[ v_c = V_S + A_1 e^{s1t} + A_2 e^{s2t} \]

3. \( \alpha < \omega_o \cdot \sqrt{\alpha^2 - \omega_o^2} = j\sqrt{\alpha^2 - \omega_o^2} \) In this case the roots \( s1 \) and \( s2 \) are complex numbers:
   
   \( s1 = -\alpha + j\sqrt{\alpha^2 - \omega_o^2} \), \( s2 = -\alpha - j\sqrt{\alpha^2 - \omega_o^2} \). System exhibits oscillatory behavior
   
   **Under Damped System**

---

**Circuit diagram:**

---

**Transient Analysis of a Linear Circuit**
Waveforms:
Calculations:
Graph:

Result:
5. Single Phase Halfwave Diode Rectifier

**Aim:** To simulate Single phase Halfwave diode rectifier with resistive load in matlab simulink.

**Apparatus:** Matlab

**Theory:**
Rectification is a process of conversion of alternating input voltage to direct output voltage. A rectifier converts ac power to dc power. In a single phase halfwave rectifier, for one cycle of supply voltage, there is one half cycle of output or load voltage. The circuit diagram is shown in the figure. During the positive half cycle, diode is forward biased, it therefore conducts from \( \omega t=0^\circ \) to \( \omega t=\pi \). During positive half cycle, output voltage \( v_o=\text{source voltage } v_s \) and load current \( i_o=v_o/R \). At \( \omega t=\pi, v_o=0 \) and for R load, \( i_o \) is also zero. As soon as \( v_s \) tends to become negative after \( \omega t=\pi \), diode D reverse biased, it is therefore turned off and goes into blocking state. Output voltage as well as output current, are zero from \( \omega t=\pi \) to \( \omega t=2\pi \). After \( \omega t=2\pi \), diode is again forward biased and conduction begins.

**Circuit Diagram:**

![Circuit Diagram](image-url)
Wave forms:
Calculations:
Graph:

Result:
6. Single Phase Full Wave Diode Bridge Rectifier

**Aim:** To simulate Single phase full wave diode bridge rectifier with resistive load in matlab simulink.

**Apparatus:** Matlab

**Theory:**
Rectification is a process of conversion of alternating input voltage to direct output voltage. A rectifier converts ac power to dc power. In a single phase full wave diode bridge rectifier, during the positive half cycle of the supply source, diodes D1 & D2 are forward biased and then conducts from $\omega t=0^\circ$ to $\omega t=\pi$. The diodes D3 & D4 are reverse biased. During positive half cycle, output voltage $v_o=$source voltage $v_s$ and load current $i_o=v_o/R$. At $\omega t=\pi$, $v_o=0$ and for R load, $i_o$ is also zero. As soon as $v_s$ tends to become negative after $\omega t=\pi$, diodes D1 & D2 are reverse biased, it is therefore turned off and goes into blocking state and diodes D3 & D4 becomes forward biased and the output voltage becomes $-v_s$.

**Circuit Diagram:**

![Circuit Diagram](image-url)
Wave forms:
Calculations:
Graph:

Result:
**7. Single Phase Full Wave Diode Bridge Rectifier With LC Filter**

**Aim:** To simulate Single phase full wave diode bridge rectifier with LC filter in matlab simulink.

**Apparatus:** Matlab simulink

**Theory:**

Rectification is a process of conversion of alternating input voltage to direct output voltage. A rectifier converts ac power to dc power. In a single phase full wave diode bridge rectifier, during the positive half cycle of the supply source, diodes D1 & D2 are forward biased and then conducts from $\omega t=0^\circ$ to $\omega t=\pi$. The diodes D3 & D4 are reverse biased. During positive half cycle, output voltage $v_o=$source voltage $v_s$ and load current $i_o=v_o/R$. At $\omega t=\pi$, $v_o=0$ and for R load, $i_o$ is also zero. As soon as $v_s$ tends to become negative after $\omega t=\pi$, diodes D1 & D2 are reverse biased, it is therefore turned off and goes into blocking state and diodes D3 & D4 becomes forward biased and the output voltage becomes $-v_s$. L-C filter: An LC filter consists of inductor L in series with the load and capacitor C across the load. It reduces the ripple from the output voltage of a single phase full wave diode rectifier. The inductor L blocks the dominant harmonics. Capacitor C provides an easy path to the nth harmonic ripple currents.

**Circuit Diagram:**

![Circuit Diagram](image)
Waveforms:
Calculations:
Graph:

Result:
8. Three Phase Halfwave Diode Rectifier

**Aim:** To simulate Three phase halfwave diode rectifier in matlab simulink.

**Apparatus:** Matlab simulink

**Theory:**

Rectification is the process of conversion of alternating input voltage to direct output voltage. In diode rectifiers, the output voltage cannot be controlled. Three-Phase Half-Wave Rectifier

For higher power application and where three-phase power supply is available, a three phase bridge rectifier, as shown in figure (1), should be used. One diode is conduct at any instant. It is the diode connected to the phase having the highest instantaneous voltage. The output voltage of the successive phase voltages and varying from \( V_m/2 \) to \( V_m \), three times per input cycle. The average output voltage is:

\[
V_{dc} = \frac{3}{2\pi} \int_{\pi/6}^{5\pi/6} V_m \sin \theta \, d\theta
\]

\[
V_{dc} = \frac{3\sqrt{3}}{2\pi} V_m = 0.827 V_m
\]

Similarly, the rms value of the output voltage can be found as:

\[
V_L = \sqrt{\frac{3}{2\pi} \int_{\pi/6}^{5\pi/6} (V_m \sin \theta)^2 \, d\theta}
\]

\[
V_L = V_m \sqrt{\frac{3}{2\pi} \left( \frac{\pi}{3} + \frac{\sqrt{3}}{4} \right)} = 0.84 \ V_m
\]

The rectifier has a three pulse characteristics, and load current \( i_d \) of less ripple contents in relative to single-phase rectifiers, which characterize by two pulse output. The ripple frequency is \( 3f \) (where \( f \) is input frequency) and the required smoothing reactor at the load side is of smaller size.
Circuit diagram:

Wave forms:

Fig.(1): a) Three-phase star rectifier.
   b) Waveforms of voltage and current of the three-phase star rectifier
Calculations:
Graph:

Result:
9. Starting Of A 5 HP 240V DC Motor With A Three-Step Resistance Starter

Aim: To simulate Starting of a 5 HP 240V DC motor with a three-step resistance starter.

Apparatus: Matlab simulink

Theory:

Starting Methods of A DC Motor

Basic operational voltage equation of a DC motor is given as

\[ E = E_b + I_a R_a \]

and hence

\[ I_a = \frac{(E - E_b)}{R_a} \]

Now, when the motor is at rest, obviously, there is no back emf \( E_b \), hence armature current will be high at starting. This excessive current will:

1. blow out the fuses and may damage the armature winding and/or commutator brush arrangement.
2. produce very high starting torque (as torque is directly proportional to armature current), and this high starting torque will produce huge centrifugal force which may throw off the armature windings.

Thus to avoid above two drawbacks, starters are used for starting of DC machine.

Thus, to avoid the above dangers while starting a DC motor, it is necessary to limit the starting current. For that purpose, starters are used to start a DC motor. There are various starters like, 3 point starter, 4 point starter, No load release coil starter, thyristor starter etc.

The main concept behind every DC motor starter is, adding external resistance to the armature winding at starting.

Circuit diagram:
Plot:

Calculations:
Graph:

Result:
1. SIMPLE AMPLITUDE MEASUREMENT

AIM: To create a block diagram for measuring amplitude of sine wave.

Apparatus: LABVIEW software

Front Panel:

The **amplitude** of a periodic variable is a measure of its change over a single **period** (such as **time** or **spatial period**).

**Characteristics of a Sine Wave**

A wave form is a graph showing the variation, usually of voltage or current, against time. The horizontal axis shows the passing of time, progressing from left to right. The vertical axis shows the quantity measured. The AMPLITUDE of a sine wave is the maximum vertical distance reached, in either direction from the centre line of the wave. As a sine wave is symmetrical about its centre line, the amplitude of the wave is half the peak to peak value.
Peak value

The PEAK value of the wave is the highest value the wave reaches above a reference value. The reference value normally used is zero. In a voltage waveform the peak value may be labelled \( V_{PK} \) or \( V_{MAX} \) (I_{PK} or I_{MAX} in a current waveform).

If the sine wave being measured is symmetrical either side of zero volts (or zero amperes), meaning that the dc level or dc component of the wave is zero volts, then the peak value must be the same as the amplitude, that is half of the peak to peak value.

Determining the Average Value (Mean) of a Sine Wave

The AVERAGE value. This is normally taken to mean the average value of only half a cycle of the wave. If the average of the full cycle was taken it would of course be zero, as in a sine wave symmetrical about zero, there are equal excursions above and below the zero line.
Using only half a cycle, as illustrated in fig the average value (voltage or current) is always 0.637 of the peak value of the wave.

\[ V_{AV} = V_{PK} \times 0.637 \]

or

\[ I_{AV} = I_{PK} \times 0.637 \]

The RMS Value

The RMS or ROOT MEAN SQUARED value is the value of the equivalent direct (non varying) voltage or current which would provide the same energy to a circuit as the sine wave measured. That is, if an AC sine wave has a RMS value of 240 volts, it will provide the same energy to a circuit as a DC supply of 240 volts.

It can be shown that the RMS value of a sine wave is 0.707 of the peak value.

\[ V_{RMS} = V_{PK} \times 0.707 \quad \text{and} \quad I_{RMS} = I_{PK} \times 0.707 \]

Also, the peak value of a sine wave is equal to 1.414 x the RMS value.
Block Diagram:

Result:
2. BUILDING ARRAYS USING FOR LOOP AND WHILE LOOP

Aim: To build an array using FOR loop and WHILE loop

Apparatus: LABVIEW software

Front Panel:

Working and manipulating with Arrays is an important part in LabVIEW development. Arrays are very powerful to use in LabVIEW. In all your applications you would probably use both One-Dimensional Arrays and Two-Dimensional Arrays.

On the Front Panel using the Control palette we can create an array as follows (Array, Matrix & Cluster subpalette):

You drag and drop the empty Array on the Front Panel, next you find a Control or Indicator (Numeric, String, Boolean, etc.) and drag it into the empty Array. You can create an Array of (almost) any kind of Control or Indicator.

2D or multidimensional Array? Just drag the mouse in the Index display to the left and increase the dimension.
On the Block Diagram we have the following Array palette available from the Functions palette in LabVIEW.

Use the Array functions to create and manipulate arrays. The most useful Array functions are

In addition to using loops to read and process elements in an array, you also can use the For Loop and the While Loop to build arrays. Wire the output of a VI or function in the loop to the loop border. If you use a While Loop, right-click the resulting tunnel and select **Enable Indexing** from
the shortcut menu. If you use a For Loop, indexing is enabled by default. The output of the tunnel is an array of every value the VI or function returns after each loop iteration.

The For Loop uses auto-indexing as its default, which is the best method when the number of values is known. Wire the variable inside the inner loop directly to an array terminal. Since one For Loop is placed inside another For Loop, a 2D array will be produced.

The While Loop is the best method when the number of values is unknown so the user or program determines the size of the array. Wire the variable inside the loop directly to an array terminal. Then right-click on the tunnel and select "enable indexing".

**Block Diagram:**

![Block Diagram](image)

**Result:**

![Result](image)
3. RANDOM SIGNAL GENERATION

Aim: To generate Random signal

Apparatus: LABVIEW software

Front Panel:

Random signals are called non deterministic signals are those signals that take random values at any given time and must be characterized statistically.

In Electronic Circuits we need to produce many different types, frequencies and shapes of Signal Waveforms such as Square Waves, Rectangular Waves, Triangular Waves, Sawtooth Waveforms and a variety of pulses and spikes.

These types of signal waveform can then be used for either timing signals, clock signals or as trigger pulses. However, before we can begin to look at how the different types of waveforms are produced, we firstly need to understand the basic characteristics that make up Electrical Waveforms.

Technically speaking, Electrical Waveforms are basically visual representations of the variation of a voltage or current over time. In plain English this means that if we plotted these voltage or current variations on a piece of graph paper against a base (x-axis) of time, ( t ) the resulting plot or drawing would represent the shape of a Waveform as shown. There are many different types of electrical waveforms available but generally they can all be broken down into two distinctive groups.
1. Uni-directional Waveforms — these electrical waveforms are always positive or negative in nature flowing in one forward direction only as they do not cross the zero axis point. Common uni-directional waveforms include Square-wave timing signals, Clock pulses and Trigger pulses.

2. Bi-directional Waveforms — these electrical waveforms are also called alternating waveforms as they alternate from a positive direction to a negative direction constantly crossing the zero axis point. Bi-directional waveforms go through periodic changes in amplitude, with the most common by far being the Sine-wave.

Whether the waveform is uni-directional, bi-directional, periodic, non-periodic, symmetrical, non-symmetrical, simple or complex, all electrical waveforms include the following three common characteristics:

1) **Period**: – This is the length of time in seconds that the waveform takes to repeat itself from start to finish. This value can also be called the Periodic Time, \( T \) of the waveform for sine waves, or the Pulse Width for square waves.

2) **Frequency**: – This is the number of times the waveform repeats itself within a one second time period. Frequency is the reciprocal of the time period, \( f = 1/T \) with the standard unit of frequency being the Hertz, \((Hz)\).

3) **Amplitude**: – This is the magnitude or intensity of the signal waveform measured in volts or amps.

**Periodic Waveforms**

**Periodic waveforms** are the most common of all the electrical waveforms as it includes Sine Waves. The AC (Alternating Current) mains waveform in your home is a sine wave and one which constantly alternates between a maximum value and a minimum value over time.

The amount of time it takes between each individual repetition or cycle of a sinusoidal waveform is known as its “periodic time” or simply the Period of the waveform. In other words, the time it takes for the waveform to repeat itself.

Then this period can vary with each waveform from fractions of a second to thousands of seconds as it depends upon the frequency of the waveform. For example, a sinusoidal waveform which takes one second to complete its cycle will have a periodic time of one second. Likewise a sine wave which takes five seconds to complete will have a periodic time of five seconds and so on.
So, if the length of time it takes for the waveform to complete one full pattern or cycle before it repeats itself is known as the “period of the wave” and is measured in seconds, we can then express the waveform as a period number per second denoted by the letter T as shown below.

**Sine Wave form**

![Sine Wave form Diagram]

Relationship between Frequency and Periodic Time:

\[
\text{Frequency} = \frac{1}{\text{Periodic time}} \quad \text{or} \quad f = \frac{1}{T} \quad \text{Hz}
\]

\[
\text{Periodic time} = \frac{1}{\text{Frequency}} \quad \text{or} \quad T = \frac{1}{f} \quad \text{sec}
\]

**Square Waveform**

Square-wave Waveforms are used extensively in electronic and micro electronic circuits for clock and timing control signals as they are symmetrical waveforms of equal and square duration representing each half of a cycle and nearly all digital logic circuits use square wave waveforms on their input and output gates. Unlike sine waves which have a smooth rise and fall waveform with rounded corners at their positive and negative peaks, square waves on the other hand have very steep almost vertical up and down sides with a flat top and bottom producing a waveform which matches its description, – “Square” as shown below.
Rectangular Waveforms

Rectangular Waveforms are similar to the square wave waveform above, the difference being that the two pulse widths of the waveform are of an unequal time period. Rectangular waveforms are therefore classed as “Non-symmetrical” waveforms as shown below.

\[
\text{Frequency} = \frac{1}{\text{"ON" time} + \text{"OFF" time}}
\]

Triangular Waveforms

Triangular Waveforms are generally bi-directional non-sinusoidal waveforms that oscillate between a positive and a negative peak value. Although called a triangular waveform, the triangular wave is actually more of a symmetrical linear ramp waveform because it is simply a slow rising and falling voltage signal at a constant frequency or rate. The rate at which the voltage changes between each ramp direction is equal during both halves of the cycle as shown below.
Sawtooth Waveforms

Sawtooth Waveforms are another type of periodic waveform. As its name suggests, the shape of the waveform resembles the teeth of a saw blade. Sawtooth waveforms can have a mirror image of themselves, by having either a slow-rising but extremely steep decay, or an extremely steep almost vertical rise and a slow-decay as shown below.

Set the inputs to the desired values and run the VI. Notice that the time changes with each run until the reset signal input is set to ON. The phase will also change if the frequency is non-integer.
Block Diagram:

Result:
4. WAVEFORM MINIMUM & MAXIMUM VALUE DISPLAY

**Aim:** To find minimum and maximum value of a given waveform

**Apparatus:** LABVIEW software

Front Panel:

![Waveform Minimum & Maximum Value Display](image)

Determines the maximum and minimum values and their associate time values for a waveform.

Enter an amplitude and frequency and run the VI. A sine wave will be generated and the maximum and minimum values and their positions returned. Note that due to sampling errors, the maximum and minimum may not be what you expect from the amplitude input.

- **waveform in** is the waveform for which you want to retrieve the maximum and minimum values.
- **error in** describes error conditions that occur before this node runs. This input provides standard error in functionality.
- **max time** is the time value at which the maximum data value was reached.
- **waveform out** returns waveform in unchanged.
- **Y max** is the maximum data value of the waveform.
- **Y min** is the minimum data value of the waveform.
- **error out** contains error information. This output provides standard error out functionality.
**min time** is the time value at which the minimum data value was reached

**Block Diagram:**

![Block Diagram Image]
Result:
5. WAVE AT INTERFACE

Aim: To observe the wave at interface

Apparatus: LABVIEW software

Front Panel:

This experiment uses Detect Zero Crossing functions to model a wave front that passes through one material, through a slab of another material, and back out into the original material. This model uses a variation of Huygen's principle, which states that each point on a wave front acts as a point source for another wave, and the sum of all those secondary waves is the overall wave, with the additional constraint that the wave fronts propagate at c/n, where c is the speed of the wave in a vacuum, and n is the index of refraction of the given material.

Block Diagram:
Result:
6. FORCE MASS SPRING DAMPER

**Aim:** To demonstrate the effect of a forcing function applied to an initially resting mass-spring-damper system

**Apparatus:** LABVIEW software

Front Panel:

This Simulation demonstrates the effect of a forcing function applied to an initially resting mass-spring-damper system.

You can adjust the frequency and signal type of the forcing signal and observe the effect upon the system.

In this you will construct a simulation diagram that represents the behavior of a dynamic system.

You will simulate a spring-mass damper system.
\[ F(t) - c\dot{x}(t) - kx(t) = m\ddot{x}(t) \]

where \( t \) is the simulation time, \( F(t) \) is an external force applied to the system, \( c \) is the damping constant of the spring, \( k \) is the stiffness of the spring, \( m \) is a mass, and \( x(t) \) is the position of the mass. \( \dot{x} \) is the first derivative of the position, which equals the velocity of the mass. \( \ddot{x} \) is the second derivative of the position, which equals the acceleration of the mass.

The following figure shows this dynamic system:

![Dynamic System Diagram]

The goal is to view the position \( x(t) \) of the mass \( m \) with respect to time \( t \). You can calculate the position by integrating the velocity of the mass. You can calculate the velocity by integrating the acceleration of the mass. If you know the force and mass, you can calculate this acceleration by using Newton's Second Law of Motion, given by the following equation:

Force = Mass \times Acceleration

Therefore,

Acceleration = Force / Mass

Substituting terms from the differential equation above yields the following equation

\[ \ddot{x} = \frac{1}{m} (F - c\dot{x} - kx) \]
Block Diagram:

Result:
7. MATRIX FUNDAMENTALS

AIM: To learn the fundamentals of matrix.

APPARATUS: LABVIEW

Front Panel:

The following example shows the fundamental behavior and functionality of the matrix data type. While the test is running, you can change the matrix inputs or select built-in or user-defined numeric operations. Change the operation mode between matrix and array versions of the operations to see the difference in results.

Instructions:

1) Run the VI.

2) Modify the inputs, operation and mode, and view the results in the output matrix control.
Block Diagram:
Result:
8. SIMPLE PENDULUM

AIM: To study the working of simple pendulum

APPARATUS: LABVIEW

Front Panel:

A pendulum is a weight suspended from a pivot so that it can swing freely. When a pendulum is displaced sideways from its resting equilibrium position, it is subject to a restoring force due to gravity that will accelerate it back toward the equilibrium position. When released, the restoring force combined with the pendulum’s mass causes it to oscillate about the equilibrium position, swinging back and forth. The time for one complete cycle, a left swing and a right swing, is called
the period. The period depends on the length of the pendulum, and also to a slight degree on the amplitude, the width of the pendulum's swing.

Consider a mass $m$ suspended from a light inextensible string of length $l$, such that the mass is free to swing from side to side in a vertical plane, as shown in Fig. This setup is known as a simple pendulum. Let $\theta$ be the angle subtended between the string and the downward vertical. Obviously, the equilibrium state of the simple pendulum corresponds to the situation in which the mass is stationary and hanging vertically down (i.e., $\theta = 0$).

\[
I = \text{constant}
\]

where $I$ is the moment of inertia of the mass, and $\tau$ is the torque acting on the system. For the case in hand, given that the mass is essentially a point particle, and is situated a distance $l$ from the axis of rotation (i.e., the pivot point), it is easily seen that $I = ml^2$

Instructions:

1) Run the VI.

2) Change the Initial Position and run the VI again to see how the simple pendulum swings with different initial positions.

3) Look at the block diagram to see how to implement the model with a cluster of formula strings and variable strings.
Block Diagram:

Result:
1. Single phase Halfwave diode rectifier

**Aim:** To write a program for calculation of parameters of Single phase halfwave diode rectifier using Sci lab

**Apparatus:** Sci lab software

**Program:**

```matlab
clear;
Vp_sec=230*2^0.5/4;
alph=asind(12/Vp_sec);
alph1=180-alph;
// the diode will conduct from 8.89 degree to 171.51 degree
Angle_conduction=alph1-alph;
printf("ConductionAngle=%.2f degree",Angle_conduction)
Idc=4;
R=1/(2*Idc*pi)*(2*Vp_sec*cosd(alph)+(2*12*alph*pi/180)-12*pi);
printf("Resistance=%.2f ohm",R)
Irms=((1/(2*pi*R^2))*(((Vp_sec^2/2+12^2)*((pi-2*alph*pi/180))+(Vp_sec^2/2*sind(2*alph))-(4*Vp_sec*12*cosd(alph))))^0.5;
P_rating=Irms^2*R;
printf("Power rating of resistor=%.2f W",P_rating)
Pdc=12*Idc;
t Charging=150/Pdc;
printf("Charging time=%.3f h",t Charging)
Rectifier efficiency = Pdc/(Pdc+Irms^2*R);
printf("Rectifier efficiency=%.2f",Rectifier efficiency)
PIV=Vp_sec+12;
printf("PIV=%.3f V",PIV)
```

**Output:**

- Conduction Angle = 163.027726 degree
- Resistance = 5.04 ohm
- Power rating of resistor = 218.51 W
- Charging time = 3.125 h
- Rectifier efficiency = 0.18
- PIV = 93.317 V
Calculations:

Result:
2. Creating the vectors

**Aim:** To create the vectors using scilab commands

**Apparatus:** Sci lab

**Exercise programs:**

1. **Create the vector** 
\( \{x_1^2, x_2^2, x_3^2, x_4^2\} \) **with** \( x=1,2,3,4 \)

**code:**
```scilab
x=1:4;
y=x.^2
```

**Result:**
```
y  =
    1.    4.    9.    16.
```

2. **Create the vector** \( \{x_1+1, x_2+1, x_3+1, x_4+1\} \) **with** \( x=1,2,3,4 \)

**code:**
```scilab
x=1:4;
y=x+1
```

**Result:**
```
y  =
    2.    3.    4.    5.
```

3. **Create the vector** \( \{x_1*y_1, x_2*y_2, x_3*y_3, x_4*y_4\} \) **with** \( x=1,2,3,4 \) & \( y=5,6,7,8 \)

**code:**
```scilab
x=1:4;
y=5:8;
z=x.*y
```

**Result:**
```
z =
    5.   12.   21.   32.
```

4. **Create the vector** \( \{x_1/y_1, x_2/y_2, x_3/y_3, x_4/y_4\} \) **with** \( x=12*(6:9) \) & \( y=1,2,3,4 \)

**code:**
```scilab
x=12*(6:9);
y=1:4;
z=x./y
```
Result:
\[ z = \]
72. 42. 32. 27.

5. Create the vector \([\sin(x_1), \sin(x_2), \sin(x_3), \sin(x_4)]\) with \(x\) is a vector of 10 values linearly chosen in the interval \([0, \pi]\)

code:
```matlab
x = linspace(0, pi, 10);
y = sin(x);
```

Result:
\[ x = \]
column 1 to 5

0. 0.3490659 0.6981317 1.0471976 1.3962634

column 6 to 10

1.7453293 2.0943951 2.443461 2.7925268 3.1415927

\[ y = \]
column 1 to 5

0. 0.3420201 0.6427876 0.8660254 0.9848078

column 6 to 10

0.9848078 0.8660254 0.6427876 0.3420201 1.225D-16

6. Compute \(y = f(x)\) and Draw the function \(f(x) = \log_{10}(r/10^x + 10^x)\) with \(r = 2.220D-16\) for \(x\) a vector of 100 values linearly chosen in the interval \([-16, 0]\)

code:
```matlab
r = 2.220D-16;
x = linspace(-16,0,100);
y = log10(r./10.^x + 10.^x);
plot(x,y)
```
Plot:

Result:
How to Contact MathWorks

www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support

@ suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information

508-647-7000 (Phone)
508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB® Primer


The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government’s needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.
**Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Edition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>December 1996</td>
<td>First printing</td>
<td>For MATLAB 5</td>
</tr>
<tr>
<td>May 1997</td>
<td>Second printing</td>
<td>For MATLAB 5.1</td>
</tr>
<tr>
<td>September 1998</td>
<td>Third printing</td>
<td>For MATLAB 5.3</td>
</tr>
<tr>
<td>September 2000</td>
<td>Fourth printing</td>
<td>Revised for MATLAB 6 (Release 12)</td>
</tr>
<tr>
<td>June 2001</td>
<td>Online only</td>
<td>Revised for MATLAB 6.1 (Release 12.1)</td>
</tr>
<tr>
<td>July 2002</td>
<td>Online only</td>
<td>Revised for MATLAB 6.5 (Release 13)</td>
</tr>
<tr>
<td>August 2002</td>
<td>Fifth printing</td>
<td>Revised for MATLAB 6.5</td>
</tr>
<tr>
<td>June 2004</td>
<td>Sixth printing</td>
<td>Revised for MATLAB 7.0 (Release 14)</td>
</tr>
<tr>
<td>October 2004</td>
<td>Online only</td>
<td>Revised for MATLAB 7.0.1 (Release 14SP1)</td>
</tr>
<tr>
<td>March 2005</td>
<td>Online only</td>
<td>Revised for MATLAB 7.0.4 (Release 14SP2)</td>
</tr>
<tr>
<td>June 2005</td>
<td>Seventh printing</td>
<td>Minor revision for MATLAB 7.0.4 (Release 14SP2)</td>
</tr>
<tr>
<td>September 2005</td>
<td>Online only</td>
<td>Minor revision for MATLAB 7.1 (Release 14SP3)</td>
</tr>
<tr>
<td>March 2006</td>
<td>Online only</td>
<td>Minor revision for MATLAB 7.2 (Release 2006a)</td>
</tr>
<tr>
<td>September 2006</td>
<td>Eighth printing</td>
<td>Minor revision for MATLAB 7.3 (Release 2006b)</td>
</tr>
<tr>
<td>March 2007</td>
<td>Ninth printing</td>
<td>Minor revision for MATLAB 7.4 (Release 2007a)</td>
</tr>
<tr>
<td>September 2007</td>
<td>Tenth printing</td>
<td>Minor revision for MATLAB 7.5 (Release 2007b)</td>
</tr>
<tr>
<td>March 2008</td>
<td>Eleventh printing</td>
<td>Minor revision for MATLAB 7.6 (Release 2008a)</td>
</tr>
<tr>
<td>October 2008</td>
<td>Twelfth printing</td>
<td>Minor revision for MATLAB 7.7 (Release 2008b)</td>
</tr>
<tr>
<td>March 2009</td>
<td>Thirteenth printing</td>
<td>Minor revision for MATLAB 7.8 (Release 2009a)</td>
</tr>
<tr>
<td>September 2009</td>
<td>Fourteenth printing</td>
<td>Minor revision for MATLAB 7.9 (Release 2009b)</td>
</tr>
<tr>
<td>March 2010</td>
<td>Fifteenth printing</td>
<td>Minor revision for MATLAB 7.10 (Release 2010a)</td>
</tr>
<tr>
<td>September 2010</td>
<td>Sixteenth printing</td>
<td>Revised for MATLAB 7.11 (R2010b)</td>
</tr>
<tr>
<td>April 2011</td>
<td>Online only</td>
<td>Revised for MATLAB 7.12 (R2011a)</td>
</tr>
<tr>
<td>September 2011</td>
<td>Seventeenth printing</td>
<td>Revised for MATLAB 7.13 (R2011b)</td>
</tr>
<tr>
<td>March 2012</td>
<td>Eighteenth printing</td>
<td>Revised for Version 7.14 (R2012a) (Renamed from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MATLAB Getting Started Guide)</td>
</tr>
<tr>
<td>September 2012</td>
<td>Nineteenth printing</td>
<td>Revised for Version 8.0 (R2012b)</td>
</tr>
<tr>
<td>March 2013</td>
<td>Twentieth printing</td>
<td>Revised for Version 8.1 (R2013a)</td>
</tr>
<tr>
<td>September 2013</td>
<td>Twenty-first printing</td>
<td>Revised for Version 8.2 (R2013b)</td>
</tr>
<tr>
<td>March 2014</td>
<td>Twenty-second printing</td>
<td>Revised for Version 8.3 (R2014a)</td>
</tr>
</tbody>
</table>
# Language Fundamentals

## Matrices and Magic Squares
- About Matrices ....................................... 2-2
- Entering Matrices .................................... 2-4
- sum, transpose, and diag ............................ 2-5
- The magic Function .................................. 2-7
- Generating Matrices ................................ 2-8

## Expressions
- Variables ........................................ 2-10
- Numbers ........................................ 2-11
- Matrix Operators .................................. 2-12
- Array Operators .................................. 2-12
- Functions ........................................ 2-14
- Examples of Expressions ............................ 2-16

## Entering Commands
- The format Function ................................ 2-17
- Suppressing Output ................................ 2-18
- Entering Long Statements ......................... 2-19
- Command Line Editing ............................... 2-19

## Indexing
- Subscripts ....................................... 2-20
- The Colon Operator ................................ 2-21
- Concatenation .................................... 2-22
- Deleting Rows and Columns ...................... 2-23
- Scalar Expansion .................................. 2-24
- Logical Subscripting .............................. 2-25
- The find Function .................................. 2-26

## Types of Arrays
- Multidimensional Arrays ......................... 2-27
- Cell Arrays ...................................... 2-29
- Characters and Text .............................. 2-31
- Structures ...................................... 2-34
Mathematics

3

Linear Algebra ............................................. 3-2
  Matrices in the MATLAB Environment ............... 3-2
  Systems of Linear Equations .......................... 3-11
  Inverses and Determinants ........................... 3-23
  Factorizations .......................................... 3-27
  Powers and Exponentials .............................. 3-35
  Eigenvalues ............................................ 3-39
  Singular Values ....................................... 3-43

Operations on Nonlinear Functions .................... 3-46
  Function Handles ..................................... 3-46
  Function Functions .................................... 3-46

Multivariate Data ....................................... 3-49

Data Analysis ........................................... 3-50
  Introduction .......................................... 3-50
  Preprocessing Data .................................... 3-50
  Summarizing Data ..................................... 3-58
  Visualizing Data ..................................... 3-63
  Modeling Data ........................................ 3-77

Graphics

4

Basic Plotting Functions ................................. 4-2
  Creating a Plot ........................................ 4-2
  Plotting Multiple Data Sets in One Graph .......... 4-3
  Specifying Line Styles and Colors ................... 4-5
  Plotting Lines and Markers ........................... 4-7
  Graphing Imaginary and Complex Data ................ 4-8
  Adding Plots to an Existing Graph ................. 4-9
  Figure Windows ....................................... 4-11
  Displaying Multiple Plots in One Figure ........... 4-11
  Controlling the Axes .................................. 4-12
Quick Start

- “MATLAB Product Description” on page 1-2
- “Desktop Basics” on page 1-3
- “Matrices and Arrays” on page 1-6
- “Array Indexing” on page 1-11
- “Workspace Variables” on page 1-13
- “Character Strings” on page 1-15
- “Calling Functions” on page 1-17
- “2-D and 3-D Plots” on page 1-18
- “Programming and Scripts” on page 1-26
- “Help and Documentation” on page 1-30
MATLAB Product Description

The Language of Technical Computing

MATLAB® is a high-level language and interactive environment for numerical computation, visualization, and programming. Using MATLAB, you can analyze data, develop algorithms, and create models and applications. The language, tools, and built-in math functions enable you to explore multiple approaches and reach a solution faster than with spreadsheets or traditional programming languages, such as C/C++ or Java®. You can use MATLAB for a range of applications, including signal processing and communications, image and video processing, control systems, test and measurement, computational finance, and computational biology. More than a million engineers and scientists in industry and academia use MATLAB, the language of technical computing.

Key Features

- High-level language for numerical computation, visualization, and application development
- Interactive environment for iterative exploration, design, and problem solving
- Mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration, and solving ordinary differential equations
- Built-in graphics for visualizing data and tools for creating custom plots
- Development tools for improving code quality and maintainability and maximizing performance
- Tools for building applications with custom graphical interfaces
- Functions for integrating MATLAB based algorithms with external applications and languages such as C, Java, .NET, and Microsoft® Excel®
Desktop Basics

When you start MATLAB, the desktop appears in its default layout.

The desktop includes these panels:

- **Current Folder** — Access your files.
- **Command Window** — Enter commands at the command line, indicated by the prompt (>).
- **Workspace** — Explore data that you create or import from files.

As you work in MATLAB, you issue commands that create variables and call functions. For example, create a variable named `a` by typing this statement at the command line:
a = 1

MATLAB adds variable a to the workspace and displays the result in the Command Window.

a =

1

Create a few more variables.

b = 2

b =

2

C = a + b

c =

3

d = cos(a)

d =

0.5403

When you do not specify an output variable, MATLAB uses the variable ans, short for answer, to store the results of your calculation.

sin(a)

ans =

0.8415

If you end a statement with a semicolon, MATLAB performs the computation, but suppresses the display of output in the Command Window.

e = a*b;
You can recall previous commands by pressing the up- and down-arrow keys, ↑ and ↓. Press the arrow keys either at an empty command line or after you type the first few characters of a command. For example, to recall the command \( b = 2 \), type \( b \), and then press the up-arrow key.
**Matrices and Arrays**

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Array Creation” on page 1-6</td>
</tr>
<tr>
<td>“Matrix and Array Operations” on page 1-7</td>
</tr>
<tr>
<td>“Concatenation” on page 1-9</td>
</tr>
<tr>
<td>“Complex Numbers” on page 1-10</td>
</tr>
</tbody>
</table>

*MATLAB* is an abbreviation for "matrix laboratory." While other programming languages mostly work with numbers one at a time, MATLAB is designed to operate primarily on whole matrices and arrays.

All MATLAB variables are multidimensional *arrays*, no matter what type of data. A *matrix* is a two-dimensional array often used for linear algebra.

**Array Creation**

To create an array with four elements in a single row, separate the elements with either a comma (,) or a space.

\[ a = [1 \ 2 \ 3 \ 4] \]

returns

\[ a = \\
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array} \]

This type of array is a *row vector*.

To create a matrix that has multiple rows, separate the rows with semicolons.

\[ a = [1 \ 2 \ 3; \ 4 \ 5 \ 6; \ 7 \ 8 \ 10] \]

\[ a = \\
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 10 \\
\end{array} \]
Another way to create a matrix is to use a function, such as `ones`, `zeros`, or `rand`. For example, create a 5-by-1 column vector of zeros.

```matlab
z = zeros(5,1)
```

```
z =
0
0
0
0
0
```

**Matrix and Array Operations**

MATLAB allows you to process all of the values in a matrix using a single arithmetic operator or function.

```matlab
a + 10
```

```
ans =
11 12 13
14 15 16
17 18 20
```

```matlab
sin(a)
```

```
ans =
0.8415 0.9093 0.1411
-0.7568 -0.9589 -0.2794
0.6570 0.9894 -0.5440
```

To transpose a matrix, use a single quote ('):

```matlab
a'
```

```
ans =
1 4 7
2 5 8
```
You can perform standard matrix multiplication, which computes the inner products between rows and columns, using the * operator. For example, confirm that a matrix times its inverse returns the identity matrix:

\[
p = a \cdot \text{inv}(a)
\]

\[
p =
\begin{bmatrix}
1.0000 & 0 & -0.0000 \\
0 & 1.0000 & 0 \\
0 & 0 & 1.0000
\end{bmatrix}
\]

Notice that \(p\) is not a matrix of integer values. MATLAB stores numbers as floating-point values, and arithmetic operations are sensitive to small differences between the actual value and its floating-point representation. You can display more decimal digits using the \text{format} command:

\[
\text{format long}
p = a \cdot \text{inv}(a)
\]

\[
p =
\begin{bmatrix}
1.000000000000000 & 0 & -0.000000000000000 \\
0 & 1.000000000000000 & 0 \\
0 & 0 & 0.999999999999998
\end{bmatrix}
\]

Reset the display to the shorter format using

\[
\text{format short}
\]

\text{format} affects only the display of numbers, not the way MATLAB computes or saves them.

To perform element-wise multiplication rather than matrix multiplication, use the .* operator:

\[
p = a \cdot * a
\]

\[
p =
\begin{bmatrix}
-8 & 1
\end{bmatrix}
\]
The matrix operators for multiplication, division, and power each have a corresponding array operator that operates element-wise. For example, raise each element of \( a \) to the third power:

\[
a .^\text{3}
\]

\[
\text{ans} =
\begin{bmatrix}
1 & 8 & 27 \\
64 & 125 & 216 \\
343 & 512 & 1000
\end{bmatrix}
\]

**Concatenation**

Concatenation is the process of joining arrays to make larger ones. In fact, you made your first array by concatenating its individual elements. The pair of square brackets \([\ ]\) is the concatenation operator.

\[
A = [a,a]
\]

\[
A =
\begin{bmatrix}
1 & 2 & 3 & 1 & 2 & 3 \\
4 & 5 & 6 & 4 & 5 & 6 \\
7 & 8 & 10 & 7 & 8 & 10
\end{bmatrix}
\]

Concatenating arrays next to one another using commas is called horizontal concatenation. Each array must have the same number of rows. Similarly, when the arrays have the same number of columns, you can concatenate vertically using semicolons.

\[
A = [a; a]
\]
A =

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

**Complex Numbers**

Complex numbers have both real and imaginary parts, where the imaginary unit is the square root of $-1$.

$$\sqrt{-1}$$

ans =

$$0.0000 + 1.0000i$$

To represent the imaginary part of complex numbers, use either $i$ or $j$.

c = [3+4i, 4+3j; -i, 10j]

c =

$$\begin{bmatrix}
3.0000 + 4.0000i & 4.0000 + 3.0000i \\
0.0000 - 1.0000i & 0.0000 + 10.0000i
\end{bmatrix}$$
Array Indexing

Every variable in MATLAB is an array that can hold many numbers. When you want to access selected elements of an array, use indexing.

For example, consider the 4-by-4 magic square A:

\[
A = \text{magic}(4)
\]

\[
A = \\
16 & 2 & 3 & 13 \\
5 & 11 & 10 & 8 \\
9 & 7 & 6 & 12 \\
4 & 14 & 15 & 1 \\
\]

There are two ways to refer to a particular element in an array. The most common way is to specify row and column subscripts, such as

\[
A(4,2)
\]

\[
\text{ans} = \\
14 \\
\]

Less common, but sometimes useful, is to use a single subscript that traverses down each column in order:

\[
A(8)
\]

\[
\text{ans} = \\
14 \\
\]

Using a single subscript to refer to a particular element in an array is called linear indexing.

