1 Introduction

MATLAB, which stands for “matrix laboratory” [4] is a formidable tool for anybody interested in linear algebra and its applications.

2 The Interactive Environment of MATLAB

MATLAB is an interactive system. Commands can be entered at the prompt >> in the command window shown in Figure 1.

It is important to remember that

• MATLAB is case sensitive;
• variables are not typed;
• typing the name of a variable causes the value of the variable to be printed;
• ending a command with a semicolon will suppress the screen display of the results of the command.

In Figure 1 we show the command window of MATLAB.

Commas and semicolons separate statements that are placed on the same line as in

\[
x = 1; 
\]

As we mentioned above semicolons suppress the output. To continue a line on the next line, one ends the line with three periods (...).

MATLAB is equipped with extensive help and documentation facilities. To display a help topic it suffices to type in the command help followed by the name of the topic or to access the MATLAB documentation using the command doc.
3 Number Representation and Arithmetic Computations

In MATLAB we deal with two representations of integers: the unsigned integers and the signed integers. The classes of unsigned integers are denoted by \texttt{unit}k, where \( k \) may assume the values 8, 16, 32 and 64 corresponding to representations over one, two, four and eight bytes, respectively.

**Example 3.1** To create an one-byte unsigned integer having the value 77 we write:

\[
\texttt{>> x = \texttt{uint8}(77)}
\]
\[
x = 77
\]

To obtain information on \( x \) we type \texttt{whos('x')}.

Matlab returns the main characteristics of \( x \):

\[
\texttt{>> whos('x')}
\]

\[
\begin{array}{cccc}
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} & \text{Attributes} \\
\text{x} & 1x1 & 1 & \text{uint8} & \\
\end{array}
\]

Since \( x \) is a scalar, \texttt{MATLAB} regards \( x \) as an \( 1 \times 1 \)-matrix.

The function \texttt{size} returns the sizes of each dimension of an array.

**Example 3.2** For the matrix \( A \) defined by

\[
A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & 1 \end{bmatrix}
\]

we obtain

\[
d = 2 \hspace{1cm} 3
\]

The call

\[
\texttt{[m,n] = \texttt{size}(A)}
\]

returns the size of the matrix \( A \) in separate variables \( m \) and \( n \). More generally,

\[
\texttt{[d1,d2,...,dn]=size(A)}
\]

returns the sizes of the first \( n \) dimensions of the array \( A \) in separate variables.

If we need to return the size of the dimension of \( A \) specified by the scalar \( \text{dim} \) into the variable \( m \) we write

\[
\texttt{m = \texttt{size}(A,\text{dim})}
\]

The function \texttt{isa} determines if its first argument belongs to the type specified by its second argument.

**Example 3.3** The type of \( x \) can be verified by typing

\[
\texttt{>> \texttt{isa}(x,'\texttt{integer}')}
\]
\[
\text{ans} = 1
\]

\[
\texttt{>> \texttt{isa}(x,'\texttt{uint8}')}
\]
\[
\text{ans} =
\]

2
The limits of any of the above integer types can be determined by using the functions `intmin` and `intmax`, as shown below:

**Example 3.4**

```
>> intmin('uint32')
an =
    0
>> intmax('uint32')
an =
   4294967295
```

For any of these unsigned types a number larger than `intmax` is mapped to `intmax` and a number lower than `intmin` is mapped to `intmin`:

```
>> uint8(1000)
an =
    255
>> uint8(-1000)
an =
     0
```

Signed integers are represented in MATLAB using the types of the form `intk`, where `k` ranges among 8, 16, 32 and 64. The leftmost digit is reserved for the sign (1, for a negative integer and 0 for a positive integer). Thus, we have:

```
>> intmax('int8')
an =
    127
>> intmin('int8')
an =
   -128
```

MATLAB makes use of the floating point representation of real numbers. This representation is using a number of \( s+1 \) binary digits and comprises three parts: the *sign bit*, the sequence of *exponent* bits and the sequence of *mantissa* (or *significant*) bits, as shown in Figure 2. The bits are numbered from right to left starting with 0. We assume that the mantissa is using \( p \) bits and the exponent is using \( q \) bits, where \( p + q = s \). Depending on the desired precision, MATLAB supports a single-precision format or a double-precision format and the latter format (as specified by the IEEE Standard 754) is the default representation. For double-precision numbers we have \( s = 63 \). The 64th bit is the bit sign \( b_{63} \) defined as

\[
b_{63} = \begin{cases} 
0 & \text{if } x \geq 0, \\
1 & \text{if } x < 0, 
\end{cases}
\]

\( q = 11 \) bits are reserved for the exponent, and \( p = 52 \) bits are used by the
significant. The number represented by the sequence of bits \((b_{63}, b_{62}, \ldots, b_0)\) is

\[
x = (-1)^{b_{63}} \cdot (1 + \sum_{i=0}^{52} b_{52-i} \cdot 2^{-i}) \cdot 2^{y-1034},
\]

where \(y\) is the equivalent of \((b_1 \cdot \ldots \cdot b_{11})_2\).

For single-precision numbers we have \(s = 32\), the exponent uses \(q = 8\) bits (biased by 127), and the significant uses the remaining \(p = 23\) bits. Accordingly, single-precision numbers are given by

\[
x = (-1)^{b_{31}} \cdot (1 + \sum_{i=0}^{22} b_{22-i} \cdot 2^{-i}) \cdot 2^{y-127},
\]

where \(y\) is the equivalent of \((b_1 \cdot \ldots \cdot b_8)_2\). Single-precision numbers require less memory than double-precision numbers, but they are represented to less precision.

Double-precision numbers are created with assignments such as \(x = 19.43\), because the default format in MATLAB is double-precision. To represent the same number in single-precision we need to write \(y = \text{single}(19.43)\).

It is important to realize that the set of real numbers that are representable in any of these formats is finite. For example, in the case of the double precision format, we have at our disposal 64 bits, which means that only \(2^{64}\) real numbers have an exact representation as double-precision numbers. For single-precision numbers the set of real numbers that have exact representations consists of \(2^{32}\) numbers. The remaining real numbers can be represented only with a degree of approximation, which has considerable consequences for numerical computing.

The finiteness of the set of reals that can be represented exactly on any floating-point system means that there exists a small gap between each double-precision number and the next larger double-precision number. The gap, which limits the precision of computations, can be determined using the \(\text{eps}\) function. If \(x\) is a double-precision number, there are no other double precision numbers in the interval \((x, x + \text{eps}(x))\).

**Example 3.5** The distance between 7 and the next double-precision number can be determined as follows:
As $x$ increases so does $\text{eps}(x)$. We have

\[
\begin{align*}
\text{shift} & \text{(7)} \\
\text{ans} & = \\
8.8818e-016
\end{align*}
\]

Entering $\text{eps}$ without arguments is equivalent to $\text{eps}(1)$. Similar considerations hold for single-precision numbers. Here the gaps between numbers are wider because there are fewer exactly representable numbers.

**Example 3.6** If we define $y$ by $y = \text{single}(7)$, then $\text{eps}(y)$ returns $4.7684e-007$, a value larger than $\text{eps}(7)$ computed above.

