MATLAB fundamentals

By now you will probably be wanting to use MATLAB to solve problems of your own. In this chapter we will look in detail at how to write MATLAB statements to solve simple problems. There are two essential requirements for successful MATLAB programming:

- You need to learn the *exact* rules for writing MATLAB statements.
- You need to develop a logical plan of attack for solving particular problems.

This chapter is devoted mainly to the first requirement: learning some basic MATLAB rules. Computer programming is a precise science (some would say it is also an art); you have to enter statements in *exactly* the right way. If you don't, you will get rubbish. There is a saying among computer programmers:

Garbage in, garbage out.

If you give MATLAB a garbage instruction, you will get a garbage result.

Once you have mastered the basic rules in this chapter, you can go on to more interesting and substantial problems.

2.1 Programs

In Chapter 1 we saw some simple examples of how to use MATLAB, by entering single commands or statements at the MATLAB prompt. However, you might want to solve problems which MATLAB can't do in one line, like finding the roots of a quadratic equation (and taking all the special cases into account). A *collection* of statements to solve such a problem is called a *program*. In this section we look at the mechanics of writing and running two short programs, without bothering too much about how they work—explanations will follow in the rest of the chapter.

Cut and paste

Suppose you want to draw the graph of $e^{-0.2x}\sin(x)$ over the domain 0 to 6π, as shown in Figure 2.1. The Windows environment lends itself to nifty *cut and paste* editing, which you would do well to master. Proceed as follows.

From the MATLAB Command Window select the **File** pull-down menu, and from it choose **New/M-file**. (If your mouse is poorly, you can select a menu by holding down the **Alt** key while pressing the underlined letter in the menu title, e.g. **Alt-F** selects the **File** menu. Select further items by moving the highlight bar with the arrow keys, and pressing **Enter**.)

This action opens a Notepad (Untitled) Window. You can regard this as a 'scratch pad' in which to write programs. Now type the following two lines in the Notepad, exactly as they appear here:

```
x = 0 : pi/20 : 6 * pi;
plot(x, exp(-0.2*x) .* sin(x), 'g'),grid
```

Incidentally, that *is* a dot (full stop, period) in front of the second `*` in the second line—explanation later! The additional argument `'g'` for `plot` will draw a green graph, just to be different.

Next, move the mouse pointer (which now looks like a very thin capital I) to the left of the `x` in the first line. Keep the left mouse button down while moving

Figure 2.1: $e^{-0.2x}\sin(x)$

the mouse pointer to the end of the second line. This process is called *dragging*. Both lines should be highlighted at this stage, probably in blue, to indicate that they have been *selected*.

Select the **Edit** menu in the Notepad Window, and click on **Copy** (or use the *hotkey* **Ctrl-C** directly from the Notepad). This action copies the highlighted text to the Windows *clipboard*.

Now go back to the Command Window. Make sure the cursor is positioned at the ≫ prompt (click there if necessary). Select the **Edit** menu, and click on **Paste** (or use the **Ctrl-V** hotkey). The contents of the clipboard will be copied into the Command Window. To execute the two lines in the program, press **Enter**. The graph should appear in a Figure Window.

This process, from highlighting (selecting) text in the Notepad, to copying it into the Command Window, is called 'cut and paste' (more correctly 'copy and paste' here, since the original text is copied from the Notepad, rather than being cut from it). It's well worth practising until you have it right.

If you need to correct the program, go back to the Notepad, click at the position of the error (this moves the *insertion point* to the right place), make the correction, and cut and paste again.

As another example, suppose you have $1 000 saved in the bank. Interest is compounded at the rate of 9% per year. What will your bank balance be after one year? Now, if you want to write a MATLAB program to find your new balance, you must be able to do the problem yourself in principle. Even with a relatively simple problem like this, it often helps first to write down a rough *structure plan*:

1. Get the data (initial balance and interest rate) into MATLAB.
2. Calculate the interest (9% of $1 000, i.e. $90).
3. Add the interest to the balance ($90 + $1 000, i.e. $1 090).
4. Display the new balance.

Go back to the Notepad. To clear out any previous text, select it as usual by dragging, and press the **Del** key. By the way, to de-select highlighted text, click

anywhere outside the selection area. Enter the following program, and then cut and paste it to the Command Window.

```
balance = 1000;
rate = 0.09;
interest = rate * balance;
balance = balance + interest;
disp( 'New balance:' );
disp( balance );
```

When you press **Enter** to run it, you should get the following output in the Command Window:

```
New balance:
        1090
```

How a program works

We will now discuss in detail how the compound interest program works.

The MATLAB system is technically called an *interpreter* (as opposed to a *compiler*). This means that each statement presented to the command line is translated (interpreted) into language the computer understands better, and then *immediately* carried out.

A fundamental concept in MATLAB is how numbers are stored in the computer's *random access memory* (RAM). If a MATLAB statement needs to store a number, space in the RAM is set aside for it. You can think of this part of the memory as a bank of boxes or *memory locations*, each of which can hold only one number at a time. These memory locations are referred to by symbolic names in MATLAB statements. So the statement

```
balance = 1000
```

allocates the number 1 000 to the memory location named `balance`. Since the contents of `balance` may be changed during a session it is called a *variable*.

The statements in our program are therefore interpreted by MATLAB as follows:

1. Put the number 1 000 into variable `balance`.
2. Put the number 0.09 into variable `rate`.
3. Multiply the contents of `rate` by the contents of `balance` and put the answer in `interest`.
4. Add the contents of `balance` to the contents of `interest` and put the answer in `balance`.
5. Display (in the Command Window) the message given in single quotes.
6. Display the contents of `balance`.

In hardly seems necessary to stress this, but these interpreted statements are carried out *in order from the top down*. When the program has finished running, the variables used will have the following values:

```
balance  : 1090
interest : 90
rate     : 0.09
```

Note that the original value of `balance` (1 000) is lost.

Try the following exercises:

1. Run the program as it stands.
2. Go back to the Notepad and change the first statement to read

   ```
   balance = 2000;
   ```

   Cut and paste back to the Command Window, and make sure that you understand what happens now when the program runs.
3. Leave out the line

   ```
   balance = balance + interest;
   ```

   and re-run. Can you explain what happens?
4. Try to rewrite the program so that the original value of `balance` is *not* lost.

A number of questions have probably occurred to you by now, such as

- What names may be used for variables?
- How can numbers be represented?
- What happens if a statement won't fit on one line?
- How can we organize the output more neatly?

These questions will be answered shortly. However, before we write any more complete programs there are some additional basic concepts which need to be introduced.

2.2 Variables and the workspace

Variables

A variable name (like `balance`) must comply with the following two rules:

1. It may consist only of the letters `a`–`z`, the digits `0`–`9` and the underscore (`_`).
2. It must start with a letter.

A variable name may be as long as you like, but MATLAB only remembers the first 19 characters.

Examples of valid variable names: `r2d2` `pay_day`

Examples of invalid names (why?): `pay-day` `2a` `name$` `_2a`

A variable is created simply by assigning a value to it, e.g.

```
a = 98
```

If you attempt to refer to a non-existent variable you may get the error message

```
Undefined function or variable
```

Case sensitivity

MATLAB is *case-sensitive*, which means it normally distinguishes between upper- and lower-case letters. So `balance`, `BALANCE` and `BaLance` are three quite different variables.

If you don't like case-sensitivity, you can switch if off with the command `casesen` (in small letters!). In this mode, `balance`, `BALANCE` and `BaLance` all refer to the *same* variable.

`casesen` 'toggles' the sensitivity (i.e. behaves like an on/off switch). If you're not sure what mode you are in, you can use `casesen on` to enforce sensitivity or `casesen off` to cancel it.

Command and function names are also case-sensitive.

The workspace

Another fundamental concept in MATLAB is the *workspace*. Enter the command `clear`, and then run the compound interest program again. Now enter the command `who`. You should see a list of variables as follows:

```
Your variables are:

ans          balance      interest     rate
```

All the variables you create during a session remain in the workspace until you `clear` them. You can use or change their values at any stage during the session.

The command `who` lists the names of all the variables in your workspace.

The *permanent variable* `ans` contains the value of the last expression evaluated but not assigned to a variable (see below).

The command `whos` lists the size of each variable as well. You should get something like the following, depending on which version of MATLAB you are using:

```
  Name        Size            Elements    Bytes    Density   Complex

   ans        1 by 1                 1        8       Full        No
balance       1 by 1                 1        8       Full        No
interest      1 by 1                 1        8       Full        No
  rate        1 by 1                 1        8       Full        No

Grand total is 4 elements using 32 bytes
```

Each variable occupies 8 *bytes* of storage. A byte is the amount of computer memory required for one character (if you are interested, one byte is the same as 8 *bits*).

These variables each have a *size* of '1 by 1', because they are *scalars*, as opposed to vectors or matrices.

The command `clear` removes all variables from the workspace (except `ans`).

A particular variable can be removed from the workspace, e.g. `clear rate` .

More than one variable can also be cleared, e.g. `clear rate balance` (separate the variable names with spaces, *not commas!*).

When you run a program, any variables created by it remain in the workspace after it has run. This means that existing variables with the same names get overwritten.

Adding commonly used constants to the workspace

If you often use the same physical or mathematical constants in your MATLAB sessions, you can enter them in the Notepad, and save the contents of the Notepad as *file*. At the start of a session you can open the file, and cut and paste it to the Command Window, e.g. you could enter the following constants in an Untitled Notepad:

```
g = 9.8;                          % acceleration due to gravity
avo = 6.023e23;                   % Avogadro's number
e = 2.718281828459045;            % base of natural log
pi_4 = pi / 4;
log10e = log10( e );
bar_to_kP = 101.325;              % atmospheres to kiloPascals
```

To save the contents of the Notepad, select **File/Save** from the Notepad menubar. A *File Dialogue Box* appears. To select a directory, first double-click on the drive letter at the top of the Directories *list box*. Then move down the list that appears, and double-click on the directory of your choice. Having selected a directory, enter a filename in the File Name box, e.g. `myconst.m`, and click on **OK**. The Notepad now has the title `MYCONST.M`.

As a general rule, you can use up to eight letters and characters for the filename, followed by a dot and an *extension* of up to three characters. Although you can use any extension for this example, it makes sense to use the extension `.m`, which has a special meaning in MATLAB as we shall see later in Section 2.12.

To see how to make use of the file `myconst.m` during a later session, first close the Notepad by double-clicking on the *control-menu box* in the top left corner. Now from the Command Window select **File/Open M-file...**. When the File Dialogue Box appears, select the correct directory from the Directories list box (double-click on a name in the list), enter the filename `myconst.m` in the File Name box, and press **Enter**, or click on **OK**. A Notepad entitled `MYCONST.M` will open, hopefully containing the six assignment statements you originally typed.

Select the group of statements, copy to the clipboard, go back to the Command Window, and paste from the clipboard. The group of statements should appear at the command prompt. Press **Enter**. The six variables are now part of the workspace, and are available for the rest of the session, or until you `clear` them.

2.3 Vectors and matrices

As mentioned earlier, the name MATLAB stands for **Mat**rix **lab**oratory because MATLAB has been designed to work with *matrices*. A matrix is a rectangular object (e.g. a *table*) consisting of rows and columns. We will postpone most of the details of proper matrices and how MATLAB works with them until Chapter 6.

A *vector* is a special type of matrix, having only one row, or one column. Vectors are also called *lists* or *arrays* in other programming languages. If you haven't come across vectors officially yet, don't worry—just think of them as lists of numbers.

MATLAB handles vectors and matrices in the same way, but since vectors are easier to think about than matrices, we will look at them first. This will enhance your understanding and appreciation of many other aspects of MATLAB.

Initializing vectors: explicit lists

To get going, try the following short exercises on the command line. (You won't need reminding about the command prompt ≫ any more, so it will no longer appear unless the context absolutely demands it.)

