CS30 Spring 2015
Lab 8

Use the command `diary` to record your answers and submit them. Submit code for the functions and scripts you write. Submit any figures.

1. (50 points) Recursive functions. Given as input a list of integers, positive or negative, return the list sorted from smallest to largest. Do this by writing your own implementation of *merge sort*, an algorithm for sorting a list. Merge sort is based on a merge step, where given two lists, each individually sorted, they are merged into one sorted list.

   (a) Write a function `MergeLists` that takes as input two sorted lists `list1, list2`, and returns as output `mergedList`. For example

   ```
   >> MergeLists([1, 4, 6], [-1, 2, 3, 4, 5, 6 ])
   ans =
       -1     1     2     3     4     4     5     6     6
   ```

   (b) Write the recursive sorting algorithm `MergeSort` which takes as input the unsorted list `list` and returns as output the sorted list `sortedList`. Merge sort works as follows. If the input list has only one element, it returns the input list. Otherwise, it splits the input list in two approximately equal sublists, calls `MergeSort` recursively on each sublist, and then calls `MergeLists` to merge the two sorted sublists. Your function should satisfy the following test cases:

   ```
   >> MergeSort([1])
   ans =
        1
   >> MergeSort([-2     3    10     3    -8    -6     8     0     9     9])
   ans =
       -8    -6    -2     0     3     3     8     9     9    10
   >> MergeSort([ -12     3     6     6    -1    -1    -1   -11     1   -14 ])
   ans =
      -14   -12   -11    -1    -1    -1     1     3     6     6
   ```

   (c) Compare the performance of your `MergeSort` with Matlab's function `sort`. Make two plots showing list length vs. run time for both implementations, with lists of length up to 1 million elements. You can call the functions on random integer lists generated using `randi`, and use the functions `tic` and `toc` to time the sorting algorithms.

2. (50 points) Newton's Method. Recall that Newton's method can be used to find roots of a function. It starts with an initial guess x_0, and proceeds iteratively. In particular, given the current value for the root x_k, Newton's method generates a better value by solving

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

   where f' is the first derivative of f.

   (a) Write a generic implementation of Newton's method `NewtonsMethod` that takes as input a handle to the function f, a handle to the first derivative function f', an initial guess and a stopping threshold ϵ, and returns as output the final iterate x_k. Newton's method iterates until $|f(x_k)| < \epsilon$. Make the initial guess and stopping threshold optional, with default values of 0 and 10^{-5}, respectively. Allow a maximum of 100 iterations, even if the stopping criterion hasn't been met. Output a warning if the maximum number of iterations was computed and the method did not converge. Test your function on some of Matlab's built-in math functions, as follows.

```
>> NewtonsMethod('sin','cos',pi/2+.1)
ans =
   12.5664
>> NewtonsMethod(@cos,@(x) -sin(x),pi/2+.1)
ans =
    1.5708
```

(b) Let $f(x) = x^2 - 2$. On the command line, set the variable `myPoly` to be a function handle for an anonymous function implementing f, and set the variable `myPolyDeriv` to be a function handle for an anonymous function implementing f'. Run the following command:

```
>> NewtonsMethod(myPoly,myPolyDeriv,.1,10^-7)
ans =
    1.4142
```

Do the same for $f(x) = x^5 - x + 1$.

```
>> NewtonsMethod(myPoly,myPolyDeriv,.1)
Warning: did not converge in 100 iterations
ans =
   1.000257561949280
```