COE3DY4 Lab #2
DSP Primitives in C++

Objective

The purpose of this lab is to establish a connection between the modeling of Digital Signal Processing (DSP) primitives used in Software Defined Radios (SDRs) in an interpreted scripting language like `Python` and their efficient implementation in `C++`.

Preparation

- Review the material covered in Lab 1
- Refresh fundamental programming concepts in `C`, focusing on memory management, pointers, and data structures
- Familiarize yourself with `C++` libraries by exploring `https://en.cppreference.com/` and `https://www.cplusplus.com/`

`C/C++` Background

This lab is not intended to serve as an introduction to `C++`. Any programming language, whether used for development, description, or analysis, is best learned by applying it to a real-world project that is a good fit for it!

To provide context for using `C++` in the 3DY4 project, a historical perspective on `C/C++` will be presented before elaborating on the in-lab experiments. The origins of both `C` and `C++` programming languages can be traced to projects in the 1970s at Bell Labs. `C` is a procedural programming language developed to support and extend the `Unix` operating system in the early 1970s. Over time, it gained popularity beyond research labs and academia and is now one of the most widely used programming languages, particularly in the embedded systems domain. This popularity stems from its status as a mid-level programming language, capable of interacting directly with digital hardware abstractions while also supporting the development of large-scale applications through its extensive set of libraries.

On one hand, `C`'s low-level features enable highly efficient use of hardware, often approaching the performance of hand-crafted assembly code, making it ideal for "bare metal" system programming. On the other hand, its high-level features—such as portability, modularity, and extensibility—allow it to be used for applications ranging from video processing to operating systems, such as `Linux`. For example, the initial versions of drivers for new hardware prototypes are often written in languages like `C`.

While keeping the above in mind, it is important to emphasize that the low-level features of `C`, which enable it to produce high-performance code, can also be perceived by some as potential weaknesses. For example, the possible misuse of data type conversions or the lack of explicit support for data hiding can lead to security vulnerabilities. Additionally, the absence of native support for object-oriented programming (OOP) may result in longer development times and complicate code maintenance. Nevertheless, `C` has served as the basis and inspiration for many modern programming languages that aim to preserve its benefits while addressing its limitations. One such language is `C++`, an OOP language developed in the late 1970s, which retains much of `C`'s efficiency in low-level features while introducing high-level constructs for better program organization and faster development.

Although there is always some performance overhead associated with high-level features in any programming language, advancements in compiler technology and the availability of rich, optimized libraries make `C++`-generated code highly competitive in many application domains. Whenever the performance overhead relative to `C` is acceptable, it is highly advantageous to utilize `C++`'s extensive and well-supported libraries, such as the Standard Template Library (STL). This not only reduces development time compared to a `C`-only implementation but also helps mitigate common issues in `C`, such as memory management, memory leaks, and memory corruption.

Based on the above reasoning, by leveraging, for example, the container classes or threading libraries provided by `C++`, the design and implementation effort during the 3DY4 project can focus on developing high-performance and real-time DSP algorithms, rather than spending extensive time dealing with implementation-induced bugs arising from low-level memory management in `C`. Many of these bugs, such as memory leaks or undefined behavior, often manifest inconsistently across different computer platforms, complicating debugging and testing efforts.

In summary, considering the nature of the SDR application, the target hardware, and the project timeline, the choice of `C++` is driven by the expectation of saving development and implementation time while still producing optimized machine code. Furthermore, the resulting code base is designed to be portable and can be re-purposed with reasonable effort for deployment on other hardware platforms.

Using the `std::vector` Container and STL Methods in `C++` for DSP Primitives and Applications

The DSP concepts in this lab are the same as those introduced in the first lab. Implementing the Discrete Fourier Transform (DFT), impulse response generation, and digital filtering via convolution in a different programming language will further reinforce your understanding. Additionally, implementing these concepts in `C++` introduces you to a compiled language that translates code directly into machine instructions, in contrast to the interpreted nature of `Python`.

An important aim of the in-lab experiments is to familiarize you with the `vector` container class from `C++`'s STL. Vectors provide the benefits of arrays, such as contiguous storage locations for their elements, which can be accessed via indices (i.e., offsets from the base address of the vector). While dynamic resizing introduces some overhead in terms of storage and compute time, it eliminates the need for explicit memory management, as required in `C` with `malloc` and `free`. Below are a few relevant features of the `vector` container from STL.

