WEBER STATE UNIVERSITY
Engineering, Applied Science & Technology

DEPARTMENT OF
ELECTRICAL & COMPUTER ENGINEERING

ECE 3210

Signals and Systems

Lab Manual

Author
Eric Gibbons

Version 7.0.0, Fall 2025

May 22, 2025

Contents

Acknowledgements

I am grateful to Paul Cuff, Jeff Ward, Fon Brown, and Chris Trampel for sharing their lab exercises. These materials were used to create much of the current lab sequence for this course.

Chapter 0

Introduction

In this lab sequence, students can put theory into practice by tackling real-world problems. They gain insights into the design process, from conceptualizing an idea to developing algorithms and implementing them in software or hardware. Along the way, students evaluate different design approaches and weigh the pros and cons of each strategy based on signal quality and implementation considerations.

0.1 Goals

The goal of the lab component of the course is threefold:

Hardware We will implement much of the theory we learn in class on real hardware. This will give you a better understanding of the limitations of the theory and the practical considerations that must be made when designing a system.

Implementing the algorithms We will implement signal processing algorithms—such as the Shazam music identification software routine—in Python.

Understanding the theory The theory covered elsewhere in the course can be incredibly dense. Completing the design flow by implementing the theory provides another angle for comprehension.

Understanding the tradeoffs and limitations In signal processing, we cannot expect something to work just because it looks good on paper. Limitations such as computational power, hardware tolerances, and practical design scaling often hinder our assumptions. There is no such thing as a free lunch in this field.

0.2 Lab techniques

0.2.1 Saving waveforms

We will use the Keysight equipment in NB 104/112 when working on hardware labs. Throughout the lab, we will need to record the data from the oscilloscope. You will be expected to make plots of the data you gather. You must save this data to a USB drive and plot it in Python. To do this, make sure your thumb drive is formatted as FAT. Plug it into the oscilloscope. The oscilloscope should recognize it by a spinning wheel at the top of the display and will eventually flash the drive's name. You can push the `Save/Recall` button. Select `Save`. You can change the save location to your drive by selecting the `Save to` option. You can also choose the channel source and the data format. When you have configured this, hit `Save`.

0.2.2 Frequency sweeps

We will repeatedly record our systems' frequency response in the lab. We could manually measure the phase and magnitude changes using a function generator and an oscilloscope, but that would be tedious. Our oscilloscope can automate this, saving you a lot of time. First, push the `Analyze` button on the oscilloscope. In the `Features` sub-menu, select `Frequency Response Analysis`. Select `Setup` to change the frequency sweep. Most of the time, we will be sweeping from some low frequency (say 10 Hz through Nyquist). Make sure you sample in frequency with enough points, which is also adjustable in the menu.

This relies on the oscilloscope's internal function generator. Ensure the system line input comes from your oscilloscope's `Gen Out` terminal. The pass-through channel on your board should go into Input 1 and the processed signal into Input 2. You can change this behavior in the oscilloscope menus, but I recommend sticking with the traditional convention.

To save this data, insert your USB stick into the oscilloscope. In the `Save/Recall` menu, set the format to `Frequency Analysis Data (*.csv)`. Push the `Save to USB` button to save the data. You can read this `csv` data into Python and plot it using `matplotlib`.

0.3 Lab reports

The lab reports in this course will use a standardized LaTeXtemplate that you can download on Canvas. LaTeXis a typesetting system that is widely used in academia. It is particularly useful for writing technical documents, as it can easily handle equations, figures, and tables. The template will provide a structure for your lab reports, including sections for the introduction, theory, procedure, results, and conclusions. You must fill in the content for each section based on your work in the lab.

There are many ways to write LaTeXdocuments, but I recommend using an online editor like Overleaf. Overleaf is a cloud-based LaTeXeditor that allows you to write and compile documents in your web browser. It is free to use and provides a user-friendly interface for writing LaTeXdocuments. You can also download the LaTeXsource files and compile them on your local machine if you prefer.

Chapter 1

Lab Basics

This laboratory sequence builds on what you have already learned in ECE 2260. The exercises in this workbook are designed to couple tightly with the material you are learning in ECE 3210. These labs can be divided into two basic categories:

1. Software (Chapters 6, 7, 11)
2. Hardware (Chapters 2, 3, 4, 5, 9, 10)

The software labs are designed to reinforce the ideas you learned in class, particularly concerning discrete-time signal processing. These will be done solely in Python. The hardware labs are designed to reinforce the ideas you learned in class, particularly about continuous-time signal processing. These will be done using simple analog circuits and some Python for simulation and basic computation. Each of these labs will have a considerable quantitative component, so you must be comfortable with the mathematical concepts you are learning in class. You are strongly advised to start working on this before coming to the lab period.

This exercise here will familiarize you with some of the lab procedures we will use throughout the semester so you can focus on the material you cover in the lab and not the lab's logistics.

1.1 Getting started

1.1.1 Parts list

You will need the following supplies to complete the lab.

Item	**Quantity**
$1\,\mathrm{k\Omega}$ resistor	1
$0.033\,\mathrm{\mu F}$ capacitor	1
$1\,\mathrm{mH}$ inductor	1

1.1.2 Repository

Please clone the repository linked on Canvas. This repository will have the following structure:

In this assignment, we will use these existing files and add some new ones as we go along.

