Lecture 5: Strings, Printing, and File I/O

1 Learning objectives

At the end of this class you should be able to...

- be able to name the three MATLAB data types most commonly used in ME 352
- be able to use the `fprintf` statement to print strings and values stored in MATLAB variables.

2 Reading

pp. 48–51, pp. 100–105

3 MATLAB Data Types

All MATLAB variables are matrices. In its simplest form, a matrix is a rectangular, two-dimensional array of values. MATLAB also supports n-dimensional matrices, but we won't need to use them in ME 352.

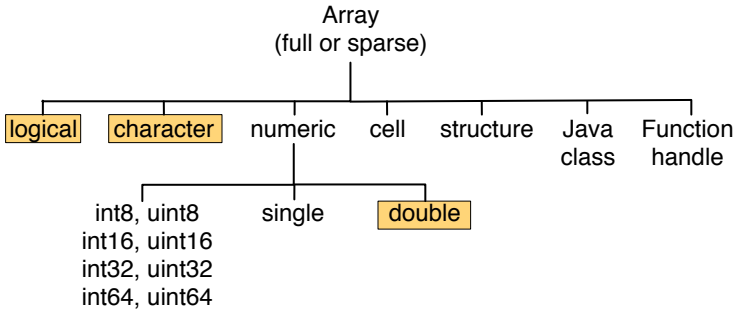

Figure 1: MATLAB data types. The `double`, `logical`, and `character` types are used in ME 352.

In MATLAB the elements of a matrix can have any of the data types represented schematically in Figure 3. A data type is a definition for how a value is stored in computer memory. A data type may also have specially defined mathematical operations associated with it. For example, matrix multiplication obeys the rules of linear algebra, not ordinary (scalar) algebra.

In early versions of MATLAB the only data types were strings and double precision numeric values.

In ME 352 we are almost always using numeric matrices that store double precision numbers. Occasionally we will use matrices with logical values (either true or false) or matrices that store arrays of alphanumeric characters (or strings).

3.1 Matrices

3.1.1 Numeric Matrices

For most computing tasks in MATLAB, we consider all variables to be numeric matrices, i.e., rectangular arrays of numbers.

3.1.2 Creating and Manipulating Matrices

- Manual
- Operations that return matrices
- subscript notation
- Colon (wildcard) notation.

3.1.3 String Matrices

To label data or manipulate files, we also need to work with strings. A string is a group of characters. In MATLAB strings are just matrices of characters

4 fprintf

The syntax of `frprintf` is based on the `fprintf` function from the C language.

```
fprintf(format)
fprintf(format,variables)
fprintf(fid,format,variables)
```

where *`format`* is a text string that controls the appearance of the output, *`variables`* is a comma-separated list of variables to be displayed according to the specification in *`format`*, and *`fid`* is a *file identifier* for the file to which the output is sent. The value of *`fid`* is only set when output is sent to a file.

Table 1: Summary of format codes for use with `fprintf` and `fscanf`.

Code	Conversion instruction
`%s`	format as a string
`%d`	format with no fractional part (integer format)
`%f`	format as a floating-point value
`%e`	format as a floating-point value in scientific notation
`%g`	format in the most compact form of either `%f` or `%e`
`\n`	insert newline in output string
`\t`	insert tab in output string

Controlling Field Width and Precision

The `%d` and `%s` format specifiers can be modified to control the field width used to display a value. This is best explained via example. Suppose that the `k` and `s` variables are assigned values

```
>> k = 2^7;   s = 'big red barn';
```

Without specifying a field width these fields are displayed with the minimum field width, i.e.,, the minimum number of characters necessary to display the contents of the variables.

```
>> fprintf('%d\n',k)
128

>> fprintf('%s\n',s)
big red barn
```

The field width specifier is an integer that precedes the `d` in `%d` or the `s` in `%s`. For example, to print the value of `k` width of 6 characters use the `%6d` format specifier.

```
>> fprintf('%6d\n123456789\n',k)
   128
123456789
```

The second line of output makes it easy to see that the value of 128 is on the right-hand end of a field that is six characters wide. Similarly

```
>> fprintf('%16s\n12345678901234567890\n',s)
    big red barn
12345678901234567890
```

Examples

Define some data and print it in the simple-minded way

```
n = 5;  x = linspace(-pi,pi,n);  y = x/1e6;  z = x*1e6;

fprintf('\nSimple fprintf statment\n');
for j=1:length(x)
  fprintf('%d %f %f %f\n',j,x(j),y(j),z(j));
end
```

Produces the following output

```
1 -3.141593 -0.000003 -3141592.653590
2 -1.570796 -0.000002 -1570796.326795
3 0.000000 0.000000 0.000000
4 1.570796 0.000002 1570796.326795
5 3.141593 0.000003 3141592.653590
```

The following sequence of modifications to the format string provide increasing levels of control over the column appearance.

Specify the Precision of the Floating Point Fields

```
fprintf('\nEqual precision control for all float types\n');
for j=1:length(x)
  fprintf('%d %.2f %.2f %.2f\n',j,x(j),y(j),z(j));
end

Equal precision control for all float types
1 -3.14 -0.00 -3141592.65
2 -1.57 -0.00 -1570796.33
3 0.00 0.00 0.00
4 1.57 0.00 1570796.33
5 3.14 0.00 3141592.65
```

Use Column Width and Precision for All Floating Point Fields

```
fprintf('\nField width and precision control for all float types\n');
for j=1:length(x)
  fprintf('%d %8.2f %11.8f %11.0f\n',j,x(j),y(j),z(j));
end

Field width and precision control for all float types
1  -3.1416 -0.00000314    -3141593
2  -1.5708 -0.00000157    -1570796
3   0.0000  0.00000000           0
4   1.5708  0.00000157     1570796
5   3.1416  0.00000314     3141593
```

