COE3DY4 Lab #4
Code Optimization

Objective

The main objective of this lab is to learn about code optimization, as well as how to benchmark and unit test refactored code.

Preparation

- Revise the material from labs 1, 2 and 3

Algorithm Design vs Implementation

Big $\mathcal{O}$ notation analyzes algorithms by characterizing their growth rate in terms of time or space complexity. While it is an important mathematical tool, it does not account for practical factors that cause runtime variations. For instance, two implementations of the same algorithm with identical Big $\mathcal{O}$ complexity can perform differently on the same hardware. Understanding the subtle but critical distinctions between the theoretical design of an algorithm and the practical details of how it is implemented in code is a key factor in achieving performance improvements.

The goal of this lab is to instill a mindset that code implementation can be iteratively improved through a systematic process of measurement, refactoring, and testing, as shown in Figure 1.

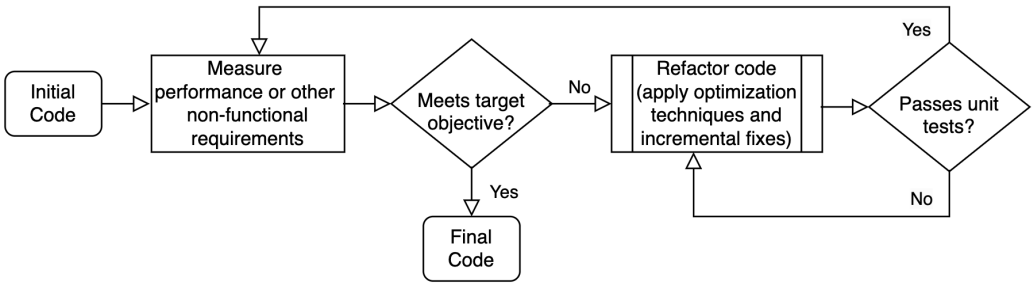

Figure 1: Iterative code optimization

In the context of this lab, refactoring involves applying intuitive optimization techniques based on Bentley's rules[1]. While code optimization techniques have evolved and now encompass much more than can be discussed in a course, the objective is to raise awareness of their importance and effectiveness, especially when applied to basic Digital Signal Processing (DSP) primitives.

[1] Jon Bentley, *Writing Efficient Programs*, Prentice Hall, 1982

Discrete Fourier Transform

Consider the Discrete Fourier Transform (DFT) code that you used in the previous labs. The pseudocode is provided in Algorithm 1:

Algorithm 1 Basic Discrete Fourier Transform (DFT) algorithm

Input: Time domain samples $x(k)$ with $k = 0 \dots N-1$
Output: Discrete frequency bins $X(m)$ with $m = 0 \dots N-1$

```
Initialize all elements of X(m) to zero
for m ∈ [0, N − 1] do
    for k ∈ [0, N − 1] do
        X(m) ⇐ X(m) + x(k) × e^{−2πj km/N}
    end for
end for
```

In the reference code provided, you have two functions: the DFT function from the previous lab and another that has been slightly modified. These are intended to help you understand the workflow for refactoring code to improve performance and conducting unit tests to confirm that the optimizations preserve correctness. It is worth mentioning that in the modified code the structure of the two `for` loops is identical to the code from the previous lab. However, as a first step toward improving the code efficiency (with negligible benefits, if any, when considered in isolation from other optimizations), instead of using the `clear` and `resize` methods, you can rely on the fact that the size of the data segments on which the DFT is applied does not change. Thus, space can be allocated in the function that calls the DFT, and, within the DFT function, the size can remain unchanged. To clear the content of the frequency bins, the `fill` method from `std::vector` can be used instead. Whenever the size of the data blocks is known at compile time, space can be allocated only once, and using `fill` instead of `clear` and `resize` is the recommended approach.