1.2 Circuits

We are going to build a simple LRC circuit seen in Fig. 2.1. First, let's predict the output behavior using phasor analysis that we developed in ECE 1270. Assume that the circuit input is $f(t) = \cos(\omega t)$ where $\omega = 2\pi 10\,\text{kHz}$. Please derive $y(t)$ for this circuit given this $f(t)$.

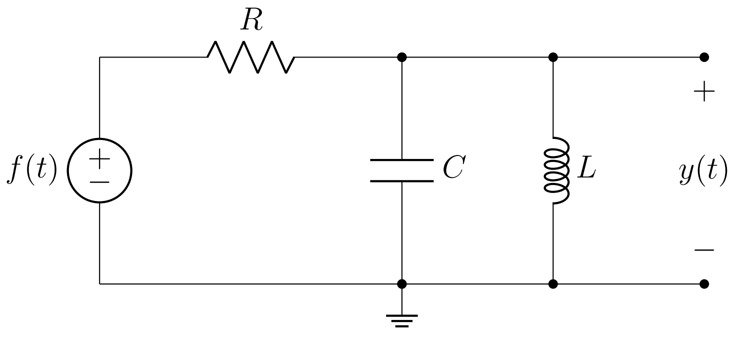

Figure 1.1: Series/parallel resonant circuit

Please use a breadboard to build this circuit. Attach the function generator and pass a 10 kHz, 2 V_{pp} sine wave through the circuit. Record two cycles of the input and output waveforms. You will need to save this data on the oscilloscope as a `.csv` file (see Sec. 0.2.1), load it into Python using whatever approach you prefer (e.g., `pandas`, `numpy`, manual `open`, etc.), and plot it. Overlay the theoretical input and output you derived earlier. Make sure your plots are appropriately labeled. Do not submit plots generated in Excel or similar software. Your plots need to look professional.

We need to measure this circuit's frequency response (Bode plot). In ECE 2260 you might have been taught to measure this by applying a sinusoidal input and measuring the output (magnitude and phase shift) at various frequencies. We will do this in a more automated way using oscilloscopes. Please see directions for how to do this in Section 0.2.2. Please perform a frequency sweep from 100 Hz to 100 kHz. Measure at least 100 points. You will need to save this data on your USB drive and plot the magnitude and phase shift of the circuit. Make sure your plots are appropriately labeled. The Bode plots you generate show the magnitude and phase shift of the circuit as a function of frequency. Note the magnitude and phase shift at 10 kHz and compare it to your theoretical prediction. Do not submit plots generated in Excel or similar software. Your plots need to look professional.

Questions and tasks

- What is $y(t)$?
- Make a plot of the observed circuit input/out waveforms and overlay the theoretical input/output.
- Plot the circuit's measured frequency response. Compare the measured magnitude and phase shift at 10 kHz to your theoretical prediction.

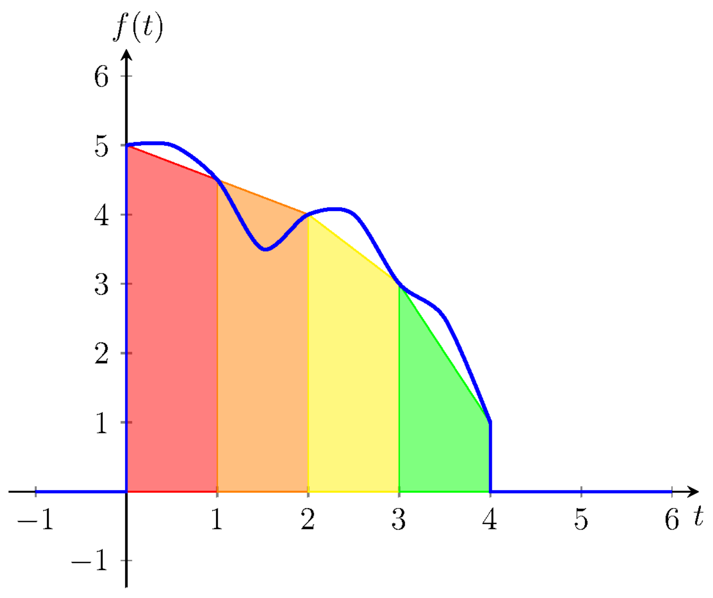

Figure 1.2: Some $f(t)$ and trapezoids for numerical integration

1.3 Computation

While you have learned a lot of cool tricks to solve integration problems analytically in your calculus courses, modern engineering uses computational tools. In this lab, we are going to look at the cumulative integral, which is to integrate from

$$y(t) = \int_{-\infty}^{t} f(\tau) d\tau.$$

This integral comes up a lot in signal processing—sometimes in evaluating the convolution integral and converting from a probability density function to a cumulative density function (take ECE 3430 for details). Unfortunately, these types of integrals can be tricky to solve, so oftentimes, we can approximate them using the trapezoidal rule. Suppose we were to wanting to discretize $y(t)$ to $y(t_k)$, we would be able to solve for y at point t_k using

$$y(t_k) \approx \sum_{k=1}^{N} \frac{f(t_{k-1}) + f(t_k)}{2} \Delta t_k.$$

where t_k is some value of time and t_{k-1} is the previous discretized time value. Similarly, Δt_k is the time between t_k and t_{k-1}.