Use Column Width, Precision Specification, and Scientific Notation

```
fprintf('\nField width, precision control and scientific notation\n');
for j=1:length(x)
  fprintf('%d %8.2f %13.3e %13.3e\n',j,x(j),y(j),z(j));
end
```

```
Field width, precision control and scientific notation
1  -3.1416    -3.142e-06    -3.142e+06
2  -1.5708    -1.571e-06    -1.571e+06
3   0.0000     0.000e+00     0.000e+00
4   1.5708     1.571e-06     1.571e+06
5   3.1416     3.142e-06     3.142e+06
```

Add Column Headers and Field Width for Index Column

```
fprintf('\nFinal touches:\n');
fprintf('\n  j     x(j)       y(j)          z(j)\n');
for j=1:length(x)
  fprintf('%3d %8.2f %13.3e %13.3e\n',j,x(j),y(j),z(j));
end
```

```
Final touches:

  j     x(j)       y(j)          z(j)
  1  -3.1416    -3.142e-06    -3.142e+06
  2  -1.5708    -1.571e-06    -1.571e+06
  3   0.0000     0.000e+00     0.000e+00
  4   1.5708     1.571e-06     1.571e+06
  5   3.1416     3.142e-06     3.142e+06
```

File Output with Optional Use of Tabs to Separate Columns

```
fprintf('\nWrite data to a file, no screen output:\n');
fout = fopen('someData.txt','wt');
fprintf(fout,'j\tx(j)\ty(j)\tz(j)\n');
for j=1:length(x)
  fprintf(fout,'%d\t%e\t%e\t%e\n',j,x(j),y(j),z(j));
end
fclose(fout);  % always close the open file handle
```

```
function demofprintf
% demofprintf  Examples of using the fprintf statement

% --- Define data
n = 5;  x = linspace(-pi,pi,n);  y = x/1e6;  z = x*1e6;

% -- Simplest print statements, no column control
printFormat1(x,y,z);

% -- Print with equal precision control for all float types
printFormat2(x,y,z);

% -- Print with column width and precision control for all float types
printFormat3(x,y,z);

% -- Print with column width, precision control and scientific notation
printFormat4(x,y,z);

% -- Final touches.  Add column headers and field width for index column
printFormat5(x,y,z);

% -- File output:  optional use of tabs to separate columns
printFormat6(x,y,z);
```

```
function printFormat1(x,y,z)

% -- Simplest print statements, no column control
fprintf('\nSimple fprintf statment\n');
for j=1:length(x)
  fprintf('%d %f %f %f\n',j,x(j),y(j),z(j));
end
```

```
function printFormat2(x,y,z)

% -- Print with equal precision control for all float types
fprintf('\nEqual precision control for all float types\n');
for j=1:length(x)
  fprintf('%d %.2f %.2f %.2f\n',j,x(j),y(j),z(j));
end
```

```
function printFormat3(x,y,z)

% -- Print with column width and precision control for all float types
fprintf('\nField width and precision control for all float types\n');
for j=1:length(x)
  fprintf('%d %8.4f %11.8f %11.0f\n',j,x(j),y(j),z(j));
end
```

```
function printFormat4(x,y,z)

% -- Print with column width, precision control and scientific notation
fprintf('\nField width, precision control and scientific notation\n');
for j=1:length(x)
  fprintf('%d %8.4f %13.3e %13.3e\n',j,x(j),y(j),z(j));
end
```

```
function printFormat5(x,y,z)

% -- Print with column headers and field width for index column
fprintf('\nFinal touches:\n');
fprintf('\n  j    x(j)        y(j)          z(j)\n');
for j=1:length(x)
  fprintf('%3d %8.4f %13.3e %13.3e\n',j,x(j),y(j),z(j));
end
```

```
function printFormat6(x,y,z)

% -- File output:  optional use of tabs to separate columns
fprintf('\nWrite data to a file, no screen output:\n');
fout = fopen('someData.txt','wt');
fprintf(fout,'j\tx(j)\ty(j)\tz(j)\n');
for j=1:length(x)
  fprintf(fout,'%d\t%e\t%e\t%e\n',j,x(j),y(j),z(j));
end
fclose(fout);  % always close the open file handle
```

5 File I/O Overview

5.1 Core Functions for Printing and File I/O

`disp`	Simplest way to display information to the command window. Direct control over appearance of output is limited.
`fprintf`	Flexible control over appearance of printed output. Syntax of specifying appearance is similar to the `fprintf` function from the C language. `fprintf` is used to write to the command window, or write to files.
`fopen`	open a file to prepare for writing to it.
`fclose`	close a file after it is no longer needed.
`sprintf`	Create strings using the format of `fprintf`. Useful for building file names with embedded parameters, e.g. `file1.dat`, `file2.dat`, `file3.dat`.
`load`	read data from a file. `load` can read `MAT` files – a binary file format used by MATLAB. `load` can only read data from text files when the data forms a rectangular array with equal number of elements in each row – i.e., no missing values.
`loadColData`	NMM Toolbox routine
`textread`	Very flexible tool for reading text files with any combination of variables on each line.

5.2 Other I/O Functions for Specialized Tasks

`csvread`	Read a text file containing values are separated by commas. CSV is an abbreviation of "comma-separated values".
`csvwrite`	Write a text file so that values are separated by commas.
`xlsread`	Read data from an Excel Spreadsheet.
`sscanf`	Read a string (not a file) using format specification to interpret characters in the string
`fileparts`	Split a string representing a file path into parts for the path, base file name, and file extension
`fullfile`	Construct a full file path from the parts of the path. `fullfile` is the inverse of `fileparts`.