As part of your in-lab experiments, you will work on improving the performance of the DFT, verifying the correctness of the refactored code, and measuring its performance improvements using the framework provided with the reference code. Refer to the Appendix for details on managing the compilation process with `cmake` and guidance on using `Google Test`[2] and `Google Benchmark`[3] to streamline test development and measure the performance gains from code optimizations.

Code Motion

Code motion moves redundant calculations outside of loops by identifying operations that produce the same result across iterations and restructuring the code to avoid their repeated computation, often by exploiting algebraic identities to simplify or precompute values.

The exponent of the complex exponentials, $-2\pi j\frac{km}{N}$, contains the angular frequency step $-2\pi j\frac{1}{N}$, which can be precomputed. An intermediate exponent, $-2\pi j\frac{m}{N}$, can be computed once per outer loop iteration by scaling the angular frequency step with the frequency bin index m. Within the inner loop, the full twiddle exponent, $-2\pi j\frac{km}{N}$, is obtained by multiplying the intermediate exponent by the sample index k. While most runtime is spent multiplying the complex exponential $e^{-2\pi j\frac{km}{N}}$ with sample $x(k)$ and accumulating to $X(m)$, avoiding redundant calculations of the exponent in the innermost loop from Algorithm 2 may yield modest performance improvements.

[2] https://github.com/google/googletest
[3] https://github.com/google/benchmark

Algorithm 2 DFT computation with code motion

Input: Time domain samples $x(k)$ with $k = 0 \dots N-1$
Output: Discrete frequency bins $X(m)$ with $m = 0 \dots N-1$

```
Initialize all elements of X(m) to zero
base_exponent ⇐ −2πj 1/N                          ▷ Angular frequency step for N-point DFT
for m ∈ [0, N − 1] do
    intermediate_exponent ⇐ base_exponent × m      ▷ intermediate_exponent ⇐ −2πj m/N
    for k ∈ [0, N − 1] do
        twiddle_exponent ⇐ intermediate_exponent × k      ▷ twiddle_exponent ⇐ −2πj km/N
        X(m) ⇐ X(m) + x(k) × e^twiddle_exponent
    end for
end for
```

While eliminating redundant calculations produces code theoretically equivalent to the original, floating-point arithmetic can introduce rounding errors, especially with single-precision floats. Using double precision mitigates this risk and ensures the optimization maintains numerical accuracy.

Precompute the N Twiddle Factors

Compilers often perform code motion during the compilation process when optimizations are enabled (e.g., `-O3`), which may explain why simple transformations, like the one above, have limited impact. However, combining code motion with problem-specific algebraic identities can achieve further optimization through both code motion and precomputation.

We will use the following notation for the twiddle factors $W_N^k = e^{-2\pi j \frac{k}{N}}$. Accounting for the periodicity property of the twiddle factors, $W_N^M = W_N^{M \bmod N}$ (simple math shows $W_N^N = 1$ and $W_N^{2N} = W_N^N \cdot W_N^N$), we can precompute only N distinct twiddle factors W_N^k for $k = 0 \dots N-1$. When calculating $X(m)$ in the innermost loop, W_N^{km} can be replaced with the precomputed $W_N^{(km) \bmod N}$.

Algorithm 3 DFT computation with a precomputed one-dimensional array with twiddle factors

Input: Time domain samples $x(k)$ with $k = 0 \dots N-1$
Output: Discrete frequency bins $X(m)$ with $m = 0 \dots N-1$

```
Precompute twiddle factors Twiddle1D(k)
for k ∈ [0, N − 1] do
    Twiddle1D(k) ⇐ e^(−2πj k/N)
end for
Initialize all elements of X(m) to zero
for m ∈ [0, N − 1] do
    for k ∈ [0, N − 1] do
        X(m) ⇐ X(m) + x(k) × Twiddle1D((k × m) mod N)
    end for
end for
```

Algorithm 3 can be further optimized by precomputing the vector of twiddle factors *Twiddle1D* and passing it as input. However, the index $(k \times m) \bmod N$ must still be computed in the innermost loop to look up the twiddle factor. This multiplication and division can be avoided by precomputing a two-dimensional twiddle factor vector *Twiddle2D* (the **DFT matrix**), which can be reused for all Fourier transforms of size N.