Example

Suppose we want to perform on some $f(t)$ which you can see in the blue line in Fig. 1.2 (obviously, this is arbitrary, and $f(t)$ isn't described by some analytical function). If we were to integrate this numerically we might have two arrays:

```
f = [5, 4.5, 4, 3, 1]
f_t = [0, 1, 2, 3, 4]
```

To perform the numerical integration, we can step through each value using trapezoids. The first value of the output (`y[0]`) is the area of the red trapezoid. The value of `y[1]` is the area of the red *and* orange trapezoids. The value of `y[2]` is the area of the red, orange, *and* yellow trapezoids. Lastly, the value of `y[3]` is the area of *all* of the trapezoids. Note that we have five data points, so we can only have four trapezoids; thus, the length of `y` is four. We will set the time array `y_t` as the trapezoids' midpoints for this lab. Therefore, we will have the following results:

```
f = [4.75, 9, 12.5, 14.5]
f_t = [0.5, 1.5, 2.5, 3.5]
```

1.3.1 Numerical solution

First, we are going to implement this in Python. (There is a Python implementation in Scipy—see `scipy.integrate.cumtrapz()`—but writing this ourselves is a good exercise.)

In a file called `ece3210_lab01.py`, write a function called `py_cumtrapz()` that should have the following form.

```python
def py_cumtrapz(f, f_time):
    """This is a cumulative integral function implemented
    solely using Python/NumPy.

    Parameters
    ----------
    f : ndarray
        array of values of f(t)
    f_time : ndarray
        array of time values associated with f

    Returns
    -------
    ndarray
        1D array of values of y(t)
    ndarray
        1D array of time values associated with y

    """
```

The input `f` are the discrete values of $f(t)$ as an array. The input `f_time` are the corresponding time values for each value in `f`. The output `y` is an array for the computed values of $y(t)$. It should be noted that because we are computing the area of the trapezoids between values of `f`, the length of `y` will be `len(f) - 1`. Additionally, we will need to generate time points for `y`, which is the return value `y_time`. This will be the midpoint of the trapezoid (the middle of time points t_k and t_{k-1}. You cannot assume that there will be uniform spacing between each time value (i.e., Δt_k is not necessarily the same as Δt_{k-1}). Your function should be able to handle non-uniform spacing.

Because we are implementing this in Python, we should make this run quickly. YOU MAY NOT USE ANY FOR LOOPS IN THIS FUNCTION. You may use primitive NumPy (i.e., `ndarray` objects) and the `np.cumsum()` function, but nothing beyond that.

Let's look at how this function should be used with a simple example. Suppose we want to compute

$$y(t) = \int_{-\infty}^{t} 5e^{-\tau}u(\tau)d\tau \tag{1.1}$$

If we were to run this through your function, it would look like

```
>>> t = np.linspace(0,10,100000)
>>> f = 5*np.exp(-t)
```

The first line creates an array of the time values t. The second line creates the values of $f(t)$ based on those time values. We can find an array representing $y(t)$ and the associated time values using the following:

```
>>> y, t_y = py_cumtrapz(f, t)
```

where y is the array for $y(t)$ and t_y are the time values associated with the values in y.

Lastly, your function should raise a ValueError if the inputs f and f_time do not have the same dimensionality.

1.3.2 Analaytical solution

Let's compare our numerical solution to an analytical solution. Solve the integral in Eq. 1.1 using pencil and paper. In your ece3210_lab01.py file, write a function called analytical_integral() that should have the following form:

```
def analytical_integral(t):
    """This is the solution to the integral

    y(t) = \int^t_{-\infty} 5e^{-\tau} u(\tau)d\tau.

    Parameters
    ----------
    t : float
        time value to evaluate the integral at

    Returns
    -------
    float
        the value of the integral at time t
    """
```

Notice that this function only takes a single input, t, the time value to evaluate the integral at. The output is the value of the integral at that time value. This function should be able to handle scalar inputs. You can code it to accept arrays, but it will not be graded for that functionality.

Now that you have methods to compute the analytical and numerical solutions, you can compare the two. Make a Python plot showing the analytical and numerical solutions from $t \in [-1, 5]$. Make sure your plot is appropriately labeled.

1.3.3 Testing

Many of the computational exercises in the lab will have a testing component. This lab will test your functions using the unittest module. These tests will check the accuracy of your functions and whether you are not using coding practices that are not allowed. A successful test will look like this:

```
$ python test_lab01.py

Testing the analytical solution
.
Testing for loops
.
Testing the cumtrapz function
.
Testing for scipy.integrate usage
.
Testing the cumtrapz function with ValueError
.
----------------------------------------------------------------------
Ran 5 tests in 0.004s

OK
```

Questions and tasks

- Implement the `py_cumtrapz()` function.
- Implement the `analytical_cumulative_integral()` function.
- Compare the analytical and numerical solutions for the integral in Eq. 1.1 over the range $t \in [-1, 5]$ in a plot.

1.4 Deliverables

1.4.1 Submission

To submit your work, write the files as requested in your repository and push them back to GitHub. Your complete repository should contain the following files:

Your lab report should be a PDF file you will upload to Canvas. You must follow the template provided on Canvas. It must be written in LaTeX(see Sec. 0.3).

1.4.2 Grading

The lab assignment will have the following grade breakdown.