- **Dynamic memory management and flexible size adjustment:** The `vector` container dynamically manages memory, allowing it to grow or shrink as needed. Key methods include:
  - `size()`: Returns the current number of elements stored in the vector.

  - `resize(size, value)`: Changes the size of the vector to `size`. If the new size is larger than the current size, only new elements are initialized to `value`. If the new size is smaller, elements at the end of the vector are removed. Frequent resizing can introduce overhead due to reallocations. To mitigate this, `reserve()` can be used to preallocate storage in advance for a specified capacity.
  - `clear()`: Removes all elements, setting the size to 0, but retains the allocated capacity.
  - `push_back(value)`: Appends `value` to the end, increasing the size by 1. If the new size exceeds the current capacity, the vector reallocates to a larger memory block and copies existing elements into the new block. This reallocation can impact performance.

- **Efficient element access and iteration:** Elements in a `vector` can be accessed in constant time using `[]` and iterated over efficiently using indices:

  ```
  for (int i = 0; i < (int)vec.size(); i++) { vec[i] = ... }
  ```

  Note that bounds are not checked with `[]`, so invalid indices may lead to undefined behavior.

- **Nested vectors for multi-dimensional data:** Multi-dimensional structures can be created by nesting `vector` objects. For example:

  ```
  std::vector<std::vector<float>> array_of_vectors;
  array_of_vectors.push_back({1.0, 2.0, 3.0});
  ```

  This approach allows you to represent multi-dimensional data of variable size.

- **Seamless integration with STL algorithms:** The `vector` container integrates seamlessly with STL algorithms. For example, `std::copy` can be used to concatenate two vectors into a new vector:

  ```
  std::vector<float> a = {1.0, 2.0, 3.0};
  std::vector<float> b = {4.0, 5.0, 6.0};
  std::vector<float> result(a.size() + b.size());

  std::copy(a.begin(), a.end(), result.begin());
  std::copy(b.begin(), b.end(), result.begin() + a.size());
  ```

  In this example, the resulting vector `result` is allocated its own memory, ensuring that the original vectors `a` and `b` remain unchanged. Depending on your requirements, the STL provides a variety of additional algorithms that operate efficiently on `vector` containers.

Detailed documentation for the `vector` container class can be found at `https://www.cplusplus.com/reference/vector/vector/`. Understanding the usage of STL containers and their methods is not only important for the project in this course but also foundational for building efficient and maintainable `C++` programs.

The source code for this lab is organized into three separate `C++` files corresponding to the labels from the previous lab. Since this is an entry-level lab for `C++`, the program files for different experiments are self-contained and can be compiled independently. Note that while some functions need to be manually moved between files to enable reuse across experiments, this is a temporary measure to ease your acclimatization to `C++` and STL. From the next lab onwards, code reuse will be handled through proper modularization.

Discrete Fourier Transform

The start-up code in `fourierTransform.cpp` provides basic functionality for generating random numbers, manipulating vectors, and measuring execution time. The DFT function is also included. As part of your **in-lab** task, you will implement the inverse DFT (IDFT) and verify its correctness. This can be done by checking whether the absolute value of each element in a vector x matches the magnitude of the corresponding element in IDFT(DFT(x)) within a small tolerance range, such as 10^{-3}. A unit testing function has already been set up for you.

An important aspect of this lab is understanding the role of numerical precision in computations. This is demonstrated through the use of conditional compilation with macros, which allow you to switch between single-precision (`float`) and double-precision (`double`) representations of real numbers. By leveraging macros, you can control precision levels during compilation. Consider the following code snippet:

```
#ifdef DOUBLE
typedef double real;
#else
typedef float real;
#endif
```

The `DOUBLE` macro can be defined directly in the source code:

```
#define DOUBLE
```

Alternatively, the macro can be defined during compilation using the `-D` option in `g++`. For example:

```
g++ -DDOUBLE -o my_executable fourierTransform.cpp
```

This approach provides flexibility, enabling you to toggle between `float` and `double` without editing the source code. The choice between these two types involves a trade-off between memory usage and computational precision:

Type	**Storage (Bytes)**	**Mantissa (Bits)**	**Exponent (Bits)**	**Precision (Digits)**	**Largest Value (Approx.)**
`float`	4	23	8	6-7	3.4×10^{38}
`double`	8	52	11	15-16	1.8×10^{308}

Table 1: Comparison of `float` and `double` properties, including storage, mantissa, exponent, precision, and largest value.

