HEILBRONN UNIVERSITY
OF APPLIED SCIENCES

Faculty of Engineering (TE)
Automotive Systems Engineering

Project for Digital Signal Processing

Project Documentation
Master Automotive Systems Engineering
by

Jakob Kurz & Mattis Tom Ritter
(Matriculation-Nr. 210262 & 210265)

presented in Wintersemester 2024 / -25 at

Hochschule Heilbronn

Reviewer:	Prof. Dr. Volker Stahl
Levy:	31. December 2024

Table of Contents

List of Abbreviations

DFT Discrete Fourier Transform

FFT Fast Fourier Transform

List of Figures

1. Basic Implementations

1.1 DFT vs. FFT

1.1.1 Implementation Overview

The implementations of the DFT and FFT are provided in the files `dft.py` and `fft.py`, respectively. The DFT computes the Fourier Transform by performing a matrix-vector multiplication using the complex-valued matrix B. This matrix represents the discrete Fourier kernel, with elements defined as $B[k, l] = e^{-2\pi ikl/N}$. The DFT directly calculates the transform by multiplying the input vector f with B, resulting in a computational complexity of $O(N^2)$.
In contrast, the FFT optimizes the Fourier Transform computation by reducing the complexity to $O(N \log N)$. The recursive FFT employs a divide-and-conquer approach, splitting the input signal into smaller subproblems and recursively computing the transform. The iterative FFT further optimizes this process by eliminating recursion and using butterfly operations, which enable efficient in-place computations.

1.1.2 Testing and Verification

The correctness of the DFT and FFT implementations is validated through comprehensive testing, focusing on both individual and comparative evaluations of the methods.

Tests for Individual Implementations

The tests in `test_dft.py` verify the correctness of the DFT and its optimized variants. These include reconstructing random real and complex signals using the inverse DFT (IDFT), ensuring that IDFT(DFT(f)) = f and DFT(IDFT(z)) = z. Additional tests compare the outputs of the standard and optimized implementations, as well as against NumPy's FFT. A cosine wave test checks the magnitude of the transformed coefficients against expected values.
Similarly, `test_fft.py` evaluates both the recursive and iterative FFT implementations. Tests confirm equivalence with NumPy's FFT results for real and complex signals, ensure

that IFFT(FFT(f)) = f, and verify that the recursive and iterative FFT produce identical outputs. These tests validate the accuracy and consistency of each implementation.

Comparative Testing of DFT and FFT

The script `test_dft_fft.py` compares the DFT with the recursive and iterative FFT. A sinusoidal test signal is processed by all three methods, and their outputs are compared element-wise within a tolerance of 10^{-9}. This ensures numerical equivalence between the methods, confirming that the DFT and FFT implementations produce consistent results.

1.1.3 Verification of Similarities

The script `vis_dft_fft_simmiliar.py` visualizes the similarities between the methods. In the frequency domain, all methods accurately identify the signal's frequency components. However, only half of the spectrum is shown due to the symmetry inherent to real-valued signals (see Fig. 1.1 left). In the time domain, all methods reconstruct the original signal with minimal deviations (see Fig. 1.1 right).

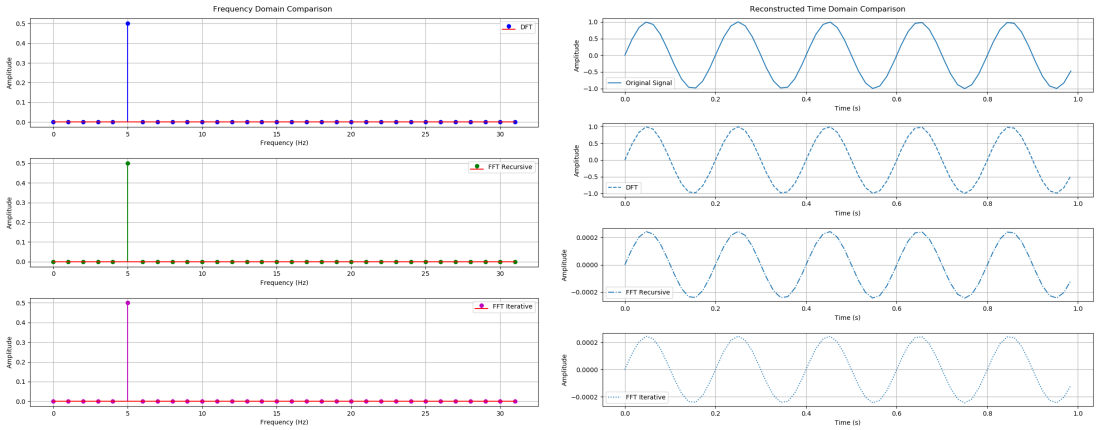

Figure 1.1: Frequency domain comparison (left) and Time-domain reconstruction validation (right)

1.1.4 Comparison of Differences

To highlight differences, the script `vis_dft_fft_different.py` visualizes the computational time and memory usage for the DFT and FFT.

