Just like before, your homework submission must contain the following to receive full credit:

- A single script called "hmwk2.m" or "hmwk2.py" that requires no arguments and prints the things I ask for in bold below. Each output should be labeled.
- A pdf called hmwk1_results.pdf with your solutions to all theory problems and also the text output from the hmwk1 script. See the posted homework example.
- As many other code files as you wish - but only things you wrote or solutions to previous homeworks.

1. (a) Write a function with signature

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

The function generates `Nd` data vectors, each containing `Nf` features. Roughly half the points should be in class "1," and the others should be in class "-1." The two classes of points should be approximately (but not perfectly) linearly separable. The parameter `kappa` determines the condition number of the data matrix D. This should only require a few lines of code using your matrix builder from homework 1.

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

Your hmwk1 script should call this function to create a dataset with 100 data vectors and 2 features per vector, and then visualize it with a 2D scatter plot. Your script should make 2 plots: one for $\kappa = 1$, and one for $\kappa = 10$.

Note: This problem can be solved very easily if you're a little clever. Your solution should be a function containing only a few lines of code.

(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 $C_{ii} = c_i = \pm 1$. Your code must conform to the following:

- You may not use any *for* loops under any circumstances.
- You may *not* explicitly form the matrix C. You may only store the column vector c.
- You must have a separate sub-routine (or inline lambda function) that evaluates f. Your main function should call this subroutine.
- You can never evaluate e^z for $z > 0$. Computing e^z is dangerous because you get `NaN` when z is big. When we use this code later for gradient descent, a single `NaN` in the gradient will propagate through the code and crash everything. 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.

Note: use logical indexing to switch between the two exponential formulas.

Call your function on a classification problem with `Nd=1000` rows, `Nf=100` columns, $\kappa = 10$, and a random normal vector x.. Then, call the method on `10000x` and verify that you don't get NaNs.

2. (a) Write a function with signature

```
function x_train, y_train, x_test, y_test = load_mnist()
```

This function should load the mnist dataset. You can either download the data from Yann LeCun's web page, or load it from the mnist.h5 file available at the following location:

`cs.umd.edu/~tomg/dat/mnist.h5`

MNIST contains sample images of handwritten digits. Each row of the "x" matrices should contain a vectorized image from the dataset. For example, the dataset `x_test` should have dimensions 1000×784 (the test set contains 1000 images, each having $28 \times 28 = 784$ pixels).

The label `y_train` and `y_test` must be in *one's hot* form. There are 10 possible classes for each image (corresponding to the values 0-9), and so the dataset `y_test` should have dimensions 1000×10. Note, the labels will not be in this form when you first load them from the h5 file.

REMARK: The hdf5 file format is a standard for scientific data, and is extremely space and time efficient. An h5 object is often 3-5× smaller than a pickled Python object. Most hdf5 file libraries are also optimized for concurrent access from multiple threads so that many processes can read the data simultaneously. In Python, this file format also allows you to create abstract objects that represent and operate on datasets too large to fit into memory.

HINT: A script to load the matrix `x_test` in Matlab and display the first image would look a little something like this.

```
%% Load the dataset
h5disp('mnist.h5') % Display the contents of the h5
x_test = h5read('mnist.h5','/x_test'); % load/store x_test
x_test = x_test'; % swap rows/cols
%% Display the first image
sample_image= reshape(xt(1,:),[28,28]);
imagesc(sample_image);
```

In python, you would do something kinda like this.

```
import h5py
import numpy as np
import matplotlib.pyplot as plt
## load the dataset, one image per row
f = h5py.File('mnist.h5') # create a file object
print f.keys() # print file contents
x_test = f['x_test']; # load/store x_test
## Display the first image in the dataset
sample_image = np.reshape(x_test[0,:],[28,28])
plt.imshow(sample_image)
plt.show()
```

(b) Create a function to implement the "SoftPlus" non-linearity, which is a smoothed form of the RELU. Your function should have prototype

```
function y = softplus(X)
```

where `X` is an array of arbitrary dimensions. The SoftPlus function is given by

$$softplus(X) = \log(1 + \exp(X))$$

where log and exp are interpreted entry-wise. The output should have the same dimensions as the input X. Your implementation should not compute the exponential of any positive numbers.

(c) Implement a function

```
function y = cross_entropy(X,Y)
```

that accepts a matrix `X` of features (one feature vector per row, they may be non-positive valued), and a matrix `Y` of ones-hot labels of the same dimensions. The function returns the average cross-entropy (log-likelihood) of the soft-max of the rows.

The softmax of a vector x is given by

$$softmax(x)_i = \frac{\exp(x_i)}{\sum_k \exp(x_k)}.$$

The cross-entropy/negative-log-likelihood of the softmax (assuming ones-hot label vector `y`) is given by

$$NLL(x, y) = -\log\left(\sum_i y_i \frac{\exp(x_i)}{\sum_k \exp(x_k)}\right).$$

Your function will return the mean value of this quantity over all the row vectors in the input.

Important: your implementation can never exponentiate a positive number! You can only compute e^z for $z \leq 0$.

(d) Create a function to implement the loss of a neural network. Your net should use a SoftPlus for all non-linearities, except the last layer which is a combined softmax and cross-entropy. For simplicity, the network will not have bias terms. Your function should have prototype

```
function y = net_objective(W,D,L)
```

and the following specifications:

- Argument `W` is a cell array (or list in Python) of weight matrices. These matrices can have arbitrary dimensions, as long as the dimensions of adjacent layers are compatible.
- Argument `D` contains features vectors (the inputs to the network), one per row.
- The matrix L contains one's hot labels, one label vector per row.
- The output y is a scalar representing the classification accuracy of the neural network, as measured by the cross-entropy loss function.
- Your function may contain only 1 `for` loop (to loop over layers of the net). You may NOT loop over feature vectors.

(e) Create random Gaussian weights for a neural network with 784 input features, followed by one hidden layer of 100 neurons, followed by an output layer of 10 neurons/categories. **Evaluate your objective function using these random weights, and MNIST training data.** Print your objective value to the console.

3. (a) Create a function with signature

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

This function accepts a 2-dimensional array (image) `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/python for the forward/inverse Fourier transforms of 2d arrays.

REMARK: The FFT is not the fastest way to do this. However, you will need an fft based implementation later in the course, so we'll create one now.

HINT: Here's a short script that computes the differences in the X-direction only.

```
% I want to filter my image with the stencil [-1 1 0].
% This is a 'backward' difference stencil.
% Remember, you must CONVOLVE with the FLIPPED stencil to get
% the linear filtering you want.  Also, you need the center of
% the stencil to be the left-most entry in the kernel.
% So, I will convolve with:
kernel = zeros(size(X));
kernel(1,1)=1;
kernel(1,2)=-1;
%  Create the diagonal matrix in decomposition K=F'DF
Dx = fft2(kernel);
% Use the eigen-decomposition to convolve stencil with X,
% and get the differences in the horizontal direction.
Gx = ifft2(Dx.*fft2(X));
%  Gx now stores the x-differences
```

(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. This function should implement a linear transformation that is the adjoint (transpose) of the gradient operator. This is called the (negative) divergence. This operator performs x-differencing on a 2D slice of G (the x-difference slice), y-differencing on the other 2D slice (the y-difference slice), and then adds the results together to get a 2d array. 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. So if your grad operator uses the stencil $[-1, 1, 0]$ for the x differences, then your div operator uses the stencil $[0, 1, -1]$. If this confuses you, try writing out a few rows of the convolution matrix that generates the differences, and then transpose it.

(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.