Contents

FORMATTED OUTPUT

In the past, we used the `disp()` command to display outputs in the *command window*. However, there are many limitations to the `disp()` command: we have limited control over formatting, it cannot output/write to a file, and it becomes quite lengthy and clumsy to output a mix of character and numerical data. Thankfully, there is a more sophisticated way to output data. This is done using the `fprintf()` command. The general form of `fprintf()` is as follows:

```
fprintf(fileID,'string with format specifiers',variable1,variable2,…)
```

Note that the form of `fprintf()` is much different than that of `disp()` – do not confuse the two as they are two completely separate commands. The syntax rules that apply to `fprint()` do not apply to `disp()` and vice versa. Let's break down the different arguments in `fprintf()`:

- The `fileID` is a variable that serves as a file identifier; it is a MATLAB variable that identifies (and directs output to) a file external to MATLAB. The file identifier is obtained from the `fopen()` command (discussed later in this document). It does not need to be named "`fileID`" – you can use any valid variable name for a file identifier in MATLAB. The file identifier is how you refer to an external file that has been opened in MATLAB using the `fopen()` command – when you send output to a specific `fileID` using `fprintf()`, MATLAB directs this output into the exact file you provided when using `fopen()`.
    - The `fileID` is an optional argument. If you do not include any `fileID`, the output of `fprintf()` will be sent to the *command window* by default.
- The second argument of `fprintf()` is a string (i.e., it is surrounded by single quotes) that contains characters/words that you want to output, as well as format specifiers. Format specifiers, described below, essentially serve as a placeholder for data that you want to output. The data that you want to output is stored in variables, which are listed as the 3rd argument(s) of `fprintf()`.
    - When using `fprintf()`, the variables' values are inserted into the string wherever you place a format specifier.
    - This differs from the `disp()` command, which simply prints from left to right in order.

- The third argument(s) of `fprintf()` are the list of variable(s) that you want to output. The variables will be printed in the order that you list them, with their value(s) being inserted into the string wherever you place a format specifier.
    - The variables that you print can be of any data type, just be sure to use the appropriate format specifier depending on the type of data you are outputting.
    - You can include any number of variables when using `fprintf()`, including none. If you include no variables, you can print a character string directly as you did with the `disp()` command. This is uncommon and typically one or more variables are included in any `fprintf()` statement.

While there are countless format specifiers (conversion characters) available in MATLAB, we will focus on only a handful of them in MAE10. Here are the most important ones to remember and the data type they are used for:

`%f` fixed-point (decimal) notation
`%e` scientific (exponential) notation
`%g` the more compact of `%f` or `%e` with no trailing zeros
`%i` integer
`%c` single character
`%s` string of characters (character vector or string array)

When using a format specifier, you always start with the percent sign (%), followed by other arguments as shown here:

In most cases, you only need the percent sign, followed by the field width, a period, and the precision, then the conversion character (which specifies the data type). Many of these arguments are optional and are used only in special circumstances. For MAE10, you **must** know how to use flags, field width, precision, and conversion characters (including the default behavior if arguments are omitted and how to provide any requested format by including certain arguments). Because there are many rules, conversion characters, flags, etc., to remember for `fprintf()`, it is a good idea to keep the most important information on your note sheet for reference during the exams.

Here is a description of the main arguments we will use in MAE10:

- The percent sign is always included and marks the location of where the value you are printing will be inserted into the string.
- Flags are optional. They can be used to exercise more control over the output. For example, they allow you to left justify, print a sign character (+ or −), pad values with zeros, etc.
- The field width is optional but almost always included. It determines how many spaces are reserved for the entire value to be printed. Note that *every single character* takes up one space; this includes periods, + or − signs, letters, numbers, etc.
    - If you specify a field width that is larger than necessary, extra blank spaces will be printed in your output. By default, the extra blank spaces go in front (to the left of) the value being printed (the values are right-aligned).
    - If you specify a field width that is smaller than the field width required to print the value, MATLAB will automatically increase the field width as necessary to fit the value. That is, the field width is the *minimum* number of spaces that output will take.
- The precision is optional but typically included for numerical data. It determines the number of digits to the right of the decimal when printing numerical data with `%f` or `%e`.
    - When using `%g`, the precision determines the total number of significant digits, which includes numbers to the left of the decimal.
    - You do not include a precision when printing character data.
- The conversion character is required. It essentially tells MATLAB what type of data will be output (integers, decimals, characters, etc.) and what format you want the data to be printed in. Be sure to use a conversion character appropriate for the data stored in the variable(s) you are outputting.

