C-Coursework

Student Number: 23762

April 16, 2021

Exercise 1

To solve this problem, we begin from the formula for the discrete Fourier transform:

$$H_n(\omega_n) = \sum_{k=0}^{N-1} h_k(t_k) e^{-\frac{2\pi nk}{N} i}, \tag{1.1}$$

and our function $h(t)$,

$$h(t_k) = e^{i\omega_1 t_k}, \tag{1.2}$$

where $\omega_1 = 1$, $N = 4$, and time period, $T = 2\pi$. We can find an expression for t_k using the fact that the sampling interval is given by $\Delta = \frac{T}{N}$. Combined with the expression $t_k = k\Delta$, (1.2) becomes,

$$h(t_k) = e^{\frac{k\pi}{2} i}, \tag{1.3}$$

which can now be combined with (1.1) to calculate values H_0, H_1, H_2, and H_3. As we are calculating the discrete Fourier transform for a relatively small value of N, we can expand the equation out as follows,

$$\begin{aligned} H_n(\omega_n) &= \sum_{k=0}^{N-1} e^{\frac{k\pi}{2} i} e^{-\frac{2\pi nk}{N} i} \\ &= e^0 + e^{\frac{\pi}{2} i} e^{-\frac{\pi n}{2} i} + e^{\pi i} e^{-\pi ni} + e^{\frac{3\pi}{2} i} e^{-\frac{3\pi n}{2} i}, \end{aligned}$$

applying Euler's formula enables us to simplify the expression further,

$$H_n(\omega_n) = 1 + e^{-\frac{\pi n}{2} i} - e^{-\pi ni} - e^{-\frac{3\pi n}{2} i}. \tag{1.4}$$

We are now ready to simply slot in the values for $n = 0, 1, 2, 3$ to find our answers.

$$
\begin{aligned}
H_0 &= 1 + e^0 - e^0 - e^0 = 0 \\
H_1 &= 1 - 1 + 1 - 1 = 0 \\
H_2 &= 1 - 1 - 1 + 1 = 0 \\
H_3 &= 1 + e^{-\frac{3\pi}{2}i} - e^{-\frac{3\pi}{i}} - e^{-\frac{3\pi}{2}i} \\
&= 1 + 1 + 1 + 1 = 4
\end{aligned}
$$

Exercise 2

The general structure of my approach to question 3 is that processes which may need to be completed more than once are turned into their own functions to prevent repetition, for example writing to a file or producing a range of evenly spaced values, and each part of question 3 will be it's own function. This will increase the length of the code, however it will allow for a more direct connection to between the task to complete in each part and the solution. These functions are then called in the *main* function with minimal additional code. To avoid excessive detail about the inner working of functions mentioned here, the code itself will be well commented and variable names will be chosen to be descriptive of what is stored in them and improve readability.

Part A

For part A, a method of representing complex numbers is required to describe the outputs of the functions h_1 & h_2:

$$h_1(t_k) = e^{it} + e^{5it}$$
$$h_2(t_k) = e^{\frac{(t-\pi)^2}{2}}$$

This will be done using by defining a custom struct $Complex$, which handles complex functionality with two doubles representing the real and imaginary components of the complex number. This can be then combined with a function for h_1 which receives a double representing the time, t. Then, Euler's formula is applied to calculate the value of $h_1(t)$. Unlike h_1, h_2 does not have an imaginary part. As such, the value of the real component will be calculated using the pow function from the "$math.h$" header file and the imaginary component is set as 0.0.

Part B

Two functions, $linspaceD$ & $linspaceComplex$ will be used to produce the samples of h_1 & h_2. Both functions will use dynamic memory management to return an array of values; however $linspaceD$ will produce a specified number of values between a start and end point, whereas $linspaceComplex$ will take a function pointer matching the template of h_1 & h_2, an array of doubles, and the size of the array of doubles to generate an array of complex numbers using the provided function. These samples will then be written to a file using a function named $writeComplex$ which takes in an array of doubles, an array of $Complex$, the size of the arrays, and a filename to write to. This functions will work as follows:

1. Create a $FILE$ to a file using "$fopen$" from $stdio.h$ in write mode. Check that the result is not $NULL$, as this is the default return if $fopen$ fails.
2. Print headers to the file. In this case, all instances of writing to a file involve a time value and a $Complex$ so the headers are "$time$", "$real$", and "$imag$".
3. Use a for loop to print all values in the time and complex arrays to the file using $fprintf$.
4. Close the file using $fclose$.