**Example 3.7** Let $x = 123/1256$. This number is not a sum of powers of 2, so it cannot be represented exactly as a double-precision number. Consider the following MATLAB dialog:

\[
\begin{align*}
\text{shift} & \text{(70)} \\
\text{ans} & = \\
1.4211e-014
\end{align*}
\]

\[
\begin{align*}
\text{shift} & \text{(700)} \\
\text{ans} & = \\
1.1369e-013
\end{align*}
\]

\[
\begin{align*}
\text{shift} & \text{(7000)} \\
\text{ans} & = \\
9.0949e-013
\end{align*}
\]

Rounding of decimal numbers can cause unexpected results. This phenomenon is known as roundoff error and it can be seen in the next example.

**Example 3.8** Define $x$ as $x = 0.1$. Using the equality test $==$ we have the following results:

\[
\begin{align*}
\text{shift} & \text{(7)} \\
\text{ans} & = \\
1
\end{align*}
\]

\[
\begin{align*}
\text{shift} & \text{(70)} \\
\text{ans} & = \\
0
\end{align*}
\]

\[
\begin{align*}
\text{shift} & \text{(700)} \\
\text{ans} & = \\
0
\end{align*}
\]

\[
\begin{align*}
\text{shift} & \text{(7000)} \\
\text{ans} & = \\
0
\end{align*}
\]
Another risk in performing floating point computations is the inadvertent subtraction of two large and close numbers are subtracted. This may result in a **catastrophic cancellation** as we show next.

**Example 3.9** Consider the following MATLAB computation.

```matlab
>> a = 5
a =
 5
>> b = 5e24
b =
 5.0000e+024
>> c = a + b - b
  c =
        0
```

MATLAB return 0 c rather than 5 since the numbers a + b and b have the same floating-point representation.

The range of double-precision numbers is determined using the functions `realmin` and `realmax`:

```matlab
>> rangeDouble = 'Double-precision numbers range between %g and %g';
>> sprintf(rangeDouble,realmin,realmax)
an =
Double-precision numbers range between 2.22507e-308 and 1.79769e+308
```

When these functions are called with the argument 'single' the corresponding values for the single-precision type are returned:

```matlab
>> rangeSingle = 'Single-precision numbers range between %g and %g';
>> sprintf(rangeSingle,realmin('single'),realmax('single'))
an =
Single-precision numbers range between 1.17549e-038 and 3.40282e+038
```

MATLAB operates with an extended set of reals; the values Inf and -Inf represent real numbers outside the representation ranges, as shown next.

```matlab
realmax('single') + .0001e+038
ans =
   Inf
-realmax('single') - .0001e+038
ans =
   -Inf
```

Values that are not real or complex numbers are represented by a the symbol NaN, an acronym for “Not a Number”. Expressions like 0/0 and Inf/Inf yield NaN, as do any arithmetic operations involving a NaN:

```matlab
x = 0/0
x =
   NaN
```

The imaginary unit of complex numbers is represented in MATLAB by either of two letters: i or j. A complex number can be created in MATLAB by writing

```matlab
z = 3 + 4i
```
or, equivalently, by using the `complex` function:

```matlab
>> z = complex(3,4)
z =
   3.0000 + 4.0000i
```

The real and imaginary parts of a complex number can be obtained using the `real` and `imag` functions:

```matlab
>> x = real(z)
x =
    3
>> y = imag(z)
y =
    4
```

The function `complex`, `real`, and `imag` can be applied to arrays, where they act componentwise:

```matlab
>> x = [1 2 3];
>> y = [4 5 6];
>> z = complex(x, y)
z =
   1.0000 + 4.0000i   2.0000 + 5.0000i   3.0000 + 6.0000i
>> zreal = real(z)
zreal =
    1   2   3
>> zimag = imag(z)
zimag =
    4   5   6
```

Conversions from types are possible using built-in MATLAB functions named after the target type. For example, to convert other numeric data to double precision we can use the MATLAB function `double`.

**Example 3.10** A signed integer created by `y = int64(-123456789122)` is converted to to double-precision floating point by `x = double(y)`.

Arithmetic operators involve the usual arithmetic operators. In MATLAB these operators are overloaded, which means that they can be applied both to numbers and to matrices whose formats accommodate the requirements of these operations. These operators include:
The slash or right matrix division $B/A$ is equivalent to $BA^{-1}$, while the backslash or left matrix division $A\backslash B$ is equivalent to $A^{-1}B$.

## 4 Matrices and Multidimensional Arrays

MATLAB accommodates both real and complex matrices. Matrices can be entered row-wise from the MATLAB console making sure that

(i) elements of a row are separated by spaces or commas;
(ii) rows are separated by a semi-colon;
(iii) the matrix is enclosed between square brackets.

For example, the matrix $A \in \mathbb{C}^{2 \times 3}$

$$A = \begin{pmatrix} 1+i & 1-i & i \\ 2 & i & 3-2i \end{pmatrix}$$

is entered as:

```
>> A = [1+i 1-i i; 2 i 3-2i]
```

MATLAB prints the content of the matrix as

```
A =
    1. + i   1. - i   i
    2.      i     3. - 2. i
```

To inspect the content of the matrix we type its name at the prompt

```
>> A
```

and, again, MATLAB prints the matrix $A$, as above.

Lines can be continued by placing ... at the end of the line. For example, the we can write

```
>> B = [1 -11 22;... 2 22 33;... 3 44 55]
```
to define the matrix 

\[ B = \begin{pmatrix} 1 & -11 & 22 \\ 2 & 22 & 33 \\ 3 & 44 & 55 \end{pmatrix} \]

Scalars can be entered in a rather straightforward manner. To enter \( x = 1 + 3i \) one writes

\( >> x = 1+3i \)

The complex unit may be denoted either by \( i \) or by \( j \).

\textbf{MATLAB} has a special notation for designating contiguous submatrices known as the \textit{colon notation}. To designate the \( i \)th row of a matrix \( A \) we can use the notation \( A(i,:) \). Similarly, the \( j \)th column is designated by \( A(:,j) \).

The submatrix of \( A \) that consists of all the rows between the \( i \)th row and the \( k \)th row is specified by \( A(i:k,:) \). Similarly, the submatrix that consists of the columns of \( A \) between the \( j \)th column and the \( h \)th column is \( A(:,j:h) \).

\textbf{Example 4.1} Let \( A \in \mathbb{R}^{3 \times 4} \) be the matrix

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

Then \( A(:,2:3) \) returns:

\[ \begin{array}{c} 2 \\ 6 \\ 10 \end{array} \]

The colon notation is also used for specifying sequences. A vector that consists of the members of the arithmetic progression \( i, i + r, i + 2r, \ldots, i + pr \), where \( i + pr \leq j \) is designated by the triple \( i : s : j \). If \( s \) is omitted, then we assume that \( s = 1 \).

