Strings and I/O functions

1. Introduction

It is the purpose of these online documents to provide information on Matlab and its uses in computer vision. The treatment is not intended to be as comprehensive as a book on Matlab would be. Instead it is assumed that the reader is already well acquainted with computer programming in other languages—e.g., conventional computer languages such as C++, Pascal or Java—so that what is required is clear presentation of examples of Matlab programming, together with cogent definitions, explanations, reminders and summaries of the nature of Matlab.

In what follows we start by taking the subject of strings and summarise how they are expressed, processed, used and converted in Matlab.

2. Strings

Strings are sequences of characters, where characters may be letters, numeric digits, punctuation marks, white spaces (including spaces, tabs and newlines) and control characters (including backspaces and tabs). Strings may have leading or trailing spaces, and some functions deal with these in special ways.

Strings may be regarded as vectors of characters, and as such they have definable lengths and may be transposed from the usual row to column format—or vice versa. As in the case of vectors, strings may be concatenated as follows:

```
>> string1 = 'good';
>> string2 = 'morning';
>> string3 = [ string1 string2 ]
string3 =
goodmorning
```

In fact, Matlab also has a function called `strcat`, which acts as follows:

```
>> string4 = strcat(string1, string2)
string4 =
goodmorning
```

Note that, in this example, each of the methods has the same effect—even to the extent of not including a space between the original two words. However, the `strcat` function would eliminate trailing spaces from the first string, though this is not evident in the above illustration.

When a set of strings is used to form a matrix of letters, enough spaces must be added to individual strings to make them equal in length, so that the matrix has a valid format. For example, in order to avoid error messages, we have to define:
>> word1 = 'morning__';
>> word2 = 'afternoon';
>> word3 = 'evening__';
>> word4 = 'night____';
>> % (for clarity, underlines are used to indicate spaces)
>> daymatrix = [ word1; word2; word3; word4 ]
daymatrix =
    morning
    afternoon
    evening
    night
>> size(daymatrix)
ans =
   4    9

However, if the char function is used to concatenate vertically, there is no need to pad the strings with trailing spaces, as this is done automatically by the function:

>> word1 = 'morning';
>> word2 = 'afternoon';
>> word3 = 'evening';
>> word4 = 'night';
>> daymatrix = char(word1, word2, word3, word4)
daymatrix =
    morning
    afternoon
    evening
    night
>> % (for clarity, underlines are again used to indicate spaces)

As spaces are crucial to the layout of language and text, and are also important when numerical quantities are included with text, a special function called blank is available for adding any number of blank spaces. For example, to add 4 spaces after 'night', as above, we can write: 

    nightword = [ 'night' blank(4) ];

    blank function also allows strings to be defined more adaptively—e.g., 
    [ 'night' blank(n) ]. Once spaces have been added by any of these methods, the total number of spaces may not be known, though it is easy to find out by applying the length function:

>> word4 = 'night
ans =
   9

Alternatively, the string search function strfind will reveal the actual number of spaces:

>> word4 = 'night

As spaces are crucial to the layout of language and text, and are also important when numerical quantities are included with text, a special function called blank is available for adding any number of blank spaces. For example, to add 4 spaces after 'night', as above, we can write: 

    nightword = [ 'night' blank(4) ];

    blank function also allows strings to be defined more adaptively—e.g., 
    [ 'night' blank(n) ]. Once spaces have been added by any of these methods, the total number of spaces may not be known, though it is easy to find out by applying the length function:

>> word4 = 'night
ans =
   9

Alternatively, the string search function strfind will reveal the actual number of spaces:
>> strfind(word4, ' ')  
ans =  
4

The strfind string search function is neatly complemented by the **strrep** string replace function, which will now be used to make spaces visible as box characters:

>> word4box = strrep(word4, ' ', '
')  
word4box =  
night

Next, we consider four functions that can be used to compare pairs of strings: they all return logical true if the strings are identical (including their lengths) and logical false otherwise. The four functions are: **strcmp**, **strcmpi**, **strncmp**, **strncmpi**. The two functions that end in 'i' are insensitive to the cases of the characters, whereas the two functions that contain the letter 'n' only examine the first $n$ characters of each string, where $n$ is the third argument in the calling function. Three examples are:

>> strcmp('cat', 'cats')  
an = 0
>> strncmp('night', 'night   ', 5)  
an = 1
>> strncmpi('Henry', 'henry', 1)  
an = 0

Crucial to comparing strings is the ability to adjust the number of white space characters (including spaces, tabs and newlines). In particular, function **deblank** has the property of removing spaces from the end of a string, whereas function **strtrim** is able to remove leading and trailing spaces, but not those in the middle of a string:

>> deblank('    night and day    ')  
an = night and day
>> strtrim('    night and day    ')  
an = night and day
>> strrep('    night and day    ', ' ', '')  
an = night and day

Note that using **strrep** is able to eliminate spaces throughout the string.

