# CS 5540 Project 2

Arjun Biddanda (aab227)
Muhammad Khan (mhk98)

March 14, 2013

## 1 Module Design

### 1.1 Language Selection and other Details

For this project, we chose to perform our analyses using OCaml. OCaml, being a functional programming language, offers a simplicity in its syntax so that it is easier to understand. We utilized the capabilities of OCaml to be modular very heavily in our design so that each module performs a given functionality that we require of it. We also chose OCaml because we both enjoyed CS 3110 very much and wanted to pay tribute to it.

#### 1.1.1 Module IO

This is the input/output module that we have designed. It contains the reader type, which is essentially an element which allows us to read a text file. This is essentially the entire function of the IO module, to be able to read any form of textual input that we give it, line by line. In addition to the reader, the large portion of functionality in this module is that of the str_to_float_lst function, which takes in a string of length $n$ consisting of floats that are delimited by spaces. We are able to parse in all of the floats that we require, and output a list of floats which is an easy stream of data that we are able to work with.

#### 1.1.2 Module Parser

The parser module is where we start to make assumptions on the type of data that we are given. We define the parser module as a functor of the IO module that we created above. This means that we are able to use all the functionality of the IO module within the Parser module. There are three things that highlight this module:

1. We have formally defined the type of a signal to be a list of floats, which essentially acts as a time series of readings, where each consecutive index in the list is the next timestep.

2. We define the data as a whole (all of the signals) to be and array $A$ of signals. The array is always of length 40, because that is the number of signals that are given in the data set.

3. Possibly the most important function in the entire project is the parse function. This is what allows us to represent our data from the text file that is given. We use the IO modules string to float list conversion function in order to generate each signal and then input it into our data array. Iteratively this takes $O(n)$ time across all of the inputs. However, the memory requirements are great due to our construction of the data. This helps us along later because of the ease of access that we have to our data.

#### 1.1.3 analysis.ml

The analysis.ml file contains all of our analytic techniques used to investigate this data. These techniques were explained in the writeup and will be further elaborated upon in a later section to discuss the design of these particular algorithms.

## 2 Analysis of Algorithms Used

### 2.1 Zero-cross Algorithm

We created a subfunction called compare which basically compares two elements in order to return a boolean value based on the expression given in the assignment page. We then recursively go through the list, picking the first two elements and calling our compare function on them. If we are able to a have a specific index $t$ return true for the compare function then we add $t$ to an accumulator. At the termination of the algorithm we return the accumulator which will give us all of the time stamps where the signal crosses zero. the algorithm runs in $O(n)$ time where $n$ is the length of the signal that is given to it.

### 2.2 Convolution

The convolution algorithm does exactly what is specified in the writeup. It takes a two given signals and then convolves them. The interesting part to this algorithm is the choosing of combinations of indices of the two signals $f$ and $g$ so that they add up to our convolved signals index. The intuition with OCaml was to do this recursively so this computes all of the possible combinations (in order) of how we can match the index. The final computation is done over a list folding of a list of $n + m - 2$ indices (this ends up being $n + m - 1$ elements) so we have an algorithm that runs in $O(n^3)$ time. Note: This is not as tight of an approximation as we could have.

### 2.3 Harr Transform and Inverse Harr Transform

There are a lot of functions that support the Harr transform function. They are the pairwise_diff and pairwise_avg functions. The defined append function is also helpful

because it is tail-recursive and thus saves us space in the stack. We define our forward Harr Transform function to be recursive over the section that is a pairwise average. This way since the pairwise elements are the first half of the aggregated list, we are able to recurse the entire algorithm only on the first half. This is how the Harr Transform was defined to be computed in the writeup.

The Inverse Harr function was a little bit trickier to implement. We know that we must recursively compute the elements from the first $2^n$ elements of the list. Therefore we created some helpers in order to assist in this function specifically. The first_n function enables us to "grab" the first n elements from a list. This is very important to our recursive case. Within the Inverse Harr function we defined a compute subfunction which given two lists (the first of pairwise averages and the second of the corresponding pairwise differences) which will return the elements that are generated from $p_{avg} \pm p_{diff}$. We then are able to proceed with our discussion of our harr transform function. We have two arguments in our larger helper, the accumulator (which are the computed elements up till that point in the recursion) and the remaining list (which will always be of length power of 2). At every recursive call, we make sure the list is not empty because then we would return our completed signal (accumulator). Otherwise we find the first $l$ elements of the remaining list (where $l$ is the length of our accumulator) and then recursively run our compute subfunction on the accumulator and this sublist of length $l$. We will set this result as new accumulator and then recurse our helper using the new accumlator and the remaining elements in the list.

In our actual implementation of the Harr Transform, we need to consider that the signal needs to be of length $2^n$ where $n$ is maximized. The function reduce_2n in analysis.ml takes a signal and does exactly this reduction. We will use this reduction within our implementation of the Harr-Transform to our data.

### 2.4 Autocorrelation and Correlograms

## 3 Implementation

There are three files that allow for the implementations of these algorithms to the data that was presented to us. These files are appropriately called mean_filter.ml, harr_transform.ml, and autocorrelation.ml. Running certain methods within each file will generate text files within the directory (which are also appropriately named). Then running the provided matlab script taking in the text file name as an argument will show the graphical output of the result that we obtained from our previous computation.

### 3.1 Segmentation of the Data

In order to avoid extremely long parsing time, we have segmented the data into 4 parts, which include 10 signals each. These take the signals in order too. We have called these "CS5540PS2_1.txt", "CS5540PS2_2.txt", "CS5540PS2_3.txt", and "CS5540PS2_14.txt". We have left the original file intact so that if you desire to spend a lot of time parsing you are able to.

### 3.2 Computing Mean Filter

The critical method here is the mean_filter function. This function grabs 3 random signals from the text file that you tell it to take in and then outputs the original signals and their filtered forms to a text file called "meanfilter.txt".

### 3.3 Computing the Harr Transform

There are two important methods in this section of our implementation:

1. The check_harr_inv function allows us to see if there are any discrepancies from the original function if we first Harr Transform that signal to get signal the coefficients $c$ and if we apply the Inverse Harr Transform to $c$ if it would return the original signal. It must be noted though that in order to compute the Harr Transform we need to reduce the length of the signal to some power of 2, so the resultant signal may not be as long as the original in terms of time steps, but will definitely be the same (ideally) up to that point. This data is then output to the file "harr_valid.txt".

2. The second function is harr_vs_filter which compares what the application of a mean filter on a signal can do in order to change its Harr coefficients. One idea must be made clear about the output here. The output are the Harr Coefficients, not signals. The output is stored in the text file called "harr_vs_filter.txt".

### 3.4 Computing the Autocorrelation

There are two methods given in here as well that are of importance are :

1. The function draw_corellograms takes in a text file of signals, picks 3 random ones and outputs the corellograms to a text file.

2. The function corellograms_w_filter compares the correlograms of 3 different signals with and without the mean filter applied to them. Then each of the files are output to a text file called

## 4 Building and Running Instructions

### 4.1 Building

All you need to do is run build.sh in the terminal. Note this project was built on a linux/mac base, so we did not have time to create a windows compatible version. The build script should generate all of the compiled versions of the individual OCaml files as well as a toplevel in which we are now able to run our functions. Now that we have our toplevel generated, we run it in our terminal.

### 4.2 Running

If we want to run any of our comparison functions, we must include the name of the module before we do the specific computation.

In addition to the data generation from our functions, we have a MatLab script which takes a given text file will plot the float lists represented by the text file against each other.