Parts D & E

The DFT will be implemented such that the later implementation of the $IDFT$ will not need to repeat code. This will be done by taking in an array of $Complex$ samples, the size of the array, an array of indexes to be skipped if there any, the size of the index array, and a boolean value which indicates whether it is the DFT or $IDFT$ being performed. Without excessive detail, the method used to implement DFT is as follows:

1. Allocate $Complex$ array for results equal in size to input samples.
2. Check if list of indexes to skip is provided, or size of that list is zero.
   (a) If list exists: Apply $qsort$ from $stdlib.h$ to ensure it is sorted as $checkIdx$ will utilise a simple binary search which depends on the input being sorted. The work done sorting is likely to be worth the decrease in work done checking indexes with a basic linear search.
   (b) Else: Skip sort, checkIdx handles a NULL input and all indexes will be considered.
3. If the boolean $IDFT$ is true
4. A for loop iterates through the empty results array, initialising the value at $0 + 0i$.
5. A nested for loop iterates through the input samples, checking the index in case it needs to be skipped. If it is not skipped, the contribution to the transformed value is calculated and added to the current value.
6. Outside of the nested loop and within the first, the final value is saved to the results array.

This function will be used to transform h_1 & h_2 to H_1 & H_2. The values of H_1 & H_2 are then printed to the console.

Parts F, G, & H

In this section, the $IDFT$ will need to be applied to H_1 & H_2 from $3.d$ & $3.e$. By taking advantage of how the DFT function will be implemented it is possible to avoid repeating a large chunk of the implementation. As such, the $IDFT$ will be as follows:

1. Apply DFT to input parameters with boolean for $IDFT$ set to true saving the result.
2. Divide all values by the number of elements.

The $IDFT$ will then be applied to H_1, skipping the n = 1 index, and to H_2, skipping the n = 0 index. This transformation will produce two new arrays h_1' & h_2' which will then be printed to two files, "$inv\_1.txt$" & "$inv\_2.txt$.

Part I

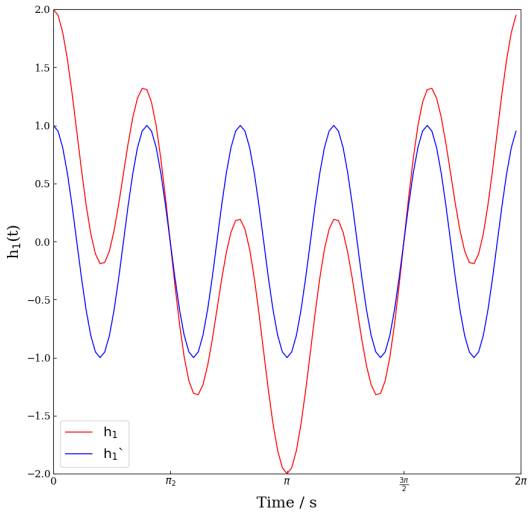
In (i), the data is read from "$h3.txt$" using a while loop and the fscanf function. This data will be stored to a custom struct, named $Measurement$, which will contain a size_t which stores an index position, a double to store a time, and a Complex. This data is then returned. The method to read from the file will be as follows:

1. Open file "$h3.txt$" in read mode using $fopen$. Perform a check in case this fails as it will default to a $NULL$ pointer.
2. Allocate a $Measurement$ array to store 200 values in.
3. A while loop is used to store the values to the results array. The two conditions for the loop will be as follows:
    - The function $fscanf$ from "$stdio.h$" returns the macro EOF when it either reaches the end of the file, or encounters an error. $fscanf$ will read in values from the file pointer provided according to a specified format until the end of the file is reached, EOF is returned, and the while loop ends.
    - The second condition requires that a variable, sz, must be less than the number of elements allocated in the results array. This ensures that even if the end of the file isn't reached, it will not attempt to read any more data than there is memory allocated for.
4. Close the file and return the results.