If you try to refer to elements outside an array on the right side of an assignment statement, MATLAB throws an error.

\[
test = A(4,5)
\]

Attempted to access A(4,5); index out of bounds because size(A)=[4,4].
However, on the left side of an assignment statement, you can specify elements outside the current dimensions. The size of the array increases to accommodate the newcomers.

\[ A(4,5) = 17 \]

\[
A =
\begin{bmatrix}
16 & 2 & 3 & 13 & 0 \\
5 & 11 & 10 & 8 & 0 \\
9 & 7 & 6 & 12 & 0 \\
4 & 14 & 15 & 1 & 17
\end{bmatrix}
\]

To refer to multiple elements of an array, use the colon operator, which allows you to specify a range of the form \texttt{start:end}. For example, list the elements in the first three rows and the second column of \( A \):

\[ A(1:3,2) \]

\[
\text{ans} =
\begin{bmatrix}
2 \\
11 \\
7
\end{bmatrix}
\]

The colon alone, without start or end values, specifies all of the elements in that dimension. For example, select all the columns in the third row of \( A \):

\[ A(3,:) \]

\[
\text{ans} =
\begin{bmatrix}
9 \\
7 \\
6 \\
12 \\
0
\end{bmatrix}
\]

The colon operator also allows you to create an equally spaced vector of values using the more general form \texttt{start:step:end}.

\[ B = 0:10:100 \]

\[
B =
\begin{bmatrix}
0 & 10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 & 90 & 100
\end{bmatrix}
\]

If you omit the middle \texttt{step}, as in \texttt{start:end}, MATLAB uses the default step value of 1.
Workspace Variables

The *workspace* contains variables that you create within or import into MATLAB from data files or other programs. For example, these statements create variables A and B in the workspace.

\[
A = \text{magic}(4); \\
B = \text{rand}(3,5,2);
\]

You can view the contents of the workspace using `whos`.

```
whos
```

```
Name      Size      Bytes  Class      Attributes
A         4x4       128    double
B         3x5x2    240    double
```

The variables also appear in the Workspace pane on the desktop.

Workspace variables do not persist after you exit MATLAB. Save your data for later use with the `save` command,

```
save myfile.mat
```

Saving preserves the workspace in your current working folder in a compressed file with a `.mat` extension, called a MAT-file.

To clear all the variables from the workspace, use the `clear` command.

Restore data from a MAT-file into the workspace using `load`. 
load myfile.mat
Character Strings

A character string is a sequence of any number of characters enclosed in single quotes. You can assign a string to a variable.

```matlab
myText = 'Hello, world';
```

If the text includes a single quote, use two single quotes within the definition.

```matlab
otherText = 'You''re right'
```

```matlab
otherText =
You're right
```

myText and otherText are arrays, like all MATLAB variables. Their class or data type is char, which is short for character.

```matlab
whos myText
```

```
Name      Size      Bytes    Class    Attributes
myText    1x12      24       char
```

You can concatenate strings with square brackets, just as you concatenate numeric arrays.

```matlab
longText = [myText,' - ',otherText]
```

```matlab
longText =
Hello, world - You're right
```

To convert numeric values to strings, use functions, such as num2str or int2str.

```matlab
f = 71;
c = (f-32)/1.8;
tempText = ['Temperature is ',num2str(c),'C']
```

```matlab
tempText =
```

1-15
Temperature is 21.6667°C
Calling Functions

MATLAB provides a large number of functions that perform computational tasks. Functions are equivalent to subroutines or methods in other programming languages.

To call a function, such as max, enclose its input arguments in parentheses:

\[
\text{A} = \begin{bmatrix} 1 & 3 & 5 \end{bmatrix};
\]
\[
\text{max(A)};
\]

If there are multiple input arguments, separate them with commas:

\[
\text{B} = \begin{bmatrix} 10 & 6 & 4 \end{bmatrix};
\]
\[
\text{max(A,B)};
\]

Return output from a function by assigning it to a variable:

\[
\text{maxA} = \text{max(A)};
\]

When there are multiple output arguments, enclose them in square brackets:

\[
[\text{maxA},\text{location}] = \text{max(A)};
\]

Enclose any character string inputs in single quotes:

\[
\text{disp('hello world')};
\]

To call a function that does not require any inputs and does not return any outputs, type only the function name:

\[
\text{clc}
\]

The clc function clears the Command Window.
2-D and 3-D Plots

In this section...

| “Line Plots” on page 1-18 |
| “3-D Plots” on page 1-23 |
| “Subplots” on page 1-24 |

Line Plots
To create two-dimensional line plots, use the plot function. For example, plot the value of the sine function from 0 to $2\pi$:

```matlab
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```
You can label the axes and add a title.

```matlab
xlabel('x')
ylabel('sin(x)')
title('Plot of the Sine Function')
```
By adding a third input argument to the `plot` function, you can plot the same variables using a red dashed line.

```
plot(x,y,'r--')
```
The 'r--' string is a line specification. Each specification can include characters for the line color, style, and marker. A marker is a symbol that appears at each plotted data point, such as a +, o, or *. For example, 'g:*' requests a dotted green line with * markers.

Notice that the titles and labels that you defined for the first plot are no longer in the current figure window. By default, MATLAB® clears the figure each time you call a plotting function, resetting the axes and other elements to prepare the new plot.
To add plots to an existing figure, use `hold`.

```matlab
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)

hold on

y2 = cos(x);
plot(x,y2,'r: ')
legend('sin','cos')
```
Until you use `hold off` or close the window, all plots appear in the current figure window.

### 3-D Plots

Three-dimensional plots typically display a surface defined by a function in two variables, $z = f(x,y)$.

To evaluate $z$, first create a set of $(x,y)$ points over the domain of the function using `meshgrid`.

```matlab
[X,Y] = meshgrid(-2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
```

Then, create a surface plot.

```matlab
surf(X,Y,Z)
```
Both the `surf` function and its companion `mesh` display surfaces in three dimensions. `surf` displays both the connecting lines and the faces of the surface in color. `mesh` produces wireframe surfaces that color only the lines connecting the defining points.

**Subplots**

You can display multiple plots in different subregions of the same window using the `subplot` function.
For example, create four plots in a 2-by-2 grid within a figure window.

```matlab
t = 0:pi/10:2*pi;
[X,Y,Z] = cylinder(4*cos(t));
subplot(2,2,1); mesh(X); title('X');
subplot(2,2,2); mesh(Y); title('Y');
subplot(2,2,3); mesh(Z); title('Z');
subplot(2,2,4); mesh(X,Y,Z); title('X,Y,Z');
```

The first two inputs to the `subplot` function indicate the number of plots in each row and column. The third input specifies which plot is active.
Programming and Scripts

In this section...

“Sample Script” on page 1-26
“Loops and Conditional Statements” on page 1-27
“Script Locations” on page 1-29

The simplest type of MATLAB program is called a *script*. A script is a file with a .m extension that contains multiple sequential lines of MATLAB commands and function calls. You can run a script by typing its name at the command line.

**Sample Script**

To create a script, use the `edit` command,

```matlab
edit plotrand
```

This opens a blank file named `plotrand.m`. Enter some code that plots a vector of random data:

```matlab
n = 50;
r = rand(n,1);
plot(r)
```

Next, add code that draws a horizontal line on the plot at the mean:

```matlab
m = mean(r);
hold on
plot([0,n],[m,m])
hold off
title('Mean of Random Uniform Data')
```

Whenever you write code, it is a good practice to add comments that describe the code. Comments allow others to understand your code, and can refresh your memory when you return to it later. Add comments using the percent (%) symbol.

```matlab
% Generate random data from a uniform distribution
```
% and calculate the mean. Plot the data and the mean.

n = 50;                 % 50 data points
r = rand(n,1);
plot(r)

% Draw a line from (0,m) to (n,m)
m = mean(r);
hold on
plot([0,n],[m,m])
hold off
title('Mean of Random Uniform Data')

Save the file in the current folder. To run the script, type its name at the
command line:

plotrand

You can also run scripts from the Editor by pressing the Run button.

Loops and Conditional Statements
Within a script, you can loop over sections of code and conditionally execute
sections using the keywords for, while, if, and switch.

For example, create a script named calcmean.m that uses a for loop to
calculate the mean of five random samples and the overall mean.

nsamples = 5;
npoints = 50;

for k = 1:nsamples
    currentData = rand(npoints,1);
    sampleMean(k) = mean(currentData);
end
overallMean = mean(sampleMean)

Now, modify the for loop so that you can view the results at each iteration.
Display text in the Command Window that includes the current iteration
number, and remove the semicolon from the assignment to sampleMean.
for k = 1:nsamples
    iterationString = ['Iteration #',int2str(k)];
    disp(iterationString)
    currentData = rand(npoints,1);
    sampleMean(k) = mean(currentData)
end
overallMean = mean(sampleMean)

When you run the script, it displays the intermediate results, and then calculates the overall mean.

calcmean

Iteration #1
sampleMean =
    0.3988

Iteration #2
sampleMean =
    0.3988  0.4950

Iteration #3
sampleMean =
    0.3988  0.4950  0.5365

Iteration #4
sampleMean =
    0.3988  0.4950  0.5365  0.4870

Iteration #5
sampleMean =
In the Editor, add conditional statements to the end of calcmean.m that display a different message depending on the value of overallMean.

```matlab
if overallMean < .49
    disp('Mean is less than expected')
elseif overallMean > .51
    disp('Mean is greater than expected')
else
    disp('Mean is within the expected range')
end
```

Run calcmean and verify that the correct message displays for the calculated overallMean. For example:

```matlab
overallMean =

0.5178
```

Mean is greater than expected

**Script Locations**
MATLAB looks for scripts and other files in certain places. To run a script, the file must be in the current folder or in a folder on the search path.

By default, the MATLAB folder that the MATLAB Installer creates is on the search path. If you want to store and run programs in another folder, add it to the search path. Select the folder in the Current Folder browser, right-click, and then select **Add to Path**.
Help and Documentation

All MATLAB functions have supporting documentation that includes examples and describes the function inputs, outputs, and calling syntax. There are several ways to access this information from the command line:

- Open the function documentation in a separate window using the `doc` command.

```
doc mean
```

- Display function hints (the syntax portion of the function documentation) in the Command Window by pausing after you type the open parentheses for the function input arguments.

```
mean(
```

- View an abbreviated text version of the function documentation in the Command Window using the `help` command.

```
help mean
```

Access the complete product documentation by clicking the help icon 🎨.
Language Fundamentals

- “Matrices and Magic Squares” on page 2-2
- “Expressions” on page 2-10
- “Entering Commands” on page 2-17
- “Indexing” on page 2-20
- “Types of Arrays” on page 2-27
Matrices and Magic Squares

In this section...

“About Matrices” on page 2-2
“Entering Matrices” on page 2-4
“sum, transpose, and diag” on page 2-5
“The magic Function” on page 2-7
“Generating Matrices” on page 2-8

About Matrices

In the MATLAB environment, a matrix is a rectangular array of numbers. Special meaning is sometimes attached to 1-by-1 matrices, which are scalars, and to matrices with only one row or column, which are vectors. MATLAB has other ways of storing both numeric and nonnumeric data, but in the beginning, it is usually best to think of everything as a matrix. The operations in MATLAB are designed to be as natural as possible. Where other programming languages work with numbers one at a time, MATLAB allows you to work with entire matrices quickly and easily. A good example matrix, used throughout this book, appears in the Renaissance engraving Melencolia I by the German artist and amateur mathematician Albrecht Dürer.
This image is filled with mathematical symbolism, and if you look carefully, you will see a matrix in the upper-right corner. This matrix is known as a magic square and was believed by many in Dürer’s time to have genuinely magical properties. It does turn out to have some fascinating characteristics worth exploring.
Entering Matrices

The best way for you to get started with MATLAB is to learn how to handle matrices. Start MATLAB and follow along with each example.

You can enter matrices into MATLAB in several different ways:

- Enter an explicit list of elements.
- Load matrices from external data files.
- Generate matrices using built-in functions.
- Create matrices with your own functions and save them in files.

Start by entering Dürer’s matrix as a list of its elements. You only have to follow a few basic conventions:

- Separate the elements of a row with blanks or commas.
- Use a semicolon, ;, to indicate the end of each row.
- Surround the entire list of elements with square brackets, [  ].

To enter Dürer’s matrix, simply type in the Command Window

\[ A = [16 \ 3 \ 2 \ 13; \ 5 \ 10 \ 11 \ 8; \ 9 \ 6 \ 7 \ 12; \ 4 \ 15 \ 14 \ 1] \]
MATLAB displays the matrix you just entered:

\[
A = \\
\begin{pmatrix}
16 & 3 & 2 & 13 \\
5 & 10 & 11 & 8 \\
9 & 6 & 7 & 12 \\
4 & 15 & 14 & 1 \\
\end{pmatrix}
\]

This matrix matches the numbers in the engraving. Once you have entered the matrix, it is automatically remembered in the MATLAB workspace. You can refer to it simply as \( A \). Now that you have \( A \) in the workspace, take a look at what makes it so interesting. Why is it magic?

**sum, transpose, and diag**

You are probably already aware that the special properties of a magic square have to do with the various ways of summing its elements. If you take the sum along any row or column, or along either of the two main diagonals, you will always get the same number. Let us verify that using MATLAB. The first statement to try is

\[
\text{sum}(A)
\]

MATLAB replies with

\[
\text{ans} = \\
\begin{pmatrix}
34 & 34 & 34 & 34 \\
\end{pmatrix}
\]

When you do not specify an output variable, MATLAB uses the variable \( \text{ans} \), short for *answer*, to store the results of a calculation. You have computed a row vector containing the sums of the columns of \( A \). Each of the columns has the same sum, the *magic* sum, 34.

How about the row sums? MATLAB has a preference for working with the columns of a matrix, so one way to get the row sums is to transpose the matrix, compute the column sums of the transpose, and then transpose the result.

MATLAB has two transpose operators. The apostrophe operator (for example, \( A' \)) performs a complex conjugate transposition. It flips a matrix about its main diagonal, and also changes the sign of the imaginary component of any complex elements of the matrix. The dot-apostrophe operator (\( A.' \)),
transposes without affecting the sign of complex elements. For matrices containing all real elements, the two operators return the same result.

So

A'

produces

\[
\begin{array}{cccc}
16 & 5 & 9 & 4 \\
3 & 10 & 6 & 15 \\
2 & 11 & 7 & 14 \\
13 & 8 & 12 & 1 \\
\end{array}
\]

and

\[
\text{sum}(A')'
\]

produces a column vector containing the row sums

\[
\begin{array}{c}
34 \\
34 \\
34 \\
34 \\
34 \\
34 \\
\end{array}
\]

For an additional way to sum the rows that avoids the double transpose use the dimension argument for the \texttt{sum} function:

\[
\text{sum}(A,2)
\]

produces

\[
\begin{array}{c}
34 \\
34 \\
34 \\
34 \\
34 \\
34 \\
\end{array}
\]

The sum of the elements on the main diagonal is obtained with the \texttt{sum} and the \texttt{diag} functions:
diag(A)

produces

ans =
 16
 10
  7
  1

and

sum(diag(A))

produces

ans =
  34

The other diagonal, the so-called antidiagonal, is not so important mathematically, so MATLAB does not have a ready-made function for it. But a function originally intended for use in graphics, fliplr, flips a matrix from left to right:

sum(diag(fliplr(A)))
ans =
  34

You have verified that the matrix in Dürer's engraving is indeed a magic square and, in the process, have sampled a few MATLAB matrix operations. The following sections continue to use this matrix to illustrate additional MATLAB capabilities.

**The magic Function**

MATLAB actually has a built-in function that creates magic squares of almost any size. Not surprisingly, this function is named magic:
B = magic(4)
B =
    16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1

This matrix is almost the same as the one in the Dürer engraving and has all the same “magic” properties; the only difference is that the two middle columns are exchanged.

To make this B into Dürer’s A, swap the two middle columns:

A = B(:, [1 3 2 4])

This subscript indicates that—for each of the rows of matrix B—reorder the elements in the order 1, 3, 2, 4. It produces:

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

**Generating Matrices**

MATLAB software provides four functions that generate basic matrices.

- **zeros**: All zeros
- **ones**: All ones
- **rand**: Uniformly distributed random elements
- **randn**: Normally distributed random elements

Here are some examples:

Z = zeros(2,4)
Z =
    0    0    0    0
    0    0    0    0
F = 5*ones(3,3)
F =
    5   5   5
    5   5   5
    5   5   5

N = fix(10*rand(1,10))
N =
    9   2   6   4   8   7   4   0   8   4

R = randn(4,4)
R =
   0.6353  0.0860 -0.3210 -1.2316
  -0.6014 -2.0046  1.2366  1.0556
   0.5512 -0.4931 -0.6313 -0.1132
  -1.0998  0.4620 -2.3252  0.3792
Expressions

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Variables” on page 2-10</td>
</tr>
<tr>
<td>“Numbers” on page 2-11</td>
</tr>
<tr>
<td>“Matrix Operators” on page 2-12</td>
</tr>
<tr>
<td>“Array Operators” on page 2-12</td>
</tr>
<tr>
<td>“Functions” on page 2-14</td>
</tr>
<tr>
<td>“Examples of Expressions” on page 2-16</td>
</tr>
</tbody>
</table>

Variables

Like most other programming languages, the MATLAB language provides mathematical expressions, but unlike most programming languages, these expressions involve entire matrices.

MATLAB does not require any type declarations or dimension statements. When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. If the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage. For example,

```matlab
num_students = 25
```

creates a 1-by-1 matrix named `num_students` and stores the value 25 in its single element. To view the matrix assigned to any variable, simply enter the variable name.

Variable names consist of a letter, followed by any number of letters, digits, or underscores. MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. `A` and `a` are not the same variable.

Although variable names can be of any length, MATLAB uses only the first \(N\) characters of the name, (where \(N\) is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first \(N\) characters to enable MATLAB to distinguish variables.
\[ N = \text{namelengthmax} \]
\[ N = 63 \]

**Numbers**

MATLAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. Scientific notation uses the letter \( e \) to specify a power-of-ten scale factor. Imaginary numbers use either \( i \) or \( j \) as a suffix. Some examples of legal numbers are:

\[
\begin{align*}
3 & \quad -99 & \quad 0.0001 \\
9.6397238 & \quad 1.60210e-20 & \quad 6.02252e23 \\
1i & \quad -3.14159j & \quad 3e5i
\end{align*}
\]

MATLAB stores all numbers internally using the *long* format specified by the IEEE® floating-point standard. Floating-point numbers have a finite *precision* of roughly 16 significant decimal digits and a finite *range* of roughly \( 10^{-308} \) to \( 10^{+308} \).

Numbers represented in the double format have a maximum precision of 52 bits. Any double requiring more bits than 52 loses some precision. For example, the following code shows two unequal values to be equal because they are both truncated:

\[
\begin{align*}
\text{x} &= 36028797018963968; \\
\text{y} &= 36028797018963972; \\
\text{x} &== \text{y} \\
\text{ans} &= \\
&= 1
\end{align*}
\]

Integers have available precisions of 8-bit, 16-bit, 32-bit, and 64-bit. Storing the same numbers as 64-bit integers preserves precision:

\[
\begin{align*}
\text{x} &= \text{uint64}(36028797018963968); \\
\text{y} &= \text{uint64}(36028797018963972); \\
\text{x} &== \text{y} \\
\text{ans} &= \\
&= 0
\end{align*}
\]

MATLAB software stores the real and imaginary parts of a complex number. It handles the magnitude of the parts in different ways depending on the
context. For instance, the \texttt{sort} function sorts based on magnitude and resolves ties by phase angle.

\begin{verbatim}
sort([3+4i, 4+3i])
ans =
  4.0000 + 3.0000i  3.0000 + 4.0000i
\end{verbatim}

This is because of the phase angle:

\begin{verbatim}
angle(3+4i)
ans =
  0.9273
angle(4+3i)
ans =
  0.6435
\end{verbatim}

The “equal to” relational operator \texttt{==} requires both the real and imaginary parts to be equal. The other binary relational operators \texttt{>, <, >=, and <=} ignore the imaginary part of the number and consider the real part only.

\section*{Matrix Operators}

Expressions use familiar arithmetic operators and precedence rules.

\begin{verbatim}
+   Addition
-   Subtraction
*   Multiplication
/   Division
\   Left division
^   Power
'   Complex conjugate transpose
( ) Specify evaluation order
\end{verbatim}

\section*{Array Operators}

When they are taken away from the world of linear algebra, matrices become two-dimensional numeric arrays. Arithmetic operations on arrays are
done element by element. This means that addition and subtraction are the same for arrays and matrices, but that multiplicative operations are different. MATLAB uses a dot, or decimal point, as part of the notation for multiplicative array operations.

The list of operators includes

+ Addition
- Subtraction
.* Element-by-element multiplication
./ Element-by-element division
.ackslash Element-by-element left division
.^ Element-by-element power
.' Unconjugated array transpose

If the Dürer magic square is multiplied by itself with array multiplication

A.*A

the result is an array containing the squares of the integers from 1 to 16, in an unusual order:

ans =
  256    9    4   169
  25  100  121    64
  81    36    49  144
  16  225  196    1

**Building Tables**

Array operations are useful for building tables. Suppose \( n \) is the column vector

\[
\begin{bmatrix}
(0:9)'
\end{bmatrix}
\]

Then

\[
\text{pows} = [n \ n.^2 \ 2.^n]
\]
builds a table of squares and powers of 2:

\[
\text{pows} =
\begin{array}{ccc}
0 & 0 & 1 \\
1 & 1 & 2 \\
2 & 4 & 4 \\
3 & 9 & 8 \\
4 & 16 & 16 \\
5 & 25 & 32 \\
6 & 36 & 64 \\
7 & 49 & 128 \\
8 & 64 & 256 \\
9 & 81 & 512 \\
\end{array}
\]

The elementary math functions operate on arrays element by element. So

```matlab
format short g
x = (1:0.1:2)’;
logs = [x log10(x)]
```

builds a table of logarithms.

\[
\text{logs} =
\begin{array}{ccc}
1.0 & 0 \\
1.1 & 0.04139 \\
1.2 & 0.07918 \\
1.3 & 0.11394 \\
1.4 & 0.14613 \\
1.5 & 0.17609 \\
1.6 & 0.20412 \\
1.7 & 0.23045 \\
1.8 & 0.25527 \\
1.9 & 0.27875 \\
2.0 & 0.30103 \\
\end{array}
\]

**Functions**

MATLAB provides a large number of standard elementary mathematical functions, including \texttt{abs}, \texttt{sqrt}, \texttt{exp}, and \texttt{sin}. Taking the square root or logarithm of a negative number is not an error; the appropriate complex result is produced automatically. MATLAB also provides many more advanced mathematical functions, including Bessel and gamma functions. Most of
these functions accept complex arguments. For a list of the elementary mathematical functions, type

```
help elfun
```

For a list of more advanced mathematical and matrix functions, type

```
help specfun
help elmat
```

Some of the functions, like \texttt{sqrt} and \texttt{sin}, are \textit{built in}. Built-in functions are part of the MATLAB core so they are very efficient, but the computational details are not readily accessible. Other functions are implemented in the MATLAB programing language, so their computational details are accessible.

There are some differences between built-in functions and other functions. For example, for built-in functions, you cannot see the code. For other functions, you can see the code and even modify it if you want.

Several special functions provide values of useful constants.

- \texttt{pi} \hspace{1cm} 3.14159265...
- \texttt{i} \hspace{1cm} Imaginary unit, \(\sqrt{-1}\)
- \texttt{j} \hspace{1cm} Same as \texttt{i}
- \texttt{eps} \hspace{1cm} Floating-point relative precision, \(\varepsilon = 2^{-52}\)
- \texttt{realmin} \hspace{1cm} Smallest floating-point number, \(2^{-1022}\)
- \texttt{realmax} \hspace{1cm} Largest floating-point number, \((2 - \varepsilon)2^{1023}\)
- \texttt{Inf} \hspace{1cm} Infinity
- \texttt{NaN} \hspace{1cm} Not-a-number

Infinity is generated by dividing a nonzero value by zero, or by evaluating well defined mathematical expressions that \textit{overflow}, that is, exceed \texttt{realmax}. Not-a-number is generated by trying to evaluate expressions like 0/0 or \texttt{Inf-Inf} that do not have well defined mathematical values.
The function names are not reserved. It is possible to overwrite any of them with a new variable, such as

```matlab
eps = 1.e-6
```

and then use that value in subsequent calculations. The original function can be restored with

```matlab
clear eps
```

### Examples of Expressions

You have already seen several examples of MATLAB expressions. Here are a few more examples, and the resulting values:

```matlab
rho = (1+sqrt(5))/2
rho =
   1.6180

a = abs(3+4i)
a =
   5

z = sqrt(besselk(4/3,rho-i))
z =
   0.3730 + 0.3214i

huge = exp(log(realmax))
huge =
   1.7977e+308

toobig = pi*huge
toobig =
   Inf
```
The format Function
The format function controls the numeric format of the values displayed. The function affects only how numbers are displayed, not how MATLAB software computes or saves them. Here are the different formats, together with the resulting output produced from a vector x with components of different magnitudes.

Note To ensure proper spacing, use a fixed-width font, such as Courier.

```matlab
x = [4/3 1.2345e-6]
format short
  1.3333   0.0000
format short e
  1.3333e+000  1.2345e-006
format short g
  1.3333  1.2345e-006
format long
  1.33333333333333  0.000000123450000
```
format long e
  1.333333333333333e+000  1.234500000000000e-006

format long g
  1.33333333333333  1.2345e-006

format bank
  1.33  0.00

format rat
  4/3  1/810045

format hex
  3ff5555555555555  3eb4b6231abfd271

If the largest element of a matrix is larger than $10^3$ or smaller than $10^{-3}$, MATLAB applies a common scale factor for the short and long formats.

In addition to the `format` functions shown above

format compact

suppresses many of the blank lines that appear in the output. This lets you view more information on a screen or window. If you want more control over the output format, use the `sprintf` and `fprintf` functions.

**Suppressing Output**

If you simply type a statement and press Return or Enter, MATLAB automatically displays the results on screen. However, if you end the line with a semicolon, MATLAB performs the computation, but does not display any output. This is particularly useful when you generate large matrices. For example,

A = magic(100);
**Entering Long Statements**

If a statement does not fit on one line, use an ellipsis (three periods), ..., followed by **Return** or **Enter** to indicate that the statement continues on the next line. For example,

\[
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7 \ldots \\
   - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
\]

Blank spaces around the =, +, and - signs are optional, but they improve readability.

**Command Line Editing**

Various arrow and control keys on your keyboard allow you to recall, edit, and reuse statements you have typed earlier. For example, suppose you mistakenly enter

\[\rho = (1 + \sqrt{5})/2\]

You have misspelled \(\sqrt{5}\). MATLAB responds with

*Undefined function 'sqt' for input arguments of type 'double'.*

Instead of retyping the entire line, simply press the ↑ key. The statement you typed is redisplayed. Use the ← key to move the cursor over and insert the missing \(r\). Repeated use of the ↑ key recalls earlier lines. Typing a few characters, and then pressing the ↑ key finds a previous line that begins with those characters. You can also copy previously executed statements from the Command History.
Subscripts
The element in row \( i \) and column \( j \) of \( A \) is denoted by \( A(i, j) \). For example, \( A(4, 2) \) is the number in the fourth row and second column. For the magic square, \( A(4, 2) \) is 15. So to compute the sum of the elements in the fourth column of \( A \), type

\[
A(1,4) + A(2,4) + A(3,4) + A(4,4)
\]

This subscript produces

\[
\text{ans} = 34
\]

but is not the most elegant way of summing a single column.

It is also possible to refer to the elements of a matrix with a single subscript, \( A(k) \). A single subscript is the usual way of referencing row and column vectors. However, it can also apply to a fully two-dimensional matrix, in which case the array is regarded as one long column vector formed from the columns of the original matrix. So, for the magic square, \( A(8) \) is another way of referring to the value 15 stored in \( A(4,2) \).

If you try to use the value of an element outside of the matrix, it is an error:

\[
t = A(4,5)
\]
Index exceeds matrix dimensions.

Conversely, if you store a value in an element outside of the matrix, the size increases to accommodate the newcomer:

```matlab
X = A;
X(4,5) = 17
```

```
X =
    16  3  2  13  0
    5 10 11  8  0
    9  6  7 12  0
    4 15 14  1 17
```

**The Colon Operator**

The colon, :, is one of the most important MATLAB operators. It occurs in several different forms. The expression

```matlab
1:10
```

is a row vector containing the integers from 1 to 10:

```
1   2   3   4   5   6   7   8   9  10
```

To obtain nonunit spacing, specify an increment. For example,

```matlab
100:-7:50
```

is

```
100  93  86  79  72  65  58  51
```

and

```matlab
0:pi/4:pi
```

is

```
0  0.7854  1.5708  2.3562  3.1416
```

Subscript expressions involving colons refer to portions of a matrix:

```matlab
A(1:k,j)
```
is the first $k$ elements of the $j$th column of $A$. Thus,

$$\text{sum}(A(1:4,4))$$

computes the sum of the fourth column. However, there is a better way to perform this computation. The colon by itself refers to all the elements in a row or column of a matrix and the keyword `end` refers to the last row or column. Thus,

$$\text{sum}(A(:, \text{end}))$$

computes the sum of the elements in the last column of $A$:

$$\text{ans} = 34$$

Why is the magic sum for a 4-by-4 square equal to 34? If the integers from 1 to 16 are sorted into four groups with equal sums, that sum must be

$$\text{sum}(1:16)/4$$

which, of course, is

$$\text{ans} = 34$$

**Concatenation**

*Concatenation* is the process of joining small matrices to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, `[]`, is the concatenation operator. For an example, start with the 4-by-4 magic square, $A$, and form

$$B = [A \ A+32; A+48 \ A+16]$$

The result is an 8-by-8 matrix, obtained by joining the four submatrices:
This matrix is halfway to being another magic square. Its elements are a rearrangement of the integers 1:64. Its column sums are the correct value for an 8-by-8 magic square:

\[
\text{sum}(B)
\]

\[
\text{ans} = 
\begin{bmatrix} 
\end{bmatrix}
\]

But its row sums, \(\text{sum}(B')\)', are not all the same. Further manipulation is necessary to make this a valid 8-by-8 magic square.

**Deleting Rows and Columns**

You can delete rows and columns from a matrix using just a pair of square brackets. Start with

\[
X = A;
\]

Then, to delete the second column of \(X\), use

\[
X(:,2) = []
\]

This changes \(X\) to

\[
X = 
\begin{bmatrix} 
16 & 2 & 13 \\
5 & 11 & 8 \\
9 & 7 & 12 \\
4 & 14 & 1 
\end{bmatrix}
\]
If you delete a single element from a matrix, the result is not a matrix anymore. So, expressions like

\[ X(1,2) = [] \]

result in an error. However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a row vector. So

\[ X(2:2:10) = [] \]

results in

\[ X = \begin{bmatrix} 16 & 9 & 2 & 7 & 13 & 12 & 1 \end{bmatrix} \]

**Scalar Expansion**

Matrices and scalars can be combined in several different ways. For example, a scalar is subtracted from a matrix by subtracting it from each element. The average value of the elements in our magic square is 8.5, so

\[ B = A - 8.5 \]

forms a matrix whose column sums are zero:

\[ B = \begin{bmatrix} 7.5 & -5.5 & -6.5 & 4.5 \\ -3.5 & 1.5 & 2.5 & -0.5 \\ 0.5 & -2.5 & -1.5 & 3.5 \\ -4.5 & 6.5 & 5.5 & -7.5 \end{bmatrix} \]

\[ \text{sum}(B) \]

\[ \text{ans} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \]

With scalar expansion, MATLAB assigns a specified scalar to all indices in a range. For example,

\[ B(1:2,2:3) = 0 \]

zeros out a portion of \( B \):
Logical Subscripting

The logical vectors created from logical and relational operations can be used to reference subarrays. Suppose \(X\) is an ordinary matrix and \(L\) is a matrix of the same size that is the result of some logical operation. Then \(X(L)\) specifies the elements of \(X\) where the elements of \(L\) are nonzero.

This kind of subscripting can be done in one step by specifying the logical operation as the subscripting expression. Suppose you have the following set of data:

\[
x = [2.1 1.7 1.6 1.5 \text{ NaN} 1.9 1.8 1.5 5.1 1.8 1.4 2.2 1.6 1.8];
\]

The NaN is a marker for a missing observation, such as a failure to respond to an item on a questionnaire. To remove the missing data with logical indexing, use \text{isfinite}(x)\), which is true for all finite numerical values and false for NaN and Inf:

\[
x = x(\text{isfinite}(x))
\]

\[
x =
2.1 1.7 1.6 1.5 1.9 1.8 1.5 5.1 1.8 1.4 2.2 1.6 1.8
\]

Now there is one observation, 5.1, which seems to be very different from the others. It is an outlier. The following statement removes outliers, in this case those elements more than three standard deviations from the mean:

\[
x = x(\text{abs}(x-\text{mean}(x)) <= 3*\text{std}(x))
\]

\[
x =
2.1 1.7 1.6 1.5 1.9 1.8 1.5 1.8 1.4 2.2 1.6 1.8
\]

For another example, highlight the location of the prime numbers in Dürer’s magic square by using logical indexing and scalar expansion to set the nonprimes to 0. (See “The magic Function” on page 2-7.)

\[
A(\text{~isprime}(A)) = 0
\]
The `find` Function

The `find` function determines the indices of array elements that meet a given logical condition. In its simplest form, `find` returns a column vector of indices. Transpose that vector to obtain a row vector of indices. For example, start again with Dürer’s magic square. (See “The magic Function” on page 2-7.)

```
k = find(isprime(A))'
```

picks out the locations, using one-dimensional indexing, of the primes in the magic square:

```
k =
   2   5   9  10  11  13
```

Display those primes, as a row vector in the order determined by `k`, with `A(k)`

```
an =
   5   3   2  11   7  13
```

When you use `k` as a left-side index in an assignment statement, the matrix structure is preserved:

```
A(k) = NaN
```

```
A =
   16   NaN   NaN   NaN
   NaN   10   NaN   8
   9    6   NaN  12
   4   15  14    1
```
Types of Arrays

In this section...

“Multidimensional Arrays” on page 2-27
“Cell Arrays” on page 2-29
“Characters and Text” on page 2-31
“Structures” on page 2-34

Multidimensional Arrays
Multidimensional arrays in the MATLAB environment are arrays with more than two subscripts. One way of creating a multidimensional array is by calling zeros, ones, rand, or randn with more than two arguments. For example,

\[ R = \text{randn}(3,4,5); \]

creates a 3-by-4-by-5 array with a total of \(3 \times 4 \times 5 = 60\) normally distributed random elements.

A three-dimensional array might represent three-dimensional physical data, say the temperature in a room, sampled on a rectangular grid. Or it might represent a sequence of matrices, \(A^{(k)}\), or samples of a time-dependent matrix, \(A(t)\). In these latter cases, the \((i,j)\)th element of the \(k\)th matrix, or the \(t_k\)th matrix, is denoted by \(A(i,j,k)\).

MATLAB and Dürer’s versions of the magic square of order 4 differ by an interchange of two columns. Many different magic squares can be generated by interchanging columns. The statement

\[ p = \text{perms}(1:4); \]

generates the \(4! = 24\) permutations of \(1:4\). The \(k\)th permutation is the row vector \(p(k,:).\) Then

\[ A = \text{magic}(4); \]
\[ M = \text{zeros}(4,4,24); \]
for k = 1:24
    M(:,:,k) = A(:,p(k,:));
end

stores the sequence of 24 magic squares in a three-dimensional array, M. The size of M is

size(M)

ans =
    4   4   24

**Note** The order of the matrices shown in this illustration might differ from your results. The `perms` function always returns all permutations of the input vector, but the order of the permutations might be different for different MATLAB versions.

The statement

```
sum(M,d)
```

computes sums by varying the dth subscript. So

```
sum(M,1)
```
is a 1-by-4-by-24 array containing 24 copies of the row vector

\[
34\quad 34\quad 34\quad 34
\]

and

\[
\text{sum}(M,2)
\]

is a 4-by-1-by-24 array containing 24 copies of the column vector

\[
34 \\
34 \\
34 \\
34
\]

Finally,

\[
S = \text{sum}(M,3)
\]

adds the 24 matrices in the sequence. The result has size 4-by-4-by-1, so it looks like a 4-by-4 array:

\[
S =
\begin{bmatrix}
204 & 204 & 204 & 204 \\
204 & 204 & 204 & 204 \\
204 & 204 & 204 & 204 \\
204 & 204 & 204 & 204 \\
\end{bmatrix}
\]

**Cell Arrays**

Cell arrays in MATLAB are multidimensional arrays whose elements are copies of other arrays. A cell array of empty matrices can be created with the `cell` function. But, more often, cell arrays are created by enclosing a miscellaneous collection of things in curly braces, `{}`. The curly braces are also used with subscripts to access the contents of various cells. For example,

\[
C = \{A \text{ sum}(A) \prod(\prod(A))\}
\]

produces a 1-by-3 cell array. The three cells contain the magic square, the row vector of column sums, and the product of all its elements. When \(C\) is displayed, you see

\[
C = 
\]
This is because the first two cells are too large to print in this limited space, but the third cell contains only a single number, $16!$, so there is room to print it.

Here are two important points to remember. First, to retrieve the contents of one of the cells, use subscripts in curly braces. For example, $C\{1\}$ retrieves the magic square and $C\{3\}$ is $16!$. Second, cell arrays contain copies of other arrays, not pointers to those arrays. If you subsequently change $A$, nothing happens to $C$.

You can use three-dimensional arrays to store a sequence of matrices of the same size. Cell arrays can be used to store a sequence of matrices of different sizes. For example,

```matlab
M = cell(8,1);
for n = 1:8
    M{n} = magic(n);
end
M
```

produces a sequence of magic squares of different order:

```matlab
M =
[ 1]
[ 2x2 double]
[ 3x3 double]
[ 4x4 double]
[ 5x5 double]
[ 6x6 double]
[ 7x7 double]
[ 8x8 double]
```
You can retrieve the 4-by-4 magic square matrix with

\[ M\{4\} \]

**Characters and Text**

Enter text into MATLAB using single quotes. For example,

\[
s = 'Hello'
\]

The result is not the same kind of numeric matrix or array you have been dealing with up to now. It is a 1-by-5 character array.
Internally, the characters are stored as numbers, but not in floating-point format. The statement

\[ a = \text{double}(s) \]

converts the character array to a numeric matrix containing floating-point representations of the ASCII codes for each character. The result is

\[ a = \begin{bmatrix} 72 & 101 & 108 & 108 & 111 \end{bmatrix} \]

The statement

\[ s = \text{char}(a) \]

reverses the conversion.

Converting numbers to characters makes it possible to investigate the various fonts available on your computer. The printable characters in the basic ASCII character set are represented by the integers 32:127. (The integers less than 32 represent nonprintable control characters.) These integers are arranged in an appropriate 6-by-16 array with

\[ F = \text{reshape}(32:127,16,6)'; \]

The printable characters in the extended ASCII character set are represented by \( F+128 \). When these integers are interpreted as characters, the result depends on the font currently being used. Type the statements

\[ \text{char}(F) \]
\[ \text{char}(F+128) \]

and then vary the font being used for the Command Window. To change the font, on the Home tab, in the Environment section, click Preferences > Fonts. If you include tabs in lines of code, use a fixed-width font, such as Monospaced, to align the tab positions on different lines.

Concatenation with square brackets joins text variables together into larger strings. The statement

\[ h = [s, ' \text{world}'] \]
joins the strings horizontally and produces

\[ h = \]
Hello world

The statement

\[ v = [s; 'world'] \]

joins the strings vertically and produces

\[ v = \]
Hello
world

Notice that a blank has to be inserted before the 'w' in h and that both words in v have to have the same length. The resulting arrays are both character arrays; h is 1-by-11 and v is 2-by-5.

To manipulate a body of text containing lines of different lengths, you have two choices—a padded character array or a cell array of strings. When creating a character array, you must make each row of the array the same length. (Pad the ends of the shorter rows with spaces.) The char function does this padding for you. For example,

\[ S = \text{char}('A','rolling','stone','gathers','momentum.') \]

produces a 5-by-9 character array:

\[ S = \]
A
rolling
stone
gathers
momentum.

Alternatively, you can store the text in a cell array. For example,

\[ C = {'A';'rolling';'stone';'gathers';'momentum.'} \]

creates a 5-by-1 cell array that requires no padding because each row of the array can have a different length:
C =
    'A'
    'rolling'
    'stone'
    'gathers'
    'momentum.'

You can convert a padded character array to a cell array of strings with

C = cellstr(S)

and reverse the process with

S = char(C)

**Structures**

Structures are multidimensional MATLAB arrays with elements accessed by textual *field designators*. For example,

S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'

creates a scalar structure with three fields:

S =
    name: 'Ed Plum'
    score: 83
    grade: 'B+'

Like everything else in the MATLAB environment, structures are arrays, so you can insert additional elements. In this case, each element of the array is a structure with several fields. The fields can be added one at a time,

S(2).name = 'Toni Miller';
S(2).score = 91;
S(2).grade = 'A-';

or an entire element can be added with a single statement:

S(3) = struct('name','Jerry Garcia',
                'score',70,'grade','C')
Now the structure is large enough that only a summary is printed:

```
S =
1x3 struct array with fields:
   name
   score
   grade
```

There are several ways to reassemble the various fields into other MATLAB arrays. They are mostly based on the notation of a *comma-separated list*. If you type

```
S.score
```

it is the same as typing

```
S(1).score, S(2).score, S(3).score
```

which is a comma-separated list.

If you enclose the expression that generates such a list within square brackets, MATLAB stores each item from the list in an array. In this example, MATLAB creates a numeric row vector containing the `score` field of each element of structure array `S`:

```
scores = [S.score]
scores =
     83   91   70
```

```
avg_score = sum(scores)/length(scores)
avg_score =
    81.3333
```

To create a character array from one of the text fields (`name`, for example), call the `char` function on the comma-separated list produced by `S.name`:

