COE3DY4 Lab #2
DSP Primitives in C++

Objective

The purpose of this lab is to provide a link between the modeling of (DSP) primitives used in Software Defined Radios (SDRs) in an interpreted and scripting language like Python to their implementation in C++.

Preparation

- Revise the material from lab 1
- Revise the fundamental programming concepts from C
- Start getting familiar with C++ libraries at https://en.cppreference.com/ and https://www.cplusplus.com/

C/C++ Background

This lab is not meant to be an introduction to C++. Any computer language used for programming, description, analysis, ..., is best learned by applying it to a real-life project that is a good fit for it! To provide the context for using C++ for the 3DY4 project, an historical perspective on C/C++ will be provided before the in-lab/take-home experiments are elaborated.

The origins of both C and C++ programming languages can be traced back to projects in the 70s at Bell Labs. C is a procedural language that was developed to support and extend the Unix operating system in early 70s. Gradually it has gained popularity beyond research labs and academia, and today it is one of the most widely-used programming languages, in particular in the embedded space. This is because it is a mid-level programming language since it can both effectively interact with digital hardware abstractions, as well as build large applications using a broad set of libraries. On the one hand, its low-level features can make the most out of the available hardware, on many occasions as efficiently as hand-crafted assembly code due to its possibility to do "bare metal" system programming. On the other hand, its high-level features, e.g., portability, modularity, extensibility, ..., can support the development of applications ranging from video processing to operating systems, e.g., Linux. For example, the first version of the drivers for a new hardware prototype are written in a low-level language, which is most commonly C.

While keeping the above in mind, it is important nonetheless to emphasize that the low-level features of C, which enable it to produce high-performance code, can also be perceived by some as its potential weakness. For example, the possible misuse of data type conversions or lack of explicit support for data hiding can lead to security vulnerabilities. Not supporting object-oriented programming (OOP) natively might also cause longer development times and complicate the code

maintenance. Nevertheless, `C` has served as the basis and/or inspiration for many other powerful modern programming languages that aim to preserve some of its benefits and address some of its limitations. This includes `C++`, an OOP language developed in late 70s, which maintains most of the efficiency of `C`'s low-level features, while further adding high-level features for program organization and faster code development. While there is always a performance overhead associated with the high-level features in any programming language, the advancements in the compiler technology and the rich set of optimized libraries make `C++`-generated code very competitive in many application areas. Whenever the performance overhead over `C` is acceptable, it is extremely beneficial to leverage `C++`'s extensive and well-supported libraries, e.g., the Standard Template Library (STL). This not only reduces the development time over a `C`-only implementation, but it can also mitigate the types of problems that are very time-consuming to deal with in `C`, e.g., memory management, memory leaks, memory corruption, ...

Based on the above reasoning, by leveraging, for example, the container classes or threading libraries from `C++`, the design and implementation effort during the 3DY4 project can prioritize implementation of high-performance/real-time DSP algorithms rather than spending unnecessary (and extensively large amounts of) time on dealing with implementation-induced bugs that originate from low-level memory management in `C`, many of which manifest on one computer platform but not on another one. In other words, to summarize, considering the nature of the SDR application, the target hardware and the project timeline, the choice of `C++` is driven by the expectation to save development and implementation time, while still expecting an optimized machine code that is generated from a code base that can be re-purposed with a reasonable effort to other hardware platforms.

Implementing DSP primitives in `C++`

The DSP concepts from this lab are the ones from the previous lab, i.e., the Discrete Fourier Transform (DFT), deriving the impulse response coefficients using the *sinc* function, digital filtering via convolution, ... While implementing them and visualizing their impact in a different programming language will further consolidate your conceptual understanding, their implementation in `C++` will also make you comfortable with a language that is compiled directly into machine code (rather than interpreted as it is the case for `Python`) and some of the libraries built-into `C++`.

The source code is organized in three different `C++` files, that follow the labels from the previous lab. Since this our first lab using `C++` the program files for different experiments are self-contained and they can be compiled independently from each other; note, although some of the functions should be manually moved between files to be reused from one experiment to another, this a convention that will **NOT** be followed during the project; nonetheless it was deemed as a necessary "relaxation" during the acclimatization to `C++`.

A key aim of the **first** experiment using `fourierTransform.cpp` is to get you used to the `vector` container class from `C++`'s STL. Vectors have the benefits of arrays, i.e., contiguous storage locations for their elements, which can be accessed via indices, i.e., offsets from the base address of the vector, that are further extended with useful features like dynamic re-sizing. Naturally, there is an overhead that comes with dynamic vector changes (both in terms of storage and compute time), however this can be outweighed by the benefits of avoiding explicit memory management, as done in `C` with `malloc` and `free`. Note, as important as writing your own memory management routines is for some applications (especially for systems software), for our SDR project using the `vector` container is a trade-off worth accepting. The detailed documentation for the `vector` container class can be

found at `https://www.cplusplus.com/reference/vector/vector/`. Note also, understanding how to use the methods from a class via its online documentation is an approach that you will be using throughout the course, and beyond.