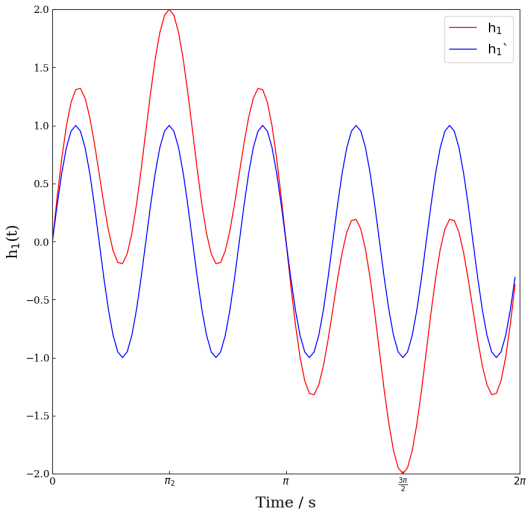
Parts J, K, & L

Then (j), (k), and (l) are very similar to their earlier counterparts. First the DFT is applied to the read in data, producing H3. The main difference is introduced when applying the IDFT, as it is only applied considering the four indexes with the largest complex amplitudes. In order to find these four largest terms, the function "$qsort$" from "$stdlib.h$" is used. To make this function compatible with the custom Complex and Measurement structs to functions, "$compareComplex$", "$compareMeasurement$", and "$compareSize\_t$" will be used. The data will be sorted from smallest complex magnitude to largest, then all indexes except the last four are added to an array of indexes to skip, and the data is then resorted from smallest to largest index. The $IDFT$ will then be performed on the samples to produce h_3'. This data is then written to a file named "$inv\_3.txt$" using "$writeComplex$".

Exercise 3

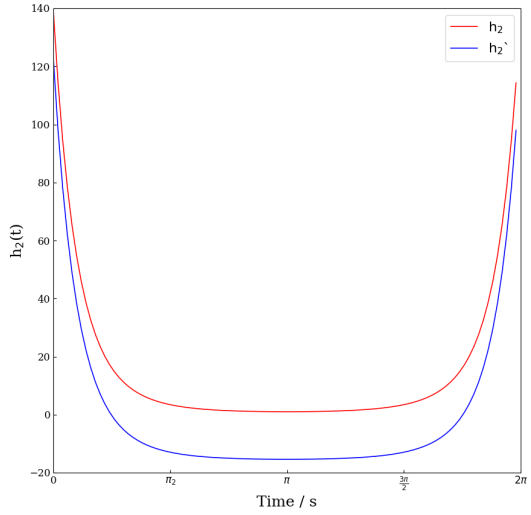


(a) Real components of $h1$ & $h1'$

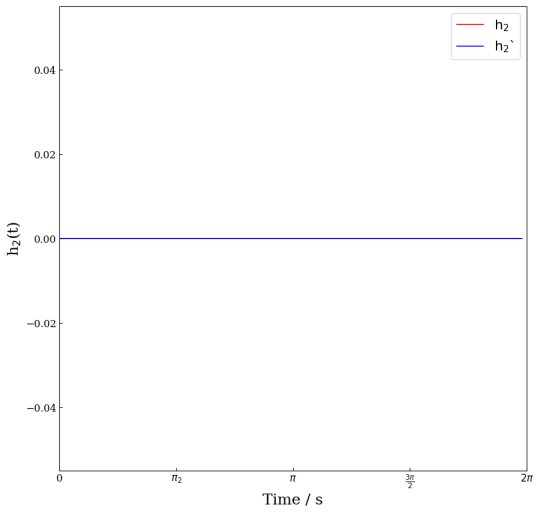


(b) Imaginary components of $h1$ & $h1'$

Figure 1: $h1$ & $h1'$.