```
names = char(S.name)
names =
   Ed Plum
   Toni Miller
   Jerry Garcia
```
Similarly, you can create a cell array from the name fields by enclosing the list-generating expression within curly braces:

```matlab
nenames = {S.name}
names =
    'Ed Plum'    'Toni Miller'    'Jerry Garcia'
```

To assign the fields of each element of a structure array to separate variables outside of the structure, specify each output to the left of the equals sign, enclosing them all within square brackets:

```matlab
[N1 N2 N3] = S.name
N1 =
    Ed Plum
N2 =
    Toni Miller
N3 =
    Jerry Garcia
```

### Dynamic Field Names

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run time. The dot-parentheses syntax shown here makes `expression` a dynamic field name:

```matlab
structName.(expression)
```

Index into this field using the standard MATLAB indexing syntax. For example, to evaluate `expression` into a field name and obtain the values of that field at columns 1 through 25 of row 7, use

```matlab
structName.(expression)(7,1:25)
```

#### Dynamic Field Names Example

The `avgscore` function shown below computes an average test score, retrieving information from the `testscores` structure using dynamic field names:

```matlab
function avg = avgscore(testscores, student, first, last)
    for k = first:last
        scores(k) = testscores.(student).week(k);
    end
```

end
avg = sum(scores)/(last - first + 1);

You can run this function using different values for the dynamic field student. First, initialize the structure that contains scores for a 25-week period:

testscores.Ann_Lane.week(1:25) = ...
    [95 89 76 82 79 92 94 92 89 81 75 93 ...
    85 84 83 86 85 90 82 82 84 79 96 88 98];

testscores.William_King.week(1:25) = ...
    [87 80 91 84 99 87 93 87 97 87 82 89 ...
    86 82 90 98 75 79 92 84 90 93 84 78 81];

Now run avgscore, supplying the students name fields for the testscores structure at run time using dynamic field names:

avgscore(testscores, 'Ann_Lane', 7, 22)
an =
    85.2500

avgscore(testscores, 'William_King', 7, 22)
an =
    87.7500
Mathematics

- “Linear Algebra” on page 3-2
- “Operations on Nonlinear Functions” on page 3-46
- “Multivariate Data” on page 3-49
- “Data Analysis” on page 3-50
Linear Algebra

In this section...

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Matrices in the MATLAB Environment”</td>
<td>3-2</td>
</tr>
<tr>
<td>“Systems of Linear Equations”</td>
<td>3-11</td>
</tr>
<tr>
<td>“Inverses and Determinants”</td>
<td>3-23</td>
</tr>
<tr>
<td>“Factorizations”</td>
<td>3-27</td>
</tr>
<tr>
<td>“Powers and Exponentials”</td>
<td>3-35</td>
</tr>
<tr>
<td>“Eigenvalues”</td>
<td>3-39</td>
</tr>
<tr>
<td>“Singular Values”</td>
<td>3-43</td>
</tr>
</tbody>
</table>

Matrices in the MATLAB Environment

- “Creating Matrices” on page 3-2
- “Adding and Subtracting Matrices” on page 3-4
- “Vector Products and Transpose” on page 3-5
- “Multiplying Matrices” on page 3-7
- “Identity Matrix” on page 3-9
- “Kronecker Tensor Product” on page 3-9
- “Vector and Matrix Norms” on page 3-10
- “Using Multithreaded Computation with Linear Algebra Functions” on page 3-11

Creating Matrices

The MATLAB environment uses the term *matrix* to indicate a variable containing real or complex numbers arranged in a two-dimensional grid. An *array* is, more generally, a vector, matrix, or higher dimensional grid of numbers. All arrays in MATLAB are rectangular, in the sense that the component vectors along any dimension are all the same length.
Symbolic Math Toolbox™ software extends the capabilities of MATLAB software to matrices of mathematical expressions.

MATLAB has dozens of functions that create different kinds of matrices. There are two functions you can use to create a pair of 3-by-3 example matrices for use throughout this chapter. The first example is symmetric:

\[ A = \text{pascal}(3) \]

\[
A =
\begin{bmatrix}
1 & 1 & 1 \\
1 & 2 & 3 \\
1 & 3 & 6
\end{bmatrix}
\]

The second example is not symmetric:

\[ B = \text{magic}(3) \]

\[
B =
\begin{bmatrix}
8 & 1 & 6 \\
3 & 5 & 7 \\
4 & 9 & 2
\end{bmatrix}
\]

Another example is a 3-by-2 rectangular matrix of random integers:

\[ C = \text{fix}(10 \times \text{rand}(3, 2)) \]

\[
C =
\begin{bmatrix}
9 & 4 \\
2 & 8 \\
6 & 7
\end{bmatrix}
\]

A column vector is an \( m \)-by-1 matrix, a row vector is a 1-by-\( n \) matrix, and a scalar is a 1-by-1 matrix. The statements

\[
u = [3; \ 1; \ 4]\]

\[v = [2 \ 0 \ -1]\]

\[s = 7\]
produce a column vector, a row vector, and a scalar:

\[
\begin{align*}
    \mathbf{u} &= \\
    \begin{bmatrix}
        3 \\
        1 \\
        4
    \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
    \mathbf{v} &= \\
    \begin{bmatrix}
        2 \\
        0 \\
        -1
    \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
    \mathbf{s} &= \\
    7
\end{align*}
\]

**Adding and Subtracting Matrices**

Addition and subtraction of matrices is defined just as it is for arrays, element by element. Adding \( \mathbf{A} \) to \( \mathbf{B} \), and then subtracting \( \mathbf{A} \) from the result recovers \( \mathbf{B} \):

\[
\begin{align*}
    \mathbf{A} &= \text{pascal}(3); \\
    \mathbf{B} &= \text{magic}(3); \\
    \mathbf{X} &= \mathbf{A} + \mathbf{B}
\end{align*}
\]

\[
\begin{align*}
    \mathbf{X} &= \\
    \begin{bmatrix}
        9 & 2 & 7 \\
        4 & 7 & 10 \\
        5 & 12 & 8
    \end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
    \mathbf{Y} &= \mathbf{X} - \mathbf{A}
\end{align*}
\]

\[
\begin{align*}
    \mathbf{Y} &= \\
    \begin{bmatrix}
        8 & 1 & 6 \\
        3 & 5 & 7 \\
        4 & 9 & 2
    \end{bmatrix}
\end{align*}
\]

Addition and subtraction require both matrices to have the same dimension, or one of them be a scalar. If the dimensions are incompatible, an error results:
\begin{verbatim}
C = fix(10*rand(3,2))
X = A + C
Error using plus
Matrix dimensions must agree.
w = v + s

w =
   9   7   6
\end{verbatim}

**Vector Products and Transpose**

A row vector and a column vector of the same length can be multiplied in either order. The result is either a scalar, the *inner* product, or a matrix, the *outer product* :

\[
\begin{align*}
  u &= [3; 1; 4] \\
v &= [2 0 -1] \\
x &= v \ast u
\end{align*}
\]

\[
\begin{array}{ccc}
  2 \\
\end{array}
\]

\[
X = u \ast v
\]

\[
\begin{array}{ccc}
  6 & 0 & -3 \\
  2 & 0 & -1 \\
  8 & 0 & -4 \\
\end{array}
\]

For real matrices, the *transpose* operation interchanges \( a_{ij} \) and \( a_{ji} \). MATLAB uses the apostrophe operator (’) to perform a complex conjugate transpose, and uses the dot-apostrophe operator (’.’) to transpose without conjugation. For matrices containing all real elements, the two operators return the same result.

The example matrix A is *symmetric*, so \( A' \) is equal to A. But, B is not symmetric:

\[
\begin{align*}
  B &= \text{magic}(3); \\
  X &= B'
\end{align*}
\]

\[
\begin{array}{ccc}
  X = \\
\end{array}
\]
Transposition turns a row vector into a column vector:

\[
x = v'
\]

\[
x = \\
2 \\
0 \\
-1
\]

If \(x\) and \(y\) are both real column vectors, the product \(x*y\) is not defined, but the two products

\[x'*y\]

and

\[y'*x\]

are the same scalar. This quantity is used so frequently, it has three different names: *inner* product, *scalar* product, or *dot* product.

For a complex vector or matrix, \(z\), the quantity \(z'\) not only transposes the vector or matrix, but also converts each complex element to its complex conjugate. That is, the sign of the imaginary part of each complex element changes. So if

\[
z = [1+2i \ 7-3i \ 3+4i; \ 6-2i \ 9i \ 4+7i]
\]

\[
z = \\
1.0000 + 2.0000i \\
7.0000 - 3.0000i \\
3.0000 + 4.0000i \\
6.0000 - 2.0000i \\
0 + 9.0000i \\
4.0000 + 7.0000i
\]

then

\[
z'\]

\[
ans = \\
1.0000 - 2.0000i \\
7.0000 + 3.0000i \\
3.0000 - 4.0000i \\
6.0000 + 2.0000i \\
0 - 9.0000i \\
4.0000 - 7.0000i
\]
The unconjugated complex transpose, where the complex part of each element retains its sign, is denoted by $z.'$:

$$z.'$$

ans =

$$\begin{bmatrix}
1.0000 + 2.0000i & 6.0000 - 2.0000i \\
7.0000 - 3.0000i & 0 + 9.0000i \\
3.0000 + 4.0000i & 4.0000 + 7.0000i
\end{bmatrix}$$

For complex vectors, the two scalar products $x'*y$ and $y'*x$ are complex conjugates of each other, and the scalar product $x'*x$ of a complex vector with itself is real.

**Multiplying Matrices**

Multiplication of matrices is defined in a way that reflects composition of the underlying linear transformations and allows compact representation of systems of simultaneous linear equations. The matrix product $C = AB$ is defined when the column dimension of $A$ is equal to the row dimension of $B$, or when one of them is a scalar. If $A$ is $m$-by-$p$ and $B$ is $p$-by-$n$, their product $C$ is $m$-by-$n$. The product can actually be defined using MATLAB for loops, colon notation, and vector dot products:

```matlab
A = pascal(3);
B = magic(3);
m = 3; n = 3;
for i = 1:m
    for j = 1:n
        C(i,j) = A(i,:)*B(:,j);
    end
end
```

MATLAB uses a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is not commutative; $AB$ is usually not equal to $BA$:

```
X = A*B
```

```
X =

15  15  15
26  38  26
```
$$ \begin{bmatrix} 41 & 70 & 39 \end{bmatrix} $$

$$ Y = B \cdot A $$

$$ Y = $$

$$ \begin{bmatrix} 15 & 28 & 47 \\
15 & 34 & 60 \\
15 & 28 & 43 \end{bmatrix} $$

A matrix can be multiplied on the right by a column vector and on the left by a row vector:

$$ u = \begin{bmatrix} 3 \\ 1 \\ 4 \end{bmatrix}; $$
$$ x = A \cdot u $$

$$ x = $$

$$ \begin{bmatrix} 8 \\
17 \\
30 \end{bmatrix} $$

$$ v = \begin{bmatrix} 2 & 0 & -1 \end{bmatrix}; $$
$$ y = v \cdot B $$

$$ y = $$

$$ \begin{bmatrix} 12 & -7 & 10 \end{bmatrix} $$

Rectangular matrix multiplications must satisfy the dimension compatibility conditions:

$$ C = \text{fix}(10 \cdot \text{rand}(3,2)); $$
$$ X = A \cdot C $$

$$ X = $$

$$ \begin{bmatrix} 17 & 19 \\
31 & 41 \\
51 & 70 \end{bmatrix} $$

$$ Y = C \cdot A $$

Error using mtimes
Inner matrix dimensions must agree.
Anything can be multiplied by a scalar:

\[
s = 7; \\
w = s \cdot v
\]

\[
w = \\
14 0 -7
\]

**Identity Matrix**

Generally accepted mathematical notation uses the capital letter \( I \) to denote identity matrices, matrices of various sizes with ones on the main diagonal and zeros elsewhere. These matrices have the property that \( AI = A \) and \( IA = A \) whenever the dimensions are compatible. The original version of MATLAB could not use \( I \) for this purpose because it did not distinguish between uppercase and lowercase letters and \( i \) already served as a subscript and as the complex unit. So an English language pun was introduced. The function

\[\text{eye}(m,n)\]

returns an \( m \)-by-\( n \) rectangular identity matrix and \( \text{eye}(n) \) returns an \( n \)-by-\( n \) square identity matrix.

**Kronecker Tensor Product**

The Kronecker product, \( \text{kron}(X,Y) \), of two matrices is the larger matrix formed from all possible products of the elements of \( X \) with those of \( Y \). If \( X \) is \( m \)-by-\( n \) and \( Y \) is \( p \)-by-\( q \), then \( \text{kron}(X,Y) \) is \( mp \)-by-\( nq \). The elements are arranged in the following order:

\[
\begin{bmatrix}
X(1,1) \cdot Y & X(1,2) \cdot Y & \ldots & X(1,n) \cdot Y \\
\vdots & \ddots & \ddots & \vdots \\
X(m,1) \cdot Y & X(m,2) \cdot Y & \ldots & X(m,n) \cdot Y
\end{bmatrix}
\]

The Kronecker product is often used with matrices of zeros and ones to build up repeated copies of small matrices. For example, if \( X \) is the 2-by-2 matrix

\[
X = \\
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

and \( I = \text{eye}(2,2) \) is the 2-by-2 identity matrix, then the two matrices
\[ \text{kron}(X,I) \]
and
\[ \text{kron}(I,X) \]
are
\[
\begin{bmatrix}
1 & 0 & 2 & 0 \\
0 & 1 & 0 & 2 \\
3 & 0 & 4 & 0 \\
0 & 3 & 0 & 4 \\
\end{bmatrix}
\]
and
\[
\begin{bmatrix}
1 & 2 & 0 & 0 \\
3 & 4 & 0 & 0 \\
0 & 0 & 1 & 2 \\
0 & 0 & 3 & 4 \\
\end{bmatrix}
\]

**Vector and Matrix Norms**

The \( p \)-norm of a vector \( x \),

\[
\|x\|_p = \left( \sum |x_i|^p \right)^{1/p},
\]

is computed by \( \text{norm}(x,p) \). This is defined by any value of \( p > 1 \), but the most common values of \( p \) are 1, 2, and \( \infty \). The default value is \( p = 2 \), which corresponds to *Euclidean length*:

\[
v = \begin{bmatrix} 2 & 0 & -1 \end{bmatrix};
\]

\[
[\text{norm}(v,1) \ \text{norm}(v) \ \text{norm}(v,\infty)]
\]

\[
\text{ans} =
\begin{bmatrix}
3.0000 & 2.2361 & 2.0000
\end{bmatrix}
\]

The \( p \)-norm of a matrix \( A \),

\[
\|A\|_p = \max_x \frac{\|Ax\|_p}{\|x\|_p},
\]
can be computed for \( p = 1, 2, \) and \( \infty \) by \( \text{norm}(A,p) \). Again, the default value is \( p = 2 \):

\[
C = \text{fix}(10\times\text{rand}(3,2));
\]

\[
[\text{norm}(C,1) \, \text{norm}(C) \, \text{norm}(C,\infty)]
\]

\[
\text{ans} = \\
19.0000 \quad 14.8015 \quad 13.0000
\]

**Using Multithreaded Computation with Linear Algebra Functions**

MATLAB software supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads. For a function or expression to execute faster on multiple CPUs, a number of conditions must be true:

1. The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.

2. The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains than several thousand elements or more.

3. The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complex functions speed up more than simple functions.

The matrix multiply \((X*Y)\) and matrix power \((X^p)\) operators show significant increase in speed on large double-precision arrays (on order of 10,000 elements). The matrix analysis functions \(\text{det}, \text{rcond}, \text{hess}, \) and \(\text{expm}\) also show significant increase in speed on large double-precision arrays.

**Systems of Linear Equations**

- “Computational Considerations” on page 3-12
Computational Considerations

One of the most important problems in technical computing is the solution of systems of simultaneous linear equations.

In matrix notation, the general problem takes the following form: Given two matrices $A$ and $b$, does there exist a unique matrix $x$, so that $Ax = b$ or $xA = b$?

It is instructive to consider a 1-by-1 example. For example, does the equation

$$7x = 21$$

have a unique solution?

The answer, of course, is yes. The equation has the unique solution $x = 3$. The solution is easily obtained by division:

$$x = \frac{21}{7} = 3.$$

The solution is not ordinarily obtained by computing the inverse of 7, that is $7^{-1} = 0.142857...$, and then multiplying $7^{-1}$ by 21. This would be more work and, if $7^{-1}$ is represented to a finite number of digits, less accurate. Similar considerations apply to sets of linear equations with more than one unknown; the MATLAB software solves such equations without computing the inverse of the matrix.

Although it is not standard mathematical notation, MATLAB uses the division terminology familiar in the scalar case to describe the solution of a general system of simultaneous equations. The two division symbols, $\text{slash}$, $/$, and $\text{backslash}$, $\backslash$, correspond to the two MATLAB functions $\text{mrdivide}$ and
\texttt{mldivide}, \texttt{mrdivide} and \texttt{mldivide} are used for the two situations where the unknown matrix appears on the left or right of the coefficient matrix:

\begin{align*}
\texttt{x = b/A} & \quad \text{Denotes the solution to the matrix equation } xA = b. \\
\texttt{x = A\backslash b} & \quad \text{Denotes the solution to the matrix equation } Ax = b.
\end{align*}

Think of “dividing” both sides of the equation $Ax = b$ or $xA = b$ by $A$. The coefficient matrix $A$ is always in the “denominator.”

The dimension compatibility conditions for $x = A\backslash b$ require the two matrices $A$ and $b$ to have the same number of rows. The solution $x$ then has the same number of columns as $b$ and its row dimension is equal to the column dimension of $A$. For $x = b/A$, the roles of rows and columns are interchanged.

In practice, linear equations of the form $Ax = b$ occur more frequently than those of the form $xA = b$. Consequently, the backslash is used far more frequently than the slash. The remainder of this section concentrates on the backslash operator; the corresponding properties of the slash operator can be inferred from the identity:

\[(b/A)' = (A'\backslash b').\]

The coefficient matrix $A$ need not be square. If $A$ is $m$-by-$n$, there are three cases:

\begin{align*}
m = n & \quad \text{Square system. Seek an exact solution.} \\
m > n & \quad \text{Overdetermined system. Find a least-squares solution.} \\
m < n & \quad \text{Underdetermined system. Find a basic solution with at most $m$ nonzero components.}
\end{align*}

**The \texttt{mldivide} Algorithm.** The \texttt{mldivide} operator employs different solvers to handle different kinds of coefficient matrices. The various cases are diagnosed automatically by examining the coefficient matrix. For more information, see the “Algorithms” section of the \texttt{mldivide} reference page.
**General Solution**

The general solution to a system of linear equations $Ax = b$ describes all possible solutions. You can find the general solution by:

1. Solving the corresponding homogeneous system $Ax = 0$. Do this using the `null` command, by typing `null(A)`. This returns a basis for the solution space to $Ax = 0$. Any solution is a linear combination of basis vectors.

2. Finding a particular solution to the nonhomogeneous system $Ax = b$.

You can then write any solution to $Ax = b$ as the sum of the particular solution to $Ax = b$, from step 2, plus a linear combination of the basis vectors from step 1.

The rest of this section describes how to use MATLAB to find a particular solution to $Ax = b$, as in step 2.

**Square Systems**

The most common situation involves a square coefficient matrix $A$ and a single right-side column vector $b$.

**Nonsingular Coefficient Matrix.** If the matrix $A$ is nonsingular, the solution, $x = A\backslash b$, is then the same size as $b$. For example:

```matlab
A = pascal(3);
u = [3; 1; 4];
x = A\u
```

```plaintext
x =
   10
  -12
    5
```

It can be confirmed that $A*x$ is exactly equal to $u$.

If $A$ and $b$ are square and the same size, $x = A\backslash b$ is also that size:

```matlab
b = magic(3);
X = A\b
```
It can be confirmed that $A\times x$ is exactly equal to $b$.

Both of these examples have exact, integer solutions. This is because the coefficient matrix was chosen to be $\text{pascal}(3)$, which is a full rank matrix (nonsingular).

**Singular Coefficient Matrix.** A square matrix $A$ is singular if it does not have linearly independent columns. If $A$ is singular, the solution to $Ax = b$ either does not exist, or is not unique. The backslash operator, $A\backslash b$, issues a warning if $A$ is nearly singular and raises an error condition if it detects exact singularity.

If $A$ is singular and $Ax = b$ has a solution, you can find a particular solution that is not unique, by typing

\[ P = \text{pinv}(A) \times b \]

$P$ is a pseudoinverse of $A$. If $Ax = b$ does not have an exact solution, $\text{pinv}(A)$ returns a least-squares solution.

For example:

\begin{align*}
A &= \begin{bmatrix}
1 & 3 & 7 \\
-1 & 4 & 4 \\
1 & 10 & 18
\end{bmatrix}
\end{align*}

is singular, as you can verify by typing

\[ \text{rank}(A) \]

\[ \text{ans} = 2 \]

Since $A$ is not full rank, it has some singular values equal to zero.
Note  For information about using pinv to solve systems with rectangular coefficient matrices, see “Pseudoinverses” on page 3-25.

Exact Solutions

For \( b = [5; 2; 12] \), the equation \( Ax = b \) has an exact solution, given by

\[
\text{pinv}(A)*b
\]

\[
\text{ans} = \\
0.3850 \\
-0.1103 \\
0.7066
\]

Verify that \( \text{pinv}(A)*b \) is an exact solution by typing

\[
A*\text{pinv}(A)*b
\]

\[
\text{ans} = \\
5.0000 \\
2.0000 \\
12.0000
\]

Least-Squares Solutions

However, if \( b = [3; 6; 0] \), \( Ax = b \) does not have an exact solution. In this case, \( \text{pinv}(A)*b \) returns a least-squares solution. If you type

\[
A*\text{pinv}(A)*b
\]

\[
\text{ans} = \\
-1.0000 \\
4.0000 \\
2.0000
\]

you do not get back the original vector \( b \).
You can determine whether $Ax = b$ has an exact solution by finding the row reduced echelon form of the augmented matrix $[A \ b]$. To do so for this example, enter

\[
\text{rref}([A \ b])
\]
\[
\text{ans} =
\begin{array}{cccc}
1.0000 & 0 & 2.2857 & 0 \\
0 & 1.0000 & 1.5714 & 0 \\
0 & 0 & 0 & 1.0000
\end{array}
\]

Since the bottom row contains all zeros except for the last entry, the equation does not have a solution. In this case, \text{pinv}(A) returns a least-squares solution.

**Overdetermined Systems**

This example shows how overdetermined systems are often encountered in various kinds of curve fitting to experimental data.

A quantity, $y$, is measured at several different values of time, $t$, to produce the following observations. You can enter the data and view it in a table with the following statements.

\[
t = [0 \ 0.3 \ 0.8 \ 1.1 \ 1.6 \ 2.3]';
\]
\[
y = [.82 \ 0.72 \ 0.63 \ 0.60 \ 0.55 \ 0.50]';
\]
\[
B = \text{table}(t,y)
\]

\[
B =
\begin{array}{ccc}
t & y \\
\hline
0 & 0.82 \\
0.3 & 0.72 \\
0.8 & 0.63 \\
1.1 & 0.6 \\
1.6 & 0.55 \\
2.3 & 0.5
\end{array}
\]
Try modeling the data with a decaying exponential function

\[ y(t) = c_1 + c_2 e^{-t}. \]

The preceding equation says that the vector \( y \) should be approximated by a linear combination of two other vectors. One is a constant vector containing all ones and the other is the vector with components \( \exp(-t) \). The unknown coefficients, \( c_1 \) and \( c_2 \), can be computed by doing a least-squares fit, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations in two unknowns, represented by a 6-by-2 matrix.

\[
E = \begin{bmatrix}
\text{ones(size(t))} & \exp(-t)
\end{bmatrix}
\]

\[
E =
\begin{bmatrix}
1.0000 & 1.0000 \\
1.0000 & 0.7408 \\
1.0000 & 0.4493 \\
1.0000 & 0.3329 \\
1.0000 & 0.2019 \\
1.0000 & 0.1003
\end{bmatrix}
\]

Use the backslash operator to get the least-squares solution.

\[
c = E\backslash y
\]

\[
c =
\begin{bmatrix}
0.4760 \\
0.3413
\end{bmatrix}
\]

In other words, the least-squares fit to the data is

\[ y(t) = 0.4760 + 0.3413e^{-t}. \]
The following statements evaluate the model at regularly spaced increments in \( t \), and then plot the result together with the original data:

\[
T = (0:0.1:2.5)';
Y = \left[ \text{ones(size}(T)) \ \exp(-T) \right] * c;
\text{plot}(T,Y,'-',[t,y,'o'])
\]

\( E \cdot c \) is not exactly equal to \( y \), but the difference might well be less than measurement errors in the original data.
A rectangular matrix $A$ is rank deficient if it does not have linearly independent columns. If $A$ is rank deficient, the least-squares solution to $AX = B$ is not unique. The backslash operator, $A\backslash B$, issues a warning if $A$ is rank deficient and produces a least-squares solution if the system has no solution and a basic solution if the system has infinitely many solutions.

**Underdetermined Systems**

This example shows how the solution to underdetermined systems is not unique. Underdetermined linear systems involve more unknowns than equations. The matrix left division operation in MATLAB finds a basic solution, which has at most $m$ nonzero components for an $m$-by-$n$ coefficient matrix.

Here is a small, random example:

```matlab
R = [6 8 7 3; 3 5 4 1];
rng(0);
b = randi(8,2,1);

R =

6   8   7   3
3   5   4   1

b =

7
8

The linear system $Rp = b$ involves two equations in four unknowns. Since the coefficient matrix contains small integers, it is appropriate to use the `format` command to display the solution in rational format. The particular solution is obtained with

```matlab
format rat
p = R\b
```

$p =$
One of the nonzero components is \( p(2) \) because \( R(:,2) \) is the column of \( R \) with largest norm. The other nonzero component is \( p(4) \) because \( R(:,4) \) dominates after \( R(:,2) \) is eliminated.

The complete general solution to the underdetermined system can be characterized by adding \( p \) to an arbitrary linear combination of the null space vectors, which can be found using the \texttt{null} function with an option requesting a rational basis.

\[
Z = \text{null}(R, 'r')
\]

\[
Z = \begin{bmatrix}
-1/2 & -7/6 \\
-1/2 & 1/2 \\
1 & 0 \\
0 & 1
\end{bmatrix}
\]

It can be confirmed that \( R*Z \) is zero and that the residual \( R*x - b \) is small for any vector \( x \), where

\[
x = p + Z*q .
\]

Since the columns of \( Z \) are the null space vectors, the product \( Z*q \) is a linear combination of those vectors:

\[
Z*q = \begin{pmatrix} \bar{x}_1 \\ \bar{x}_2 \end{pmatrix} \begin{pmatrix} u \\ w \end{pmatrix} = u\bar{x}_1 + w\bar{x}_2 .
\]

To illustrate, choose an arbitrary \( q \) and construct \( x \).

\[
q = [-2; 1] ;
\]

\[
x = p + Z*q ;
\]

Calculate the norm of the residual.
format short
norm(R*x - b)

ans =

2.6645e-15

Using Multithreaded Computation with Systems of Linear Equations

MATLAB software supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads. For a function or expression to execute faster on multiple CPUs, a number of conditions must be true:

1 The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.

2 The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains several thousand elements or more.

3 The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complicated functions speed up more than simple functions.

inv, lscov, linsolve, and mldivide show significant increase in speed on large double-precision arrays (on order of 10,000 elements or more) when multithreading is enabled.

Iterative Methods for Solving Systems of Linear Equations

If the coefficient matrix A is large and sparse, factorization methods are generally not efficient. Iterative methods generate a series of approximate solutions. MATLAB provides several iterative methods to handle large, sparse input matrices.
pcg
Preconditioned conjugate gradients method. This method is appropriate for Hermitian positive definite coefficient matrix A.

bicg
BiConjugate Gradients Method

bicgstab
BiConjugate Gradients Stabilized Method

bicgstabl
BiCGStab(l) Method

cgs
Conjugate Gradients Squared Method

gmres
Generalized Minimum Residual Method

lsqr
LSQR Method

minres
Minimum Residual Method. This method is appropriate for Hermitian coefficient matrix A.

qmr
Quasi-Minimal Residual Method

symmlq
Symmetric LQ Method

tfqmr
Transpose-Free QMR Method

Inverses and Determinants

• “Introduction” on page 3-24
• “Pseudoinverses” on page 3-25
### Introduction

If $A$ is square and nonsingular, the equations $AX = I$ and $XA = I$ have the same solution, $X$. This solution is called the inverse of $A$, is denoted by $A^{-1}$, and is computed by the function `inv`.

The determinant of a matrix is useful in theoretical considerations and some types of symbolic computation, but its scaling and round-off error properties make it far less satisfactory for numeric computation. Nevertheless, the function `det` computes the determinant of a square matrix:

```matlab
A = pascal(3)
A =
 1 1 1
 1 2 3
 1 3 6
d = det(A)
X = inv(A)
d =
 1
X =
 3 -3 1
-3 5 -2
1 -2 1
```

Again, because $A$ is symmetric, has integer elements, and has determinant equal to one, so does its inverse. However,

```matlab
B = magic(3)
B =
 8 1 6
 3 5 7
 4 9 2
d = det(B)
X = inv(B)
d =
```

```matlab
Closer examination of the elements of $X$, or use of format rat, would reveal that they are integers divided by 360.

If $A$ is square and nonsingular, then, without round-off error, $X = \text{inv}(A) * B$ is theoretically the same as $X = \text{A \backslash B}$ and $Y = B * \text{inv}(A)$ is theoretically the same as $Y = B / A$. But the computations involving the backslash and slash operators are preferable because they require less computer time, less memory, and have better error-detection properties.

**Pseudoinverses**

Rectangular matrices do not have inverses or determinants. At least one of the equations $AX = I$ and $XA = I$ does not have a solution. A partial replacement for the inverse is provided by the Moore-Penrose pseudoinverse, which is computed by the `pinv` function:

```matlab
format short
C = fix(10*gallery('uniformdata',[3 2],0));
X = pinv(C)

X =
    0.1159   -0.0729    0.0171
   -0.0534    0.1152    0.0418
```

The matrix

$$Q = X * C$$

$$Q =
\begin{bmatrix}
1.0000 & 0.0000 \\
0.0000 & 1.0000
\end{bmatrix}$$

is the 2-by-2 identity, but the matrix
\[ P = C \times X \]

\[
P = \\
\begin{bmatrix}
0.8293 & -0.1958 & 0.3213 \\
-0.1958 & 0.7754 & 0.3685 \\
0.3213 & 0.3685 & 0.3952 \\
\end{bmatrix}
\]

is not the 3-by-3 identity. However, \( P \) acts like an identity on a portion of the space in the sense that \( P \) is symmetric, \( P^*C \) is equal to \( C \), and \( X^*P \) is equal to \( X \).

**Solving a Rank-Deficient System.** If \( A \) is \( m \)-by-\( n \) with \( m > n \) and full rank \( n \), each of the three statements

\[
x = A \backslash b \\
x = \text{pinv}(A) \ast b \\
x = \text{inv}(A^\ast A) \ast A^\ast b
\]

theoretically computes the same least-squares solution \( x \), although the backslash operator does it faster.

However, if \( A \) does not have full rank, the solution to the least-squares problem is not unique. There are many vectors \( x \) that minimize \( \text{norm}(A \times x - b) \)

The solution computed by \( x = A \backslash b \) is a basic solution; it has at most \( r \) nonzero components, where \( r \) is the rank of \( A \). The solution computed by \( x = \text{pinv}(A) \ast b \) is the minimal norm solution because it minimizes \( \text{norm}(x) \). An attempt to compute a solution with \( x = \text{inv}(A^\ast A) \ast A^\ast b \) fails because \( A^\ast A \) is singular.

Here is an example that illustrates the various solutions:

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12 \\
\end{bmatrix};
\]

does not have full rank. Its second column is the average of the first and third columns. If

\[
b = A(:,2)
\]
is the second column, then an obvious solution to $A\times x = b$ is $x = [0 \ 1 \ 0]'$. But none of the approaches computes that $x$. The backslash operator gives

$$x = A\backslash b$$

Warning: Rank deficient, rank = 2, tol = 1.4594e-014.

$$x = 
\begin{bmatrix}
0.5000 \\
0 \\
0.5000
\end{bmatrix}$$

This solution has two nonzero components. The pseudoinverse approach gives

$$y = \text{pinv}(A)*b$$

$$y = 
\begin{bmatrix}
0.3333 \\
0.3333 \\
0.3333
\end{bmatrix}$$

There is no warning about rank deficiency. But $\text{norm}(y) = 0.5774$ is less than $\text{norm}(x) = 0.7071$. Finally,

$$z = \text{inv}(A'*A)*A'*b$$

fails completely:

Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 9.868649e-018.

$$z =
\begin{bmatrix}
-0.8594 \\
1.3438 \\
-0.6875
\end{bmatrix}$$

**Factorizations**

- “Introduction” on page 3-28
- “Cholesky Factorization” on page 3-28
- “LU Factorization” on page 3-29
- “QR Factorization” on page 3-31
Introduction
All three of the matrix factorizations discussed in this section make use of triangular matrices, where all the elements either above or below the diagonal are zero. Systems of linear equations involving triangular matrices are easily and quickly solved using either forward or back substitution.

Cholesky Factorization
The Cholesky factorization expresses a symmetric matrix as the product of a triangular matrix and its transpose

\[ A = R'R, \]

where \( R \) is an upper triangular matrix.

Not all symmetric matrices can be factored in this way; the matrices that have such a factorization are said to be positive definite. This implies that all the diagonal elements of \( A \) are positive and that the offdiagonal elements are “not too big.” The Pascal matrices provide an interesting example. Throughout this chapter, the example matrix \( A \) has been the 3-by-3 Pascal matrix. Temporarily switch to the 6-by-6:

\[ A = \text{pascal}(6) \]

\[
A = \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 3 & 6 & 10 & 15 & 21 \\
1 & 4 & 10 & 20 & 35 & 56 \\
1 & 5 & 15 & 35 & 70 & 126 \\
1 & 6 & 21 & 56 & 126 & 252
\]

The elements of \( A \) are binomial coefficients. Each element is the sum of its north and west neighbors. The Cholesky factorization is

\[ R = \text{chol}(A) \]

\[ R = \]
The elements are again binomial coefficients. The fact that $R^T \cdot R$ is equal to $A$ demonstrates an identity involving sums of products of binomial coefficients.

**Note**  The Cholesky factorization also applies to complex matrices. Any complex matrix that has a Cholesky factorization satisfies $A^T = A$ and is said to be *Hermitian positive definite*.

The Cholesky factorization allows the linear system

$$Ax = b$$

to be replaced by

$$R^T R x = b.$$ 

Because the backslash operator recognizes triangular systems, this can be solved in the MATLAB environment quickly with

$$x = R \backslash (R^T \backslash b)$$

If $A$ is $n$-by-$n$, the computational complexity of chol$(A)$ is $O(n^3)$, but the complexity of the subsequent backslash solutions is only $O(n^2)$.

**LU Factorization**

LU factorization, or Gaussian elimination, expresses any square matrix $A$ as the product of a permutation of a lower triangular matrix and an upper triangular matrix

$$A = LU,$$
where $L$ is a permutation of a lower triangular matrix with ones on its diagonal and $U$ is an upper triangular matrix.

The permutations are necessary for both theoretical and computational reasons. The matrix

$$
\begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}
$$

cannot be expressed as the product of triangular matrices without interchanging its two rows. Although the matrix

$$
\begin{bmatrix}
\varepsilon & 1 \\
1 & 0
\end{bmatrix}
$$

can be expressed as the product of triangular matrices, when $\varepsilon$ is small, the elements in the factors are large and magnify errors, so even though the permutations are not strictly necessary, they are desirable. Partial pivoting ensures that the elements of $L$ are bounded by one in magnitude and that the elements of $U$ are not much larger than those of $A$.

For example:

$$[L,U] = lu(B)$$

$L = 
\begin{bmatrix}
1.0000 & 0 & 0 \\
0.3750 & 0.5441 & 1.0000 \\
0.5000 & 1.0000 & 0
\end{bmatrix}$

$U = 
\begin{bmatrix}
8.0000 & 1.0000 & 6.0000 \\
0 & 8.5000 & -1.0000 \\
0 & 0 & 5.2941
\end{bmatrix}$

The LU factorization of $A$ allows the linear system

$$A*x = b$$
to be solved quickly with
\[ x = U \backslash (L \backslash b) \]

Determinants and inverses are computed from the LU factorization using
\[ \text{det}(A) = \text{det}(L) \times \text{det}(U) \]

and
\[ \text{inv}(A) = \text{inv}(U) \times \text{inv}(L) \]

You can also compute the determinants using \( \text{det}(A) = \text{prod} (\text{diag}(U)) \), though the signs of the determinants might be reversed.

**QR Factorization**

An *orthogonal* matrix, or a matrix with orthonormal columns, is a real matrix whose columns all have unit length and are perpendicular to each other. If \( Q \) is orthogonal, then
\[ Q'Q = 1. \]

The simplest orthogonal matrices are two-dimensional coordinate rotations:
\[
\begin{bmatrix}
  \cos(\theta) & \sin(\theta) \\
  -\sin(\theta) & \cos(\theta)
\end{bmatrix}.
\]

For complex matrices, the corresponding term is *unitary*. Orthogonal and unitary matrices are desirable for numerical computation because they preserve length, preserve angles, and do not magnify errors.

The orthogonal, or QR, factorization expresses any rectangular matrix as the product of an orthogonal or unitary matrix and an upper triangular matrix. A column permutation might also be involved:
\[ A = QR \]

or
\[ AP = QR, \]
where $Q$ is orthogonal or unitary, $R$ is upper triangular, and $P$ is a permutation.

There are four variants of the QR factorization—full or economy size, and with or without column permutation.

Overdetermined linear systems involve a rectangular matrix with more rows than columns, that is $m$-by-$n$ with $m > n$. The full-size QR factorization produces a square, $m$-by-$m$ orthogonal $Q$ and a rectangular $m$-by-$n$ upper triangular $R$:

\[ C = \text{gallery('uniformdata',[5 4], 0)}; \]
\[ [Q,R] = \text{qr}(C) \]

\[
Q = \\
\begin{bmatrix}
0.6191 & 0.1406 & -0.1899 & -0.5058 & 0.5522 \\
0.1506 & 0.4084 & 0.5034 & 0.5974 & 0.4475 \\
0.3954 & -0.5564 & 0.6869 & -0.1478 & -0.2008 \\
0.3167 & 0.6676 & 0.1351 & -0.1729 & -0.6370 \\
0.5808 & -0.2410 & -0.4695 & 0.5792 & -0.2207 \\
\end{bmatrix}
\]

\[
R = \\
\begin{bmatrix}
1.5346 & 1.0663 & 1.2010 & 1.4036 \\
0 & 0.7245 & 0.3474 & -0.0126 \\
0 & 0 & 0.9320 & 0.6596 \\
0 & 0 & 0 & 0.6648 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

In many cases, the last $m - n$ columns of $Q$ are not needed because they are multiplied by the zeros in the bottom portion of $R$. So the economy-size QR factorization produces a rectangular, $m$-by-$nQ$ with orthonormal columns and a square $n$-by-$n$ upper triangular $R$. For the 5-by-4 example, this is not much of a saving, but for larger, highly rectangular matrices, the savings in both time and memory can be quite important:

\[ [Q,R] = \text{qr}(C,0) \]
\[ Q = \]
In contrast to the LU factorization, the QR factorization does not require any pivoting or permutations. But an optional column permutation, triggered by the presence of a third output argument, is useful for detecting singularity or rank deficiency. At each step of the factorization, the column of the remaining unfactored matrix with largest norm is used as the basis for that step. This ensures that the diagonal elements of $R$ occur in decreasing order and that any linear dependence among the columns is almost certainly be revealed by examining these elements. For the small example given here, the second column of $C$ has a larger norm than the first, so the two columns are exchanged:

$$[Q, R, P] = qr(C)$$

$$Q =
\begin{bmatrix}
-0.3522 & 0.8398 & -0.4131 \\
-0.7044 & -0.5285 & -0.4739 \\
-0.6163 & 0.1241 & 0.7777 \\
\end{bmatrix}$$

$$R =
\begin{bmatrix}
-11.3578 & -8.2762 \\
0 & 7.2460 \\
0 & 0 \\
\end{bmatrix}$$

$$P =
\begin{bmatrix}
0 & 1 \\
1 & 0 \\
\end{bmatrix}$$
When the economy-size and column permutations are combined, the third output argument is a permutation vector, rather than a permutation matrix:

\[
[Q,R,p] = qr(C,0)
\]

\[
Q =
\begin{bmatrix}
-0.3522 & 0.8398 \\
-0.7044 & -0.5285 \\
-0.6163 & 0.1241
\end{bmatrix}
\]

\[
R =
\begin{bmatrix}
-11.3578 & -8.2762 \\
0 & 7.2460
\end{bmatrix}
\]

\[
p =
\begin{bmatrix}
2 \\
1
\end{bmatrix}
\]

The QR factorization transforms an overdetermined linear system into an equivalent triangular system. The expression

\[
\text{norm}(A\cdot x - b)
\]

equals

\[
\text{norm}(Q\cdot R\cdot x - b)
\]

Multiplication by orthogonal matrices preserves the Euclidean norm, so this expression is also equal to

\[
\text{norm}(R\cdot x - y)
\]

where \( y = Q'\cdot b \). Since the last \( m-n \) rows of \( R \) are zero, this expression breaks into two pieces:

\[
\text{norm}(R(1:n,1:n)\cdot x - y(1:n))
\]

and

\[
\text{norm}(y(n+1:m))
\]
When $A$ has full rank, it is possible to solve for $x$ so that the first of these expressions is zero. Then the second expression gives the norm of the residual. When $A$ does not have full rank, the triangular structure of $R$ makes it possible to find a basic solution to the least-squares problem.

**Using Multithreaded Computation for Factorization**

MATLAB software supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads. For a function or expression to execute faster on multiple CPUs, a number of conditions must be true:

1. The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.

2. The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains several thousand elements or more.

3. The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complex functions speed up more than simple functions.

$lu$ and $qr$ show significant increase in speed on large double-precision arrays (on order of 10,000 elements).

**Powers and Exponentials**

- “Positive Integer Powers” on page 3-36
- “Inverse and Fractional Powers” on page 3-36
- “Element-by-Element Powers” on page 3-36
- “Exponentials” on page 3-37
Positive Integer Powers
If $A$ is a square matrix and $p$ is a positive integer, $A^p$ effectively multiplies $A$ by itself $p-1$ times. For example:

$$A = \begin{bmatrix} 1 & 1 & 1; & 1 & 2 & 3; & 1 & 3 & 6 \end{bmatrix}$$

$$X = A^2$$

$$X = \begin{bmatrix} 3 & 6 & 10; & 6 & 14 & 25; & 10 & 25 & 46 \end{bmatrix}$$

Inverse and Fractional Powers
If $A$ is square and nonsingular, $A^{-p}$ effectively multiplies $\text{inv}(A)$ by itself $p-1$ times:

$$Y = A^{-3}$$

$$Y = \begin{bmatrix} 145.0000 & -207.0000 & 81.0000; & -207.0000 & 298.0000 & -117.0000; & 81.0000 & -117.0000 & 46.0000 \end{bmatrix}$$

Fractional powers, like $A^{(2/3)}$, are also permitted; the results depend upon the distribution of the eigenvalues of the matrix.

Element-by-Element Powers
The $\cdot^2$ operator produces element-by-element powers. For example:

$$X = A.\cdot^2$$
A =
\[
\begin{pmatrix}
1 & 1 & 1 \\
1 & 4 & 9 \\
1 & 9 & 36
\end{pmatrix}
\]

**Exponentials**

The function

\(\text{sqrtm}(A)\)

computes \(A^{(1/2)}\) by a more accurate algorithm. The \(m\) in \(\text{sqrtm}\) distinguishes this function from \(\text{sqrt}(A)\), which, like \(A.^(1/2)\), does its job element-by-element.

A system of linear, constant coefficient, ordinary differential equations can be written

\[
dx/dt = Ax,
\]

where \(x = x(t)\) is a vector of functions of \(t\) and \(A\) is a matrix independent of \(t\). The solution can be expressed in terms of the matrix exponential

\[
x(t) = e^{tA}x(0).
\]

The function

\(\text{expm}(A)\)

computes the matrix exponential. An example is provided by the 3-by-3 coefficient matrix,

\[
A = \begin{bmatrix}
0 & -6 & -1 \\
6 & 2 & -16 \\
-5 & 20 & -10
\end{bmatrix}
\]

\[
A =
\[
\begin{pmatrix}
0 & -6 & -1 \\
6 & 2 & -16 \\
-5 & 20 & -10
\end{pmatrix}
\]
and the initial condition, $x(0)$.

$$x_0 = [1 \ 1 \ 1]$$

The matrix exponential is used to compute the solution, $x(t)$, to the differential equation at 101 points on the interval $0 \leq t \leq 1$.