- circuit I/O plot: 15pts.
- frequency response plot: 20pts
- correct `py_cumtrapz()`: 30pts
- numerical integration plot: 15pts
- report: 20pts

Chapter 2

Impulse response

Impulse functions are essential tools in signal processing and provide valuable insights into the dynamics of systems. During this experiment, you will derive the impulse function of a circuit, plot it in Python, and measure it using an oscilloscope. By exploring the properties of impulse functions, we gain a deeper understanding of how signals interact with systems, enabling us to analyze and manipulate them effectively.

2.1 Procedure

2.1.1 Parts list

You will need the following supplies to complete the lab.

Item	**Quantity**
$1\,\mathrm{k\Omega}$ resistor	1
$0.033\,\mathrm{\mu F}$ capacitor	1
$1\,\mathrm{mH}$ inductor	1

2.1.2 Repository

Please clone the repository linked on Canvas.

2.1.3 Theoretical impulse response

Consider the system in Fig. 2.1. Let $R = 1\,\mathrm{k\Omega}$, $C = 0.033\,\mathrm{\mu F}$, and $L = 1\,\mathrm{mH}$. Derive a differential equation governing this circuit and solve for the impulse response $h(t)$ for this system analytically using $h(t) = P(D)y_n(t)u(t)$ as described in Section 2.3 of the text. Then write a Python script to plot $h(t)$ over the interval $[0,\ 300\,\mathrm{\mu s}]$. Please label the graph's axes of $h(t)$.

Questions and tasks

- Include brief details of your derivation of $h(t)$.
- What is the expression of $h(t)$ that you derived?

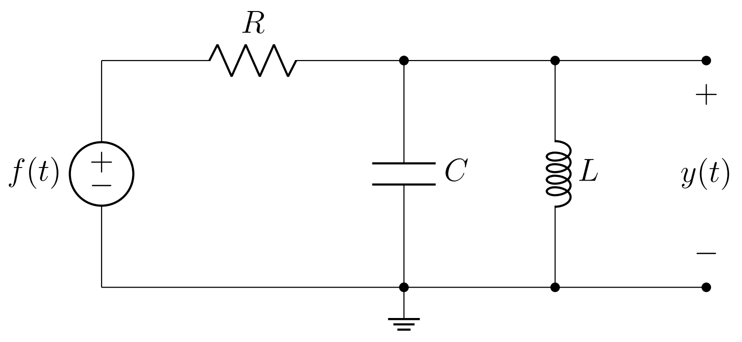

Figure 2.1: Series/parallel resonant circuit

2.1.4 Hardware implementation

The goal of this laboratory procedure is to verify that the impulse response computed in the preliminary section matches reality. Ideally, we would like to provide a delta function, $\delta(t)$, to the system's input and measure the response at its output. There is one major problem with this approach (aside from trying to safely measure a 30 kV signal): we do not have equipment that will generate a delta function. We can, however, come close enough for practical purposes.

Assemble the circuit shown in Fig. 2.1. Configure the function generator to produce pulses (not a square wave) at 1 kHz. (The 1 ms period gives the system sufficient time for the response to decay between pulses). Adjust the pulse duty cycle to 0.1% and adjust the low and high voltage levels to 0 V and 10 V, respectively. This will produce a pulse 1 µs wide and 10 V in amplitude. Show that this pulse approximates $10^{-5}\delta(t)$. (Hint, integrate it.)

Alternatively, you can use the `Pulse` waveform on the function generator. This will allow you to set the pulse width and period directly. Set the frequency to 1 kHz and the pulse width to 1 µs. Set the amplitude to 10 V and the offset to 5 V.

Compare the response shown on the oscilloscope to the impulse response you calculated in the prelab (considering the factor of 10^{-5}). Save the first 300 µs worth of data from the oscilloscope as described in Sec. 0.2.1. Plot the theoretical $h(t)$ and the measured impulse response on the same plot. Make sure the plot is properly labeled. Do these align?

Questions and tasks

- Plot the theoretical $h(t)$ on the same plot as the measured impulse function.
- Do these plots agree? Why or why not?

2.2 Deliverables

2.2.1 Submission

Write up the results of this lab in the technical memo format (see Canvas for a template example). Submit the PDF to Canvas. Upload your Python code to GitHub.

2.2.2 Grading

The lab assignment will have the following grade breakdown:

- Derivation: 15pts.
- Python plot: 20pts.
- Circuit plot: 20pts.
- Report: 35pts.

Chapter 3

Convolution

We can analyze any linear and time-invariant system by convolving its impulse response with the input signal. Convolution is a fundamental operation in signal processing and is used to model the output of a system when given its impulse response and input signal. In this experiment, you will derive the impulse response of a circuit, plot it in Python, and measure it using an oscilloscope. By exploring the properties of convolution, we gain a deeper understanding of how signals interact with systems, enabling us to analyze and manipulate them effectively.

3.1 Procedure

3.1.1 Parts list

You will need the following supplies to complete the lab.

Item	**Quantity**
$10\,\mathrm{k\Omega}$ resistor	1
$0.047\,\mu\mathrm{F}$ capacitor	1

3.1.2 Repository

Please clone the repository linked on Canvas.