We have already met the **disp**, **fprintf** and **input** functions, all of which use strings. Of these, the **disp** function is used to print a line of text on the screen, e.g.:
The `fprintf` function is used to print formatted text and variables on the screen, e.g.:

```matlab
>> vlength = 6;
>> fprintf('The vector length is %d.\n', vlength)
The vector length is 6.
```

The `input` function is used to prompt for values of input variables, e.g.:

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

We now introduce the `sprintf` function which can be used to generate strings. The value of `sprintf` is that it permits programmable numbers to be included in strings:

```matlab
>> radius = 2.1;
>> perimeter = 2*pi*radius;
>> perimstr = sprintf('the perimeter of a circle of radius %.2f is %.2f', radius, perimeter)
perimstr =
    the perimeter of a circle of radius 2.10 is 13.19
```

In this example, `ellipsis` (...) is used carefully to avoid breaking strings. However, if strings have to be broken in the middle, they need to be reconnected using concatenation, e.g.:

```matlab
>> perimstr = sprintf(['the perimeter of a circle of ', ...
    'radius %.2f is %.2f'], radius, perimeter)
perimstr =
    the perimeter of a circle of radius 2.10 is 13.19
```

`sprintf`, `fprintf` and several other input–output commands use so-called placeholders to provide necessary formatting information:

- `%d` – indicates a decimal number
- `%f` – indicates a floating point number
- `%c` – indicates a single character
- `%s` – indicates a string.

In the first three cases, a number can be inserted immediately after the `%` to specify the field width, which is the total number of characters to be displayed: in the case of `%f`, the field width includes the decimal point. In addition, the number of places of decimal (the precision field) can be specified after the decimal point. Examples are:
%5d – means that 5 digits will appear, including leading blanks
%5f – means that 4 digits will appear, including the decimal point
%5.2f – means that 2 digits will appear before the decimal point and 2 after it
%.2f – means that 2 digits will appear after the decimal point
%3c – means that 3 characters will appear
%10s – means that, if necessary, the string must be padded on the left with spaces to bring the field width to at least 10.

If the field width is larger than necessary, leading blanks and/or trailing zeros will be printed.

Finally, to include a single quote character in a string, use two adjacent single quote marks; to include a % character in a string, use two adjacent single % signs; and to include a newline character in a string, use the characters '\n'—a notation that may be repeated several times if necessary. Though a newline will typically be inserted at the end of a string, it can also be inserted near the beginning.

The input function can be used to input numbers or strings, as will be seen from the following examples:

```matlab
>> days = input('How many days are there in a week? ')
How many days are there in a week? 7
days =
    7
>> day = input('What day is it today? ', 's')
What day is it today? Monday
day =
    Monday
```

The difference between the two cases arises because the default input is a number: if a string input is required, this must be flagged by including 's' (for string) as a second argument in the input function.

### 3. String conversion

Conversion functions provide a vital means for obtaining and using strings. In particular, we can convert ASCII character codes to single characters and vice versa, using the functions `int32` and `char`:

```matlab
>> int32('d')
an =
    100
>> char(100)
an =
    d
```

To convert strings containing several characters, there are two valuable functions `num2str` and `str2num`, which act in the following ways:
>> num2str(3.1415926535)
ans =
 3.1415
>> % (4 digits is the default number of decimal places)
>> num2str(3.1415926535, '%8.6f')
ans =
 3.141592
>> str2num('3.141592')
ans =
 3.1415
>> str2num('1 2 3 5 7 11 13')
ans =
 1 2 3 5 7 11 13

In this last example, a string of numbers separated by spaces is converted into a sequence of numbers. (Note that it is not obvious from the last line whether it represents a string of characters or the corresponding sequence of numbers, though in this case it represents the latter.)