\textbf{Example 4.2} Consider the following sequences generated using the colon notation:

\[ \begin{array}{c} \gg 1:2:8 \end{array} \]

\[ \begin{array}{c} \gg 1:2:9 \end{array} \]

\[ \begin{array}{c} \gg 5:-0.6:1 \end{array} \]

\[ \begin{array}{c} \gg 1:5 \end{array} \]
Special matrix can be generated using built-in functions as shown below. For example, the function \( \text{eye}(m,n) \) generates a matrix that contains a unit submatrix \( I_p \), where \( p = \min\{m, n\} \), as shown next.

\[
\begin{align*}
\text{>>}I &= \text{eye}(3,3) \\
I &= \\
&\begin{bmatrix}
1. & 0. & 0. \\
0. & 1. & 0. \\
0. & 0. & 1.
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\text{>>}I &= \text{eye}(4,3) \\
I &= \\
&\begin{bmatrix}
1. & 0. & 0. \\
0. & 1. & 0. \\
0. & 0. & 1. \\
0. & 0. & 0.
\end{bmatrix}
\end{align*}
\]

\[
\begin{align*}
\text{>>}I &= \text{eye}(3,4) \\
I &= \\
&\begin{bmatrix}
1. & 0. & 0. & 0. \\
0. & 1. & 0. & 0. \\
0. & 0. & 1. & 0.
\end{bmatrix}
\end{align*}
\]

Random matrices having elements uniformly distributed in the interval \((0, 1)\) can be generated using the function \( \text{rand} \). For example,

\[
\begin{align*}
\text{>>}A &= \text{rand}(p,q) \\
\text{>>}A &= \text{rand}(2,3)
\end{align*}
\]

produces the result

\[
A =
\begin{bmatrix}
0.3616361 & 0.5664249 & 0.3321719 \\
0.2922267 & 0.4826472 & 0.5935095
\end{bmatrix}
\]

Random number producing functions can be used to produce multidimensional arrays whose entries belong to certain intervals.

**Example 4.3** To create a \( 3 \times 4 \times 2 \) 3-dimensional array whose entries belong to the \((0, 1)\) interval we can write

\[
\begin{align*}
\text{>>} A &= \text{rand}(3,4,2) \\
\text{>>} A &= \text{rand}(3,4,2)
\end{align*}
\]

resulting in the array

\[
A(:, :, 1) =
\begin{bmatrix}
0.8147 & 0.9134 & 0.2785 & 0.9649 \\
0.9058 & 0.6324 & 0.5469 & 0.1576 \\
0.1270 & 0.9075 & 0.9575 & 0.9706
\end{bmatrix}
\]
A(:,:,2) =

0.9572  0.1419  0.7922  0.0357
0.4854  0.4218  0.9595  0.8491
0.8003  0.9157  0.6557  0.9340

The number of elements of an array can be determined using the function `numel`, which returns the number of elements of an array $A$. Its effect is equivalent with $\prod(\text{size}(A))$.

Experimentation in data mining is greatly helped by the capability of MATLAB for generating pseudorandom numbers having a variety of distributions.

The function `rand` produces uniformly distributed pseudorandom numbers in the interval $(0,1)$. Depending on the number of arguments, this function can return an $n \times n$ matrix of such numbers (for `rand(n)`), or an $m \times n$ matrix (when using the call `rand(m,n)`). Calls of the form `rand(n,'double')` or `rand(n,'single')` return matrices of numbers that belong to the specified types.

**Example 4.4** To generate a $3 \times 3$ matrix of pseudorandom numbers in $(0,1)$ we write $A = \text{rand}(3)$. This may yield

\[
A =
\begin{bmatrix}
0.4505 & 0.9133 & 0.5383 \\
0.0838 & 0.1524 & 0.9961 \\
0.2290 & 0.8258 & 0.0782
\end{bmatrix}
\]

The function `randi` generates a matrix in $\mathbb{R}^{m \times n}$ whose entries are pseudorandom integers from a uniform discrete distribution on an interval $[h,k]$ if called as `randi([h k], [m,n])`.

**Example 4.5** The call $A = \text{randi}([5, 10],[3, 3])$ returns a $3 \times 3$ matrix with integer components in the interval $[5,10]$:

\[
A =
\begin{bmatrix}
7 & 5 & 10 \\
5 & 9 & 5 \\
10 & 9 & 7
\end{bmatrix}
\]

If interval $[h,k]$ is replaced with a single argument 1, then the `randi` returns a matrix with pseudorandom components in the interval $[1,1]$. For instance, the call $-B = \text{randi}(10,2,4)$—returns a $2 \times 4$ matrix of numbers in the interval $[1,10]$.

The function `permute` rearranges the dimensions of an array in the order specified by its second argument that is a vector.

**Example 4.6** Let $A$ be a 3-dimensional array whose entries belong to the interval $[0,15]$ produced by the statement $A = \text{randi}([0,15],[3,4,2])$:

\[
A(:,:,1) =
\begin{bmatrix}
10 & 6 & 11 & 0 \\
12 & 10 & 0 & 1 \\
11 & 2 & 4 & 13
\end{bmatrix}
\]
To permute the first and third dimensions of the 3-dimensional array \( A \) defined above, we write

\[
B = \text{permute}(A, [3 2 1])
\]

resulting in the array \( B \) having the dimension vector \([2 4 3]\):

\[
B(:, :, 1) = \begin{bmatrix} 10 & 6 & 11 & 0 \\ 11 & 0 & 12 & 7 \end{bmatrix}
\]

\[
B(:, :, 2) = \begin{bmatrix} 12 & 10 & 0 & 1 \\ 5 & 7 & 12 & 7 \end{bmatrix}
\]

\[
B(:, :, 3) = \begin{bmatrix} 11 & 2 & 4 & 13 \\ 15 & 6 & 2 & 10 \end{bmatrix}
\]

The repositioning of the elements of array \( A \) is shown in Figure 3.

To generate pseudorandom numbers following a normal distribution one can use the function `randn`. The call `randn(m)` returns an \( m \times m \)-matrix following...
the normal distribution \( N(0,1) \); using a syntax similar to \texttt{rand} and \texttt{randi} it is possible to generate matrices of other formats. An additional parameter for the functions of this family located last on their list of parameters allow the generation of values that belong to a more restricted type. For example, a call like \texttt{randi(10,100,1,'uint32')} returns an array of 100 4-byte integers, while \texttt{randn(10,'double')} returns an array of \texttt{double} numbers.

The sequence of pseudorandom numbers produced by any of the generating functions is determined by the internal state of an internal uniform pseudorandom number generator. Resetting this generator to the same state allows computations to be repeated.

The function \texttt{normrand} can be used to produce randomly distributed arrays normally distributed. If \( M \) and \( S \) are arrays, then \texttt{normrand} returns an array of random numbers chosen from a normal distribution with mean \( M \) and standard deviation \( S \) having the same format as \( M \) and \( S \). If either \( M \) or \( S \) are scalars, the result is an array having the format of the other parameter. The function \texttt{normrnd(M,S,[p,q])} (that we use next) returns a \( p \times q \) array.