3.2 Theoretical system responses

3.2.1 Impulse response

We will first need to determine the impulse response of our system. We will use a simple RC circuit as shown in Fig. 3.1. Let $R = 10\,\mathrm{k\Omega}$ and $C = 0.047\,\mu\mathrm{F}$. Derive a differential equation governing this circuit and solve for the impulse response $h(t)$ for this system analytically using $h(t) = b_n\delta(t) + P(D)y_n(t)u(t)$ as described in Section 2.3 of the text.

In a file called `impulse_response.py`, write a function in Python that takes in the values of R and C and returns the impulse response of the system. The function declaration should look like the following:

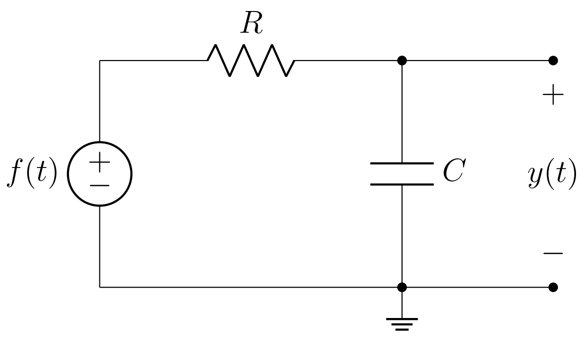

Figure 3.1: Basic RC circuit

```
def impulse_response(t, R, C):
    """Defines the impulse response h(t) of an RC circuit.

    Parameters
    ----------
    t : float
        time point to evaluate the response
    R : float
        resistor value in Ohms
    C : float
        capacitor value in Farads

    Returns
    -------
    float
        h(t) value at time t
    """
```

3.2.2 Square wave response

Next, we will determine the system's response to a square wave input. We will use a square wave that has a pulse width of $T = 1\,\text{ms}$ and an amplitude of $A = 5\,\text{V}$, which would give a system input

$$f(t) = A\left(u(t) - u(t-T)\right).$$

Now that we know the $h(t)$, we must find the system's output for this $x(t)$. Evaluate

$$y(t) = h(t) * f(t).$$

In a new Python file called `square_wave_response.py`, write a function that takes in the values of R and C and returns the system's response to a square wave input. The function declaration should look like the following:

```
def square_response(t, A, T, R, C):
    """Defines the system response to a square wave input of 5V and a period of
    T.

    Parameters
    ----------
    t : float
        time point to evaluate the response
    A : float
        amplitude of the square wave in V
    T : float
        pulse width of the square wave in seconds
    R : float
        resistor value in Ohms
    C : float
        capacitor value in Farads

    Returns
    -------
    float
        y(t) value at time t
    """
```

3.2.3 Ramp response

Finally, we will determine the system's response to a ramp input. We will use a ramp input that has a slope of $A = 25\,000\,\mathrm{kV\,s^{-1}}$ and duration $T = 200\,\mu\mathrm{s}$, which we can define as a system input

$$f(t) = At\left(u(t) - u(t-T)\right).$$

Once again, we must analyze the system's output for this input. Evaluate

$$y(t) = h(t) * f(t).$$

In a new Python file called `ramp_response.py`, write a function that takes in the values of A, T, R, and C and returns the system's response to a ramp input. The function declaration should look like the following:

```
def ramp_response(t, A, T, R, C):
    """Defines the system response to a ramp wave input of with a slope of A
   and a period of T.

    Parameters
    ----------
    t : float
        time point to evaluate the response
    A : float
        slope of the ramp in V/s
    T : float
        duration of the ramp in s
    R : float
        resistance in Ohms
    C : float
        capacitance in Farads

    Returns
    -------
    float
        y(t) value at time t
    """
```

3.2.4 Testing

In a file called `test_lab03.py`, there are unit tests for each function you wrote. Run the tests to ensure that your functions are working properly and your derivations are correct. You can run the tests by running the following command in the terminal:

```
$ python test_lab04.py

Testing impulse function
.
Testing ramp response function
.
Testing square response function
.
----------------------------------------------------------------------
Ran 3 tests in 0.004s

OK
```

3.3 Experimental system responses

3.3.1 Experimental setup

Build the circuit seen in Fig. 3.1. Make sure that you have the correct capacitor and resistor! Connect the input of the circuit to the function generator and the output to the oscilloscope. The output load on the function generator must be set to `High-Z`.

3.3.2 Measuring the impulse response

Measure the system's impulse response similar to what you did in Chapter 2. Use the same function parameters in Section 2.1.4 to generate the impulse response. Measure and save the output. Plot this using the same plot as the theoretical impulse response you derived earlier. Make sure that you properly scale each so that they match. If you do this work in the `impulse_function.py` file, make sure you put it in a `main` section so that it will not run when the tests are run.

3.3.3 Measuring the square wave response

Measure the system's response to a square wave. Use the `Pulse` waveform to generate the square wave. Set `Frequency` to 200 Hz, `amplitude` to 5 Vpp, `Offset` to +2.5 V, and `Pulse Width` to 1 ms. Measure and save at least one period of the output. Plot this using the same plot as the theoretical square wave response you derived earlier. Make sure that the timing for both waveforms aligns and that both responses start at 0.0 s. If you do this work in the `square_wave_response.py` file, put it in a `main` section so that it will not run when the tests are run.