The best way to understand the behavior of `fprintf()` is to practice using different arguments, format specifiers, and flags to output various types of data stored in different variables. Having a few examples on your note sheet can also be helpful, but don't overdo it. Let's start with a simple example and build up from there.

In `myfile.m`:

```
midterm1 = 84.5;
midterm2 = 95.1;
fprintf('Score#1 = %f',midterm1)
fprintf('Score#2 = %f',midterm2)
```

Sample Output:

```
Score#1 = 84.500000Score#2 = 95.100000
```

This may not be the output you expected – the output from both separate `fprintf()` commands appeared on the same line! Unlike the `disp()` command, separate `fprintf()` commands do not automatically start each output on a new line. To tell `fprintf()` to insert a new line, we must include the special operator `\n`. A `\n` can be placed anywhere inside of the string in an `fprintf()` command, and a new line (also called a carriage return) will be inserted at that location. In most cases, you will want to include a `\n` at the end of your `fprintf()` so that the next `fprintf()` will print on a new line, rather than continue immediately following the previous output along the same line. However, be aware that a `\n` can be used anywhere inside the string, and more than one `\n` can be used in the same `fprintf()` if desired. There are also other special operators you can include in `fprintf()`, such as `\t` for a horizontal tab. Using tabs to separate data when printing to a file is particularly useful as several programs (such as Microsoft Excel and even MATLAB) can easily import tab-delimited data into separate columns.

The other thing you should notice from the above example is that the values output were shown with a precision of six. That is, there are six numbers after the decimal point. This is the default for `%f` (and `%e`); if you do not specify a precision, it will default to six. MATLAB automatically adds trailing zeros to give the requested precision if the value being printed does not already include the requested number of digits after the decimal.

Let's modify the previous example by doing two things: adding a `\n` and including a precision for printing out our values (instead of using the default for `%f`).

In `myfile.m`:

```
midterm1 = 84.5;
midterm2 = 95.1;
fprintf('Score#1 = %.1f\n',midterm1)
fprintf('Score#2 = %.2f\n',midterm2)
```

Sample Output:

```
Score#1 = 84.5
Score#2 = 95.10
```

This time, we used a precision of `1` for the first score and a precision of `2` for the second score. We also included the `\n` operator to ensure that a new line is inserted at the end of each `fprintf()`. Notice that we still did not include a field width – it is okay to include a precision without a field width. If you do so, MATLAB will automatically determine how many spaces are required to fit the number (and you will not end up with any extra blank spaces).

Let's continue with the same example but now we are going to include a field width, combine our two midterm scores into a single array, and use just one `fprintf()` command for output.

In `myfile.m`:

```
midterm = [84.5 95.1];
fprintf('Score =%8.2f\n',midterm)
```

Sample Output:

```
Score =    84.50
Score =    95.10
```

In this example, we made a few changes. Here we print out the `1x2` midterm array using just one `fprintf()` command. This brings up an important point about `fprintf()` – if you do not include enough format specifiers to print all the values stores in the variable(s) (e.g., two values in the midterm array but only one format specifier), the entire `fprintf()` statement will be restarted until all values stored in the variable(s) are printed. This is typically referred to as "recycling" format specifiers. Thus, on the first pass through the `fprintf(), midterm(1)` is printed. Because the end of the `fprintf()` is reached but there is another value in the variable `midterm`, the `fprintf()` is restarted, and `midterm(2)` is printed. Both values are printed using a precision of `2` and field width of `8` as determined by the format specifier. The value takes a total of `5` spaces to be printed (`4` for the numbers, `1` for the period), so `3` extra spaces are left over out of the `8` we reserved. These blank spaces are placed in front of the number by default, and the number is aligned to the right. This means there are `3` blank spaces total between the equals sign and the value.