**Example 4.7** We begin by generating 15 random points in \( \mathbb{R}^2 \) using
\[
U = \texttt{normrnd(0,1,[15 2])}
\]
Starting from \( U \) we produce another set of 15 points \( V \) applying a rotation by 30 degrees, a scaling by 0.5 and a translation by 2 and we add some noise. To this end we write

\[
\begin{align*}
>> S &= \begin{bmatrix} \sqrt{3}/2 & -1/2 \; 1/2 & \sqrt{3}/2 \end{bmatrix}; \\
>> V &= \texttt{normrnd(0.5*U*S+2,0.05,[15 2])}
\end{align*}
\]

Submatrices can be extracted by indicating ranges of indices or by using the placeholder \( ; \), which stands for all rows or columns, respectively. To extract the second column of the matrix \( A \) we write:

\[
\begin{align*}
\texttt{A(:,2)} = 0.5664249 \\
&0.4826472
\end{align*}
\]

We list below several built-in functions that create or process matrices.
Function | Description
--- | ---
ones($m,n$) | creates a matrix in $\mathbb{R}^{m \times n}$ containing 1s
zeros($m,n$) | creates a matrix in $\mathbb{R}^{m \times n}$ containing 0s
eye($m,n$) | creates a matrix in $\mathbb{R}^{m \times n}$ having 1s on its main diagonal
toeplitz($u$) | creates a Toeplitz matrix whose first row is $u$
diag($u$) | creates a diagonal matrix having $u$ on its main diagonal
diag($A$) | yields the diagonal of matrix $A$
triu($A$) | gives the upper part of $A$
tril($A$) | gives the lower part of $A$
linspace($a,b,n$) | creates a vector in $\mathbb{R}^n$ whose components divide $[a,b]$ in $n-1$ equal subintervals
kron($A$, $B$) | computes the Kronecker product of $A$ and $B$
rank($A$) | rank of a matrix $A$
$A'$ | the Hermitian adjoint $A^H$ of $A$
$A + B$ | the sum of matrices $A$ and $B$

Operations introduced by a dot apply elementwise. As an example consider operations specified below:

- $A^2$ a matrix that contains the squares of the components of $A$
- $A \ast B$ the Hadamard product of $A$ and $B$
- $A/B$ a matrix that contains the ratios of corresponding components of $A$ and $B$

If $A$ is a real matrix, then $A'$ is the transpose of $A$; for complex matrices $A'$ denotes the Hermitian adjoint $A^H$.

Occasionally, we need to make sure that numerical representation of a symmetric matrix $A \in \mathbb{R}^{n \times n}$ does not affect its symmetry. This can be achieved by the statement $A = (A + A')/2$.

The function diag mentioned above has several variants. For example, diag($V,k$), where $V$ is an $n$-dimensional vector and $k$ is an integer returns a square matrix of size $n + |k|$ having the components of $V$ on its $k$th diagonal. The value $k = 0$ corresponds to the main diagonal, positive values correspond to diagonals above the main diagonal, and negative values of $k$ refer to diagonals below the main diagonal. The effect of diag($V$) is identical to the effect of diag($V,0$).

If $X$ is a matrix, diag($X,k$) a vector whose components are the elements of the $k$th diagonal of $X$.

**Example 4.8** The expression

\[
\text{diag(ones(4,1)) + diag(ones(3,1),1) + \diag(ones(3,1),-1)}
\]

generates the tridiagonal matrix

```
ans =
  1 1 0 0
  1 1 1 0
  0 1 1 1
  0 0 1 1
```
The function `blkdiag` produces a block-diagonal matrix from its matrix input arguments.

**Example 4.9** If we start with the matrices defined by

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

the call `blkdiag(A,B,C)` will produce the matrix

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

Block matrices can be formed using vertical or horizontal concatenation of matrices. To concatenate vertically two matrices \(A\) and \(B\) the numbers of column of \(A\) must equal the number of columns of \(B\) and the operation is realized as \(C = [A ; B]\). Horizontal concatenation requires equality of the numbers of rows and can be obtained as \(E = [A \ D]\).

**Example 4.10** Let \(A, B\) and \(D\) be the matrices

\[
\begin{array}{c}
>> A=[1 2 3; 4 5 6] \\
A = \\
1 & 2 & 3 \\
4 & 5 & 6 \\
\end{array}
\]

\[
\begin{array}{c}
>> B= [7 8 9; 10 11, 12; 13 14 15] \\
B = \\
7 & 8 & 9 \\
10 & 11 & 12 \\
13 & 14 & 15 \\
\end{array}
\]

\[
\begin{array}{c}
>> D = [17 18; 19 20] \\
D = \\
17 & 18 \\
19 & 20 \\
\end{array}
\]

The vertical concatenation of \(A\) and \(B\) and the horizontal concatenation of \(A\) and \(D\) are shown below.

\[
\begin{array}{c}
>> C = [A \ ; \ B] \\
C = \\
1 & 2 & 3 \\
\end{array}
\]
The function \texttt{repmat} replicates a matrix \( A \) (referred to as a \textit{tile}) and creates a new matrix \( B \) which consists of an \( m \times n \) tiling of copies of \( A \) when we use the call \( B = \texttt{repmat}(A,m,n) \). The format of \( B \) is \( (pm) \times (qn) \), when the format of \( A \) is \( p \times q \). The statement \texttt{repmat}(A,n) creates an \( n \times n \) tiling.

\textbf{Example 4.11} Let \( A \) be the matrix

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

To create a \( 3 \times 2 \) tiling using \( A \) as a tile we write \( B = \texttt{repmat}(A,3,2) \), which results in

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

Real matrices can be sorted in ascending or descending order using the function \texttt{sort}. If \( v \) is a vector, \texttt{sort}(\( v \)) sorts the elements of \( v \) in ascending order. If \( X \) is a matrix, \texttt{sort}(\( X \)) sorts each column of \( X \) in ascending order. This function is polymorphic: if \( X \) is an array of strings, then \texttt{sort}(\( X \)) sorts the strings in ASCII dictionary order.

Another variant of \texttt{sort}, \texttt{sort}(\( X,d,m \)) sorts \( X \) on the dimension \( d \), in either ascending or descending order, as specified by the third parameter \( m \), which can assume the values 'ascend' or 'descend'. If this function is called with two output parameters as in \([Y,I] = \texttt{sort}(X,d,m)\), then the function returns additionally an index matrix.

\textbf{Example 4.12} Starting from the unidimensional array

\[
\begin{bmatrix}
9 & 2 & 8 & 5 & 11 & 3 & 7 \\
\end{bmatrix}
\]

the function call \([Y,I] = \texttt{sort}(X)\) returns the matrices
$Y = \begin{bmatrix} 2 & 3 & 5 & 7 & 8 & 9 & 11 \end{bmatrix}$

$I = \begin{bmatrix} 2 & 6 & 4 & 7 & 3 & 1 & 5 \end{bmatrix}$

Clearly, $Y$ contains the sorted element of $X$, while $I$ gives the position of each of the elements of $Y$ in the original matrix $X$.

If $X$ is a complex matrix, the elements are sorted in the order of their absolute values and elements that tie for the absolute value are sorted in the order of their angle.

A sparse matrix is a matrix whose elements are, to a large extent, equal to 0. Such a matrix can be represented by the position and value of its non-zero elements using the MATLAB function `sparse`.

**Example 4.13** Starting from the matrix $A$

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

the function call $B = \text{sparse}(A)$ returns $B$, the representation of $A$ as a sparse matrix:

$B = \begin{bmatrix} \text{(1,1)} & 1 \\ \text{(2,1)} & 3 \\ \text{(2,2)} & 1 \\ \text{(1,4)} & 2 \end{bmatrix}$

The format of $B$ is the same as the format of $A$.

All matrix operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices. In most cases, operations on mixtures of sparse and full matrices return full matrices. The exceptions include situations where the result of a mixed operation is structurally sparse. For instance, the Hadamard product $A \cdot S$ is at least as sparse as $S$.