3.3.4 Measuring the ramp response

Measure the system's response to a ramp wave. Here, we will need to use an arbitrary waveform. The `ramp_wave.arb` file is in your repository. Put this on a USB drive and insert it into the function generator. In the function generator, select `Waveform`, then `Arb`, then `Arbs`, then `Select Arb`, then scroll to `External` on the screen, and then finally select `ramp_wave.arb`. In the waveform parameters menu, make sure the sample rate is set to at $500\,\mathrm{kSa\,s^{-1}}$, the amplitude is set to 5 Vpp, and the offset is set to 0.0 V. Measure and save at least one period of the output. Plot this using the same plot as the theoretical ramp response you derived earlier. Make sure that the timing for both waveforms aligns and that both responses start at 0.0 s. If you do this work in the `ramp_response.py` file, make sure you put it in a `main` section so that it will not run when the tests are run.

Questions and tasks

- Plot the theoretical impulse response on the same plot as the measured impulse function.
- Plot the theoretical square wave response on the same plot as the measured square wave function.
- Plot the theoretical ramp response on the same plot as the measured ramp function.
- Do these align?
- What are the differences between the theoretical and measured responses?
- What are the sources of error in this experiment?
- How could you improve the accuracy of the measurements?
- What are some practical applications of convolution in signal processing?

3.4 Deliverables

3.4.1 Submission

Write up the results of this lab in the technical memo format (see Canvas for a template example). Submit the PDF to Canvas. Upload your Python code to GitHub.

3.4.2 Grading

The lab assignment will have the following grade breakdown:

- `test_impulse()`: 10pts.
- `test_square_response()`: 15pts.
- `test_ramp_response()`: 15pts.
- Impulse response plot: 10pts.
- Square wave response plot: 15pts.
- Ramp response plot: 15pts.
- Report: 20pts.

Chapter 4

Fourier Series and Total Harmonic Distorion

In this course, we have primarily focused on linear and time-invariant systems. These two properties imply that if a system is excited by a sinusoidal input, the output will also be a sinusoidal signal. The output will have the same frequency as the input but may have a different amplitude and phase. However, some systems behave in a non-linear manner. This means that the system's output will contain frequency components not present in the input signal. This is known as *harmonic distortion.*

The idea of harmonic distortion is important in many applications. For example, in audio systems, harmonic distortion can cause the sound to be distorted. In power systems, harmonic distortion can cause poor power quality. This exercise will explore the harmonic distortion concept by analyzing a signal's Fourier series representation.

Total Harmonic Distortion (THD) is a standard measure of signal distortion that quantifies the amount of harmonic distortion present in a signal. It is defined as the ratio of the sum of the powers of all harmonic components to the power of the fundamental frequency component. THD is expressed as a percentage and is used to evaluate the quality of a signal. A lower THD value indicates that the signal has less harmonic distortion and is closer to a pure sinusoidal signal. THD is commonly used in audio systems to measure the quality of sound signals and in power systems to assess the quality of electrical signals. For example, if you purchase a power amplifier, a manufacturer may provide the THD value as a measure of the quality of the amplifier.

THD is defined mathematically as

$$\text{THD} = \frac{\sqrt{V_2^2 + V_3^2 + V_4^2 + \dots}}{V_1} \tag{4.1}$$

though oftentimes it is a percentage, so it is common to multiply by 100 to get a percentage. The V_n terms are the amplitudes of the harmonics, and V_1 is the amplitude of the fundamental frequency. We will be able to identify these terms by analyzing the Fourier series of a signal.

4.1 Procedure

4.1.1 Parts list

You will need the following supplies to complete the lab.

Item	**Quantity**
$1\,\mathrm{k\Omega}$ resistor	2
LF353 opamp	1

4.1.2 Repository

Please clone the repository linked on Canvas.

4.2 Theoretical system responses

4.2.1 Background

One common source of harmonic distortion is called clipping. Clipping occurs when the input signal to a system exceeds the system's output range. When this happens, the system's output is limited to the maximum or minimum value of the output range. This causes the output signal to distort, as the system cannot accurately reproduce the input signal. Clipping is a common source of harmonic distortion in audio systems, as it can cause the sound to be distorted[1].

An example of a clipping signal is seen in Fig. 4.1. The input signal is a sinusoidal wave, but the output signal comes from an inverting op-amp circuit where the output exceeds the rail voltage, causing clipping. The output signal contains frequency components not present in the input signal, an example of harmonic distortion.