```matlab
X = []; for t = 0:.01:1
    X = [X expm(t*A)*x0];
end
```

A three-dimensional phase plane plot shows the solution spiraling in towards the origin. This behavior is related to the eigenvalues of the coefficient matrix.

```matlab
plot3(X(1,:),X(2,:),X(3,:),'-o')
```
**Eigenvalues**

- “Eigenvalue Decomposition” on page 3-40
- “Multiple Eigenvalues” on page 3-41
- “Schur Decomposition” on page 3-42
Eigenvalue Decomposition

An eigenvalue and eigenvector of a square matrix $A$ are, respectively, a scalar $\lambda$ and a nonzero vector $v$ that satisfy

$$Av = \lambda v.$$ 

With the eigenvalues on the diagonal of a diagonal matrix $\Lambda$ and the corresponding eigenvectors forming the columns of a matrix $V$, you have

$$AV = V\Lambda.$$ 

If $V$ is nonsingular, this becomes the eigenvalue decomposition

$$A = V\Lambda V^{-1}.$$ 

A good example is provided by the coefficient matrix of the ordinary differential equation of the previous section:

$$A = \begin{bmatrix} 0 & -6 & -1 \\ 6 & 2 & -16 \\ -5 & 20 & -10 \end{bmatrix}$$

The statement

$$\text{lambda} = \text{eig}(A)$$

produces a column vector containing the eigenvalues. For this matrix, the eigenvalues are complex:

$$\text{lambda} = \begin{bmatrix} -3.0710 \\ -2.4645 + 17.6008i \\ -2.4645 - 17.6008i \end{bmatrix}$$

The real part of each of the eigenvalues is negative, so $e^{\lambda t}$ approaches zero as $t$ increases. The nonzero imaginary part of two of the eigenvalues, $\pm \omega$, contributes the oscillatory component, $\sin(\omega t)$, to the solution of the differential equation.
With two output arguments, `eig` computes the eigenvectors and stores the eigenvalues in a diagonal matrix:

\[ [V, D] = \text{eig}(A) \]

\[
V = \\
\begin{bmatrix}
-0.8326 & 0.2003 - 0.1394i & 0.2003 + 0.1394i \\
-0.3553 & -0.2110 - 0.6447i & -0.2110 + 0.6447i \\
-0.4248 & -0.6930 & -0.6930 \\
\end{bmatrix}
\]

\[
D = \\
\begin{bmatrix}
-3.0710 & 0 & 0 \\
0 & -2.4645 + 17.6008i & 0 \\
0 & 0 & -2.4645 - 17.6008i \\
\end{bmatrix}
\]

The first eigenvector is real and the other two vectors are complex conjugates of each other. All three vectors are normalized to have Euclidean length, \( \text{norm}(v,2) \), equal to one.

The matrix \( V*D*\text{inv}(V) \), which can be written more succinctly as \( V*D/V \), is within round-off error of \( A \). And, \( \text{inv}(V)*A*V \), or \( V\backslash A*V \), is within round-off error of \( D \).

**Multiple Eigenvalues**

Some matrices do not have an eigenvector decomposition. These matrices are not diagonalizable. For example:

\[
A = \begin{bmatrix}
6 & 12 & 19 \\
-9 & -20 & -33 \\
4 & 9 & 15 \\
\end{bmatrix}
\]

For this matrix

\[ [V, D] = \text{eig}(A) \]

produces

\[
V = \\
\begin{bmatrix}
-0.4741 & -0.4082 & -0.4082 \\
0.8127 & 0.8165 & 0.8165 \\
\end{bmatrix}
\]
There is a double eigenvalue at $\lambda = 1$. The second and third columns of $V$ are the same. For this matrix, a full set of linearly independent eigenvectors does not exist.

**Schur Decomposition**

The MATLAB advanced matrix computations do not require eigenvalue decompositions. They are based, instead, on the Schur decomposition

$$A = USU^T.$$  

where $U$ is an orthogonal matrix and $S$ is a block upper triangular matrix with 1-by-1 and 2-by-2 blocks on the diagonal. The eigenvalues are revealed by the diagonal elements and blocks of $S$, while the columns of $U$ provide a basis with much better numerical properties than a set of eigenvectors. The Schur decomposition of this defective example is

$$[U, S] = \text{schur}(A)$$

$$U =
\begin{bmatrix}
-0.4741 & 0.6648 & 0.5774 \\
0.8127 & 0.0782 & 0.5774 \\
-0.3386 & -0.7430 & 0.5774 \\
\end{bmatrix}$$

$$S =
\begin{bmatrix}
-1.0000 & 20.7846 & -44.6948 \\
0 & 1.0000 & -0.6096 \\
0 & 0 & 1.0000 \\
\end{bmatrix}$$

The double eigenvalue is contained in the lower 2-by-2 block of $S$. 

\[
\begin{bmatrix}
-0.3386 & -0.4082 & -0.4082 \\
-1.0000 & 0 & 0 \\
0 & 1.0000 & 0 \\
0 & 0 & 1.0000 \\
\end{bmatrix}
\]
Note If \( A \) is complex, \texttt{schur} returns the complex Schur form, which is upper triangular with the eigenvalues of \( A \) on the diagonal.

**Singular Values**

A *singular value* and corresponding *singular vectors* of a rectangular matrix \( A \) are, respectively, a scalar \( \sigma \) and a pair of vectors \( u \) and \( v \) that satisfy

\[
Av = \sigma u \\
A^T u = \sigma v.
\]

With the singular values on the diagonal of a diagonal matrix \( \Sigma \) and the corresponding singular vectors forming the columns of two orthogonal matrices \( U \) and \( V \), you have

\[
AV = U \Sigma \\
A^T U = V \Sigma.
\]

Since \( U \) and \( V \) are orthogonal, this becomes the singular value decomposition

\[
A = U \Sigma V^T.
\]

The full singular value decomposition of an \( m \)-by-\( n \) matrix involves an \( m \)-by-\( m \) \( U \), an \( m \)-by-\( n \) \( \Sigma \), and an \( n \)-by-\( n \) \( V \). In other words, \( U \) and \( V \) are both square and \( \Sigma \) is the same size as \( A \). If \( A \) has many more rows than columns, the resulting \( U \) can be quite large, but most of its columns are multiplied by zeros in \( \Sigma \). In this situation, the *economy* sized decomposition saves both time and storage by producing an \( m \)-by-\( n \) \( U \), an \( n \)-by-\( n \) \( \Sigma \) and the same \( V \).

The eigenvalue decomposition is the appropriate tool for analyzing a matrix when it represents a mapping from a vector space into itself, as it does for an ordinary differential equation. However, the singular value decomposition is the appropriate tool for analyzing a mapping from one vector space into another vector space, possibly with a different dimension. Most systems of simultaneous linear equations fall into this second category.

If \( A \) is square, symmetric, and positive definite, then its eigenvalue and singular value decompositions are the same. But, as \( A \) departs from symmetry
and positive definiteness, the difference between the two decompositions increases. In particular, the singular value decomposition of a real matrix is always real, but the eigenvalue decomposition of a real, nonsymmetric matrix might be complex.

For the example matrix

\[ A = \begin{bmatrix} 9 & 4 \\ 6 & 8 \\ 2 & 7 \end{bmatrix} \]

the full singular value decomposition is

\[ [U, S, V] = \text{svd}(A) \]

\[ U = \begin{bmatrix} 0.6105 & -0.7174 & 0.3355 \\ 0.6646 & 0.2336 & -0.7098 \\ 0.4308 & 0.6563 & 0.6194 \end{bmatrix} \]

\[ S = \begin{bmatrix} 14.9359 & 0 \\ 0 & 5.1883 \\ 0 & 0 \end{bmatrix} \]

\[ V = \begin{bmatrix} 0.6925 & -0.7214 \\ 0.7214 & 0.6925 \end{bmatrix} \]

You can verify that \( U*S*V' \) is equal to \( A \) to within round-off error. For this small problem, the economy size decomposition is only slightly smaller:

\[ [U, S, V] = \text{svd}(A, 0) \]

\[ U = \begin{bmatrix} 0.6105 & -0.7174 & 0.3355 \\ 0.6646 & 0.2336 & -0.7098 \\ 0.4308 & 0.6563 & 0.6194 \end{bmatrix} \]
\[
\begin{bmatrix}
0.6105 & -0.7174 \\
0.6646 & 0.2336 \\
0.4308 & 0.6563
\end{bmatrix}
\]

\[
S = \begin{bmatrix}
14.9359 & 0 \\
0 & 5.1883
\end{bmatrix}
\]

\[
V = \begin{bmatrix}
0.6925 & -0.7214 \\
0.7214 & 0.6925
\end{bmatrix}
\]

Again, \( U^*S^*V' \) is equal to \( A \) to within round-off error.
Operations on Nonlinear Functions

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Function Handles” on page 3-46</td>
</tr>
<tr>
<td>“Function Functions” on page 3-46</td>
</tr>
</tbody>
</table>

Function Handles
You can create a handle to any MATLAB function, and then use that handle as a means of referencing the function. A function handle is typically passed in an argument list to other functions, which can then execute, or evaluate, the function using the handle.

Construct a function handle in MATLAB using the at sign, @, before the function name. The following example creates a function handle for the sin function and assigns it to the variable fhandle:

```matlab
fhandle = @sin;
```

You can call a function by means of its handle in the same way that you would call the function using its name. The syntax is

```matlab
fhandle(arg1, arg2, ...);
```

The function plot_fhandle, shown below, receives a function handle and data, generates y-axis data using the function handle, and plots it:

```matlab
function plot_fhandle(fhandle, data)
    plot(data, fhandle(data))
end
```

When you call plot_fhandle with a handle to the sin function and the argument shown below, the resulting evaluation produces a sine wave plot:

```matlab
plot_fhandle(@sin, -pi:0.01:pi)
```

Function Functions
A class of functions called “function functions” works with nonlinear functions of a scalar variable. That is, one function works on another function. The function functions include
- Zero finding
- Optimization
- Quadrature
- Ordinary differential equations

MATLAB represents the nonlinear function by the file that defines it. For example, here is a simplified version of the function humps from the matlab/demos folder:

```
function y = humps(x)
y = 1./((x-.3).^2 + .01) + 1./((x-.9).^2 + .04) - 6;
```

Evaluate this function at a set of points in the interval $0 \leq x \leq 1$ with

```
x = 0:.002:1;
y = humps(x);
```

Then plot the function with

```
plot(x,y)
```
The graph shows that the function has a local minimum near $x = 0.6$. The function `fminsearch` finds the minimizer, the value of $x$ where the function takes on this minimum. The first argument to `fminsearch` is a function handle to the function being minimized and the second argument is a rough guess at the location of the minimum:

$$ p = \text{fminsearch}(@humps,.5) $$

$$ p = 0.6370 $$

To evaluate the function at the minimizer,

$$ \text{humps}(p) $$

$$ \text{ans} = 11.2528 $$

Numerical analysts use the terms *quadrature* and *integration* to distinguish between numerical approximation of definite integrals and numerical integration of ordinary differential equations. MATLAB quadrature routines are `quad` and `quadl`. The statement

$$ Q = \text{quadl}(@humps,0,1) $$

computes the area under the curve in the graph and produces

$$ Q = 29.8583 $$

Finally, the graph shows that the function is never zero on this interval. So, if you search for a zero with

$$ z = \text{fzero}(@humps,.5) $$

you will find one outside the interval

$$ z = -0.1316 $$
Multivariate Data

MATLAB uses column-oriented analysis for multivariate statistical data. Each column in a data set represents a variable and each row an observation. The \((i,j)\)th element is the \(i\)th observation of the \(j\)th variable.

As an example, consider a data set with three variables:

- Heart rate
- Weight
- Hours of exercise per week

For five observations, the resulting array might look like

\[
D = \begin{bmatrix}
72 & 134 & 3.2 \\
81 & 201 & 3.5 \\
69 & 156 & 7.1 \\
82 & 148 & 2.4 \\
75 & 170 & 1.2 \\
\end{bmatrix}
\]

The first row contains the heart rate, weight, and exercise hours for patient 1, the second row contains the data for patient 2, and so on. Now you can apply many MATLAB data analysis functions to this data set. For example, to obtain the mean and standard deviation of each column, use

\[
\text{mu} = \text{mean}(D), \quad \text{sigma} = \text{std}(D)
\]

\[
\text{mu} = \\
\begin{bmatrix}
75.8 & 161.8 & 3.48 \\
\end{bmatrix}
\]

\[
\text{sigma} = \\
\begin{bmatrix}
5.6303 & 25.499 & 2.2107 \\
\end{bmatrix}
\]

For a list of the data analysis functions available in MATLAB, type

\text{help datafun}

If you have access to the Statistics Toolbox™ software, type

\text{help stats}
Data Analysis

Introduction
Every data analysis has some standard components:

- Preprocessing — Consider outliers and missing values, and smooth data to identify possible models.
- Summarizing — Compute basic statistics to describe the overall location, scale, and shape of the data.
- Visualizing — Plot data to identify patterns and trends.
- Modeling — Give data trends fuller descriptions, suitable for predicting new values.

Data analysis moves among these components with two basic goals in mind:

1. Describe the patterns in the data with simple models that lead to accurate predictions.

2. Understand the relationships among variables that lead to the model.

This section explains how to carry out a basic data analysis in the MATLAB environment.

Preprocessing Data
This example shows how to preprocess data for analysis.

Overview

Begin a data analysis by loading data into suitable MATLAB® container variables and sorting out the "good" data from the "bad." This is a preliminary step that assures meaningful conclusions in subsequent parts of the analysis.

Loading the Data

Begin by loading the data in count.dat:

load count.dat
The 24-by-3 array count contains hourly traffic counts (the rows) at three intersections (the columns) for a single day.

**Missing Data**

The MATLAB NaN (Not a Number) value is normally used to represent missing data. NaN values allow variables with missing data to maintain their structure - in this case, 24-by-1 vectors with consistent indexing across all three intersections.

Check the data at the third intersection for NaN values using the isnan function:

```matlab
c3 = count(:,3); % Data at intersection 3
c3NaNCount = sum(isnan(c3))
```

```
c3NaNCount =
      0
```

isnan returns a logical vector the same size as c3, with entries indicating the presence (1) or absence (0) of NaN values for each of the 24 elements in the data. In this case, the logical values sum to 0, so there are no NaN values in the data.

NaN values are introduced into the data in the section on Outliers.

**Outliers**

Outliers are data values that are dramatically different from patterns in the rest of the data. They might be due to measurement error, or they might represent significant features in the data. Identifying outliers, and deciding what to do with them, depends on an understanding of the data and its source.

One common method for identifying outliers is to look for values more than a certain number of standard deviations $\sigma$ from the mean $\mu$. The following
code plots a histogram of the data at the third intersection together with lines at $\mu$ and $\mu + \eta$, for $\eta = 1, 2$:

```matlab
bin_counts = hist(c3); % Histogram bin counts
N = max(bin_counts); % Maximum bin count
mu3 = mean(c3); % Data mean
sigma3 = std(c3); % Data standard deviation

hist(c3) % Plot histogram
hold on
plot([mu3 mu3],[0 N],'r','LineWidth',2) % Mean
X = repmat(mu3+(1:2)*sigma3,2,1);
Y = repmat([0;N],1,2);
plot(X,Y,'g','LineWidth',2) % Standard deviations
legend('Data','Mean','Stds')
hold off
```
The plot shows that some of the data are more than two standard deviations above the mean. If you identify these data as errors (not features), replace them with NaN values as follows:

\[
\text{outliers} = (c3 - \text{mu3}) > 2*\text{sigma3};
\]
\[
c3m = c3; \quad \% \text{Copy } c3 \text{ to } c3m
\]
\[
c3m(\text{outliers}) = \text{NaN}; \quad \% \text{Add NaN values}
\]

**Smoothing and Filtering**
A time-series plot of the data at the third intersection (with the outlier removed in Outliers) results in the following plot:

```matlab
plot(c3m,'o-')
hold on
```

The NaN value at hour 20 appears as a gap in the plot. This handling of NaN values is typical of MATLAB plotting functions.
Noisy data shows random variations about expected values. You might want to smooth the data to reveal its main features before building a model. Two basic assumptions underlie smoothing:

- The relationship between the predictor (time) and the response (traffic volume) is smooth.
- The smoothing algorithm results in values that are better estimates of expected values because the noise has been reduced.

Apply a simple moving average smoother to the data using the MATLAB `convn` function:

```matlab
span = 3; % Size of the averaging window
window = ones(span,1)/span;
smoothed_c3m = convn(c3m,window,'same');

h = plot(smoothed_c3m,'ro-');
legend('Data','Smoothed Data')
```
The extent of the smoothing is controlled with the variable span. The averaging calculation returns NaN values whenever the smoothing window includes the NaN value in the data, thus increasing the size of the gap in the smoothed data.

The filter function is also used for smoothing data:

```matlab
smoothed2_c3m = filter(window,1,c3m);
delete(h)
```
The smoothed data are shifted from the previous plot. `convn` with the `same` parameter returns the central part of the convolution, the same length as the data. `filter` returns the initial part of the convolution, the same length as the data. Otherwise, the algorithms are identical.

Smoothing estimates the center of the distribution of response values at each value of the predictor. It invalidates a basic assumption of many fitting algorithms, namely, that the errors at each value of the predictor are
Accordingly, you can use smoothed data to identify a model, but avoid using smoothed data to fit a model.

### Summarizing Data
This example shows how to summarize data.

#### Overview

Many MATLAB® functions enable you to summarize the overall location, scale, and shape of a data sample.

One of the advantages of working in MATLAB® is that functions operate on entire arrays of data, not just on single scalar values. The functions are said to be *vectorized*. Vectorization allows for both efficient problem formulation, using array-based data, and efficient computation, using vectorized statistical functions.

#### Measures of Location

Summarize the location of a data sample by finding a "typical" value. Common measures of location or "central tendency" are computed by the functions `mean`, `median`, and `mode`:

```matlab
load count.dat
x1 = mean(count)
x2 = median(count)
x3 = mode(count)
```

```plaintext
x1 =
  32.0000  46.5417  65.5833

x2 =
  23.5000  36.0000  39.0000
```
Like all of its statistical functions, the MATLAB® functions above summarize data across observations (rows) while preserving variables (columns). The functions compute the location of the data at each of the three intersections in a single call.

**Measures of Scale**

There are many ways to measure the scale or "dispersion" of a data sample. The MATLAB® functions `max`, `min`, `std`, and `var` compute some common measures:

\[
dx1 = \text{max}(\text{count}) - \text{min}(\text{count})
\]
\[
dx2 = \text{std}(\text{count})
\]
\[
dx3 = \text{var}(\text{count})
\]

\[
dx1 = \begin{bmatrix} 107 & 136 & 250 \end{bmatrix}
\]
\[
dx2 = \begin{bmatrix} 25.3703 & 41.4057 & 68.0281 \end{bmatrix}
\]
\[
dx3 = \begin{bmatrix} 1.0e+03 *  \\
0.6437 & 1.7144 & 4.6278 \end{bmatrix}
\]
functions compute the scale of the data at each of the three intersections in a single call.

**Shape of a Distribution**

The shape of a distribution is harder to summarize than its location or scale. The MATLAB® hist function plots a histogram that provides a visual summary:

```matlab
figure
hist(count)
legend('Intersection 1', ...
    'Intersection 2', ...
    'Intersection 3')
```
Parametric models give analytic summaries of distribution shapes. Exponential distributions, with parameter $\mu$ given by the data mean, are a good choice for the traffic data:

```matlab
c1 = count(:,1); % Data at intersection 1
[bin_counts,bin_locations] = hist(c1);
bin_width = bin_locations(2) - bin_locations(1);
hist_area = (bin_width)*(sum(bin_counts));
figure
```
```matlab
hist(c1)
hold on

mu1 = mean(c1);
exp_pdf = @(t)(1/mu1)*exp(-t/mu1); % Integrates
% to 1

t = 0:150;
y = exp_pdf(t);
plot(t,(hist_area)*y,'r','LineWidth',2)
legend('Distribution','Exponential Fit')
```
Methods for fitting general parametric models to data distributions are beyond the scope of this section. Statistics Toolbox™ software provides functions for computing maximum likelihood estimates of distribution parameters.

**Visualizing Data**

- “Overview” on page 3-63
- “2-D Scatter Plots” on page 3-63
- “3-D Scatter Plots” on page 3-66
- “Scatter Plot Arrays” on page 3-68
- “Exploring Data in Graphs” on page 3-69

**Overview**
You can use many MATLAB graph types for visualizing data patterns and trends. Scatter plots, described in this section, help to visualize relationships among the traffic data at different intersections. Data exploration tools let you query and interact with individual data points on graphs.

---

**Note** This section continues the data analysis from “Summarizing Data” on page 3-58.

---

### 2-D Scatter Plots
A two-dimensional scatter plot, created with the `scatter` function, shows the relationship between the traffic volume at the first two intersections:

```matlab
load count.dat
c1 = count(:,1); % Data at intersection 1
c2 = count(:,2); % Data at intersection 2

figure
scatter(c1,c2,'filled')
xlabel('Intersection 1')
ylabel('Intersection 2')
```
The covariance, computed by the \texttt{cov} function measures the strength of the linear relationship between the two variables (how tightly the data lies along a least-squares line through the scatter):

\[ C_{12} = \text{cov}([c_1 \ c_2]) \]

\[ C_{12} = 1.0e+03 \ * \]
The results are displayed in a symmetric square matrix, with the covariance of the $i$th and $j$th variables in the $(i, j)$th position. The $i$th diagonal element is the variance of the $i$th variable.

Covariances have the disadvantage of depending on the units used to measure the individual variables. You can divide a covariance by the standard deviations of the variables to normalize values between $+1$ and $-1$. The \texttt{corrcoef} function computes correlation coefficients:

\begin{verbatim}
R12 = corrcoef([c1 c2])

R12 =

1.0000 0.9331
0.9331 1.0000
\end{verbatim}

\begin{verbatim}
r12 = R12(1,2) \% Correlation coefficient

r12 =

0.9331
\end{verbatim}

\begin{verbatim}
r12sq = r12^2 \% Coefficient of determination

r12sq =

0.8707
\end{verbatim}
Because it is normalized, the value of the correlation coefficient is readily comparable to values for other pairs of intersections. Its square, the *coefficient of determination*, is the variance about the least-squares line divided by the variance about the mean. Thus, it is the proportion of variation in the response (in this case, the traffic volume at intersection 2) that is eliminated or statistically explained by a least-squares line through the scatter.

### 3-D Scatter Plots

A three-dimensional scatter plot, created with the `scatter3` function, shows the relationship between the traffic volume at all three intersections. Use the variables `c1`, `c2`, and `c3` that you created in the previous step:

```matlab
figure
c3 = count(:,3); % Data at intersection 3
scatter3(c1,c2,c3,'filled')
xlabel('Intersection 1')
ylabel('Intersection 2')
zlabel('Intersection 3')
```
Measure the strength of the linear relationship among the variables in the three-dimensional scatter by computing eigenvalues of the covariance matrix with the `eig` function:

```matlab
vars = eig(cov([c1 c2 c3]))
```

```matlab
vars =

1.0e+03 *
```

explained = max(vars)/sum(vars)

explained = 0.9777

The eigenvalues are the variances along the principal components of the data. The variable explained measures the proportion of variation explained by the first principal component, along the axis of the data. Unlike the coefficient of determination for two-dimensional scatters, this measure distinguishes predictor and response variables.

**Scatter Plot Arrays**

Use the `plotmatrix` function to make comparisons of the relationships between multiple pairs of intersections:

```matlab
figure
plotmatrix(count)
```
The plot in the \((i, j)\)th position of the array is a scatter with the \(i\)th variable on the vertical axis and the \(j\)th variable on the horizontal axis. The plot in the \(i\)th diagonal position is a histogram of the \(i\)th variable.

**Exploring Data in Graphs**

Using your mouse, you can pick observations on almost any MATLAB graph with two tools from the figure toolbar:

- Data Cursor
• **Data Brushing**

These tools each place you in exploratory modes in which you can select data points on graphs to identify their values and create workspace variables to contain specific observations. When you use data brushing, you can also copy, remove or replace the selected observations.

For example, make a scatter plot of the first and third columns of count:

```matlab
load count.dat
scatter(count(:,1),count(:,3))
```

Select the Data Cursor Tool and click the rightmost data point. A datatip displaying the point’s $x$ and $y$ value is placed there.

![Figure 1](image_url)
Datatips display $x$-, $y$-, and $z$- (for three-dimensional plots) coordinates by default. You can drag a datatip from one data point to another to see new values or add additional datatips by right-clicking a datatip and using the context menu. You can also customize the text that datatips display using MATLAB code.

Data brushing is a related feature that lets you highlight one or more observations on a graph by clicking or dragging. To enter data brushing mode, click the left side of the Data Brushing tool on the figure toolbar. Clicking the arrow on the right side of the tool icon drops down a color palette for selecting the color with which to brush observations. This figure shows the same scatter plot as the previous figure, but with all observations beyond one standard deviation of the mean (as identified using the Tools > Data Statistics GUI) brushed in red.

scatter(count(:,1),count(:,3))
After you brush data observations, you can perform the following operations on them:

- Delete them.
- Replace them with constant values.
- Replace them with NaN values.
- Drag or copy, and paste them to the Command Window.
- Save them as workspace variables.

For example, use the Data Brush context menu or the **Tools > Brushing > Create new variable** option to create a new variable called `count13high`. 
A new variable in the workspace results:

count13high

count13high =
   61  186
   75  180
  114  257

Linked plots, or data linking, is a feature closely related to data brushing. A plot is said to be linked when it has a live connection to the workspace data it depicts. The copies of variables stored in a plot object’s XData, YData, (and, where appropriate, ZData), automatically updated whenever the workspace variables to which they are linked change or are deleted. This causes the graphs on which they appear to update automatically.

Linking plots to variables lets you track specific observations through different presentations of them. When you brush data points in linked plots, brushing one graph highlights the same observations in every graph that is linked to the same variables.
Data linking establishes immediate, two-way communication between figures and workspace variables, in the same way that the Variable Editor communicates with workspace variables. You create links by activating the Data Linking tool on a figure’s toolbar. Activating this tool causes the Linked Plot information bar, displayed in the next figure, to appear at the top of the plot (possibly obscuring its title). You can dismiss the bar (shown in the following figure) without unlinking the plot; it does not print and is not saved with the figure.

The following two graphs depict scatter plot displays of linked data after brushing some observations on the left graph. The common variable, count carries the brush marks to the right figure. Even though the right graph is not in data brushing mode, it displays brush marks because it is linked to its variables.

```matlab
figure
scatter(count(:,1),count(:,2))
xlabel ('count(:,1)')
ylabel ('count(:,2)')
figure
scatter(count(:,3),count(:,2))
xlabel ('count(:,3)')
ylabel ('count(:,2)')
```
The right plot shows that the brushed observations are more linearly related than in the left plot.

Brushed data observations appear highlighted in the brushing color when you display those variables in the Variable Editor, as you can see here:

```
openvar count
```
In the Variable Editor, you can alter any values of linked plot data, and the graphs will reflect your edits. To brush data observation from the Variable Editor, click its Brushing Tool button. If the variable you brush is currently depicted in a linked plot, the observations you brush highlight in the plot, as well as in the Variable Editor. When you brush a variable that is a column in a matrix, the other columns in that row are also brushed. That is, you can brush individual observations in a row or column vector, but all
columns in a matrix highlight in any row you brush, not just the observations you click.

**Modeling Data**

- “Overview” on page 3-77
- “Polynomial Regression” on page 3-77
- “General Linear Regression” on page 3-78

**Overview**

Parametric models translate an understanding of data relationships into analytic tools with predictive power. Polynomial and sinusoidal models are simple choices for the up and down trends in the traffic data.

**Polynomial Regression**

Use the `polyfit` function to estimate coefficients of polynomial models, then use the `polyval` function to evaluate the model at arbitrary values of the predictor.

The following code fits the traffic data at the third intersection with a polynomial model of degree six:

```matlab
load count.dat
c3 = count(:,3); % Data at intersection 3
tdata = (1:24)';
p_coeffs = polyfit(tdata,c3,6);

figure
plot(c3,'o-')
hold on
tfit = (1:0.01:24)';
yfit = polyval(p_coeffs,tfit);
plot(tfit,yfit,'r-','LineWidth',2)
legend('Data','Polynomial Fit','Location','NW')
```
Mathematics

The model has the advantage of being simple while following the up-and-down trend. The accuracy of its predictive power, however, is questionable, especially at the ends of the data.

**General Linear Regression**

Assuming that the data are periodic with a 12-hour period and a peak around hour 7, it is reasonable to fit a sinusoidal model of the form:

\[ y = a + b \cos\left(\frac{2\pi}{12}(t - 7)\right) \]
The coefficients $a$ and $b$ appear linearly. Use the MATLAB® mldivide (backslash) operator to fit general linear models:

```matlab
load count.dat
c3 = count(:,3); % Data at intersection 3
tdata = (1:24)';
X = [ones(size(tdata)) cos((2*pi/12)*(tdata-7))];
s_coeffs = X\c3;
figure
plot(c3,'o-')
hold on
tfit = (1:0.01:24)';
yfit = [ones(size(tfit)) cos((2*pi/12)*(tfit-7))]\s_coeffs;
plot(tfit,yfit,'r-','LineWidth',2)
legend('Data','Sinusoidal Fit','Location','NW')
```
Use the `lscov` function to compute statistics on the fit, such as estimated standard errors of the coefficients and the mean squared error:

\[
[s\_coeffs, stdx, mse] = \text{lscov}(X, c3)
\]

\[
s\_coeffs =
\begin{align*}
65.5833 \\
73.2819
\end{align*}
\]
Check the assumption of a 12-hour period in the data with a *periodogram*, computed using the *fft* function:

```matlab
Fs = 1; % Sample frequency (per hour)
n = length(c3); % Window length
Y = fft(c3); % DFT of data
f = (0:n-1)*(Fs/n); % Frequency range
P = Y.*conj(Y)/n; % Power of the DFT

figure
plot(f,P)
xlabel('Frequency')
ylabel('Power')

predicted_f = 1/12

predicted_f =

0.0833
```
The peak near 0.0833 supports the assumption, although it occurs at a slightly higher frequency. The model can be adjusted accordingly.

**See Also**  
`isnan` | `convn` | `filter` | `mean` | `median` | `mode` | `max` | `min` | `std` | `var`  
`hist` | `scatter` | `cov` | `corrcoef` | `scatter3` | `eig` | `plotmatrix` | `polyfit`  
`polyval` | `mldivide` | `lscov` | `fft`
Graphics

- “Basic Plotting Functions” on page 4-2
- “Creating Mesh and Surface Plots” on page 4-17
- “Plotting Image Data” on page 4-24
- “Printing Graphics” on page 4-26
- “Working with Handle Graphics Objects” on page 4-29
Basic Plotting Functions

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Creating a Plot” on page 4-2</td>
</tr>
<tr>
<td>“Plotting Multiple Data Sets in One Graph” on page 4-3</td>
</tr>
<tr>
<td>“Specifying Line Styles and Colors” on page 4-5</td>
</tr>
<tr>
<td>“Plotting Lines and Markers” on page 4-7</td>
</tr>
<tr>
<td>“Graphing Imaginary and Complex Data” on page 4-8</td>
</tr>
<tr>
<td>“Adding Plots to an Existing Graph” on page 4-9</td>
</tr>
<tr>
<td>“Figure Windows” on page 4-11</td>
</tr>
<tr>
<td>“Displaying Multiple Plots in One Figure” on page 4-11</td>
</tr>
<tr>
<td>“Controlling the Axes” on page 4-12</td>
</tr>
<tr>
<td>“Adding Axis Labels and Titles” on page 4-14</td>
</tr>
<tr>
<td>“Saving Figures” on page 4-15</td>
</tr>
</tbody>
</table>

Creating a Plot

The `plot` function has different forms, depending on the input arguments.

- If `y` is a vector, `plot(y)` produces a piecewise linear graph of the elements of `y` versus the index of the elements of `y`.
- If you specify two vectors as arguments, `plot(x,y)` produces a graph of `y` versus `x`.

For example, these statements use the colon operator to create a vector of `x` values ranging from 0 to `2π`, compute the sine of these values, and plot the result:

```plaintext
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```

Add axis labels and a title.
Basic Plotting Functions

```matlab
xlabel('x = 0:2\pi')
ylabel('Sine of x')
title('Plot of the Sine Function','FontSize',12)
```

The characters \pi create the symbol $\pi$ and the FontSize property increases the size the text used for the title:

![Plot of the Sine Function](image)

**Plotting Multiple Data Sets in One Graph**

Multiple x-y pair arguments create multiple graphs with a single call to `plot`. MATLAB® uses a different color for each line.

For example, these statements plot three related functions of $x$:

```matlab
x = 0:pi/100:2*pi;
y = sin(x);
y2 = sin(x-.25);
y3 = sin(x-.5);
```
The `legend` function provides an easy way to identify the individual lines:

```matlab
legend('sin(x)','sin(x-.25)','sin(x-.5)')
```
Specifying Line Styles and Colors

It is possible to specify color, line styles, and markers (such as plus signs or circles) when you plot your data using the `plot` command:

```
plot(x,y,'color_style_marker')
```

`color_style_marker` is a string containing from one to four characters (enclosed in single quotes) constructed from a color, a line style, and a marker type. For example,
```matlab
plot(x,y,'r:+')
```
plots the data using a red-dotted line and places a + marker at each data point.

The strings are composed of combinations of the following elements.

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>'c'</td>
<td>cyan</td>
</tr>
<tr>
<td></td>
<td>'m'</td>
<td>magenta</td>
</tr>
<tr>
<td></td>
<td>'y'</td>
<td>yellow</td>
</tr>
<tr>
<td></td>
<td>'r'</td>
<td>red</td>
</tr>
<tr>
<td></td>
<td>'g'</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>'b'</td>
<td>blue</td>
</tr>
<tr>
<td></td>
<td>'w'</td>
<td>white</td>
</tr>
<tr>
<td></td>
<td>'k'</td>
<td>black</td>
</tr>
<tr>
<td>Line style</td>
<td>'-'</td>
<td>solid</td>
</tr>
<tr>
<td></td>
<td>'--'</td>
<td>dashed</td>
</tr>
<tr>
<td></td>
<td>':'</td>
<td>dotted</td>
</tr>
<tr>
<td></td>
<td>'.-'</td>
<td>dash-dot</td>
</tr>
<tr>
<td></td>
<td>no character</td>
<td>no line</td>
</tr>
<tr>
<td>Marker type</td>
<td>'+'</td>
<td>plus mark</td>
</tr>
<tr>
<td></td>
<td>'o'</td>
<td>unfilled circle</td>
</tr>
<tr>
<td></td>
<td>'*'</td>
<td>asterisk</td>
</tr>
<tr>
<td></td>
<td>'x'</td>
<td>letter x</td>
</tr>
<tr>
<td></td>
<td>'s'</td>
<td>filled square</td>
</tr>
<tr>
<td></td>
<td>'d'</td>
<td>filled diamond</td>
</tr>
<tr>
<td></td>
<td>'^'</td>
<td>filled upward triangle</td>
</tr>
<tr>
<td></td>
<td>'v'</td>
<td>filled downward triangle</td>
</tr>
<tr>
<td></td>
<td>'&gt;'</td>
<td>filled right-pointing triangle</td>
</tr>
<tr>
<td></td>
<td>'&lt;'</td>
<td>filled left-pointing triangle</td>
</tr>
<tr>
<td></td>
<td>'p'</td>
<td>filled pentagram</td>
</tr>
<tr>
<td></td>
<td>'h'</td>
<td>filled hexagram</td>
</tr>
<tr>
<td></td>
<td>no character</td>
<td>no marker</td>
</tr>
</tbody>
</table>
Plotting Lines and Markers

If you specify a marker type, but not a line style, MATLAB creates the graph using only markers, but no line. For example,

\[
\text{plot}(x, y, 'ks')
\]

plots black squares at each data point, but does not connect the markers with a line.

The statement

\[
\text{plot}(x, y, 'r:+')
\]

plots a red-dotted line and places plus sign markers at each data point.

Placing Markers at Every Tenth Data Point

You can use fewer data points to plot the markers than you use to plot the lines. This example plots the data twice using a different number of points for the dotted line and marker plots:

\[
x1 = 0:pi/100:2*pi;
x2 = 0:pi/10:2*pi;
plot(x1, sin(x1), 'r:', x2, sin(x2), 'r+')
\]
Graphing Imaginary and Complex Data

When you pass complex values as arguments to `plot`, MATLAB ignores the imaginary part, except when you pass a single complex argument. For this special case, the command is a shortcut for a graph of the real part versus the imaginary part. Therefore,

`plot(Z)`

where `Z` is a complex vector or matrix, is equivalent to

`plot(real(Z),imag(Z))`

For example,

```matlab
t = 0:pi/10:2*pi;
plot(exp(i*t),'-o')
axis equal
```
Basic Plotting Functions

draws a 20-sided polygon with little circles at the vertices. The \texttt{axis equal} command makes the individual tick-mark increments on the $x$- and $y$-axes the same length, which makes this plot more circular in appearance.

