Lab II : Introductory Octave

Example 1 : The for loop

As you will remember from the lecture on shell, control structures are special keywords that alter the flow of the code. In Octave, the same concept exists with slightly different syntax.

```
octave:1> for n=1:5
> printf("The number now is %d, the next number is %d.\n",n,n+1);
> endfor
The number now is 1, the next number is 2
The number now is 2, the next number is 3
The number now is 3, the next number is 4
The number now is 4, the next number is 5
The number now is 5, the next number is 6
```

The above example is a very simple exercise in using the for loop. After the keyword for, the variable and its range is specified. The for loop is terminated with the keyword endfor. Although it suffices to just use end to terminate all the control structures, I prefer the full keyword so as to keep track of nested control structures.

Now, open up your emacs (or your favorite editor) and type in the commands in the above example. The name you should give your file is not important but it should have the extension ".m". Let's say you call your file for_loop.m. You can then run it from Octave by just calling with the file name omitting the extension.

```
hande@p439a:~$ cat for_loop.m
for n=1:5
   printf("The number now is %d, the next number is %d.\n",n,n+1);
endfor
hande@p439a:~$ octave
octave:1> for_loop
The number now is 1, the next number is 2.
The number now is 2, the next number is 3.
The number now is 3, the next number is 4.
The number now is 4, the next number is 5.
The number now is 5, the next number is 6.
```

The printf statement in the body of the for loop is in fact a C function. The special charaters %d designates an integer. The
n character at the end tells the print statement to start a newline at the end of the line being printed. Finally, the variables after the double quotes, n and n+1 are what should be substituted in place of the %d's.

You can change the limits and the increment of the for loop. Modify your for_loop.m file to instead have the following and observe what happens.

```
  for n=1:2:10
  ...
```

```
  for n=0:0.1:1
  ...
```

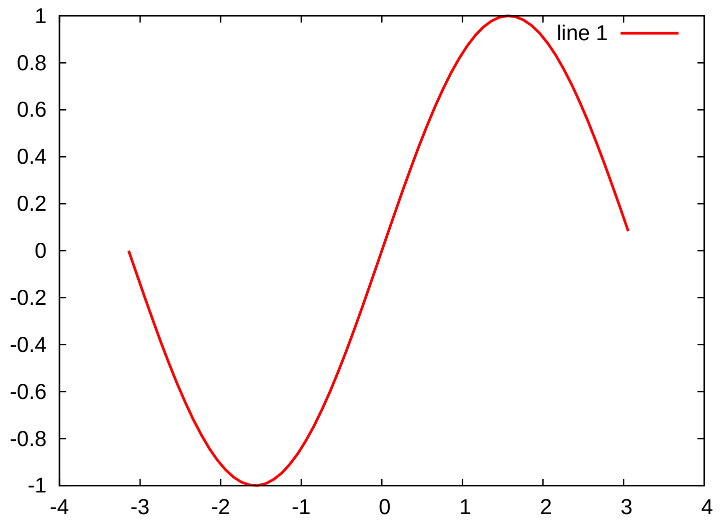
```
  N=[1 4 10 55]
  for n=N
  ...
```

Example 2 : Plotting

Often, we would like to make a graph of our results. Octave does not have an intrinsic plotting function. Instead it uses another plotting program in Linux, which is called gnuplot. In order to make a plot in Octave, we must first define the range in terms of an array. Then we can plot the function as a function of that array

```
octave:1> x=-pi:0.1:pi;
octave:2> y=sin(x);
octave:3> plot(x,y)
```

The resulting plot will look like the following :