**Example 4.14** Consider the matrix $D$

$D = \begin{bmatrix} 5 & 2 & 4 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \\ 1 & 0 & 3 \end{bmatrix}$

Its sparse form is

$E = \begin{bmatrix} \text{(1,1)} & 5 \\ \text{(4,1)} & 1 \\ \text{(1,2)} & 2 \\ \text{(2,2)} & 1 \\ \text{(1,3)} & 4 \\ \text{(2,3)} & 2 \end{bmatrix}$
The sparse form of the matrix $A \cdot D$ can be computed as $\text{sparse}(A \cdot D)$ and is

\[
\begin{array}{cccc}
(1,1) & 7 \\
(2,1) & 15 \\
(1,2) & 2 \\
(2,2) & 7 \\
(1,3) & 10 \\
(2,3) & 14 \\
\end{array}
\]

The same result can be obtained by multiplying the sparse forms of the matrices $A$ and $D$, that is, by writing $B \cdot E$.

The function call $S = \text{sparse}(i,j,s,m,n,nzmax)$ uses three vectors of equal length $i$, $j$, and $s$ to generate an $m \times n$ sparse matrix $S$ such that $S(i(k),j(k)) = s(k)$, with space allocated for $\text{nzmax}$ nonzeros. Any elements of $s$ that are zero are ignored, along with the corresponding values of $i$ and $j$. Any elements of $s$ that have duplicate values of $i$ and $j$ are added together.

To convert a sparse matrix to a full representation we can use the function $\text{full}$. Its use is illustrated in Example 4.16.

The function call $[B,d] = \text{spdiags}(A)$ starts with a rectangular matrix $A$ of format $m \times n$ and returns a sparse matrix containing all non-zero diagonals of $A$. The components of the vector $d$ indicate the position of these diagonals. **Example 4.15** If $A$ is the $3 \times 4$ matrix

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

that has 6 diagonals. The call $[B,d] = \text{spdiags}(A)$ produces the results

\[
B =
\begin{bmatrix}
0 & 0 & 1 & 2 & 3 & 4 \\
0 & 5 & 6 & 7 & 8 & 0 \\
9 & 10 & 11 & 12 & 0 & 0
\end{bmatrix}
\]

\[
d =
\begin{bmatrix}
-2 \\
-1 \\
0 \\
1 \\
2 \\
3
\end{bmatrix}
\]

*Note that the first column of $B$ is the $-2^\text{nd}$ diagonal of $A$ that consists of 9, the second column is the $-1^\text{st}$ diagonal of $A$, etc.*

Other useful formats of $\text{spdiags}$ exist. We mention just one, $A = \text{spdiags}(B,d,m,n)$ that creates an $m \times n$ sparse matrix $A$ from the columns of $B$ and places them along the diagonals specified by the vector $d$. 

18
Example 4.16 Starting with the matrix \( B \) and the vector \( d \) given by

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

\[
d = \\
-2 & -1 & 0 & 1
\]

the call \( A = \text{spdiags}(B,d,5,4) \) produces the sparse matrix

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

The normal representation of matrices can be obtained by applying the function \( \text{full} \). Thus, \( C = \text{full}(A) \) produces

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

Note that if the length of the diagonals of \( C \) is insufficient to accommodate the full columns of \( B \), these columns are truncated.

Multidimensional arrays generalize matrices. Such arrays can be created starting with matrices and then extending these matrices.

Example 4.17 After creating the matrix \( A \) as

\[
>> A = [1 0 2 1; -1 1 0 3; 1 2 3 4]
\]

\[
A =
\begin{bmatrix}
1 & 0 & 2 & 1 \\
-1 & 1 & 0 & 3 \\
1 & 2 & 3 & 4
\end{bmatrix}
\]

19
a second matrix with the same format as the initial matrix can be added by writing:

\[ A(:,:,2) = \begin{bmatrix} 5 & 1 & 6 & 7 \\ 8 & 9 & 0,2 & -1 \\ -1 & 3 & 1 & 0 \end{bmatrix} \]

This results in an multidimensional array having the format \(3 \times 4 \times 2\) that is displayed as

\[
A(:,:,1) = \\
1 & 0 & 2 & 1 \\
-1 & 1 & 0 & 3 \\
1 & 2 & 3 & 4 \\
A(:,:,2) = \\
5 & 1 & 6 & 7 \\
8 & 9 & 0 & 2 \\
-1 & 3 & 1 & 0 \\
\]

Multidimensional array can be created using the concatenation function \texttt{cat}.

**Example 4.18** To add a third page to the previously created array \(A\) we write

\[
A = \texttt{cat}(3, A, \begin{bmatrix} 6 & 7 & 0 & 8 \\ -1 & -2 & -3 & 0 \\ 4 & 5 & 6 & 3 \end{bmatrix})
\]

resulting in

\[
A(:,:,1) = \\
1 & 0 & 2 & 1 \\
-1 & 1 & 0 & 3 \\
1 & 2 & 3 & 4 \\
A(:,:,2) = \\
5 & 1 & 6 & 7 \\
8 & 9 & 0 & 2 \\
-1 & 3 & 1 & 0 \\
A(:,:,3) = \\
6 & 7 & 0 & 8 \\
-1 & -2 & -3 & 0 \\
4 & 5 & 6 & 3 \\
\]

A fourth page of this array that consists of repeated occurrences of 8 can be added by writing

\[
A(:,:,4) = 8 
\]
The functions `prod` and `sum` return the product of the elements and the sum of the elements of an array, respectively. The sum of the elements of a matrix $A$ can be computed along any of its dimensions by using the function `sum(A,d)`, where $d$ is the dimension. The function `sum(A)` computes the sum of all elements of $A$. For `prod` similar conventions apply.

**Example 4.19** For the matrix

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

we can execute the following computations:

```matlab
>> sum(X)
ans =
   11  10   2   7
>> sum(X,1)
ans =
   11  10   2   7
>> sum(X,2)
ans =
   18   3
```

For the three-dimensional array created in Example 4.17, `prod(A,[1,2])` returns an $1 \times 1 \times 3$ array whose elements are the products of each page of $A$; similarly, `sum(A,[1,2])` returns an array whose elements are the sums of each page of $A$:

```matlab
>> sum(A,[1,2])
ans(:,:,1) =
   17
ans(:,:,2) =
   41
ans(:,:,3) =
   33
```

The `repmat` function can be used to produce multidimensional arrays. Its format in this case is `repmat(A,r)`, where $A$ is an array and $r$ is a vector that specifies the repetition scheme.