1. Enter a statement like

   ```
   x = [1 3 0 -1 5]
   ```

   Can you see that you have created a vector (list) with five *elements*? (Make sure to leave out the semi-colon so that you can see the list.)
2. Enter the command `disp(x)` to see how MATLAB displays a vector.
3. Enter the command `whos`. Under the heading `Size` you will see that `x` is 1 by 5, which means 1 row and 5 columns. You will also see that the total number of elements is 5.
4. You can put commas instead of spaces between the elements if you like. Try this:

   ```
   a = [5,6,7]
   ```

5. Don't forget the commas (or spaces) between elements, otherwise you could end up with something quite different, e.g.

   ```
   x = [130-15]
   ```

   What do you think this gives?
6. You can use one vector in the list for another one, e.g. type in the following:

   ```
   a = [1 2 3];
   b = [4 5];
   c = [a -b];
   ```

   Can you work out what `c` will look like before displaying it?
7. And what about this?

   ```
   a = [1 3 7];
   a = [a 0 -1];
   ```

8. Enter the following

   ```
   x = [ ]
   ```

   and then enter `whos`. The size of `x` is given as 0 by 0 because `x` is an *empty* vector (or empty matrix). This means `x` is defined, and can be used where a vector or matrix is appropriate without causing an error; however, it has no size or value.

   Making `x` empty is *not* the same as saying `x = 0` (`x` has size 1 by 1), or `clear x` (which removes `x` from the workspace, making it undefined).

   An empty vector may be used to remove elements from a vector (see below on **Subscripts**).

These are all examples of the *explicit list* method of *initializing* vectors. Remember the following important rules:

1. Elements in the list must be enclosed in *square* not round brackets.
2. Elements in the list must be separated *either by spaces or by commas*.

Initializing vectors: the colon operator

A vector can also be generated (initialized) with the *colon operator*, as we have seen. Enter the following statements:

```
x = 1:10
```

(elements are the integers 1, 2, ..., 10);

```
x = 1:0.5:4
```

(elements are the values 1, 1.5, ..., 4 in increments of 0.5—note that if the colons separate three values, the *middle* value is the increment);

```
x = 10:-1:1
```

(elements are the integers 10, 9, ..., 1, since the increment is *negative*);

```
x = 1:2:6
```

(elements are 1, 3, 5; note that when the increment is positive but not equal to 1, the last element is not allowed to exceed the value after the second colon);

```
x = 0:-2:-5
```

(elements are $0, -2, -4$; note that when the increment is negative but not equal to -1, the last element is not allowed to be less than the value after the second colon);

```
x = 1:0
```

(a complicated way of generating an empty vector!).

`linspace`

The function `linspace` can be used to initialize a vector of equally spaced values, e.g.

```
linspace(0, pi/2, 10)
```

creates a vector of 10 equally spaced points from 0 to $\pi/2$ (inclusive).

Transposing vectors

All these vectors are *row vectors*. Each has one row and several columns. To generate the *column vectors* that are often needed in mathematics, you need to *transpose* the vectors, i.e. you need to interchange their rows and columns. This is done with the single quote, or *apostrophe* (`'`), which is the nearest MATLAB can get to the mathematical dash (′) that is often used to indicate the transpose.

Enter `x = 1:5` and then enter `x'` to transpose it.

Or you can do it like this directly:

```
y = [1 4 8 0 -1]'
```

Subscripts

We can refer to particular elements of a vector by means of *subscripts*. Try the following:

1. Enter `r = rand(1,7)`
   This gives you a row vector of seven *random numbers*.
2. Now enter `r(3)`
   This will give you the *third* element of `r`. The number 3 is the *subscript*.
3. Now enter `r(2:4)`
   This should give you the second, third and fourth elements.
4. What about `r(1:2:7)`?
5. And `r([1 7 2 6])`?
6. You can use an empty vector to *remove* elements from a vector, e.g.

   ```
   r([1 7 2]) = [ ]
   ```

   will remove elements 1, 7 and 2.

To summarize:

A subscript is indicated inside *round* brackets.
A subscript may be a scalar or a vector.
In MATLAB subscripts always start at 1.
Fractional subscripts are always rounded *down*, e.g. `x(1.9)` refers to element `x(1)`.

Matrices

A *matrix* may be thought of as a table consisting of rows and columns. You enter a matrix just as you do a vector, except that a semi-colon is used to indicate the end of a row, e.g. the statement

```
a = [1 2 3; 4 5 6]
```

results in

```
a =
     1     2     3
     4     5     6
```

A matrix may be transposed, e.g. with a initialized as above, the statement `a'` results in

```
a =
     1     4
     2     5
     3     6
```

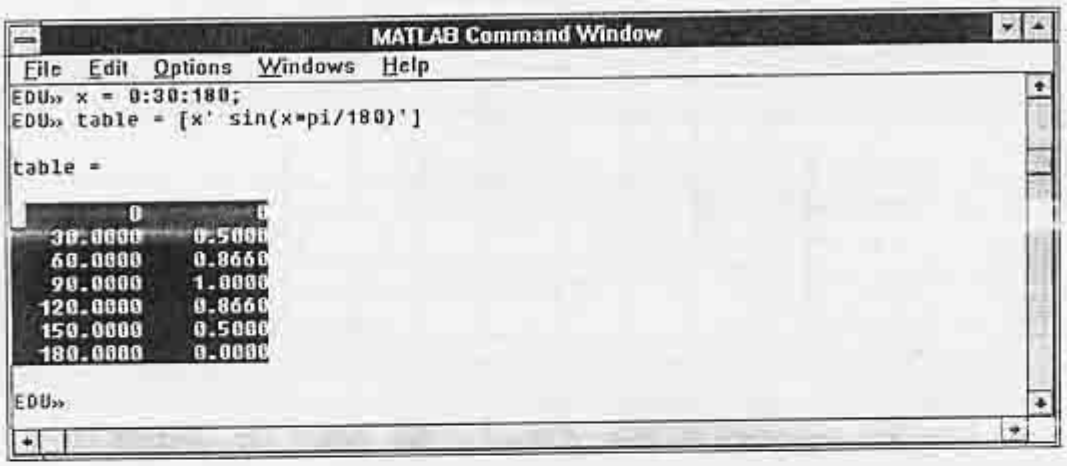

Figure 2.2: Selected output in the Command Window

A matrix can be constructed from column vectors of the same length. The statements

```
x = 0:30:180;
table = [x' sin(x*pi/180)']
```

result in

```
table =
         0         0
   30.0000    0.5000
   60.0000    0.8660
   90.0000    1.0000
  120.0000    0.8660
  150.0000    0.5000
  180.0000    0.0000
```

Editing output with cut and paste

You can also use cut and paste techniques to tidy up the output from MATLAB statements, if this is necessary for some sort of presentation. Generate the table of angles and sines as shown above. Select all seven rows of output in the Command Window, as shown in Figure 2.2. Copy the selected output to the clipboard, use **File/New/M-file** to create an Untitled Notepad, and paste the clipboard contents to it. The output will appear in the Notepad. You can then edit the output, e.g. by inserting text headings above each column (this is easier than trying to get headings to line up over the columns with a `disp` statement). The edited output can in turn be pasted into a report, or printed as it is (the **File** menu has a number of printing options).

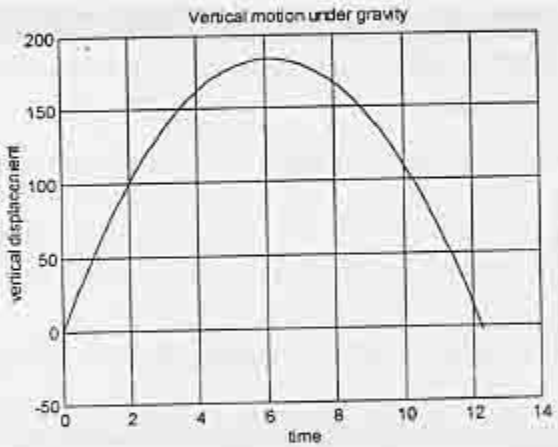

Figure 2.3: Distance-time graph of stone thrown vertically upward

2.4 Vertical motion under gravity

If a stone is thrown vertically upward with an initial speed u, its vertical displacement s after a time t has elapsed is given by the formula $s = ut - gt^2/2$, where g is the acceleration due to gravity. Air resistance has been ignored. We would like to compute the value of s over a period of about 12.3 seconds at intervals of 0.1 seconds, and to plot the distance-time graph over this period, as shown in Figure 2.3. The structure plan for this problem is as follows:

1. Get the data (g, u and t) into MATLAB.
2. Calculate the value of s according to the formula.
3. Plot the graph of s against t.
4. Stop.

This plan may seem trivial to you, and a waste of time writing down. Yet you would be surprised how many beginners, preferring to rush straight to the computer, start with step 2 instead of step 1. It is well worth developing the mental discipline of structure-planning your program first. You could even use cut and paste to plan as follows:

1. Type the structure plan into the Notepad.
2. Paste a second copy of the plan directly below the first.
3. Translate each line in the second copy into a MATLAB statement, or statements (add % comments as in the example below).
4. Finally, paste all the translated MATLAB statements into the Command Window, and run them.
5. If necessary, go back to the Notepad to make corrections, and re-paste the corrected statements to the Command Window.

You might like to try this as an exercise, before looking at the final program, which is as follows:

```
% Vertical motion under gravity
g = 9.8;                          % acceleration due to gravity
u = 60;                           % initial velocity (metres/sec)
t = 0 : 0.1 : 12.3;               % time in seconds
s = u * t - g / 2 * t .^ 2;       % vertical displacement in metres

plot(t, s, 'w'), title( 'Vertical motion under gravity' ), ...
     xlabel( 'time' ), ylabel( 'vertical displacement' ), grid
disp( [t' s'] )                   % display a table
```

The graphical output is shown in Figure 2.3.

Note the following points:

1. Anything in a line following the percentage symbol % is ignored by MATLAB and may be used as a comment (description).
2. The statement `t = 0 : 0.1 : 12.3` sets up a vector.
3. The formula for `s` is evaluated for every element of the vector `t`, making another vector.
4. The expression `t.^2` squares *each element* in `t`. This is called an *array operation*, and is different to squaring the vector itself, which is a *matrix* operation, as we shall see later.
5. More than one statement can be entered on the same line if the statements are separated by commas.
6. A statement or group of statements can be continued to the next line with an *ellipsis* of three or more dots ....
7. The statement `disp([t' s'])` first transposes the row vectors `t` and `s` into columns, and constructs a matrix from these two columns, which is then displayed.

You might want to save the program under a helpful name, like `throw.m` if you think you might come back to it. In that case, it would be worth keeping the structure plan as part of the file; just insert % symbols in front of each line of the structure plan (use cut and paste). This way, the structure plan reminds you what the program does when you look at it again after some months.

2.5 Programming style

Programs that are written any old how, while they may do what is required, can be difficult to follow when read a few months later, in order to correct or update them (and programs that are worth writing will need to be maintained in this way).

Some programmers delight in writing terse and obscure code; there is at least one annual competition for the most incomprehensible C program. A large body of responsible programmers, however, believe it is extremely important to develop the art of writing programs which are well laid out, with all the logic clearly described. This is known as *programming style*, and is demonstrated in many of the programs in this book. Guidelines for good style are laid out in the Epilogue.

The program in Section 2.4 has been written with this in mind:

- There is a comment at the beginning describing what the program does.
- All the variables have been described with comments.

- Spaces have been used on either side of the equal signs and the operators, e.g. as in `u * t` .
- Blank lines may be used to separate different parts of the program.

You may like to develop your own style; the point is that you *must* pay attention to readability.