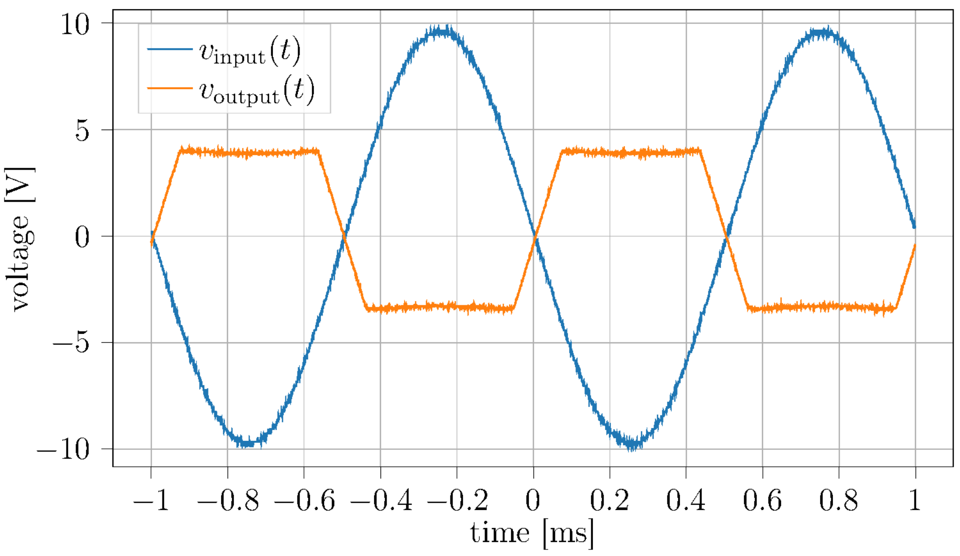

Figure 4.1: Clipping of a sinusoidal signal

[1]In some applications, this distortion is desirable, such as in electric guitar amplifiers.

We see that this output signal is not a pure sinusoidal signal. Instead, it contains frequency components at multiples of the input frequency. This is because the output signal can be represented as a sum of sinusoidal signals at different frequencies, also known as a Fourier series.

4.2.2 Fourier series representation

Your first task is representing a clipped sinusoidal signal as a trignomometric Fourier series

$$f(t) = a_0 + \sum_{n=1}^{\infty} a_n \cos(n\omega t) + b_n \sin(n\omega t).$$

You will want to be able to represent this in generic terms because we will modify the frequency and the clipping level throughout the lab. See Fig. 4.2 for an example of two periods of a clipped sine wave. This figure shows the signal $f(t)$ in blue and the clipping level in red. The clipping level is the maximum and minimum values of the output signal. In Fig. 4.2 the clipping level is 0.75. If unclipped, $f(t)$ would have a maximum amplitude 1. We also notice that this signal has odd symmetry, simplifying the derivation considerably[2] since we only need to concern ourselves with the sine terms and b_n coefficients.

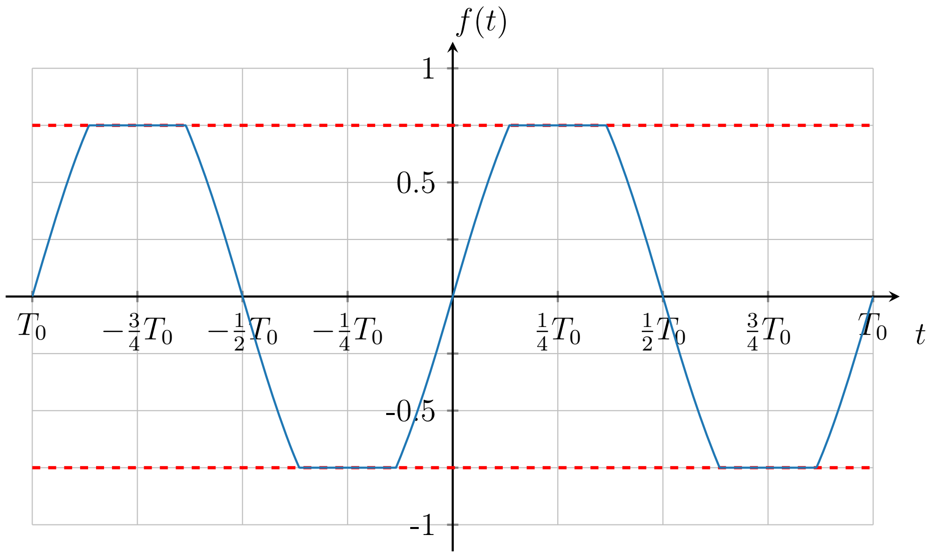

Figure 4.2: Clipped sine example

You must compute the b_n coefficients for the Fourier series representation of the clipped sine wave. In a file called `clipped_sine.py`, write a Python function `compute_bn()` to find the b_n coefficients based on your derivation. The function should take the clipping level V_{clip} and the frequency ω_0 as arguments. We will assume that the nominal amplitude of the unclipped sine wave is 1. The function should return the b_n coefficient for the n^{th} coefficient. The function declaration should look like the following:

```python
def compute_bn(n, level, omega_0):
    """Calculates the b_n coefficient for a clipped sine wave.

    Parameters
```

[2]See your ECE 2260 textbook for a description of the symmetry simplifications.

```
    ----------
    n : int
        Harmonic number
    level : float
        clipping level
    omega_0 : float
        fundamental frequency of the signal

    Returns
    -------
    float
        b_n coefficient of n-th harmonic
    """
```

Once you have a way to find the b_n coefficients, you can reconstruct the clipped signal using the Fourier series representation. Write a Python function that takes the b_n coefficients and the number of terms to include in the series. The function should return the reconstructed signal. The function declaration should look like the following:

```
def fourier_series(t_array, b_n, omega_0):
    """Reconstructs a signal from its Fourier series.

    Parameters)
    ----------
    t_array : ndarray
        time array
    b_n : ndarray
        b_n coefficients, indexing starts at 1 (ignore b_n[0])
    omega_0 : float
        fundamental frequency of the signal

    Returns
    -------
    ndarray
        reconstructed signal
    """
```

Please be careful with the indexing with the `b_n`. To simplify the indexing, you should ignore the b_0 term (`b_n[0]`) and start the indexing at 1. This will make the indexing match the harmonic number.