\section*{Adding Plots to an Existing Graph}

The \texttt{hold} command enables you to add plots to an existing graph. When you type,

\begin{verbatim}
hold on
\end{verbatim}

MATLAB does not replace the existing graph when you issue another plotting command. Instead, MATLAB combines the new graph with the current graph.

For example, these statements first create a contour plot of the \texttt{peaks} function, then superimpose a \texttt{pcolor} (pseudocolor) plot of the same function:

\begin{verbatim}
% Obtain data from evaluating peaks function
\end{verbatim}
[x,y,z] = peaks;
% Create pseudocolor plot
pcolor(x,y,z)
% Remove edge lines a smooth colors
shading interp
% Hold the current graph
hold on
% Add the contour graph to the pcolor graph
contour(x,y,z,20,'k')
% Return to default
hold off

The **hold on** command combines the **pcolor** plot with the **contour** plot in one figure.
**Figure Windows**

Graphing functions automatically open a new figure window if there are no figure windows already created. If there are multiple figure windows open, MATLAB uses the one that is designated as the “current figure” (the last figure used or clicked on).

To make an existing figure window the current figure, you can click the mouse while the pointer is in that window or you can type,

\[ \text{figure(n)} \]

where \( n \) is the number in the figure title bar.

To open a new figure window and make it the current figure, type

\[ \text{figure} \]

**Clearing the Figure for a New Plot**

When a figure already exists, most plotting commands clear the axes and use this figure to create the new plot. However, these commands do not reset figure properties, such as the background color or the colormap. If you have set any figure properties in the previous plot, you might want to use the `clf` command with the `reset` option,

\[ \text{clf reset} \]

before creating your new plot to restore the figure’s properties to their defaults.

**Displaying Multiple Plots in One Figure**

The `subplot` command enables you to display multiple plots in the same window or print them on the same piece of paper. Typing

\[ \text{subplot}(m,n,p) \]

partitions the figure window into an \( m \)-by-\( n \) matrix of small subplots and selects the \( p \)th subplot for the current plot. The plots are numbered along the first row of the figure window, then the second row, and so on. For example, these statements plot data in four different subregions of the figure window:
t = 0:pi/10:2*pi;
[X,Y,Z] = cylinder(4*cos(t));
subplot(2,2,1); mesh(X)
subplot(2,2,2); mesh(Y)
subplot(2,2,3); mesh(Z)
subplot(2,2,4); mesh(X,Y,Z)

Controlling the Axes
The axis command provides a number of options for setting the scaling, orientation, and aspect ratio of graphs.
**Automatic Axis Limits and Tick Marks**

By default, MATLAB finds the maxima and minima of the data and chooses the axis limits to span this range. MATLAB selects the limits and axis tick mark values to produce a graph that clearly displays the data. However, you can set your own limits using the `axis` or `xlim`, `ylim`, and `zlim` functions.

**Note** Changing the limits of one axis can cause other limits to change to better represent the data. To disable automatic limit setting, enter the `axis manual` command.

**Setting Axis Limits**

The `axis` command enables you to specify your own limits:

```matlab
axis([xmin xmax ymin ymax])
```

or for three-dimensional graphs,

```matlab
axis([xmin xmax ymin ymax zmin zmax])
```

Use the command

```matlab
axis auto
```

to enable automatic limit selection again.

**Setting the Axis Aspect Ratio**

The `axis` command also enables you to specify a number of predefined modes. For example,

```matlab
axis square
```

makes the x-axis and y-axis the same length.

```matlab
axis equal
```

makes the individual tick mark increments on the x-axes and y-axes the same length. This means

```matlab
plot(exp(i*[0:pi/10:2*pi]))
```
followed by either axis square or axis equal turns the oval into a proper circle:

axis auto normal

returns the axis scaling to its default automatic mode.

**Setting Axis Visibility**
You can use the axis command to make the axis visible or invisible.

axis on

makes the axes visible. This is the default.

axis off

makes the axes invisible.

**Setting Grid Lines**
The grid command toggles grid lines on and off. The statement

grid on

turns the grid lines on, and

grid off

turns them back off again.

**Adding Axis Labels and Titles**
The xlabel, ylabel, and zlabel commands add x-, y-, and z-axis labels. The title command adds a title at the top of the figure and the text function inserts text anywhere in the figure.

You can produce mathematical symbols using LaTeX notation in the text string, as the following example illustrates:

```matlab
 t = -pi:pi/100:pi;
y = sin(t);
plot(t,y)
```
axis([-pi pi -1 1])
xlabel('-\pi \leq \it{t} \leq \pi')
ylabel('\sin(t)')
title('Graph of the sine function')
text(0.5,-1/3,'\it{Note the odd symmetry.}')

The location of the text string is defined in axes units (that is, the same units as the data). The annotation function enables you to place text in normalized figure units.

**Saving Figures**

Save a figure by selecting **Save** from the **File** menu. This writes the figure to a file, including property data, its menus, uicontrols, and all annotations (i.e., the entire window). If you have not saved the figure before, the **Save As** dialog displays. This dialog box provides options to save the figure as a FIG-file or export it to a graphics format.
If you have previously saved the figure, using **Save** again saves the figure “silently,” without displaying the **Save As** dialog.

To save a figure using a standard graphics format for use with other applications, such as TIFF or JPG, select **Save As** (or **Export Setup**, if you want additional control) from the **File** menu.

**Note** Whenever you specify a format for saving a figure, that file format is used again the next time you save that figure or a new one. If you do not want to save in the previously used format, use **Save As** and be sure to set the **Save as type** drop-down menu to the kind of file you want to write.

You can also save from the command line—use the `saveas` command, including any options to save the figure in a different format. The more restricted `hgexport` command, which saves figures to either bitmap or metafile files, depending on the rendering method in effect, is also available.

**Saving Workspace Data**

You can save the variables in your workspace by selecting **Save Workspace As** from the figure **File** menu. You can reload saved data using the **Import Data** item in the figure **File** menu. MATLAB supports a variety of data file formats, including MATLAB data files, which have a `.mat` extension.

**Generating MATLAB Code to Recreate a Figure**

You can generate MATLAB code that recreates a figure and the graph it contains by selecting **Generate code** from the figure **File** menu. This option is particularly useful if you have developed a graph using plotting tools and want to create a similar graph using the same or different data.
Creating Mesh and Surface Plots

In this section...

“About Mesh and Surface Plots” on page 4-17
“Visualizing Functions of Two Variables” on page 4-17

About Mesh and Surface Plots
MATLAB defines a surface by the $z$-coordinates of points above a grid in the $x$-$y$ plane, using straight lines to connect adjacent points. The `mesh` and `surf` functions display surfaces in three dimensions.

- `mesh` produces wireframe surfaces that color only the lines connecting the defining points.
- `surf` displays both the connecting lines and the faces of the surface in color.

MATLAB colors surfaces by mapping $z$-data values to indexes into the figure colormap.

Visualizing Functions of Two Variables
To display a function of two variables, $z = f(x,y)$,

1. Generate $X$ and $Y$ matrices consisting of repeated rows and columns, respectively, over the domain of the function.

2. Use $X$ and $Y$ to evaluate and graph the function.

The `meshgrid` function transforms the domain specified by a single vector or two vectors $x$ and $y$ into matrices $X$ and $Y$ for use in evaluating functions of two variables. The rows of $X$ are copies of the vector $x$ and the columns of $Y$ are copies of the vector $y$.

Example — Graphing the sinc Function
This example evaluates and graphs the two-dimensional sinc function, $\sin(r)/r$, between the $x$ and $y$ directions. $R$ is the distance from the origin, which
Graphics

is at the center of the matrix. Adding \texttt{eps} (a MATLAB command that returns a small floating-point number) avoids the indeterminate 0/0 at the origin:

\begin{verbatim}
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
mesh(X,Y,Z,'EdgeColor','black')
\end{verbatim}

By default, MATLAB uses the current colormap to color the mesh. However, this example uses a single-colored mesh by specifying the \texttt{EdgeColor} surface property.

You can create a mesh with see-through faces by disabling hidden line removal:

\begin{verbatim}
hidden off
\end{verbatim}
Example — Colored Surface Plots
A surface plot is similar to a mesh plot except that the rectangular faces of the surface are colored. The color of each face is determined by the values of \( Z \) and the colormap (a colormap is an ordered list of colors). These statements graph the \( \text{sinc} \) function as a surface plot, specify a colormap, and add a color bar to show the mapping of data to color:

\[
\text{surf}(X,Y,Z) \\
\text{colormap hsv} \\
\text{colorbar}
\]

Making Surfaces Transparent
You can make the faces of a surface transparent to a varying degree. Transparency (referred to as the alpha value) can be specified for the whole
object or can be based on an alphamap, which behaves similarly to colormaps. For example,

```
surf(X,Y,Z)
colormap hsv
alpha(.4)
```

produces a surface with a face alpha value of 0.4. Alpha values range from 0 (completely transparent) to 1 (not transparent).

**Illuminating Surface Plots with Lights**

Lighting is the technique of illuminating an object with a directional light source. In certain cases, this technique can make subtle differences in surface shape easier to see. Lighting can also be used to add realism to three-dimensional graphs.
Creating Mesh and Surface Plots

This example uses the same surface as the previous examples, but colors it red and removes the mesh lines. A light object is then added to the left of the “camera” (the camera is the location in space from where you are viewing the surface):

```matlab
surf(X,Y,Z,'FaceColor','red','EdgeColor','none')
camlight left; lighting phong
```

**Manipulating the Surface**

The figure toolbar and the camera toolbar provide ways to explore three-dimensional graphics interactively. Display the camera toolbar by selecting **Camera Toolbar** from the figure **View** menu.

The following picture shows both toolbars with the **Rotate 3D** tool selected.
These tools enable you to move the camera around the surface object, zoom, add lighting, and perform other viewing operations without issuing commands.

The following picture shows the surface viewed by orbiting the camera toward the bottom using **Rotate 3D**. You can see the tool’s cursor icon on the surface. As you drag, the viewing azimuth and elevation read out in the lower-left corner of the axes.
Plotting Image Data

In this section...

“About Plotting Image Data” on page 4-24
“Reading and Writing Images” on page 4-25

About Plotting Image Data
Two-dimensional arrays can be displayed as images, where the array elements determine brightness or color of the images. For example, the statements

load durer
whos

Name Size Bytes Class
X 648x509 2638656 double array
caption 2x28 112 char array
map 128x3 3072 double array

load the file durer.mat, adding three variables to the workspace. The matrix X is a 648-by-509 matrix and map is a 128-by-3 matrix that is the colormap for this image.

MAT-files, such as durer.mat, are binary files that can be created on one platform and later read by the MATLAB software on a different platform.

The elements of X are integers between 1 and 128, which serve as indices into the colormap, map. Then

image(X)
colormap(map)
axis image

reproduces Albrecht Dürer’s etching. A high-resolution scan of the magic square in the upper-right corner is available in another file. Type

load detail
and then use the up arrow key on your keyboard to reexecute the `image`, `colormap`, and `axis` commands.

**Reading and Writing Images**

You can read standard image files (TIFF, JPEG, BMP, and so on, using the `imread` function. The type of data returned by `imread` depends on the type of image you are reading.

You can write MATLAB data to a variety of standard image formats using the `imwrite` function.
Printing Graphics

In this section...

“Overview of Printing” on page 4-26
“Printing from the File Menu” on page 4-26
“Exporting the Figure to a Graphics File” on page 4-27
“Using the Print Command” on page 4-27

Overview of Printing
You can print a MATLAB figure directly on a printer connected to your computer or you can export the figure to one of the standard graphics file formats that MATLAB supports. There are two ways to print and export figures:

- Use the Print, Print Preview, or Export Setup GUI options under the File menu.
- Use the print command to print or export the figure from the command line.

The print command provides greater control over drivers and file formats. The Print Preview dialog box gives you greater control over figure size, proportions, placement, and page headers.

Printing from the File Menu
There are two menu options under the File menu that pertain to printing:

- The Print Preview option displays a dialog box that lets you lay out and style figures for printing while previewing the output page, and from which you can print the figure. It includes options that formerly were part of the Page Setup dialog box.
- The Print option displays a dialog box that lets you choose a printer, select standard printing options, and print the figure.
Use **Print Preview** to determine whether the printed output is what you want. Click the Print Preview dialog box **Help** button to display information on how to set up the page.

**Exporting the Figure to a Graphics File**

The **Export Setup** option in the **File** menu opens a GUI that enables you to set graphic characteristics, such as text size, font, and style, for figures you save as graphics files. The **Export Setup** dialog lets you define and apply templates to customize and standardize output. After setup, you can export the figure to a number of standard graphics file formats, such as EPS, PNG, and TIFF.

**Using the Print Command**

The **print** command provides more flexibility in the type of output sent to the printer and allows you to control printing from function and script files. The result can be sent directly to your default printer or stored in a specified output file. A wide variety of output formats is available, including TIFF, JPEG, and PNG.

For example, this statement saves the contents of the current figure window as a PNG graphic in the file called `magicsquare.png`.

```
print -dpng magicsquare.png
```
To save the figure at the same size as the figure on the screen, use these statements:

```matlab
set(gcf,'PaperPositionMode','auto')
print -dpng -r0 magicsquare.png
```

To save the same figure as a TIFF file with a resolution of 200 dpi, use the following command:

```matlab
print -dtiff -r200 magicsquare.tiff
```

If you type `print` on the command line

`print`

the current figure prints on your default printer.
Working with Handle Graphics Objects

In this section...

“Graphics Objects” on page 4-29
“Setting Object Properties” on page 4-31
“Functions for Working with Objects” on page 4-34
“Specifying Axes or Figures” on page 4-35
“Finding the Handles of Existing Objects” on page 4-37

Graphics Objects

Graphics objects are the basic elements used to display graphs and user interface components. These objects are organized into a hierarchy, as shown by the following diagram.

When you call a plotting function, MATLAB creates the graph using various graphics objects, such as a figure window, axes, lines, text, and so on. Each object has a fixed set of properties, which you can use to control the behavior and appearance of your graph.

For example, the following statement creates a figure with a white background color and does not display the figure toolbar:

```matlab
figure('Color','white','Toolbar','none')
```
Common Graphics Objects

When you call a function to create a graph, MATLAB creates a hierarchy of graphics objects. For example, calling the `plot` function creates the following graphics objects:

- Figure — Window that contains axes, toolbars, menus, and so on.
- Axes — Coordinate system that contains the lines representing the data
- Lineseries — Lines that represent the value of data passed to the `plot` function.
- Text — Labels for axes tick marks and optional titles and annotations.

Different types of graphs use different objects to represent data. All data objects are contained in axes and all objects (except root) are contained in figures.

The root is an abstract object that primarily stores information about your computer or MATLAB states. You cannot create an instance of the root object. The handle of the root object is always 0.

Object Handles

When MATLAB creates a graphics object, MATLAB assigns an identifier to the object. This identifier is called a handle. You can use this handle to access the object’s properties with the `set` and `get` functions. For example, the following statements create a graph and return a handle to a `lineseries` object in `h`:

```matlab
x = 1:10;
y = x.^3;
h = plot(x,y);
```

You can use the handle `h` to set the properties of the lineseries object. For example, you can set its Color property:

```matlab
set(h,'Color','red')
```

You can also specify the lineseries properties when you call the plotting function:

```matlab
h = plot(x,y,'Color','red');
```
You can query the lineseries properties to see the current value:

```matlab
get(h,'LineWidth')
```

The `get` function returns the answer (in units of points for `LineWidth`):

```matlab
ans =
    0.5000
```

**Finding the Properties of an Object**

If you call `get` with only a handle, MATLAB returns a list of the object’s properties:

```matlab
get(h)
```

If you call `set` with only a handle, MATLAB returns a list of the object’s properties with information about possible values:

```matlab
set(h)
```

**Setting Object Properties**

All object properties have default values. However, you can change the settings of some properties to customize your graph. There are two ways to set object properties:

- Specify values for properties when you create the object.
- Set the property value on an object that already exists.

**Setting Properties from Plotting Commands**

You can specify object property value pairs as arguments to many plotting functions, such as `plot`, `mesh`, and `surf`.

For example, plotting commands that create lineseries or surfaceplot objects enable you to specify property name/property value pairs as arguments. The command

```matlab
[x,y,z] = peaks;
surf(x,y,z,...
     'FaceColor','interp',...
'EdgeColor', [.7, .7, .7])

plots the data in the variables x, y, and z using a surfaceplot object with interpolated face color and light gray colored edges.

Setting Properties of Existing Objects
To modify the property values of existing objects, use the set function.
Plotting functions return the handles of the data objects that they create (lines, surfaces, images, and so on). For example, the following statements plot a 5-by-5 matrix (creating five lineseries objects, one per column), and then set the Marker property to square and the MarkerFaceColor property to green:

```matlab
y = magic(5);
h = plot(y);
set(h, 'Marker', 's', 'MarkerFaceColor', 'g')
```

In this case, h is a vector containing five handles, one for each of the five lineseries in the graph. The set statement sets the Marker and MarkerFaceColor properties of all lineseries to the same values.

To set a property value on one object, index into the handle array:

```matlab
set(h(1), 'LineWidth', 2)
```

Setting Multiple Property Values
If you want to set the properties of each lineseries to a different value, you can use cell arrays to store all the data and pass it to the set command. For example, create a plot and save the lineseries handles:

```matlab
h = plot(magic(5));
```

Suppose you want to add different markers to each lineseries and color the marker’s face color the same color as the lineseries. You need to define two cell arrays—one containing the property names and the other containing the desired values of the properties.

The prop_name cell array contains two elements:

```matlab
prop_name(1) = {'Marker'};
prop_name(2) = {'MarkerFaceColor'};
```
The `prop_values` cell array contains 10 values: five values for the `Marker` property and five values for the `MarkerFaceColor` property. Notice that `prop_values` is a two-dimensional cell array. The first dimension indicates which handle in `h` the values apply to and the second dimension indicates which property the value is assigned to:

```matlab
prop_values(1,1) = {'s'};
prop_values(1,2) = {get(h(1),'Color')};
prop_values(2,1) = {'d'};
prop_values(2,2) = {get(h(2),'Color')};
prop_values(3,1) = {'o'};
prop_values(3,2) = {get(h(3),'Color')};
prop_values(4,1) = {'p'};
prop_values(4,2) = {get(h(4),'Color')};
prop_values(5,1) = {'h'};
prop_values(5,2) = {get(h(5),'Color')};
```

The `MarkerFaceColor` is always assigned the value of the corresponding line’s color (obtained by getting the lineseries `Color` property with the `get` function).

After defining the cell arrays, call `set` to specify the new property values:

```matlab
set(h,prop_name,prop_values)
```
### Functions for Working with Objects
This table lists functions commonly used when working with objects.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>allchild</td>
<td>Find all children of specified objects.</td>
</tr>
<tr>
<td>ancestor</td>
<td>Find ancestor of graphics object.</td>
</tr>
<tr>
<td>copyobj</td>
<td>Copy graphics object.</td>
</tr>
<tr>
<td>delete</td>
<td>Delete an object.</td>
</tr>
<tr>
<td>findAll</td>
<td>Find all graphics objects (including hidden handles).</td>
</tr>
<tr>
<td>Function</td>
<td>Purpose</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>findobj</td>
<td>Find the handles of objects having specified property values.</td>
</tr>
<tr>
<td>gca</td>
<td>Return the handle of the current axes.</td>
</tr>
<tr>
<td>gcf</td>
<td>Return the handle of the current figure.</td>
</tr>
<tr>
<td>gco</td>
<td>Return the handle of the current object.</td>
</tr>
<tr>
<td>get</td>
<td>Query the values of an object's properties.</td>
</tr>
<tr>
<td>ishandle</td>
<td>True if the value is a valid object handle.</td>
</tr>
<tr>
<td>set</td>
<td>Set the values of an object's properties.</td>
</tr>
</tbody>
</table>

**Specifying Axes or Figures**

MATLAB always creates an axes or figure if one does not exist when you execute a plotting command. However, when you are creating graphics from a program file, it is good practice to create and specify the parent axes and figure explicitly, particularly if others people use your program. Specifying the parent prevents the following problems:

- Your program overwrites the graph in the current figure. A figure becomes the current figure whenever a user clicks it.
- The current figure might be in an unexpected state and not behave as your program expects.

The following example shows a MATLAB function that evaluates a mathematical expression over the range specified in the input argument x, and then plots the results. A second call to the plot function plots the mean value of the results as a red line.

```matlab
function myfunc(x)
    % Evaluate the expression using the input argument
    y = 1.5*cos(x) + 6*exp(-.1*x) + exp(.07*x).*sin(3*x);

    % Calculate the mean
    ym = mean(y);

    % Create a figure, axes parented to that axes
```
% and the using the axes
hfig = figure('Name','Function and Mean');
hax = axes('Parent',hfig);
plot(hax,x,y)

% Hold the current plot and add a red line along the mean value
hold on
plot(hax,[min(x) max(x)],[ym ym],'Color','red')
hold off

% Add a tick label that shows the mean value
% and add a title and axis labels
ylab = get(hax,'YTick');
set(hax,'YTick',sort([ylab ym]))
title ('y = 1.5cos(x) + 6e^{-0.1x} + e^{0.07x}sin(3x)')
xlabel('X Axis'); ylabel('Y Axis')
end

First, define a value for the input argument and call the function:

```matlab
x = -10:.005:40;
myfunc(x)
```
Finding the Handles of Existing Objects

The `findobj` function enables you to obtain the handles of graphics objects by searching for objects with particular property values. With `findobj` you can specify the values of any combination of properties, which makes it easy to pick one object out of many. `findobj` also recognizes regular expressions.

For example, you might want to find the blue line with square marker having blue face color. You can also specify which figures or axes to begin searching from, if there are more than one. The following four sections provide examples illustrating how to use `findobj`. 
Finding All Objects of a Certain Type
Because all objects have a Type property that identifies the type of object, you can find the handles of all occurrences of a particular type of object. For example,

```matlab
h = findobj('Type','patch');
```

finds the handles of all patch objects.

Finding Objects with a Particular Property
You can specify multiple properties to narrow the search. For example,

```matlab
h = findobj('Type','line','Color','r','LineStyle',':');
```

finds the handles of all red dotted lines.

Limiting the Scope of the Search
You can specify the starting point in the object hierarchy by passing the handle of the starting figure or axes as the first argument. For example,

```matlab
h = findobj(gca,'Type','text','String','\pi/2');
```

finds the string $\pi/2$ only within the current axes.

Using findobj as an Argument
Because findobj returns the handles it finds, you can use it in place of the handle argument. For example,

```matlab
set(findobj('Type','line','Color','red'),'LineStyle',':');
```

finds all red lines and sets their line style to dotted.
Programming

- “Control Flow” on page 5-2
- “Scripts and Functions” on page 5-10
Control Flow

In this section...

“Conditional Control — if, else, switch” on page 5-2
“Loop Control — for, while, continue, break” on page 5-5
“Program Termination — return” on page 5-7
“Vectorization” on page 5-8
“Preallocation” on page 5-8

Conditional Control — if, else, switch

Conditional statements enable you to select at run time which block of code to execute. The simplest conditional statement is an if statement. For example:

% Generate a random number
a = randi(100, 1);

% If it is even, divide by 2
if rem(a, 2) == 0
    disp('a is even')
    b = a/2;
end

if statements can include alternate choices, using the optional keywords elseif or else. For example:

a = randi(100, 1);

if a < 30
    disp('small')
elseif a < 80
    disp('medium')
else
    disp('large')
end

Alternatively, when you want to test for equality against a set of known values, use a switch statement. For example:
Control Flow

```matlab
[dayNum, dayString] = weekday(date, 'long', 'en_US');

switch dayString
    case 'Monday'
        disp('Start of the work week')
    case 'Tuesday'
        disp('Day 2')
    case 'Wednesday'
        disp('Day 3')
    case 'Thursday'
        disp('Day 4')
    case 'Friday'
        disp('Last day of the work week')
    otherwise
        disp('Weekend!')
end
```

For both `if` and `switch`, MATLAB executes the code corresponding to the first true condition, and then exits the code block. Each conditional statement requires the `end` keyword.

In general, when you have many possible discrete, known values, `switch` statements are easier to read than `if` statements. However, you cannot test for inequality between `switch` and `case` values. For example, you cannot implement this type of condition with a `switch`:

```matlab
yourNumber = input('Enter a number: ');

if yourNumber < 0
    disp('Negative')
elseif yourNumber > 0
    disp('Positive')
else
    disp('Zero')
end
```

**Array Comparisons in Conditional Statements**

It is important to understand how relational operators and `if` statements work with matrices. When you want to check for equality between two variables, you might use
if A == B, ...

This is valid MATLAB code, and does what you expect when A and B are scalars. But when A and B are matrices, A == B does not test if they are equal, it tests where they are equal; the result is another matrix of 0s and 1s showing element-by-element equality. (In fact, if A and B are not the same size, then A == B is an error.)

A = magic(4);    B = A;    B(1,1) = 0;

A == B
ans =
     0  1  1  1
     1  1  1  1
     1  1  1  1
     1  1  1  1

The proper way to check for equality between two variables is to use the isequal function:

if isequal(A, B), ...

isequal returns a scalar logical value of 1 (representing true) or 0 (false), instead of a matrix, as the expression to be evaluated by the if function. Using the A and B matrices from above, you get

isequal(A, B)
ans =
     0

Here is another example to emphasize this point. If A and B are scalars, the following program will never reach the “unexpected situation”. But for most pairs of matrices, including our magic squares with interchanged columns, none of the matrix conditions A > B, A < B, or A == B is true for all elements and so the else clause is executed:

if A > B
     'greater'
elseif A < B
     'less'
elseif A == B
     'equal'
else
    error('Unexpected situation')
end

Several functions are helpful for reducing the results of matrix comparisons to scalar conditions for use with if, including

isequal
isempty
all
any

**Loop Control — for, while, continue, break**

This section covers those MATLAB functions that provide control over program loops.

**for**

The `for` loop repeats a group of statements a fixed, predetermined number of times. A matching `end` delineates the statements:

```matlab
for n = 3:32
    r(n) = rank(magic(n));
end
r
```

The semicolon terminating the inner statement suppresses repeated printing, and the `r` after the loop displays the final result.

It is a good idea to indent the loops for readability, especially when they are nested:

```matlab
for i = 1:m
    for j = 1:n
        H(i,j) = 1/(i+j);
    end
end
```
**while**

The *while* loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching *end* delineates the statements.

Here is a complete program, illustrating *while*, *if*, *else*, and *end*, that uses interval bisection to find a zero of a polynomial:

```plaintext
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
    x = (a+b)/2;
    fx = x^3 - 2*x - 5;
    if sign(fx) == sign(fa)
        a = x; fa = fx;
    else
        b = x; fb = fx;
    end
end
x
```

The result is a root of the polynomial $x^3 - 2x - 5$, namely

$$x = 2.09455148154233$$

The cautions involving matrix comparisons that are discussed in the section on the *if* statement also apply to the *while* statement.

**continue**

The *continue* statement passes control to the next iteration of the *for* loop or *while* loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for *continue* statements in nested loops. That is, execution continues at the beginning of the loop in which the *continue* statement was encountered.

The example below shows a *continue* loop that counts the lines of code in the file `magic.m`, skipping all blank lines and comments. A *continue* statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered:
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
fprintf('%d lines\n',count);
fclose(fid);

**break**
The *break* statement lets you exit early from a *for* loop or *while* loop. In nested loops, *break* exits from the innermost loop only.

Here is an improvement on the example from the previous section. Why is this use of *break* a good idea?

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
    x = (a+b)/2;
    fx = x^3-2*x-5;
    if fx == 0
        break
    elseif sign(fx) == sign(fa)
        a = x; fa = fx;
    else
        b = x; fb = fx;
    end
end
x
```

**Program Termination — return**
This section covers the MATLAB *return* function that enables you to terminate your program before it runs to completion.
**return**

return terminates the current sequence of commands and returns control to the invoking function or to the keyboard. return is also used to terminate keyboard mode. A called function normally transfers control to the function that invoked it when it reaches the end of the function. You can insert a return statement within the called function to force an early termination and to transfer control to the invoking function.

**Vectorization**

One way to make your MATLAB programs run faster is to vectorize the algorithms you use in constructing the programs. Where other programming languages might use for loops or do loops, MATLAB can use vector or matrix operations. A simple example involves creating a table of logarithms:

```matlab
x = .01;
for k = 1:1001
    y(k) = log10(x);
    x = x + .01;
end
```

A vectorized version of the same code is

```matlab
x = .01:.01:10;
y = log10(x);
```

For more complicated code, vectorization options are not always so obvious.

**Preallocation**

If you cannot vectorize a piece of code, you can make your for loops go faster by preallocating any vectors or arrays in which output results are stored. For example, this code uses the function zeros to preallocate the vector created in the for loop. This makes the for loop execute significantly faster:

```matlab
r = zeros(32,1);
for n = 1:32
    r(n) = rank(magic(n));
end
```
Without the preallocation in the previous example, the MATLAB interpreter enlarges the $r$ vector by one element each time through the loop. Vector preallocation eliminates this step and results in faster execution.
Scripts and Functions

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Overview” on page 5-10</td>
</tr>
<tr>
<td>“Scripts” on page 5-11</td>
</tr>
<tr>
<td>“Functions” on page 5-12</td>
</tr>
<tr>
<td>“Types of Functions” on page 5-14</td>
</tr>
<tr>
<td>“Global Variables” on page 5-16</td>
</tr>
<tr>
<td>“Command vs. Function Syntax” on page 5-16</td>
</tr>
</tbody>
</table>

Overview

The MATLAB product provides a powerful programming language, as well as an interactive computational environment. You can enter commands from the language one at a time at the MATLAB command line, or you can write a series of commands to a file that you then execute as you would any MATLAB function. Use the MATLAB Editor or any other text editor to create your own function files. Call these functions as you would any other MATLAB function or command.

There are two kinds of program files:

- Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace.
- Functions, which can accept input arguments and return output arguments. Internal variables are local to the function.

If you are a new MATLAB programmer, just create the program files that you want to try out in the current folder. As you develop more of your own files, you will want to organize them into other folders and personal toolboxes that you can add to your MATLAB search path.

If you duplicate function names, MATLAB executes the one that occurs first in the search path.

To view the contents of a program file, for example, myfunction.m, use
type myfunction

**Scripts**

When you invoke a *script*, MATLAB simply executes the commands found in the file. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, to be used in subsequent computations. In addition, scripts can produce graphical output using functions like `plot`.

For example, create a file called `magicrank.m` that contains these MATLAB commands:

```matlab
% Investigate the rank of magic squares
r = zeros(1,32);
for n = 3:32
    r(n) = rank(magic(n));
end
r
bar(r)
```

Typing the statement

```
magicrank
```

causes MATLAB to execute the commands, compute the rank of the first 30 magic squares, and plot a bar graph of the result. After execution of the file is complete, the variables `n` and `r` remain in the workspace.
Functions

Functions are files that can accept input arguments and return output arguments. The names of the file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.

A good example is provided by `rank`. The file `rank.m` is available in the folder `toolbox/matlab/matfun`

You can see the file with

type `rank`

Here is the file:

```matlab
function r = rank(A,tol)
% RANK Matrix rank.
% RANK(A) provides an estimate of the number of linearly
% independent rows or columns of a matrix A.
```
% RANK(A,tol) is the number of singular values of A
% that are larger than tol.
% RANK(A) uses the default tol = max(size(A)) * norm(A) * eps.

s = svd(A);
if nargin==1
    tol = max(size(A)') * max(s) * eps;
end
r = sum(s > tol);

The first line of a function starts with the keyword function. It gives the
function name and order of arguments. In this case, there are up to two input
arguments and one output argument.

The next several lines, up to the first blank or executable line, are comment
lines that provide the help text. These lines are printed when you type

help rank

The first line of the help text is the H1 line, which MATLAB displays when
you use the lookfor command or request help on a folder.

The rest of the file is the executable MATLAB code defining the function. The
variable s introduced in the body of the function, as well as the variables on
the first line, r, A and to1, are all local to the function; they are separate from
any variables in the MATLAB workspace.

This example illustrates one aspect of MATLAB functions that is not
ordinarily found in other programming languages—a variable number of
arguments. The rank function can be used in several different ways:

rank(A)
r = rank(A)
r = rank(A,1.e-6)

Many functions work this way. If no output argument is supplied, the result
is stored in ans. If the second input argument is not supplied, the function
computes a default value. Within the body of the function, two quantities
named nargin and nargout are available that tell you the number of input
and output arguments involved in each particular use of the function. The
rank function uses nargin, but does not need to use nargout.
Types of Functions
MATLAB offers several different types of functions to use in your programming.

Anonymous Functions
An anonymous function is a simple form of the MATLAB function that is defined within a single MATLAB statement. It consists of a single MATLAB expression and any number of input and output arguments. You can define an anonymous function right at the MATLAB command line, or within a function or script. This gives you a quick means of creating simple functions without having to create a file for them each time.

The syntax for creating an anonymous function from an expression is

\[ f = @(\text{arglist}) \text{expression} \]

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable \( x \), and then uses \( x \) in the equation \( x.^2 \):

\[ \text{sqr} = @(x) x.^2; \]

To execute the \text{sqr} function defined above, type

\[ a = \text{sqr}(5) \]
\[ a = 25 \]

Primary and Subfunctions
Any function that is not anonymous must be defined within a file. Each such function file contains a required primary function that appears first, and any number of subfunctions that can follow the primary. Primary functions have a wider scope than subfunctions. That is, primary functions can be called from outside of the file that defines them (for example, from the MATLAB command line or from functions in other files) while subfunctions cannot. Subfunctions are visible only to the primary function and other subfunctions within their own file.
The rank function shown in the section on “Functions” on page 5-12 is an example of a primary function.

**Private Functions**

A private function is a type of primary function. Its unique characteristic is that it is visible only to a limited group of other functions. This type of function can be useful if you want to limit access to a function, or when you choose not to expose the implementation of a function.

Private functions reside in subfolders with the special name `private`. They are visible only to functions in the parent folder. For example, assume the folder `newmath` is on the MATLAB search path. A subfolder of `newmath` called `private` can contain functions that only the functions in `newmath` can call.

Because private functions are invisible outside the parent folder, they can use the same names as functions in other folders. This is useful if you want to create your own version of a particular function while retaining the original in another folder. Because MATLAB looks for private functions before standard functions, it will find a private function named `test.m` before a nonprivate file named `test.m`.

**Nested Functions**

You can define functions within the body of another function. These are said to be nested within the outer function. A nested function contains any or all of the components of any other function. In this example, function B is nested in function A:

```matlab
function x = A(p1, p2)
    ...
    B(p2)
        function y = B(p3)
            ...
        end
    end
    ...
end
```

Like other functions, a nested function has its own workspace where variables used by the function are stored. But it also has access to the workspaces of all functions in which it is nested. So, for example, a variable that has
a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary. Similarly, a variable that is assigned in a nested function can be read or overwritten by any of the functions containing that function.

**Global Variables**

If you want more than one function to share a single copy of a variable, simply declare the variable as `global` in all the functions. Do the same thing at the command line if you want the base workspace to access the variable. The global declaration must occur before the variable is actually used in a function. Although it is not required, using capital letters for the names of global variables helps distinguish them from other variables. For example, create a new function in a file called `falling.m`:

```matlab
function h = falling(t)
global GRAVITY
h = 1/2*GRAVITY*t.^2;
```

Then interactively enter the statements

```matlab
global GRAVITY
GRAVITY = 32;
y = falling((0:.1:5)');
```

The two global statements make the value assigned to `GRAVITY` at the command prompt available inside the function. You can then modify `GRAVITY` interactively and obtain new solutions without editing any files.

**Command vs. Function Syntax**

You can write MATLAB functions that accept string arguments without the parentheses and quotes. That is, MATLAB interprets

```matlab
foo a b c
```

as

```matlab
foo('a','b','c')
```

However, when you use the unquoted command form, MATLAB cannot return output arguments. For example,
legend apples oranges

creates a legend on a plot using the strings apples and oranges as labels. If you want the legend command to return its output arguments, then you must use the quoted form:

[legh, objh] = legend('apples', 'oranges');

In addition, you must use the quoted form if any of the arguments is not a string.

**Caution** While the unquoted command syntax is convenient, in some cases it can be used incorrectly without causing MATLAB to generate an error.

**Constructing String Arguments in Code**

The quoted function form enables you to construct string arguments within the code. The following example processes multiple data files, August1.dat, August2.dat, and so on. It uses the function int2str, which converts an integer to a character, to build the file name:

```matlab
for d = 1:31
    s = ['August' int2str(d) '.dat'];
    load(s)
    % Code to process the contents of the d-th file
end
```
Symbols and Numerics
: operator 1-11 2-21
2-D scatter plots
getting started 3-63
3-D scatter plots
getting started 3-66

A
algorithms
   vectorizing 5-8
ans function 2-5
array operators 1-7 2-13
arrays 1-6
   and matrices 2-12
   cell 2-29
   character 2-31
   columnwise organization 3-49
   deleting rows and columns 2-23
   elements 2-11
   generating with functions and operators 1-7 2-8
   listing contents 1-13 2-10
   multidimensional 2-27
   notation for elements 2-11
   preallocating 5-8
   structure 2-34
   variable names 2-10
arrow keys for editing commands 2-19
aspect ratio of axes 4-13
axes
   managing 4-12
   visibility 4-14
axis
   labels 1-18 4-14
   titles 4-14
   axis function 4-12

B
break function 5-7
built-in functions
defined 2-15

C
cell arrays 2-29
char function 2-33
character arrays 2-31
Cholesky factorization 3-28
coefficient of determination
described 3-66
colon operator 1-11 2-21
colormap 4-19
colors
   lines for plotting 1-18 4-5
command line
   editing 1-3 2-19
complex numbers 2-11
   plotting 4-8
concatenation
defined 1-9 2-22
   of strings 2-32
constants
   special 2-15
continue function 5-6
continuing statements on multiple lines 2-19
control keys for editing commands 2-19
correlation coefficient
   example using corrcoef 3-65
covariance
   example using cov 3-64

data analysis
   getting started 3-50
decomposition
eigenvalue 3-40
Schur 3-42
singular value 3-43
deleting array elements 2-23
desktop
for MATLAB 1-3
determinant of matrix 3-23
diaq function 2-5
documentation 1-30
dot product 3-6

E
editing command lines 1-3 2-19
eigenvalues 3-40
eigenvectors 3-40
elements of arrays 2-11
entering matrices 2-4
examples 1-30
expressions
examples 2-16
using in MATLAB 2-10
eye
   derivation of the name 3-9

F
factorization
   Cholesky 3-28
   Hermitian positive definite 3-29
   LU 3-29
   partial pivoting 3-30
   positive definite 3-28
   QR 3-31
figure function 4-11
figure windows 4-11
   with multiple plots 1-18 4-11
find function 2-26
finding object handles 4-37
flip1r function 2-7
floating-point numbers 2-11
flow control 5-2
for loop 5-5
format
   of output display 2-17
format function 2-17
function files 5-10
function functions 3-46
function handles
   defined 3-46
   using 3-48
function keyword 5-13
function of two variables 1-18 4-17
function program files
   naming 5-12
functions
   built-in, defined 2-15
   calling 1-17
   defined 5-12
   how to find 2-14
   how to find help on 1-30
   variable number of arguments 5-13
G
Gaussian elimination 3-29
global variables 5-16
graphics
   files 4-27
   Handle Graphics 4-29
   objects 4-29
   printing 4-29
   grids 4-14
H
Handle Graphics 4-29
   finding handles 4-37
help functions 1-37
Hermitian positive definite matrix 3-29
hold function 1-18 4-9
i

identity matrix 3-9
images 4-24
imaginary numbers 1-10 2-11
inner product 3-5
inverse of matrix 3-23

K

keys for editing in Command Window 2-19
Kronecker tensor matrix product 3-9

L

lighting 4-20
limits
  axes 4-13
line continuation 2-19
line styles of plots 1-18 4-5
linear equations
  minimal norm solution 3-26
  overdetermined systems 3-17
  rectangular systems 3-25
linear regression
  getting started 3-77
linear systems of equations
  full 3-11
linear transformation 3-2
local variables 5-13
logical vectors 2-25
LU factorization 3-29

M

magic function 2-7
magic square 2-5
markers 1-18 4-7
MAT-file 4-24
mathematical functions
  listing elementary 2-14
  listing matrix 2-15
MATLAB
  desktop 1-3
matrices 1-6 3-1 to 3-2
  as linear transformation 3-2
  creating 1-6 2-20
  creation 3-2
  determinant 3-23
  entering 2-4
  identity 3-9
  inverse 3-23
  orthogonal 3-31
  pseudoinverse 3-25
  rank deficiency 3-20
  symmetric 3-5
  triangular 3-28
matrix 1-6 2-2
  antidiagonal 2-7
  main diagonal 2-6
  swapping columns 1-11 2-8
  transpose 2-5
matrix operations
  addition and subtraction 3-4
  division 3-12
  exponentials 3-37
  multiplication 3-7
  powers 3-35
  transpose 3-5
matrix products
  Kronecker tensor 3-9
mesh plot 1-18 4-17
modeling data
  getting started 3-77
Moore-Penrose pseudoinverse 3-25
multidimensional arrays 2-27
multiple plots per figure 1-18 4-11
multivariate data
  organizing 3-49
N
norms
  vector and matrix 3-10
numbers 2-11
  complex 1-10 2-11
  floating-point 2-11

O
object properties 4-31
objects
  finding handles 4-37
  graphics 4-29
operators 2-12
  colon 1-11 2-21
orthogonal matrix 3-31
outer product 3-5
output
  controlling format 2-17
  suppressing 1-3 2-18
overdetermined
  rectangular matrices 3-17
overlaying plots 1-18 4-9

P
periodogram 3-77
plot
  titles 1-18
plot function 1-18 4-2
plotting
  adding plots 1-18 4-9
  basic 4-2
  complex data 4-8
  complex numbers 4-8
  contours 4-9
  functions 4-2
  line colors 1-18 4-5
  line styles 1-18 4-5
  lines and markers 1-18 4-7
  mesh and surface 1-18 4-17
  multiple plots 1-18 4-11
  overview 1-18
polynomial regression
  getting started 3-77
PostScript 4-27
preallocation 5-8
principal components 3-68
print function 4-26
printing
  graphics 4-26
program files
  creating 1-26 5-10
  function 5-10
  scripts 1-26 5-10
pseudoinverse
  of matrix 3-25

Q
QR factorization 3-31

R
rank deficiency
  detecting 3-33
  rectangular matrices 3-20
rectangular matrices
  identity 3-9
  overdetermined systems 3-17
  pseudoinverse 3-25
  QR factorization 3-31
  rank deficient 3-20
  rank deficient 3-20
  singular value decomposition 3-43
return function 5-8

S
scalar
  as a matrix 3-3
scalar expansion 1-11 2-24
scalar product 3-6
scatter plot arrays
   getting started 3-68
Schur decomposition 3-42
scientific notation 2-11
script files 1-26 5-10
scripts 1-26 5-11
semicolon to suppress output 1-3 2-18
singular value matrix decomposition 3-43
solving linear systems of equations
   full 3-11
special constants
   infinity 2-15
   not-a-number 2-15
statements
   continuing on multiple lines 2-19
strings
   concatenating 2-32
structures 2-34
subplot function 4-11
subscripting
   with logical vectors 2-25
subscripts 1-11 2-20
sum function 2-5
suppressing output 1-3 2-18
surface plot 1-18 4-17
symmetric matrix
   transpose 3-5

T
text
   entering in MATLAB 2-31
TIFF 4-27
title
   figure 1-18 4-14
transpose 1-7

complex conjugate 3-6
unconjugated complex 3-6
transpose function 2-5
triangular matrix 3-28

U
unitary matrices
   QR factorization 3-31

V
variables 1-13 2-10
   global 5-16
   local 5-13
vector products
   dot or scalar 3-6
   outer and inner 3-5
vectorization 5-8
vectors 1-6 2-2
   column and row 3-3
   logical 2-25
   multiplication 3-5
   preallocating 5-8
visibility of axes 4-14
visualizing data
   getting started 3-63

W
while loop 5-6
windows
   in MATLAB 1-3
   windows for plotting 4-11
wireframe
   surface 1-18 4-17
LabVIEW™

Getting Started with LabVIEW
Worldwide Technical Support and Product Information
ni.com

Worldwide Offices
Visit ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

National Instruments Corporate Headquarters
11500 North Mopac Expressway  Austin, Texas 78759-3504  USA  Tel: 512 683 0100

For further support information, refer to the Technical Support and Professional Services appendix. To comment on National Instruments documentation, refer to the National Instruments website at ni.com/info and enter the Info Code feedback.

