Lab session 3: Signals and Systems

This (last) lab session consists of 4 problems. Problem 1 is worth 10 points, problem 3 is worth 20 points, and the remaining problems are worth 30 points. You get 10 points for free, totalling 100 points.

Problem 1: 1D correlator

The input for this problem are a *template* $h[\,]$, and a discrete input signal $x[\,]$. Your output should be the *steady state* of the signal $y[\,]$ that is obtained by correlating the signal $x[\,]$ with the template $h[\,]$. We assume (in all problems of this lab) that the number of samples of the template h (notation $|h|$) is less than the number of samples of the input x. Recall that the *steady state* of the correlation is defined as:

$$y[d] = \sum_{i=0}^{|h|-1} x[i+d]h[i] \quad \text{for } 0 \leq d < 1 + |x| - |h|$$

On Nestor, you can find the source file `corr1d.c` which computes this correlation. You can use this program as a starting point for your solution. The algorithm in the routine `correlator1D` implements the correlation straight from the definition using nested loops. This is inefficient, and can be improved by using the convolution theorem. Modify the program such that it computes the correlation via the convolution theorem and the version of the FFT that uses prime numbers (which is actually calledf the NTT=Number Theoretic Transform). You can find a reference implementation of the NTT (which is the solution of exercise 5 of lab 2) on Nestor in the file `ntt.c`. You may assume that the input signal x and the template h do not contain negative numbers.

Example 1:

input:

```
3: [2,3,5]
10: [1,2,3,5,4,420,1,12,13,15]
```

output:

```
8: [23,38,41,2122,1273,903,103,138]
```

Example 2:

input:

```
3: [17,2,19]
10: [1,6,12,11,19,4,21,12,3,9]
```

output:

```
8: [257,335,587,301,730,338,438,381]
```

Example 3:

input:

```
2: [1,42]
10: [1,42,1,42,1,42,10,52,10,52]
```

output:

```
9: [1765,84,1765,84,1765,462,2194,472,2194]
```

Problem 2: 1D Pearson correlator

As you can see from the results in problem 1, direct correlation is usually not very useful. We can see that clearly in the first example input. The pattern `[2,3,5]` is present in the signal at index location 1, but the correlation value at index 1 is only 38 (which is not the maximum of the correlation series). Moreover, the pattern is also present at the end of the input signal in the sequence `[12,13,15]` but a constant value (10) is added to it. A good template matcher should be insensitive for this, and therefore we rarely use direct correlation and apply *Pearson correlation* instead. The Pearson correlation takes on values between (and including) $+1$ and -1, where 1 is total positive correlation (perfect match), 0 is no correlation, and -1 is total negative correlation (perfect correlation, but minima are maxima and vice versa). The Pearson correlation is defined as (see `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`):

$$y[d] = \frac{\sum_{i=0}^{|h|-1}(x[i+d] - \bar{x})(h[i] - \bar{h})}{\sqrt{\sum_{i=0}^{|h|-1}(x[i+d] - \bar{x})^2}\sqrt{\sum_{i=0}^{|h|-1}(h[i] - \bar{h})^2}}$$

Again, we assume x to be the signal, and h to be a (smaller) template. In the above formula, the sums therefore range from 0 to $|h| - 1$. Moreover, the notation $\bar{h}$ denotes the mean value of h (which is a constant for any d since

we compute only the steady state), and $\bar{x}$ is the mean value of the samples from x that overlap with h given a value for d (and thus is not constant!).
Extend the code for problem 1 with a function `pearsonCorrelator1D`, which computes first a correlation (using `correlator1D` and next corrects the result such that we get the steady state of a Pearson Correlation. This involves some calculus (see for example the above wikipedia link). Note that the output of the direct correlation is an array of integers, so you need to copy this result into an array of `doubles` before you start this correction. The input/output behaviour of your program must be the same as in problem 1, except that the output must be Pearson correlation values, rounded to 5 digits after the decimal dot.

Example 1:

input:

```
3: [2,3,5]
10: [1,2,3,5,4,420,1,12,13,15]
```

output:

```
8: [0.98198,1.00000,0.32733,0.94423,-0.19509,-0.74065,0.80296,1.00000]
```

Example 2:

input:

```
3: [17,2,19]
10: [1,6,12,11,19,4,21,12,3,9]
```

output:

```
8: [0.15959,-0.54127,0.67900,-0.92966,1.00000,-0.82675,-0.10762,0.90419]
```

Example 3:

input:

```
2: [1,42]
10: [1,42,1,42,1,42,10,52,10,52]
```

output:

```
9: [1.00000,-1.00000,1.00000,-1.00000,1.00000,-1.00000,1.00000,-1.00000,1.00000]
```

Problem 3: 1D template matching

The input for this problem has almost the same format as in the previous two exercises. The difference lies in the first line, which contains a floating point number T (threshold) which has a value from the range $-1 \leq T \leq 1$. The remaining two lines contains a template h and a signal x (both with positive numbers). The output of this problem should be a list of index locations where the Pearson Correlation is greater or equal to T, and the correlation value itself. For each location you should print one line of output.