When we are printing more than one array using `fprintf()`, all values in the first array are printed before moving to the next array. The following example illustrates this point:

In `myfile.m`:

```
x = 1:5;
y = x.^2;
fprintf('%i \t %i \n',x,y)
```

Sample Output:

```
1       2
3       4
5       1
4       9
16      25
```

This is probably not the desired output – all of the values in `x` were printed before the values in `y`. If we want to print the values in a two column table, we need to combine the `x` and `y` vectors into a larger 2D matrix. However, we need to be careful because `fprintf()` prints ***down the columns*** of arrays when they are 2D. Let's look at an example to illustrate:

In `myfile.m`:

```
x = 1:5;
y = x.^2;
z = [x;y]
fprintf('%i \t %i \n',z)
```

Sample Output:

```
z =

     1     2     3     4     5
     1     4     9    16    25
1     1
2     4
3     9
4     16
5     25
```

Here we printed the values in `z` (in column order), with each pair of values separated by a tab. The creation of the `z` array was left unsuppressed so that you can see how the values are arranged in the `z` array compared with how `fprintf()` prints these values.

Let's look at one more example with a `3x3` matrix to ensure we understand how `fprintf()` operates on 2D arrays:

In `myfile.m`:

```
A = [1 2 3; 4 5 6; 7 8 9]
fprintf('%3i%3i%3i\n',A)
```

Sample Output:

```
A =

     1     2     3
     4     5     6
     7     8     9
```

```
  1   4   7
  2   5   8
  3   6   9
```

Here we printed the values in `A` using `%i` for integers, with a field width of `3`. This means that `3` spaces were reserved for each value. Since each value only takes up `1` of these spaces, we have `2` extra blank spaces in front of each number. Additionally, noticed that we used what we learned from the previous examples – we included only `3` format specifiers but have `9` values total to print. After the `fprintf()` finishes the first pass, there are still values in `A` that need to be printed, so the entire `fprintf()` is restarted until all values in `A` have been printed. Now look carefully at the order of the output compared to the original `A` array. The `fprintf()` printed the values in ***column order***, starting at the top left and going down column one before moving to column two, and so on. If we wanted to print out the `A` array in its original shape, we need to transpose it before using `fprintf()`. Using the `transpose()` command on an array before printing its values using `fprintf()` is common.

Let's revisit homework 1 problem 4 but use `fprintf()` for the output instead of `disp()`. In this final example, we are going to put together everything that we have learned so far:

In `myfile.m`:

```
x = [0:0.1*pi:pi];
tablearray = [x ; cos(x) ; sin(x)];
a = 'x'; b = 'cos(x)'; c = 'sin(x)';
fprintf('%10s%10s%10s\n',a,b,c)
fprintf('%10.2f%10.2f%10.2f\n',tablearray)
```

Sample Output:

```
         x    cos(x)    sin(x)
      0.00      1.00      0.00
      0.31      0.95      0.31
      0.63      0.81      0.59
      0.94      0.59      0.81
      1.26      0.31      0.95
      1.57      0.00      1.00
      1.88     -0.31      0.95
      2.20     -0.59      0.81
      2.51     -0.81      0.59
      2.83     -0.95      0.31
      3.14     -1.00      0.00
```

Be sure that you understand exactly how the code in the M-File produces the sample output shown above.

In the following example, we are going to print a mix of alphanumeric data using different format specifiers with various field widths and precisions. A detailed breakdown and explanation of how each `fprintf()` statement in the M-File produces the sample output is included below.

In `myfile.m`:

```
score(1) = 92.5;
score(2) = 81.0;
average = (score(1) + score(2)) / 2;
names = char('Paul' , 'Frank');
fprintf('The test scores were %8.3f and %1.1f\n', score(1), score(2))
fprintf('The test scores were %6.3f and %9.0f\n', score(1), score(2))
fprintf('The test scores were %9.1e and %9.4e\n', score(1), score(2))
fprintf('Hello %10s and %8s \n', names(1,:), names(2,:))
fprintf('%s received a score of %e \n', names(2,:), score(1))
```

Sample Output:

```
The test scores were   92.500 and 81.0
The test scores were 92.500 and        81
The test scores were   9.3e+01 and 8.1000e+01
Hello      Paul  and    Frank
Frank received a score of 9.250000e+01
```