2.6 Operators, expressions and statements

Any program worth its salt actually does something. What it basically does is to evaluate *expressions*, such as

```
u * t - g / 2 * t .^ 2
```

and to execute (carry out) statements, such as

```
balance = balance + interest
```

The MATLAB *User's Guide* states 'MATLAB is an *expression* based language. It interprets and evaluates typed expressions.' Expressions are constructed from a variety of things, such as numbers, variables, and operators. First we need to look at numbers.

Numbers

Numbers can be represented in MATLAB in the usual decimal form (*fixed point*), with an optional decimal point, e.g.

```
1.2345    -123    .0001
```

A number may also be represented in *scientific notation*, e.g. 1.2345×10^9 may be represented in MATLAB as `1.2345e9`. This is also called *floating point* notation. The number has two parts: the *mantissa*, which may have an optional decimal point (`1.2345` in this example) and the *exponent* (`9`), which must be an integer (signed or unsigned). Mantissa and exponent must be separated by the letter `e` (or `E`). The mantissa is multiplied by the power of 10 indicated by the exponent.

Note that the following is *not* scientific notation: `1.2345*10^9`. It is actually an *expression* involving two arithmetic operations (`*` and `^`) and therefore more time consuming. It's also rather uncool. You should try to avoid it.

Use scientific notation if the numbers are very small or very large, since there's less chance of making a mistake, e.g. represent 0.000000001 as `1e-9`.

On computers using standard floating point arithmetic, numbers are represented to approximately 16 significant decimal digits. The *relative accuracy* of numbers is given by the function `eps`, which is defined as the distance between 1.0 and the next largest number. Enter `eps` to see its value on your computer.

The range of numbers is roughly $\pm 10^{-308}$ to $\pm 10^{308}$. Precise values are returned by the MATLAB functions `realmin` and `realmax`.

Operation	*Algebraic form*	MATLAB
Addition	$a + b$	`a + b`
Subtraction	$a - b$	`a - b`
Multiplication	$a \times b$	`a * b`
Right division	a/b	`a / b`
Left division	b/a	`a \ b`
Exponentiation	a^b	`a ^ b`

Table 2.1: Arithmetic operations between two scalars

Exercises

Enter the following numbers at the command prompt in scientific notation (answers are below):

1.234×10^5, -8.765×10^{-4}, 10^{-15}, -10^{12}.

`(1.234e5, -8.765e-4, 1e-15, -1e12)`

Arithmetic operators

The evaluation of expressions is achieved by means of *arithmetic operators*. The arithmetic operations on two *scalar* constants or variables are shown in Table 2.1. Operators operate on *operands* (`a` and `b` in the table.)

Left division seems a little curious: divide the right operand by the *left* operand. For *scalar* operands the expressions `1/3` and `3\1` have the same numerical value (a colleague of mine speaks the latter as '3 under 1'). However, *matrix* left division has an entirely different meaning, as we shall see later.

Precedence of operators

Precedence	*Operator*
1	Parentheses (round brackets)
2	Exponentiation, left to right
3	Multiplication and division, left to right
4	Addition and subtraction, left to right

Table 2.2: Precedence of arithmetic operations

Several operations may be combined in one expression, e.g. `g * t ^ 2`. MATLAB has strict rules about which operations are performed first in such cases; these are called *precedence* rules. The precedence rules are shown in Table 2.2. Note that parentheses (round brackets) have the highest precedence. Note also the difference between round and square brackets. Round brackets are used to alter the precedence of operators, and to denote subscripts, while square brackets are used to initialize vectors.

When operators in an expression have the same precedence the operations are carried out from left to right. So `a / b * c` is evaluated as `(a / b) * c` and *not* as `a / (b * c)`.

Exercises

1. Evaluate the following MATLAB expressions yourself before checking the answers in MATLAB:

```
1 + 2 * 3
4 / 2 * 2
1+2 / 4
1 + 2\4
2*2 ^ 3
2 * 3 \ 3
2 ^ (1 + 2)/3
1/2e-1
```

2. Use MATLAB to evaluate the following expressions. Answers are in brackets.

   (a) $\frac{1}{2 \times 3}$ (0.1667)

   (b) $2^{2 \times 3}$ (64)

   (c) $1.5 \times 10^{-4} + 2.5 \times 10^{-2}$ (0.0252; use scientific notation)

The colon operator

The colon operator has a lower precedence than `+` as the following shows:

```
1+1:5
```

The addition is first carried out, and then a vector with elements 2, ..., 5 is initialized.

You may be surprised at the following:

```
1+[1:5]
```

Were you? The value 1 is added to *each element* of the vector `1:5`. In this context, the addition is called an *array operation*, because it operates on each element of the vector (array). Array operations are discussed below.

See Appendix B for a complete list of MATLAB operators and their precedences.

The transpose operator

The transpose operator has the highest precedence. Try

```
1:5'
```

5 is transposed first (into itself since it is a scalar!), and then a row vector is formed. Use square brackets if you want to transpose the whole vector:

```
[1:5]'
```

Array operations

Enter the following statements at the command line:

```
a = [2 4 8];
b = [3 2 2];
a .* b
a ./ b
```

`a .* b` is called an *array operation*, or an *element-by-element* operation because the operation (multiplication here) is performed element by element. (An array is another name for a vector.) The operator `.*` is defined so that `a .* b` results in the following vector (called the *array product*):

```
[a(1)*b(1)  a(2)*b(2)  a(3)*b(3)]
```

A common application of element-by-element multiplication is in finding the *scalar product* (also called the *dot* product) of two vectors `x` and `y`, which is defined as

$$x \cdot y = \sum_i x_i y_i .$$

The MATLAB function `sum(z)` finds the sum of the elements of the vector `z`, so the statement `sum(a .* b)` will find the scalar product of `a` and `b` (30 in the example above).

You will have seen that `a ./ b` gives element-by-element division.

Now try `[2 3 4] .^ [4 3 1]`. The ith element of the first vector is raised to the power of the ith element of the second vector.

The period (dot) is necessary for the array operations of multiplication, division and exponentiation, because these operations are defined differently for matrices; the operation is then called a *matrix operation* (see Chapter 6).

With `a` and `b` as defined above, try `a + b` and `a - b`. For addition and subtraction, array operations and matrix operations are the same, so we don't need the period to distinguish between them.

When array operations are applied to two vectors, they must both be the same size.

Array operations also apply to operations between a scalar and a non-scalar. Check this with `3 .* a` and `a .^ 2`. Multiplication and division operations between scalars and non-scalars can be written with or without the period, i.e. if `a` is a vector, `3 .* a` is the same as `3 * a`.

Exercises

Use MATLAB array operations to do the following:

1. Add 1 to each element of the vector `[2 3 -1]`.
2. Multiply each element of the vector `[1 4 8]` by 3.
3. Find the array product of the two vectors `[1 2 3]` and `[0 -1 1]`. (Answer: `[0 -1 3]`)
4. Square each element of the vector `[2 3 1]`.

Expressions

An *expression* is a formula consisting of variables, numbers, operators, and function names. An expression is evaluated when you enter it at the MATLAB prompt, e.g. evaluate 2π as follows:

```
2 * pi
```

MATLAB's response is:

```
ans =
    6.2832
```

Note that MATLAB uses the *permanent variable* called `ans` (which stands for answer) for the last expression to be evaluated but not assigned to a variable.

If an expression is terminated with a semi-colon (;) its value is not displayed, although it is still evaluated and stored in `ans`.

Statements

MATLAB *statements* are frequently of the form

variable = expression

e.g.

```
s = u * t - g / 2 * t .^ 2;
```

This is an example of an *assignment* statement, because the value of the expression on the *right* is *assigned* to the variable (`s`) on the *left*. Assignment always works in this way.

Note that the object on the left-hand side of the assignment must be a variable name. A common mistake is to get the statement the wrong way round, like

```
a + b = c
```

Basically anything that you enter on a line, which MATLAB accepts, is a statement, so a statement could be an assignment, a command, or simply an expression, e.g.

```
x = 29;                 % assignment
clear                   % command
pi/2                    % expression
```

As we have seen, a semi-colon at the end of an assignment or expression suppresses any output. This is useful for suppressing irritating output of intermediate results (or large matrices).

A statement which is too long to fit onto one line may be continued to the next line with an *ellipsis* of at least three dots, e.g.

```
x = 3 * 4 - 8 ....
    / 2 ^ 2;
```

Statements on the same line may be separated by commas (output not suppressed) or semi-colons (output suppressed), e.g.

```
a = 2; b = 3, c = 4;
```

Note that the commas and semi-colons are not technically part of the statements; they are *separators*.

Statements may involve array operations, in which case the variable on the left-hand side may become a vector or matrix.

Statements and commands

The distinction between MATLAB *statements* and *commands* can be a little fuzzy, since either can be entered on the command line. However, it is helpful to think of commands as changing the general environment in some way, e.g. `load`, `save`, and `clear`. Statements, on the other hand, do the sort of thing we usually associate with programming, such as evaluating expressions and carrying out assignments, making decisions (`if`), and repeating (`for`).

Vectorization of formulae

With array operations you can easily evaluate a formula repeatedly for a whole lot of data. This is one of MATLAB's most useful and powerful features, and you should always be looking for ways to exploit it.

As an example, consider the formula for compound interest. An amount of money A invested over a period of n years with an annual interest rate of r grows to an amount $A(1+r)^n$. Suppose we want to calculate final balances for investments of \$750, \$1 000, \$3 000, \$5 000 and \$11 999, over 10 years, with an interest rate of 9%. The following program (`comp.m`) uses array operations on a vector of initial investments to do this.

```
format bank
A = [750 1000 3000 5000 11999];
r = 0.09;
n = 10;
B = A * (1 + r) ^ n;
disp( [A' B'] )
```

Output:

```
   750.00     1775.52
  1000.00     2367.36
  3000.00     7102.09
  5000.00    11836.82
 11999.00    28406.00
```

Note:

1. In the statement `B = A * (1 + r) ^ n` , the expression `(1 + r) ^ n` is evaluated first, because exponentiation has a higher precedence than multiplication. This is a scalar operation.
2. After that the array product between the vector `A` and the scalar `(1 + r) ^ n` is formed.
3. `*` may be used instead of `.*` because the array multiplication is between a scalar and a non-scalar (although `.*` would not be wrong).
4. A table is displayed, with columns given by the transposes of `A` and `B`.

This process is called *vectorization* of a formula.

See if you can adjust the program `comp.m` to find the balances for a single amount `A` ($1 000) over periods of 1, 5, 10, 15 and 20 years. **Hint:** use a vector for `n`: `[1 5 10 15 20]` .

Danger!

I have seen quite a few beginners make the following mistake. Run the `comp.m` program, and then enter the statement

```
A(1+r)^n
```

at the prompt. The output is a *scalar:*

```
5.6314e+028
```

Any ideas?

Well, the basic error is that the multiplication symbol `*` has been left out after the `A`. However, MATLAB still obligingly gives a result! This is because round brackets after the `A` signify an element of a vector, in this case element number `1+r`, which rounds down to 1, since `r` has the value 0.09 (fractional subscripts are always rounded down). `A(1)` has the value 750, so MATLAB obediently calculates 750^{10}.

The curious thing is that the same wrong answer is given even if `A` is a scalar, e.g. `A = 750`, because MATLAB treats a scalar as a vector with one element (as `whos` demonstrates).

