Assignment #2

This assignment is due on April 5th on hour before class starts via email to **christian.wallraven+AMS2016@gmail.com**.

If you are done with the assignment, make one zip-file of the `assignment1` directory and call this <LASTNAME_FIRSTNAME_A2.zip> (e.g.: HONG_GILDONG_A2.zip).

Please make sure to comment the code, so that I can understand what it does. Uncommented code will reduce your points!

Also, please read the assignment text carefully and make sure to implement EVERYTHING that is written here – if you forget to address something I wrote, this will also reduce your points! Precision is key ☺!

Part1 Derivatives, function handles, plotting (60 points):

The goal of this part is to implement a **function** called `derive`, with which you can **numerically** derive a given input function. The function definition should be:

```
function [derivative]=derive(function_handle,x_values)
```

Note, that the first input argument is a handle to a function! Hence you would call this function like this, for example: `d=derive(@sin,[-pi:0.1:pi]).`

Now, your function needs to implement the numerical derivative. For this, we simply use the definition from class:

$$f'(x_1) = \lim_{x_2 \to x_1} \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

Unfortunately, you cannot do limits in computers, since they do not have infinite precision. So, we are left with approximating this limit. We do this by simply ignoring it, trying to use very small differences between the x-values instead.

$$f'(x_1) \approx \frac{f(x_2) - f(x_1)}{x_2 - x_1} \text{ with } x_2 - x_1 \text{ small!}$$

Implement this way of calculating the derivative as

```
function d=derive(f,x)
```

For this you need to loop through all elements of the `x_values` and apply the above approximation equation for each element of the `x_values` array. **Attention! You need to stop the loop at** `length(x_values)-1`. This means also that your function returns an array that contains a derivative for all x, except the last one!

Now, you create a test-script `testDerivative.m` that calls this function and plots the result.

Let's check the sin-function first. We know that the derivative of *sin* is *cos*. So call the function `d=derive(@sin,[-pi:0.1:pi])` and plot this data together with the actual *cos* function into the same plot in your script. Use a green line for your numerical derivative and a red line for the *cos*-function. You should find that the green line is close to the red one, but that it does not quite match it.

One reason for this may be that the difference between x2 and x1 is only 0.1, which makes the above approximation rather inaccurate. We therefore can try to make this difference smaller.

In your script, insert a for-loop that changes the difference of this array `[-pi:difference:pi]` from `difference=1` in 15 steps, dividing `difference` by 2 in every step. For each of the difference steps, I would like you to plot the error between your derivative and that of the original *cos*-function. If `n` is `length(d)`and `f=cos(x_values)` then

$$\text{error} = \sqrt{\sum_{i=1}^{n} (d(i) - f(i))^2}$$

Note that you can calculate this error in Matlab **without a for-loop!** Note also the different lengths of `d` and `f`!

So you should have something like this:

```
difference=1;

for step=1:15

      x = [-pi:difference:pi];

      d = …; f = …;

      err(step) = … % to avoid confusion with Matlab “error”

      difference=difference/2;

end

figure; loglog(difference_values,err,’ro-‘); grid
```

Note that you should plot the actual **values** of `difference` and not the steps! The loglog-plot will create a plot that has two logarithmic scales so you can interpret this easier. **What is the difference that will create an error smaller than 10^-6?** Insert this value as a comment into the script.

There is actually another reason for the discrepancy that you observed in the plot. This is that the approximation only holds well for the **middle value** between x2 and x1. So, when plotting the result of the derivative, we actually need to change the `x_values`. Perhaps it would be a better idea to use x-values that are in the middle between the original values. Copy the for-loop from above to make another one, but now plotting the error for the **new x-values**. You can use `hold` to hold the current figure so that the new data gets plotted into the same figure for comparison.

```
hold on; difference=1;

for step=1:15

      x = [-pi:difference:pi];

      d = …; realx=…

      f = f(realx); % obviously you need to use the new values

      err(step) = … % to avoid confusion with Matlab “error”
```

```
      difference=difference/2;
end
loglog(difference_values,err,'go-'); % use green plot
```

How large is the difference? What is the value for which you reach 10^-6 error now? Insert your comments into the script.

Now it's time to check some other functions and plot their derivative compared to the "real" solution. Plot **the derivatives** of the following functions (**NOT the functions themselves!!)** using a few more calls in your script `testDerivative.m`

$$f(x) = \log(x)$$

$$f(x) = \frac{1}{\sqrt{x}}$$

$$f(x) = \frac{\cos(x)}{x}$$

Each derivative should get its own plot. In addition, use "nice" intervals for plotting the function derivatives (note that many of these functions have **problems** around x=0!). Obviously, to compare your numerical derivatives to the "real" solutions, you need to derive the above functions yourself ☺. To pass the functions to your script, you will need to define them yourselves first by using anonymous function handles, e.g:

```
f1= @(x) x.^2;
```