© 2003–2013 National Instruments. All rights reserved.
Important Information

Warranty
The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER’S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THEREFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner’s failure to follow the National Instruments installation, operation, or maintenance instructions; owner’s modification of the product; owner’s abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright
Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

For end-user license agreements (EULAs) and copyright notices, conditions, and disclaimers, including information regarding certain third-party components used in LabVIEW, refer to the Copyright topic of the LabVIEW Help.

Trademarks
Refer to the NI Trademarks and Logo Guidelines at ni.com/trademarks for more information on National Instruments trademarks.

ARM, Keil, and µVision are trademarks or registered of ARM Ltd or its subsidiaries.

LEGO, the LEGO logo, WEDO, and MINDSTORMS are trademarks of the LEGO Group. ©2013 The LEGO Group.

TETRIX by Pitso is a trademark of Pitso, Inc.©2013

FIELDBUS FOUNDATION™ and FOUNDATION™ are trademarks of the Fieldbus Foundation.

EtherCAT® is a registered trademark of Beckhoff Automation GmbH.

CANopen® is a registered Community Trademark of CAN in Automation e.V.

DeviceNet™ and EtherNet/IP™ are trademarks of ODVA.

Go!, SensorDAQ, and Vernier are registered trademarks of Vernier Software & Technology. Vernier Software & Technology and vernier.com are trademarks or trade dress.

Xilinx is the registered trademark of Xilinx, Inc.

Taptite and Trilobular are registered trademarks of Research Engineering & Manufacturing Inc.

FireWire® is the registered trademark of Apple Inc.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Handle Graphics®, MATLAB®, Real-Time Workshop®, Simulink®, Stateflow®, and xPC TargetBox® are registered trademarks, and TargetBox™ and Target Language Compiler™ are trademarks of The MathWorks, Inc.

Tektronix®, Tek, and Tektronix, Enabling Technology are registered trademarks of Tektronix, Inc.

The Bluetooth® word mark is a registered trademark owned by the Bluetooth SIG, Inc.

The ExpressCard™ word mark and logos are owned by PCMCIA and any use of such marks by National Instruments is under license.

The mark LabWindows is used under a license from Microsoft Corporation. Windows is a registered trademark of Microsoft Corporation in the United States and other countries.
Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents
For patents covering National Instruments products/technology, refer to the appropriate location: Help»Patents in your software, the patents.txt file on your media, or the National Instruments Patent Notice at ni.com/patents.

Export Compliance Information
Refer to the Export Compliance Information at ni.com/legal/export-compliance for the National Instruments global trade compliance policy and how to obtain relevant HTS codes, ECCNs, and other import/export data.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS
(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED “SYSTEM FAILURES”). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS’ TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.
Contents

About This Manual
Related Documentation ...................................................................................................... ix

Chapter 1
Getting Started with LabVIEW Virtual Instruments

Building a Virtual Instrument ................................................................................................. 1-1
  Launching LabVIEW ........................................................................................................... 1-2
  Opening a New VI from a Template ............................................................................... 1-2
  Adding a Control to the Front Panel ............................................................................... 1-4
  Changing a Signal Type .................................................................................................... 1-6
  Wiring Objects on the Block Diagram ............................................................................. 1-8
  Running a VI ................................................................................................................... 1-9
  Modifying a Signal ........................................................................................................... 1-10
  Displaying Two Signals on a Graph ................................................................................ 1-14
  Customizing a Knob Control ........................................................................................... 1-15
  Customizing a Waveform Graph ..................................................................................... 1-17

Summary .............................................................................................................................. 1-18

Front Panel ......................................................................................................................... 1-18
  Block Diagram ................................................................................................................ 1-18
  Front Panel and Block Diagram Tools ............................................................................. 1-18
  Running and Stopping a VI .............................................................................................. 1-18
  Express VIs ...................................................................................................................... 1-19
  LabVIEW Documentation Resources .......................................................................... 1-19
  Property Dialog Boxes ................................................................................................... 1-19
  Shortcuts .......................................................................................................................... 1-20

Chapter 2
Customizing a VI

Building a VI from a Blank VI .............................................................................................. 2-1
  Opening a Blank VI .......................................................................................................... 2-1
  Adding an Express VI That Simulates a Signal ............................................................... 2-2
  Searching the Help and Modifying a Signal .................................................................... 2-3
  Customizing a User Interface from the Block Diagram .................................................. 2-4
  Configuring a VI to Run Continuously until the User Stops It ....................................... 2-6
  Using the Error List Window ........................................................................................... 2-7
  Controlling the Speed of Execution ................................................................................ 2-8
  Using a Table to Display Data ......................................................................................... 2-9
  Searching for Examples .................................................................................................... 2-10
Contents

Summary ........................................................................................................................... 2-11
Using the LabVIEW Help Resources ............................................................................ 2-11
Customizing the Block Diagram Code ......................................................................... 2-12
  Creating Controls and Indicators ............................................................................ 2-12
  Controlling When a VI Stops Running .................................................................. 2-12
  Errors and Broken Wires ..................................................................................... 2-13
  Displaying Data in a Table .................................................................................... 2-13
Using the NI Example Finder ....................................................................................... 2-13
Shortcuts ....................................................................................................................... 2-14

Chapter 3
Analyzing and Saving a Signal

Building an Analysis VI ............................................................................................... 3-1
  Modifying a VI Created from a Template .............................................................. 3-2
  Adding a Signal ...................................................................................................... 3-3
  Adding Two Signals .............................................................................................. 3-4
  Filtering a Signal .................................................................................................. 3-5
  Modifying the Appearance of Graphs .................................................................... 3-7
  Analyzing the Amplitude of a Signal ..................................................................... 3-7
  Controlling the Speed of Execution ...................................................................... 3-8
  Adding a Warning Light ........................................................................................ 3-9
  Setting a Warning Level Limit ............................................................................. 3-9
  Warning the User .................................................................................................. 3-10
  Configuring a VI to Save Data to a File ............................................................... 3-11
  Saving Data to a File ............................................................................................ 3-12
  Adding a Button That Stores Data When Clicked ............................................... 3-12
  Saving Data When Prompted by a User ............................................................... 3-13
  Viewing Saved Data ............................................................................................. 3-14

Summary ....................................................................................................................... 3-14
  Controls and Indicators ....................................................................................... 3-14
  Filtering Data ........................................................................................................ 3-14
  Saving Data .......................................................................................................... 3-14

Chapter 4
Hardware: Acquiring Data and Communicating with Instruments (Windows)

Hardware and Software Requirements ......................................................................... 4-1
Acquiring a Signal in NI-DAQmx ............................................................................... 4-2
  Creating an NI-DAQmx Task ............................................................................. 4-2
  Graphing Data from a DAQ Device ..................................................................... 4-4
Editing an NI-DAQmx Task ..................................................................................... 4-5
  Visually Comparing Two Voltage Readings ....................................................... 4-6
Communicating with an Instrument: Using Instrument Drivers and the Instrument
I/O Assistant .................................................................................................................. 4-6
  Getting Started with the Instrument Driver Finder .................................................. 4-7
  Finding and Installing Instrument Drivers ............................................................... 4-7
  Using Instrument Drivers ....................................................................................... 4-8
  Selecting an Instrument Using the Instrument I/O Assistant ............................... 4-9
  Acquiring and Parsing Information for an Instrument ............................................. 4-9
  Wiring a Command to an Instrument .................................................................... 4-10
Summary ....................................................................................................................... 4-11
  DAQ Assistant Express VI ..................................................................................... 4-11
  Tasks in NI-DAQmx .................................................................................................. 4-11
  Instrument Drivers ................................................................................................. 4-12
  Instrument I/O Assistant Express VI ....................................................................... 4-12

Chapter 5
Using Other LabVIEW Features
All Controls and Indicators ........................................................................................ 5-1
All VIs and Functions ................................................................................................... 5-1
  VIs .......................................................................................................................... 5-2
  Functions ............................................................................................................... 5-2
Data Types .................................................................................................................. 5-2
When to Use Other LabVIEW Features .................................................................... 5-4

Appendix A
Technical Support and Professional Services
Glossary
Index
About This Manual

Use this manual as a tutorial to familiarize yourself with the LabVIEW graphical programming environment and the basic LabVIEW features you use to build data acquisition and instrument control applications.

This manual contains exercises that you can use to learn how to develop basic applications in LabVIEW. These exercises take a short amount of time to complete and help you get started with LabVIEW.

The end of each chapter includes a summary of the main concepts taught in that chapter. Use these summaries to review what you learned.

Related Documentation

The following documents contain information that you may find helpful as you read this manual:

- LabVIEW Installation Guide—Refer to this guide for information about installing LabVIEW, modules and toolkits, drivers, and hardware.

- LabVIEW Help—Use the LabVIEW Help to access information about LabVIEW programming concepts, step-by-step instructions for using LabVIEW, and reference information about LabVIEW VIs, functions, palettes, menus, tools, properties, methods, events, dialog boxes, and so on. The LabVIEW Help also lists the LabVIEW documentation resources available from National Instruments. Access the LabVIEW Help by selecting Help » LabVIEW Help.

- LabVIEW Quick Reference Card—Use this card as a reference for information about keyboard shortcuts and help resources.
Getting Started with LabVIEW Virtual Instruments

LabVIEW programs are called virtual instruments, or VIs, because their appearance and operation imitate physical instruments, such as oscilloscopes and multimeters. LabVIEW contains a comprehensive set of tools for acquiring, analyzing, displaying, and storing data, as well as tools to help you troubleshoot code you write.

In LabVIEW, you build a user interface, or front panel, with controls and indicators. Controls are knobs, push buttons, dials, and other input mechanisms. Indicators are graphs, LEDs, and other output displays. After you build the front panel, you add code using VIs and structures to control the front panel objects. The block diagram contains this code.

You can use LabVIEW to communicate with hardware such as data acquisition, vision, and motion control devices, as well as GPIB, PXI, VXI, RS232, and RS485 instruments.

Building a Virtual Instrument

In the following exercises, you will build a VI that generates a signal and displays that signal in a graph. After you complete the exercises, the front panel of the VI will look similar to the front panel in the following figure.

You can complete the exercises in this chapter in approximately 40 minutes.

Figure 1-1. Front Panel of the Acquiring a Signal VI
Launching LabVIEW

The Getting Started window appears when you launch LabVIEW. Use this window to create new projects and open existing files. You also can access resources to expand the capability of LabVIEW and information to help you learn about LabVIEW.

The Getting Started window disappears when you open an existing file or create a new file and reappears when you close all open front panels and block diagrams. You also can display the window from the front panel or block diagram by selecting View » Getting Started Window.

Opening a New VI from a Template

LabVIEW provides built-in template VIs that include the subVIs, functions, structures, and front panel objects you need to get started building common measurement applications.

Complete the following steps to create a VI that generates a signal and displays it in the front panel window.

1. Launch LabVIEW.
2. Select File » New to display the New dialog box.
3. From the Create New list, select VI » From Template » Tutorial (Getting Started) » Generate and Display. This template VI generates and displays a signal.

A preview and a brief description of the template VI appear in the Description section. The following figure shows the New dialog box and the preview of the Generate and Display template VI.
4. Click the **OK** button to create a VI from the template. You also can double-click the name of the template VI in the **Create New** list to create a VI from a template.

LabVIEW displays two windows: the front panel window and the block diagram window.

5. Examine the front panel window.

The user interface, or front panel, appears with a gray background and includes controls and indicators. The title bar of the front panel indicates that this window is the front panel for the Generate and Display VI.

**Note** If the front panel is not visible, you can display the front panel by selecting **Window»Show Front Panel**. You also can switch between the front panel window and block diagram window at any time by pressing the `<Ctrl-E>` keys. The `<Ctrl>` key in keyboard shortcuts corresponds to the (Mac OS X) `<Command>` key or (Linux) `<Alt>` key.
Chapter 1  Getting Started with LabVIEW Virtual Instruments

6. Select **Window»Show Block Diagram** and examine the block diagram of the VI. The block diagram appears with a white background and includes VIs and structures that control the front panel objects. The title bar of the block diagram indicates that this window is the block diagram for the Generate and Display VI.

7. On the front panel toolbar, click the **Run** button, shown below. You also can press the <Ctrl-R> keys to run a VI.

A sine wave appears on the graph in the front panel window.

8. Stop the VI by clicking the front panel **STOP** button, shown below.

---

**Adding a Control to the Front Panel**

Front panel controls simulate the input mechanisms on a physical instrument and supply data to the block diagram of the VI. Many physical instruments have knobs you can turn to change an input value.

Complete the following steps to add a knob control to the front panel.

---

**Tip** Throughout these exercises, you can undo recent edits by selecting **Edit»Undo** or pressing the <Ctrl-Z> keys.

1. If the **Controls** palette, shown in Figure 1-3, is not visible in the front panel window, select **View»Controls Palette**.

**Tip** You can right-click any blank space in the front panel or the block diagram to display a temporary version of the **Controls** or **Functions** palette. The **Controls** or **Functions** palette appears with a thumbtack icon in the upper left corner. Click the thumbtack to pin the palette so it is no longer temporary.
2. If you are a new LabVIEW user, the Controls palette opens with the Modern palette, shown in the following figure, visible by default. If you do not see the Modern palette, click Modern on the Controls palette to display the Modern palette.

![Controls Palette](image)

Figure 1-3. Controls Palette

3. Move the cursor over the icons on the Modern palette to locate the Numeric Controls palette.

   When you move the cursor over icons on the Controls palette, the name of the subpalette, control, or indicator appears in a tip strip below the icon.

   **Note** Some palette objects display a short name on the palette that is different from the name that appears in the tip strip. The short name abbreviates the name of the palette object so that it fits in the space available on the palette. If you have difficulty finding a palette object by its short name, use the Search button on the Controls or Functions palette to find the palette object by name.

4. Click the Numeric Controls icon to display the Numeric Controls palette.
5. Click the Knob control on the Numeric Controls palette to attach the control to the cursor, and then add the knob to the front panel to the left of the waveform graph.

   You will use this knob in a later exercise to control the amplitude of a signal.
6. Select File>Save As and save the VI as Acquiring a Signal.vi in an easily accessible location.
Changing a Signal Type

The block diagram has a blue icon labeled Simulate Signal. This icon represents the Simulate Signal Express VI. An Express VI is a component of the block diagram that you can configure to perform common measurement tasks. The Simulate Signal Express VI simulates a sine wave by default.

Complete the following steps to change this signal to a sawtooth wave.

1. Display the block diagram by pressing the <Ctrl-E> keys or by clicking the block diagram. Locate the Simulate Signal Express VI, shown below. The Simulate Signal Express VI simulates a signal based on the configuration that you specify.

2. Right-click the Simulate Signal Express VI and select Properties from the shortcut menu to display the Configure Simulate Signal dialog box. (Mac OS X) Press <Ctrl>-click to perform the same action as right-click.

   **Tip** You also can double-click the Express VI to display the Configure Simulate Signal dialog box.

3. Select Sawtooth from the Signal type pull-down menu. The waveform on the graph in the Result Preview section changes to a sawtooth wave. The Configure Simulate Signal dialog box should appear similar to the following figure.
4. Click the **OK** button to save the current configuration and close the **Configure Simulate Signal** dialog box.

5. Move the cursor over the down arrows at the bottom of the Simulate Signal Express VI. The down arrows indicate you can reveal hidden inputs and outputs by extending the border of the Express VI.

6. When a double-headed arrow appears, shown below, click and drag the border of the Express VI to add two rows. When you release the border, the **Amplitude** input appears. In Figure 1-4, notice that **Amplitude** is an option in the **Configure Simulate Signal** dialog box. When inputs, such as **Amplitude**, appear on the block diagram and in the configuration dialog box, you can configure the inputs in either location.
Wiring Objects on the Block Diagram

To use the knob to change the amplitude of the signal, you must connect two objects on the block diagram.

Complete the following steps to wire the knob to the Amplitude input of the Simulate Signal Express VI.

1. On the block diagram, move the cursor over the Knob terminal, shown below.

The cursor becomes an arrow, or the Positioning tool, shown below. Use the Positioning tool to select, position, and resize objects.

Note: You can resize only loops and structures on the block diagram. Go to the front panel to resize objects you have added to the front panel.

2. Use the Positioning tool to select the Knob terminal and make sure it is to the left of the Simulate Signal Express VI and inside the gray loop, shown below.

The terminals inside the loop are representations of front panel controls and indicators. Terminals are entry and exit ports that exchange information between the front panel and block diagram.

3. Deselect the Knob terminal by clicking a blank space on the block diagram. If you want to use a different tool with an object, you must deselect the object to switch the tool.

4. Move the cursor over the arrow on the Knob terminal, shown below.

The cursor becomes a wire spool, or the Wiring tool, shown below. Use the Wiring tool to wire objects together on the block diagram.
5. When the Wiring tool appears, click the arrow on the **Knob** terminal and then click the arrow on the **Amplitude** input of the Simulate Signal Express VI, shown below, to wire the two objects together.

![Wiring tool diagram](image)

A wire appears and connects the two objects. Data flows along this wire from the Knob terminal to the Express VI.

6. Select **File»Save** to save the VI.

**Running a VI**

Running a VI executes the solution.

Complete the following steps to run the Acquiring a Signal VI.

1. Display the front panel by pressing the <Ctrl-E> keys or by clicking the front panel.

2. Click the **Run** button or press the <Ctrl-R> keys to run the VI.

   To indicate that the VI is running, the **Run** button changes to a darkened arrow, shown below. You can change the value of most controls while a VI runs, but you cannot edit the VI in other ways while the VI runs.

3. Move the cursor over the knob, hold the mouse button down, and turn the knob to adjust the amplitude of the sawtooth wave.

   The amplitude of the sawtooth wave changes as you turn the knob. As you change the amplitude, the cursor displays a tip strip that indicates the numeric value of the knob. The y-axis on the graph scales automatically to account for the change in amplitude.

4. Click the **STOP** button, shown below, to stop the VI.

![STOP button](image)

The **STOP** button stops the VI after the loop completes its current iteration. The **Abort Execution** button, shown below, stops the VI immediately, before the VI finishes the current iteration. Aborting a VI that uses external resources, such as external hardware, might leave the resources in an unknown state by not resetting or releasing them properly. Design the VIs you create with a stop button to avoid this problem.
Chapter 1 Getting Started with LabVIEW Virtual Instruments

Modifying a Signal
Complete the following steps to scale the signal by 10 and display the results in the front panel graph.

1. In the block diagram, use the Positioning tool to click the wire that connects the Simulate Signal Express VI to the Waveform Graph terminal, shown below.

2. Press the <Delete> key to delete this wire.

3. If the Functions palette, shown in the following figure, is not visible, select View» Functions Palette to display it. The Functions palette opens with the Programming palette visible by default. Select the Express palette by clicking Express on the Functions palette.
4. On the **Arithmetic & Comparison** palette, select the Formula Express VI, shown below, and place it on the block diagram between the Simulate Signal Express VI and the **Waveform Graph** terminal. You can move the **Waveform Graph** terminal to the right to make more room between the Express VI and the terminal.

The **Configure Formula** dialog box appears when you place the Express VI on the block diagram. When you place an Express VI on the block diagram, the configuration dialog box for that VI always appears automatically.
Chapter 1     Getting Started with LabVIEW Virtual Instruments

Note If you place an object too close to another object on the block diagram, automatic wiring might wire the two objects together. Delete the wires if the automatic wiring is wrong. To configure automatic wiring, select Tools»Options then select Block Diagram from the Category list. Remove the checkmark from the Enable auto wiring checkbox to turn off automatic wiring.

5. Click the Help button, shown below, in the bottom right corner of the Configure Formula dialog box to display the LabVIEW Help topic for this Express VI.

The Formula help topic describes the Express VI, the configuration dialog box options, and the inputs and outputs of the Express VI. Each Express VI has a corresponding help topic you can access by clicking the Help button in the configuration dialog box or by right-clicking the Express VI and selecting Help from the shortcut menu.

6. In the Formula topic, find the dialog box option whose description indicates that it enters a variable into the formula.

7. Minimize the LabVIEW Help to return to the Configure Formula dialog box.

8. Change the text in the Label column of the dialog box option you read about, shown below, from x1 to Sawtooth to indicate the input value to the Formula Express VI. When you click in the Formula text box at the top of the Configure Formula dialog box, the text changes to match the label you entered.

9. Define the value of the scaling factor by entering *10 after Sawtooth in the Formula text box. You can use the Input buttons in the configuration dialog box or you can use the *, 1, and 0 keyboard buttons to enter the scaling factor. If you use the Input buttons in the configuration dialog box, LabVIEW places the formula input after the Sawtooth input in the Formula text box. If you use the keyboard, click in the Formula text box after Sawtooth and enter the formula you want to appear in the text box. The Configure Formula dialog box should appear similar to the following figure.
Note If you enter an invalid formula in the Formula text box, the Errors LED in the upper right corner turns from green to gray and displays the text Invalid Formula.

10. Click the OK button to save the current configuration and close the Configure Formula dialog box.

11. Move the cursor over the arrow on the Sawtooth output of the Simulate Signal Express VI.

12. When the Wiring tool appears, click the arrow on the Sawtooth output and then click the arrow on the Sawtooth input of the Formula Express VI, shown below, to wire the two objects together.
Chapter 1  Getting Started with LabVIEW Virtual Instruments

13. Use the Wiring tool to wire the Result output of the Formula Express VI to the Waveform Graph terminal.

Examine the wires connecting the Express VIs and terminals. The arrows on the Express VIs and terminals indicate the direction that the data flows along these wires. The block diagram should appear similar to the following figure. Use the block diagram figures as a reference. The arrangement of objects on your block diagram does not need to match the figure exactly.

![Figure 1-7. Block Diagram of the Acquiring a Signal VI](image)

**Tip** You can right-click any wire and select Clean Up Wire from the shortcut menu to have LabVIEW automatically find a route for the wire around existing objects on the block diagram. LabVIEW also routes a wire to decrease the number of bends in the wire. You also can click the Clean Up Diagram button on the block diagram toolbar to have LabVIEW automatically reroute all existing wires and rearrange objects on the block diagram to generate a cleaner look.

14. Press the <Ctrl-S> keys or select File»Save to save the VI.

Displaying Two Signals on a Graph

To compare the signal generated by the Simulate Signal Express VI and the signal modified by the Formula Express VI on the same graph, use the Merge Signals function.

Complete the following steps to display two signals on the same graph.

1. In the block diagram, move the cursor over the arrow on the Sawtooth output of the Simulate Signal Express VI.

2. Use the Wiring tool to wire the Sawtooth output to the Waveform Graph terminal. The Merge Signals function, shown below, appears where the two wires connect.

A function is a built-in execution element, comparable to an operator, function, or statement in a text-based programming language. The Merge Signals function takes the two separate signals and combines them so that both can display on the same graph.
The block diagram should appear similar to the following figure.

**Figure 1-8. Block Diagram Showing the Merge Signals Function**

3. Press the <Ctrl-S> keys or select **File»Save** to save the VI.
4. Return to the front panel, run the VI, and turn the knob control.
   
   The graph plots the original sawtooth wave and the scaled sawtooth wave with 10 times the amplitude, as you specified in the Formula Express VI. The maximum value on the y-axis automatically scales as you turn the knob.
5. Click the **STOP** button to stop the VI.

**Customizing a Knob Control**

The knob control changes the amplitude of the sawtooth wave, so labeling it **Amplitude** accurately describes the behavior of the knob.

Complete the following steps to customize the appearance of the knob.

1. Right-click the front panel knob and select **Properties** from the shortcut menu to display the **Knob Properties** dialog box. Click the **Appearance** tab to display the **Appearance** page.
2. In the **Label** section on the **Appearance** page, delete the label **Knob**, and enter **Amplitude** in the text box.
Chapter 1  Getting Started with LabVIEW Virtual Instruments

The **Knob Properties** dialog box should appear similar to the following figure.

Figure 1-9. Knob Properties Dialog Box

3. Click the **Scale** tab. In the **Scale Style** section, place a checkmark in the **Show color ramp** checkbox.
   The knob in the front panel window updates to reflect these changes.

4. Click the **OK** button to save the current configuration and close the **Knob Properties** dialog box.

5. Save the VI.

6. Reopen the **Knob Properties** dialog box and experiment with other properties of the knob. For example, on the **Scale** page, try changing the colors for the **Marker text color** by clicking the color box.

7. Click the **Cancel** button to avoid applying any changes you made while experimenting. If you want to keep the changes you made, click the **OK** button.
Customizing a Waveform Graph
The waveform graph indicator displays the two signals. To indicate which plot is the scaled signal and which is the simulated signal, you can customize the plots.

Complete the following steps to customize the appearance of the waveform graph indicator.

1. In the front panel window, move the cursor over the top of the plot legend on the waveform graph.
   Though the graph has two plots, the plot legend displays only one plot.
2. When a double-headed arrow appears, shown in the following figure, click and drag the border of the plot legend to add one item to the legend. When you release the mouse button, the second plot name appears.

   ![Figure 1-10. Expanding a Plot Legend](image)

3. Right-click the waveform graph and select Properties from the shortcut menu to display the Graph Properties dialog box.
4. On the Plots page, select Sawtooth from the top pull-down menu. In the Colors section, click the Line color box to display the color picker. Select a new line color.
5. Select Sawtooth (Formula Result) from the top pull-down menu.
6. Place a checkmark in the Do not use waveform names for plot names checkbox. This action lets you edit the labels on the graph.
7. In the Name text box, delete the current label and change the name of this plot to Scaled Sawtooth.
8. Click the OK button to save the current configuration and close the Graph Properties dialog box.
   The plot color and plot legend change.
Chapter 1   Getting Started with LabVIEW Virtual Instruments

9. Reopen the Graph Properties dialog box and experiment with other properties of the graph. For example, on the Scales page, try disabling automatic scaling and changing the minimum and maximum value of the y-axis.
10. Click the Cancel button to avoid applying any changes you made while experimenting. If you want to keep the changes you made, click the OK button.
11. Save and close the VI.

Summary

The following topics are a summary of the main concepts you learned in this chapter.

Front Panel
The front panel is the user interface of a VI. You build the front panel by using controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls and indicators are located on the Controls palette.

Controls are knobs, push buttons, dials, and other input mechanisms. Controls simulate instrument input mechanisms and supply data to the block diagram of the VI.

Indicators are graphs, LEDs, and other displays. Indicators simulate instrument output mechanisms and display data the block diagram acquires or generates.

Block Diagram
The block diagram contains the graphical source code, also known as G code or block diagram code, for how the VI runs. The block diagram code uses graphical representations of functions to control the front panel objects. Front panel objects appear as icon terminals on the block diagram. Wires connect control and indicator terminals to Express VIs, VIs, and functions. Data flows through the wires in the following ways: from controls to VIs and functions, from VIs and functions to indicators, and from VIs and functions to other VIs and functions. The movement of data through the nodes on the block diagram determines the execution order of the VIs and functions. This movement of data is known as dataflow programming.

Front Panel and Block Diagram Tools
The Positioning tool appears when you move the cursor over an object in the front panel window or on the block diagram. The cursor becomes an arrow that you can use to select, position, and resize objects. The Wiring tool appears when you move the cursor over a terminal of a block diagram object. The cursor becomes a spool that you can use to connect objects on the block diagram through which you want data to flow.

Running and Stopping a VI
Running a VI executes the solution of the VI. Click the Run button or press the <Ctrl-R> keys to run a VI. The Run button changes to a darkened arrow to indicate the VI is running. You can stop a VI immediately by clicking the Abort Execution button. However, aborting a VI that uses
external resources might leave the resources in an unknown state. Design the VIs you create with a stop button to avoid this problem. A stop button stops a VI after the VI completes its current iteration.

Express VIs
Use Express VIs located on the Functions palette for common measurement tasks. When you place an Express VI on the block diagram, the dialog box you use to configure that Express VI appears by default. Set the options in this configuration dialog box to specify how the Express VI behaves. You also can double-click an Express VI or right-click an Express VI and select Properties from the shortcut menu to display the configuration dialog box. If you wire data to an Express VI and run it, the Express VI displays real data in the configuration dialog box. If you close and reopen the Express VI, the VI displays sample data in the configuration dialog box until you run the VI again.

Express VIs appear on the block diagram as expandable nodes with icons surrounded by a blue field. You can resize an Express VI to display its inputs and outputs. The inputs and outputs you can display for the Express VI depend on how you configure the VI.

LabVIEW Documentation Resources
The LabVIEW Help contains information about LabVIEW programming concepts, step-by-step instructions for using LabVIEW, and reference information about LabVIEW VIs, functions, palettes, menus, tools, properties, methods, events, dialog boxes, and so on. The LabVIEW Help also lists the LabVIEW documentation resources available from National Instruments. To access help information for Express VIs, click the Help button in the configuration dialog box while you configure an Express VI. You also can access the LabVIEW Help by right-clicking a VI or function on the block diagram or on a pinned palette and selecting Help from the shortcut menu or by selecting Help»LabVIEW Help.

After you install a LabVIEW add-on such as a toolkit, module, or driver, the documentation for that add-on appears in the LabVIEW Help or appears in a separate help system you access by selecting Help»Add-On Help, where Add-On Help is the name of the separate help system for the add-on.

Property Dialog Boxes
Use property dialog boxes or shortcut menus to configure how controls and indicators appear or behave in the front panel window. Right-click a control or indicator on the front panel and select Properties from the shortcut menu to access the property dialog box for that object. You cannot access property dialog boxes for a control or indicator when a VI is running.
Chapter 1  Getting Started with LabVIEW Virtual Instruments

Shortcuts
This chapter introduced the following keyboard shortcuts.

Note The <Ctrl> key in shortcuts corresponds to the (Mac OS X) <Command> key or (Linux) <Alt> key.

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Ctrl-R&gt;</td>
<td>Runs a VI.</td>
</tr>
<tr>
<td>&lt;Ctrl-Z&gt;</td>
<td>Undoes the last action.</td>
</tr>
<tr>
<td>&lt;Ctrl-E&gt;</td>
<td>Switches between the block diagram and the front panel window.</td>
</tr>
<tr>
<td>&lt;Ctrl-S&gt;</td>
<td>Saves a VI.</td>
</tr>
</tbody>
</table>
Customizing a VI

You can choose one of many LabVIEW template VIs to use as a starting point when building VIs. However, sometimes you need to build a VI for which a template is not available. This chapter teaches you how to build and customize a VI without using a template.

Building a VI from a Blank VI

In the following exercises, you will open a blank VI and add structures and Express VIs to the block diagram to build a new VI. You will build a VI that generates a signal, reduces the number of samples in the signal, and displays the resulting data in a front panel table. After you complete the exercises, the front panel of the VI will look similar to the front panel in the following figure.

You can complete the exercises in this chapter in approximately 45 minutes.

Opening a Blank VI

If no template is available for the VI you want to build, you can start with a blank VI and add Express VIs to accomplish a specific task.
Complete the following steps to open a blank VI.

1. In the Getting Started window, click the Create Project button to display the Create Project dialog box.
   The Create Project dialog box provides common starting points for LabVIEW projects.

2. Select Blank VI from the list of items and click Finish.
   A blank front panel window and block diagram window appear.

3. Display the block diagram.

4. If the Functions palette is not visible, right-click any blank space on the block diagram to display a temporary version of the Functions palette. Click the thumbtack, shown below, in the upper left corner of the Functions palette to pin the palette so it is no longer temporary.

Adding an Express VI That Simulates a Signal

Complete the following steps to find the Express VI you want to use and add it to the block diagram.

1. Select Help»Show Context Help from the front panel or the block diagram to display the Context Help window, shown in Figure 2-2. You also can click the Show Context Help Window button, shown below, on the front panel or block diagram toolbar to display the Context Help window.

2. On the Functions palette, select the Express»Input palette and move the cursor over one of the Express VIs on the Input palette.
   When you move the cursor over a VI, the Context Help window displays information about that VI.
3. Use the information that appears in the **Context Help** window to find the Express VI that can simulate a sine wave signal. Keep the **Context Help** window open. The context help provides useful information as you complete the rest of this exercise.

4. Select the Express VI that can simulate a sine wave signal and place it on the block diagram. The **Configure Simulate Signal** dialog box appears.

5. Move the cursor over the various options in the **Configure Simulate Signal** dialog box, such as **Frequency (Hz)** and **Amplitude**. Read the information that appears in the **Context Help** window.

6. Configure the Simulate Signal Express VI to generate a sine wave with a frequency of 10.7 Hz and an amplitude of 2. The signal in the **Result Preview** window changes to reflect the configured sine wave.

7. Click the **OK** button to save the current configuration and close the **Configure Simulate Signal** dialog box.

8. Move the cursor over the Simulate Signal Express VI and read the information that appears in the **Context Help** window. The **Context Help** window displays information about how you configured the Simulate Signal Express VI in addition to the standard context help description.

9. Save the VI as **Reduce Samples.vi** in an easily accessible location.

### Searching the Help and Modifying a Signal

Complete the following steps to use the **LabVIEW Help** to search for the Express VI that reduces the number of samples in a signal.

1. On the block diagram, move the cursor over the Simulate Signal Express VI to display the **Context Help** window and click the **Detailed help** link to display the **Simulate Signal Express VI** topic in the **LabVIEW Help**. You might have to enlarge or scroll down in the **Context Help** window to see the **Detailed help** link.

   You also can access the **LabVIEW Help** by right-clicking a VI or function on the block diagram or on a pinned palette and selecting **Help** from the shortcut menu or by selecting **Help»LabVIEW Help**.

2. Click the **Search** tab, enter **sample compression** in the **Type in the word(s) to search for** text box, and press the <Enter> key. You can place quotation marks around the phrase to search for the exact phrase. For example, you can enter "**sample compression**" to narrow the search results.

   This word choice reflects what you want this Express VI to do—compress, or reduce, the number of samples in a signal.

   You also can click the **Index** tab to search keywords and concepts.

3. Double-click the **Sample Compression** topic in the search results to display the topic that describes the Sample Compression Express VI.

4. After you read the description of the Express VI, return to the block diagram.
5. Press the <Ctrl-Space> keys to open the Quick Drop dialog box. (Mac OS X) Press the <Command-Shift-Space> keys.

When you know which item you want to add to the block diagram or front panel, you can use the Quick Drop dialog box to quickly find the item and add it to the block diagram or front panel.

**Tip** You can speed up the initial launch of the Quick Drop dialog box by enabling the Load palettes during launch radio button on the Controls/Functions Palette page of the Options dialog box. Selecting to load palettes while LabVIEW launches might slow down the launching of LabVIEW slightly.

6. Enter Sample Compression into the Quick Drop dialog box, press <Enter>, and place the Sample Compression Express VI on the block diagram.

7. Configure the Sample Compression Express VI to reduce the signal by a factor of 25, and set the reduction method to Mean.

8. Click the OK button to save the current configuration and close the Configure Sample Compression dialog box.

9. Use the Wiring tool to wire the Sine output of the Simulate Signal Express VI to the Signals input of the Sample Compression Express VI.

### Customizing a User Interface from the Block Diagram

In the previous exercises, you added controls and indicators to the front panel using the Controls palette. You also can create controls and indicators from the block diagram.

Complete the following steps to create controls and indicators from the block diagram.

1. On the block diagram, right-click the Mean output of the Sample Compression Express VI and select Create Numeric Indicator from the shortcut menu to create a numeric indicator. A Mean indicator shown below, appears on the block diagram.

2. Right-click the Mean output of the Sample Compression Express VI and select Insert Input/Output from the shortcut menu to insert the Enable input.

In previous exercises, you learned to add inputs and outputs by expanding the Express VI using the down arrows. Using the shortcut menu is a different way to display and select the inputs and outputs of an Express VI.
3. Right-click the **Enable** input and select **Create Control** from the shortcut menu to create a switch. A Boolean control, shown below, appears on the block diagram.

Control terminals have a thicker border than indicator terminals. Also, an arrow appears on the right of the terminal if the terminal is a control, and an arrow appears on the left of the terminal if the terminal is an indicator.

4. Right-click the wire that connects the **Sine** output of the Simulate Signal Express VI to the **Signals** input of the Sample Compression Express VI and select **Create Graph Indicator** from the shortcut menu.

5. Use the Wiring tool to wire the **Mean** output of the Sample Compression Express VI to the **Sine** graph indicator.

The Merge Signals function appears.

6. Arrange the objects on the block diagram so they appear similar to the following figure.

**Figure 2-3. Block Diagram of the Reduce Samples VI**

7. Display the front panel.

The controls and indicators you added appear in the front panel with labels that correspond to the inputs and outputs from which you created the controls and indicators.

- **Note** You might need to scroll or resize the front panel to see all controls and indicators.

8. Save the VI.
Configuring a VI to Run Continuously until the User Stops It

In the current state, the VI runs once, generates one signal, and then stops running. To run the VI until a condition occurs, you can use a While Loop.

Complete the following steps to add a While Loop to the block diagram.

1. Display the front panel and run the VI.
   The VI runs once and then stops. The front panel does not have a stop button.
2. Display the block diagram.
3. Click the Search button, shown below, on the Functions palette, and enter While in the text box. LabVIEW searches as you type the first few letters and displays any matches in the search results text box.

If there are objects with the same name, use the information in the brackets to the right of each object name to decide which object to select. Some objects are located on multiple palettes because you can use them for multiple applications.

4. Double-click While Loop <<Execution Control>> to display the Execution Control subpalette and temporarily highlight the While Loop on the subpalette.
5. Select the While Loop on the Execution Control palette.
6. Move the cursor to the upper left corner of the block diagram. Click and drag the cursor diagonally to enclose all the Express VIs and wires, as shown in the following figure.

Figure 2-4. Placing the While Loop around the Express VIs
7. Release the mouse to place the While Loop around the Express VIs and wires.
   The While Loop, shown below, appears with a STOP button wired to the conditional terminal. This While Loop is configured to stop when the user clicks the STOP button.