**Example 4.20** Let $A$ be the matrix defined by

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
\end{bmatrix}
\]
Copies of the matrix $A$ are repeated in a $2 \times 3 \times 2$ multiarray by writing

$$B = \text{repmat}(A, [2 3 2])$$

This results in

$B(:,:,1) =$

$$
\begin{array}{ccccccc}
1 & 2 & 3 & 1 & 2 & 3 & 1 \\
4 & 5 & 6 & 4 & 5 & 6 & 4 \\
1 & 2 & 3 & 1 & 2 & 3 & 1 \\
4 & 5 & 6 & 4 & 5 & 6 & 4 \\
\end{array}
$$

$B(:,:,2) =$

$$
\begin{array}{ccccccc}
1 & 2 & 3 & 1 & 2 & 3 & 1 \\
4 & 5 & 6 & 4 & 5 & 6 & 4 \\
1 & 2 & 3 & 1 & 2 & 3 & 1 \\
4 & 5 & 6 & 4 & 5 & 6 & 4 \\
\end{array}
$$

If, instead, we have written

$$C = \text{repmat}(A, [2 3 1])$$

this would have resulted in

$C =$

$$
\begin{array}{ccccccc}
1 & 2 & 3 & 1 & 2 & 3 & 1 \\
4 & 5 & 6 & 4 & 5 & 6 & 4 \\
1 & 2 & 3 & 1 & 2 & 3 & 1 \\
4 & 5 & 6 & 4 & 5 & 6 & 4 \\
\end{array}
$$

Obviously, the last dimension ($1$) of $C$ is useless and can be eliminated using the function $\text{squeeze}$ which removes remove dimensions of length $1$ of an array:

$$D = \text{squeeze}(C)$$

5 Cell Arrays

A cell array is an array whose components called $\text{cells}$ can contain any type of data.

A cell array can be created using the constructor $\{\}$.  

**Example 5.1** To create a $4 \times 2$-cell array containing numbers, strings and arrays we can write

\[
\text{cellAr} = \{1, 9, 4, 3; 'Febr', \text{randi}(2,3), \{11; 12; 13\}, \text{randi}(2,2)\}
\]

resulting in
cellAr =
2 x 4 cell array
{'Febr'} {3 x 3 double} {3 x 1 cell} {2 x 2 double}

An empty cell array can be created using C = {}; to create an empty cell array of format 2 x 3 x 4 we can write D = cell(2,3,4).

**Example 5.2** A cell array of text and data can be created by writing

A = {'one','two','three';1,2,3}

resulting in

A =
2 x 3 cell array
{'one'} {'two'} {'three'}
{[ 1]} {[ 2]} {[ 3]}

Components of a cell array can be accessed individually using curly braces, or as set of cells using small parentheses.

**Example 5.3** Here A refers to the cell array introduced in Example 5.2. To access the component (2,3) of A we write A{2,3}.
To access the leftmost four cells of the cell array A we write A(1:2,1:2) and obtain

2 x 2 cell array

{'one'} {'two'}
{[ 1]} {[ 2]}
Example 6.1 The result of a comparison between the arrays \( x \) and \( y \) is the array \( \text{ans} \) given below:

\[
\begin{align*}
\text{>> } x &= [1 \ 5 \ 2 \ 4 \ 9 \ 6] \\
x &= \begin{bmatrix}
1 & 5 & 2 & 4 & 9 & 6
\end{bmatrix} \\
\text{>> } y &= [7 \ 3 \ 1 \ 3 \ 2 \ 9] \\
y &= \begin{bmatrix}
7 & 3 & 1 & 3 & 2 & 9
\end{bmatrix} \\
\text{>> } x > y \\
\text{ans} &= \begin{bmatrix}
0 & 1 & 1 & 1 & 1 & 0
\end{bmatrix}
\end{align*}
\]

Example 6.2 A scalar \( a \) can be compared with an array \( B \). The result is an array of logical values having the same format as \( B \).

\[
\begin{align*}
a &= 5 \\
\text{>> } B &= [1 \ 5 \ 2; \ 4 \ 9 \ 6; \ 6 \ 1 \ 7; \ 7 \ 3 \ 1] \\
B &= \begin{bmatrix}
1 & 5 & 2 \\
4 & 9 & 6 \\
6 & 1 & 7 \\
7 & 3 & 1
\end{bmatrix} \\
\text{>> } a <= B \\
\text{ans} &= \begin{bmatrix}
0 & 1 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 0 & 0
\end{bmatrix}
\end{align*}
\]

Logical values result from application of a relational operator or through the application of certain functions and, as we saw, can be grouped in arrays. An array of logical values can be used to extract certain components of a numerical array.

The function `logical` converts a numerical array to a logic array. For example, \( M = \text{logical(eye(3))} \) results in array of logical values:

\[
M = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

If \( A \) is an array and \( M \) is a logical array having the same format, then \( A(M) \) returns the values of \( A \) at the indices where the components of \( M \) are equal to 1, as in

\[
\text{>> } A &= [1 \ 2 \ 3; \ 4 \ 5 \ 6; \ 7 \ 8 \ 9] \\
A &= \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]
>> A(1)
ans =
 1
 5
 9

The `find` function applied to an array produces a list of the positions where the components of the array are non-zero.

For an one-dimensional array `X`, `find(X)` returns an one dimensional array `I`, as in the following example:

```matlab
X =
 2 0 -1 0 0 3
>> I = find(X)
I =
 1 3 6
```

If `X` is a matrix, `I` contains the places of the non-zero components of `X`, where `X` is regarded as an array obtained by concatenating its columns vertically. For example, we have

```matlab
X =
 1 0 3 0 2
 0 0 2 1 1
>> I = find(X)
I =
 1 5 6 8 9 10
```

The call `[I,J] = find(X)` returns the row and column indices of the non-zero components of `X`, as shown next:

```matlab
I =
 1
 1
 2
 2
 1
 2
J =
 1
 3
 3
 4
 5
 5
```

**Example 6.3** Let `A` and `B` two matrices:
if logical expression statements

elseif logical expression statements

else statements end

Figure 4: Syntactic Diagram of the if-then-else structure

>> A = [1 2 5; 3 0 2; 6 1 7]
A =
     1     2     5
     3     0     2
     6     1     7
>> B = [5 9 1; 4 3 8; 5 6 7]
B =
     5     9     1
     4     3     8
     5     6     7

The answer to the comparison A < B is the logical matrix

    1     1     0
    1     1     1
    0     1     0

If a matrix is regarded as an array obtained by concatenating its columns vertically, then C = find(A < B) will return

C =
    1
    2
    4
    5
    6
    8

MATLAB has four basic control structure: if-then-else, for, while and switch. Their semantics, discussed next, is somewhat different from similar control structures in common programming languages.

The syntactic diagram of the if-then-else structure is shown in Figure 4.
Example 6.4 The following piece of MATLAB code

```matlab
if(sum(sum(A < B))== size(A,1)*size(A,2))
    disp('A is less than B')
elseif(sum(sum(A > B))== size(A,1)*size(A,2))
    disp('A is greater than B')
elseif(sum(sum(A == B))== size(A,1)*size(A,2))
    disp('A and B are equal')
else
    disp('A and B are incomparable')
end
```

applied to two matrices \( A \) and \( B \) having the same format will return

\( A \) and \( B \) are incomparable

when

\[
A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 2 & 1 \\ 6 & 6 \end{pmatrix}.
\]

The function `sum` when applied to a matrix returns a vector that contains the sum of the columns of the matrix. Therefore, `sum(sum(A))` returns the sum of elements of \( A \).

The calls `size(A,1)` and `size(A,2)` return the dimensions of \( A \). Note that a condition like `sum(sum(A < B))== size(A,1)*size(A,2)` is satisfied if all entries of the matrix \( A < B \) are equal to 1, that is, if every element of \( A \) is less than the corresponding element of \( B \).

The syntax of the `for` structure is shown in Figure 5.

The list of values can be entered as a vector or as a colon expression.

Example 6.5 Either

```matlab
>> for i=[5 7 9 11]
    disp(sqrt(i))
end
```

or

```matlab
>> for i=5:2:12
    disp(sqrt(i))
end
```
The syntax diagram of the while structure is given in Figure 6.

