Introduction to Matlab

1. The nature of Matlab

Matlab is a computer language developed for the specific purpose of matrix manipulation. In fact, Matlab is short for 'Matrix Laboratory' and not as one might at first imagine 'Maths laboratory'. Matlab differs from many other languages developed for numerical computation. The latter are typified by older languages such as Fortran, Algol, Pascal, Java, C, C++, C# and many others. While these languages can be regarded as 'general purpose', so is Matlab as it can be used to express every operation that can be carried out by these predecessor languages. However, the particular value of Matlab is (a) that it compactly expresses matrix operations, and (b) that it has many specific matrix operations that are not present in the predecessor languages—though as the latter are general purpose they can be used to code the same procedures. To be sure, Matlab is not alone in being able to compactly express and implement matrix operations: Python is another such language. However, we here concentrate on Matlab, as it has achieved wide usage for many years and its popularity within the computer vision community is high (while much the same can be said of Python, it would not be easy to cover the syntax and use of both at once, and we shall defer consideration of Python until later).

Tables 1 and 2 compare the basic programming notations of C++ and Matlab. There is no remarkable difference between the two. Both have their idiosyncrasies, as indeed do all computer languages: these reflect the inclinations and dispositions of their creators and too much attention should not be focussed on them—albeit they present pitfalls for those who move from one language to another. Note that Tables 1 and 2 are intended to be illustrative rather than comprehensive. In what follows we will focus mainly on Matlab and not be constrained by what operators C++ does or doesn't possess. However, before leaving C++ behind, note that it (and most previous languages) requires the computer programmer to declare variables. This has the dual function of reserving space for each variable and also of forcing the compiler to check that the programmer hasn't made a mistake and really means each new variable to be given a separate meaning. For example, if two statements in a C++ program make use of variables such as xx and xxx or key and keys, they will only be accepted if they have individually been declared at the head of the program: if not, an error message will be flagged up. On the other hand, one of Matlab's basic aims is to make pre-declaration unnecessary, so as to relieve pressure on the programmer and permit an easier approach to program development. For example, one can straightaway write the following command in Matlab:

\[
\text{numobjects} = 6
\]

In this case Matlab will create the variable \text{numobjects} and assign the value 6 to it. Later on, a similar statement can change the value 6 to any other value, such as 9: at that stage Matlab will not recreate the variable, but just assign the new value to it.

This facility can be extended to vectors and matrices. For example, we can define the following vector:

\[
\text{vector1} = [1 2 4 7 9 5]
\]
Table 1 Comparison of basic C++ and Matlab operators. This table is intended to be illustrative rather than comprehensive.

<table>
<thead>
<tr>
<th>operation</th>
<th>C++</th>
<th>Matlab</th>
<th>different</th>
</tr>
</thead>
<tbody>
<tr>
<td>comment</td>
<td>// /* */</td>
<td>% %{ %}</td>
<td>√</td>
</tr>
<tr>
<td>unary plus</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>unary minus</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>subtract</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>multiply</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>divide by</td>
<td>/</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>divide into</td>
<td>\</td>
<td>\</td>
<td>√</td>
</tr>
<tr>
<td>power</td>
<td>^</td>
<td>^</td>
<td></td>
</tr>
<tr>
<td>modulus</td>
<td>%</td>
<td>%</td>
<td></td>
</tr>
<tr>
<td>pointwise</td>
<td>.*</td>
<td>.*</td>
<td>√</td>
</tr>
<tr>
<td>pointwise division by</td>
<td>./</td>
<td>./</td>
<td>√</td>
</tr>
<tr>
<td>pointwise division into</td>
<td>.\</td>
<td>.\</td>
<td>√</td>
</tr>
<tr>
<td>pointwise power</td>
<td>.^</td>
<td>.^</td>
<td>√</td>
</tr>
<tr>
<td>transpose</td>
<td>'</td>
<td>'</td>
<td></td>
</tr>
<tr>
<td>assignment</td>
<td>=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>(precedence) brackets</td>
<td>( )</td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>greater than or equal to</td>
<td>&gt;=</td>
<td>&gt;=</td>
<td></td>
</tr>
<tr>
<td>less than or equal to</td>
<td>&lt;=</td>
<td>&lt;=</td>
<td></td>
</tr>
<tr>
<td>(test for) equality</td>
<td>==</td>
<td>==</td>
<td></td>
</tr>
<tr>
<td>(test for) inequality</td>
<td>!=</td>
<td>~=</td>
<td>√</td>
</tr>
<tr>
<td>(logical) or</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(logical) and</td>
<td>&amp;&amp;</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>(logical) not</td>
<td>!</td>
<td>~</td>
<td>√</td>
</tr>
<tr>
<td>end of statement</td>
<td>;</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>omit to display result</td>
<td>;</td>
<td>;</td>
<td>√</td>
</tr>
<tr>
<td>if</td>
<td>if ( )</td>
<td>if ] [newline]</td>
<td>√</td>
</tr>
<tr>
<td>then</td>
<td>{ }</td>
<td>{ }</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>else if</td>
<td>else if</td>
<td>elseif</td>
<td>√</td>
</tr>
<tr>
<td>while</td>
<td>while ( )</td>
<td>while</td>
<td>√</td>
</tr>
<tr>
<td>do</td>
<td>do</td>
<td>do</td>
<td></td>
</tr>
<tr>
<td>for</td>
<td>for ( ; ; )</td>
<td>for : :</td>
<td>√</td>
</tr>
</tbody>
</table>
Table 2 Comparison of C++ and Matlab conversion and rounding operators. As in the case of Table 1, this table is intended to be illustrative rather than comprehensive. Note that the C++ conversion functions are liable to be implementation dependent and may not be exactly equivalent to the Matlab functions.

