1 Goals

- CODE: Output numbers and text using formatted printf
- CODE: Open and close files for writing
- Practice with logical indexing, loops

2 Background

If you have computed a number and want to display this on a plot you have to convert it into a string, i.e. we have to convert between the two forms:

```
x_number=4.367
x_string='4.367'
```

The first is a floating point number (suitable for plotting or mathematical computation), and the latter is a 5-char string array (suitable for display using `title` or `text` commands, or which can be written to a file). For quick tasks MATLAB includes commands `num2str`, `str2num`, `int2str` and some others which can convert from numbers (floating point or integer) to strings and vice versa but for best results one should learn how to use the `printf/scanf` family of functions.

Up to this point we've used `disp()` to display values to the screen. As you will see, `fprintf()` can be used for the same purpose, but it is more powerful because it is able to format the output and combine different types of variables in a nicer way. It can also be used to output (or "print") to a file instead of the command window. An important use of `fprintf` is in program debugging. Adding `fprintf` statements at crucial places in the program allows you to display the values of variables in a clean and elegant way instead of filling up the screen with numbers by simply removing the `;` from the end of a line.

We begin with `fprintf`, which outputs strings to the screen. `sprintf` works basically the same way but writes the string to a Matlab variable; `sscanf` and `fscanf` read from strings or files.

Here is a typical example:

```
>>fprintf('\nthis number %d and this one %.4f and these characters: %s\n\n',...
          4,3.6,'qwerty')

this number 4 and this one 3.6000 and these characters: qwerty
```

`fprintf` and `sprintf` take as arguments a string containing a mix of characters and format specifiers, followed by a series of values or variables (here their values are explicitly shown). Format specifiers begin with a '%', and end with a conversion character. There are lots of conversion characters but the ones you will use most are `d` for integers, `e`, `f`, and `g` for floating point numbers, and `s` for string data. Between the '%' and the conversion character there are various optional parameters, but most of the time you only need to use either a width (an integer giving the number of characters to use), and/or a precision after a decimal point. If you don't specify these then MATLAB will try to do something sensible.

The `\n` symbol represents a special 'non-printing' character - the CARRIAGE RETURN. It is non-printing because it isn't a letter or number, but it is used to say that the line has ended and the next characters will be on a new line.

Hint: Type `help fprintf` or `doc fprintf` (and read it!!).

3 The Lab

Create a script (called `lab09print.m`) and use it for parts 1–3 of this lab. Save your `lab09print.m` file as a future reference tool to look up how formatted output works and how to add labels to figures.

1. Try various formats. Enter into your `lab09print.m` script the lines

```
fprintf('1234567890\n');  % This will label the columns
fprintf('%8.0f\n',100*pi);
```

   and then make multiple copies of the second line,

   - first changing `0` in `8.0` to one of 1,3,10 (4 cases)
   - and then repeating these 4 cases with `f` changed to `e`.
   - then repeat the above, removing the '8' – e.g., `fprintf('%0.0f\n',100*pi);` (another 8 cases)

   Run the script. The number π will be printed out in 16 different ways, one way for each `fprintf` line. The way in which the number is printed (number of decimal places, number of digits in total, whether or not scientific notation is used) depends on the format specifiers. Look for at the output and check that you understand how the format specifiers work. You may want to use `clc` to clear the screen in between your different runs. Take some of the `\n` out and see why you need them here!

   Now try using `%g` instead of `%f`.

2. Check that you get the same result if you use the following more general syntax to print to the screen:

```
fid=1;  % 1 is the FID for "standard output"
fprintf(fid, '1234567890\n');  % This will label the columns
fprintf(fid, '%8.0f\n',100*pi);
```

   and check that setting `fid` to 2 also prints the output to the screen, but in red:

```
fid=2;  % 2 is the FID for "standard error"
fprintf(fid, '1234567890\n');  % This will label the columns
fprintf(fid, '%8.0f\n',100*pi);
```

   These are useful because you can always check the formatting of stuff you want to print to a file (*e.g.*,, later in this lab `fid` will be the file identifier of the file to which we will print) by setting `fid=1` or `fid=2` and printing your output to the screen instead first.