Example 1:

input:

```
1.0
3: [2,3,5]
10: [1,2,3,5,4,420,1,12,13,15]
```

output:

```
1 1.00000
7 1.00000
```

Example 2:

input:

```
0.9
3: [17,2,19]
10: [1,6,12,11,19,4,21,12,3,9]
```

output:

```
4 1.00000
7 0.90419
```

Example 3:

input:

```
0.9
2: [1,42]
10: [1,42,1,42,1,42,10,52,10,52]
```
output:
```
0 1.00000
2 1.00000
4 1.00000
6 1.00000
8 1.00000
```

Problem 4: mini OCR (Optical Character Recognition) using a 2D Pearson correlator

In this problem you will perform Pearson correlation on digital gray scale images. The input for this problem are a PGM (simple image format) image, and a small template image (also a PGM image). Most of the code has been written for you, and can be found in the file `pearson2D.c` on Nestor.

The input image that we will use is called `emmius.pgm`. It is taken from of a scan of a page from an old book (from 1614) written by the founder of the university of Groningen (Ubbo Emmius). The template is a letter (the letter 'm') from this page, and is called `M.pgm`. These two file names are hard-coded in the code of this exercise. Of course, you normally would not do that, but in this case it is necessary to pass the test in Themis. So, do not change these file names! You can display these images using the Linux command `display`.

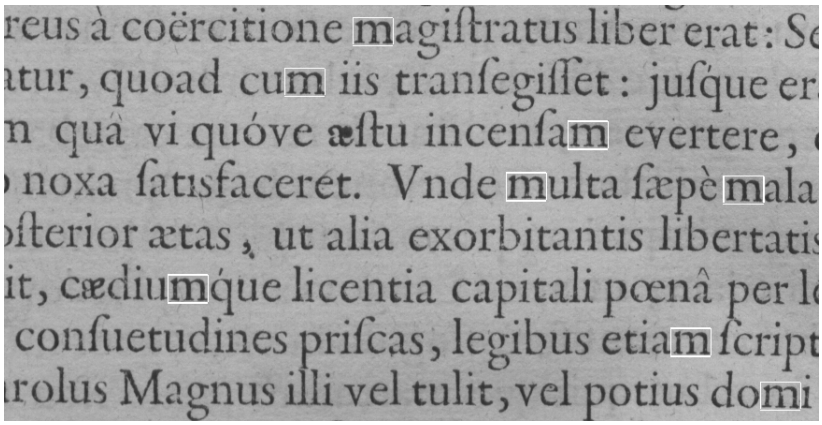
The object of this exercise is to count (and of course, not by hand!) how many times the letter 'm' occurs on the page. The number of occurrences must be printed on the output. Note that this problem has only one test case in Themis. It is considered cheating (we check this!) to submit a program that only prints a number without calculation.

The file `pearson2D.c` is a completely functional program, except for the routine `fft2D`, which should compute the 2-dimensional FFT of an image. In the file you will see that the code of this routine looks like:

```
void fft2D(int direction, int width, int height, double **re, double **im) {
  /* width and height must both be powers of two ! */
  /* Note that the parameters re and im denotes the REal part and
   * the IMaginary part of the 2D image.
   */
  printf("fft2D: YOU HAVE TO IMPLEMENT THE BODY OF THIS FUNCTION YOURSELF\n");
  printf("MAKE USE OF THE fft1D() FUNCTION\n");
  exit(0);
}
```

Note that in this problem we use the classical FFT (not the NTT, although it would work equally well). A nice property of the 2D FFT is that it can be implemented using only 1D FFTs. The 2D FFT can be computed by performing only 1D FFTs in two passes. In the first pass, we perform a 1D FFT on each row (real to complex) of the image. In the second pass, we perform a 1D FFT on each column resulting from the first pass (complex to complex). The same holds for the inverse 2D FFT. The one-dimensional FFTs have been implemented for you, and are already present in the file `pearson2D.c`. Note that this implementation of the 1D FFT is not recursive, but that is not important. This version is slightly more efficient than the one presented in the lecture (but also a bit faster).

Once you implemented the function `fft2D` correctly, you can run the program. The output is the number of occurrences of the letter 'm'. Moreover, as a side-product, the program produces the files `matches.pgm`. Use the program `display` to see the result. It should look like the following image:


reus à coërcitione magiſtratus liber erat: Se
atur, quoad cum iis tranſegiſſet: juſque er
n quâ vi quóve æſtu incenſam evertere,
noxa ſatisfaceret. Vnde multa ſæpè mala
oſterior ætas, ut alia exorbitantis libertatis
it, cædiumque licentia capitali pœnâ per le
conſuetudines priſcas, legibus etiam ſcript
rolus Magnus illi vel tulit, vel potius domi