Lecture 4

Input and output

1 Introduction

The “deliverable” of a computer program is its output. Output may be in graphical form as in a two-dimensional function plot, or it may be in text form as in a table of data values. In addition we often need to provide a program with input data, either interactively from the console or from a disk file. We will cover graphics in future lectures. Here we look at various ways to input and output data to and from the console and disk files.

2 Basic input/output (I/O)

Scilab/Matlab automatically displays the value of a variable when you type its name at the command line. For example

```
-->x = 2.3;

-->x
 x  =
    2.3
```

In a program, on the other hand, the appearance of a variable name does not produce printed output. Moreover, even with console I/O we may want more control over the output format. We also need a way for a program to display values and ask for input, either from the user or from a file. There are various ways to approach this. We start with the simplest.

2.1 The disp function

Most basic console I/O can be implemented with two functions: `disp` and `input.` The `disp` function can be used to display the values of variables without the extra "ans = " prefix. For example

```
-->[x,y]
 ans  =
    2.3    4.1

-->disp([x,y])
    2.3    4.1
```

This works from within a program also. You can also combine it with the `string()` function (`num2str()` in Matlab) and the string concatenate operation. This creates one long string as in the following

```
-->disp(string(x)+' plus '+string(y)+' equals '+string(x+y));

 2.3 plus 4.1 equals 6.4
```

This is generally good enough for most basic program output.

Exercise 1:

2.2 *The input function*

The `input()` function can be used to prompt the user for input. We will illustrate this with a few examples

```
-->x = input('please enter the value of x :');
please enter the value of x :3.2

-->disp(x)
    3.2
```

You can input arrays

```
-->y = input('enter a 1-by-3 vector : ')
enter a 1-by-3 vector : [1,2,3]
 y  =
    1.    2.    3.
```

and strings

```
-->fname = input('output file : ')
output file : 'test.txt'
 fname  =
    test.txt
```

Here's a little snippet of code that prompts the user for an array of data and prints the average value.

```
z = input('enter a 1-by-n array of numbers : ');
disp('the average value is '+string(mean(z)));
```

this produces

```
enter a 1-by-n array of numbers : [1,2,3,4,5,6]
the average value is 3.5
```

Exercise 2:

2.3 *The save and load functions*

The save and load functions allow you to “dump” and “recover” any or all variables you have created.

```
-->A = [1,2;3,4];
-->x = 1:5;
-->s = 'hello there';
-->save('test.dat');
```

The save command saves all variable names and values, essentially your entire Scilab session. You can exit Scilab and in a later session use the load command to recove these saved values.

```
-->load('test.dat')

-->A
 A  =
    1.    2.
```

```
      3.    4.

-->x
 x  =
      1.    2.    3.    4.    5.


-->s
 s  =
 hello there
```

This works the same in Matlab, except the file name should not have an extension (for example 'test') as Matlab appends the .mat extension automatically. You can explicitly specify the variables you want to save/load as in

```
-->save('Ax.dat','A','x'); //Scilab
>> save('Ax','A','x'); %Matlab
```

This saves only the variables A and x. To load one or more specific variables you do the following

```
-->load('Ax.dat','A'); //Scilab
>> load('Ax','A'); %Matlab
```

Exercise 3:

3 Formatted I/O

Scilab/Matlab implement versions of the C fprintf and sprintf functions for formatted output.

3.1 mfprintf (Scilab) & fprintf (Matlab)

The C fprintf function ("file print formatted") allows you great flexibility for generating formatted output to the console or to a file. Scilab implements this as `mfprintf` while Matlab uses `fprintf`. We will illustrate the `mfprintf` function in Scilab. For Matlab, simply leave off the initial `m`.

The calling sequence for `mfprintf` is

```
mfprintf(fd,format,var_1,var_2,...,var_n);
```

`fd` is a *file descriptor* and `format` is a string that specifies how you want the output formatted. `var_1` through `var_n` are the variables you want displayed. In Scilab the number 6 is the file descriptor for the console. This is also stored in the protected variable `%io(2)`.

An example is

```
-->fd = %io(2);
-->x = 1.23;
-->mfprintf(fd,'the value of x is %f\n',x);
the value of x is 1.230000
```

In Matlab the screen corresponds to `fd = 1`.