Figure 1.2 compares execution time and memory usage for a fixed signal length. The DFT requires significantly more time due to its quadratic complexity of $O(N^2)$, while both FFT implementations achieve logarithmic complexity of $O(N \log N)$. The recursive FFT is slightly faster than the iterative FFT in this case. Regarding memory usage, the iterative FFT is the most efficient, whereas the recursive FFT demands more due to its reliance on recursion.

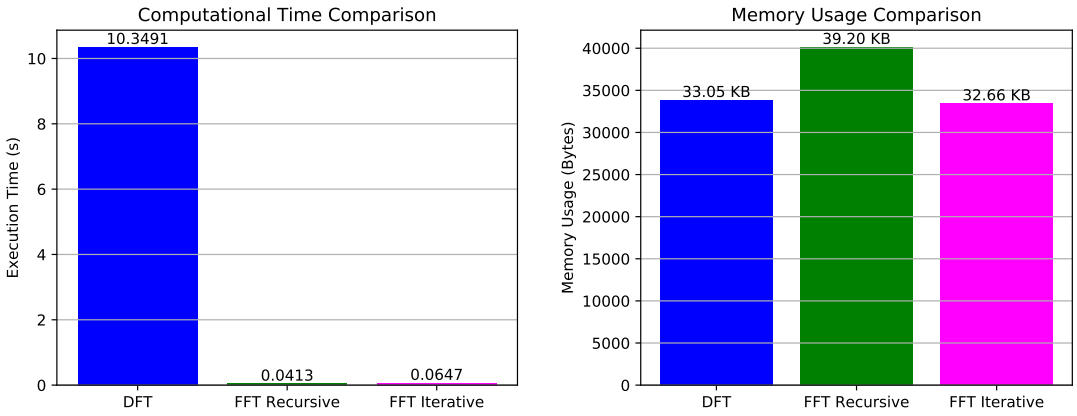

Figure 1.2: Computational time for fixed signal length (left). Memory usage comparison (right).

Figure 1.3 examines how execution time and memory usage scale with increasing signal length. The execution time for the DFT grows quadratically, whereas the FFT scales logarithmically. Memory usage increases quadratically for the DFT as well as for both FFT methods. Nevertheless, the iterative FFT outperforms the other methods in this scenario.

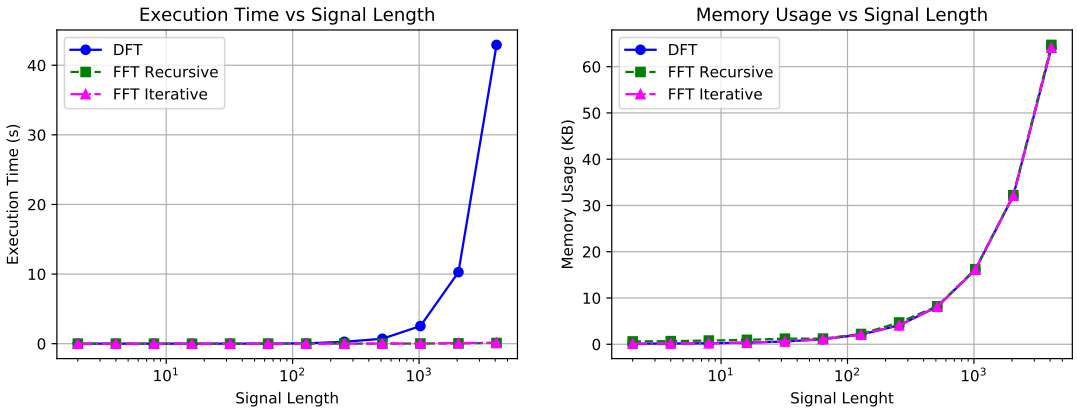

Figure 1.3: Execution time vs. signal length (left). Memory usage vs. signal length (right).