Precompute the DFT Matrix

The **DFT matrix**, *Twiddle2D*, is defined as a two-dimensional matrix of size $N \times N$:

$$Twiddle2D = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N^1 & W_N^2 & \cdots & W_N^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & \cdots & W_N^{(N-1)(N-1)} \end{bmatrix},$$

Stated differently, $Twiddle2D[m,k] = W_N^{km} = e^{-2\pi j \frac{km}{N}}$, where m is the index for frequency bins and k is the index for time domain samples. The DFT can be computed as a matrix-vector multiplication, as shown below. This approach allows for a clear separation of the DFT calculation and the precomputation of *Twiddle2D*, which can be reused for multiple DFTs of the same size N.

While precomputation is an effective technique to avoid recalculations, it is equally important to consider how data is stored and moved between memory and the central processing unit (CPU), also known as the microprocessor.

In Algorithm 4, the outer loop traverses the frequency bins, and each inner loop iteration accesses a full row of the precomputed DFT matrix *Twiddle2D*. Since the matrix is stored in row-major format, these elements are contiguous in memory, providing good spatial *locality*. When the CPU fetches one element, subsequent elements are already loaded into the *cache*, a small, fast memory near the CPU that stores recently accessed data to reduce access times. This data access pattern minimizes cache misses and improves performance.

Algorithm 4 DFT with outer loop over m (frequency bins)

Input: Time domain samples $x(k)$ with $k = 0 \ldots N-1$
Input: Precomputed DFT matrix *Twiddle2D* of size $N \times N$
Output: Discrete frequency bins $X(m)$ with $m = 0 \ldots N-1$

```
Initialize all elements of X(m) to zero
for m ∈ [0, N − 1] do
    for k ∈ [0, N − 1] do
        X(m) ⇐ X(m) + x(k) × Twiddle2D[m, k]
    end for
end for
```

Although Algorithm 4 and Algorithm 5 have the same complexity from a Big $\mathcal{O}$ perspective, their data access patterns differ. In Algorithm 5, the innermost loop iterates over frequency bins m, and for each time sample k, the algorithm accesses different rows of *Twiddle2D*. This results in frequent access to non-contiguous memory locations with a large stride, meaning that consecutive memory accesses are far apart. Since the data for each iteration is not stored close together, the CPU often encounters *cache misses*. If the cache becomes full, cache evictions occur, where older data is replaced. Repeated evictions and reloads can lead to cache thrashing, where more time is spent on managing the cache than executing instructions.

Algorithm performance is also influenced by the number of memory writes. In Algorithm 5, all N frequency bins $X(m)$ are updated in each iteration of the innermost loop, leading to more frequent writes to memory. In contrast, Algorithm 4 updates only one frequency bin per iteration. This bin can be kept in a CPU register and written to memory only once at the end of the innermost loop, minimizing the overhead associated with memory writes.

Algorithm 5 DFT with outer loop over k (time samples) - **loop reordering**

Input: Time domain samples $x(k)$ with $k = 0 \dots N-1$
Input: Precomputed DFT matrix *Twiddle2D* of size $N \times N$
Output: Discrete frequency bins $X(m)$ with $m = 0 \dots N-1$

Initialize all elements of $X(m)$ to zero
for $k \in [0, N-1]$ **do**
  for $m \in [0, N-1]$ **do**
    $X(m) \Leftarrow X(m) + x(k) \times \mathit{Twiddle2D}[m, k]$
  end for
end for

DFT with Loop Unrolling

Loop unrolling is an optimization technique where multiple iterations of a loop are combined into a single iteration by replicating the loop body. This reduces the number of loop control operations, such as increments and comparisons, and allows the compiler to optimize performance.