8. Display the front panel and run the VI.
   The VI now runs until you click the STOP button. A While Loop executes the VIs and functions inside the loop until the user clicks the STOP button.

9. Click the STOP button and save the VI.

Using the Error List Window
If a VI contains an indicator you do not want to use, you can delete it.

Complete the following steps to remove the Mean indicator from the front panel.
1. Display the front panel and move the cursor over the Mean indicator until the Positioning tool appears.
2. Click the Mean indicator, shown below, to select it and press the <Delete> key.

3. Display the block diagram.
   A wire appears as a dashed black line with a red x in the middle, shown below. The dashed black line is a broken wire. The Run button, shown below, appears broken to indicate the VI cannot run.

4. Click the broken Run button to display the Error list window.
   The Error list window lists all errors in the VI and provides details about each error. You can use the Error list window to locate errors.
5. In the errors and warnings list, select the Wire: has loose ends error and click the Help button to display more information about the error.

   Tip  You also can move the Wiring tool over a broken wire to display a tip strip that describes why the wire is broken. This information also appears in the Context Help window when you move the Wiring tool over a broken wire.
In the **errors and warnings** list, double-click the **Wire: has loose ends** error to highlight the broken wire.

7. Press the <Ctrl-B> keys to delete the broken wire. Pressing the <Ctrl-B> keys deletes all broken wires on the block diagram. You can press the <Delete> key to delete only the selected wire.

8. Select View»Error List to display the **Error list** window. No errors appear in the **errors and warnings** field.

   **Tip** You also can press the <Ctrl-L> keys to display the **Error list** window.

9. Click the **Close** button to close the **Error list** window. The **Run** button no longer appears broken.

### Controlling the Speed of Execution

To plot the points on the waveform graph more slowly, you can add a time delay to the block diagram.

Complete the following steps to control the speed at which the VI runs.

1. On the block diagram, search for the Time Delay Express VI, shown below, on the **Functions** palette and place it inside the While Loop.

   ![Time Delay Express VI](image)

   You can use the Time Delay Express VI to control the execution rate of the VI.

2. Enter `0.25` in the **Time delay (seconds)** text box. This time delay specifies how fast the loop runs. With a 0.25 second time delay, the loop iterates once every quarter of a second.

3. Click the **OK** button to save the current configuration and close the **Configure Time Delay** dialog box.

4. Display the front panel and run the VI.

5. Click the **Enable** switch and examine the change on the graph. If the **Enable** switch is on, the graph displays the reduced signal. If the **Enable** switch is off, the graph does not display the reduced signal.

6. Click the **STOP** button to stop the VI.
Using a Table to Display Data

Complete the following steps to display a collection of mean values in a front panel table.

1. Display the front panel.
2. On the Controls palette search for the Express Table indicator and add it to the right of the waveform graph.
3. Display the block diagram.
   LabVIEW wired the Table terminal to the Build Table Express VI.
4. If the Build Table Express VI and the Table terminal are not selected already, click an open area on the block diagram to the left of the Build Table Express VI and the Table terminal. Drag the cursor diagonally until the selection rectangle encloses the Build Table Express VI and the Table terminal, shown below.

   ![Diagram of selection process](image)

   A moving dashed outline, called a marquee, highlights the Build Table Express VI, the Table terminal, and the wire joining the two.
5. Drag the objects into the While Loop to the right of the Sample Compression Express VI. If you drag objects near the border of the While Loop, the loop resizes to enclose the Build Table Express VI and the Table terminal after you release the mouse button.
6. Use the Wiring tool to wire the Mean output of the Sample Compression Express VI to the Signals input of the Build Table Express VI.
7. The block diagram should appear similar to the following figure.

   ![Block Diagram of the Reduce Samples VI](image)
Chapter 2 Customizing a VI

8. Display the front panel and arrange the controls and indicators as shown in Figure 2-1.
9. Run the VI.
10. Click the Enable switch.
   If the Enable switch is on, the table displays the mean values of every 25 samples of the
   sine wave. If the Enable switch is off, the table does not record the mean values.
11. Stop the VI.
12. Experiment with properties of the table by using the Table Properties dialog box. For
    example, try changing the number of columns to one.
13. Save and close the VI.

Searching for Examples
To learn more about how you can use a certain VI, you can search for and view an example that
uses the VI.

Complete the following steps to find and open an example that uses the Amplitude and Level
Measurements Express VI.
1. Select Help » LabVIEW Help to display the LabVIEW Help.
2. Click the Search tab. In the Type in the word(s) to search for text box enter amplitude
   and level measurements express VI and press the <Enter> key. (Mac OS X and
   Linux) Choose the Full Text option in the Search Options section of the Search the
   LabVIEW Help dialog box to narrow the search results.

   Tip (Windows) Before you search, you can narrow the search results by placing a
   checkmark in the Search titles only checkbox near the bottom of the help window.
   You also can use operators such as AND, OR, and NEAR in the Type in the word(s) to
   search for text box to narrow the search results.
3. (Windows) Click the Location column header to sort the search results by content type.
   Reference topics contain reference information about LabVIEW objects such as VIs,
   functions, palettes, menus, and tools. How-To topics contain step-by-step instructions for
   using LabVIEW. Concept topics contain information about LabVIEW programming
   concepts.

   Tip You can use the Favorites tab of the LabVIEW Help to save and quickly access
   help topics you use often. When you view a help topic you may want to access later,
   navigate to the Favorites tab and click the Add button.
4. Double-click the Amplitude and Level Measurements Express VI search result to
display the reference topic that describes the Amplitude and Level Measurements Express
VI.
5. After you read the description of the Express VI, click the Open example button in the Example section near the bottom of the topic to open an example that uses the Amplitude and Level Measurements Express VI.

6. Run the VI and move the vertical pointer slides. The amplitude and frequency of the signal change as you move the vertical pointer slides.

7. Stop the VI.

8. Select Window»Show Block Diagram and read the block diagram comments.

9. Close the example VI and return to the Amplitude and Level Measurements Express VI topic in the LabVIEW Help.

10. Click the Find related examples button to open the NI Example Finder and display a list of examples similar to the example that uses this VI. The NI Example Finder searches among hundreds of examples, including all installed examples and the examples located on the NI Developer Zone at ni.com/zone. You can modify an example to fit an application, or you can copy and paste from one or more examples into a VI that you create.

   **Note** Always select File»Save As when you save a modified example to avoid overwriting the example program in the NI Example Finder.

   You also can right-click a VI or function on the block diagram or on a pinned palette and select Examples from the shortcut menu to display a help topic with links to examples for that VI or function. To launch the NI Example Finder and browse or search examples, select Help»Find Examples.

   **Note** Not all VIs have an example.

11. After you experiment with the NI Example Finder and the example VIs, close the NI Example Finder.

**Summary**

The following topics are a summary of the main concepts you learned in this chapter.

**Using the LabVIEW Help Resources**

In this chapter, you learned to use the help resources in the following ways:

- The Context Help window displays basic information about LabVIEW objects when you move the cursor over each object. Objects with context help information include VIs, functions, structures, palettes, dialog box components, and so on. To access the Context Help window, select Help»Show Context Help or press the <Ctrl-H> keys. (Mac OS X) Press the <Command-Shift-H> keys.

- When you move the cursor over an Express VI on the block diagram, the Context Help window displays a brief description of the Express VI and information about how you configured the Express VI.
Chapter 2 Customizing a VI

• The LabVIEW Help contains detailed information about LabVIEW objects. To access the LabVIEW Help topic for an object, move the cursor over the object and click the Detailed help link in the Context Help window. You also can right-click an object on the block diagram or on a pinned palette and select Help from the shortcut menu.

• To navigate the LabVIEW Help, use the Contents, Index, and Search tabs. Use the Contents tab to get an overview of the topics and structure of the help. Use the Index tab to find a topic by keyword. Use the Search tab to search the help for a word or phrase.

• If you find an object in the LabVIEW Help you want to use, you can click an Add to the block diagram button to place the object on the block diagram.

• On the Search tab of the LabVIEW Help, use operators such as AND, OR, and NEAR to narrow the search results. To search for an exact phrase, place quotation marks around the phrase. Before you search, you also can narrow the search results by placing a checkmark in the Search titles only checkbox near the bottom of the help window.

• On the Search tab of the LabVIEW Help, you can click the Location column header above the list of search results to sort the results by content type. Reference topics contain reference information about LabVIEW objects such as VIs, functions, palettes, menus, and tools. How-To topics contain step-by-step instructions for using LabVIEW. Concept topics contain information about LabVIEW programming concepts.

Customizing the Block Diagram Code

You can use many controls, indicators, Express VIs, and structures to customize a VI. The following examples review a few common ways to customize VIs, including creating controls and indicators, configuring when a VI stops, correcting broken wires, and displaying data in a table.

Creating Controls and Indicators

Create controls and indicators on the block diagram by right-clicking the Express VI input, output, or wire, selecting Create from the shortcut menu, and selecting among the available options. LabVIEW wires the control or indicator you created to the input, output, or wire you right-clicked.

Control terminals have a thicker border than indicator terminals. Also, an arrow appears on the right of the terminal if the terminal is a control, and an arrow appears on the left of the terminal if the terminal is an indicator.

Controlling When a VI Stops Running

Use a While Loop to run the code enclosed within the loop continually. A While Loop stops running when a stop condition occurs. After you place or move an object in a While Loop near the border, the loop resizes to add space for that object.

The Execution Control palette includes objects you can use to control the number of times a VI runs, as well as the speed at which the VI runs.
Errors and Broken Wires

The Run button appears broken when the VI you are creating or editing contains errors. If the Run button is still broken when you finish wiring the block diagram, the VI is broken and cannot run.

Click the broken Run button or select View » Error List to find out why a VI is broken. You can use the Error list window to locate errors. Click the Help button for more information about the error. Double-click the error in the errors and warnings field to highlight the problem causing the error.

A broken wire appears as a dashed black line with a red X in the middle. Broken wires occur for a variety of reasons, such as if you delete wired objects. The VI cannot run if the block diagram contains broken wires.

Move the Wiring tool over a broken wire to display a tip strip that describes why the wire is broken. This information also appears in the Context Help window when you move the Wiring tool over a broken wire. Right-click the wire and select List Errors from the shortcut menu to display the Error list window. Click the Help button for more information about why the wire is broken.

Displaying Data in a Table

The table indicator displays generated data. Use the Build Table Express VI to build a table of generated data.

Using the NI Example Finder

Use the NI Example Finder to browse or search examples installed on your computer or on the NI Developer Zone at ni.com/zone. These examples demonstrate how to use LabVIEW to perform a wide variety of test, measurement, control, and design tasks. Select Help » Find Examples to launch the NI Example Finder.

Examples can show you how to use specific VIs or functions. You can right-click a VI or function on the block diagram or on a pinned palette and select Examples from the shortcut menu to display a help topic with links to examples for that VI or function. You can modify an example VI to fit an application, or you can copy and paste from one or more examples into a VI that you create.

Always select File » Save As when you save a modified example to avoid accidentally overwriting the example program in the NI Example Finder.
Chapter 2  Customizing a VI

Shortcuts
This chapter introduced the following keyboard shortcuts.

Note The <Ctrl> key in these shortcuts corresponds to the (Mac OS X) <Command> key or (Linux) <Alt> key.

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Ctrl-N&gt;</td>
<td>Opens a new, blank VI.</td>
</tr>
<tr>
<td>&lt;Ctrl-H&gt;</td>
<td>Shows or hides the Context Help window.</td>
</tr>
<tr>
<td></td>
<td>(Mac OS X) Press the &lt;Command-Shift-H&gt; keys.</td>
</tr>
<tr>
<td>&lt;Ctrl-Space&gt;</td>
<td>Displays the Quick Drop dialog box.</td>
</tr>
<tr>
<td></td>
<td>(Mac OS X) Press the &lt;Command-Shift-Space&gt; keys.</td>
</tr>
<tr>
<td>&lt;Ctrl-B&gt;</td>
<td>Deletes all broken wires in a VI.</td>
</tr>
<tr>
<td>&lt;Ctrl-L&gt;</td>
<td>Displays the Error list window.</td>
</tr>
</tbody>
</table>
LabVIEW includes a set of Express VIs that help you analyze signals. This chapter teaches you how to use LabVIEW to perform a basic analysis of a signal and how to save the analyzed data to a file.

**Note**  The exercises in this chapter use Express VIs that are available only in the LabVIEW Full and Professional Development Systems.

### Building an Analysis VI

In the following exercises, you will build a VI that generates a signal, filters the signal, indicates if the signal exceeds a certain limit, and records the data. After you complete the exercises, the front panel of the VI will look similar to the front panel in the following figure.

You can complete the exercises in this chapter in approximately 40 minutes.

**Figure 3-1. Front Panel of the Save Data VI**
Chapter 3  Analyzing and Saving a Signal

Modifying a VI Created from a Template

Complete the following steps to create a VI that generates, analyzes, and displays a signal.

1. Select File»New to display the New dialog box.
2. From the Create New list, select VI»From Template»Tutorial (Getting Started)»Generate, Analyze, and Display. This template VI simulates a signal and analyzes it for its root mean square (RMS) value.
3. Click the OK button or double-click the name of the template to create a VI from the template.
4. If the Context Help window is not visible, press the <Ctrl-H> keys to display the window. (Mac OS X) Press the <Command-Shift-H> keys.
5. Display the block diagram by pressing the <Ctrl-E> keys.
6. Move the cursor over the Amplitude and Level Measurements Express VI, shown below.

The Context Help window displays information about the behavior of the Express VI. Keep the Context Help window open. It will provide useful information as you complete the rest of this exercise.

7. Display the front panel and remove the RMS indicator, shown below.

You will not use the RMS functionality of the Amplitude and Level Measurements Express VI for this exercise. However, you can use the Generate, Analyze, and Display template VI with the RMS functionality in the future to reduce development time.

8. Display the block diagram and remove any broken wires that result from removing the RMS indicator. To remove all broken wires from the block diagram, you can press the <Ctrl-B> keys.
9. Then return to the front panel window and right-click the waveform graph indicator. Select Properties from the shortcut menu. The Graph Properties dialog box appears.
10. On the Appearance page, place a checkmark in the Visible checkbox in the Label section and enter Unfiltered Signal in the text box.
11. Click the OK button to save the configuration and close the Graph Properties dialog box.
12. Run the VI.
    The signal appears in the graph.
13. Click the STOP button to stop the VI.

Adding a Signal

The Simulate Signal Express VI simulates a sine wave by default. You can customize the simulated signal by changing the options in the Configure Simulate Signal dialog box.

Complete the following steps to create an additional simulated signal that adds uniform white noise to the sine wave.

1. On the block diagram, use the Positioning tool to select the Simulate Signal Express VI. Hold down the <Ctrl> key and click and drag to create an additional Simulate Signal Express VI on the block diagram. (Mac OS X) Hold down the <Option> key and drag. (Linux) You also can hold down the middle mouse button and drag.
2. Release the mouse button to place the copied Simulate Signal Express VI below the original Simulate Signal Express VI. LabVIEW updates the name of the copied Simulate Signal Express VI to Simulate Signal2.
3. Double-click the Simulate Signal2 Express VI to display the Configure Simulate Signal dialog box.
4. Select Sine from the Signal type pull-down menu.
5. Enter 60 in the Frequency (Hz) text box.
6. Enter 0.1 in the Amplitude text box.
7. Place a checkmark in the Add noise checkbox to add noise to the sine signal.
8. Select Uniform White Noise from the Noise type pull-down menu.
9. Enter 0.1 in the Noise amplitude text box.
10. Enter -1 in the Seed number text box.
11. In the Timing section, select the Run as fast as possible option.
12. In the Signal Name section, remove the checkmark from the Use signal type name checkbox.
13. Enter 60 Hz and Noise in the Signal name text box.

When you change the signal name in the Configure Simulate Signal dialog box, LabVIEW changes the name of the signal output on the block diagram. Changing the signal name makes it easier for you to identify the signal type when you view the Express VI on the block diagram.

The Result Preview section displays a random signal. The Configure Simulate Signal dialog box should appear similar to the following figure.
Adding Two Signals

To add two signals together to create one signal, you can use the Formula Express VI. Rather than displaying two signals on one graph, the Formula Express VI adds both signals together to create a single signal on the graph. You can use this Express VI to add noise to a signal.

Complete the following steps to add the 60 Hz and Noise signal to the Sine signal.

1. In the block diagram window, triple-click the wire that connects the Sine output of the Simulate Signal Express VI to the Signals input of the Amplitude and Level Measurements Express VI and to the Unfiltered Signal indicator. Remove the wire.

2. On the Functions palette, click the Search button to search for the Formula Express VI, shown below, and add it to the block diagram between the Simulate Signal Express VIs and the Amplitude and Level Measurements Express VI. The Configure Formula dialog box appears.

14. Click the OK button to save the current configuration and close the Configure Simulate Signal dialog box.
3. In the **Label** column, change the label for **X1** to **Sine** and the label for **X2** to **60 Hz and Noise**.
   The Formula Express VI automatically enters the first input, **Sine**, in the **Formula** text box.
4. Click the + button and then the **X2** button to add **Sine** and **60 Hz and Noise** together in the **Formula** text box.
5. Click the **OK** button to save the current configuration and close the **Configure Formula** dialog box.
6. Use the Wiring tool to wire the **Sine** output of the Simulate Signal Express VI to the **Sine** input of the Formula Express VI.
7. Wire the **60 Hz and Noise** output of the Simulate Signal2 Express VI to the **60 Hz and Noise** input of the Formula Express VI.
8. Wire the **Result** output of the Formula Express VI to the **Unfiltered Signal** indicator and to the **Signals** input of the Amplitude and Level Measurements Express VI.
9. Display the front panel by pressing the `<Ctrl-E>` keys.
10. Run the VI.
    The signal with added noise appears in the graph.
11. Click the **STOP** button to stop the VI.
12. Select **File»Save As** and save the VI as **Analysis.vi** in an easily accessible location.

### Filtering a Signal

You can use the Filter Express VI to process signals through filters and windows.

Complete the following steps to configure the Filter Express VI to filter the signal using an infinite impulse response (IIR) filter.

1. Display the block diagram window and remove the wire that connects the **Result** output of the Formula Express VI to the **Signals** input of the Amplitude and Level Measurements Express VI.
2. Remove all broken wires that result from removing the wire.
3. Search for the Filter Express VI, shown below, and add it to the block diagram between the Simulate Signal2 Express VI and the Amplitude and Level Measurements Express VI. The Configure Filter dialog box appears.

4. In the Filter Specifications section, change the Cutoff Frequency (Hz) to 25.
5. Click the OK button to save the configuration and close the Configure Filter dialog box.
6. Display the front panel.
7. Click the Unfiltered Signal waveform graph indicator and press the <Ctrl> key while you drag with the Positioning tool to create an additional waveform graph indicator.
8. Triple-click the Unfiltered Signal 2 label above the new waveform graph indicator and enter Filtered Signal to change the label of the indicator. You also can change the label on the Appearance page of the Graph Properties dialog box.
9. On the block diagram, wire the Result output of the Formula Express VI to the Signal input of the Filter Express VI.
10. Wire the Filtered Signal output of the Filter Express VI to the Signals input of the Amplitude and Level Measurements Express VI and to the input of the Filtered Signal waveform graph indicator.
11. Select File » Save. The block diagram of the Analysis VI should appear similar to the following figure.

Figure 3-3. Block Diagram of the Analysis VI
Modifying the Appearance of Graphs

You can use the Display Format page of the Graph Properties dialog box to specify how the scales of the x-axis and y-axis appear on the graph.

Complete the following steps to change the format of the x-axis and y-axis of the Unfiltered Signal and Filtered Signal graphs.

1. In the front panel window, right-click the Unfiltered Signal graph indicator and select Properties from the shortcut menu. The Graph Properties dialog box appears.
2. On the Display Format page, select Time (X-Axis) from the top pull-down menu.
3. Select the Default editing mode option.
4. In the Type list, select Automatic formatting.
5. In the Digits field, enter 6 and select Significant digits from the Precision Type pull-down menu.
6. Place a checkmark in the Hide trailing zeros checkbox.
7. Select Amplitude (Y-Axis) from the top pull-down menu and repeat steps 3 through 6 so the y-axis configuration matches the x-axis configuration.
9. Remove the checkmark from the Autoscale checkbox.
10. Enter -2.5 in the Minimum text box and 2.5 in the Maximum text box.
11. Click the OK button to save the configuration and close the Graph Properties dialog box.
12. Repeat steps 1 through 11 to configure the Filtered Signal graph indicator.

The x-axis and the y-axis on the Unfiltered Signal and Filtered Signal graph indicators change to reflect the new configuration.

Analyzing the Amplitude of a Signal

You can use the Amplitude and Level Measurements Express VI to analyze the voltage characteristics of a signal.

Complete the following steps to reconfigure the Express VI to measure the peak-to-peak amplitude values of the signal.

1. On the block diagram, double-click the Amplitude and Level Measurements Express VI to display the Configure Amplitude and Level Measurements dialog box.
2. In the Amplitude Measurements section, remove the checkmark from the RMS checkbox.
3. Place a checkmark in the Peak to peak checkbox. Peak to peak appears in the Results section with the corresponding value of the measurement.
Chapter 3  Analyzing and Saving a Signal

4. Click the OK button to save the current configuration and close the Configure Amplitude and Level Measurements dialog box.

The RMS output of the Amplitude and Level Measurements Express VI changes to reflect the new Peak to Peak output, shown below.

![Output Image]

You will use the Peak to Peak output in a later exercise.

Controlling the Speed of Execution

To plot the points on the waveform graphs more slowly, you can add a time delay to the block diagram. A time delay slows the speed at which a VI runs.

Complete the following steps to control the speed at which the VI runs.

1. On the block diagram, search for the Time Delay Express VI.
2. Place the Time Delay Express VI inside the While Loop. The Configure Time Delay dialog box appears.
3. Enter 1.000 in the Time delay (seconds) text box and click the OK button.
4. Display the front panel and run the VI. The VI runs more slowly.
   The loop iterates once every second.
5. Stop the VI.

Another way to control the speed of the VI is to alter the rate of data acquisition. On the block diagram, double click the Simulate Signal Express VI to display the Configure Simulate Signal dialog box. Locate the Timing section in the dialog box. The Timing section contains a number of ways to alter the rate of data acquisition and the speed at which a VI runs.

For example, one of the default settings of the VI is Simulate Acquisition Timing. This means that the VI mimics the acquisition rate of a hardware device. You can select Run as fast as possible to display data more quickly. In the Samples per second (Hz) text box, the default value is 1000, while the default value in the Number of Samples text box is 100. This means that the VI will output 100 data points spanning 0.1 second. You can change these values to change the amount of data the VI displays, as well as the rate at which the VI displays the data.
Adding a Warning Light

If you want a visual cue to indicate when a value exceeds a specified limit, you can use a warning light.

Complete the following steps to add a warning light to the VI.

1. Display the Controls palette by right-clicking any blank space in the front panel window.
2. On the Express palette, select the LEDs palette.
3. Select the round LED indicator and add it to the front panel to the left of the waveform graphs.
4. Double-click the Boolean label above the LED and enter Warning to change the label of the LED.
   You will use this LED in a later exercise to indicate when a value has exceeded its limit.
5. Select File » Save As to display the Save As dialog box.
6. Read the various dialog box options. Select the Copy and Substitute copy for original options to create a copy of the original VI and immediately edit the copy.
7. Click the Continue button and save the VI as Warning Light.vi in an easily accessible location.

Setting a Warning Level Limit

To specify the value at which you want the LED to light, use the Comparison Express VI.

Complete the following steps to compare the peak-to-peak value to a limit you set.

1. On the block diagram, search for the Comparison Express VI and place it to the right of the Amplitude and Level Measurements Express VI. The Configure Comparison dialog box appears.
2. In the Compare Condition section, select the > Greater option.
3. In the Comparison Inputs section, select Value and enter 2 in the Value numeric control to assign a constant value at which you want the LED to light.
4. Click the OK button to save the current configuration and close the Configure Comparison dialog box.
   The name of the Comparison Express VI changes to reflect the operation of the Express VI, shown below. Greater indicates that the Express VI does a greater than comparison.
5. Wire the Peak to Peak output of the Amplitude and Level Measurements Express VI to the Operand 1 input of the Greater Express VI.
6. Move the cursor over the wire that connects the Peak to Peak output to the Operand 1 input.
7. When the Positioning tool appears, right-click the wire that connects the Peak to Peak output to the Operand 1 input and select Create Numeric Indicator from the shortcut menu.

A Peak to Peak terminal, shown below, appears on the block diagram. If the Peak to Peak terminal appears to be on top of the wires between the Express VIs, move the Express VIs and Peak to Peak terminal around to create more space. For example, move the Peak to Peak terminal into blank space above the Express VIs.

Tip You also can press <Ctrl> and drag a rectangle in open space to add more working space to the front panel or block diagram.

Warning the User

After you specify the values at which you want the LED to light, you must wire the LED to the Greater Express VI.

Complete the following steps to provide a visual cue when the peak-to-peak value of the signal exceeds a specified limit.

1. In the block diagram window, move the Warning terminal to the right of the Greater Express VI. Make sure the Warning terminal is inside the While Loop, as shown in the following figure.

Figure 3-4. Block Diagram of the Warning Light VI
2. Wire the **Result** output of the Greater Express VI to the **Warning** terminal. The block diagram should appear similar to Figure 3-4.

- **Note**: Red coercion dots appear automatically at the **Peak to Peak** and **Warning** input terminals to alert you that you wired two different data types together and LabVIEW converted the value passed into the node to a different representation. The block diagram places the coercion dots on the border of the terminal where the conversion takes place. For this exercise, the conversion does not affect how the VI runs. Refer to the *LabVIEW Help* for more information about coercion dots.

3. Display the front panel.
   A numeric indicator labeled **Peak to Peak** appears in the front panel window. This indicator displays the peak-to-peak value of the signal.

4. Run the VI.
   When the peak-to-peak value exceeds 2.0, the **Warning** indicator lights.

5. Click the **STOP** button to stop the VI.

6. Save the VI.

**Configuring a VI to Save Data to a File**

To store information about the data a VI generates, use the Write To Measurement File Express VI.

Complete the following steps to build a VI that saves peak-to-peak values and other information to a LabVIEW data file.

1. Search for the Write To Measurement File Express VI and add it to the block diagram below and to the right of the Amplitude and Level Measurements Express VI.
   The **Configure Write To Measurement File** dialog box appears.
   - The **Filename** text box displays the full path to the output file, test.lvm. A .lvm file is a tab-delimited text measurement file you can open with a spreadsheet application or a text-editing application. LabVIEW saves data with up to six digits of precision in a .lvm file. LabVIEW saves the .lvm file in the default **LabVIEW Data** directory. LabVIEW installs the LabVIEW Data directory in the default file directory of the operating system. When you want to view the data, use the file path displayed in the **Filename** text box to access the test.lvm file.

2. In the **Configure Write to Measurement File** dialog box, locate the If a file already exists section and select the Append to file option to write all the data to the test.lvm file without erasing any existing data in the file.

3. In the Segment Headers section, select the One header only option to create only one header in the file to which LabVIEW writes the data.
Chapter 3   Analyzing and Saving a Signal

4. Enter the following text in the File Description text box: Sample of peak to peak values. LabVIEW appends the text you enter in this text box to the header of the file.
5. Click the OK button to save the current configuration and close the Configure Write To Measurement File dialog box.

Saving Data to a File

When you run the VI, LabVIEW saves the data to the test.lvm file.

Complete the following steps to generate the test.lvm file.

1. On the block diagram, wire the Peak to Peak output of the Amplitude and Level Measurements Express VI to the Signals input of the Write To Measurement File Express VI.
2. Select File»Save As and save the VI as Save Data.vi in an easily accessible location.
3. Display the front panel and run the VI.
4. Click the front panel STOP button.
5. To view the data you saved, open the test.lvm file in the LabVIEW Data directory with a spreadsheet or text-editing application.
   The file has a header that contains information about the Express VI.
6. Close the file after you finish looking at it and return to the Save Data VI.

Adding a Button That Stores Data When Clicked

If you want to store only certain data points, you can configure the Write To Measurement File Express VI to save peak-to-peak values only when a user clicks a button.

Complete the following steps to add a button to the VI and configure how the button responds when a user clicks it.

1. Display the front panel and search the Controls palette for a rocker button. Select one of the rocker buttons and place it to the right of the waveform graphs.
2. Right-click the rocker button and select Properties from the shortcut menu to display the Boolean Properties dialog box.
3. Change the label of the button to Write to File.
4. On the Operation page of the Boolean Properties dialog box, select Latch when pressed from the Button behavior list.
   Use the Operation page to specify how a button behaves when a user clicks it. To see how the button reacts to a click, click the button in the Preview Selected Behavior section.
5. Click the OK button to save the current configuration and close the Boolean Properties dialog box.
6. Save the VI.
Saving Data When Prompted by a User

Complete the following steps to build a VI that logs data to a file when the user clicks a button on the front panel.

1. In the block diagram window, double-click the Write To Measurement File Express VI to display the **Configure Write To Measurement File** dialog box.
2. Change the filename `test.lvm` in the **Filename** text box to `Selected Samples.lvm` to save the data to a different file.
3. Click the **OK** button to save the current configuration and close the **Configure Write To Measurement File** dialog box.
4. Right-click the **Signals** input of the Write To Measurement File Express VI and select **Insert Input/Output** from the shortcut menu to insert the **Comment** input.
5. Right-click the **Comment** input of the Write To Measurement File Express VI and select **Select Input/Output>Enable** from the shortcut menu to replace the **Comment** input with the **Enable** input.

   The inputs and outputs of an Express VI appear in a predetermined order when you add new inputs and outputs. To select a specific input, you might need to add an input first, and then change the input to the specific one you want to use by right-clicking the input and selecting **Select Input/Output** from the shortcut menu.

6. Move the **Write to File** terminal to the left of the Write To Measurement File Express VI.
7. Wire the **Write to File** terminal to the **Enable** input of the Write To Measurement File Express VI.

The block diagram should appear similar to the following figure.

**Figure 3-5.** Block Diagram of the Save Data VI
Chapter 3  Analyzing and Saving a Signal

Viewing Saved Data

Complete the following steps to view the data that you save to the Selected Samples.lvm file.

1. Display the front panel and run the VI. Click the Write to File button several times.
2. Click the STOP button.
3. Open the Selected Samples.lvm file with a spreadsheet or text-editing application.
   The Selected Samples.lvm file differs from the test.lvm file. test.lvm recorded all the data generated by the Save Data VI, whereas Selected Samples.lvm recorded data only when you clicked the Write to File button.
4. Close the file after you finish looking at it.
5. Save and close the VI.

Summary

The following topics are a summary of the main concepts you learned in this chapter.

Controls and Indicators

You can configure front panel controls and indicators to perform tasks depending on what you want a VI to do. In this chapter, you learned to use controls and indicators in the following ways:

- You can build VIs that perform a task when certain conditions occur, such as displaying a warning light when a value exceeds a certain limit.
- You can build VIs that let users control when an Express VI executes by using buttons and the Enable input. You can configure the buttons to operate in one of six ways using the Operation page of the Boolean Properties dialog box.

Filtering Data

The Filter Express VI processes signals through filters and windows. You can use the Filter Express VI to remove noise from a signal.

Saving Data

The Write To Measurement File Express VI saves data that a VI generates and analyzes to a .lvm, .tdm, or .tdms measurement file. The text-based measurement file (.lvm) is a tab-delimited text file you can open with a spreadsheet application or a text-editing application. LabVIEW saves data with up to six digits of precision in a .lvm file. In addition to the data an Express VI generates, the .lvm file includes headers that contain information about the data, such as the date and time LabVIEW generated the data. The binary measurement file (.tdm) is a binary file that contains waveform data. Binary .tdm files provide higher accuracy for floating-point numbers, take up less space on disk, and perform faster than text-based measurement files (.lvm). The TDM Streaming file (.tdms) is a binary file that provides faster writing performance than the .tdm file format and allows a simpler interface for defining properties.
LabVIEW installs the `LabVIEW Data` directory in the default file directory of the operating system to help you organize and locate the data files LabVIEW generates. Refer to the `LabVIEW Help` for more information about writing data to and reading data from `.lvm` and `.cdm` files.
Hardware: Acquiring Data and Communicating with Instruments (Windows)

LabVIEW has the capability to connect and interact with a large number of hardware devices. This chapter introduces you to two Express VIs that make acquiring data and communicating with traditional, third-party instruments easier.

Hardware and Software Requirements

In the first exercise, you use the DAQ Assistant Express VI to acquire data with a DAQ device. This exercise requires data acquisition hardware and that you have NI-DAQmx installed. Refer to the NI-DAQmx Readme for more information about platforms supported by NI-DAQmx software.

Refer to the Taking Measurements book on the Contents tab in the LabVIEW Help for information about acquiring data and communicating with instruments on all platforms.

Note With NI-DAQmx 7.4 or later you can create NI-DAQmx simulated devices in MAX. An NI-DAQmx simulated device is a software replica of a DAQ device. Refer to the Measurement & Automation Explorer Help for NI-DAQmx for detailed instructions on creating an NI-DAQmx simulated device that you can use to complete the first exercise.

In the second exercise, you use the NI Instrument Driver Finder to find and install instrument drivers. To use the Instrument Driver Finder, you must have Internet access. In the second exercise, you also use the Instrument I/O Assistant Express VI to communicate with a traditional, third-party instrument. This exercise requires an instrument and that you have the Instrument I/O Assistant installed.

Refer to the Controlling Instruments book on the Contents tab in the LabVIEW Help for more information about communicating with instruments.

Note LabVIEW supports the DAQ and Instrument I/O Assistants used in this chapter on Windows only. The Instrument Driver Finder is available on Windows and Linux.
Acquiring a Signal in NI-DAQmx

You will use the DAQ Assistant Express VI to create a task in NI-DAQmx. NI-DAQmx is a programming interface you can use to communicate with data acquisition devices. Refer to the Getting Started with LabVIEW»Getting Started with DAQ»Taking an NI-DAQmx Measurement in LabVIEW book on the Contents tab in the LabVIEW Help for information about additional ways to create NI-DAQmx tasks.

In the following exercises, you will create an NI-DAQmx task that continuously takes a voltage reading and plots the data on a waveform graph.

You can complete the exercises in this chapter in approximately 30 minutes.

Creating an NI-DAQmx Task

In NI-DAQmx, a task is a collection of one or more channels, which contains timing, triggering, and other properties. Conceptually, a task represents a measurement or generation you want to perform. For example, you can create a task to measure temperature from one or more channels on a DAQ device.

Complete the following steps to create and configure a task that reads a voltage level from a DAQ device.

1. Open a new, blank VI.
2. On the block diagram, display the Functions palette and select Express»Input to display the Input palette.
3. Select the DAQ Assistant Express VI, shown below, on the Input palette and place it on the block diagram. The DAQ Assistant launches and the Create New Express Task dialog box appears.
4. Click Acquire Signals»Analog Input to display the Analog Input options.
5. Select Voltage to create a new voltage analog input task.
   The dialog box displays a list of channels on each installed DAQ device. The number of channels listed depends on the number of channels you have on the DAQ device.
6. In the Supported Physical Channels list, select the physical channel to which the device connects the signal, such as ai0, and then click the Finish button. The DAQ Assistant opens a new dialog box, shown in the following figure, that displays options for configuring the channel you selected to complete a task.

4-2 | ni.com
Figure 4-1. Configuring a Task Using the DAQ Assistant
7. In the DAQ Assistant dialog box select the **Configuration** tab and locate the **Voltage Input Setup** section.
8. Locate the **Settings** tab. In the **Signal Input Range** section enter 10 for the **Max** value and enter -10 for the **Min** value.
9. Locate the **Timing Settings** section at the bottom of the **Configuration** page. From the **Acquisition Mode** pull-down menu, select **N Samples**.
10. Enter a value of 1000 in the **Samples to Read** text box.
11. Click the **OK** button to save the current configuration and close the DAQ Assistant. LabVIEW builds the VI.
12. Save the VI as **Read Voltage.vi** in an easily accessible location.

**Graphing Data from a DAQ Device**

You can use the task you created in the previous exercise to graph the data acquired from a DAQ device.

Complete the following steps to plot the data from the channel on a waveform graph and change the name of the signal.

1. On the block diagram, right-click the **data** output and select **Create»Graph Indicator** from the shortcut menu.
2. Display the front panel and run the VI three or four times. Observe the waveform graph. **Voltage** appears in the plot legend at the top of the waveform graph.
3. On the block diagram, right-click the DAQ Assistant Express VI and select **Properties** from the shortcut menu to open the DAQ Assistant.
4. Right-click **Voltage** in the list of channels and select **Rename** from the shortcut menu to display the **Rename a channel or channels** dialog box.

   **Tip** You also can select the name of the channel and press the <F2> key to display the **Rename a channel or channels** dialog box.

5. In the **New Name** text box, enter **First Voltage Reading**, and click the **OK** button.
6. In the **DAQ Assistant** dialog box, click the **OK** button to save the current configuration and close the DAQ Assistant.
7. Display the front panel and run the VI. **First Voltage Reading** appears in the waveform graph plot legend.
8. Save the VI.
Editing an NI-DAQmx Task

You can add a channel to the task so you can compare two separate voltage readings. You also can customize the task to acquire the voltage readings continuously.

Complete the following steps to add a new channel to the task and acquire data continuously.

1. In the block diagram window, double-click the DAQ Assistant Express VI to open the DAQ Assistant.
2. Click the **Add Channels** button, shown below, and select **Voltage** to display the **Add Channels To Task** dialog box.

   ![Add Channels Button]

3. Select any unused physical channel in the **Supported Physical Channels** list, and click the **OK** button to return to the DAQ Assistant.
4. Rename the channel **Second Voltage Reading**.
5. In the **Timing Settings** section of the **Configuration** page, select **Continuous Samples** from the **Acquisition Mode** pull-down menu.
   
   When you set timing and triggering options in the DAQ Assistant, these options apply to all the channels in the list of channels.
6. Click the **OK** button to save the current configuration and close the DAQ Assistant. The **Confirm Auto Loop Creation** dialog box appears.
7. Click the **Yes** button. LabVIEW places a While Loop around the DAQ Assistant Express VI and the graph indicator on the block diagram. A stop button appears wired to the **stop** input of the DAQ Assistant Express VI. The **stopped** output of the Express VI is wired to the conditional terminal of the While Loop. The block diagram should appear similar to the following figure.

   ![Block Diagram of the Read Voltage VI]

   **Figure 4-2. Block Diagram of the Read Voltage VI**

   If an error occurs or you click the **stop** button while the VI is running, the DAQ Assistant Express VI stops reading data and the **stopped** output returns a TRUE value and stops the While Loop.
Visually Comparing Two Voltage Readings

Because you have two voltage readings displayed on a graph, you can customize the plots to distinguish between the two.

Complete the following steps to customize the plot color of the front panel waveform graph.

1. Expand the plot legend of the waveform graph to display two plots.
2. Run the VI.
   Two plots appear on the graph and the plot legend displays both plot names.
3. Click the icon that is to the right of First Voltage Reading in the plot legend and select Color from the shortcut menu. Using the color picker, select a color, such as yellow, so the plot is easy to read.
4. Change the plot color of Second Voltage Reading.
5. Stop the VI.
6. Save the VI.
7. Close the VI. The Getting Started window opens.

Communicating with an Instrument: Using Instrument Drivers and the Instrument I/O Assistant

An instrument driver is a set of software routines that control a programmable instrument. Each routine corresponds to a programmatic operation such as configuring, reading from, writing to, and triggering the instrument. Instrument drivers simplify instrument control and reduce test program development time by eliminating the need to learn the programming protocol for each instrument. Use an instrument driver for instrument control when possible. National Instruments provides thousands of instrument drivers for a wide variety of instruments.

In the following exercises, you will use instrument drivers and the Instrument I/O Assistant to communicate with an instrument. You must have an instrument installed to fully complete the following exercises.

Note These exercises refer to traditional, third-party instruments. Refer to ni.com/modularinstruments for more information about communicating with NI modular instruments.
Getting Started with the Instrument Driver Finder

Use the NI Instrument Driver Finder to search for and install LabVIEW Plug and Play instrument drivers without leaving the LabVIEW development environment.

Tip You also can visit the NI Instrument Driver Network at ni.com/idnet to find a driver for an instrument, request a driver for an instrument, and read helpful articles and tutorials about using instrument drivers.

Complete the following steps to launch and configure the NI Instrument Driver Finder.

1. From the Getting Started window, click Find Drivers and Add-ons and click the Connect to Instruments link. You also can launch the Instrument Driver Finder by selecting Help»Find Instrument Drivers or Tools»Instrumentation»Find Instrument Drivers.

2. Click the Login button to sign-in using your NI.com profile. If you do not have an ni.com profile, skip to step 4.

3. If you already have an ni.com profile, enter your email address and password and click the Login button.

4. If you do not have an ni.com profile, select the No, I need to create a profile option and click the Create Profile button. This action launches a browser window where you can create an ni.com profile. After you create a profile, return to the Instrument Driver Finder window and login with your new information.