<table>
<thead>
<tr>
<th>operation</th>
<th>C++</th>
<th>Matlab</th>
<th>different</th>
</tr>
</thead>
<tbody>
<tr>
<td>convert to 8-bit signed</td>
<td>int8( )</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>convert to 16-bit</td>
<td>int16( )</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>convert to 32-bit</td>
<td>(int)</td>
<td>int32( )</td>
<td>√</td>
</tr>
<tr>
<td>convert to 64-bit</td>
<td>(long)</td>
<td>int64( )</td>
<td>√</td>
</tr>
<tr>
<td>convert to 8-bit</td>
<td></td>
<td>uint8( )</td>
<td>√</td>
</tr>
<tr>
<td>convert to 16-bit</td>
<td></td>
<td>uint16( )</td>
<td>√</td>
</tr>
<tr>
<td>convert to 32-bit</td>
<td>(unsigned int)</td>
<td>uint32( )</td>
<td>√</td>
</tr>
<tr>
<td>convert to 64-bit</td>
<td>(unsigned long)</td>
<td>uint64( )</td>
<td>√</td>
</tr>
<tr>
<td>convert to type single</td>
<td>(float)</td>
<td>single( )</td>
<td>√</td>
</tr>
<tr>
<td>convert to type double</td>
<td>(double)</td>
<td>double( )</td>
<td>√</td>
</tr>
<tr>
<td>convert to type logical</td>
<td>(bool)</td>
<td>logical( )</td>
<td>√</td>
</tr>
<tr>
<td>round to nearest integer</td>
<td>(round)</td>
<td>round( )</td>
<td>√</td>
</tr>
<tr>
<td>round upwards</td>
<td>(ceil)</td>
<td>ceil( )</td>
<td>√</td>
</tr>
<tr>
<td>round downwards</td>
<td>(floor)</td>
<td>floor( )</td>
<td>√</td>
</tr>
</tbody>
</table>

Here Matlab will create the 6-element vector called `vector1`, and assign the six element values to it. Later on, a similar statement will be able to assign a different set of six values to the same vector variable.

In fact, Matlab is more powerful than this simple example indicates. If we now make the new statement

```
vector1 = [ 1 2 4 7 9 5 4 3 2 1 ]
```

Matlab will not only extend the vector from 6 to 10 elements—thereby effectively redefining it—but will also assign all the given values to it. We shall later see that this automatic extension of a vector (or matrix) variable can be taken for granted; we will also see how vector size can be reduced. Meanwhile, note that automatic extension of size should not give rise to error messages. Indeed, the whole point of this facility is to make programming easier and more transparent. On the other hand, when vectors are large, increasing the size—typically a few elements at a time—tends to lead to inefficiency and slow-running programs. This is because increasing the size of the vector means that its storage has to be reallocated and all its elements have to be copied to a new set of storage locations; and if this is done repeatedly, it is even possible that the computer will spend more time moving the existing elements than processing the overall vector!