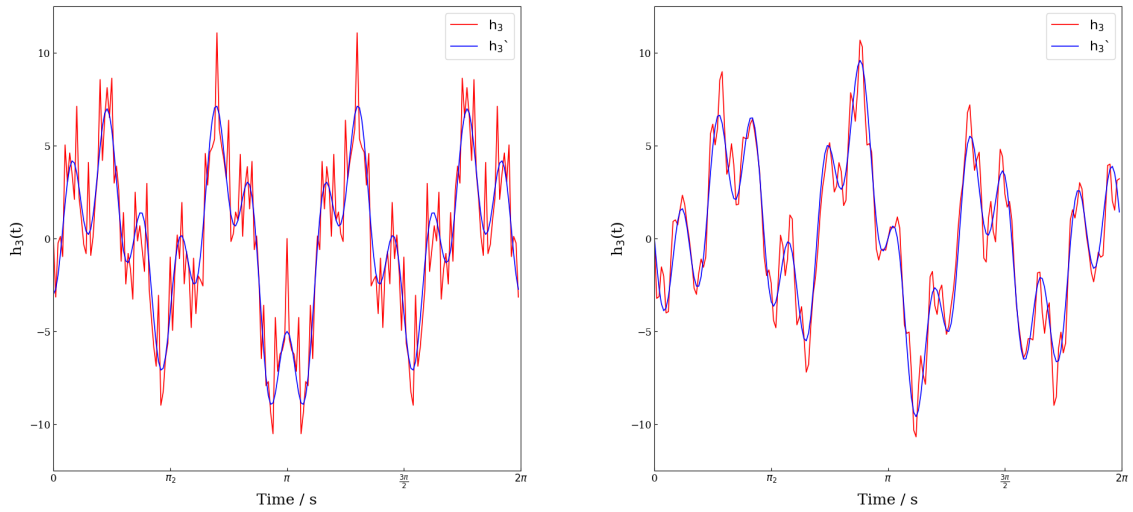


(a) Real components of $h2$ & $h2'$



(b) Imaginary components of $h2$ & $h2'$

Figure 2: $h2$ & $h2'$.



(a) Real components of $h3$ & $h3'$

(b) Imaginary components of $h3$ & $h3'$

Figure 3: $h3$ & $h3'$.

Exercise 4

When the discrete Fourier transform is applied to data produced by a function of time, the result is frequency data composed of complex numbers. This frequency data relates the magnitude of the frequency to its significance in the original data, and the argument of the frequency to the phase shift of the sine wave at that frequency.

Figure 1

Figure 1 shows two subfigures of h_1 and h_1'. Once Euler's formula is considered, it is fairly simple to see that the components of h_1 are given by $cos(t) + cos(5t)$ and $sin(t) + sin(5t)$ for the real and imaginary parts respectively. These waves can both be broken down into the sum of two sine/cosine waves with frequencies of 1 rads^{-1} and 5 rads^{-1}.

After applying the DFT to obtain H_1, the IDFT is applied while ignoring the $n = 1$ index, producing h_1'. The main feature of h_1' is that the real and imaginary parts are both normal cosine and sine waves respectively, both with a frequency of 5 rads^{-1}.

This can be interpreted as when the signal is converted from the frequency domain to the time domain without considering the contribution due to the value of $H_1(\omega_1)$, it only leaves the higher frequency signal to be reproduced by the inverse transformation.

Figure 2

Figure 2 shows the components of h_2 and h_2' overlayed. The imaginary component remained zero in both the original and transformed versions. However, the real component for h_2' is translated downwards. In this case, the $IDFT$ was performed while skipping the zeroth index which was also the index with the frequency of largest complex magnitude and contributes

most in the original signal. As the the zeroth index is skipped for every calculation, every point in the result is shifted $\frac{H_2(\omega_0)}{N}$ downwards, which in this case is roughly 16.38.

Figure 3

Figure 3 shows two versions of h_3 and h_3'. The first shows the complex amplitudes of h_3 & h_3' overlayed, whereas the second shows an overlay of the real & imaginary components. In this case, the IDFT only considers the four data points in H_3 (DFT of h_3) with the largest complex amplitudes to produce h_3'.

Performing the IDFT to only the four largest magnitudes in H_3 is analogous to regenerating the original signal from the four most dominant frequencies in the frequency domain, but ig. As a result we get a partial, but not total, reconstruction of the original signal.

H_1

The printed results for H_1 show mostly zeroes, except two values of $100 + 0i$ at the first and fifth indexes. This makes sense when considering there are only two frequencies present in h_1, 1 rads^{-1} & 5 rads^{-1}. Additionally, both frequencies contribute to the signal equally which is reflected in the frequency domain by their equal magnitudes.

H_2

The Fourier transform of h_2, like the original, is without imaginary components.