For the first `fprintf()`:

- 8 spaces total are allotted for `score(1)` and 3 of the spaces are allotted for numbers to the right of the decimal. The decimal itself takes 1 space, so 4 spaces are left for numbers to the left of the decimal. Since `92` takes up 2 spaces, there are 2 extra blank spaces left (placed in front of the number).
- 1 space is allotted for `score(2)` and 1 space is allotted for numbers to the right of the decimal. The decimal also takes 1 space, so we can see right away the field width we provided is not sufficiently large to fit the number. Thus, MATLAB increases the field width as necessary to give the requested precision and fit the number. Since `81` take up 2 spaces, the number takes up a total of 4 spaces. There are no extra blank spaces.

For the second `fprintf()`:

- 6 spaces are allotted for `score(1)`, which is the perfect amount. 3 of the spaces are allotted for numbers to the right of the decimal. The decimal takes 1 space and there are no extra blank spaces.

- Notice that there is no decimal point for `score(2)`. This is due to zero spaces being allotted to numbers after the decimal (a precision of zero). In this case, there are 7 extra blank spaces.

For the third `fprintf()`:

- We are now using the `%e` format specifier, so numbers will be printed in scientific notation. 9 spaces are allotted for `score(1)` with 1 of the spaces being allotted for numbers to the right of the decimal. Notice that the '5' is truncated (and the 2 is rounded up) – we requested a precision of 1, which means that only one number after the decimal will be displayed. The decimal point takes up 1 space and the `e+01` takes up 4 spaces. Thus, there are 2 extra blank spaces in front.
- With a `%9.4e`, our field width is not large enough to fit the number. `8.1000` takes up 6 spaces, and `e+01` takes up another 4, meaning we need 10 spaces total. MATLAB automatically adjusts our field width to 10 to give us a precision of 4, and no extra blank spaces result.
    - Note that with scientific notation, there is always only one number to the left of the decimal.

For the fourth `fprintf()`:

- We are now using the `%s` format specifier, which is for printing strings of character data. With the `%10s` we reserve a total of 10 spaces, but only 4 are required to print `Paul`, so we have 6 extra blank spaces in front.
    - NOTE: One of the extra spaces is actually from a padded space in the `names` array (each row has 5 elements).
- Similarly, `Frank` takes up only 5 of the 8 spaces reserved, so we have 3 extra blank spaces in front.

For the fifth and final `fprintf()`:

- We use a `%e` format with no field width or precision, so MATLAB uses the default precision of 6 and adjusts the field width to fit the number. This number takes a total of 12 spaces to be printed.

Writing Data to a File: The `fopen()` and `fclose()` Commands

So far we have discussed a variety of ways to format and output data using `fprintf()`, but all output so far has been directed into the *command window.* While this is useful for testing purposes and to quickly look at small quantities of data, in practice we want to export our data into files external to MATLAB. This is particularly true if we are generating large quantities of data; hundreds, thousands, or even millions of lines of data are not uncommon in engineering computations. We would not want to output this quantity of data into the *command window*, so we write it to a file outside of MATLAB instead. Once we write the data to an external file, we can easily share it with others, keep a copy for backup, and use various other programs (or perhaps even MATLAB) to further process and analyze the

data. The good news is that everything you have learned so far about `fprintf()` applies exactly the same when we are printing data into a file. The only difference is that we now include the file identifier, the optional first argument of `fprintf()`, that we omitted in all the previous examples. When we include the file identifier, output is directed to that file rather than to the *command window*.

To get a file identifier, we need to first open the file using the `fopen()` command. The general form of the `fopen()` command is the following:

```
fileID = fopen('filename','permission')
```