Because gradually increasing the size of a vector (or matrix) variable tends to be inefficient, it is far better to arrange some sort of predeclaration, so that it is given a sensible size in advance of its first use. There are several ways in which this can be achieved—for example, by imagining some sort of worst case and initialising the vector as follows:
vector1 = [ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ]

and following this by changing to the desired values, such as those given earlier. Clearly, there ought to be less clumsy ways of achieving the declaration than this. A neat way is to use the `zeros` function:

```matlab
vector2 = zeros(1, 16)
```

where the initial parameter of 1 indicates that this is a matrix of one row, i.e., a vector.

Thus far, we have introduced Matlab as a programming language that is akin to other languages such as C++, but which is targeted particularly at matrix manipulation. Its hallmark is that when dealing with matrices it can make whole matrix computations with a minimum of programming. To understand this, we need only examine the following snippets of code, in which C++ has to have its running parameters specified carefully, whereas in Matlab they can be hidden from view and run autonomously:

```cpp
for (i=1; i<=k; i++) {
    C[i] = A[i]*B[i];
}
```

```matlab
C = A .* B
```

Here the main subtlety is that Matlab makes use of a new facility—that of pointwise operations on matrix elements: in this case each element of A is multiplied by the corresponding element of B. We shall see many more ways in which vectors and matrices can be manipulated in the following section.

Meanwhile, there is one more important aspect to Matlab—that it is designed so that it can be implemented interactively by applying line by line instructions (though sets of instructions can also be used to construct entire stored programs). When written on a line by line basis, each instruction is typed on a keyboard following the Matlab prompt '>>'. Thus the earlier command using the zeros function is entered as:

```matlab
>> vector2 = zeros(1, 16);
```

In Matlab, the semicolon after the instruction has the effect of suppressing display of the final result: the effect of removing it is thus:

```matlab
>> vector2 = zeros(1, 16)
vector2 =
     0     0     0     0     0     0     0     0     0     0     0     0     0     0     0
```

On the other hand, when a new variable is not defined, Matlab simply lists the values in the final result using the dummy display variable 'ans':
In what follows we shall use the above display format extensively to define various functions and show how they work. However, it will be clearer not to display the very last prompt, but merely to show the main commands and the results they lead to. For clarity, all Command Window displays will be shown against a light grey background.

2. Vector and matrix manipulation

Vectors can be defined as matrices with a single row of elements: the generic name for variables of either type is matrices. Following the way we created vectors in Section 1, the simplest way of creating matrices is by means of statements such as:

$$ \text{mat1} = [1 \ 2 \ 3; 4 \ 5 \ 6; 7 \ 8 \ 9]; $$

where the semicolons within the square brackets indicate the ends of the rows. (Note that commas can be used instead of spaces to separate the elements in each row.) As already explained, omitting the final semicolon allows us to see the final result:

$$ \text{mat1} = [1 \ 2 \ 3; 4 \ 5 \ 6; 7 \ 8 \ 9] $$

Clearly, the original statement sets the size of the matrix as well as the specific element values.

As pointed out in Section 1, it is often valuable to predeclare matrices without necessarily assigning values to the matrix elements: it has already been seen that the \texttt{zeros} function can be useful for achieving this. An alternative is to use the \texttt{ones} function, which does much the same but sets all the matrix elements to 1, which can be more useful in some cases. In both cases, the first parameter in the function is the number of rows, and the second is the number of columns: if only one number is provided, it is understood that the numbers of columns and rows are the same, giving a square matrix:

$$ \text{mat2} = \text{zeros}(2, 4) $$

$$ \text{mat2} = $$

$$ 0 \ 0 \ 0 \ 0 $$

$$ 0 \ 0 \ 0 \ 0 $$
>> mat3 = ones(2, 4)
mat3 =
 1 1 1 1
 1 1 1 1

>> mat4 = zeros(3)
mat4 =
 0 0 0
 0 0 0
 0 0 0

Similarly, the **eye** operator expresses the sound-alike symbol 'I' commonly used to denote the identity matrix: this can be used to generate any identity matrix, e.g.:

>> mat5 = eye(3)
mat5 =
 1 0 0
 0 1 0
 0 0 1

whereas the **diag** function can be used to generate a diagonal matrix from a given vector:

>> vec1 = [2 5 3 6];  
>> mat6 = diag(vec1)
mat6 =
 2 0 0 0
 0 5 0 0
 0 0 3 0
 0 0 0 6

In fact, the **diag** function can also be used to convert a diagonal matrix into a vector—in which case it will convert mat6 back to vec1.