**Example 6.6** The sum of the terms of the harmonic series that are not smaller than 0.01 is computed by the following code fragment:

```plaintext
s = 0;
n = 1;
while(1/n >= 0.01)
    s = s + 1/n;
    n = n + 1;
end;
fprintf('Sum is %g\n',s)
```

which returns

Sum is 5.18738

The switch structure has the following syntactic diagram:

The statements following the first case where the switch expression matches the case expression are executed; the second list of statements is executed when the second case expression matches the switch expression. If no case expression is a match for the switch expression then the statements that follow otherwise are executed, if this option exists. Unlike the similar C structure, only one case is executed.

The switch expression can be a scalar or a string.

**Example 6.7** The following code fragment

```plaintext
>> animal = cougar;
>> switch lower(animal)
    case {'ostrich','emu','turkey'}
        disp('bird');
    case {'tiger','lion','cougar','leopard'}
        disp('cat');
    case {'alligator','crocodile','frog'}
        disp('reptile');
end
```

28
will display cat.

7 Indexing

Access to individual component of vectors and matrices can be done by indexing. For example, to access the component $a_{ij}$ of matrix $A$ we write $A(i,j)$.

The index of a vector can be another vector.

Example 7.1 Let $v = [1:3:22]$ and let $u = [1 \ 4 \ 5]$. Then, we have

```
>> v=1:3:22
v =
    1   4    7   10   13   16   19   22
>> u=[1 4 5]
u =
    1   4    5
>> v(u)
an =
    1   10   13
```

This technique can be applied to swapping the two halves of $v$ by writing

```
>> v([5:8 1:4])
an =
    13   16   19   22    1   4    7   10
```
The special operator `end` allows us to access the final position of a vector as shown next.

**Example 7.2** For the vector `v` defined in Example 7.1, the expression `v(1:2:end)` returns

```
ans =
  1  7 13 19
```

that is, the components of `v` located in odd-numbered positions.

A popular technique in MATLAB is the **logical indexing**. If the indexing expression is of logical type, an indexing expression will extract those elements of the array that make that expression `true`.

**Example 7.3** To extract the even number components of the vector `v` defined in Example 7.1 we can write `v(mod(v,2)==0)`. This will result in

```
ans =
  4 10 16 22
```

**Example 7.4** Suppose that we have a matrix that has entries that are not defined; such entries can be represented by the special value `NaN` (not a number) and can be recognized by the logical function `isnan` that returns `true` if its input is `NaN`. Starting from the matrix

```
>> X = [1 NaN 3 4; 8 -2 NaN NaN; 0 1 NaN 5]
X =
  1   NaN  3   4
  8  -2   NaN   NaN
  0   1   NaN   5
```

the statement `X(isnan(X))=0` results in the substitution of all `NaN` values by 0:

```
X =
  1   0   3   4
  8  -2   0   0
  0   1   0   5
```

### 8 Functions

Functions can be created in files having the extension `.m` which have the same name as the functions they define. In Example 8.1 we define the function `kronsum`; thus, the file is named `kronsum.m`.

The first line of the file has the form

```
function [Y1,...,Yq] = f(X1,...,Xp)
```

where `X1,...,Xp` are the input arguments, `f` is the name of the function and `Y1,...,Yq` are the values computed by the function.

**Example 8.1** The Kronecker’s sum, `A\oplus B` is computed by the function `kronsum` shown below.
function [S] = kronsum(A,B)
%KRONSUM computes the Kronecker sum of matrices A and B
        if (ndims(A) ~= 2 || ndims(B) ~= 2)
            return;
        end
        [rowsA,colsA] = size(A);
        [rowsB,colsB] = size(B);
        if (rowsA ~= colsA) || (rowsB ~= colsB)
            return;
        end
        S = kron(eye(colsB),A) + kron(B,eye(colsA));
end

We ensure that the arguments presented to \texttt{kronsum} are matrices by using
the function \texttt{ndims} that returns the number of dimensions of the arguments.
The formats of the arguments are computed using the function \texttt{size} and the
computation proceeds only if the two arguments are square matrices. Finally,
the last line of the function computes effectively the value of the result.

Note the presence of comment lines introduced by \%. The first comment line
is returned when we use the command \texttt{lookfor} or request help.

\textbf{Example 8.2} The function \texttt{datagen} serves as a generator of data sets in $\mathbb{R}^2$
that contain a prescribed number of points that are grouped around given centers.

definitions = datagen(spec)
%DATAGEN produces a set of points in $\mathbb{R}^2$ starting from a
% k x 4 matrix called spec. Points are grouped in
% k clusters; the centers of the j-th cluster has
% coordinates [spec(j,1) spec(j,2)]; the j-th cluster
% contains spec(j,3) points and the diameter is
% spec(j,4)
%
% determine the number of clusters
noc = size(spec,1);
%
% npgen gives the number of points currently generated
npgen = 0;

for j=1:noc
    centr = [spec(j,1) spec(j,2)];
    T(npgen+1:npgen+spec(j,3),:) = ... 
    spec(j,4)*randn(spec(j,3),2)+...
    repmat(centr,spec(j,3),1);
    npgen = npgen+spec(j,3);
end

\textit{Starting from the matrix} \texttt{spec}

\begin{verbatim}
spec =
    1.0000 1.0000 50.0000 1.0000
    1.0000 8.0000 40.0000 1.3000
\end{verbatim}
and calling the function defined above, \( T = \text{datagen}(\text{spec}) \) we obtain the set of objects shown in Figure 8.

9 Matrix Computations

As a “matrix laboratory”, MATLAB offers built-in functions for a wide variety of matrix computations. We offer a few example that involve commonly used functions.

The function \( \text{trace} \) can be applied only to a square matrix \( A \) and \( \text{trace}(A) \) returns \( \text{trace}(A) \).

The function \( \text{abs} \) applied to a matrix \( A \) returns the matrix of absolute values of the elements of \( A \).

Example 9.1 Let \( A \) be the matrix

\[
A = \begin{pmatrix}
1.0000 + 1.0000i & 3.0000 + 4.0000i \\
2.0000 - 5.0000i & -7.0000
\end{pmatrix}
\]

We obtain:

\[
>> B = \text{abs}(A)
\]

\[
B = \\
1.4142 & 5.0000 \\
5.3852 & 7.0000
\]
The function \texttt{rref} produces the reduced row echelon form of a matrix \( A \), when called as \( \texttt{R = rref(A)} \). A variant of this function, \([\texttt{R,v}] = \texttt{rref(A)}\) also yields a vector \( v \) so that \texttt{length(r)} is the rank of \( A \), and \( A(:,r) \) is a basis for the range of \( A \). Roundoff errors may cause this algorithm to produce a rank for \( A \) that is different from the actual rank.

\textbf{Example 9.2} \textit{Starting from the matrix}

\begin{verbatim}
A =
1  2  3  4  5  6
7  8  9 10 11 12
1  3  5  7  9 11
\end{verbatim}

The function call \([\texttt{R,r]} = \texttt{rref(A)}\) returns

\begin{verbatim}
R =
1  0 -1  -2  -3 -4
0  1  2  3  4  5
0  0  0  0  0  0
\end{verbatim}

\begin{verbatim}
r =
1  2
\end{verbatim}

showing that the rank of \( A \) is 2.