1.1.5 Issues during Development

When the performance tests of the FFT were conducted, issues with the implementation of the iterative algorithm surfaced. It allocated more than twice as much data a the DFT and recursive FFT and was significantly slower than the recursive algorithm. The problem was the bit inversion of the signal, where a string-based approach and a copy of the original signal have been used.
After changing this the iterative FFT requires the least data of all implementations and

is almost as fast as the recursive FFT. Usually the iterative algorithm should be faster than the recursive one, which was not achieved.
Further tries of improving the algorithm did not lead to a faster computation time. As that is only a slight performance issue and does not effect the overall functionality, the issue is neglected.

1.2 Modulation and Demodulation (Non- and Quadrature)

1.2.1 Implementation Overview

Modulation and demodulation are essential processes for signal transformation during transmission and recovery. The implementation includes non-quadrature and quadrature approaches, as defined in the Python script `modulation.py`. Non-quadrature modulation multiplies the input signal with a cosine function of a given modulation frequency, effectively shifting the signal in the frequency domain. Demodulation reverses this process by multiplying the modulated signal with twice the cosine function to recover the original signal. Quadrature modulation extends this concept by combining two input signals, one modulated with a cosine function and the other with a sine function, enabling dual-channel transmission. The quadrature demodulation separates these two components to recover the original signals.

1.2.2 Purpose and Testing

The `modulation.py` implementation is validated in the test script `test_modulation.py`, which covers various modulation and demodulation scenarios. For the `modulate` function, tests verify that the modulated signal aligns with expected cosine-modulated values for a range of input frequencies and amplitudes. The `demodulate` function is tested to ensure it accurately reconstructs the original signal after modulation and demodulation.
Quadrature modulation and demodulation undergo specific tests to confirm the proper separation and scaling of the two orthogonally modulated signals during demodulation. The tests include scenarios with independent input signals as well as edge cases, such as all-zero inputs or constant signals, to ensure numerical stability and accuracy. Each test employs assertions and comparisons against expected outcomes to validate the functionality and robustness of the methods.

1.2.3 Visualization of Modulation Processes

The script `vis_modulation.py` provides visual insights into the modulation and demodulation processes. Figure 1.4 illustrates these processes. The left panel shows the original, modulated, and demodulated signals for the non-quadrature case. Modulated signals are visibly shifted and scaled, while demodulated signals closely match the originals. The right panel depicts quadrature modulation and demodulation, highlighting the successful separation and reconstruction of two orthogonal signal components. These visualizations confirm the implementation's correctness and reliability.

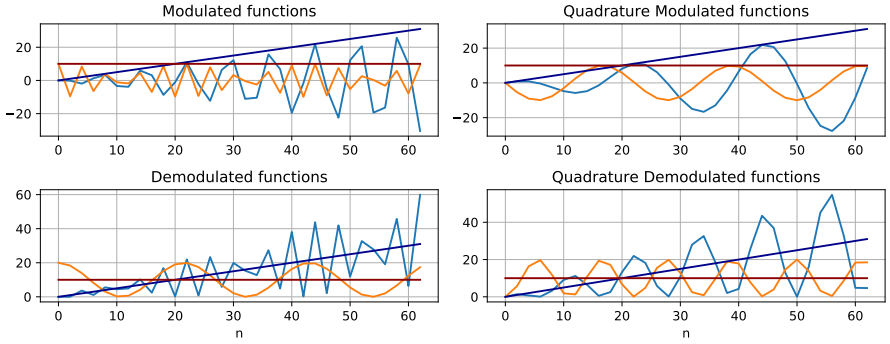

Figure 1.4: Non-quadrature modulation and demodulation (left). Quadrature modulation and demodulation (right).

1.3 Convolution

Convolution is implemented in time-domain (`/module/convolution.py`) and as fast convolution using the FFT (`/module/fast_convolution.py`). To verify the correctness of the implementations, a sawtooth signal is convolved with a shifted impulse signal. The results are the same for both implementations within a tolerance smaller than $\pm 10^{-15}$. When convolving with a shifted impulse signal, the convolution is equal to the shifted signal. This is shown in Figure 1.5.