Algorithm 6 DFT with outer loop over m and loop unrolling (unrolling factor of 4)

Input: Time domain samples $x(k)$ with $k = 0 \dots N-1$
Input: Precomputed DFT matrix *Twiddle2D* of size $N \times N$
Output: Discrete frequency bins $X(m)$ with $m = 0 \dots N-1$

Initialize all elements of $X(m)$ to zero
$\mathit{unrolled\_limit} \Leftarrow N - (N \mod 4)$
for $m \in [0, N-1]$ **do**
  for $k \in [0, \mathit{unrolled\_limit} - 1]$ `with step size 4` **do**
    $X(m) \Leftarrow X(m) + x(k) \times \mathit{Twiddle2D}[m, k]$
      $+x(k+1) \times \mathit{Twiddle2D}[m, k+1]$
      $+x(k+2) \times \mathit{Twiddle2D}[m, k+2]$
      $+x(k+3) \times \mathit{Twiddle2D}[m, k+3]$
  end for
  for $k \in [\mathit{unrolled\_limit}, N-1]$ **do**
    $X(m) \Leftarrow X(m) + x(k) \times \mathit{Twiddle2D}[m, k]$
  end for
end for

In Algorithm 6, loop unrolling is applied by processing four elements of $x(k)$ and *Twiddle2D* in each iteration. This reduces the total number of loop iterations and, consequently, the number of branch instructions controlling loop execution. CPUs use branch prediction to keep their instruction pipelines full, as waiting for the outcome of a branch condition can cause the pipeline to stall. To avoid such stalls, the CPU predicts the outcome of branches and continues fetching and executing instructions speculatively. However, if the prediction is incorrect, a *branch misprediction* occurs, forcing the CPU to discard partially executed instructions and restart the pipeline, wasting cycles and degrading performance.

Additionally, unrolling increases opportunities for instruction-level parallelism, where independent operations (e.g., multiple multiplications and additions) can be executed simultaneously without waiting on each other. Modern CPUs can further optimize this through vectorization, allowing a single instruction to perform operations on multiple data points at once.

Convolution

Convolution, as demonstrated in previous labs, is a fundamental DSP primitive. While block convolution is commonly used in practice for software-defined radio (SDR) projects, this lab focuses on learning code optimization through single-pass convolution as a case study.

Single-Pass Convolution and Inner Loop Traversal Order (Signal vs Kernel)

There are two ways to organize the nested loops in a convolution algorithm: traversal over the **signal** (input data x) in the inner loop or traversal over the **kernel** (impulse response h) in the inner loop.

Algorithm 7 Single-pass convolution with inner loop over **signal** x

Input: Signal samples $x(k)$ with $k = 0 \dots N-1$ ▷ N is the size of the input data vector
Input: Kernel weights $h(k)$ with $k = 0 \dots M-1$ ▷ M is the size of the impulse response
Output: Output samples $y(n)$ with $n = 0 \dots N+M-2$

```
Initialize all elements of y(n) to zero
for n ∈ [0, N + M − 2] do
    for k ∈ [0, N − 1] do
        if (n − k) ≥ 0 and (n − k) < M then
            y(n) ⇐ y(n) + x(k) × h(n − k)
        end if
    end for
end for
```

In Algorithm 7, the inner loop iterates over the signal x, while the kernel h is accessed conditionally. Although at most M (kernel size) partial products are computed for each output sample $y(n)$, the conditional check is executed N times (signal size), which is usually much larger than M. This makes the algorithm inefficient and impractical.

By exploiting the property that $x * h = h * x$, the convolution can be rewritten as follows:

Algorithm 8 Single-pass convolution with inner loop over **kernel** h

Input: Signal samples $x(k)$ with $k = 0 \dots N-1$
Input: Kernel weights $h(k)$ with $k = 0 \dots M-1$
Output: Output samples $y(n)$ with $n = 0 \dots N+M-2$

```
Initialize all elements of y(n) to zero
for n ∈ [0, N + M − 2] do
    for k ∈ [0, M − 1] do
        if (n − k) ≥ 0 and (n − k) < N then
            y(n) ⇐ y(n) + h(k) × x(n − k)
        end if
    end for
end for
```

Algorithm 8 iterates over the kernel h in the inner loop, reducing conditional checks by a factor of $\frac{N}{M}$, making it our reference algorithm. As with the DFT optimizations, both implementations are provided. In your in-lab experiments, you will refactor convolution, unit test the modified code, and measure performance using the provided framework.