We next consider various queries that may be made to determine the properties of strings. The relevant functions are: isletter, isspace, ischar, isstrprop (each of these is best imagined as being part of a question!). All four functions give rise to single or multiple logical true or false output arguments. Typical results are shown below for the case of a UK-style postcode:

>> isletter('ZY21 5BU') % letter
ans =
 1 1 0 0 0 0 1 1
>> isspace('ZY21 5BU') % white space character
ans =
 0 0 0 0 1 0 0 0
>> ischar('ZY21 5BU') % means a vector of characters
ans =
 1
>> isstrprop('ZY21 5BU', 'alpha') % letter
ans =
 1 1 0 0 0 0 1 1
>> isstrprop('ZY21 5BU', 'alphanum') % letter or numeric digit
ans =
 1 1 1 1 0 1 1 1
>> isstrprop('ZY21 5BU', 'digit') % numeric digit
ans =
 0 0 1 1 0 1 0 0
>> isstrprop('ZY21 5BU', 'upper') % uppercase
ans =
 1 1 0 0 0 0 1 1
>> isstrprop('ZY21 5BU', 'lower') % lowercase
ans =
 0 0 0 0 0 0 0 0
Finally, strings can be converted to all uppercase or all lowercase using the functions `upper` and `lower`:

```matlab
>> lower('ZY21 5BU')
an =
    zy21 5bu
>> upper('Henry')
an =
    HENRY
>> lower('Henry')
an =
    henry
```

4. **More on Input–Output operations**

Matlab has the useful facility that it can record the variables and data of a complete workspace to a single file, from which it can be fully recovered at the start of a new session. The appropriate commands are `save` and `load`, examples being:

```matlab
>> save workspace;
>> load workspace;
```

where the file in which the data is stored will be workspace.mat: the extension .mat is reserved for this one purpose only. However, this will often be a somewhat clumsy approach, and it will be better to record only relevant files. To save individual variables and matrices, we can use commands of the following type:

```matlab
>> save variable filename.dat -ascii;
```

where an ascii-readable data-file `filename.dat` is being written to. We can access this data-file and **type** it or **load** it using the following commands:

```matlab
>> type filename.dat;
>> load filename.dat;
```

Interestingly, `save` automatically starts by opening a new file, and if one of the same name is already present, it overwrites it. If `save` is to be used to extend an existing file, the **append** qualifier is needed:

```matlab
>> save variable filename.dat -ascii -append
```

Note that .mat is the default data-file extension. Note also that .mat files store not only the data values but also the variable names: .dat files store only the data values.

Considerably more flexibility is available if the `fprintf` function is used to save files and they are subsequently read by the `fgetl`, `fgets` or `fscanf` functions. The reason is that these files can be used in conjunction with suitable formatting commands to extract numerical and textual data from the input file: the textual data may include variable names or other letters or characters—typically those that indicate the meaning
of the data. Functions \texttt{fgetl} and \texttt{fgets} each read a string of data from the given file, one line at a time—up to the next newline character: note that \texttt{fgetl} eliminates the newline character, whereas \texttt{fgets} retains it. Clearly, these two functions have to operate in a loop if they are to access all the data in a file. On the other hand, \texttt{fscanf} is able to read the whole of a file in a single operation—if it is provided with a suitable formatting command to show how the data in the file is arranged. There is also a function \texttt{textscan} which operates similarly, albeit viewing the input file one line at a time.

For any of these output and input functions to work, the file needs to be opened using its filename, together with an append qualifier, which may be ‘r’, ‘w’ or ‘a’—which respectively mean read, write and append. To use \texttt{fprintf} we therefore have to specify ‘w’ or ‘a’ as appropriate:

```matlab
finfo = fopen('filename', 'w');
if finfo == -1
    disp('file not found')
else
    fprintf(finfo, 'the output data is %0.2f\n', v1);
    closeinfo = fclose(finfo);
    if closeinfo == 0
        disp('file closed')
    else
        disp('file not found')
    end
end
```

The above code starts by opening the file to be written, then writes to it, and finally closes it. As indicated above, best programming practice requires tests as to whether the file has been opened correctly, whether, on trying to close it, it actually exists, and whether it has been closed correctly.

To read from the same file using \texttt{fgetl}, we again have to open it, then read all the lines of data in the file, then close it. However, at this stage, it is not clear how many lines of data are in the file, and if there is no file header giving this information, we merely need to read one line at a time until the end of the file is reached. In what follows we shall ignore the need to check for opening and closing errors, and concentrate on reading the file one line at a time:

```matlab
finfo = fopen('filename', 'r');
% check for file opening errors
while ~feof(finfo)
    fgetl(finfo, nextline);
    % nextline will have to be dissected using
    % a function such as strtok (see below)
    % and the result acted upon
end
closeinfo = fclose(finfo);
% check for file closing errors
```

It was remarked earlier that `fscanf` is able to read the whole of a file in a single operation, rather than a line at a time. This is only true if the data can be fed to a suitably defined matrix that can properly represent the input data. However, note that sometimes the amount of data is unpredictable, and rather than reading the data a line at a time until the end of file is reached, `fscanf` reads the data into a matrix with one dimension being left free: in that case the last column of the matrix is written as `inf`: this acts as a dummy variable which can in principle adjust itself to a value as high as infinity.