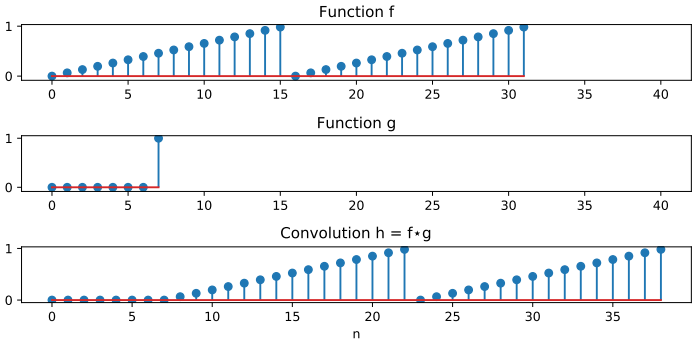

Figure 1.5: Convolution of a sawtooth signal with a shifted impulse signal.

1.4 Low-Pass Filter

The filter is implemented in `/module/low_pass_filter.py`. By convolving the input signal with the filter function of length N, the signal is low-pass filtered.

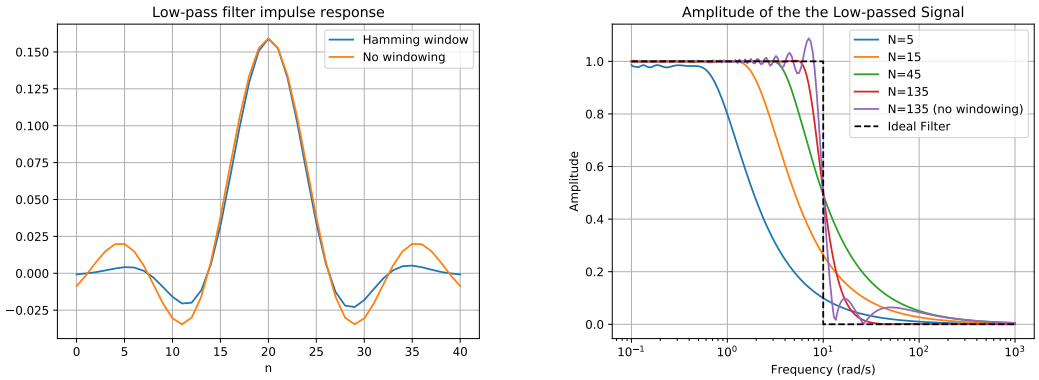

Figure 1.6: Impulse response of the low-pass filter with N=20 (left). Amplitude of the low-passed signal with different filter lengths (right).

As filter function, a *sinc* function with applied Hamming windowing is used. Figure 1.6 (left) shows the impulse response with and without a Hamming window. It can be observed that the Hamming window reduces the ripples at the beginning and end of the impulse response.

The frequency response of the low-pass filter is determined by filtering cosine waves with increasing frequencies and calculating the highest amplitude of the output signal. This is shown for different filter lengths in Figure 1.6 (right). Here, the cut-off frequency is 10 rad/s. It can be observed that the steepness of the transition band increases with the filter length. If no Hamming window is applied, there are ripples in the passband and the stopband.

The performance of the low-pass filter highly depends on the implementation of the convolution. Table 1.1 shows the timing comparison when using fast convolution and convolution in time-domain. For the given parameters, the fast convolution is on average more than 2.4 times faster then in time-domain.

	Fast Convolution	**Time-Domain Convolution**
Average time	0.2997 s	0.7241 s
Standard deviation	0.0397 s	0.0603 s
Min time	0.2598 s	0.6684 s
Max time	0.5171 s	1.269 s
Median time	0.2864 s	0.715 s

Table 1.1: Timing comparison for Low-Pass Filter using fast convolution and time-domain convolution. Function length = 20000, Filter length = 50, Number of tests = 200.

2. Overall system

The overall system recreates the transmission of two signals over a common medium using modulation. On the sender side, the signals are low-pass filtered, modulated and added. The sum of both modulated signals is then transmitted to the receiver, where the signal is demodulated and low-pass filtered again.

