Your solutions to this homework can be broken up into as many files as you wish. However, your upload to the submit server must contain a single script called "hmwk1.m" or "hmwk1.py" that requires no arguments. This script should generate the outputs required by **some** problems below (when you need to put something in your hmwk1 script it will be stated in bold).

This is probably the most difficult homework I will assign (because I anticipate people may have some trouble with the concepts in problem 2). I anticipate this homework should be doable in 2-3 hours — but feel free to ask questions to help yourself out.

1. (a) Write a function with signature

```
function D, c = create_classification_problem(Nd, Nf, h)
```

The function generates `Nd` data vectors, each containing `Nf` features. Half the points should be in class "1" the others should be in class "-1." The two classes of points should be approximately linearly separable. The parameter `h` determines how hard/easy it is to separate the points. When `h=0`, this classes should be easy to separate using a plane. When `h` is large, the problem becomes very difficult to separate with a plane.

The function returns a feature matrix `D` with `Nd` rows and `Nf` columns. It also returns a column vector `c` containing the class ± 1 labels of the corresponding feature vectors.

Your hmwk1 script should call this function to create a dataset with 200 data vectors and 2 features per vector, and then visualize it with a 2D scatter plot. Do this for two different values of `h`.

(b) Create a function with signature

```
function y = logreg_objective(x,D,c)
```

that evaluates the logistic regression objective function

$$f(CDx)$$

where

$$f(z) = \sum_i \ln(1 + e^{-z_i}) = \sum_i -z_i + \ln(e^{z_i} + 1)$$

and C is a diagonal matrix with $C_{ii} = c_i$. (your code should not explicitly form C, but rather use element-wise multiplication with the column vector c.)

Note: computing e^z is dangerous because you get back `NaN` when z is big. When we use this code later for gradient descent, a single `NaN` in the gradient will ruin everything. Therefore, it is best to never evaluate e^z for positive z. For this reason, I suggest you use the formula $\ln(1 + e^{-z_i})$ when $z_i \geq 0$, and use $-z_i + \ln(e^{z_i} + 1)$ when $z_i < 0$. In exact arithmetic, these formulas are both the same. But in finite precision, this avoids the problem of the exponential function becoming unstable.

BIG HINT: use logical indexing to switch between the two formulas. For example, in Matlab you could do this:

```
% ...figure out dimensions of the problem, then....
f = zeros(Nd,1); %  allocate space
z = c.*(D*x);    %  Compute CDx
f(z>=0) = log(1+exp(-z(z=>0)); % compute terms in f
f(z<0) = -z(z<0)+log(1+exp(z(z>0));
y = sum(f);   %  sum everything
```

Numpy has the same logical indexing capabilities as Matlab, so a Python code should be almost identical.

(c) Create a function with signature

```
function g =  logreg_grad(x,D,c)
```

This function should return the gradient of the logistic obective you created in question 2.

(d) Use numerical derivatives to verify that your gradient function is indeed correct using a sample problem with 5 features (i.e., the vector x is of length 5). Your code should inspect several entries in the gradient (i.e. compute numerical derivatives with respect to several entries in x), or else automatically test them all. **Your hmwk1 script should print out the numerical and analytical derivatives to show that they agree to several digits of precision.**

2. (a) Create a function with signature

```
function G = grad2d(X)
```

This function accepts a 2-dimensional array `X` and returns a 3-dimensional array containing the image gradient. The 3 dimensional array contains a 2-dimensional array of x-differences AND a 2-dimensional array of y-differences, both sandwiched together along the third dimension. You must use the fast Fourier transform (which also means circulant boundary conditions). Note: use `fft2` and `ifft2` in Matlab for the forward/inverse Fourier transforms of 2d arrays.

(b) Create a function with signature

```
function d = div2d(G)
```

This function should accept a 3-dimensional array `X` and return a 2-dimensional array containing the adjoint of the gradient operator, which is called the (negative) divergence. This operator performs x-differencing on a 2D slice of G, y-differencing on the other 2D slice, and then adds the results to together. The sum (the return value) should be a 2D matrix. Note, the adjoint of a convolution operator is simply a convolution with the flipped stencil. Therefore, if you used forward differences for the gradient, you must use backward differences for the divergence.

(c) Verify that your gradient and divergence operators are indeed adjoints of one another using a non-deterministic test. Your test should be based on the following observation: the adjoint (transpose) of a linear operator (matrix) satisfies

$$\langle x, Ay \rangle = \langle A^T x, y \rangle.$$

To test whether two operators A and B are adjoints, we simply generate random x and y and then compute the quantities $\langle x, Ay \rangle$ and $\langle A^T x, y \rangle$. The two results should agree. If they do, then you have proved with probability 1 that A and B are adjoints.

Your hmwk1 script should print out these two inner products to show they are indeed equal.