Another interesting operator is the **colon operator**. In Matlab, colons are often used to express ranges of numbers. For example 1: 4 means 1 2 3 4, and 3: 2: 7 means 3 5 7; in the latter case the number between the two colons is used to give the (repeated) increment between the numbers in the sequence. Using this notation, we can write:

>> vec2 = 3: 2: 7
vec2 =
 3 5 7

Note that, in such cases, square brackets are not needed to help define the vector.

Interestingly, it is very simple to form a vector by **concatenating** two existing vectors. For example:
While the methods presented above may seem simple and interesting, they may also appear non-systematic and lacking in generality. In particular, it is often necessary to convert matrices from one format to another: clearly we need a well-defined way of achieving this. The \texttt{reshape} function is a useful means for doing this. It is designed to convert a matrix of any shape into one of any other shape with the same total number of elements—albeit the number of elements must be the same in all rows and similarly in all columns. By way of example, we now define matrix \texttt{mat7} and then apply the reshape function to it:

\begin{verbatim}
>> mat7 = [1 4 5 2; 3 8 6 9; 7 1 2 6]
mat7 =
    1  4  5  2
    3  8  6  9
    7  1  2  6
>> reshape(mat7, 2, 6)
ans =
    1  7  8  5  2  9
    3  4  1  6  2  6
\end{verbatim}

Again, the first parameter in the reshape function brackets is the target number of rows and the second is the target number of columns. An obvious use of the reshape function is to convert a matrix into a vector or a vector into a matrix. For example:

\begin{verbatim}
>> reshape(mat7, 1, 12)
ans =
    1  3  7  4  8  1  5  6  2  2  9  6
>> reshape(vec1, 2, 2)
ans =
    2  3
    5  6
\end{verbatim}

The reshape function evidently provides a powerful approach for the systematic setting up of matrices. However, though the conversion process may at first seem to lead to a somewhat arbitrary ordering, it actually embodies the simple rule of going down each column in turn to create a unique sequence of elements which are then placed in turn in the new matrix format. This ordering of the elements is called \textit{linear indexing}.

Another interesting function is \texttt{repmat}. This allows a given matrix to be replicated \(m\) times vertically and \(n\) times horizontally, to give a matrix \(mn\) times larger. For example:

\begin{verbatim}
>> reshape(mat7, 1, 12)
ans =
    1  3  7  4  8  1  5  6  2  2  9  6
>> reshape(vec1, 2, 2)
ans =
    2  3
    5  6
\end{verbatim}
Next, the transpose of a matrix—which is used very widely in matrix computations—is given either by applying the transpose function, transpose(), or more conveniently by applying the prime operator ‘:

```matlab
>> vec2t = vec2'
vec2t =
    3
    5
    7
```

Similarly

```matlab
>> vec2tt = vec2t'
vec2tt =
    3
    5
    7
```

and thus returns to being a normal row vector. More generally, we have:

```matlab
>> mat7t = mat7'
mat7t =
     1
     3
     7
     4
     8
     1
     5
     6
     2
     2
     9
     6
```

### 2.1 Scalar and pointwise operations on matrices

There are several important operations on matrices that fall into different patterns. First we consider scalar operations.

Multiplication by a scalar (i.e., a single number) is achieved by commands such as the following:

```matlab
>> vector*2;
>> matrix*7;
```

It results in all elements of the vector or matrix being multiplied by the same factor. The corresponding operation occurs with division, e.g.:

```matlab
>> matrix/5;
```

This can also be done, in an identical fashion, for scalar addition and subtraction.

In many situations it is necessary to know the size of the matrix before we can dovetail it into a complete calculation. The size of a matrix is defined as the vector
[no. of rows, no. of columns]: it can be found merely by applying the size function, e.g., size(matrix). It should be added that a vector is more conveniently described by its length than by its size. Specifically, a vector is a matrix with a single row, so the size vector necessarily takes the form [1, length], where length is the number of columns in the vector (or equivalently, the number of matrix elements). The following Command Window display should clarify the situation:

```
>> size(mat7)
ans =
  3  4
>> length(vec3)
ans =
   7
>> size(vec3)
ans =
  1  7
```

Pointwise operations on pairs of matrices follow a different pattern. The matrices have to be of identical sizes and combine to form another matrix of identical size. In the case of addition and subtraction, the definition of the combined matrix is obvious—to add or subtract corresponding matrix elements to give the corresponding elements for the combined matrix. This means that, in the case of addition, we can simply write:

```
>> C = A + B;
```

where A, B and C should be taken to mean the complete matrices, including all the matrix elements within them. There is no ambiguity about what is meant here once it is realised that the plus sign has to be interpreted as \( mn \) plus signs, one for each matrix element of each matrix, where \( m \) and \( n \) are the (common) numbers of rows and columns of the three matrices.