2.1 Testing with simple signals

The file `project.py` contains the implementation of the overall system. For testing, two simple signals are used. The first signal is a sum of cosine waves with different frequencies and phases (Equation 2.1) and the second signal is a sawtooth wave (Equation 2.2).

$$f_1(t) = 3 + cos(t+1) + 2cos(3t+2) - 5cos(4t-1) + cos(13t) \tag{2.1}$$

$$f_2(t) = 3 \cdot (t \mod \pi) \tag{2.2}$$

Both signals are sampled 32 times in the interval from 0 to 2π. Each step of the transmission is shown in Figure A.1 (top), where the left side depicts the operations in time-domain and the right side in frequency-domain, by showing the magnitude of the first half of the Fourier coefficients. The low-pass filtered signals are compared to the reconstructed signals in the bottom figure.
The cut-off frequency is choosen to be $\omega_c = 7\ rad/s$. As seen in the second row, the low-pass filter removes the high frequency components. Furthermore the sampling rate is increased by factor 2 after the low-pass filter, because modulation would not be possible without violating the sampling theorem. Then the signals are modulated with $\omega_{mod_1} = 8\ rad/s$ and $\omega_{mod_2} = 23\ rad/s$, so that their frequency bands do not overlap. After adding, demodulation and low-pass filtering, the reconstructed signals are nearly equal to the original, low-passed signals.
An alternative approach is to use quadrature modulation. This is shown in Figure A.2, where the signals are modulated with the same modulation frequency $\omega_{mod} = 8\ rad/s$, but with a cosine and a sine wave, respectively.

When a overlap of the frequency bands is introduced, the signals cannot be reconstructed properly. For this example the signals are sampled 64 times. The modulation frequencies are $\omega_{mod_1} = 8\ rad/s$ and $\omega_{mod_2} = 19\ rad/s$. This is shown in Figure A.3, where the signals contain noise of the other signal. Most significant is a noise in the sawtooth signal with a frequency of 7 rad/s.

2.2 Testing with sound files

The overall system is also tested with sound files (`project_soundfile.py`). The first sound file is a Jodler and the second a squeaking violin.

- `soundfiles/Jodler.wav`
- `soundfiles/Violine.wav`

The sampling rate is $f_s = 44100\ Hz$, respectively $w_s \approx 277088\ rad/s$. The signals are low-pass filtered with a cut-off frequency $\omega_c = 20000\ rad/s$. The modulation frequencies are $\omega_{mod_1} = 25000\ rad/s$ and $\omega_{mod_2} = 70000\ rad/s$. As the signals are well separated in the frequency domain and the sampling frequency is sufficient, the reconstruction is successful. The soundfiles after each operation are saved in the `soundfiles` directory, in order to listen to the (intermediate) results. Figure A.4 depicts the overall system with sound files.