Exercises

1. Evaluate the following MATLAB expressions yourself (don't use MATLAB to help!). Answers are in brackets.

```
2 / 2 * 3                (3)
2 / 3 ^ 2                (2/9)
(2 / 3) ^ 2              (4/9)
2 + 3 * 4 - 4            (10)
2 ^ 2 * 3 / 4 + 3        (6)
2 ^ (2 * 3) / (4 + 3)    (64/7)
2 * 3 + 4                (10)
2 ^ 3 ^ 2                (64)
-4 ^ 2                   (−16; ^ has higher precedence than -)
```

2. Use MATLAB to evaluate the following expressions. Answers are in brackets again.
   (a) $\sqrt{2}$ (1.4142; use `sqrt` or `^0.5`)
   (b) $\frac{3+4}{5+6}$ (0.6364; use brackets)
   (c) the sum of 5 and 3 divided by their product (0.5333)
   (d) 2^{3^2} (512)
   (e) the square of 2π (39.4784; use `pi`)
   (f) $2\pi^2$ (19.7392)
   (g) $1/\sqrt{2\pi}$ (0.3989)
   (h) $\frac{1}{2\sqrt{\pi}}$ (0.2821)
   (i) the cube root of the product of 2.3 and 4.5 (2.1793)
   (j) $\frac{1-\frac{2}{3+2}}{1+\frac{2}{3-2}}$ (0.2)
   (k) $1000(1+0.15/12)^{60}$ (2107.2, e.g. $1 000 deposited for 5 years at 15% per year, with the interest compounded monthly)
   (l) $(0.0000123+5.678\times10^{-3})\times0.4567\times10^{-4}$ (2.5988×10^{-7}; use scientific notation, e.g. `1.23e-5` ...; do *not* use `^`)

3. Be careful of using too many brackets unnecessarily in an expression. Can you spot the errors in the following expression (test your corrected version with MATLAB):

```
(2(3+4)/(5*(6+1))^2
```

4. Set up a vector `n` with elements 1, 2, 3, 4, 5. Use MATLAB array operations on the vector `n` to set up the following four vectors, each with five elements:
   (a) 2, 4, 6, 8, 10
   (b) 1/2, 1, 3/2, 2, 5/2
   (c) 1, 1/2, 1/3, 1/4, 1/5
   (d) $1, 1/2^2, 1/3^2, 1/4^2, 1/5^2$

5. Suppose `a` and `b` are defined as follows:

```
a = [2 -1 5 0];
b = [3 2 -1 4];
```

Evaluate by hand the vector `c` in the following statements. Check your answers

(g) `c = (2).^b+a;`
(h) `c = 2*b/3.0.*a;`
(i) `c = b*2.*a;`

6. Water freezes at 32° and boils at 212° on the Fahrenheit scale. If C and F are Celsius and Fahrenheit temperatures, the formula

$$F = 9C/5 + 32$$

converts from Celsius to Fahrenheit.

Use the MATLAB command line to convert a temperature of 37° C (normal human temperature) to Fahrenheit (98.6°).

7. Engineers often have to convert from one unit of measurement to another; this can be tricky sometimes. You need to think through the process carefully.

For example, convert 5 acres to hectares, given that an acre is 4 840 square yards, a yard is 36 inches, an inch is 2.54 cm, and a hectare is 10 000 m^2. The best approach is to develop a formula to convert x acres to hectares. You can do this as follows.

$$\begin{aligned}
\text{one square yard} &= (36 \times 2.54)^2 \text{ cm}^2 \\
\text{so one acre} &= 4\,840 \times (36 \times 2.54)^2 \text{ cm}^2 \\
&= 0.4047 \times 10^8 \text{ cm}^2 \\
&= 0.4047 \text{ hectares} \\
\text{so } x \text{ acres} &= 0.4047x \text{ hectares}
\end{aligned}$$

Once you've got the formula, MATLAB can do the rest:

```
x = 5;                  % acres
h = 0.4047 * x;         % hectares
disp( h )
```

Develop formulae for the following conversions, and use some MATLAB statements to find the answers, which are in brackets.

(a) Convert 22 yards (an imperial cricket pitch) to metres. (20.117 metres)
(b) One pound (weight) = 454 grams. Convert 75 kilograms to pounds. (165.20 pounds)
(c) On 2 March 1995 rates of exchange were: one pound sterling = 5.63 SA rand, and one SA rand = 0.279 US dollars. Convert 100 US dollars to pounds sterling. (63.66 pounds)
(d) Convert 49 metres/second (terminal velocity for a falling man-shaped object) to km/hour. (176.4 km/hour)
(e) One atmosphere pressure = 14.7 pounds per square inch (psi) = 101.325 kilo Pascals (kPa). Convert 40 psi to kPa. (275.71 kPa)
(f) One calorie = 4.184 joules. Convert 6.25 kilojoules to calories. (1.494 kilocalories)

2.7 Output

There are two basic ways of getting output from MATLAB:

Mathematical Programming

OPTIMIZATION PROBLEMS

In an *optimization problem* one seeks to maximize or minimize a specific quantity, called the *objective*, which depends on a finite number of input variables. These variables may be independent of one another, or they may be related through one or more *constraints*.

Example 1.1 The problem

$$\begin{aligned} \text{minimize:} \quad & z = x_1^2 + x_2^2 \\ \text{subject to:} \quad & x_1 - x_2 = 3 \\ & x_2 \geq 2 \end{aligned}$$

is an optimization problem for the objective z. The input variables are x_1 and x_2, which are constrained in two ways: x_1 must exceed x_2 by 3, and also x_2 must be greater than or equal to 2. It is desired to find values for the input variables which minimize the sum of their squares, subject to the limitations imposed by the constraints.

A *mathematical program* is an optimization problem in which the objective and constraints are given as mathematical functions and functional relationships (as they are in Example 1.1). Mathematical programs treated in this book have the form

$$\begin{aligned} \text{optimize:} \quad & z = f(x_1, x_2, \ldots, x_n) \\ \text{subject to:} \quad & \left.\begin{array}{l} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \cdots\cdots\cdots\cdots \\ g_m(x_1, x_2, \ldots, x_n) \end{array}\right\} \begin{array}{c} \leq \\ = \\ \cdots \\ \geq \end{array} \left\{\begin{array}{l} b_1 \\ b_2 \\ \cdots \\ b_m \end{array}\right. \end{aligned} \tag{1.1}$$

Each of the m constraint relationships in (1.1) involves one of the three signs $\leq$, $=$, $\geq$. *Unconstrained* mathematical programs are covered by the formalism (1.1) if each function g_i is chosen as zero and each constant b_i is chosen as zero.

LINEAR PROGRAMS

A mathematical program (1.1) is *linear* if $f(x_1, x_2, \ldots, x_n)$ and each $g_i(x_1, x_2, \ldots, x_n)$ $(i = 1, 2, \ldots, m)$ are linear in each of their arguments—that is, if

$$f(x_1, x_2, \ldots, x_n) = c_1x_1 + c_2x_2 + \cdots + c_nx_n \tag{1.2}$$

and

$$g_i(x_1, x_2, \ldots, x_n) = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \tag{1.3}$$

where c_j and a_{ij} $(i = 1, 2, \ldots, m;\ j = 1, 2, \ldots, n)$ are known constants.

Any other mathematical program is *nonlinear*. Thus, Example 1.1 describes a nonlinear program, in view of the form of z.

INTEGER PROGRAMS

An *integer program* is a linear program with the additional restriction that the input variables be integers. It is not necessary that the coefficients in (1.2) and (1.3), and the constants in (1.1), also be integers, but this will very often be the case.

QUADRATIC PROGRAMS

A *quadratic program* is a mathematical program in which each constraint is linear—that is, each constraint function has the form (1.3)—but the objective is of the form

$$f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_i x_j + \sum_{i=1}^{n} d_i x_i \tag{1.4}$$

where c_{ij} and d_i are known constants.

The program given in Example 1.1 is quadratic. Both constraints are linear, and the objective has the form (1.4), with $n = 2$ (two variables), $c_{11} = 1$, $c_{12} = c_{21} = 0$, $c_{22} = 1$, and $d_1 = d_2 = 0$.

PROBLEM FORMULATION

Optimization problems most often are stated verbally. The solution procedure is to model the problem with a mathematical program and then solve the program by the techniques described in Chapters 2 through 15. The following approach is recommended for transforming a word problem into a mathematical program:

STEP 1 Determine the quantity to be optimized and express it as a mathematical function. Doing so serves to define the input variables.

STEP 2 Identify all stipulated requirements, restrictions, and limitations, and express them mathematically. These requirements constitute the constraints.

STEP 3 Express any hidden conditions. Such conditions are not stipulated explicitly in the problem but are apparent from the physical situation being modeled. Generally they involve nonnegativity or integer requirements on the input variables.

SOLUTION CONVENTION

In any mathematical program, we seek *a* solution. If a number of equally optimal solutions exist, then any one will do. *There is no preference between equally optimal solutions if there is no preference stipulated in the constraints.*

Solved Problems

1.1 The Village Butcher Shop traditionally makes its meat loaf from a combination of lean ground beef and ground pork. The ground beef contains 80 percent meat and 20 percent fat, and costs the shop 80¢ per pound; the ground pork contains 68 percent meat and 32 percent fat, and costs 60¢ per pound. How much of each kind of meat should the shop use in each pound of meat loaf if it wants to minimize its cost and to keep the fat content of the meat loaf to no more than 25 percent?

The objective is to minimize the cost (in cents), z, of a pound of meat loaf, where

z = 80 times the poundage of ground beef used plus 60 times the poundage of ground pork used

Defining

x_1 = poundage of ground beef used in each pound of meat loaf
x_2 = poundage of ground pork used in each pound of meat loaf

we express the objective as

$$\text{minimize:} \quad z = 80x_1 + 60x_2 \tag{1}$$

Each pound of meat loaf will contain $0.20x_1$ pound of fat contributed from the beef and $0.32x_2$ pound of fat contributed from the pork. The total fat content of a pound of meat loaf must be no greater than 0.25 lb. Therefore,

$$0.20x_1 + 0.32x_2 \leq 0.25 \tag{2}$$

The poundages of beef and pork used in each pound of meat loaf must sum to 1; hence,

$$x_1 + x_2 = 1 \tag{3}$$

Finally, the butcher shop may not use negative quantities of either meat, so that two hidden constraints are $x_1 \geq 0$ and $x_2 \geq 0$. Combining these conditions with (1), (2), and (3), we obtain

$$\begin{aligned} &\text{minimize:} && z = 80x_1 + 60x_2 \\ &\text{subject to:} && 0.20x_1 + 0.32x_2 \leq 0.25 \\ & && x_1 + x_2 = 1 \\ &\text{with:} && \text{all variables nonnegative} \end{aligned} \tag{4}$$

System (4) is a linear program. As there are only two variables, a graphical solution may be given.

1.2 Solve the linear program (4) of Problem 1.1 graphically.

See Fig. 1-1. The *feasible region*—the set of points (x_1, x_2) satisfying all the constraints, including the nonnegativity conditions—is the heavy line segment in the figure. To determine z^*, the minimal value of z, we arbitrarily choose values of z and plot the graphs of the associated objectives. By choosing $z = 70$ and then $z = 75$, we obtain the objectives

$$70 = 80x_1 + 60x_2 \qquad \text{and} \qquad 75 = 80x_1 + 60x_2$$

respectively. Their graphs are the dashed lines in Fig. 1-1. It is seen that z^* will be assumed at the upper endpoint of the feasible segment, which is the intersection of the two lines

$$0.20x_1 + 0.32x_2 = 0.25 \qquad \text{and} \qquad x_1 + x_2 = 1$$

Simultaneous solution of these equations gives $x_1^* = 7/12$, $x_2^* = 5/12$; hence,

$$z^* = 80(7/12) + 60(5/12) = 71.67¢$$

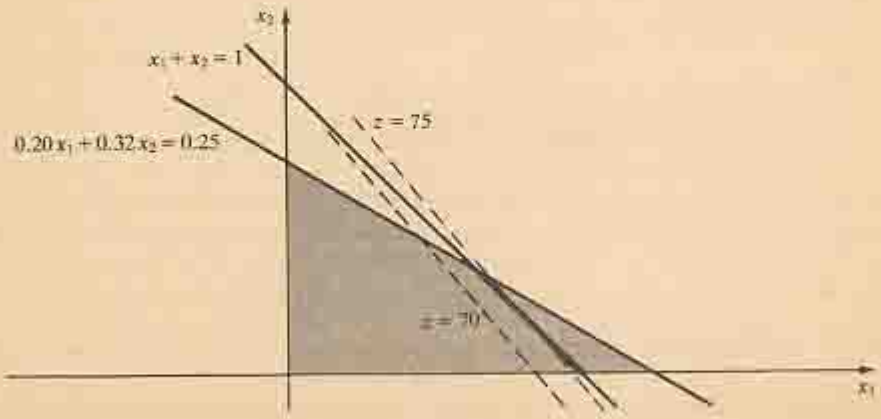

Fig. 1-1

1.3 A furniture maker has 6 units of wood and 28 h of free time, in which he will make decorative screens. Two models have sold well in the past, so he will restrict himself to those two. He estimates that model I requires 2 units of wood and 7 h of time, while model II requires 1 unit of wood and 8 h of time. The prices of the models are \$120 and \$80, respectively. How many screens of each model should the furniture maker assemble if he wishes to maximize his sales revenue?