- The start-up code from `fourierTransform.cpp` gives you some basic functionality in terms of generating random numbers, manipulating vectors and execution time measurement. Your main learning objective throughout this experiment (and this lab in general) is to get you comfortable to write `C++` functions for DSP that use the `vector` container class. Therefore, your **in-lab** task for this experiment is to implement the inverse DFT (IDFT) and confirm that it works correctly. This can be achieved by comparing if the absolute value of each element from a vector x equals the magnitude of the corresponding element from IDFT(DFT(x)) (within a small tolerance range, e.g., less than 10^{-3}).

- The purpose of the **second** experiment using `filterDesign.cpp` is to get you more proficient with writing DSP primitives in `C++` for digital filtering. As part of your **in-lab** work, you are asked to implement in `C++` the pseudocode that you have already written in `Python` in the previous lab for both the generation of the filter coefficients, i.e., the impulse response for a low-pass filter, as well as for performing the digital filtering, i.e., through the convolution of the impulse response with an input sequence (single-pass convolution only for this experiment). In the reference code you are already given functions for generating sines, adding them and dumping internal data in raw text files to be used for plotting. It is very important to emphasize that visualizing waves, spectra, ..., will be a regular part of bringing-up, troubleshooting, improving, ... your project software. A common practice is to dump internal data in log files and use third-party tools for visualization. Due to the nature of the problem, there is an inherent flexibility when choosing a third-party tool. The reference code is set up to write multiple vectors in a human readable format and use `gnuplot`, which is a widely used tool in the open source community. A sample `.gnuplot` script with some basic features needed at this time is provided. If you choose to continue with `gnuplot` as part of the project, more details can be found at `http://www.gnuplot.info/`.

  As a side note, it is worth mentioning that during the project it will be at the discretion of each group to choose their preferred method for third-party visualization, e.g., you can even import the data that is logged by the machine code generated from `C++` back into `Python` and use `matplotlib`. An analogy to the physical world is that using oscilloscopes (or simply scopes) is critical during system development and implementation. However, scopes will not be a part of the final product itself. For this reason, there are always decisions to be made when choosing a scope based on different criteria: project needs, performance targets, budgets, preferences ... and one of the reasons for using `gnuplot` in this experiment is to make you aware of alternatives that should still produce a visual output of your experiments that confirm the underlying phenomena in the same way as when using tools that you have been exposed to before.

- In the **third** experiment that uses `blockProcessing.cpp` and `wavio.py` you will use your own DSP primitives written in `C++` and apply them in the context of a more realistic application, i.e., audio filtering. Your **in-lab** work is concerned with integrating the code for impulse response generation and convolution for digital filtering from the previous experiment to process audio data in a single-pass.

  An important point worth making is that, since working with headers of `.wav` files can be a cumbersome task of additional (and unnecessary) workload in the context of this course,

you are provided with a `Python` script (`wavio.py`); this script can be used to extract raw audio data from `.wav` files and write it to a binary file; likewise it can be used to read raw audio data from a binary file to produce a `.wav` file that can be used by third-party audio players. The default code assumes a 32-bit floating point format for the raw audio samples that are passed between `C++` and `Python`. This assumption might seem to be too extreme for audio applications; however, as it will become more clear in the forthcoming lab, the samples from the radio-frequency (RF) front-end will be translated to a 32-bit floating point format; hence, the decision to use a 32-bit float representation for raw binary data as soon as in this lab. How to adapt the `Python` script and the `C++` code in order to pass binary files back and forth should be self-explanatory using the comments from the start-up code.

Take-home exercises

In addition to **completing** *and* **submitting** your in-lab experiments, to further improve your understanding of DSP, as well as to become more proficient with `C++`, you should perform the following:

- Using the existing method for measuring the execution time from `fourierTransform.cpp`, measure how long does it take to run a DFT/IDFT pair as the number of samples N in the random array increases by a factor of 2 from 2^7 to 2^{14} (a total of 8 measurement points). Interpret the runtimes and relate them to the big $\mathcal{O}$ analysis of the computational time complexity of the DFT/IDFT algorithms.
- After you have completed the in-lab requirements for `filterDesign.cpp` write a function to generate a square wave with a duty cycle of 50%; assume the frequency of the square wave is a random value ranging from 5 to 20 Hz that changes from one run to another; similarly, the dynamic range of the square wave is from -A to A, where A is another random value from 3 to 7 that varies from one run to another. To validate that your low-pass filter coefficients for the impulse response and the single-pass convolution for digital filtering have been properly implemented, you should extract the first harmonic of the square wave through low-pass filtering. Use `gnuplot` for visualization of both time domain signals for the square wave and its (extracted) first harmonic, as well as the magnitude spectrum in the frequency domain.
- In `blockProcessing.cpp` provide support to process an audio file in blocks rather than filtering all the data in a single pass. Assume the block size is a parameter (that is passed as an argument to your filtering function) that is larger than the number of filter taps. Note, conceptually this is the same task as one of the take-home exercises from the previous lab, however you will need to refactor your Python code to `C++` and integrate it into `blockProcessing.cpp` to assess it.

Your report should have three sections, one for each of the above items. One page is sufficient and it should not be exceeded unless there are out-of-ordinary points to be made.

Your sources should stay in the `src` sub-folder of the GitHub repo; your report (in .pdf, .txt or .md format) should be included in the `doc` sub-folder.

Your submission, which is to be done via a push to the master branch of your group's GitHub repo for this lab, is due 14 hours before your next lab session is scheduled. Late submissions will be penalized.

This lab is worth 5 % of your total grade.