With multiplication and division, there is a question of how to write the multiplication, as the usual sign for scalar matrix element multiplication is \( \times \). Instead, pointwise multiplication of whole matrices is denoted by \( \times \), e.g.:

```
>> C = A .* B;
```

Similarly, pointwise division is denoted by \( \div \); there is also a special form of matrix division in which \( \div \) denotes divided into rather than divided by. The need for the latter notation will be dealt with later.

A pointwise operation is also defined for power and is denoted by \( ^\). It could be questioned why pointwise addition and subtraction are not denoted by \( +\) and \( -\). In fact, this is possible in principle but simply unnecessary, as \( +\) and \( -\) carry the required meaning without any further elaboration.

In the case of multiplication, there is a third type of multiplication—namely, the standard form of matrix multiplication used in mathematics: in fact, the mathematical definition of matrix multiplication is given by:
This is a non-trivial combination of addition and multiplication, which is known simply as matrix multiplication because it is so important and crops up frequently in matrix calculations. We see from the mathematical definition that, if two matrices are to be multiplied together, the number of columns in the first matrix (A) must equal the number of rows in the second matrix (B). Assuming that this is so, we can write down the Matlab calculation in the simple form:

```
>> C = A * B;
```

where the sizes of the respective matrices are \([l, n], [l, m], [m, n]\). Note that in some situations, one or other of the matrices may have its rows and columns interchanged: this could happen if we have two matrices each of size \([2, 3]\) which have to be multiplied together. However, it is straightforward to deal with this by taking the transpose of one of the matrices. A case where this commonly happens is when we need to take the dot-product of two vectors \(u = (u_1, u_2, u_3), v = (v_1, v_2, v_3)\). The desired result is the scalar quantity \(u_1v_1 + u_2v_2 + u_3v_3\): in Matlab this is obtained merely by writing \(u*v'\)—i.e., \(v\) is transposed before performing the multiplication.

### 3. Selectively addressing matrix elements

So far we have tried to understand how to set up matrices in Matlab—i.e., how to predeclare them and how to assign values to the matrix elements; we have also sought to adjust the shapes and sizes of the matrices and we have explored how they can be combined using various addition, subtraction, multiplication and division operators. Powerful as the various matrix operations are, we often need to extract particular matrix element values from the matrices and make appropriate computations using them. Indeed, hitherto we have missed some very basic operations, such as how to access individual matrix element values.

In fact, for a matrix \(M\) of size \([m, n]\), the matrix element \((i, j)\)—where \(1 \leq i \leq m, 1 \leq j \leq n\)—is simply accessed in the most obvious manner, viz., \(M(i, j)\). (Note that when this idea is applied to a vector \(V\), the result is even simpler: viz., \(V(i)\)).

Note that square brackets have to be used when defining a matrix, but round brackets have to be used for addressing the elements of a matrix.

We can expand this idea further. It is possible to refer to a whole subset \(M_s\) of a matrix \(M\), merely by listing the relevant rows and columns to be included. For example, we can specify \(M_{s1} = M([1, 3, [2, 3, 5])\), meaning that only rows 1 and 3 and columns 2, 3 and 5 of matrix \(M\) are to be included in the subset matrix \(M_s\). Or if the lists are for unbroken ranges of values, we can use the colon operator for addressing them—e.g., \(M_{s2} = M(1: 3, 2: 5)\). Taking these in turn, we have:
>> M = [ 1 4 5 2 6; 3 8 6 9 3; 7 1 2 6 1; 4 0 7 2 8 ]
M =
 1 4 5 2 6
 3 8 6 9 3
 7 1 2 6 1
 4 0 7 2 8
>> Ms1 = M([1 3],[2 3 5])
Ms1 =
 4 5 6
 1 2 1
>> Ms2 = M(1:3,2:5)
Ms2 =
 4 5 2 6
 8 6 9 3
 1 2 6 1

Fortunately, in Matlab to is possible to use the colon on its own if we need to specify the whole of a row or column. Thus, writing Ms = M(2,:) means that the subset matrix contains (only) the whole of row 2 of M, whereas Ms = M([1 3 4],:) means that the subset matrix contains the whole of rows 1, 3 and 4. In these cases the colon is interpreted as meaning that the column number is unspecified and that all columns must be included.