The \textit{LU} decomposition of a matrix \( A \) is computed by the function call \([\texttt{[L,U,P]} = \texttt{lu(A)}\) which returns a lower triangular matrix \( L \), an upper triangular matrix \( U \) and a permutation matrix \( P \) such that \( PA = LU \).

\textbf{Example 9.3} \textit{Starting from the matrix}

\begin{verbatim}
>> A=[1 0 1; 2 1 1; 1 -1 2]
A =
1  0  1
2  1  1
1 -1  2
\end{verbatim}

considered in Example ?? and applying the function \texttt{lu} we obtain:

\begin{verbatim}
>> [L,U,P]=lu(A)
L =
1.0000  0  0
0.5000  1.0000  0
0.5000  0.3333  1.0000
U =
2.0000  1.0000  1.0000
0  -1.5000  1.5000
0  0  0
P =
0  1  0
0  0  1
1  0  0
\end{verbatim}

When the function \texttt{lu} is called with two output arguments, the first argument contains the matrix \( P'L \), where \( P' \) is the permutation matrix discussed before.
Of course, the first output argument is not a lower triangular matrix, as shown next.

```matlab
>> [L,U] = lu(A)
L =
   0.5000    0.3333    1.0000
   1.0000         0    0.0000
   0.5000    1.0000         0
U =
    2.0000    1.0000    1.0000
    0   -1.5000    1.5000
    0         0         0
```

### Exercises and Supplements

1. Write an MATLAB function that when applied to a matrix \( A \) returns bases for both the null space of \( A \) and the range of \( A \).

2. Write an MATLAB function that starts with three real numbers \( x, y, z \) and an integer \( n \) and returns a tri-diagonal \( n \times n \)-matrix having all elements on the main diagonal equal to \( a \), all elements immediately located under diagonal equal to \( b \) and all elements immediately above the diagonal equal to \( c \).

3. Write an MATLAB function that returns a true value if the number of nonzero components of a vector of integers is odd.

To measure the time used by MATLAB operations one can use a pair of MATLAB functions named `tic` and `toc`. The function `tic` (with no argument) starts a stopwatch; `toc` reads the stopwatch and displays the elapsed time in seconds since the most recent invocation of `toc`.

A similar role can be played by the function `cputime` which returns the CPU time in seconds that was used by the MATLAB computation since this computation began. For example,

```matlab
t = cputime;
n
n
ctime -t
```
returns the cpu time elapsed between these consecutive calls of `cputime`.

4. Write an MATLAB function that starts from two integer parameters \( n \) and \( k \) with \( k \leq n \) and returns a matrix \( S \in \mathbb{R}^{k \times m} \) whose columns contain the distinct subsets of the set \( \{1, \ldots, n\} \) with no more than \( k \) elements, so \( m = 1 + \binom{n}{1} + \cdots + \binom{n}{k} \). For example if \( n = 4 \) and \( k = 3 \) the function
should return the matrix

\[
\begin{array}{ccccccccccccc}
0 & 1 & 2 & 3 & 4 & 1 & 1 & 2 & 2 & 3 & 1 & 1 & 2 \\
0 & 0 & 0 & 0 & 0 & 2 & 3 & 4 & 4 & 4 & 2 & 2 & 3 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 4 & 4 & 4
\end{array}
\]

Compute and plot the time used by the algorithm for various values of \( n \) and \( k \).

5. Write an MATLAB function that tests if two integer matrices commute; examine the difficulties of writing such a function that deals with real-number matrices.

6. Write an MATLAB script that picks \( n \) points at random on the circle of radius 1 and draws the corresponding polygonal contour.

**Hint:** Generate uniformly \( n \) angles \( \alpha_1, \ldots, \alpha_n \) in the interval \([0, 2\pi]\) and then graph the points \((\cos \alpha_i, \sin \alpha_i)\).

The MATLAB function \texttt{fzero} can be used to determine the zeros \( z \) of a nonlinear single-argument function \( f : \mathbb{R} \to \mathbb{R} \) located near a number \( u \in \mathbb{R} \) using two arguments: a string \( s \) describing the function to investigate and the number \( u \). For example to find the zeros of the function \( f \) given by \( f(x) = x^3 - 6x^2 + 11x - 6 \) we write:

\[
>> z = \text{fzero}('x^3 -6*x^2 + 11*x -6',1.3)
\]

which results in

\[
\begin{align*}
z &= 1.0000 \\
&
\end{align*}
\]

An alternative technique is to use an anonymous function; such a function can be written as

\[
F = @(x)x^3 -6*x^2 + 11*x -6
\]

Then, we could write

\[
z = \text{fzero}(F,4)
\]

which results in

\[
\begin{align*}
z &= 3 \\
&
\end{align*}
\]

7. Experiment with the family of functions \( f_a : \mathbb{R} \to \mathbb{R} \) defined by \( f(x) = x^3 - 10x^2 + 21x + a \) for \( a \in \mathbb{R} \) and determine the number of zeros for various values of \( a \).

8. Consider the complex row vector:

\[
>> v = [1+i 2-i 1.5+2*i]
\]

Which MATLAB expression will compute \( v^m \) correctly, \( v' \) or \( v.^' \) and why?
9. The least element of a matrix may occur in several positions. Write a MATLAB function that has a matrix \( A \) as an argument and replaces its minimal element in each position of the matrix.

10. Write an MATLAB function that generates a random \( m \times n \) matrix with integer entries distributed uniformly between \( a \) and \( b \).

11. Write an MATLAB function that will start with a vector of complex numbers and return a vector having its components in the set \( \{1, 2, 3, 4\} \) corresponding to the orthants where the image of each complex number is placed.

12. Write an MATLAB function that will start with a vector of complex numbers and circularly shift its components one position to the left or to the right, as specified by a parameter of the function.

13. Write an MATLAB function that accepts four square matrices \( A_1, A_2, A_3 \) and \( A_4 \) in \( \mathbb{R}^{n \times n} \) and outputs the minimum and the maximum trace of a product \( A_{i_1}A_{i_2}A_{i_3}A_{i_4} \), where

\[
\left( \begin{array}{cccc}
1 & 2 & 3 & 4 \\
i_1 & i_2 & i_3 & i_4
\end{array} \right)
\]

is a permutation.

14. Write an MATLAB function that starts with two vectors that represent the sequences of coefficients of two polynomials and generates the sequence of coefficients of the product of these polynomials.

15. The Lagrange interpolating polynomial is a polynomial of degree no larger than \( n - 1 \) whose graph passes through the points \((x_1, y_1), \ldots, (x_n, y_n)\) in \( \mathbb{R}^2 \) and is given by

\[
p(x) = \sum_{i=1}^{n} y_i \prod_{1 \leq j \leq n \text{ and } j \neq i} \frac{(x - x_j)}{(x_i - x_j)}.
\]

Given two vectors \( x, y \in \mathbb{R}^n \) write an MATLAB function that computes the coefficients of the Lagrange interpolating polynomial.

### Bibliographical Comments

MATLAB popularity in the technical and research communities has generated a substantial literature. We mention, as especially useful, such titles as [4, 2, 1, 5] and [3]. Space limitations did not allow us to present the outstanding visualization capabilities of MATLAB for scientific data. The reader should consult [2].

### References