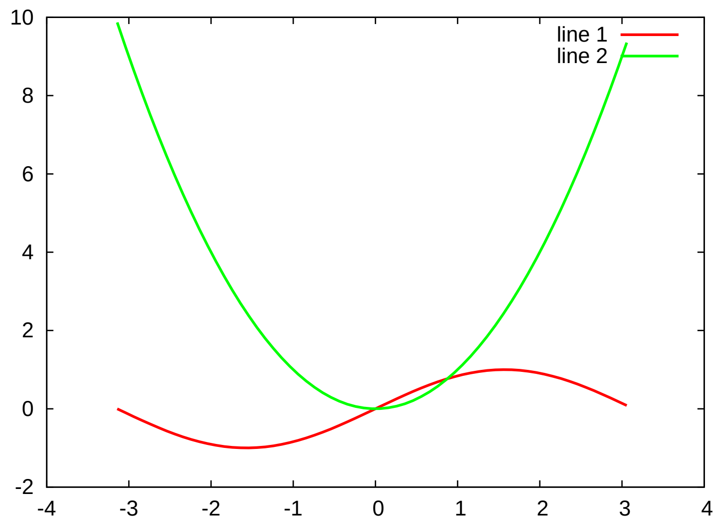
In fact, it is not necessary to make a second array, `y` for the vertical axis, we can draw the graph directly with a single comment.

```
octave:1> x=-pi:0.1:pi;
octave:2> plot(x,sin(x))
```

Sometimes, we would like to superimpose two or more plots.

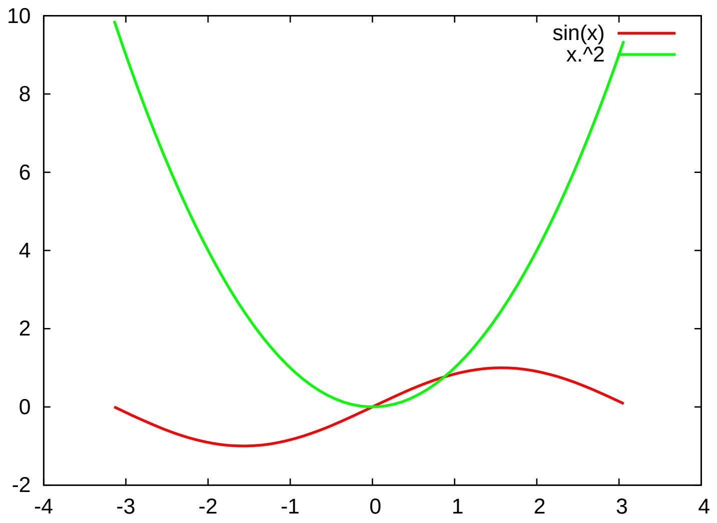
```
octave:1> x=-pi:0.1:pi;
octave:2> plot(x,sin(x),x,x.^2)
```

We can just keep adding to this single `plot` command as many commands as we like. This produce the following graph.



Plots are more readable with *legends*, which are explanatory expressions on the upper right corner of the plot, color-coded for each graph on the plot. The syntax is rather clumsy but you will get used to it in time.

```
octave:1> x=-pi:0.1:pi;
octave:2> plot(x,sin(x),';sin(x);',x,x.^2,';x.^2;');
```