The objective is to maximize revenue (in dollars), which we denote as z:

z = 120 times the number of model I screens produced plus 80 times the number of model II screens produced

Letting

x_1 = number of model I screens to be produced
x_2 = number of model II screens to be produced

we express the objective as

$$\text{maximize:} \quad z = 120x_1 + 80x_2 \tag{1}$$

The furniture maker is subject to a wood constraint. As each model I requires 2 units of wood, $2x_1$ units must be allocated to them; likewise, $1x_2$ units of wood must be allocated to the model II screens. Hence the wood constraint is

$$2x_1 + x_2 \leq 6 \tag{2}$$

The furniture maker also has a time constraint. The model I screens will consume $7x_1$ hours and the model II screens $8x_2$ hours; and so

$$7x_1 + 8x_2 \leq 28 \tag{3}$$

It is obvious that negative quantities of either screen cannot be produced, so two hidden constraints are $x_1 \geq 0$ and $x_2 \geq 0$. Furthermore, since there is no revenue derived from partially completed screens, another hidden condition is that x_1 and x_2 be integers. Combining these hidden conditions with (*1*), (*2*), and (*3*), we obtain the mathematical program

$$\begin{aligned} &\text{maximize:} && z = 120x_1 + 80x_2 \\ &\text{subject to:} && 2x_1 + x_2 \leq 6 \\ &&& 7x_1 + 8x_2 \leq 28 \\ &\text{with:} && \text{all variables nonnegative and integral} \end{aligned} \tag{4}$$

System (*4*) is an integer program. As there are only two variables, a graphical solution may be given.

1.4 Give a graphical solution of the integer program (*4*) of Problem 1.3.

See Fig. 1-2. The feasible region is the set of integer points (marked by crosses) within the shaded area. The dashed lines are the graphs of the objective function when z is arbitrarily given the values 240, 330, and 380. It is seen that the z-line through the point (3, 0) will furnish the desired maximum: thus, the furniture maker should assemble three model I screens and no model II screens, for a maximum revenue of

$$z^* = 120(3) + 80(0) = \$360$$

Observe that this optimal answer is *not* achieved by first solving the associated linear program (the same problem without the integer constraints) and then moving to the closest feasible integer point. In fact, the feasible region for the associated linear program is the shaded area of Fig. 1-2; so the optimal solution occurs at the circled corner point. But at the closest feasible integer point, (2, 1), the objective function has the value $z = 120(2) + 80(1) = \$320$ or \$40 less than the true optimum.

An alternate solution procedure for Problem 1.3 is given in Problem 7.8.

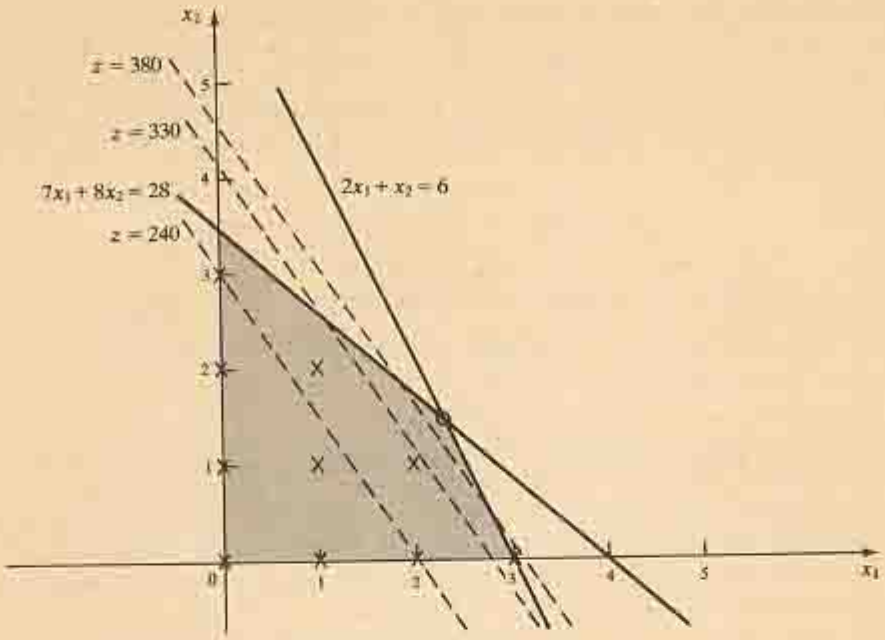

Fig. 1-2

1.5 Universal Mines Inc. operates three mines in West Virginia. The ore from each mine is separated into two grades before it is shipped; the daily production capacities of the mines, as well as their daily operating costs, are as follows:

	High-Grade Ore, tons/day	Low-Grade Ore, tons/day	Operating Cost, $1000/day
Mine I	4	4	20
Mine II	6	4	22
Mine III	1	6	18

Universal has committed itself to deliver 54 tons of high-grade ore and 65 tons of low-grade ore by the end of the week. It also has labor contracts that guarantee employees in each mine a full day's pay for each day or fraction of a day the mine is open. Determine the number of days each mine should be operated during the upcoming week if Universal Mines is to fulfill its commitment at minimum total cost.

Let x_1, x_2, and x_3, respectively, denote the numbers of days that mines I, II, and III will be operated during the upcoming week. Then the objective (measured in units of $1000) is

$$\text{minimize:} \quad z = 20x_1 + 22x_2 + 18x_3 \tag{1}$$

The high-grade ore requirement is

$$4x_1 + 6x_2 + x_3 \geq 54 \tag{2}$$

and the low-grade ore requirement is

$$4x_1 + 4x_2 + 6x_3 \geq 65 \tag{3}$$

As no mine may operate a negative number of days, three hidden constraints are $x_1 \geq 0$, $x_2 \geq 0$, and $x_3 \geq 0$. Moreover, as no mine may operate more than 7 days in a week, three other hidden constraints are $x_1 \leq 7$, $x_2 \leq 7$, and $x_3 \leq 7$. Finally, in view of the labor contracts, Universal Mines has nothing to gain in operating a mine for part of a day; consequently, x_1, x_2, and x_3 are required to be

integral. Combining the hidden conditions with (1), (2), and (3), we obtain the mathematical program

$$\begin{aligned} \text{minimize:} \quad & z = 20x_1 + 22x_2 + 18x_3 \\ \text{subject to:} \quad & 4x_1 + 6x_2 + x_3 \geq 54 \\ & 4x_1 + 4x_2 + 6x_3 \geq 65 \\ & x_1 \leq 7 \\ & x_2 \leq 7 \\ & x_3 \leq 7 \\ \text{with:} \quad & \text{all variables nonnegative and integral} \end{aligned} \tag{4}$$

System (4) is an integer program; its solution is determined in Problem 7.4.

1.6 A manufacturer is beginning the last week of production of four different models of wooden television consoles, labeled I, II, III, and IV, each of which must be assembled and then decorated. The models require 4, 5, 3, and 5 h, respectively, for assembling and 2, 1.5, 3, and 3 h, respectively, for decorating. The profits on the models are \$7, \$7, \$6, and \$9, respectively. The manufacturer has 30 000 h available for assembling these products (750 assemblers working 40 h/wk) and 20 000 h available for decorating (500 decorators working 40 h/wk). How many of each model should the manufacturer produce during this last week to maximize profit? Assume that all units made can be sold.

The objective is to maximize profit (in dollars), which we denote as z. Setting

x_1 = number of model I consoles to be produced in the week
x_2 = number of model II consoles to be produced in the week
x_3 = number of model III consoles to be produced in the week
x_4 = number of model IV consoles to be produced in the week

we can formulate the objective as

$$\text{maximize:} \quad z = 7x_1 + 7x_2 + 6x_3 + 9x_4 \tag{1}$$

There are constraints on the total time available for assembling and the total time available for decorating. These are, respectively, modeled by

$$4x_1 + 5x_2 + 3x_3 + 5x_4 \leq 30\,000 \tag{2}$$

$$2x_1 + 1.5x_2 + 3x_3 + 3x_4 \leq 20\,000 \tag{3}$$

As negative quantities may not be produced, four hidden constraints are $x_i \geq 0$ $(i = 1, 2, 3, 4)$. Additionally, since this is the last week of production, partially completed models at the week's end would remain unfinished and so would generate no profit. To avoid such possibilities, we require an integral value for each variable. Combining the hidden conditions with (1), (2), and (3), we obtain the mathematical program

$$\begin{aligned} \text{maximize:} \quad & z = 7x_1 + 7x_2 + 6x_3 + 9x_4 \\ \text{subject to:} \quad & 4x_1 + 5x_2 + 3x_3 + 5x_4 \leq 30\,000 \\ & 2x_1 + 1.5x_2 + 3x_3 + 3x_4 \leq 20\,000 \\ \text{with:} \quad & \text{all variables nonnegative and integral} \end{aligned} \tag{4}$$

System (4) is an integer program; its solution is determined in Problem 6.4.

1.7 The Aztec Refining Company produces two types of unleaded gasoline, regular and premium, which it sells to its chain of service stations for \$12 and \$14 per barrel, respectively. Both types are blended from Aztec's inventory of refined domestic oil and refined foreign oil, and must meet the following specifications:

Linear Programming: Standard Form

A method for solving linear programs involving many variables is described in Chapter 4. To initialize the method, one must transform all inequality constraints into equalities and must know one feasible, nonnegative solution.

NONNEGATIVITY CONDITIONS

Any variable not already constrained to be nonnegative is replaced by the difference of two new variables which are so constrained. (See Problem 2.6.)

Linear constraints (Chapter 1) are of the form:

$$\sum_{j=1}^{n} a_{ij}x_j \sim b_i \tag{2.1}$$

where $\sim$ stands for one of the relations $\leq$, $\geq$, $=$ (not necessarily the same one for each i). The constants b_i may always be assumed nonnegative.

Example 2.1 The constraint $2x_1 - 3x_2 + 4x_3 \leq -5$ is multiplied by -1 to obtain $-2x_1 + 3x_2 - 4x_3 \geq 5$, which has a nonnegative right-hand side.

SLACK VARIABLES AND SURPLUS VARIABLES

A linear constraint of the form $\sum a_{ij}x_j \leq b_i$ can be converted into an equality by adding a new, nonnegative variable to the left-hand side of the inequality. Such a variable is numerically equal to the difference between the right- and left-hand sides of the inequality and is known as a *slack variable*. It represents the waste involved in that phase of the system modeled by the constraint.