- Where `fileID` is the variable that serves as a file identifier – it can be any valid variable name and does not need to be called "`fileID`". After using `fopen()`, the `fileID` is how you refer to the file in MATLAB (not by its filename and extension that appears in your computers file browser).
- The `'filename'` is the exact name of the file that you want to open. This **is** the filename exactly as it appears in your computer's file browser *with the file extension*. The filename is case sensitive and the extension must be included. It should exactly match the file on your computer.
    - Always place the full filename and extension in single quotes.
- The permission is exactly that – it determines what permissions you are granting MATLAB with the file you are opening. The main two permissions you need to know for MAE10 are:
    - `'r'` for read only permission – this means that you plan to read in data from the file (using `fscanf()` for example). You would not use a `'r'` permission if you are going to be writing data into the file using `fprintf()`.
        - If you do not include a permission at all, (e.g., you omit the permission argument and just include the filename in `fopen()`), the default permission is read only.
    - `'w'` for write permission – this means that you plan to write data into the file (using `fprintf()` for example).
        - **Be careful using `'w'` permission!** If you include the `'w'` permission the file will be created if it does not exist and it will be ***immediately erased*** if it already exist.
    - The letter(s) for the permission are always placed in single quotes.

There are other arguments that can be included in `fopen()`, but this is essentially all you need to know for MAE10.

Let's repeat the example we already covered with the `3x3` array `A`, but this time we are going to print the output into a file called `'matrix.txt'` instead of to the *command window*.

In `myfile.m`:

```
A = [1 2 3; 4 5 6; 7 8 9];
myfile = fopen('matrix.txt','w');
fprintf(myfile,'%3i%3i%3i\n',A);
fclose(myfile);
```

Inside `matrix.txt`:

```
  1   4   7
  2   5   8
  3   6   9
```

In this example, we have no output in the *command window*; the `A` matrix is suppressed (as are the `fopen()` and `fclose()` commands) and the `fprintf()` output is directed into the file `matrix.txt` via the file identifier `myfile`. Once again all of the rules for `fprintf()` that you have learned previous apply exactly the same whether you are printing into a file or to the *command window*.

Here is a summary of the key points to remember when using `fprintf()`:

- Recall which arguments are optional and which are required as summarized above.
    - For MAE10, you **must** know how to use flags, field width, precision, and conversion characters.
    - The most commonly used form of a format specifier in `fprintf()` will be: `%+X.Yf`
        - Where `+` can be replaced with any valid flag.
        - Where `X` is the minimum number of spaces reserved for the variable's value.
        - Where `Y` is the number of digits to display to the right of the decimal.
            - Note that `X` and `Y` are separated by a period.
        - Where `f` can be replaced with any valid conversion character.
            - Note that with some conversion characters (e.g., strings and integers), a precision `Y` is never included.
- Know what the default behavior is when using each type of conversion character.
    - For example, the default precision when using `%f` or `%e` is 6.
- If you specify a field width that is too small (not enough spaces total) to fit the value being printed with the requested precision, MATLAB will increase the field width to provide the requested precision (and no extra blank spaces will result).
    - In order words, the field with is the *minimum* number of spaces reserved for printing the value.

- If you specify a field width that is larger than necessary, any extra blank spaces will be placed in front of the value by default (the value is aligned to the right).
    - Optionally, you can include a flag to left-align values.
    - You will only end up with extra blank spaces if you reserve a field width that is larger than necessary.
- Remember that *any* character takes 1 space of the field width. This includes numbers, signs, letters, periods, commas, etc.
- Remember that variables are printed in the order they are listed (from left to right), and their values are inserted into the `fprintf()` string wherever a percent sign and conversion character are included.
    - When printing arrays, all values in a given array are printed before moving onto the next array in the variable list.
- When printing 2D or larger arrays using `fprintf()`, values are printed in column order. That is, starting in the left-most column, all values in a given column are printed (from the top down) before moving to the next column to the right.
- If you do not include enough format specifiers (i.e., more values in the variable list than format specifiers in the string), the entire `fprintf()` statement will be restarted until all values are printed.
- If you include more format specifiers than necessary (i.e., more format specifier than values in the variable list), the `fprintf()` will be terminated early (after the last value is printed).
- When the `fileID` argument is omitted, `fprintf()` will output to the screen.
    - If a `fileID` is included, the output of the `fprintf()` will be directed to the external file with that `fileID`. Be sure that you have used `fopen()` to open a file and create the `fileID` variable before using `fprintf()` to print to the file (and use `fclose()` to close the file after you are finished printing data into it).
- We never need brackets when using `fprintf()` like we did with `disp()`. Additionally, `num2str()` is never necessary with `fprintf()` because we simply provide the appropriate format specifier when we are printing numerical or character data.