# HW1 Report

**Group Member: Miao Jiang (miajiang), Snehil Vishwakarma(snehvish) , John Henderson (jphender)**

**Files in Repository:**
Github Site: https://github.iu.edu/cs-b657-sp2017/miajiang-jphender-snehvish-a1
Please see the most updated code in master's branch.

**Modules in Part 1:**

**Implementation Details of part1:** All the functions of part 1 are implemented in "watermark.cpp"
To run all the parts (spectrogram generation, interference noise reduction, and watermark addition and check) can be called sequentially with the shell file (makenrun.sh) by typing in
**> ./makenrun.sh**

To run each part sequentially, we have all separate shell files (makenrun_specto.sh, makenrun_noise.sh, makenrun_add_wm.sh, makenrun_check_wm.sh) and then can be called individually (except makenrun_add_wm.sh & makenrun_check_wm.sh)
**> ./makenrun_specto.sh**
**> ./makenrun_noise.sh**
{
    **> ./makenrun_add_wm.sh**
    **> ./makenrun_check_wm.sh**
}

We have a separate folder called "pictures", which hold all the pictures for testing of this part. If you want to add any image to the test module, create a folder with the same name of the image and place the folder in "/part1/pictures" and then place ".png" image inside the newly created folder.
All the scripts (except noise reduction) recurse over all the folders in "pictures" and generate output for every folder inside it with respective names.
The output names for spectogram are "spectogram_<filename>_.png"
The output names for noise reduction are "cleaned_noise1.png" and "opt_cleaned_noise1.png"
The output names for watermark addition are "watermark_<filename>_<watermarkvalue>.png"
(We are saving the <watermarkvalue> to the created image to use it to check for false positives and missed watermarks while checking for watermark)
The output for watermark checking is either "False Positives/ Missed WaterMark/ Correct Watermark" detection as output statements. (We tried to maximise Correct Watermarks and reduce the other 2)

**Functions used & their implementation idea:**

**COMMON in ALL PARTS**
Whenever we need the fourier domain from the spatial domain we call
*void fft(const SDoublePlane &input, SDoublePlane &fft_real, SDoublePlane &fft_imag)*
Whenever we need the spatial domain from the fourier domain we call
*void ifft(const SDoublePlane &input_real, const SDoublePlane &input_imag, SDoublePlane &output_real)*

For every input image, we check if it is a perfect square with side in 2^k. If not, we pad the image with average of the entire image equally around the border. (The original image is in the center of the new image) This helps maintain the distribution of frequencies of the entire image. We tried to preserve frequency distribution because we are dealing with fourier transform.
The function used is
*void add_padding(SDoublePlane &input)*
Which calls in SImage.h
*void paddingval(int max, double val)*
Which modifies the data inside the bare 2-d array by calling in DTwoDimArray.h
*void addval(int max, double val)*

**Part 1.1**
*SDoublePlane fft_magnitude(const SDoublePlane &fft_real, const SDoublePlane &fft_imag)*
This fun takes in the real and imaginary parts of the input image, and creates a spectrogram using the provided formula, which is derived from the complex components of the fourier transform.
It provides us with real positive or negative values, which we scale between 0 to 255 before generating the spectogram.

**Part 1.2**
*SDoublePlane remove_interference(const SDoublePlane &input)*
This function is used specifically to remove the interference noise from the given “noise1.png”. Initially we take the fourier transform of the input image and calculate the log of magnitude of the image’s fft. Again we scale it between 0 to 255 to give us positive real values, with 0 being black and 255 being white.
Using a box of appropriate size to cover the central part of the scaled magnitude, we preserve the actual image frequencies. Then we look for all log(magnitude) values which are greater than 128, which represent the noise frequencies, as observed from the spectrogram of the scaled magnitude (which is also in the folder containing “noise1.png”). Whenever we strike a noise frequency (> 128), we look for the nearest frequency which is <= 128, and replace the noise with it, both in the real domain and the imaginary domain. This is an act of smoothing in the fourier transform.
Once all the noise frequencies are removed, we combine these 2 domains using ifft(inverse fast fourier transform). The received image should be and is noise free, in file “cleaned_noise1.png”

***Something Extra:*** The image generated is grayish, due to it's old image properties and smoothening of frequencies. Thus we ran a small customised(just for this image) contrasting step to generate a much more clear image, in terms of contrast. That we have added as "opt_cleaned_noise1.png". This is not a complete generic process and will only work on grayed images like this one.

**Part 1.3**

*vector<bool> rng(int N, int l)*

This function is used to generate the boolean vector from the watermarking key "N" and the defined length "l".

*SDoublePlane mark_image(const SDoublePlane &input, int N)*

This function is adding the watermark to the real part of fourier transform of the image by generating the boolean vector and using tried and tested values for "l" and "α" using the formula given in the assignment.

The "l"-evenly spaced bins are placed on a circle of radius "r" whose value is carefully derived from "l" to give all those "l" points on integral values of the real fourier grid every single time. We change the "l" points in the real part of fourier transform of the image and then combine it with the imaginary part to give the watermarked image.