For certain DSP primitives, such as the DFT with a high number of frequency bins, using `double` can improve numerical precision. However, this comes at the cost of doubling the memory requirements compared to `float`, which may be infeasible or affect memory access performance in memory-constrained environments.

As part of this lab, after successfully implementing the IDFT and passing the unit tests with a tolerance of 10^{-3}, experiment with running the unit tests under finer tolerance levels. Observe how the results of the unit tests differ when `real` is defined as `double` versus `float`. This experiment is designed to help you understand the practical implications of precision in numerical computations and the trade-offs involved in DSP applications.

Digital Filter Design

The purpose of the **second** experiment using `filterDesign.cpp` is to help you become more proficient in writing DSP primitives in `C++` for digital filtering. As part of your **in-lab** work, you will implement in `C++` the pseudocode that you previously developed in `Python` during the last lab. This includes generating the filter coefficients (i.e., the impulse response for a low-pass filter) and performing digital filtering via the convolution of the impulse response with an input sequence (single-pass convolution for this experiment only).

The reference code provides functions for generating sine waves, combining them, and dumping internal data into text files for plotting. Visualizing waves, spectra, and other relevant data is a critical part of debugging, troubleshooting, and improving your project software. In DSP, it is common practice to log internal data to files and use third-party tools for visualization. While there is no restriction on the choice of visualization tools, the reference code is configured to write multiple vectors in a human-readable format and leverages `gnuplot`, a widely used open-source tool. A sample `.gnuplot` script with essential features for this experiment is provided in the `data` subfolder. Additional details about the file format used by `gnuplot` in this experiment can be found in the Appendix. Detailed documentation for `gnuplot` is available at `http://www.gnuplot.info/`.

As a side note, during the project, each group will have the discretion to choose their preferred method for third-party visualization. For instance, you may import data logged by the `C++` program back into `Python` and use `matplotlib` for visualization. An analogy to the physical world is the use of oscilloscopes (or simply scopes) during system development and debugging. Scopes are critical for testing and validating system behavior, but they are not part of the final product itself. Similarly, the choice of visualization tools will depend on your project needs, performance targets, budget, and personal preferences. One of the reasons for introducing `gnuplot` in this experiment is to make you aware of alternative tools that can produce meaningful visual outputs, helping you confirm the underlying phenomena, much like the measurement tools you have used physical lab.

Digital Filtering of Data Streams Divided into Blocks

In the **third** experiment, which uses `blockProcessing.cpp` and `wavio.py`, you will apply your own DSP primitives written in `C++` to a more realistic application: audio filtering. Your **in-lab** task involves integrating the impulse response generation and convolution code from the previous experiment to process audio data in a single pass.

To simplify working with `.wav` file headers — an otherwise cumbersome and unnecessary task in the context of this course — you are provided with a `Python` script (`wavio.py`). This script extracts raw audio data from `.wav` files and writes it to a binary file. Similarly, it can read raw audio data from a binary file and produce a `.wav` file that is compatible with third-party audio players.

The default implementation assumes a 32-bit floating-point format for raw audio samples exchanged between `Python` and `C++`. While this may seem excessive for standard audio applications, it aligns with the real format (`float` or `double`) that will be used in subsequent labs to represent samples from the radio-frequency (RF) front-end. This decision ensures consistency and prepares you for future experiments. Comments in the start-up code provide guidance on adapting both the `Python` script and the `C++` code to facilitate passing binary files back and forth.

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 execution time from `fourierTransform.cpp`, measure the runtime of the DFT/IDFT pair as the number of samples, N, increases by a factor of 2 from 2^{10} to 2^{14} (a total of 5 measurement points). Analyze the runtimes and compare them to the big $\mathcal{O}$ computational time complexity of the DFT/IDFT algorithms.