3. Format specifiers are vectorized which means that the format string will be used repeatedly if the variables following are arrays. Try this:

```
fprintf(1,'%10.2f\n',exp(1:10));
```

   and this:

```
fprintf(1,'%10.2f %10.2f\n',exp(1:10));
```

   and this:

```
fprintf(1,'%10.2f %10.2f\n', exp(1:10), sin(1:10))
```

4. Until now we have been writing things out to the screen. You can write things out to a string which can be displayed in a figure by using `sprintf`. For example:

```
n=3;
x=cumsum(randn(50,n));
plot(x);
ylim([-20 20]);
str=sprintf('we are showing %d lines',n);
text(1,10,str);
```

Try this code, and make sure you understand how it works. Type `help cumsum` to see what this function does. Look at `x` and `str` to see what values are they contain.

5. Download the file `commute.mat` from the class web site. This contains the commuting data you uploaded in week 2. It is in MATLAB binary format. Write a new script `lab09.m` and load the file. You should see 3 variables in your workspace: `km` which contains commute distance in km, `mins` which contains the corresponding commute time in mins, and `md` which is a character array containing the corresponding mode of commuting. The variable `md` uses the character B for bus, C for car, W for walk and R for bike.

   Plot time (y-axis) versus distance (x-axis) using the following symbols and colors for each mode of transportation: bus – blue squares, bike – magenta diamonds, walk – red circles, car – green triangles. Include legend, title, axis labels. Make sure your points are ONLY plotted as symbols — don't attempt to connect them with lines unless you want to look at a confusing mess!

   Hint: Use the array `md` to set up logical indexing to the arrays `km` and `mins` for different modes of transport.

   Now, for each mode of transportation, calculate the mean distance, the mean time, and the mean speed. The mean speed is the mean of all the individual speeds, NOT the mean distance divided by the mean time. Draw a thick line from `(0,0)` to `(mean distance, mean time)` for each mode and label the endpoints `speed = XX.X kmh`.

6. Finally, output these statistics to a file in a nice format. In order to write data to a file, we first have to open it. The function `fopen` opens a file and returns a file identifier, which can then be used as the first argument to `fprintf` to write to that file. After you've finished writing, you must close the file. For example:

```
fid=fopen('temp.txt','w');  % opens a file called 'temp.txt' for writing
fprintf(fid,'write this line\n');
fclose(fid); % close file after you are finished writing everything.
```

   We are now using the file identifier to "print" to the file specified by `fid`. Add to your script `lab09.m` some code that will use the data loaded from `commute.mat` and create a file called `commute.txt` that contains 5 columns as follows.

   - column 1 – transportation mode
   - column 2 – number of people using that mode
   - column 3 – average commute time using that mode
   - column 4 – average distance using that mode
   - column 5 – average speed using that mode

   The output should have 4 rows of information, one for each transportation mode, and should have column headings to indicate what is in that column and the units if any. The final output should be in a nice (i.e., pleasing to look at) human-readable format. The precision of real numbers should be 1 decimal place, and integers should be displayed as integers (i.e. the integer 5 should *not* be displayed as 5.0).

   Hints:

   - A correct choice of the field width is key to getting the columns aligned - make it large! Avoid using the `\t` tab spacing, since tabs are set at different widths on different editors.
   - Replacing fid for the output file with `fid=1` is helpful for testing your output by writing it to the screen instead of the file.

4 To Hand In

Submit via CANVAS s script `lab09.m` that

- loads `commmute.mat`
- produces the figure in item 5, and
- produces the text file in item 6.

I will run it, check your figure, and the contents and elegance (*i.e.*, nice format) of `commute.txt`. From this point in the course onward, we will place more emphasis on nice coding style than we have for previous labs / assignments. That is, your code should not only work but also be elegantly coded. Make sure you are not spewing output to the screen unnecessarily, that you have commented your code, and written your code as efficiently as possible.

As usual include your partner info in your `script` as comment lines.