Code

```c
#include <math.h>       // pow, cos, sin, sqrt
#include <stdbool.h>    // bool
#include <stdio.h>      // printf, fprintf, scanf
#include <stdlib.h>     // malloc, free, qsort, etc.

// Global variables for pi & e
double pi = 3.14159265358979323846;
double e = 2.7182818284590452354;

// Generic write error message, then exit execution
void errorExit(const char *err_msg) {
    fprintf(stderr, err_msg);
    exit(-1);
}

// ----------------------------------------------------
// Implementing some simple complex number functionality

typedef struct Complex {
```

```
    double r; // Real part;
    double i; // Imaginary part;
} Complex;

// Calculate square of modulus for complex numbers.
double cMod(const Complex z) { return sqrt(z.r * z.r + z.i * z.i); }

double cModPtr(const Complex *z) { return cMod(*z); }

Complex cConjugate(Complex *z) {
    z->i *= -1;
    return *z;
}

void printComplex(const Complex *z) {
    printf("(%lf + %lfi)", z->r, z->i);
}

// Compares the amplitudes of two Complex numbers
int compareComplex(const void *a, const void *b) {
    // Calculate amplitude of Complex vars
    const double mod_fa = cModPtr((const Complex *) a);
    const double mod_fb = cModPtr((const Complex *) b);

    // Comparison of two Complex numbers
    // If nan values detected --> error out;
    // else --> return appropriate value for comparison
    if (isnan(mod_fa) || isnan(mod_fb)) {
        errorExit("\n<compareComplex> nan values provided.\n");
    } else if (mod_fa > mod_fb) {
        // a > b --> 1
        return 1;
    } else if (mod_fa < mod_fb) {
        // a < b --> -1
        return -1;
    }
    // Must be equal... Return 0
    return 0;
}
```

```
// This function has been reformatted due to not fitting
// in the LaTeX document.
// Write Complex data to specified file
void writeComplex(const double *time_data,
```

```
                const Complex *z_data,
                const size_t N,
                const char *filename)
{
    // Open file, check for failure
    FILE *fp = fopen(filename, "w");
    if (!fp) {
        errorExit("\n<writeComplex> Failed to open file.\n");
    }

    // Print column titles
    fprintf(fp, "time,real,imag\n");

    // Write time & Complex data
    size_t i;
    for (i = 0; i < N; ++i) {
        fprintf(fp, "%lf,%lf,%lf\n", time_data[i], z_data[i].r, z_data[i].i);
    }

    // Close file
    fclose(fp);
}

// End of complex number functions
// -------------------------------------------------------
// Simple struct for returning data in 3j+

typedef struct Measurement {
    size_t n;
    double time;
    Complex z;
} Measurement;

// Custom comparison function for qsort with measurements
int compareMeasurement(const void *a, const void *b) {
    // a & b converted back to Measurement
    const Measurement *a2 = (Measurement *) a;
    const Measurement *b2 = (Measurement *) b;

    // Return result of comparison between the amplitudes of
    // the complex numbers stored in a2 & b2.
    return compareComplex((void *) &a2->z, (void *) &b2->z);
}

// End of Measurement struct/functions
// -------------------------------------------------------
// General functionality:
// Functions which facilitate completing the questions
```

```c
// Compares two size_t
int compareSize_t(const void *a, const void *b) {
    return (*(size_t *) a - *(size_t *) b);
}

// Implementation of h1 & h2
Complex h_1(const double time) {
    // Calculate h1 using Euler's formula.
    Complex result = {
            .r = cos(time) + cos(5 * time),
            .i = sin(time) + sin(5 * time)
    };
    return result;
}

Complex h_2(const double time) {
    // calculate exponent, theta, of h2
    double theta = (time - pi) * (time - pi) / 2;
    // Initialize Complex number with result.
    Complex result = {
            .r = exp(theta),
            .i = 0
    };
    return result;
}

// Implementation of binary search to for a given value in a list
// ls MUST be sorted
int checkIdx(const size_t *ls, const size_t sz, const size_t x) {
    size_t current_idx, left_idx = 0, right_idx = sz - 1;
    // Prechecks:
    // If NULL passed to ls or sz = 0, empty list --> no skipped values / value not found
    if (ls == NULL || sz == 0) {
        return 0;
    }
    // Check when sz == 1 as while loop --> segfault if sz == 1
    if (sz == 1) {
        // Value found!
        if (ls[0] == x)
            return 1;
            // Value not found!
        else
            return 0;