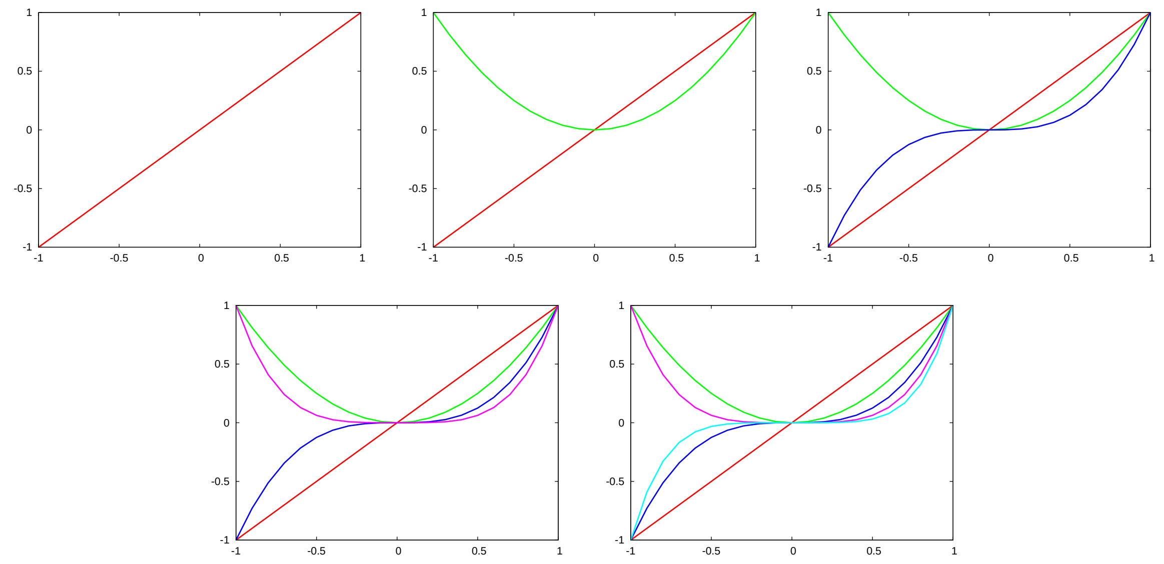
Sometimes the above way of superimposing plot may not be well-suited to what you would like to do. For example, if you would like to write to loop to superimpose plots, where at every stop of the loop you want to add another plot to the collection, you simply cannot write everything in a single `plot` command. In that case, you should use the `hold` command. This time, let's write a script called `plot_poly.m`, which superimposes plots of polynomials of orders 1 to 5.

```
hande@p439a:~$ cat plot_poly.m
## Superimpose polynomials of orders 1 through 5.
hold off ## To lift any previous holds
x=-1:0.1:1;

for n=1:5
  plot(x,x.^n,';;');
  hold on
  pause
endfor

hold off  ## To lift the hold so that any new graph will not be plotted
          ## over the polynomials
hande@p439a:~$ octave
octave:1> plot_poly
```

The `pause` statement in the code above makes running the function a little more fun by waiting for a *keyboard input* from you. This means that after every plot, it will pause and wait for you to hit a key (any key will do). This way you can see the gradual accumulation of the graphs. The `hold off` at the beginning of the script clears any previous holds. The `hold off` at the end lifts the hold on the current accumulated plot so that plots from a different script do not interfere. The sequence of outputs should thus look like :



You see that `Octave` even is smart enough to color each plot differently.

Example 3 : The factorial

The Taylor expansion of an entire function around zero is given by

$$f(x) = \sum_n f^{(n)}(0)\frac{x^n}{n!}$$

which for an exponential yields

$$e^x = \sum_n \frac{x^n}{n!}$$

We'll now see how quickly this series converges to the real function.

In this example, we will write a function to calculate the factorial of an integer. Our function should not only calculate factorial of a correct input (nonnegative integer) but also warn the user when the input is incorrect.

```
hande@p439a:~$ cat factorial.m
## Usage f=factorial(n)
function f=factorial(n)

if ( (n<0) || (rem(n,1)~=0) )
  printf("n cannot be a negative number. Exiting...\n");
  f=’’undefined’’;
  return
endif

f=1;
for num=1:n
  f=f*num;
endfor

endfunction
```

The above function performs the following tasks :

- Using the `if` statement, it checks whether the input has been entered correctly. The double vertical lines, `||`, is a logical OR statement. If the number you entered is smaller than zero or a noninteger, the code returns `undefined` for the output and exits into the `Octave` prompt with a warning sign. The `rem` function calculates the remainder of a number when divided by another number. If the remainer of `n` is not zero then `n` is not an integer.

- It starts with `f=1` and at every step of the four loop update `f` by multiplying it with the current `n`. So the output for, say n=5, proceeds as follows

```
num=1  f(old)=1   → f(new)=f(old)*num=1
num=2  f(old)=1   → f(new)=f(old)*num=2
num=3  f(old)=2   → f(new)=f(old)*num=6
num=4  f(old)=6   → f(new)=f(old)*num=24
num=5  f(old)=24  → f(new)=f(old)*num=120
```