A. Figures

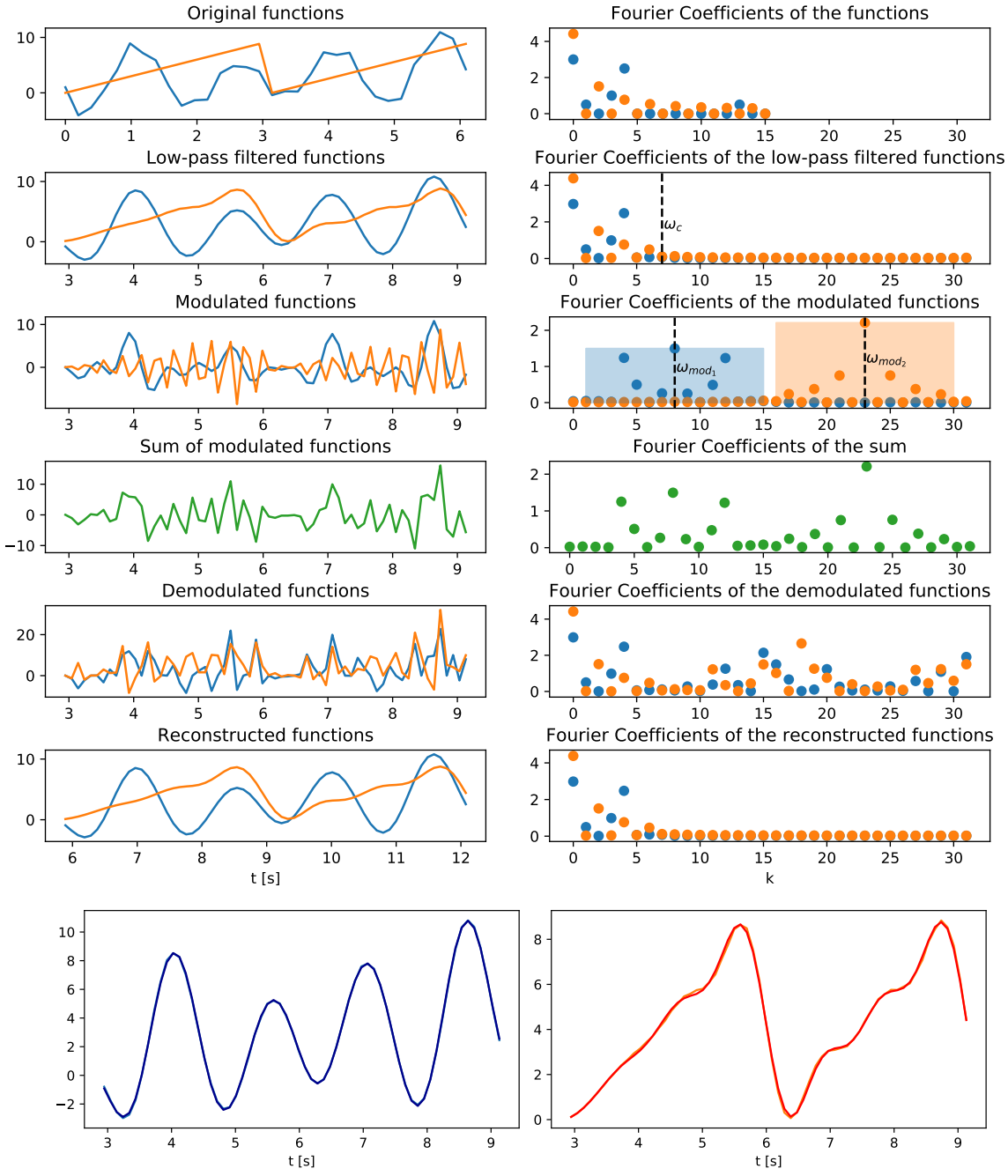

Figure A.1: Overall sytsem (top). Reconstructed vs. low-pass filtered signals (bottom).

