University of Pittsburgh Department of Electrical and Computer Engineering
ECE 0301: ECE Problem Solving With C++

ECE 0301: Programming Problem 8

DUE SUNDAY OCTOBER 27, 11:59 PM
LATE SUBMISSION WITH 30% PENALTY BY TUESDAY 11:59 PM FOLLOWING THE DUE DATE

In this problem you will develop a Signal class with overloaded arithmetic and comparison operators. Follow instructions below to implement the class, write your own tests for the class methods, and submit your code via Gradescope to check your solution against the autograder tests.

Background: A time-domain signal can be expressed as a function $f(t)$, where t is time, and $f(t)$ is the value of the signal at that point in time. For the purpose of this assignment, we will assume that the time data is ordered and linearly spaced over a time interval $t_0 \leq t \leq t_{N-1}$, so that

$$t_n = t_{n-1} + \Delta t = t_0 + n\Delta t$$

where Δt is the time between samples, or the *time step*. The reciprocal of Δt is the *sampling rate* or *sampling frequency*:

$$f_s = \frac{1}{\Delta t}$$

and we can express the sample times as

$$t_n = t_0 + \frac{n}{f_s}.$$

In this assignment, you will develop a class for a time-domain signal. The class will initialize a signal with all zeros given a specified sampling frequency and number of samples, and there will be methods available to modify the signal values to generate sinusoids.

For simplicity we will assume that all signals start at time 0, and that the sampling frequency and number of samples for a given Signal instance cannot be changed after instantiation. 1. Follow the steps below to declare methods in Signal.hpp and implement the methods in Signal.cpp. It is **strongly recommended** that you read through all of the instructions and add "placeholder" definitions for all methods to get your project to compile, so that you can debug your code in small steps rather than trying to implement everything before testing. Add tests for each method in test_signal.cpp.

1. Download the starter code from Canvas and open the folder in VSCode with File > Open Folder. Run CMake: Configure. Note that the project will not compile until you add the methods that are called by the provided test cases.
2. Open the files Signal.hpp, Signal.cpp, and test_signal.cpp.
    a. Note the public static member variable **MAX_SAMPLES** defined for the Signal class. You will use **statically allocated arrays** for all arrays in this assignment (do not use pointers, do not use dynamic memory management).
    b. Note the provided private struct **Point**, with fields **time** and **value**.
    c. Note the following **private** member variables declared in Signal.hpp. Do not change these, as they are used by the provided class methods:
        i. An integer variable to store the number of samples.
        ii. An integer variable to store the sampling frequency.
        iii. An array of Point structs.
    d. Finally, note the provided **Signal** class methods **zeros, constant, export_signal,** and **export_csv**, these methods will be explained later in the instructions.
3. Declare and implement the following public constructors and accessor methods:
    a. A **default constructor** that sets the number of samples to 101, and the sampling frequency to 100 Hz.
        i. Use the provided public member function **zeros**. This method fills the array member with Point structs with values of zero at uniformly spaced times according to the sampling frequency and number of samples.
    b. **get_samples:** A const accessor method that returns the value of the member variable for the number of samples.
    c. **get_sampling_freq**: A const accessor method that returns the value of the member variable for the sampling frequency.
    d. A constructor that takes as arguments the number of samples and sampling frequency, and initializes the values of the member variables accordingly.
        i. If the provided number of samples is negative or greater than the value MAX_SAMPLES, or the frequency argument is zero or negative, the constructor should throw an exception of type **std::invalid_argument**. Remember to include the <stdexcept> library.
4. One of the provided test cases uses the provided **export_signal** method to test that your constructors function correctly. The export_signal method takes as arguments two arrays of double values. It exports the values from the member array of Point structs by copying the time values to the first array argument, and copying the signal to the second array argument. Make sure that you understand this test case so that you can use export_signal in your own tests for the remaining functions.
5. If at any time you wish to visualize your signal, you can use the provided **export_csv** function, which takes an output file name as an argument (for example, “output”, not including the file extension).

   a. The signal times and values will be exported to a csv file with the specified file name, from which the data can be plotted in Excel, or in Matlab using the provided script **read_signal_plot.m**. The commented test case at the beginning of test_signal.cpp shows example syntax for exporting the signal to a file.
   b. The file will be saved in the build/ folder unless you use a launch.json file to change the current working directory.
6. Declare and implement a public member function **sinusoid** that fills the member array with samples of a sinusoidal signal, according to the following equation:

$$s(t) = A\cos(2\pi f_0 t + \phi).$$

   a. This member function should accept three parameters: a double that specifies the amplitude *A*, an int that specifies the sinusoid frequency f_0 (in Hertz), and a double that specifies the phase ϕ (in radians), respectively.
   b. The function should use the existing time values for the Signal instance, and re-calculate the signal value at each time point according to the above equation.
   c. The function should return nothing.
   d. If either the provided amplitude or frequency argument is negative, the function should throw a **std::invalid_argument** exception.
   e. Remember to include the **<cmath>** library to use the cosine function and M_PI.
7. Declare and implement the following overloaded operators as **public member functions**. Note that while addition/subtraction are often implemented as non-members, you should implement them as member functions in this assignment to allow access to private members of the class.
   a. **Equal to (==):** Should take in a Signal argument by **const reference**, and return true if the argument Signal and the current Signal instance have equal sampling frequencies, numbers of samples, and signal values; return false otherwise.
      i. Because the Point struct does not have an equality operator defined, you should compare the individual double fields **time** and **value**.
      ii. Hint: The provided function **constant** fills a signal with constant values equal to the provided argument. This may be a useful function to use for initial testing rather than trying to debug copying sinusoidal signals right away.
   b. **Not equal to (!=):** Opposite functionality as "equal to"
   c. **Addition (+):** Add two signals, meaning add the signal values at each time point (element-wise addition). Implemented as a **member function**, this should take in a Signal argument by **const reference**, and return a Signal object (by value).
      i. You should assume for this assignment that the sampling **frequency** must be equal in order to add two signals. Therefore this function should compare the sampling frequency of the two signals to be added, and if

the sampling frequencies are not equal, throw an exception of type **std::logic_error**.

ii. Assuming the sampling frequencies are equal, the function should only **add** values up to the length of the signal with fewer samples, and then **copy** remaining values from the signal with more samples, to avoid access to uninitialized values in the shorter signal.

Files to submit:

Submit the following files for this assignment:

1. Signal.hpp
2. Signal.cpp
3. test_signal.cpp

You again have the option to generate a zip file for submission with the following steps:

1. Open your project folder in VSCode
2. Set the build target to “submission”: Open the command palette, search for and run CMake: Set Build Target, and select “submission”
3. Run CMake: Build
4. A zip file should be created in your build/ folder. Upload this file to Gradescope
5. Ensure that you set the built target back to your executable if you continue testing
6. If you make changes to any files, you will need to build the submission target again to generate a new submission zip.