Assignment 9: Digital Signal Processing

This homework assignment, worth 20 marks and 5% of your final grade, is due by 4:00 PM on Friday, Nov. 28th. Submit your solution (a printed and stapled/bound copy of `Edit.cpp`) to the ECE 312 LEC A1 assignment box outside the ECERF reception (2nd floor). Include your name and your student ID, the percentage of the submission (excluding code and ideas given with the assignment) original by you, and the names of all other contributors to your submission (indicate the percent taken from each).

Electronic and wrong-box submissions will not be accepted. If any part of the assignment is submitted late, the whole assignment will be penalized 2 marks, i.e., 10% of the maximum mark, per business day. Submissions received, in whole or in part, after 4:00 PM on Tuesday, Dec. 2nd, will receive 0 marks. Do not cheat, do not plagiarize, do not misrepresent facts, and do not participate in an offence!

Objectives

Digital signal processing (DSP) is a major application area for embedded systems. Unlike analog signal processing, DSP is largely defined by software, running on a microprocessor that may be embedded between an analog-to-digital and/or a digital-to-analog converter (ADC and/or DAC). In this assignment, you will program, document, and test a zero-crossing detector, with hysteresis for robustness, that also counts the number of samples between zero crossings. This DSP will employ state-space models.

Getting Started

In Visual Studio (Express) 2012 or 2013, create a new one-project solution with `Main.cpp` and `Edit.cpp` as source files. Put `Input.txt` in the project's working directory. Set it and `Output.txt` as input and output files, respectively, as follows. Right click on your project in the Solution Explorer and select Properties. Choose Configuration Properties ≫ Debugging in the left pane. In the right pane, by Command Arguments, enter "`Input.txt Output.txt`" without quotes and click on OK.

Build and run the project. Verify that it creates an `Output.txt` file successfully. If you cannot build and run the code, try the ETL E5-012 lab, where Visual Studio 2012 is available on Linux via VMware Player. Visual Studio 2012 may also be available on the Windows PCs in the ETL E4-013 lab.

Technical Specifications

Write/edit the `zeroCrossCounter` function (`Edit.cpp`). Although some code is given there for Getting Started, you may delete it. Do not change any other code, including the `zeroCrossCounter` function interface. Do not add `include` statements or modify existing ones. Your entire solution should be programmed into the `cpp` file. You may add other functions if that will help you develop or refine a solution. Do not use any global variables. You may use static local variables if they prove helpful.

At each moment, the `zeroCrossCounter` function takes two integers, `input` (e.g., from an ADC) and `thresh`, as inputs (passed by value), and gives one integer as an output (`return` value). Let n denote the moment, let u_{in} and u_{th} denote the inputs, respectively, and let y_{ZCC} denote the output. The system, i.e., the DSP, is then defined by three nonlinear time-invariant subsystems connected in sequence:

$$(1)\quad y_{\text{sgn}}[n] = h_{\text{sgn}}(u_{\text{in}}[n], u_{\text{th}}[n])$$
$$(2)\quad y_{\text{ZC}}[n] = h_{\text{ZC}}(y_{\text{sgn}}[n], x_{\text{ZC}}[n])$$
$$x_{\text{ZC}}[n+1] = f_{\text{ZC}}(y_{\text{sgn}}[n], x_{\text{ZC}}[n])$$
$$(3)\quad y_{\text{ZCC}}[n] = h_{\text{C}}(y_{\text{ZC}}[n], x_{\text{C}}[n])$$
$$x_{\text{C}}[n+1] = f_{\text{C}}(y_{\text{ZC}}[n], x_{\text{C}}[n])$$

The first subsystem, with two inputs and no states, implements a sign function with threshold:

$$h_{\text{sgn}}(u_{\text{in}}, u_{\text{th}}) = \begin{cases} 1, & u_{\text{in}} > |u_{\text{th}}| \\ -1, & u_{\text{in}} < -|u_{\text{th}}| \\ 0, & \text{otherwise} \end{cases}$$

The second subsystem, with one state, implements a zero-crossing detector with hysteresis:

$$h_{\text{ZC}}(u_{\text{ZC}}, x_{\text{ZC}}) = \begin{cases} 1, & u_{\text{ZC}} x_{\text{ZC}} < 0 \\ 0, & \text{otherwise} \end{cases}$$
$$f_{\text{ZC}}(u_{\text{ZC}}, x_{\text{ZC}}) = \begin{cases} 1, & u_{\text{ZC}} > 0 \\ -1, & u_{\text{ZC}} < 0 \\ x_{\text{ZC}}, & \text{otherwise} \end{cases}$$

The third subsystem, also with one state, implements a counter with nonzero trigger:

$$h_{\text{C}}(u_{\text{C}}, x_{\text{C}}) = \begin{cases} x_{\text{C}} + 1, & u_{\text{C}} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$
$$f_{\text{C}}(u_{\text{C}}, x_{\text{C}}) = \begin{cases} 0, & u_{\text{C}} \neq 0 \\ x_{\text{C}} + 1, & \text{otherwise} \end{cases}$$

Finally, as explained in Notes and Hints below, initial states are taken to be zero.

Presentation Specifications

Format your code professionally. Compare your placement of curly braces, i.e., `{` and `}`, to that of the given code. Use a similar approach or another professional approach. Code of the same code block should be left aligned and the indentation should consistently match the nesting level. In Visual Studio, selecting EDIT ≫ Advanced ≫ Format Document from the menu may correct alignment errors.

Ensure that long indented lines do not overflow without indentation when printed. Either break them into multiple indented lines (the compiler ignores line breaks) or print in landscape mode.

Write suitable comment headers for all functions you write/edit, which clearly describe the purpose, inputs, and outputs of each function. Functions that use static local and/or global variables are not fully defined this way. Describe, in the comment header, the state-driven behaviour of functions that use such variables. This is one good reason to minimize/simplify the use of state variables.

Review your working solution and look for ways to simplify it so that it would be easier to follow. For example, use refactoring to reduce code smell. This is another good reason to minimize/simplify the use of state variables. Whereas short variable names are common in math, descriptive variable names are preferred in programs. Finally, give helpful comments beyond those in the comment header.

Notes and Hints

While DSP is often introduced using convolution and difference equation models, applicability of these representations to nonlinear and/or time-varying cases is limited. State-space models, at the heart of advanced signal processing tools such as Simulink, have no such restrictions. For example, consider a (non)linear time-invariant state-space model with one input u, one state x, and one output y:

$$y[n] = h(u[n], x[n])$$
$$x[n+1] = f(u[n], x[n])$$

The square-bracket notation above means u, x, and y are functions of the integer n, which represents discrete time. By convention, $n = 0$ represents the moment when the DSP begins. Unless otherwise specified, the initial state, $x[0]$, is assumed to be zero. Finally, h and f are (non)linear functions.

Do *not* submit the output file, `Output.txt`, generated by your code, which corresponds to the given input file, `Input.txt`. Your code is marked against the specifications, regardless of the output. However, your solution has (a) technical error(s) if your output does not match the given one. Debug and correct the error(s). If your output does match, your solution may still have (a) technical error(s). Reread the Technical Specifications and verify that you have met all requirements exactly.

(Whereas the input file has only one sample each of u_{in} and u_{th} per line, the output file has up to ten samples of y_{ZCC} per line, to save space. This behaviour is programmed into `Main.cpp`.)