Example 2.2 The first constraint in Problem 1.6 is

$$4x_1 + 5x_2 + 3x_3 + 5x_4 \leq 30\,000$$

The left-hand side of this inequality models the total number of hours used to assemble all television consoles, while the right-hand side is the total number of hours available. This inequality is transformed into the equation

$$4x_1 + 5x_2 + 3x_3 + 5x_4 + x_5 = 30\,000$$

by adding the slack variable x_5 to the left-hand side of the inequality. Here x_5 represents the number of assembly hours available to the manufacturer but not used.

A linear constraint of the form $\sum a_{ij}x_j \geq b_i$ can be converted into an equality by subtracting a new, nonnegative variable from the left-hand side of the inequality. Such a variable is numerically equal to the difference between the left- and right-hand sides of the inequality and is known as a *surplus variable*. It represents excess input into that phase of the system modeled by the constraint.

Example 2.3 The first constraint in Problem 1.5 is

$$4x_1 + 6x_2 + x_3 \geq 54$$

The left-hand side of this inequality represents the combined output of high-grade ore from three mines, while the right-hand side is the minimum tonnage of such ore required to meet contractual obligations. This inequality is transformed into the equation

$$4x_1 + 6x_2 + x_3 - x_4 = 54$$

by subtracting the surplus variable x_4 from the left-hand side of the inequality. Here x_4 represents the amount of high-grade ore mined over and above that needed to fulfill the contract.

GENERATING AN INITIAL FEASIBLE SOLUTION

After all linear constraints (with nonnegative right-hand sides) have been transformed into equalities by introducing slack and surplus variables where necessary, add a new variable, called an *artificial variable*, to the left-hand side of each constraint equation that does not contain a slack variable. Each constraint equation will then contain either one slack variable or one artificial variable. A nonnegative initial solution to this new set of constraints is obtained by setting each slack variable and each artificial variable equal to the right-hand side of the equation in which it appears and setting all other variables, including the surplus variables, equal to zero.

Example 2.4 The set of constraints

$$\begin{aligned} x_1 + 2x_2 &\leq 3 \\ 4x_1 + 5x_2 &\geq 6 \\ 7x_1 + 8x_2 &= 15 \end{aligned}$$

is transformed into a system of equations by adding a slack variable, x_3, to the left-hand side of the first constraint and subtracting a surplus variable, x_4, from the left-hand side of the second constraint. The new system is

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 3 \\ 4x_1 + 5x_2 - x_4 &= 6 \\ 7x_1 + 8x_2 &= 15 \end{aligned} \tag{2.2}$$

If now artificial variables x_5 and x_6 are respectively added to the left-hand sides of the last two constraints in system (2.2), the constraints without a slack variable, the result is

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 3 \\ 4x_1 + 5x_2 - x_4 + x_5 &= 6 \\ 7x_1 + 8x_2 + x_6 &= 15 \end{aligned}$$

A *nonnegative* solution to this last system is $x_3 = 3$, $x_5 = 6$, $x_6 = 15$, and $x_1 = x_2 = x_4 = 0$. (Notice, however, that $x_1 = 0$, $x_2 = 0$ is *not* a solution to the original set of constraints.)

Occasionally, an initial solution can be generated easily without a full complement of slack and artificial variables. An example is Problem 2.5.

PENALTY COSTS

The introduction of slack and surplus variables alters neither the nature of the constraints nor the objective. Accordingly, such variables are incorporated into the objective function with zero coefficients. Artificial variables, however, do change the nature of the constraints. Since they are added to only one side of an equality, the new system is equivalent to the old system of constraints if and only if the artificial variables are zero. To guarantee such assignments in the optimal solution (in contrast to the initial solution), artificial variables are incorporated into the objective function

with very large positive coefficients in a minimization program or very large negative coefficients in a maximization program. These coefficients, denoted by either M or $-M$, where M is understood to be a large positive number, represent the (severe) penalty incurred in making a unit assignment to the artificial variables.

In hand calculations, penalty costs can be left as $\pm M$. In computer calculations, M must be assigned a numerical value, usually a number three or four times larger in magnitude than any other number in the program.

STANDARD FORM

A linear program is in *standard form* if the constraints are all modeled as equalities and if one feasible solution is known. In matrix notation, standard form is

$$\begin{aligned} \text{optimize:} \quad & z = \mathbf{C}^T\mathbf{X} \\ \text{subject to:} \quad & \mathbf{AX} = \mathbf{B} \\ \text{with:} \quad & \mathbf{X} \geq \mathbf{0} \end{aligned} \tag{2.3}$$

where $\mathbf{X}$ is the column vector of unknowns, including all slack, surplus, and artificial variables; $\mathbf{C}^T$ is the row vector of the corresponding costs; $\mathbf{A}$ is the coefficient matrix of the constraint equations; and $\mathbf{B}$ is the column vector of the right-hand sides of the constraint equations. [*Note*: In the remainder of this book, vectors shall normally be represented as one-columned matrices, and we shall simply say "vector" instead of "column vector." Superscript T designates transposition.] If $\mathbf{X}_0$ denotes the vector of slack and artificial variables only, then the initial feasible solution is given by $\mathbf{X}_0 = \mathbf{B}$, where it is understood that all variables in $\mathbf{X}$ not included in $\mathbf{X}_0$ are assigned zero values.

Solved Problems

2.1 Put the following program in standard matrix form:

$$\begin{aligned} \text{maximize:} \quad & z = x_1 + x_2 \\ \text{subject to:} \quad & x_1 + 5x_2 \leq 5 \\ & 2x_1 + x_2 \leq 4 \\ \text{with:} \quad & x_1 \text{ and } x_2 \text{ nonnegative} \end{aligned}$$

Adding slack variables x_3 and x_4, respectively, to the left-hand sides of the constraints, and including these new variables with zero cost coefficients in the objective, we have

$$\begin{aligned} \text{maximize:} \quad & z = x_1 + x_2 + 0x_3 + 0x_4 \\ \text{subject to:} \quad & x_1 + 5x_2 + x_3 = 5 \\ & 2x_1 + x_2 + x_4 = 4 \\ \text{with:} \quad & \text{all variables nonnegative} \end{aligned} \tag{1}$$

Since each constraint equation contains a slack variable, no artificial variables are required; an initial feasible solution is $x_3 = 5$, $x_4 = 4$, $x_1 = x_2 = 0$. System (*1*) is in the standard form (*2.3*) if we define

$$\mathbf{X} = [x_1, x_2, x_3, x_4]^T \qquad \mathbf{C} = [1, 1, 0, 0]^T$$

$$\mathbf{A} = \begin{bmatrix} 1 & 5 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 5 \\ 4 \end{bmatrix} \qquad \mathbf{X}_0 = \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$$

2.2 Put the following program in standard form:

$$\begin{aligned} \text{maximize:} \quad & z = 80x_1 + 60x_2 \\ \text{subject to:} \quad & 0.20x_1 + 0.32x_2 \leq 0.25 \\ & x_1 + \quad x_2 = \ 1 \\ \text{with:} \quad & x_1 \text{ and } x_2 \text{ nonnegative} \end{aligned}$$

To convert the first constraint into an equality, add a slack variable x_3 to the left-hand side. Since the second constraint, an equation, does not contain a slack variable, add an artificial variable x_4 to its left-hand side. Both new variables are included in the objective function, the slack variable with a zero cost coefficient and the artificial variable with a very large negative cost coefficient, yielding the program

$$\begin{aligned} \text{maximize:} \quad & z = 80x_1 + 60x_2 + 0x_3 - Mx_4 \\ \text{subject to:} \quad & 0.20x_1 + 0.32x_2 + x_3 \quad = 0.25 \\ & x_1 + \quad x_2 \quad + x_4 = \ 1 \\ \text{with:} \quad & \text{all variables nonnegative} \end{aligned}$$

This program is in standard form, with an initial feasible solution $x_3 = 0.25$, $x_4 = 1$, $x_1 = x_2 = 0$.

2.3 Redo Problem 2.2 if the objective is to be minimized.

The only change is in the cost coefficient associated with the artificial variable; it becomes $+M$ instead of $-M$.

2.4 Put the following program in standard form:

$$\begin{aligned} \text{maximize:} \quad & z = 5x_1 + 2x_2 \\ \text{subject to:} \quad & 6x_1 + \ x_2 \geq \ 6 \\ & 4x_1 + 3x_2 \geq 12 \\ & x_1 + 2x_2 \geq \ 4 \\ \text{with:} \quad & x_1 \text{ and } x_2 \text{ nonnegative} \end{aligned}$$

Subtracting surplus variables x_3, x_4, and x_5, respectively, from the left-hand sides of the constraints, and including each new variable with a zero cost coefficient in the objective, we obtain

$$\begin{aligned} \text{maximize:} \quad & z = 5x_1 + 2x_2 + 0x_3 + 0x_4 + 0x_5 \\ \text{subject to:} \quad & 6x_1 + \ x_2 - x_3 \quad\quad = \ 6 \\ & 4x_1 + 3x_2 \quad - x_4 \quad = 12 \\ & x_1 + 2x_2 \quad\quad - x_5 = \ 4 \\ \text{with:} \quad & \text{all variables nonnegative} \end{aligned}$$

Since no constraint equation contains a slack variable, we next add artificial variables x_6, x_7, and x_8, respectively, to the left-hand sides of the equations. We also include these variables with very large negative cost coefficients in the objective. The program becomes

$$\begin{aligned} \text{maximize:} \quad & z = 5x_1 + 2x_2 + 0x_3 + 0x_4 + 0x_5 - Mx_6 - Mx_7 - Mx_8 \\ \text{subject to:} \quad & 6x_1 + \ x_2 - x_3 \quad\quad + x_6 \quad\quad = \ 6 \\ & 4x_1 + 3x_2 \quad - x_4 \quad\quad + x_7 \quad = 12 \\ & x_1 + 2x_2 \quad\quad - x_5 \quad\quad + x_8 = \ 4 \\ \text{with:} \quad & \text{all variables nonnegative} \end{aligned}$$

This program is in standard form, with an initial feasible solution $x_6 = 6$, $x_7 = 12$, $x_8 = 4$, $x_1 = x_2 = x_3 = x_4 = x_5 = 0$.

2.5 Put the following program in standard matrix form:

$$\begin{aligned} \text{minimize:} \quad & z = x_1 + 2x_2 + 3x_3 \\ \text{subject to:} \quad & 3x_1 \quad + 4x_3 \le 5 \\ & 5x_1 + x_2 + 6x_3 = 7 \\ & 8x_1 \quad + 9x_3 \ge 2 \\ \text{with:} \quad & \text{all variables nonnegative} \end{aligned}$$

Adding a slack variable x_4 to the left-hand side of the first constraint, subtracting a surplus variable x_5 from the left-hand side of the third constraint, and then adding an artificial variable x_6 only to the left-hand side of the third constraint, we obtain the program

$$\begin{aligned} \text{minimize:} \quad & z = x_1 + 2x_2 + 3x_3 + 0x_4 + 0x_5 + Mx_6 \\ \text{subject to:} \quad & 3x_1 \quad + 4x_3 + x_4 \qquad\quad = 5 \\ & 5x_1 + x_2 + 6x_3 \qquad\quad = 7 \\ & 8x_1 \quad + 9x_3 \quad - x_5 + x_6 = 2 \\ \text{with:} \quad & \text{all variables nonnegative} \end{aligned}$$

This program is in standard form, with an initial feasible solution $x_4 = 5$, $x_2 = 7$, $x_6 = 2$, $x_1 = x_3 = x_5 = 0$. It has the form of system (*2.3*) if we define

$$\mathbf{X} = [x_1, x_2, x_3, x_4, x_5, x_6]^T \qquad \mathbf{C} = [1, 2, 3, 0, 0, M]^T$$

$$\mathbf{A} = \begin{bmatrix} 3 & 0 & 4 & 1 & 0 & 0 \\ 5 & 1 & 6 & 0 & 0 & 0 \\ 8 & 0 & 9 & 0 & -1 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 5 \\ 7 \\ 2 \end{bmatrix} \qquad \mathbf{X}_0 = \begin{bmatrix} x_4 \\ x_2 \\ x_6 \end{bmatrix}$$

In this case, x_2 can be used to generate the initial solution rather than adding an artificial variable to the second constraint to achieve the same result. In general, whenever a variable appears in one and only one constraint equation, and there with a positive coefficient, that variable can be used to generate part of the initial solution by first dividing the constraint equation by the positive coefficient and then setting the variable equal to the right-hand side of the equation. an artificial variable need not be added to the equation.