  Then, recompile the code with the `-O1` compiler flag (e.g., `g++ -O1 ...`), which enables basic code optimizations. Measure and observe the updated runtimes for the same sample sizes (2^{10} to 2^{14}) and comment on the findings from this exercise in your report.

- After completing the in-lab requirements for `filterDesign.cpp`, write a function to multiply **two** tones. Assume the frequencies of the tones are random values between 25 and 75 Hz, and their amplitudes are random values between 3 and 6, varying with each run. To validate the implementation of your low-pass filter coefficients (impulse response) and single-pass convolution for digital filtering, filter the signal obtained from the multiplication of the two tones, using a cutoff frequency equal to the higher of the two tone frequencies.

  Use `gnuplot` to visualize both the time-domain signals and the magnitude spectrum in the frequency domain. Note that you will need to update the format of the `example.dat` file (to exclude the third tone) and adjust the x and y axis ranges in `example.gnuplot` as needed.

- In `blockProcessing.cpp`, you must implement support for processing an audio file in blocks rather than filtering all the data in a single pass. Conceptually, this task is similar to one of the take-home exercises from the previous lab.

  In addition to audio filtering, perform a unit test to ensure the correctness of state saving in your block convolution function. You can use the unit test function from `fourierTransform.cpp` as a template, along with supporting methods such as `generateRandomSamples`, `areVectorsClose`, and `maxDifference`. In addition to specifying the maximum value for random numbers and the precision of real numbers, your unit test should accept three integer arguments: the length of the random x vector, the length of the kernel h, and the number of blocks.

  The unit test should generate random vectors for x and h, perform block convolution with state saving, and then compare the result against the output when performing block convolution with the number of blocks doubled or halved.

The report must include three sections, one for each of the above tasks. Each section should be concise, with a maximum length of one page for the entire report unless there are extraordinary points to elaborate.

The source code must reside in the `src` sub-folder of your group's GitHub repository, and your report (in `.pdf`, `.txt`, or `.md` format) must be placed in the `doc` sub-folder.

Your submission, which should be pushed to the main branch of your group's GitHub repository for this lab, is due 16 hours before the start of your next lab session. Late submissions will incur penalties.

This lab contributes 6% to your total grade.

Appendix - Using gnuplot with `filterDesign`

The `plotaddedSinesSpectrum` function from `filterDesign.cpp` logs data in a tabular format to the file `example.dat`, which is used by `gnuplot` for visualization. Each column in the file corresponds to specific aspects of the data. Below is a mapping of the data columns to the three sections in the `example.gnuplot` script.

Section	**X-Axis**	**Y-Axis Columns**	**Purpose**
Sines	`index`	`sine(0)`, `sine(1)`, `sine(2)`	Visualizes individual sine wave components in the time domain.
Filter In/Out	`index`	`data in`, `data out`	Compares the raw input signal and the filtered output signal in the time domain.
Spectrum	`index`	`spectrum in`, `spectrum out`	Compares the frequency-domain representation of the input and filtered output signals.

Table 2: Mapping of data columns from `example.dat` to sections in `example.gnuplot`.

Data Column Details

The data file is structured as follows:

- **Column 1: `index`** — Sample indices for plotting.
- **Columns 2-4: `sine(0)`, `sine(1)`, `sine(2)`** — Individual sine wave components from the vector `sv` in `plotaddedSinesSpectrum`.
- **Columns 5-6: `data in`, `data out`** — The input signal (`x`) and the filtered output signal (`y`).
- **Columns 7-8: `spectrum in`, `spectrum out`** — Magnitudes of the input (`Xf`) and output (`Yf`) signals in the frequency domain, normalized by the data size.

Adjusting the Number of Samples for Plotting

By default, the `example.gnuplot` script plots all 512 samples from the logged file. To plot a subset of samples:

- Open the gnuplot script and locate the `xrange` command for the Sines or Filter In/Out section. For example:

  ```
  set xrange [0:511]
  ```

- Adjust the range to include only the desired samples. For instance, to plot samples from index 50 to 190:

  ```
  set xrange [50:190]
  ```

- Save the modified script and re-run gnuplot to generate the updated plot.

Adjusting the `xrange` enables flexibility for analyzing specific portions of the logged data, which is particularly useful when debugging or focusing on subsets of samples.