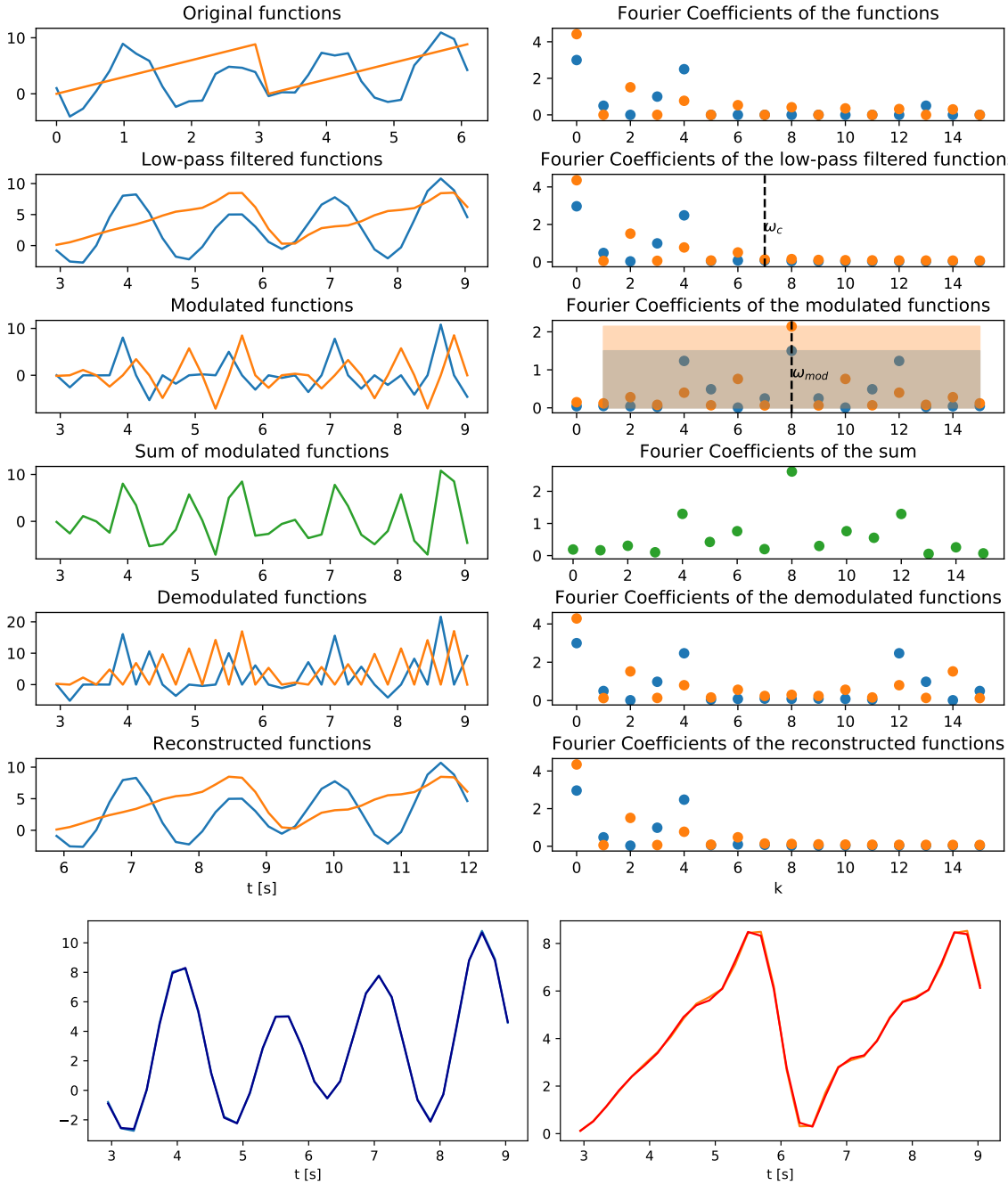

Figure A.2: Overall sytsem with quadrature modulation (top). Reconstructed vs. low-pass filtered signals (bottom).

