Assignment #2

This assignment is due on April 11th one hour before class via email to **christian.wallraven+AMS2019@gmail.com**.

If you are done with the assignment, make one zip-file of the `assignment2` directory and call this zip-file `STUDENTID1_STUDENTID2_STUDENTID3_A2.zip` (e.g.: `2016010000_2017010001_A2.zip` for a team consisting of two students or `2016010000_2017010001_2017010002_A2.zip` for a three-student team). The order of the IDs does not matter, but the correctness of the IDs does! **Please double-check that the name of the file is correct!!**

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 (40 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,h)
```

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],0.1).`

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

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

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 h-values instead.

$f'(x) = \frac{f(x+h)-f(x)}{h}$, with h very small

Implement this way of calculating the derivative in your function. 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. Use h=0.1.

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],0.1)` 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 h 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 h from `h=1` in 10 steps, dividing h by 2 in every step. For each of the 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!**

So you should have something like this:

```
h=1;

x = [-pi:0.1:pi];

f = …;

for step=1:10

      d = …;

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

      h=h/2;

end

figure; loglog(<h_values>,err,’ro-‘); grid
```

Note that you should plot the actual **values** of h 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 value of h that will create an error smaller than 10^-2?** Insert this value as a comment 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`

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

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

$$f(x)=\frac{\sin(x)}{x+5}$$

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 or around other points that could be singularities!). 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;
```

Part2 Derivatives, images, plotting (20 points):

The goal of this part is to derive an image of a square on a background just like shown in class. First, we will need to create such an image.

In a script called `deriveImage.m` define a short function

```
imf = @(x,y) …
```

That outputs a true value if both **x and y** are between **30 and 70**.

Next, we need to use this function to create an image. For this, we will call it using the output of the `meshgrid` function that creates a grid of x and y points, with which we can call the `imf` function.

So, now do:

```
[x,y] = meshgrid(1:150,1:150);
```

```
img = double(imf(x,y));
```

We need to convert the output of `imf` to double, since it returns a logical value. If you want to take a look at the image, simply do

```
imagesc(img);
```

So now we have a square. We need to next take the derivative of this data. Note, as shown in class, we can do this in the x-direction (that gives us one picture of the x-derivatives) and in the y-direction (that gives us another picture of the y-derivatives).

How do we get the derivatives? We could try to re-use our derive function from Part1, but that would be a bit tedious to call. Instead we realize that taking the derivative in an image means to simply do the following:

```
dx = pixel(x+1)-pixel(x)/((x+1)–x) = pixel(x+1) – pixel(x);
```

Fortunately, Matlab has a very efficient function for this already, called `diff` that calculates the difference between subsequent values in an array.

We can now simply call `diff` with each **row** of the picture to get the x-derivatives per row. Next, we call it with each **column** of the picture to get the y-derivatives.

Implement this using two for-loops that do

```
dx(row,:) = diff(img(…))
```

```
dy(:,column) = …
```

and show the resulting derivative images `dx` and `dy` using `imagesc`. Note that the result has a different number of x- and y-values.

As a 5-point bonus, run the same code with an image of your choice! Insert all necessary commands into the script to load and show the two derivative images. Make sure to convert the image to double after reading it in using `imread` and include the image with your submission!!!