A fairly basic instruction using `fscanf` to read a digit followed by two floating point numbers is:

```matlab
finfo = fopen('filename', 'r');
outputmat = fscanf(finfo, '%d%f%f
', [3, inf]);
fclose(finfo);
finalmat = outputmat';
```

Here the number of rows in the output matrix has to be three, to match the number of numbers per line in the input data; on the other hand, the number of input lines is unknown, and so the second dimension of the matrix is marked as `inf`. Note that this assumes that the numerical data in the source file will be in the form of a matrix of \( n \) rows and 3 columns. This means that if the final matrix is to appear in exactly the same form as in the data-file, after being read in by the `fscanf` function, it will have to be transposed. It is necessary for the read operation to specify the output vector as `\[3, inf\]` rather than `\[inf, 3\]`, because, by definition, `fscanf` reads data one line at a time into a column in the input matrix.

Note that `fscanf` can also read characters using the format specifier `%c`. This will look for non-numerical characters, such as letters of the alphabet, or spaces or even newline characters. If the characters that are read by `fscanf` are placed in a numeric matrix (such as `outputmat` in the above example), they will be converted to their ASCII codes, as all the elements of a matrix have to be of the same type.

Finally, note that `%f` does not support a precision field for reading: but `%f`, `%d` and `%c` all support field widths for reading.

### 5. The `strtok` string parsing function

It is often necessary to be able to parse strings of characters into shorter, meaningful sequences—as we do when we are reading text. For example, we may wish to parse the sentence "The little dog laughed!", in preparation for interpreting each word as a noun or other part of speech, and enabling us to work out the meaning. Fortunately, Matlab has a function called `strtok` which is able to parse strings of characters. At each stage, it achieves this by looking for the next delimiter—typically a space—and dividing the string into a head and a tail. Repeated applications of this technique are required to interpret a whole line of text, as will be seen from the following example:
Notice that at each stage the delimiter is attached to the following tail. Curiously, however, any delimiter that occurs at the beginning of the line is eliminated by the \texttt{strtok} function. Technically, the 'head' is known in Matlab as the \texttt{token}, and always stops immediately before the delimiter.

Any set of characters can be used as a delimiter, which is specified as the second argument in the \texttt{strtok} brackets. We illustrate this in a slightly more complicated case:

```matlab
>> lineA = 'Good morning, children!';
>> [ word1, tail1 ] = strtok(lineA)
  word1 =
    Good
  tail1 =
    morning, children!
>> [ word2, tail2 ] = strtok(tail1)
  word2 =
    morning,
  tail2 =
    children!
```

```matlab
>> lineB = 'Today is Thursday 16th November, 2017.';
>> [ word1, tail1 ] = strtok(lineB);
  word1 =
    Today
>> [ word2, tail2 ] = strtok(tail1);
  word2 =
    is
>> [ word3, tail3 ] = strtok(tail2);
  word3 =
    Thursday
>> [ word4, tail4 ] = strtok(tail3, 't');
  word4 =
    16
>> [ word5, tail5 ] = strtok(tail4);
  word5 =
    th
>> [ word6, tail6 ] = strtok(tail5);
  word6 =
    November,
>> [ word7, tail7 ] = strtok(tail6, '.');
  word7 =
    2017
>> tail7 =
  .
>> daynum = str2num(word4)
  daynum =
    16
>> yearnum = str2num(word7)
  yearnum =
    2017
```
6. Index

alpha, 6
alphanum, 6
append, 7
ascii, 7
blank, 2
char, 2, 5
concatenation, 1
deblank, 3
delimiter, 9
digit, 6
disp, 3, 8
ellipsis, 4
close, 8
feof, 8
fgetl, 7, 8
fgets, 7, 8
field width, 4, 9
fopen, 9
fprintf, 3, 4, 7, 8
fscanf, 7, 8, 9
fscanf, 7, 8, 9
inf, 9
input, 3
int32, 5
ischar, 6
isletter, 6
isspace, 6
isstrprop, 6
load, 7
lower, 6, 7
num2str, 5
parsing, 9
placeholder, 4
precision field, 4, 9
save, 7
sprintf, 4
str2num, 5, 10
strcat, 1
strcmpr, 3
strcmpr, 3
strrep, 3
strtok, 8, 9, 10
strtrim, 8
textscan, 8
token, 10
type, 7
upper, 6, 7
white space, 1