Single-Pass Convolution with Precomputed Range

To reduce overhead from conditionals, we can precompute the valid range of indices for the inner loop to avoid checks during each iteration of the innermost loop.

Algorithm 9 Single-pass convolution with precomputed valid range for the inner loop

Input: Signal samples $x(k)$ with $k = 0 \dots N-1$
Input: Kernel weights $h(k)$ with $k = 0 \dots M-1$
Output: Output samples $y(n)$ with $n = 0 \dots N+M-2$

```
Initialize all elements of y(n) to zero
for n ∈ [0, N + M − 2] do
    k_min ⇐ max(0, n − N + 1)          ▷ Compute the minimum valid index for k
    k_max ⇐ min(M, n + 1)              ▷ Compute the maximum valid index for k
    for k ∈ [k_min, k_max − 1] do
        y(n) ⇐ y(n) + h(k) × x(n − k)
    end for
end for
```

In Algorithm 9, the valid range of indices $k_{\min}$ and $k_{\max}$ is precomputed in the outer loop before the inner loop starts. By performing these boundary checks once per output sample $y(n)$, branch instructions in the inner loop are eliminated, reducing overhead from repeated conditionals.

Single-Pass Convolution with Precomputed Range and Unrolling

As with the DFT, loop unrolling can further optimize performance for convolution.

Algorithm 10 Single-pass convolution with precomputed range and unrolling (factor of 4)

Input: Signal samples $x(k)$ with $k = 0 \dots N-1$
Input: Kernel weights $h(k)$ with $k = 0 \dots M-1$
Output: Output samples $y(n)$ with $n = 0 \dots N+M-2$

```
Initialize all elements of y(n) to zero
for n ∈ [0, N + M − 2] do
    k_min ⇐ max(0, n − N + 1)
    k_max ⇐ min(M, n + 1)
    unrolled_limit ⇐ k_max − ((k_max − k_min) mod 4)   ▷ Adjust limit for unrolling (factor of 4)
    for k ∈ [k_min, unrolled_limit − 1] with step size 4 do
        y(n) ⇐ y(n) + h(k) × x(n − k) + h(k + 1) × x(n − (k + 1))
                    + h(k + 2) × x(n − (k + 2)) + h(k + 3) × x(n − (k + 3))
    end for
    for k ∈ [unrolled_limit, k_max − 1] do
        y(n) ⇐ y(n) + h(k) × x(n − k)
    end for
end for
```

Algorithm 10 improves performance by precomputing the valid range of indices in the outer loop and unrolling the inner loop body by a factor of 4. Unrolling accesses consecutive elements of $x(n-k)$ and $h(k)$, leveraging the spatial locality inherent to convolution. The reduced control flow allows independent arithmetic operations, such as multiplying and adding $x(n-k)$ and $h(k)$, to execute in parallel across multiple execution units, increasing instruction throughput.

In-lab experiments

As part of the in-lab work, perform the following tasks:

- Two variants of the reference DFT code from Algorithm 1 (`DFT_reference` and `DFT_init_bins`) are implemented as standalone functions in `fourier.cpp` in the `src` sub-folder, with prototypes declared in `fourier.h` in the `include` sub-folder. Unit tests for `DFT_init_bins`, comparing it to `DFT_reference`, are in `dft_test.cpp` in the `test` sub-folder. Benchmarking code for both functions is in `dft_bench.cpp` in the `bench` sub-folder. As a first step, review the workflow for implementing, testing, and benchmarking, as summarized in the Appendix.
- Based on the provided pseudocode, you are required to implement each DFT algorithm as a standalone function, along with its unit tests and benchmarks, following the approach used for `DFT_reference` and `DFT_init_bins`. These include Algorithm 2 (DFT with code motion), Algorithm 3 (precomputation of a one-dimensional twiddle factor vector), Algorithm 4 and Algorithm 5 (DFT matrix with frequency bins traversed in the outer and inner loops), and Algorithm 6 (DFT matrix with loop unrolling). The `generate_DFT_matrix` function is provided, so your focus will be on implementing these algorithms.
- For convolution, two variants are provided: Algorithm 7 (iterate over signal x) and Algorithm 8 (iterate over kernel h). These implementations (`convolveFIR_inefficient` and `convolveFIR_reference`) are in `filter.cpp` (in `src`), with function prototypes in `filter.h` (in `include`). Unit tests are in `conv_test.cpp` (in `test`), and benchmarks in `conv_bench.cpp` (in `bench`). As part of the in-lab work, you will implement Algorithm 9 (precompute valid range) and Algorithm 10 (range precomputation with unrolling), with unit tests and benchmarks, following the same approach as for the provided functions.
- Implement three new algorithms for single-pass convolution. The first applies loop reordering to Algorithm 8 by iterating over kernel h in the outer loop and output y in the inner loop. Next, optimize this loop-reordered implementation by precomputing valid index ranges to eliminate branches in the innermost loop (as in Algorithm 9). Finally, apply loop unrolling to this loop-reordered implementation (as in Algorithm 10). Implement, test, and benchmark these algorithms to assess performance improvements.

Take-home exercises

In addition to **completing** *and* **submitting** your in-lab experiments, to further improve your understanding code optimization, you should perform the following:

- The file `fmSupportLib.py` in the `model` sub-folder contains a matrix-based function for power spectral density (PSD) computation. A unit test verifies that it matches the `Python estimatePSD` implementation from the previous lab. Confirm that `estimatePSD` and `matrixPSD` give equivalent results.
- Refactor the `matrixPSD` code to `C++` and create two new files: `psd_test.cpp` and `psd_bench.cpp` to unit test and benchmark it against your `estimatePSD` from the previous lab. When benchmarking PSD, iterate over both DFT size and block size (for the samples), following the same principles as the in-lab convolution benchmarks, which varied signal and kernel sizes.
- Explore at least two optimizations learned in this lab to your `matrixPSD`. Unit test each optimized version and benchmark them using the principles covered in this lab.

The report must have one section for the last in-lab (loop reordering for convolution) and one for the last take-home exercise (`matrixPSD` benchmarking). 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 (lab cross-exam). Late submissions will incur penalties.

This lab contributes 7% to your total grade.

Appendix

This lab is organized into various sub-folders, as will be the case for the project, each serving a specific purpose. Below is an overview of these sub-folders and the types of files stored in them.

Project Directory Structure

- `bench`: This sub-folder contains benchmarking code used to measure the performance of various DSP primitives. Due to the way the `CMakeLists.txt` file discovers which files should be included in the benchmark build, all filenames in this sub-folder must have a `_bench` suffix.
- `build`: This sub-folder holds build artifacts. Change to this sub-folder and run the following pair of commands: `cmake .. && make`. The first command (`cmake ..`) will autogenerate the `Makefile` based on the content of the `CMakeLists.txt` file in the root folder. The second command (`make`) will generate three executables: `project`, `bench_primitives`, and `test_primitives`. Note that the first compile may take some time. Running the `project` executable built from the reference code will produce the same output as the `experiment` code provided in the previous lab. Details on how to use `bench_primitives` and `test_primitives` (the focus of this lab) will be elaborated later in this Appendix.
- `data`: This sub-folder stores data files used by the project, including example scripts (e.g., for `gnuplot`). This is where `.raw` files with RF samples should be stored, along with all files (log data, images, etc.) generated when running your `project`.
- `doc`: This sub-folder stores documentation files, such as project specifications and reports.
- `include`: This sub-folder contains header files that define function prototypes and declarations for use throughout the project. Any new header files should be added to this sub-folder.
- `model`: This sub-folder contains `Python` scripts. These are the same files from the previous lab, with the addition of `matrixPSD` in `fmSupportLib.py`.
- `src`: This sub-folder contains the main `C++` source files for the project. It includes the same files released for the previous lab, with a few new functions added to `fourier.cpp` and `filter.cpp` to help you get started with benchmarking and unit testing. The file `experiment.cpp` has been renamed to `project.cpp`. Note that the `Makefile` has been removed from this sub-folder, as it is now autogenerated in the `build` sub-folder.
- `test`: This sub-folder contains unit testing code. Following the same principles as the benchmarking files, all test filenames must have a `_test` suffix and be placed in this directory.