In the format string `%f` stands for a floating-point number, `%e` stands for an exponential-format number (scientific notation), `%d` for a decimal number (integer) and `%s` for a string. In the output these are replaced by the variable values. Here's another example

```
x = 1.23;
y = 4;
z = %pi*1e10;
str = 'testing one two three';
mfprintf(fd,'%f %d %e %s\n',x,y,z,str);
1.230000 4 3.141593e+010 testing one two three
```

The `\n` symbol denotes a "new line." If you omit this then subsequent `mfprintf` commands will be appended to the same line. Consider the following.

```
mfprintf(fd,'the value of x is %f',x);
mfprintf(fd,' and y is %f\n',y);
```

produces

```
the value of x is 1.230000 and y is 3.210000
```

Finally, if you want to control the precise format of the numerical output, the syntax for a floating point number is `%m.nf` where m is the total number of spaces (you need one for the decimal point and you might need one for the sign) and n is the number of decimal places. For example

```
-->mfprintf(fd,'%4.2f\n',x);
1.23

-->mfprintf(fd,'%6.2f\n',x);
  1.23

-->mfprintf(fd,'%6.3f\n',x);
 1.230
```

For `%d` and `%s` formats you can use the syntax `%md` or `%ms` where m is the total number of spaces to be displayed. Note that if you don't allocate enough, the full value will be printed anyway. If you allocate "too much" then blank space will be added. In the following example we generate a formatted table of trig values.

```
N = 4;
x = linspace(0,%pi/2,N);
y = sin(x);
z = cos(x);
mfprintf(fd,'\n'); //creates blank line at start
mfprintf(fd, '%6s %6s %6s\n','x','sin','cos');
for i=1:N
  mfprintf(fd,'%6.3f %6.3f %6.3f\n',x(i),y(i),z(i));
end
```

Note the `mfprintf(fd,'\n');` statement used to clear any previously "open" lines of output. The output is

```
     x    sin    cos
 0.000  0.000  1.000
 0.524  0.500  0.866
 1.047  0.866  0.500
 1.571  1.000  0.000
```

A couple more points. Consider the following.

```
-->mfprintf(fd,'%3d\n',k);
  5

-->mfprintf(fd,'%-3d\n',k);
5

-->mfprintf(fd,'%03d\n',k);
005
```

The format `%-3d` causes the output to be left aligned as opposed to the default right alignment. The format `%03d` causes the output to be right aligned but all remaining space to the left is filled with zeros.

In keeping with the "vectorized" nature of Scilab/Matlab, the `mfprintf` (and `fprintf`) function is also vectorized. For example

```
-->x = 1:3
 x  =
    1.    2.    3.
-->mfprintf(fd,'%d %d %d\n',x)
1 2 3
```

Scilab/Matlab recognizes that x is an array. It fills in the 3 `%d` formats with `x(1)`, `x(2)` and `x(3)`. Now consider this

```
A = [1,2;3,4];
-->mfprintf(fd,'%f %f\n',A)
1.000000 2.000000
3.000000 4.000000
```

`mfprintf` repeats itself for each row of matrix A. In addition to `mfprintf` there is a Scilab function `mprintf` that does not require the file descriptor argument and prints directly to the console.

```
-->x = 2;
-->mprintf('%f %f %f\n',x,x^2,x^3)
2.000000 4.000000 8.000000
```

The advantage of using `mfprintf` with `fd = %io(2)` for console output is that it is very simple to modify your code to output to a file. You merely need to assign the `fd` variable using the `mopen` command described below.

Exercise 4:

3.2 msprintf (Scilab) & sprintf (Matlab)

The `msprinf` function (`sprintf` in Matlab) is similar to the `mfprintf` function except that instead of writing output to the console or a file, it writes it to a string that can be assigned to a variable. Consider the following

```
-->k = 3;
-->name = msprintf('file%03d.txt',k)
 name  =
 file003.txt
```

This has created a string with the value of `k` embedded. An example where this is very useful is in creating frames for an animation where `k` goes from `1` to `N` and each file output is a single frame.

3.3 Opening and closing files

If we want output to go to a disk file instead of the console, we need to create and/or open a file, write to it, and close the file. Opening a file is done with the `mopen` command (`fopen` in Matlab).

```
[fd,err] = mopen('test.txt','wt');
```

This opens the file `test.txt` for output in the current directory. If it doesn't exist it is created. If it does exist is it overwritten. The `'wt'` notation indicates that we are opening this file for writing in text format. You can also write in binary format, but we won't cover that. `'at'` designates a text file opened for appending. To open a text file for reading we use `'rt'`.

```
[fd,err] = mopen('test.txt','rt');
```

The resulting file descriptor `fd` can be used to refer to the file and `err` is an error flag. It is zero or empty if the open process worked properly and non-zero otherwise. You will get an error if you try to open a file for reading that doesn't exist, or a file for writing in a directory where you don't have write permission. Good programming practice is to always include error checking. For example

```
[fd,err] = mopen('test.txt','rt');
if (err)
  error('cannot open test.txt');
end
```

If there is an error opening `'test.txt'` you will get a message and then Scilab will exit using the `error()` function. It is very important to always close a file when you have finished reading or writing. This is done with the `mclose(fd)` command (`fclose(fd)` in Matlab).