    }

    // Until all indexes are checked
    while (left_idx <= right_idx) {
```

```
        // Check index in the middle between left_idx & right_idx
        current_idx = floor((left_idx + right_idx) / 2.);

        if (ls[current_idx] < x) {
            // Sorted list --> if ls[current_idx] < x, then all vals before current_idx < x.
            left_idx = current_idx + 1;
        } else if (ls[current_idx] > x) {
            // Sorted list --> if ls[current_idx] > x, then all vals after current_idx > x.
            right_idx = current_idx - 1;
        } else {
            // Value found!
            return 1;
        }
    }
    // Value not found!
    return 0;
}

// Produces range of N long doubles from start -> end exclusive.
// Simple version of numpy.linspace
double *linspaceD(const double start, const double end, const size_t N) {
    // Allocate memory for result, calculate increment size for N values in given range
    double *arr = (double *) malloc(N * sizeof(double));
    if (!arr) {
        errorExit("\n<linespaceLd> malloc failed.\n");
    }

    // Calculate increment for N evenly spaced values from start -> end
    // 'val' keeps track of current value
    double increment = (double) (end - start) / N;
    double val = start;

    // Add values to arr, incrementing val every time
    size_t i;
    for (i = 0; i < N; i++) {
        arr[i] = val;
        val += increment;
    }

    return arr;
}

// Same as linspaceD, except values are generated using provided function, f.
Complex *linspaceComplex(Complex (*f)(double), const double *samples, const size_t N) {
    // Alloc memory & check for results
    Complex *arr = (Complex *) malloc(N * sizeof(Complex));
    if (!arr) {
        errorExit("\n<linespaceComplex> malloc failed.\n");
```

```
    }

    // Generate value for every sample in samples using specified function
    size_t i;
    for (i = 0; i < N; ++i) {
        arr[i] = f(samples[i]);
    }
    return arr;
}

// This function has been reformatted due to not fitting
// in the LaTeX document.
// Discrete Fourier Transform taking array of samples
// samples --> Array of pointers to Complex values to transform.
// N --> Number of elements in samples.
// skip_n --> Array of indexes to skip or include, depending on what skip is set to
// sz --> Number of elements in skip_n
// IDFT --> Boolean for whether DFT or IDFT is being calculated
Complex *DFT(const Complex *samples,
             const size_t N,
             const size_t *skip_n,
             const size_t sz,
             const bool IDFT)
{
    // Alloc memory for resulting array & declare vars
    Complex *arr = (Complex *) malloc(N * sizeof(Complex));
    if (!arr) {
        errorExit("\n<DFT> malloc failed.\n");
    }

    // Sorting skip_n to ensure binary search in checkIdx works.
    // Only necessary if skip_n exists
    if (skip_n != NULL && sz != 0) {
        qsort((void *) skip_n, sz, sizeof(size_t), compareSize_t);
    }

    // H_n & h_k keep track of current Complex numbers
    // theta & theta_k store value of exponent h_k(t_k).exp(-2.pi.n.k/N)
    // Calculated via Euler's formula
    Complex H_n, h_k;
    double theta_nk, theta_n, theta = 2. * pi / N;

    // If performing DFT (IDFT == false), multiply theta by -1
    if (!IDFT)
        theta *= -1;

    size_t n, k;
```

```
    // For every values H_n, sum contributions of all h_k then add to resulting array
    for (n = 0; n < N; n++) {
        // Calculate theta for this index n
        theta_n = theta * n;

        // initialize H_n
        H_n.r = 0.;
        H_n.i = 0.;

        for (k = 0; k < N; k++) {
            if (!checkIdx(skip_n, sz, k)) {
                // Adjust value of theta for current sample
                theta_nk = theta_n * k;

                // Retrieve k'th sample
                h_k = samples[k];

                // (a + bi)(c + di) = (ac - bd) + (ad + bc)i
                // h_k(t_k).exp(-2.pi.n.k/N):
                H_n.r += (h_k.r * cos(theta_nk) - h_k.i * sin(theta_nk));
                H_n.i += (h_k.r * sin(theta_nk) + h_k.i * cos(theta_nk));
            }
        }
        // Add to result array
        arr[n] = H_n;
    }

    return arr;
}

// This function has been reformatted due to not fitting
// in the LaTeX document.
// Calculates IDFT of N provided samples.
// Parameters are the same as DFT, minus bool IDFT which is implicit
// in the use of the function.
Complex *IDFT(const Complex *samples,