Managing the Build using `cmake`

The `CMakeLists.txt` file in the root directory defines the build system. It is automatically parsed when running `cmake ..` from the `build` sub-folder. The `cmake ..` command is only required when adding new files to the `src`, `test`, or `bench` directories. For changes within existing files, recompilation with only `make` is sufficient.

The C++ standard is set to C++17, with default flags `-Wall` for warnings and `-O3` for optimization. The `DOUBLE` macro redefines the real type as `double` instead of `float`, as in previous labs.

Dependencies such as `googletest` and `googlebenchmark` are fetched using `FetchContent`. The `include` sub-folder is added to the include paths. Files are filtered by naming conventions to exclude them from specific targets. By default, three executables are generated in `Release` mode. Additional documentation is provided for `Debug` mode or generating individual executables.

The autogenerated `Makefile` builds executables (run `make` in the `build` sub-folder). The `project` executable excludes test and benchmark files. The `test_primitives` executable excludes benchmark files and `project.cpp`, where the `main` function for the `project` resides. It links with `gtest_main`, and Google Test is configured to discover and run tests. Similarly, the `bench_primitives` executable excludes test files and `project.cpp`, linking with the `benchmark` library.

Unit Testing using the `Google Test` Framework

The `DFT_Fixture` class in `dft_test.cpp` (located in the `test` sub-folder) manages the setup for DFT-related tests. It defines constants like `N` (input size) and member variables such as `x`, `Xf_reference`, and `Xf_test`, which store the input signal and output vectors. The constructor initializes these vectors, and the `SetUp()` method populates `x` with random values generated by the custom `generate_random_values` function (from `iofunc.cpp` in the `src` sub-folder). The reference output is also computed using `DFT_reference` within the `SetUp()`.

The `TEST_F` macro defines test cases that operate on the fixture. Each test computes the output of the algorithm under test (e.g., `DFT_init_bins`) and stores it in `Xf_test`. Macros such as `ASSERT_EQ` and `EXPECT_NEAR` compare reference and test outputs. To temporarily disable a unit test, add the `DISABLED_` prefix to the test name. After recompilation, tests are executed by running the `test_primitives` executable.

Benchmarking using the `Google Benchmark` Framework

The `Bench_DFT_reference` function in `dft_bench.cpp` (in the `bench` sub-folder) illustrates principles that apply to all benchmarks. It sets the problem size using `state.range(0)`, based on registered input ranges. The `state` object handles timing, iterations, and metadata, so users only need to specify input ranges during registration. The function allocates input and output vectors and fills the input with random values via the custom `generate_random_values` function. The benchmark loop, controlled by `for (auto _ : state)`, runs multiple iterations and averages results for reliable performance measurements.

Benchmarks are registered using the `BENCHMARK` macro. The `RangeMultiplier` sets the input size increment (e.g., doubling), while `Range` defines the minimum and maximum sizes. After recompilation, benchmarks are run by executing `bench_primitives`. By default, all registered benchmarks are executed. To run a subset, use the `--benchmark_filter` flag. For example, to benchmark only functions with "DFT" in their name, run: `./bench_primitives --benchmark_filter=DFT.*`