A colon on its own acts as a wildcard, which can be used for addressing a whole row or column of a matrix.

On some occasions it is useful to eliminate several rows or columns of a matrix. This is achieved by setting the relevant elements to '[]' (this notation can be taken to mean 'empty' or 'null'). For example, if we take vec4 = 3:7 = [3 4 5 6 7] and set vec4(4) = [], the result is that vec4 = [3 4 5 7]. Note that only whole rows and/or whole columns may be removed from matrices, as all rows must have the same numbers of elements, and similarly for columns. For example, we can remove the whole of row 4 by applying the operation: M(4,:) = []; note that while [] refers to an individual matrix element, the colon operator automatically applies the operation to all elements of a row or column (in this example, row 4).

Having learnt the basics of matrix element addressing, there are more exotic operations that can be carried out. In particular, we turn our attention to the task of searching for elements that fulfil particular criteria. The find function is key to this type of operation. Basically, the find function returns the indices of a matrix that meet a prespecified condition.
3.1 The find function

As an example, let us start with vector `vec5` and use the find function to find which elements are equal to 3:

```matlab
cvec5 = [ 4 3 2 3 5 6 ];
>> ffind(cvec5 == 3)
ans =
   2   4
```

Similarly, we can search for elements of `mat8` that are equal to 6:

```matlab
cmat8 = [ 4 3 2 3; 5 6 8 7; 3 1 6 8 ]
cmat8 =
   4   3   2   6
   5   6   8   7
   3   1   6   8
>> ffind(cmat8 == 6)
ans =
   5
   9
  10
```

In the latter case, the result may initially appear surprising, but this is because the operation proceeds sequentially downwards column by column using the technique of linear indexing described earlier. The reason for using linear indexing with the find function is that find is often employed for eliminating a number of the matrix elements—a process that could result in the new matrix having rows or columns of unequal length, and this is not permissible with ordinary matrices. To prevent the output from becoming a shapeless array of numbers, the matrix is transparently converted to a single dimension. In addition, the result of applying the find operation to a matrix is a column vector containing the index values that have been identified. However, note that this procedure is not necessary with a vector, and the result of applying find to a row vector is a row vector—as can be seen from the previous example.

A powerful method for proceeding with the analysis of find-based searches is illustrated by the following example, in which we look for all the negative elements in vector `vec6`:

```matlab
cvec6 = [ -1 0 1 2 3 4 0 -2 0 3 5 ]
cvec6 =
   -1   0   1   2   3   4   0   -2   0   3   5
>> ffnegind = ffind(cvec6 < 0)
ffnegind =
   1   8
>> vec6n = vec6(ffnegind)
vec6n =
   -1   -2
```
Here, \textit{negind} is a row vector containing the list of indices. The above display shows that the negative values of the original vector can be printed out automatically, as well as their indices. However, it is salutory that the whole calculation can be written in the following much simpler form, without overt calculation of the index vector:

\begin{verbatim}
>> vec6n = vec6(find(vec6 < 0))
vec6n =
  -1  -2
\end{verbatim}

It is even more salutory that, if the original vector values are required but not the indices, there is no need to apply the \texttt{find} function: instead a far simpler logical index formulation can be used:

\begin{verbatim}
>> vec6n = vec6(vec6 < 0)
vec6n =
  -1  -2
\end{verbatim}

A further interesting calculation is that of eliminating negative values from a vector. This can be carried out by using the empty element operator, \texttt{[]}. Indeed, it is impressive that this procedure can be applied in a single simple operation without any intermediate variables:

\begin{verbatim}
>> vec6(vec6 < 0) = []
vec6 =
  0  1  2  3  4  0  0  3  5
\end{verbatim}

The reason that this short-cut works properly is because the calculation proceeds one element at a time with no interaction between the elements: it is essentially a parallel processing procedure in which we are applying the necessary conditional operation:

\begin{verbatim}
for all vector elements do {
  if vector element < 0
    mark vector element for removal
}
for all vector elements do {
  remove the marked vector elements and renumber the remaining elements;
}
\end{verbatim}

(Clearly, confusion could result if re-indexing were not carried out in a systematic manner.)