              const size_t N,
              const size_t *skip_n,
              const size_t sz)
{
    // Apply DFT to samples, setting IDFT param as true and passing skip_n and sz on to DFT
    Complex *result = DFT(samples, N, skip_n, sz, true);

    // Divide all members of result by N.
    for (size_t i = 0; i < N; ++i) {
        result[i].r /= N;
        result[i].i /= N;
    }
```

```
    return result;
}

// End of general functions
// ------------------------------------------------
// Question answers

// Implementation for 3b
Complex **q_3b(const double *times, const size_t N) {
    // Allocating memory for arrays to store results in
    Complex *h1_vals = (Complex *) malloc(N * sizeof(Complex));
    Complex *h2_vals = (Complex *) malloc(N * sizeof(Complex));
    Complex **results = (Complex **) malloc(2 * sizeof(Complex *));

    if (!h1_vals || !h2_vals || !results) {
        errorExit("\n<q_3b> malloc failed.\n");
    }

    // Generate values of h_1(t) & h_2(t)
    h1_vals = linspaceComplex(h_1, times, N);
    h2_vals = linspaceComplex(h_2, times, N);

    // Write to file
    writeComplex(times, h1_vals, N, "h1.txt");
    writeComplex(times, h2_vals, N, "h2.txt");

    // Add to results array so they can be returned and used later
    results[0] = h1_vals;
    results[1] = h2_vals;

    return results;
}

// Questions 3.d & 3.e:
//         - 3.d: Perform DFT on h1(t) & h2(t) --> H1(w) & H2(w)
//                typeof(Hn(w)) = Complex*
//         - 3.e: Print results to screen
// Also returns array of {H1(w), H2(w)} for use in part 3.f
Complex **q_3de(const double *times, Complex **samples, const size_t N) {
    size_t i;
    // Allocating memory for results; will store {H1, H2}
    Complex **results = (Complex **) malloc(2 * sizeof(Complex *));
    if (!results) {
        errorExit("\n<q_3de> malloc failed.\n");
    }

    // Calculating H1 & H2 using DFT
```

```
    Complex *H1 = DFT(samples[0], N, NULL, 0, false);
    Complex *H2 = DFT(samples[1], N, NULL, 0, false);

    // Add H1 & H2 --> results
    results[0] = H1;
    results[1] = H2;

    // Printing H1 & H2 using custom print_Complex function
    printf("\nPrinting H1:\n");
    for (i = 0; i < N; ++i) {
        printf("t = %lf\tH1 = ", times[i]);
        printComplex(&H1[i]);
        putchar('\n');
    }
    printf("\nPrinting H2:\n");
    for (i = 0; i < N; ++i) {
        printf("t = %lf\tH2 = ", times[i]);
        printComplex(&H2[i]);
        putchar('\n');
    }

    return results;
}

// Applies Inverse Fourier Transform (IDFT) to provided sets of H_x in samples.
// In this case samples will provided by result of q_3de.
Complex **q_3f(Complex **samples, size_t N) {
    // Arrays containing the indexes to skip in H1 & H2
    size_t *h1_prime_skip = (size_t *) malloc(sizeof(size_t));
    size_t *h2_prime_skip = (size_t *) malloc(sizeof(size_t));

    h1_prime_skip[0] = 1;
    h2_prime_skip[0] = 0;

    // Calculate Inverse Fourier Transform of H1 & H2 respectively.
    Complex *h1_prime = IDFT(samples[0], N, h1_prime_skip, 1);
    Complex *h2_prime = IDFT(samples[1], N, h2_prime_skip, 1);

    // Freeing h1_pr... & h2_pr... to prevent memory leaks.
    free(h1_prime_skip);
    free(h2_prime_skip);

    // Allocate memory for results, check for fail.
    Complex **results = (Complex **) malloc(2 * sizeof(Complex *));
    if (!results) {
        errorExit("\n<q_3f> malloc failed.\n");
    }
```

```c
    // Pack h1_prime & h2_prime into results so it can be returned for q_3f
    results[0] = h1_prime;
    results[1] = h2_prime;

    return results;
}

void q_3g(Complex **data, double *times, size_t N) {
    // Write data to respective files
    writeComplex(times, data[0], N, "inv_1.txt");
    writeComplex(times, data[1], N, "inv_2.txt");
}

// Read data in from h3.txt, return as array of Complexes
Measurement *q_3i(const char *filename, size_t N) {
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        errorExit("\n<q_3h> Failed to open h3.txt.\n");
    }

    // Allocating vars to keep track of array size and arrays for data
    Measurement *results = (Measurement *) malloc(N * sizeof(Measurement));
    if (!results) {
        errorExit("\n<q_3h> malloc failed.\n");
    }

    int n;
    double time, real = 0., imag = 0.;
    Complex z;
    size_t sz = 0;

    // Use fscanf to read from file
    while (fscanf(fp, "%d, %lf, %lf, %lf", &n, &time, &real, &imag) != EOF && sz < N) {
        // Adding values to results