SDoublePlane check_image(const SDoublePlane &input, int N)

This function takes in an image and retrieve the real part of the fourier transform of the given image. Then with the help of "l" which depends on (size of the image) we take out the "l" points and save it in a vector.

As we have "N" we can generate the boolean vector using "rng" function.

Then we compare the vectors by generating the correlation coefficient(r).

Then we compare "r" to pre decided threshold "t" and according to it judge if the correct watermark is present or not, or is it a false positive or not.

**Experiment Result:**

**Part 1.1**

As it is formula based without any approximation, it gives us the perfect result in every case (scaled between 0 to 255)

**Part 1.2**

For the given image "noise1.png" we could successfully remove visibly noticeable noise and generate a noise free image "cleaned_noise1.png".

We generated a much more clear and contrasted image "opt_cleaned_noise1.png" using customised values for just this file.

**Part 1.3**

For the decided values of “l”, “α” and “t” we generate almost perfect detection of watermarks, and avoid false positives and wrong or none watermarks, but just for normal images which are not heavily contrasted.

For heavily contrasted images, it generates false positives and sometimes misses the watermark, varying from image to image.

This is because “α” and “t” are independent and not derived from images, on which they depend and should be extracted from. This is our future scope.

**Implementation Details of Part 2:** All the functions for part 2 are implemented in detect.cpp. Executable file is detect. All other codes are pre-existed. To test the function. Just type :

```
>./detect <image.png>
```

to test our car detection program.

Here are some important functions that help achieve the functions:

Function used to crop the sub image:

```
SDoublePlane clip_image(const SDoublePlane &input, int row, int col, int height, int width)
```

Here is a list of functions which we have implemented in detect.cpp:

For functions we used for convolution:

```
SDoublePlane convolve_general(const SDoublePlane &input, const SDoublePlane &filter)
```

Function we used for convolution separable:

```
SDoublePlane convolve_separable(const SDoublePlane &input, const SDoublePlane &row_filter, const
SDoublePlane &col_filter)
```

For functions that used for edge detection:

```
SDoublePlane sobel_gradient_filter(const SDoublePlane &input, bool _gx)
```

For Functions that used for making hough Transformation:

```
SDoublePlane hough(const SDoublePlane &input)
void get_hough_col_info(const SDoublePlane &input)
```

Helper functions to extract features from the Hough Transformation:

```
double get_orth_ratio(const SDoublePlane &input, int angle)
void get_maxes_for_col(const SDoublePlane &input, int
the_col)
int get_hough_max_col(const SDoublePlane &input)
long count_edge_pixels(const SDoublePlane &input)
```

Main function execute the whole process pipeline automatically when the program received the filename of the image.

**Basic strategy for Part 2.** Since implementation began, our basic strategy for car detection has evolved into the following:

- Implement edge detection.
- Implement Hough transformation.
- Use edge detection and Hough transformation to provide salient features for car detection.
- Synthesize the new features into a discriminant function which classifies an image as a car using linear regression.
- When running car detection, use our discriminant function to classify tiles of a given input image.

**Regression features.** We have designed a handful of features for car detection based on intuition and example images provided by the course instructors for use with our car detector. Most of these features involve what we will call “line-iness”, that is, the general manifestation of lines or edges in a certain angle of orientation. Though there certainly exist better means of doing so, we measure this by first smoothing our Hough transformation in the vertical direction, then taking the sum of the absolute values of the derivatives between every two points in a column of the transformation. Using this measure, we implement the following features.

- Max “line-iness” - how strongly lines manifest themselves in the most prominent direction. We do not use our “line-iness” measure directly, but instead as a ratio between the most and least prominent “line-iness” for all angles of a given image clip. We expect that a car will present *fairly* prominent lines.
- Ratio of max “line-iness” to its orthogonal - As we expect the overhead view a car to be relatively rectangular, we anticipate some manifestation of lines in perpendicular directions, though certainly not as much as, say, two crossing parking lot lines.
- Number of edge pixels - As cars generally have lots of edges, we count the edge pixels of each image clip, expecting a clip which represents a car to include plenty of edge pixels.

**Experiment Result:**

Though we were unable to test our results with the ‘a1-eval’ program for precision and recalls, the reason is that though we locate the car in a similar range, we still haven’t hit the coordinate that the text file gives us, which makes this metric not meaningful. Results of car detection appear to be reasonably accurate, which is acceptable for a machine learning algorithm (We could not make precision very high like 100% otherwise it may cause overfitting). When tested on ‘SRSC.png’ and ‘Informatics.png’, our car detection finds a good number of cars while producing relatively few false positives. Results when testing ‘Plaza.png’ are less impressive. This may be partially due to not using “side views”, like those found in ‘Plaza.png’, when training our linear regression model. However, if it was added to the model, maybe it will cause some overfitting issue for new testing image.

**Possibilities for Improvement:**
We haven't make scale for image because we get satisfied result for these images. If we have more testing images, we would try that.

Also we tried to implement some object (template) matching algorithm but the matching algorithm is not as good as expected. Time is too limited to implement an impressive one instead. In some other research, template matching works very well in similar situations. In the later part of the class, if we were taught the knowledge we could consider use it.