2.6 Put the following program in standard form:

$$\begin{aligned} \text{minimize:} \quad & z = 25x_1 + 30x_2 \\ \text{subject to:} \quad & 4x_1 + 7x_2 \ge 1 \\ & 8x_1 + 5x_2 \ge 3 \\ & 6x_1 + 9x_2 \ge -2 \end{aligned}$$

Since both x_1 and x_2 are unrestricted, we set $x_1 = x_3 - x_4$ and $x_2 = x_5 - x_6$, where all four new variables are required to be nonnegative. Substituting these quantities into the given program and then multiplying the last constraint by -1 to force a nonnegative right-hand side, we obtain the equivalent program:

$$\begin{aligned} \text{minimize:} \quad & z = 25x_3 - 25x_4 + 30x_5 - 30x_6 \\ \text{subject to:} \quad & 4x_3 - 4x_4 + 7x_5 - 7x_6 \ge 1 \\ & 8x_3 - 8x_4 + 5x_5 - 5x_6 \ge 3 \\ & -6x_3 + 6x_4 - 9x_5 + 9x_6 \le 2 \\ \text{with:} \quad & \text{all variables nonnegative} \end{aligned}$$

This program is converted into standard form by subtracting surplus variables x_7 and x_8, respectively, from the left-hand sides of the first two constraints; adding a slack variable x_9 to the left-hand side of the third constraint; and then adding artificial variables x_{10} and x_{11}, respectively, to the left-hand sides of the first two constraints. Doing so, we obtain

$$\begin{aligned}
\text{minimize:}\quad & z = 25x_3 - 25x_4 + 30x_5 - 30x_6 + 0x_7 + 0x_8 + 0x_9 + Mx_{10} + Mx_{11} \\
\text{subject to:}\quad & 4x_3 - 4x_4 + 7x_5 - 7x_6 - x_7 \qquad + x_{10} \qquad = 1 \\
& 8x_3 - 8x_4 + 5x_5 - 5x_6 \qquad - x_8 \qquad + x_{11} = 3 \\
& -6x_3 + 6x_4 - 9x_5 + 9x_6 \qquad + x_9 \qquad = 2 \\
\text{with:}\quad & \text{all variables nonnegative}
\end{aligned}$$

An initial solution to this program in standard form is

$$x_{10} = 1 \qquad x_{11} = 3 \qquad x_9 = 2 \qquad x_3 = x_4 = x_5 = x_6 = x_7 = x_8 = 0$$

Supplementary Problems

Put each of the following programs in matrix standard form.

2.7

$$\begin{aligned}
\text{minimize:}\quad & z = 2x_1 - x_2 + 4x_3 \\
\text{subject to:}\quad & 5x_1 + 2x_2 - 3x_3 \geq -7 \\
& 2x_1 - 2x_2 + x_3 \leq 8 \\
\text{with:}\quad & x_1 \text{ nonnegative}
\end{aligned}$$

2.8

$$\begin{aligned}
\text{maximize:}\quad & z = 10x_1 + 11x_2 \\
\text{subject to:}\quad & x_1 + 2x_2 \leq 150 \\
& 3x_1 + 4x_2 \leq 200 \\
& 6x_1 + x_2 \leq 175 \\
\text{with:}\quad & x_1 \text{ and } x_2 \text{ nonnegative}
\end{aligned}$$

2.9 Problem 2.8 with the three constraint inequalities reversed.

2.10

$$\begin{aligned}
\text{minimize:}\quad & z = 3x_1 + 2x_2 + 4x_3 + 6x_4 \\
\text{subject to:}\quad & x_1 + 2x_2 + x_3 + x_4 \geq 1000 \\
& 2x_1 + x_2 + 3x_3 + 7x_4 \geq 1500 \\
\text{with:}\quad & \text{all variables nonnegative}
\end{aligned}$$

2.11

$$\begin{aligned}
\text{minimize:}\quad & z = 6x_1 + 3x_2 + 4x_3 \\
\text{subject to:}\quad & x_1 + 6x_2 + x_3 = 10 \\
& 2x_1 + 3x_2 + x_3 = 15 \\
\text{with:}\quad & \text{all variables nonnegative}
\end{aligned}$$

2.12

$$\begin{aligned}
\text{maximize:}\quad & z = 7x_1 + 2x_2 + 3x_3 + x_4 \\
\text{subject to:}\quad & 2x_1 + 7x_2 \qquad = 7 \\
& 5x_1 + 8x_2 \qquad + 2x_4 = 10 \\
& x_1 \qquad + x_3 \qquad = 11 \\
\text{with:}\quad & x_1,\ x_2, \text{ and } x_3 \text{ nonnegative}
\end{aligned}$$

2.13

$$
\begin{aligned}
\text{minimize:} \quad & z = 10x_1 + 2x_2 - x_3 \\
\text{subject to:} \quad & x_1 + x_2 \leq 50 \\
& x_1 + x_2 \geq 10 \\
& x_2 + x_3 \leq 30 \\
& x_2 + x_3 \geq 7 \\
& x_1 + x_2 + x_3 = 60 \\
\text{with:} \quad & \text{all variables nonnegative}
\end{aligned}
$$

Linear Programming: The Simplex Method

THE SIMPLEX TABLEAU

The *simplex method* is a matrix procedure for solving linear programs in the standard form

$$\text{optimize:} \quad z = \mathbf{C}^T\mathbf{X}$$

$$\text{subject to:} \quad \mathbf{AX} = \mathbf{B}$$

$$\text{with:} \quad \mathbf{X} \geq \mathbf{0}$$

where $\mathbf{B} \geq \mathbf{0}$ and a basic feasible solution $\mathbf{X}_0$ is known (Problem 3.15). Starting with $\mathbf{X}_0$, the method locates successively other basic feasible solutions having better values of the objective, until the optimal solution is obtained. For minimization programs, the simplex method utilizes Tableau 4-1, in which $\mathbf{C}_0$ designates the cost vector associated with the variables in $\mathbf{X}_0$.

		$\mathbf{X}^T$ $\mathbf{C}^T$	
$\mathbf{X}_0$	$\mathbf{C}_0$	$\mathbf{A}$	$\mathbf{B}$
		$\mathbf{C}^T - \mathbf{C}_0^T\mathbf{A}$	$-\mathbf{C}_0^T\mathbf{B}$

Tableau 4-1

For maximization programs, Tableau 4-1 applies if the elements of the bottom row have their *signs reversed.*

Example 4.1 For the minimization program of Problem 2.5, $\mathbf{C}_0 = [0, 2, M]^T$. Then,

$$\mathbf{C}^T - \mathbf{C}_0^T\mathbf{A} = [1, 2, 3, 0, 0, M] - [0, 2, M]\begin{bmatrix} 3 & 0 & 4 & 1 & 0 & 0 \\ 5 & 1 & 6 & 0 & 0 & 0 \\ 8 & 0 & 9 & 0 & -1 & 1 \end{bmatrix}$$

$$= [1, 2, 3, 0, 0, M] - [10 + 8M, 2, 12 + 9M, 0, -M, M] = [-9 - 8M, 0, -9 - 9M, 0, M, 0]$$

$$-\mathbf{C}_0^T\mathbf{B} = -[0, 2, M]\begin{bmatrix} 5 \\ 7 \\ 2 \end{bmatrix} = -14 - 2M$$

and Tableau 4-1 becomes

		x_1 1	x_2 2	x_3 3	x_4 0	x_5 0	x_6 M	
x_4	0	3	0	4	1	0	0	5
x_2	2	5	1	6	0	0	0	7
x_6	M	8	0	9	0	−1	1	2
		$-9-8M$	0	$-9-9M$	0	M	0	$-14-2M$

A TABLEAU SIMPLIFICATION

For each j $(j = 1, 2, \ldots, n)$, define $z_j \equiv \mathbf{C}_0^T \mathbf{A}_j$, the dot product of $\mathbf{C}_0$ with the jth column of $\mathbf{A}$. The jth entry in the last row of Tableau 4-1 is $c_j - z_j$ (or, for a maximization program, $z_j - c_j$), where c_j is the cost in the second row of the tableau, immediately above $\mathbf{A}_j$. Once this last row has been obtained, the second row and second column of the tableau, corresponding to $\mathbf{C}^T$ and $\mathbf{C}_0$, respectively, become superfluous and may be eliminated.

THE SIMPLEX METHOD

STEP 1 Locate the most negative number in the bottom row of the simplex tableau, excluding the last column, and call the column in which this number appears the *work column*. If more than one candidate for most negative number exists, choose one.

STEP 2 Form ratios by dividing each *positive* number in the work column, excluding the last row, into the element in the same row and last column. Designate the element in the work column that yields the *smallest* ratio as the *pivot element*. If more than one element yields the same smallest ratio, choose one. If no element in the work column is positive, the program has no solution.

STEP 3 Use elementary row operations to convert the pivot element to 1 and then to reduce all *other* elements in the work column to 0.

STEP 4 Replace the x-variable in the pivot row and first column by the x-variable in the first row and pivot column. This new first column is the current set of basic variables (see Chapter 3).

STEP 5 Repeat Steps 1 through 4 until there are no negative numbers in the last row, excluding the last column.

STEP 6 The optimal solution is obtained by assigning to each variable in the first column that value in the corresponding row and last column. All other variables are assigned the value zero. The associated z^*, the optimal value of the objective function, is the number in the last row and last column for a maximization program, but the *negative* of this number for a minimization program.

MODIFICATIONS FOR PROGRAMS WITH ARTIFICIAL VARIABLES

Whenever artificial variables are part of the initial solution $\mathbf{X}_0$, the last row of Tableau 4-1 will contain the penalty cost M (see Chapter 2). To minimize roundoff error (see Problem 4.6), the following modifications are incorporated into the simplex method; the resulting algorithm is the *two-phase method*.

Change 1: The last row of Tableau 4-1 is decomposed into two rows, the first of which involves those terms not containing M, while the second involves the coefficients of M in the remaining terms.

Example 4.2 The last row of the tableau in Example 4.1 is

$$-9-8M \quad 0 \quad -9-9M \quad 0 \quad M \quad 0 \quad -14-2M$$

Under Change 1 it would be transformed into the two rows

$$\begin{array}{cccccccc} -9 & 0 & -9 & 0 & 0 & 0 & -14 \\ -8 & 0 & -9 & 0 & 1 & 0 & -2 \end{array}$$

Change 2: Step 1 of the simplex method is applied to the last row created in Change 1 (followed by Steps 2, 3, and 4), until this row contains no negative elements. Then Step 1 is applied to those elements in the next-to-last row that are positioned over zeros in the last row.

Change 3: Whenever an artificial variable ceases to be basic—i.e. is removed from the first column of the tableau as a result of Step 4—it is deleted from the top row of the tableau, as is the entire column under it. (This modification simplifies hand calculations but is not implemented in many computer programs.)

Change 4: The last row can be deleted from the tableau whenever it contains all zeros.

Change 5: If *nonzero* artificial variables are present in the final basic set, then the program has no solution. (In contrast, zero-valued artificial variables may appear as basic variables in the final solution when one or more of the original constraint equations is redundant.)