        z.r = real;
        z.i = imag;

        results[sz].n = n;
        results[sz].time = time;
        results[sz].z = z;

        sz++;
    }

    fclose(fp);

    return results;
}
```

```
// Apply DFT to h3
Measurement *q_3j(const Measurement *data, const size_t N) {
    Complex *z_arr = (Complex *) malloc(N * sizeof(Complex));
    if (!data) {
        errorExit("\n<q_3j> malloc failed.\n");
    }

    // Copying data into array
    // DFT is then applied to it
    size_t i;
    for (i = 0; i < N; ++i) {
        z_arr[i] = data[i].z;
    }

    // Applying DFT to data with DFT_samples
    Complex *DFT_data = DFT(z_arr, N, NULL, 0, false);

    Measurement *results = (Measurement *) malloc(N * sizeof(Measurement));
    if (!results) {
        errorExit("<q_3j> \nFailed malloc for results in q_3j.\n");
    }

    // Copying transformed data into new array.
    for (i = 0; i < N; ++i) {
        results[i].n = data[i].n;
        results[i].time = data[i].time;
        results[i].z = DFT_data[i];
    }

    // Freeing memory not being passed out of the scope
    free(z_arr);
    free(DFT_data);

    return results;
}

// Applies IDFT to 4 terms of H3 with largest amplitude
Complex *q_3k(Measurement *samples, const size_t N, size_t n_largest_vals) {
    // Allocating memory for transformed version of data
    Complex *result = (Complex *) malloc(N * sizeof(Complex));
    if (!result) {
        errorExit("\n<q_3k> malloc failed.\n");
    }

    // Sorting samples_sorted based on magnitude of Complex numbers stored within
    qsort(samples, N, sizeof(Measurement), compareMeasurement);
```

```
    // Variables required for
    size_t i, n;
    size_t *skip_n;

    n = N - n_largest_vals;
    if (n <= 0 || n > N) {
        printf("n = %ld\n", n);
        errorExit("\n<q_3k> Performing IDFT on invalid n.");
    }

    // Allocate memory for values to skip
    skip_n = (size_t *) malloc(n * sizeof(size_t));
    if (!skip_n) {
        errorExit("\n<q_3k> malloc failed.\n");
    }

    // Copy up to n-th value into skip_n
    for (i = 0; i < n; ++i) {
        skip_n[i] = samples[i].n;
    }

    // Resort back into order by index.
    qsort((void *) samples, N, sizeof(Measurement), compareSize_t);

    // Create array of Complex vals from original samples set
    Complex *z_arr = (Complex *) malloc(N * sizeof(Complex));
    if (!z_arr) {
        errorExit("\n<q_3k> malloc failed.\n");
    }

    // Copy Complex data to perform IDFT on.
    for (i = 0; i < N; ++i) {
        z_arr[i] = samples[i].z;
    }

    result = IDFT(z_arr, N, skip_n, n);

    // Free memory to prevent leak.
    free(skip_n);
    free(z_arr);

    return result;
}

// This function has been reformatted due to not fitting
// in the LaTeX document.
void q_3l(const double *times,
          const Complex *data,
```

```
          const size_t N,
          const char *filename)
{
    writeComplex(times, data, N, filename);
}

// End of question answers
// -----------------------------------------------------
// main()

int main(int argc, char *argv[]) {
    // Set number of samples & generate time values
    size_t i, N = 100;
    double *times = linspaceD(0, 2 * pi, N);

    // Complete Q3.b, then pass the values --> array of array of Complexes
    Complex **h1_and_h2 = q_3b(times, N);

    // Complete Q3.d & Q3.e, then pass the values --> array of array of Complexes
    Complex **H1_and_H2 = q_3de(times, h1_and_h2, N);

    // Perform IDFT on H1 & H2
    Complex **prime_h1_and_h2 = q_3f(H1_and_H2, N);

    // Write h1' & h2' to file
    q_3g(prime_h1_and_h2, times, N);

    // Data provided uses N=200
    N = 200;

    // Read in h3
    Measurement *h3 = q_3i("h3.txt", N);

    // Apply DFT to h3
    Measurement *H3 = q_3j(h3, N);

    // Apply IDFT to H3 only considering 4 largest magnitudes
    Complex *i_h3 = q_3k(H3, N, 4);

    times = (double *) realloc(times, N * sizeof(double));
    if (!times) {
        errorExit("\nFailed malloc for times in main.\n");
    }

    for (i = 0; i < N; ++i) {
        times[i] = h3[i].time;
    }
```

```
    q_3l(times, i_h3, N, "inv_3.txt");

    // Freeing memory
    free(times);
    free(h1_and_h2[0]);
    free(h1_and_h2[1]);
    free(h1_and_h2);
    free(H1_and_H2[0]);
    free(H1_and_H2[1]);
    free(H1_and_H2);
    free(prime_h1_and_h2[0]);
    free(prime_h1_and_h2[1]);
    free(prime_h1_and_h2);
    free(h3);
    free(H3);
    free(i_h3);

    return 0;
}
```