Here's an example

```
A = [1,2;3,4];
[fd,err] = mopen('A.txt','wt');
if (err)
  error('cannot open file');
end
mfprintf(fd,'%f %f\n',A);
mclose(fd);
```

This creates the file `A.txt` which contains the following.

```
1.000000 2.000000
3.000000 4.000000
```

A common source of errors when trying to open a file for reading is that the file does not exist. A way to test for this is using the `isfile()` command. In the following example a file named `'test.txt'` exists the current directory but a file named `'test2.txt'` does not.

```
-->isfile('test.txt')
```

```
 ans  =
  T

-->isfile('test2.txt')
 ans  =
  F
```

We might use this as follows

```
name = 'test.txt';
if (isfile(name))
  [fd,err] = mopen(name,'rt');
  if (err)
    mfprintf(%io(2),'cannot open file %s\n',name);
  else
    mfprintf(%io(2),'file %s is now open\n',name);
  end
else
  mfprintf(%io(2),'file %s does not exist\n',name);
end
```

This produces the output

```
file test.txt is now open
```

whereas changing the first line to

```
name = 'test2.txt';
```

gives us the message

```
file test2.txt does not exist
```

Exercise 5:

3.4 mfscanf (Scilab) and fscanf (Matlab)

The `mfscanf` function is a modification of the C `fscanf` function and allows you to perform formatted input from a file. Suppose we use a text editor to create a text file named data.txt that contains

```
1 2 3
4 5 6
```

We can read these numbers into a vector x as follows (note that for compactness we are not including error checking for the `mopen` command).

```
[fd,err] = mopen('data.txt','rt');
[n,x] = mfscanf(6,fd,'%d');
mclose(fd);
disp(x');
1.    2.    3.    4.    5.    6.
```

Variable `n` indicates the number of successful reads. It is -1 if the end of file was reached before all desired data were read. The 6 tells `mfscanf` to read six times in the `%d` format. These are assigned as the elements of x. Here a variation

```
[fd,err] = mopen('data.txt','rt');
[n,x] = mfscanf(3,fd,'%d');
[n,y] = mfscanf(3,fd,'%d');
mclose(fd);

-->x'
 ans  =    1.    2.    3.

-->y'
 ans  =    4.    5.    6.
```

This reads 3 numbers and assigns them to x, then another 3 numbers are read and assigned to y. In Matlab the ordering is slightly different

```
[fd,err] = fopen('data.txt','rt');
[n,x] = fscanf(fd,'%d',3);
[n,y] = fscanf(fd,'%d',3);
fclose(fd);
```

Here another example in Scilab. This time the file data.txt looks like this

```
east     2   23.75
west     4  -94.5
south    1    8.2
north    3   -7.9
```

The following commands

```
[fd,err] = mopen('data.txt','rt');
[n,s,d,x] = mfscanf(4,fd,'%s %d %f');
mclose(fd);
```

fill the arrays s, d and x with the corresponding string, decimal and floating point entries

```
-->s'
 ans  = !east  west  south  north  !

-->d'
 ans  =  2.    4.    1.    3.

-->x'
 ans  =  23.75  - 94.5    8.1999998  - 7.9000001
```

4 *meof (Scilab) and feof (Matlab)*

Suppose we have a file named `'test.txt'` which has the following contents

```
1
   2
3

4
```

What do we do if we know the file contains some numbers but we don't know how many? One way to read all the available numbers in the file is to read them one at a time followed by a check for an *end-of-file* condition using the `meof` function (`feof` in Matlab). This returns a non-zero value if the last input operation reached the end of the file. Here's an example of how we can use this in a program.

```
[fd,err] = mopen('test.txt','rt');
i = 1; //use i for an array index
while (~meof(fd)) //while we haven't reach the end of the file
  x(i) = mfscanf(fd,'%f'); //read the next number
  i = i+1; //increment the array index
end
mclose(fd);
disp(x');
```

This produces the output

```
1.    2.    3.    4.
```

We open the file and then as long as we have not reached the end-of-file condition we read a single number into an element of an array `x`, increment the array index `i` and try again. This reads in the values 1, 2, 3 and 4 then stops when the end of the file is reached. One thing to notice is that Scilab ignores the “white space” in the file (spaces, tabs and line-feed characters) and only looks for printable characters.

Exercise 6:

There are many other input/output functions. In the Scilab Help Browser see the sections titled

```
Files : Input/Output functions
Input/Output functions
```

5 Spreadsheet support

Scilab/Matlab can read and write data in spreadsheet format. We will only consider spreadsheets with numeric data and using comma-delimited text format (csv files). Suppose the spreadsheet ss.csv looks like

A	B	C	D
1	2	3	
4	5	6	

The actual file in a text editor looks like this

```
1,2,3
4,5,6
```

In Scilab we read this file using

```
-->M = csvRead('ss.csv')
 M  =

    1.    2.    3.
    4.    5.    6.
```

To write a csv file we use the command

```
csvWrite(M,'new.csv');
```

It is possible to specify a separator other than a comma and to read and write strings. See the Spreadsheet section of the help menu for more information.