Another way of carrying out searches is to use logical addressing. First, we generate a logical array which indicates whether the result of a conditional expression is true (1) or false (0). This logical array is in exactly the same format as the original matrix or vector, and for \texttt{vec6} gives the result:
Similarly, for matrix \( \text{mat9} \):

```matlab
>> mat9 = [ 1 -4 5 -2; 3 8 -6 9; -7 1 2 -6 ]
mat9 =
     1    4    5   -2
     3    8   -6    9
    -7    1    2   -6
>> logind_m9 = mat9 < 0
logind_m9 =
    0    1    0    1
    0    0    1    0
    1    0    0    1
```

Applying this matrix to the original matrix, \( \text{mat9} \), gives:

```matlab
>> mat9n = mat9(logind_m9)
mat9n =
    -4
    -2
    -6
    -7
    -6
```

Similarly, for \( \text{vec6} \) we get:

```matlab
>> vec6 = [ -1 0 1 2 3 4 0 -2 0 3 5 ];
vec6 =
    -1    0    1    2    3    4    0   -2    0    3    5
>> logind_v6 = vec6 < 0
logind_v6 =
    1    0    0    0    0    0    0    1    0    0    0
```

In both cases, if all we need is the relevant matrix element values, we can shorten the whole procedure in such cases by writing a single overall instruction:

```matlab
>> vec6(vec6 < 0);
>> mat9(mat9 < 0);
```

in each case giving the same results as before.

Overall, the power of the logical indexing approach lies in other things that can be carried out with it, such as listing sets of special values—e.g., those relating to negative element values—rather than attending to the whole of the original matrix.
4. Command Window operators

help provides online help
clc clears the command window: eliminates the clutter resulting from previous commands
clear var eliminates the variable var
clear eliminates all variables from the workspace
disp function used to print a line of text on the screen, e.g.:

```
>> disp('The main calculation is now complete."
The main calculation is now complete.
```
fprintf function used to print formatted text and variables on the screen, e.g.:

```
>> vlength = 6;
>> fprintf('The vector length is %d.\n', vlength)
The vector length is 6.
```
input function used to prompt for values of input variables, e.g.:

```
>> vlength = input('What is the size of the vector? 
What is the size of the vector? 5
vlength =
5
```

5. Other useful operators

dend variable indicating the last element of a vector
numel function indicating the total number of elements in a matrix: for a vector it has the same value as the length of the vector
isequal function giving a logical value showing whether two matrices are identical
sign function outputting a matrix indicating by 1, -1, 0 whether the given matrix elements are positive, negative or zero
cumsum function operating on a row vector to present the cumulative sum of all elements up to the current one
sum function outputting a row vector indicating the sum of each column of a matrix
max function outputting a row vector indicating the maximum of each column of a matrix
min function outputting a row vector indicating the minimum of each column of a matrix
sort function that sorts the elements in each column of a matrix into ascending numerical order; adding a second parameter of 2 makes the function sort the elements of each row into ascending numerical order
rand function for creating matrices of (real) random numbers in the range (0, 1): two parameters are used to specify the matrix size; if the latter is not specified, a single random number is generated
randi function for creating matrices of random integers: a vector is used to specify the range of the integer values, and two parameters are used to specify the matrix size; if the latter is not specified, a single random integer is generated
6. Index

clc, 15
clear var, 15
clear, 15
colon operator, 6
concatenation, 6, 7
cumsum, 15
diag, 6
disp, 15
dot-product, 10
empty, 11
end, 15
eye, 6
find, 11, 12, 13
fprintf, 15
help, 15
input, 15
isequal, 15
length, 9
linear indexing, 7, 12
logical addressing, 13, 14
matrix multiplication, 10
max, 15
min, 15
numel, 15
ones, 5, 6, 8
pointwise operations, 9, 10
addition, 10
subtraction, 10
multiplication, 9
divided by, 9
divided into, 9
power, 10
rand, 15
randi, 15
repmat, 7, 8
reshape, 7
scalar operations, 8, 9, 10
multiply, 8, 9, 10
divide, 8
sign, 15
size, 8, 9
sort, 15
sum, 15
transpose, 8
zeros, 4, 5, 6