Solved Problems

4.1

$$\begin{aligned} \text{maximize:} \quad & z = x_1 + 9x_2 + x_3 \\ \text{subject to:} \quad & x_1 + 2x_2 + 3x_3 \leq 9 \\ & 3x_1 + 2x_2 + 2x_3 \leq 15 \\ \text{with:} \quad & \text{all variables nonnegative} \end{aligned}$$

This program is put into matrix standard form by first introducing slack variables x_4 and x_5 in the first and second constraint inequalities, respectively, and then defining

$$\mathbf{X} = [x_1, x_2, x_3, x_4, x_5]^T \qquad \mathbf{C} = [1, 9, 1, 0, 0]^T$$

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 1 & 0 \\ 3 & 2 & 2 & 0 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 9 \\ 15 \end{bmatrix} \qquad \mathbf{X}_0 = \begin{bmatrix} x_4 \\ x_5 \end{bmatrix}$$

The costs associated with the components of $\mathbf{X}_0$, the slack variables, are zero; hence $\mathbf{C}_0 = [0, 0]^T$. Tableau 4-1 becomes

		x_1	x_2	x_3	x_4	x_5	
		1	9	1	0	0	
x_4	0	1	2	3	1	0	9
x_5	0	3	2	2	0	1	15

To compute the last row of this tableau, we use the tableau simplification and first calculate each z_j by inspection: it is the dot product of column 2 and the jth column of A. We then subtract the corresponding cost c_j from it (maximization program). In this case, the second column is zero, and so $z_j - c_j = 0 - c_j = -c_j$. Hence, the bottom row of the tableau, excluding the last element, is just the negative of row 2. The last element in the bottom row is simply the dot product of column 2 and the final, B-column, and so it too is zero. At this point, the second row and second column of the tableau are superfluous. Eliminating them, we obtain Tableau 1 as the complete initial tableau.

	x_1	x_2	x_3	x_4	x_5	
x_4	1	2*	3	1	0	9
x_5	3	2	2	0	1	15
$(z_j - c_j)$	−1	−9	−1	0	0	0

Tableau 1

	x_1	x_2	x_3	x_4	x_5	
x_2	1/2	1	3/2	1/2	0	9/2
x_5	2	0	−1	−1	1	6
	7/2	0	25/2	9/2	0	81/2

Tableau 2

We are now ready to apply the simplex method. The most negative element in the last row of Tableau 1 is -9, corresponding to the x_2-column; hence this column becomes the work column. Forming the ratios $9/2 = 4.5$ and $15/2 = 7.5$, we find that the element 2, marked by the asterisk in Tableau 1, is the pivot element, since it yields the smallest ratio. Then, applying Steps 3 and 4 to Tableau 1, we obtain Tableau 2. Since the last row of Tableau 2 contains no negative elements, it follows from Step 6 that the optimal solution is $x_2^* = 9/2$, $x_3^* = 6$, $x_1^* = x_3^* = x_4^* = 0$, with $z^* = 81/2$.

4.2

$$\text{minimize:} \quad z = 80x_1 + 60x_2$$

$$\text{subject to:} \quad 0.20x_1 + 0.32x_2 \leq 0.25$$

$$x_1 + x_2 = 1$$

$$\text{with:} \quad x_1 \text{ and } x_2 \text{ nonnegative}$$

Adding a slack variable x_3 and an artificial variable x_4 to the first and second constraints, respectively, we convert the program to standard matrix form, with

$$\mathbf{X} = [x_1, x_2, x_3, x_4]^T \qquad \mathbf{C} = [80, 60, 0, M]^T$$

$$\mathbf{A} = \begin{bmatrix} 0.20 & 0.32 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0.25 \\ 1 \end{bmatrix} \qquad \mathbf{X}_0 = \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$$

Substituting these matrices, along with $\mathbf{C}_0 = [0, M]^T$, into Tableau 4-1, we obtain Tableau 0. Since the bottom row involves M, we apply Change 1; the resulting Tableau 1 is the initial tableau for the two-phase method.

		x_1	x_2	x_3	x_4	
		80	60	0	M	
x_3	0	0.20	0.32	1	0	0.25
x_4	M	1	1	0	1	1
		$80 - M$	$60 - M$	0	0	$-M$

Tableau 0

	x_1	x_2	x_3	x_4	
x_3	0.20	0.32	1	0	0.25
x_4	1*	1	0	1	1
$(c_j - z_j)$:	80	60	0	0	0
	-1	-1	0	0	-1

Tableau 1

	x_1	x_2	x_3	
x_3	0	0.12*	1	0.05
x_1	1	1	0	1
	0	-20	0	-80
	0	0	0	0

Tableau 2

Using both Step 1 of the simplex method and Change 2, we find that the most negative element in the last row of Tableau 1 (excluding the last column) is -1, which appears twice. Arbitrarily selecting the x_1-column as the work column, we form the ratios $0.25/0.20 = 1.25$ and $1/1 = 1$. Since the element 1, starred in Tableau 1, yields the smallest ratio, it becomes the pivot. Then, applying Steps 3 and 4 and Change 3 to Tableau 1, we generate Tableau 2. Observe that x_1 replaces the artificial variable x_4 in the first column of Tableau 2, so that the entire x_4-column is absent from Tableau 2. Now, with no artificial variables in the first column and with Change 3 implemented, the last row of the tableau should be all zeros. It is; and by Change 4 this row may be deleted, giving

$$0 \quad -20 \quad 0 \quad -80$$

as the new last row of Tableau 2.

Repeating Steps 1 through 4, we find that the x_2-column is the new work column (recall that the last element in the last row is excluded under Step 1), the starred element in Tableau 2 is the new pivot, and the elementary row operations yield Tableau 3, in which all calculations have been rounded to four

We are now ready to apply the simplex method. The most negative element in the last row of Tableau 1 is -9, corresponding to the x_2-column; hence this column becomes the work column. Forming the ratios $9/2 = 4.5$ and $15/2 = 7.5$, we find that the element 2, marked by the asterisk in Tableau 1, is the pivot element, since it yields the smallest ratio. Then, applying Steps 3 and 4 to Tableau 1, we obtain Tableau 2. Since the last row of Tableau 2 contains no negative elements, it follows from Step 6 that the optimal solution is $x_2^* = 9/2$, $x_3^* = 6$, $x_1^* = x_4^* = 0$, with $z^* = 81/2$.

4.2

$$\text{minimize:} \quad z = 80x_1 + 60x_2$$

$$\text{subject to:} \quad 0.20x_1 + 0.32x_2 \leq 0.25$$

$$x_1 + x_2 = 1$$

$$\text{with:} \quad x_1 \text{ and } x_2 \text{ nonnegative}$$

Adding a slack variable x_3 and an artificial variable x_4 to the first and second constraints, respectively, we convert the program to standard matrix form, with

$$\mathbf{X} = [x_1, x_2, x_3, x_4]^T \qquad \mathbf{C} = [80, 60, 0, M]^T$$

$$\mathbf{A} = \begin{bmatrix} 0.20 & 0.32 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0.25 \\ 1 \end{bmatrix} \qquad \mathbf{X}_0 = \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$$

Substituting these matrices, along with $\mathbf{C}_0 = [0, M]^T$, into Tableau 4-1, we obtain Tableau 0. Since the bottom row involves M, we apply Change 1; the resulting Tableau 1 is the initial tableau for the two-phase method.

		x_1	x_2	x_3	x_4	
		80	60	0	M	
x_3	0	0.20	0.32	1	0	0.25
x_4	M	1	1	0	1	1
		$80 - M$	$60 - M$	0	0	$-M$

Tableau 0

	x_1	x_2	x_3	x_4	
x_3	0.20	0.32	1	0	0.25
x_4	1*	1	0	1	1
$(c_j - z_j)$:	80	60	0	0	0
	-1	-1	0	0	-1

Tableau 1

	x_1	x_2	x_3	
x_3	0	0.12*	1	0.05
x_1	1	1	0	1
	0	-20	0	-80
	0	0	0	0

Tableau 2

Using both Step 1 of the simplex method and Change 2, we find that the most negative element in the last row of Tableau 1 (excluding the last column) is -1, which appears twice. Arbitrarily selecting the x_1-column as the work column, we form the ratios $0.25/0.20 = 1.25$ and $1/1 = 1$. Since the element 1, starred in Tableau 1, yields the smallest ratio, it becomes the pivot. Then, applying Steps 3 and 4 and Change 3 to Tableau 1, we generate Tableau 2. Observe that x_1 replaces the artificial variable x_4 in the first column of Tableau 2, so that the entire x_4-column is absent from Tableau 2. Now, with no artificial variables in the first column and with Change 3 implemented, the last row of the tableau should be all zeros. It is; and by Change 4 this row may be deleted, giving

$$0 \quad -20 \quad 0 \quad -80$$

as the new last row of Tableau 2.

Repeating Steps 1 through 4, we find that the x_2-column is the new work column (recall that the last element in the last row is excluded under Step 1), the starred element in Tableau 2 is the new pivot, and the elementary row operations yield Tableau 3, in which all calculations have been rounded to four

	x_1	x_2	x_3	x_4	x_5	
x_1	1	0	−0.2144	0.07144*	0	0.4284
x_5	0	1	0.3571	−0.7855	1	3.284
x_2	0	1	0.2858	−0.4286	0	3.429
	0	0	−0.5000	−0.5001	0	9.001
	0	0	−0.0002	0.0001	0	0.0005

Tableau 4

	x_1	x_2	x_3	x_4	x_5	
x_4	14.00	0	−3.001	1	0	6.000
x_5	11.00	0	−2.000	0	1	7.997
x_2	6.000	1	−1.000	0	0	6.001
	7.001	0	−2.001	0	0	12.00

Tableau 5

Tableau 4 is the first tableau containing no artificial variables in its first column, hence, with Change 3 implemented, the last row of the tableau should be zero. To within roundoff errors it is zero, so we delete it from the Tableau. Tableau 5, however, presents a problem that cannot be ignored: the work column is the x_3-column and all the elements in that column are negative! It follows from Step 2 that the original program has no solution. (It is easy to show graphically that the feasible region is infinite and that the objective function can be made arbitrarily large by choosing feasible points with arbitrarily large coordinates.)

4.4

$$\begin{aligned} \text{maximize:} \quad & z = 2x_1 + 3x_2 \\ \text{subject to:} \quad & x_1 + 2x_2 \leq 2 \\ & 6x_1 + 4x_2 \geq 24 \\ \text{with:} \quad & \text{all variables nonnegative} \end{aligned}$$

This program is put in standard form by introducing a slack variable x_3 to the first constraint, and both a surplus variable x_4 and an artificial variable x_5 to the second constraint. Then Tableau 4-1, with Change 1, becomes Tableau 1.

		x_1	x_2	x_3	x_4	x_5	
		2	3	0	0	$-M$	
x_3	0	1*	2	1	0	0	2
x_5	$-M$	6	4	0	−1	1	24
$(z_j - c_j)$:		−2	−3	0	0	0	0
		−6	−4	0	1	0	−24

Tableau 1

	x_1	x_2	x_3	x_4	x_5	
x_1	1	2	1	0	0	2
x_5	0	−8	−6	−1	1	12
	0	1	2	0	0	4
	0	8	6	1	0	−12

Tableau 2

Applying the two-phase algorithm to Tableau 1 (the pivot element is starred), we generate Tableau 2. Now, there are no negative entries in the last row of Tableau 2, and in the next-to-last row there is no negative entry positioned above a zero of the last row. Thus, the two-phase method signals that optimality has been achieved. But the nonzero artificial variable x_5 is still basic! By Change 5, the original program has no solution. (In this case $\mathscr{S}$ is empty, as the constraint inequalities and the nonnegativity conditions cannot be satisfied simultaneously.)