You are now ready to search for, install, and use instrument drivers with the Instrument Driver Finder.

Finding and Installing Instrument Drivers

Complete the following steps to search for and install an instrument driver using the Instrument Driver Finder.

1. On the Configure Search page, click the Scan for Instruments button. This action prompts the NI Instrument Driver Finder to search for connected instruments. All results display under the Connected Instruments folder in the left-hand sidebar. If you do not currently have an instrument connected, the Instrument Driver Finder will return a result that reads <no connected instrument detected>.

2. Expand the Connected Instruments folder to display the search results and select an instrument from the list.

3. Select a manufacturer from the Manufacturer pull-down menu, and enter any keywords in the Additional Keywords section. Then click the Search button.

A list of available instrument drivers appears on the Search Results page. The driver result for the most recent version of LabVIEW appears first in the list.

If your search does not return any results, a sidebar with search tips appears. Refer to the IDNet website at ni.com/idnet for more information about searching for instrument drivers.
Note The NI Instrument Driver Finder displays drivers only for LabVIEW 7.0 or later. If you need to download an older version of an instrument driver, go to the IDNet website at ni.com/idnet to search for and download instrument drivers compatible with older versions of LabVIEW.

4. Select the driver you want to install and then click the Install button.
   After the driver installs successfully, the Instrument Driver Installation window appears. This window contains options for exploring and using the new driver. After installing, the new driver also appears in the Configure Search page under the Installed Instrument Driver folder.

5. Click the Install another driver button and click the Back button to return to the Configure Search window.

   Note You also can create your own instrument drivers. Refer to the Controlling Instruments»Using Instrument Drivers book on the Contents tab in the LabVIEW Help for more information about creating instrument drivers.

Using Instrument Drivers
After installing an instrument driver, you can explore example programs to learn more about using the instrument driver.

1. Double-click the newly installed instrument driver in the Installed Instrument Driver folder to display the Start Using Instrument Driver page.

2. The Start Using Instrument Driver page allows you to explore and customize the new driver. The following are recommendations to help you start using the new driver.
   • To view the new driver in the Project Explorer window, click the Open Project button. In the Project Explorer window, you can explore the VIs, folders, and supplemental files that make up the driver. You also can access the driver readme file in the Project Explorer window.
   • To view the driver’s palette, click the Open Palette button. From the palette, you can select and add the driver VIs to the front panel and the block diagram.
   • To view an example program, double-click on the listed example programs in the Examples section of the Start Using Instrument Driver page.

   Note Not all drivers have all options available. For example, if a driver does not have a project file, the Open Project button appears dimmed. Refer to the IDNet website at ni.com/idnet for more information about all instrument drivers.
Selecting an Instrument Using the Instrument I/O Assistant

If a driver is not available for an instrument, you can use the Instrument I/O Assistant Express VI to communicate with the instrument.

**Note** You must have the Instrument I/O Assistant installed to use the Instrument I/O Assistant Express VI. You install the Instrument I/O Assistant from the National Instruments Device Drivers CD.

Complete the following steps to use the Instrument I/O Assistant Express VI to select an instrument.

1. Turn on the instrument you want to use. The instrument must be powered on to use the Instrument I/O Assistant Express VI.
2. Open a new VI and display the block diagram window.
3. From the Input palette, select the Instrument I/O Assistant Express VI and add it to the block diagram. The Instrument I/O Assistant dialog box appears.
4. If the help is not visible to the right of the dialog box, click the Show Help button, shown below, in the upper right corner of the Instrument I/O Assistant dialog box.

   ![Show Help](image)

   The help appears to the right of the dialog box. The top help window contains how-to information about using the Instrument I/O Assistant. The bottom help window provides context-sensitive help about components in the dialog box.
5. Click the Select Instrument link in the top help window and follow the instructions in the help window to select the instrument with which you want to communicate.
6. If necessary, configure the properties of the instrument.
7. If you want to minimize the help window, click the Hide Help button, shown below, in the upper right corner of the Instrument I/O Assistant dialog box.

   ![Hide Help](image)

Acquiring and Parsing Information for an Instrument

After you select the instrument, you can send commands to the instrument to retrieve data. In this exercise, you will learn to use the Instrument I/O Assistant Express VI to acquire and parse identification information for an instrument.
Complete the following steps to communicate with the instrument.

1. In the Instrument I/O Assistant dialog box, click the Add Step button, expand the pull-down menu, and click the Query and Parse step.

2. Enter *IDN? in the Enter a command text box.
   *IDN? is a query that most instruments recognize. The response is an identification number string that describes the instrument. If the instrument does not accept this command, refer to the reference manual for the instrument for a list of commands the instrument does accept.

3. Click the Run this step button, shown below.

   ![Run this step button]

   The Instrument I/O Assistant sends the command to the instrument, and the instrument returns its identification information.

4. Select ASCII only from the pull-down menu below the Byte index column of the response window to parse the instrument name as an ASCII string. You also can use the Instrument I/O Assistant to parse ASCII numbers and binary data.

5. Click the Parsing help button, shown below, in the Instrument I/O Assistant dialog box to display information about parsing data.

6. In the ASCII representation column of the response window, click the value you want to parse.

7. Enter a name for the token, or parsed data selection, in the Token name text box.
   The name that you entered in the Token name text box is the output of the Instrument I/O Assistant Express VI, shown below.

Wiring a Command to an Instrument

After you acquire data from the instrument, you can add an input parameter to an instrument command. The parameter becomes an input to the VI or function.

Complete the following steps to add a parameter to a command.

1. Click the Add Step button, expand the pull-down menu, and click the Write step.

2. Enter *IDN? in the Enter a command text box.
3. Highlight the command in the **Enter a command** text box and click the **Add parameter** button to add a parameter to the command.

4. Enter a default value for the parameter in the **Test value** text box.

5. Enter a name for the parameter in the **Parameter name** text box. You use this name to reference the parameter in the application.

6. Click the **OK** button to save the current configuration and close the **Instrument I/O Assistant** dialog box.

**Summary**

The following topics are a summary of the main concepts you learned in this chapter.

**DAQ Assistant Express VI**

You can use the DAQ Assistant Express VI to interactively build measurement channels or tasks.

Add the DAQ Assistant Express VI to the block diagram to configure channels and tasks for use with NI-DAQmx for data acquisition. NI-DAQmx is a programming interface you can use to communicate with data acquisition devices. You can use the DAQ Assistant Express VI to control devices supported by NI-DAQmx.

Refer to the Getting Started with LabVIEW»Getting Started with DAQ»Taking an NI-DAQmx Measurement in LabVIEW book on the Contents tab in the LabVIEW Help for information about the DAQ Assistant.

Refer to the NI-DAQmx Readme for information about devices supported by NI-DAQmx. If NI-DAQmx does not support the device you want to use, refer to the Taking Measurements book on the Contents tab in the LabVIEW Help for information about using Traditional NI-DAQ (Legacy) for data acquisition.

**Tasks in NI-DAQmx**

In NI-DAQmx, a task is a collection of one or more virtual channels with timing, triggering, and other properties. Conceptually, a task represents a measurement or generation you want to perform.

For example, you can configure a collection of channels for analog input operations. After you create a task, you can access the single task instead of configuring the channels individually to perform analog input operations. After you create a task, you can add or remove channels from that task.

Refer to the Taking Measurements book on the Contents tab in the LabVIEW Help for more information about channels and tasks.
Chapter 4   Hardware: Acquiring Data and Communicating with Instruments (Windows)

Instrument Drivers

Use the NI Instrument Driver Finder to search for and install LabVIEW Plug and Play instrument drivers without leaving the LabVIEW development environment.

An instrument driver is a set of software routines that control a programmable instrument. Each routine corresponds to a programmatic operation such as configuring, reading from, writing to, and triggering the instrument. Use an instrument driver for instrument control when possible. National Instruments provides thousands of instrument drivers for a wide variety of instruments.

Refer to the Controlling Instruments»Using Instrument Drivers book on the Contents tab in the LabVIEW Help for more information about the Instrument Driver Finder.

You also can visit the NI Instrument Driver Network at ni.com/idnet to find a driver for an instrument, or you can create your own instrument drivers. Refer to the Controlling Instruments»Using Instrument Drivers book on the Contents tab in the LabVIEW Help for more information about creating instrument drivers.

Instrument I/O Assistant Express VI

If a driver is not available for an instrument, you can use the Instrument I/O Assistant Express VI to communicate with the instrument. You can use the Instrument I/O Assistant to communicate with message-based instruments and graphically parse the response. Start the Instrument I/O Assistant by adding the Instrument I/O Assistant Express VI to the block diagram or by double-clicking the Instrument I/O Assistant Express VI icon on the block diagram.

Refer to the Instrument I/O Assistant Help for information about communicating with an external device. Display the Instrument I/O Assistant Help by clicking the Show Help button in the Instrument I/O Assistant dialog box.
Using Other LabVIEW Features

The previous chapters in this manual introduce you to most of the LabVIEW features you need to build common measurement applications. As you familiarize yourself with the LabVIEW environment, you might find that you need to enhance VIs or that you need more fine-tuned control of the processes the VIs perform. This chapter introduces you to some of the concepts you should be familiar with as you start using other LabVIEW features. Refer to the Fundamentals book on the Contents tab in the LabVIEW Help for more information about these concepts. The Concepts books contain information about LabVIEW programming concepts, and the How-To books contain step-by-step instructions for using LabVIEW.

All Controls and Indicators

The controls and indicators located on the Express subpalette of the Controls palette are a subset of the complete set of built-in controls and indicators available in LabVIEW. On other subpalettes you can find all the controls and indicators that you can use to create the front panel. However, subpalettes other than the Express subpalette categorize controls and indicators by functionality instead of having a subpalette for controls and a subpalette for indicators.

For example, the top level of the Express subpalette has a Numeric Controls subpalette and a Numeric Indicators subpalette. On the Modern, Classic, and System subpalettes, these controls and indicators are located on the Numeric subpalette because they are all numeric objects.

Click the Customize button on the pinned Controls palette and select Change Visible Palettes from the shortcut menu to display the Change Visible Palettes dialog box. Then place checkmarks in the checkboxes next to the categories you want to view on the Controls palette.

Refer to the Fundamentals»Building the Front Panel book on the Contents tab in the LabVIEW Help for more information about using the complete set of built-in controls and indicators available in LabVIEW.

All VIs and Functions

The Express VIs and structures located on the Express subpalette of the Functions palette are a small subset of the complete set of built-in VIs, functions, and structures available in LabVIEW.
Click the **Customize** button on the pinned **Functions** palette and select **Change Visible Palettes** from the shortcut menu to display the **Change Visible Palettes** dialog box. Then place checkmarks in the checkboxes next to the categories you want to view on the **Functions** palette.

LabVIEW uses colored icons to distinguish between functions, VIs, and Express VIs. Icons for functions have pale yellow backgrounds, most icons for VIs have white backgrounds, and icons for Express VIs appear surrounded by pale blue fields.

Express VIs appear on the block diagram as expandable nodes with icons surrounded by a blue field. Unlike Express VIs, most functions and VIs on the block diagram appear as icons rather than expandable nodes.

**VIs**

You can use an existing VI or a VI you create as a subVI. When you place a VI on the block diagram, the VI is a subVI. When you double-click a subVI, its front panel appears, rather than a dialog box in which you can configure options.

The icon for a VI appears in the upper right corner of the front panel and block diagram. This icon is the same as the icon that appears when you place the VI on the block diagram. You can use the default icon or create a custom icon using the **Icon Editor**.

Refer to the **Fundamentals»Creating VIs and SubVIs** book on the Contents tab in the LabVIEW Help for more information about creating VIs, configuring them as subVIs, and creating icons.

You also can save the configuration of an Express VI as a subVI. Refer to the **Fundamentals»Building the Block Diagram** book on the Contents tab in the LabVIEW Help for more information about creating subVIs from Express VIs.

**Functions**

Functions are the fundamental operating elements of LabVIEW. Unlike VIs, functions do not have front panels or block diagrams. Functions provide the basic building blocks for programming a VI, interfacing with hardware and software, and performing other essential tasks in LabVIEW. Refer to the **Fundamentals»Building the Block Diagram** book on the Contents tab in the LabVIEW Help for more information about functions.

**Data Types**

On the block diagram of a VI, the terminals for the front panel objects are different colors. The color and symbol of a terminal indicate the data type of the corresponding control or indicator. Colors also indicate the data types of wires, inputs, and outputs. The color of inputs and outputs of Express VIs indicate what type of data the input or output accepts or returns.

Data types indicate which objects, inputs, and outputs you can wire together. For example, a switch has a green border, so you can wire a switch to any Express VI input with a green label.
A knob has an orange border, so you can wire a knob to any Express VI input with an orange label. However, you cannot wire a knob to an input with a green label. The wires you create are the same color as the terminal.

Express VIs generate and acquire data using the dynamic data type. The dynamic data type appears as a dark blue terminal, shown below. Most Express VIs accept or return dynamic data. You can wire dynamic data to any indicator or input that accepts numeric, waveform, or Boolean data. Wire dynamic data to an indicator that can best present the data. Such indicators include graphs, charts, and numeric indicators.

Most other VIs and functions in LabVIEW do not accept dynamic data. To use a built-in VI or function to analyze or process dynamic data, you must convert the dynamic data to numeric, Boolean, waveform, or array data.

Use the Convert from Dynamic Data Express VI to convert dynamic data to numeric, Boolean, waveform, and array data for use with other VIs and functions. When you wire dynamic data to an array indicator, LabVIEW inserts the Convert from Dynamic Data Express VI on the block diagram.

Use the Convert to Dynamic Data Express VI to convert numeric, Boolean, waveform, and array data to dynamic data for use with Express VIs.

Refer to the Fundamentals » Building the Block Diagram book on the Contents tab in the LabVIEW Help for more information about data types.
When to Use Other LabVIEW Features

The Express VIs, structures, and controls and indicators located on the Express subpalettes of the Controls and Functions palettes provide the functionality you need to build common measurement applications. The following list describes the applications that require you to use the VIs, functions, structures, controls, and indicators located on subpalettes other than the Express subpalette.

- **Programmatically control properties and methods for the LabVIEW environment, VIs, and controls and indicators**—You can control programmatically how a VI behaves when it runs, set the appearance of a control or indicator, or control how the LabVIEW environment behaves. Refer to the Fundamentals»Programmatically Controlling VIs book on the Contents tab in the LabVIEW Help for more information about these features.

- **Call code written in text-based languages**—You can use LabVIEW to communicate with applications written in a text-based programming language, such as C or C++. Refer to the Fundamentals»Calling Shared Libraries in LabVIEW book on the Contents tab in the LabVIEW Help for more information about these features.

- **Communicate with VIs across a network**—You can call a VI that resides on another computer running LabVIEW. Refer to the Fundamentals»Transferring Data over a Network book on the Contents tab in the LabVIEW Help for more information about these features.

- **Share data within an application or across a network**—You can create configured software items called shared variables to share data among VIs or between locations on the block diagram that you cannot connect with wires. Refer to the Fundamentals»Transferring Data over a Network book on the Contents tab in the LabVIEW Help for more information about these features.

- **Publish VIs on the Web**—You can publish the front panel of any VI on the Web, where users can interact with the front panel. Refer to the Fundamentals»Transferring Data over a Network book on the Contents tab in the LabVIEW Help for more information about these features.

- **Save data to a variety of file formats**—In addition to the text-based measurement file format, you can create files that other applications can use, such as text files and spreadsheet files. Refer to the Fundamentals»File I/O book on the Contents tab in the LabVIEW Help for more information about these features.

- **Customize menus**—You can configure which menu items appear when a user runs a VI. You also can create custom menus. Refer to the Fundamentals»Creating VIs and SubVIs book on the Contents tab in the LabVIEW Help for more information about these features.

- **Use LabVIEW projects**—You can create projects to group together LabVIEW files and files not specific to LabVIEW, create build specifications, and deploy or download files to multiple targets from one location. Use the Create Project dialog box to select templates and sample projects that help you get started with LabVIEW projects.

  You must use a project to build applications and shared libraries. You also must use a project to work with an RT, FPGA, PDA, Touch Panel, DSP, or embedded target. Refer to the specific module documentation for more information about using projects with these
targets. Refer to the Fundamentals»Working with Projects and Targets book on the Contents tab in the LabVIEW Help for more information about using LabVIEW projects.

- **Access other Windows applications**—You can use LabVIEW as a .NET or ActiveX client to access the objects, properties, methods, and events associated with .NET server or ActiveX applications. Refer to the Fundamentals»Windows Connectivity book on the Contents tab in the LabVIEW Help for more information about these features.

- **Write mathematical formulas, equations, and scripts**—You can use various nodes to perform mathematical operations on the block diagram. You also can use the LabVIEW MathScript text-based language to write mathematical functions and scripts. Refer to the Fundamentals»Formulas and Equations book on the Contents tab in the LabVIEW Help for more information about these features.
Technical Support and Professional Services

Log in to your National Instruments ni.com User Profile to get personalized access to your services. Visit the following sections of ni.com for technical support and professional services:

- **Support**—Technical support at ni.com/support includes the following resources:
  - **Self-Help Technical Resources**—For answers and solutions, visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support, as well as exclusive access to self-paced online training modules at ni.com/self-paced-training. All customers automatically receive a one-year membership in the Standard Service Program (SSP) with the purchase of most software products and bundles including NI Developer Suite. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit ni.com/ssp for more information.

For information about other technical support options in your area, visit ni.com/services, or contact your local office at ni.com/contact.

- **Training and Certification**—Visit ni.com/training for training and certification program information. You can also register for instructor-led, hands-on courses at locations around the world.

- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.
# Glossary

## A

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic scaling</td>
<td>Ability of scales to adjust to the range of plotted values. On graph scales, autoscaling determines maximum and minimum scale values.</td>
</tr>
</tbody>
</table>

## B

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block diagram</td>
<td>Pictorial description or representation of a program or algorithm. The block diagram consists of executable icons called nodes and wires that carry data between the nodes. The block diagram is the source code for the VI. The block diagram resides in the block diagram window of the VI.</td>
</tr>
<tr>
<td>Boolean controls and indicators</td>
<td>Front panel objects to manipulate and display Boolean (TRUE or FALSE) data.</td>
</tr>
<tr>
<td>broken Run button</td>
<td>Button that replaces the Run button when a VI cannot run because of errors.</td>
</tr>
<tr>
<td>broken VI</td>
<td>VI that cannot run because of errors; signified by a broken arrow in the broken Run button.</td>
</tr>
</tbody>
</table>
1. Physical—A terminal or pin at which you can measure or generate an analog or digital signal. A single physical channel can include more than one terminal, as in the case of a differential analog input channel or a digital port of eight lines. A counter also can be a physical channel, although the counter name is not the name of the terminal where the counter measures or generates the digital signal.

2. Virtual—A collection of property settings that can include a name, a physical channel, input terminal connections, the type of measurement or generation, and scaling information. You can define NI-DAQmx virtual channels outside a task (global) or inside a task (local). Configuring virtual channels is optional in Traditional NI-DAQ (Legacy) and earlier versions, but is integral to every measurement you take in NI-DAQmx. In Traditional NI-DAQ (Legacy), you configure virtual channels in MAX. In NI-DAQmx, you can configure virtual channels either in MAX or in your program, and you can configure channels as part of a task or separately.

3. Switch—A switch channel represents any connection point on a switch. It can be made up of one or more signal wires (commonly one, two, or four), depending on the switch topology. A virtual channel cannot be created with a switch channel. Switch channels may be used only in the NI-DAQmx Switch functions and VIs.

**Checkbox**
Small square box in a dialog box which you can select or clear. Checkboxes generally are associated with multiple options that you can set. You can select more than one checkbox.

**Conditional terminal**
Terminal of a While Loop that contains a Boolean value that determines if the VI performs another iteration.

**Context Help window**
Window that displays basic information about LabVIEW objects when you move the cursor over each object. Objects with context help information include VIs, functions, constants, structures, palettes, properties, methods, events, and dialog box components.

**Control**
Front panel object for entering data to a VI interactively or to a subVI programmatically, such as a knob, push button, or dial.
### Controls palette
Palette that contains front panel controls, indicators, and decorative objects.

### current VI
VI whose front panel, block diagram, or Icon Editor is the active window.

---

### D

#### DAQ
See data acquisition (DAQ).

#### DAQ Assistant
A graphical interface for configuring measurement tasks, channels, and scales.

#### DAQ device
A device that acquires or generates data and can contain multiple channels and conversion devices. DAQ devices include plug-in devices, PCMCIA cards, and DAQPad devices, which connect to a computer USB or IEEE 1394 port. SCXI modules are considered DAQ devices.

#### data acquisition (DAQ)
1. Acquiring and measuring analog or digital electrical signals from sensors, acquisition transducers, and test probes or fixtures.
2. Generating analog or digital electrical signals.

#### data flow
Programming system that consists of executable nodes that execute only when they receive all required input data. The nodes produce output data automatically when they execute. LabVIEW is a dataflow system. The movement of data through the nodes determines the execution order of the VIs and functions on the block diagram.

#### data type
Format for information. In LabVIEW, acceptable data types for most VIs and functions are numeric, array, string, Boolean, path, refnum, enumeration, waveform, and cluster.

#### default
Preset value. Many VI inputs use a default value if you do not specify a value.

#### device
An instrument or controller you can access as a single entity that controls or monitors real-world I/O points. A device often is connected to a host computer through some type of communication network. See also DAQ device and measurement device.

#### drag
To use the cursor on the screen to select, move, copy, or delete objects.
### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>driver</strong></td>
<td>Software that controls a specific hardware device, such as a DAQ device.</td>
</tr>
<tr>
<td><strong>dynamic data type</strong></td>
<td>Data type used by Express VIs that includes the data associated with a signal and attributes that provide information about the signal, such as the name of the signal or the date and time LabVIEW acquired the data. Attributes specify how the signal appears on a graph or chart.</td>
</tr>
<tr>
<td><strong>Error list window</strong></td>
<td>Window that displays errors and warnings occurring in a VI and in some cases recommends how to correct the errors.</td>
</tr>
<tr>
<td><strong>error message</strong></td>
<td>Indication of a software or hardware malfunction or of an unacceptable data entry attempt.</td>
</tr>
<tr>
<td><strong>Express VI</strong></td>
<td>A subVI designed to aid in common measurement tasks. You configure an Express VI using a configuration dialog box.</td>
</tr>
<tr>
<td><strong>For Loop</strong></td>
<td>Iterative loop structure that executes its subdiagram a set number of times. Equivalent to text-based code: <code>For i = 0 to n - 1, do...</code>.</td>
</tr>
<tr>
<td><strong>front panel</strong></td>
<td>Interactive user interface of a VI. Front panel appearance imitates physical instruments, such as oscilloscopes and multimeters.</td>
</tr>
<tr>
<td><strong>function</strong></td>
<td>Built-in execution element, comparable to an operator, function, or statement in a text-based programming language.</td>
</tr>
<tr>
<td><strong>Functions palette</strong></td>
<td>Palette that contains VIs, functions, block diagram structures, and constants.</td>
</tr>
</tbody>
</table>
Getting Started with LabVIEW

**graph**
2D display of one or more plots. A graph receives and plots data as a block.

**I/O**
Input/Output. The transfer of data to or from a computer system involving communications channels, operator input devices, and/or data acquisition and control interfaces.

**icon**
Graphical representation of a node on a block diagram.

**indicator**
Front panel object that displays output, such as a graph or LED.

**instrument driver**
A set of high-level functions that control and communicate with instrument hardware in a system.

**Instrument I/O Assistant**
Add-on launched from the Instrument I/O Assistant Express VI that communicates with message-based instruments and graphically parses the response.

**L**

**label**
Text object used to name or describe objects or regions on the front panel or block diagram.

**LabVIEW**
Laboratory Virtual Instrument Engineering Workbench. LabVIEW is a graphical programming language that uses icons instead of lines of text to create programs.

**LED**
Light-emitting diode.

**legend**
Object a graph or chart owns to display the names and plot styles of plots on that graph or chart.

**M**

**MAX**
*See Measurement & Automation Explorer.*

**Measurement & Automation Explorer**
The standard National Instruments hardware configuration and diagnostic environment for Windows.

**measurement device**
DAQ devices such as the E Series multifunction I/O (MIO) devices, SCXI signal conditioning modules, and switch modules.
Glossary

menu bar  Horizontal bar that lists the names of the main menus of an application. The menu bar appears below the title bar of a window. Each application has a menu bar that is distinct for that application, although some menus and commands are common to many applications.

NI-DAQ  Driver software included with all NI DAQ devices and signal conditioning components. NI-DAQ is an extensive library of VIs and ANSI C functions you can call from an application development environment (ADE), such as LabVIEW, to program an NI measurement device, such as the M Series multifunction I/O (MIO) DAQ devices, signal conditioning modules, and switch modules.

NI-DAQmx  The latest NI-DAQ driver with new VIs, functions, and development tools for controlling measurement devices. The advantages of NI-DAQmx over earlier versions of NI-DAQ include the DAQ Assistant for configuring channels and measurement tasks for your device for use in LabVIEW, LabWindows™/CVI™, and Measurement Studio; NI-DAQmx simulation for most supported devices for testing and modifying applications without plugging in hardware; and a simpler, more intuitive API for creating DAQ applications using fewer functions and VIs than earlier versions of NI-DAQ.

node  Program execution element. Nodes are analogous to statements, operators, functions, and subroutines in text-based programming languages. On a block diagram, nodes include functions, structures, and subVIs.

numeric controls and indicators  Front panel objects to manipulate and display numeric data.

object  Generic term for any item on the front panel or block diagram, including controls, indicators, structures, nodes, wires, and imported pictures.

Operating tool  Tool to enter data into controls or to operate them.
| P | palette | Displays objects or tools you can use to build the front panel or block diagram. |
| | plot | Graphical representation of an array of data shown either on a graph or a chart. |
| | Positioning tool | Tool to move and resize objects. |
| | project | A collection of LabVIEW files and files not specific to LabVIEW that you can use to create build specifications and deploy or download files to targets. |
| | Project Explorer window | Window in which you can create and edit LabVIEW projects. |
| | Properties dialog boxes | Dialog boxes accessed from the shortcut menu of a control or indicator that you can use to configure how the control or indicator appears in the front panel window. |
| | pull-down menus | Menus accessed from a menu bar. Pull-down menu items are usually general in nature. |
| | PXI | PCI eXtensions for Instrumentation. A modular, computer-based instrumentation platform. |
| R | RMS | Root Mean Square. |
| S | sample | Single analog or digital input or output data point. |
| | scale | Part of graph, chart, and some numeric controls and indicators that contains a series of marks or points at known intervals to denote units of measure. |
| | shortcut menu | Menu accessed by right-clicking an object. Menu items pertain to that object specifically. |
Glossary

string  Representation of a value as text.
structure  Program control element, such as a Flat Sequence structure, Stacked Sequence structure, Case structure, For Loop, While Loop, or Timed Loop.
subpalette  Palette that you access from another palette that is above the subpalette in hierarchy.
subVI  VI used on the block diagram of another VI. Comparable to a subroutine.

task  A collection of one or more channels, timing, triggering, and other properties in NI-DAQmx. A task represents a measurement or generation you want to perform.
template VI  VI that contains common controls and indicators from which you can build multiple VIs that perform similar functions. Access template VIs from the New dialog box.
terminal  Object or region on a node through which data pass.
tip strip  Small yellow text banners that identify the terminal name and make it easier to identify terminals for wiring.
tool  Special cursor to perform specific operations.
toolbar  Bar that contains command buttons to run and debug VIs.

Traditional NI-DAQ (Legacy)  An older driver with outdated APIs for developing data acquisition, instrumentation, and control applications for older National Instruments DAQ devices. You should use Traditional NI-DAQ (Legacy) only in certain circumstances. Refer to the NI-DAQ Readme for more information about when to use Traditional NI-DAQ (Legacy), including a complete list of supported devices, operating systems, and application software and language versions.
**V**

<table>
<thead>
<tr>
<th><strong>VI</strong></th>
<th>See virtual instrument (VI).</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual instrument (VI)</td>
<td>Program in LabVIEW that models the appearance and function of a physical instrument.</td>
</tr>
<tr>
<td>VXI</td>
<td>VME eXtensions for Instrumentation (bus).</td>
</tr>
</tbody>
</table>

**W**

<table>
<thead>
<tr>
<th><strong>waveform</strong></th>
<th>Multiple voltage readings taken at a specific sampling rate.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>waveform chart</strong></td>
<td>Indicator that plots data points at a certain rate.</td>
</tr>
<tr>
<td><strong>While Loop</strong></td>
<td>Loop structure that repeats a section of code until a condition occurs.</td>
</tr>
<tr>
<td><strong>wire</strong></td>
<td>Data path between nodes.</td>
</tr>
<tr>
<td><strong>Wiring tool</strong></td>
<td>Tool to define data paths between terminals.</td>
</tr>
</tbody>
</table>
Index

A
acquiring
  information for instruments, 4-9
  signals, 4-2
Acquiring a Signal VI block diagram (figure), 1-14
adding
  See also creating
  channels to tasks, 4-5
  controls from the block diagram, 2-5
  controls to the front panel, 1-4
  graph indicators, 2-5
  inputs to Express VIs, 1-7, 2-4, 3-13
  multiple signals, 3-4
  numeric indicators, 2-4
  signals, 3-3
  visual cues on front panel, 3-9
  warning lights, 3-9
Amplitude and Level Measurements Express VI, 3-2
  analyzing voltage, 3-7
  analyzing signals, 3-7
applications
  building, 5-4
  communicating with across networks, 5-4
Arithmetic & Comparison palette, 1-11
VIs, 1-1
buttons
  adding, 3-12
  Run, 1-4

C
calling code from text-based languages, 5-4, 5-5
changing signal types, 1-6
channels, 4-2
  adding to tasks, 4-5
  renaming, 4-4
communicating
  with instruments, 4-6
  with LabVIEW applications across networks, 5-4
Comparison Express VI, 3-9
configuration dialog boxes, 1-19
configuring
  controls, 1-19
  indicators, 1-19
Context Help window, 2-11
  button, 2-2, 3-2
  displaying configuration of Express VIs, 3-2
  displaying errors, 2-13
  figure, 2-2
  showing, 2-2
controlling
  execution speed, 2-8
  VIs programmatically, 5-4
controls, 1-18, 3-14
  adding from the block diagram, 2-5
  adding to the front panel, 1-4
  configuring, 1-19
  creating, 2-5, 2-12
  customizing, 1-15
  data types, 5-3
  numeric, 5-1
  palette, 1-4
Controls palette, 1-4
  figure, 1-5
  showing all categories, 5-1
Index

Convert from/to Dynamic Data Express VIs, 5-3
drivers
  instrument, 4-6
  NI resources, A-1
dynamic data, converting from and to, 5-3

creating
  See also adding controls, 2-5, 2-12
graph indicators, 2-5
  indicators, 2-12
  NI-DAQmx tasks, 4-2

customizing
  block diagrams, 2-12
  controls, 1-15
  front panels, 2-4
  indicators, 1-17
  menus, 5-4
  simulated signals, 3-3

customizing block diagrams, 2-12
controls, 1-15
front panels, 2-4
indicators, 1-17
menus, 5-4
simulated signals, 3-3

data
displaying
  from DAQ devices, 4-4
  in tables, 2-9, 2-13
dynamic, converting from and to, 5-3
graphing from DAQ devices, 4-4
saving
to a file, 3-11, 3-12
  when prompted by a user, 3-13
  storing, 3-12
data flow, 1-9, 1-14, 1-18
data types
dynamic, 5-3
  overview, 5-3
deleting wires, 1-10
deselecting objects, 1-8
diagnostic tools (NI resources), A-1
displaying
data from DAQ devices, 4-4
data in tables, 2-9, 2-13
errors in Context Help window, 2-13
  signals in a graph, 1-14
documentation
  introduction to this manual, ix
  NI resources, A-1

datasheet, x

dataflow
  displaying from DAQ devices, 4-4
  in tables, 2-9, 2-13
dynamic, converting from and to, 5-3
graphing from DAQ devices, 4-4
saving
to a file, 3-11, 3-12
  when prompted by a user, 3-13
  storing, 3-12
data flow, 1-9, 1-14, 1-18
data types
dynamic, 5-3
  overview, 5-3
deleting wires, 1-10
deselecting objects, 1-8
diagnostic tools (NI resources), A-1
displaying
data from DAQ devices, 4-4
data in tables, 2-9, 2-13
errors in Context Help window, 2-13
  signals in a graph, 1-14
documentation
  introduction to this manual, ix
  NI resources, A-1

datasheet, x

dataflow
  displaying from DAQ devices, 4-4
  in tables, 2-9, 2-13
dynamic, converting from and to, 5-3
graphing from DAQ devices, 4-4
saving
to a file, 3-11, 3-12
  when prompted by a user, 3-13
  storing, 3-12
data flow, 1-9, 1-14, 1-18
data types
dynamic, 5-3
  overview, 5-3
deleting wires, 1-10
deselecting objects, 1-8
diagnostic tools (NI resources), A-1
displaying
data from DAQ devices, 4-4
data in tables, 2-9, 2-13
errors in Context Help window, 2-13
  signals in a graph, 1-14
documentation
  introduction to this manual, ix
  NI resources, A-1

datasheet, x

dataflow
  displaying from DAQ devices, 4-4
  in tables, 2-9, 2-13
dynamic, converting from and to, 5-3
graphing from DAQ devices, 4-4
saving
to a file, 3-11, 3-12
  when prompted by a user, 3-13
  storing, 3-12
data flow, 1-9, 1-14, 1-18
data types
dynamic, 5-3
  overview, 5-3
deleting wires, 1-10
deselecting objects, 1-8
diagnostic tools (NI resources), A-1
displaying
data from DAQ devices, 4-4
data in tables, 2-9, 2-13
errors in Context Help window, 2-13
  signals in a graph, 1-14
documentation
  introduction to this manual, ix
  NI resources, A-1

datasheet, x

dataflow
  displaying from DAQ devices, 4-4
  in tables, 2-9, 2-13
dynamic, converting from and to, 5-3
graphing from DAQ devices, 4-4
saving
to a file, 3-11, 3-12
  when prompted by a user, 3-13
  storing, 3-12
data flow, 1-9, 1-14, 1-18
data types
dynamic, 5-3
  overview, 5-3
deleting wires, 1-10
deselecting objects, 1-8
diagnostic tools (NI resources), A-1
displaying
data from DAQ devices, 4-4
data in tables, 2-9, 2-13
errors in Context Help window, 2-13
  signals in a graph, 1-14
documentation
  introduction to this manual, ix
  NI resources, A-1

datasheet, x

dataflow
  displaying from DAQ devices, 4-4
  in tables, 2-9, 2-13
dynamic, converting from and to, 5-3
graphing from DAQ devices, 4-4
saving
to a file, 3-11, 3-12
  when prompted by a user, 3-13
  storing, 3-12
data flow, 1-9, 1-14, 1-18
data types
dynamic, 5-3
  overview, 5-3
deleting wires, 1-10
deselecting objects, 1-8
diagnostic tools (NI resources), A-1
displaying
data from DAQ devices, 4-4
data in tables, 2-9, 2-13
errors in Context Help window, 2-13
  signals in a graph, 1-14
documentation
  introduction to this manual, ix
  NI resources, A-1

datasheet, x

dataflow
  displaying from DAQ devices, 4-4
  in tables, 2-9, 2-13
dynamic, converting from and to, 5-3
graphing from DAQ devices, 4-4
saving
to a file, 3-11, 3-12
  when prompted by a user, 3-13
  storing, 3-12
data flow, 1-9, 1-14, 1-18
data types
dynamic, 5-3
  overview, 5-3
deleting wires, 1-10
deselecting objects, 1-8
diagnostic tools (NI resources), A-1
displaying
data from DAQ devices, 4-4
data in tables, 2-9, 2-13
errors in Context Help window, 2-13
  signals in a graph, 1-14
documentation
  introduction to this manual, ix
  NI resources, A-1

F
grouping, 5-4

files
  saving to other formats, 5-4

Filter Express VI, 3-6
Formula Express VI, 1-11, 3-4
front panel, 1-3, 1-18, 2-9
  Acquiring a Signal VI (figure), 1-1
  adding controls, 1-4
  visual cues, 3-9
controls, 1-18, 3-14
customizing, 2-4
indicators, 1-18
modifying, 2-7
showing, 1-9
Warning Light VI (figure), 3-1
functions, 5-2
Merge Signals, 1-14, 2-5
Functions palette
figure, 1-11
showing all categories, 5-2

G
Getting Started window, 1-2, 3-2
graph indicators, creating, 2-5
graphing
data from DAQ devices, 4-4
two signals, 1-14
grouping files, 5-4

H
help
Context Help window, 2-2, 2-11, 3-2
LabVIEW Help, 1-19
searching, 2-3, 2-10, 2-12
LabVIEW resources, 1-19, 2-11
searching, 2-3, 2-10, 2-12
technical support, A-1

I
indicators, 1-18, 3-14, 5-1
adding numeric, 2-4
configuring, 1-19
creating, 2-12
customizing, 1-17
data type, 5-3
numeric, 5-1
removing, 2-7
Input palette, 2-2
inputs, Express VI, 1-19
Instrument Driver Network, 4-6
instrument drivers, 4-6
finding, 4-12
installing, 4-12

NI resources, A-1
Instrument I/O Assistant Express VI,
4-8, 4-12
instruments
acquiring information, 4-9
communicating, 4-6
parsing information, 4-9
selecting, 4-9
introduction to this manual, ix

K
knob control, customizing (figure), 1-16
KnowledgeBase, A-1

L
LabVIEW
help resources, 2-11
other features, 5-1
projects, 5-4
LabVIEW Help, 1-19
searching, 2-3, 2-10, 2-12
LEDs, palette, 3-9
LVM. See .lvm files
.lvm files, 3-12, 3-14

M
manual. See documentation
marquee, 2-9
menus, customizing, 5-4
Merge Signals function, 2-5
figure, 1-15
modifying
front panels, 2-7
signals, 1-10, 2-3

National Instruments support and services,
A-1
New dialog box, 1-2, 3-2
figure, 1-3
NI Example Finder, 2-11
NI Instrument Driver Finder, 4-12
NI Instrument Driver Network, 4-6
NI support and services, A-1

© National Instruments | 1-3
Index

NI-DAQmx tasks, 4-11
creating, 4-2
numeric controls, 5-1
    palette, 1-5

O
objects
deselecting, 1-8
wiring on the block diagram, 1-8
Operating tool, 1-9
outputs, Express VI, 1-19

P
palettes
    Arithmetic & Comparison, 1-11
    Controls, 1-4
    Execution Control, 2-6, 2-12
    Functions, 1-10
    Input, 2-2
    LEDs, 3-9
    searching, 2-6
    showing all categories, 5-1, 5-2
parsing information for instruments, 4-9
placing objects on the block diagram from the
    help, 2-12
Positioning tool, 1-8
programmatically controlling VIs, 5-4
programming examples (NI resources), A-1
projects, 5-4
property dialog boxes, 1-19, 1-20
publishing VIs on the Web, 5-4

T
tables, 2-9
displaying data, 2-13
tasks
    adding new channels, 4-5
    NI-DAQmx, 4-11
TDM. See .tdm files
    .tdm files, 3-14
    .tdms files, 3-14
technical support, A-1
template VIs, 1-2
text-based languages, calling code, 5-4, 5-5
Time Delay Express VI, 2-8
tools
    Operating, 1-9
    Positioning, 1-8
    Wiring, 1-9
training and certification (NI resources), A-1
troubleshooting (NI resources), A-1

S
saving data
different file formats, 5-4

Save Data VI block diagram
    (figure), 3-13
to files, 3-11, 3-12, 3-14
when prompted by user, 3-13, 3-14
searching
    examples, 2-11
    help, 2-3, 2-10, 2-12
    palettes, 2-6
selecting
    instruments, 4-9
    objects, 1-8
shared libraries, building, 5-4
signals
    acquiring, 4-2
    analyzing, 3-7
    changing type, 1-6
    graphing, 1-14
    modifying, 1-10, 2-3
Simulate Signal Express VI, 1-6
simulated signals, customizing, 3-3
software (NI resources), A-1
subVIs, 5-2
support, technical, A-1
system requirements, ix

R
related documentation, ix
removing indicators, 2-7
Run button, 1-4
    broken, 2-7, 2-13
running VIs, 1-9
    continuously, 2-6
U
user interface. See front panel

V
virtual instruments. See VIs
VIs, 1-1
  blank, 2-1
  building, 1-1
  customizing menus, 5-4
  icons, 5-2
  new, 2-1
  programmatically controlling, 5-4
  publishing on the Web, 5-4
  running, 1-9
    continuously, 2-6
  subVIs, 5-2
  template, 1-2
voltage, analyzing, 3-7

W
Warning Light VI block diagram
  (figure), 3-10
warning lights, adding, 3-9
Web resources, A-1
While Loop, 2-7
wires
  broken, 2-7, 2-13
  deleting, 1-10
wiring
  objects on the block diagram, 1-8
  tool, 1-8
Write to Measurement File Express VI, 3-12,
  3-14
  saving data, 3-12