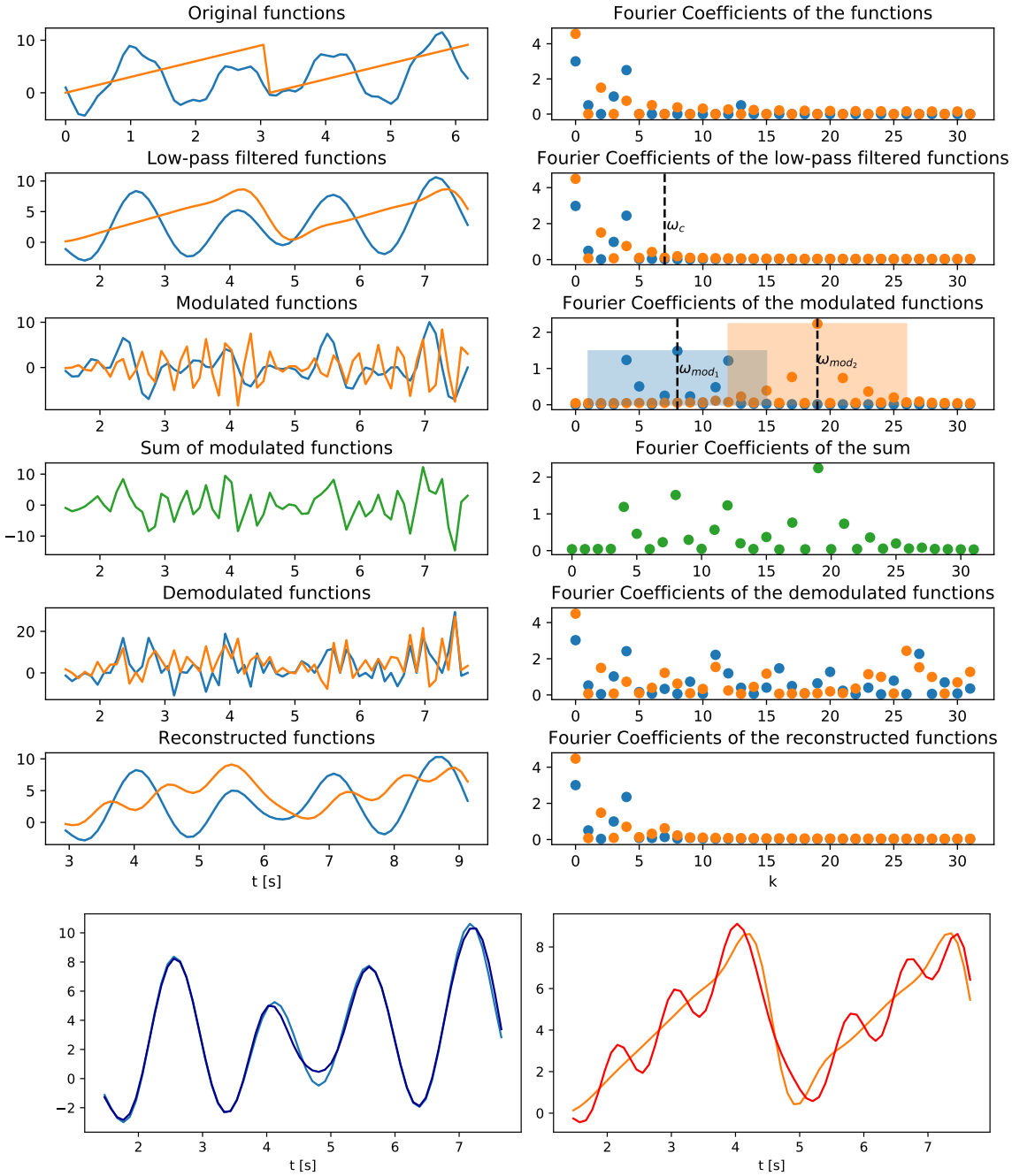

Figure A.3: Overall sytsem with overlapping frequency bands (top). Reconstructed vs. low-pass filtered signals (bottom).

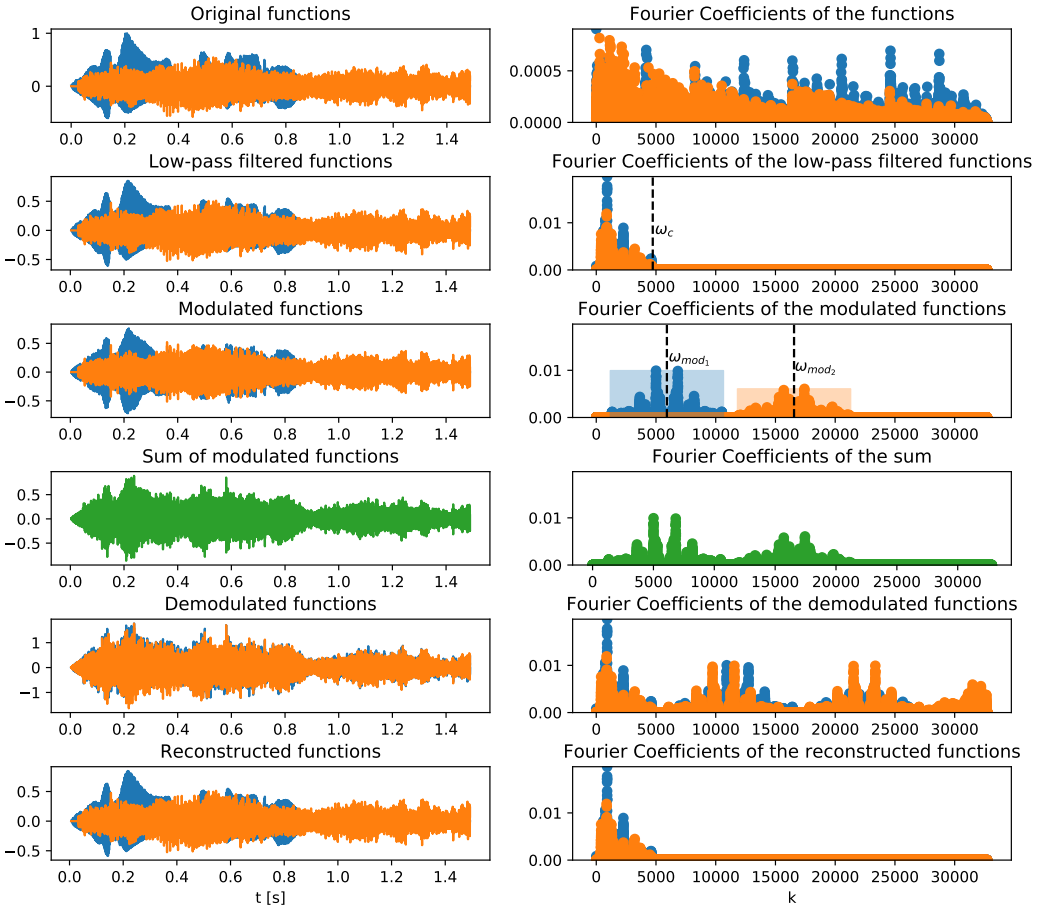

Figure A.4: Overall system with sound files.