Make a plot of several clipped sine waves using your Python functions. Let the fundamental frequency be 1 kHz. Vary the clipping level from 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 0.99. Use 1000 harmonics. Plot all of the signals on the same plot. Include all the necessary labels and a legend.

4.2.3 Total Harmonic Distortion

As mentioned, THD is computed by comparing the sum of the powers of the harmonics to the power of the fundamental frequency as seen in Eq. 4.1. We can compute the THD of a signal by computing the Fourier series of the signal and then using the coefficients to compute the THD. Thus, Eq. 4.1 can be rewritten as

$$\mathrm{THD} = \frac{\sqrt{b_2^2 + b_2^2 + b_3^2 + \dots}}{b_1}. \tag{4.2}$$

Write a Python function that computes the THD of a signal. The function should have the b_n coefficients and the number of harmonics as arguments. The function should return the THD of the signal. The function declaration should look like the following:

```python
def compute_thd(b_n):
    """Calculates the total harmonic distortion of an odd function.

    Parameters
    ----------
    b_n : ndarray
        b_n coefficients for odd function

    Returns
    -------
    float
        Total harmonic distortion
    """
```

Compute the THD of clipped sine waves using the same clipping levels. Use 1000 harmonics. Plot the THD as a function of the clipping level. Include all the necessary labels.

4.2.4 Testing

There are a lot of places where you could go wrong here, so you are provided unit tests for each function to ensure they work. Run the tests to ensure that your functions are working correctly and your derivations are correct. You can run the tests by running the following command in the terminal:

```
$ python test_lab05.py

Testing b_n coefficient function
.
Testing THD computation function
.
Testing Fourier series function
.
----------------------------------------------------------------------
Ran 3 tests in 0.008s

OK
```

Questions and tasks

- Include a derivation for the b_n coefficients in your report.
- Include a plot of the clipped sine waves, show at least one period of the signal.
- What is the relationship between the clipping level and the THD?
- What is the relationship between the number of harmonics and the THD?

4.3 Experimental system responses

4.3.1 Experimental setup

Next, we will build the circuit seen in Fig. 4.3. Make sure that you have the correct resistor values and that the op-amp is connected correctly. Set the rail voltage to the op-amp to ±5 V.

Connect the input of the circuit to the function generator. Measure this input voltage as channel 2 in the oscilloscope. The BNC T-connector can connect the function generator directly to the oscilloscope. Using an oscilloscope, connect the circuit output to the oscilloscope. Measure this output voltage as channel 1 in the oscilloscope. The output load on the function generator must be set to `High-Z`. Generate a sine wave at 1 kHz with an amplitude of 6 V_{pp}. You will notice that there is little clipping at this lower voltage.

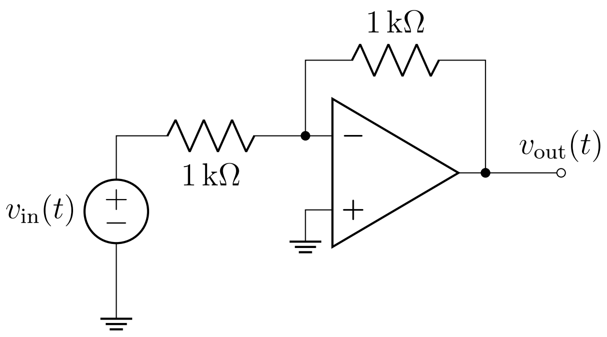

Figure 4.3: Clipping circuit

4.3.2 Data collection

Decrease the time resolution on the oscilloscope to 2 ms div^{-1}. Turn on the `FFT` function on the oscilloscope. Adjust the FFT to set the window to `Flat Top`. Set the span to 50 kHz and the center to 25 kHz. This visualization will allow you to see the harmonics of the signal. At this point in the course, you should not know exactly what you are seeing here, but the spikes at integer multiples of the the fundamental frequency (1 kHz) are the harmonics of the signal. Record this signal. Plug a USB drive into the oscilloscope and use the `CSV` option to save the data. This will record the voltage data in both channels and the frequency representation in the FFT at the bottom of the `.csv` file. Increase the sine wave amplitude by 2 V_{pp} and retake the same measurements. Keep incrementing the amplitude until the function generator maxes out at 20 V_{pp}.

4.3.3 Data analysis

At this point in the semester, you do not know how to work with FFT data, so for this stage, you are provided with code that will do it for you. Use the `data_thd.pyc` functions to compute the clipping ratio and the THD for each measurement you took. To use this code:

```python
import data_thd

thd, clipping = data_thd.thd_from_data("scope_XX.csv")
```

The module is documented, so if you want to learn more, look at the help section. The code itself is byte-compiled, so you will need to be using the appropriate Python version to use it. Find the clipping ratio and THD for each measurement you took. On the plot you made earlier with the theoretical THD values, plot the experimental THD values as a function of clipping ratios. Include all the necessary labels.

Questions and tasks

- How well do the theoretical THD values match the experimental THD values?
- What would cause a discrepancy between the theoretical and experimental THD values?

4.4 Deliverables

4.4.1 Submission

Write up the results of this lab in the technical memo format (see Canvas for a template example). Submit the PDF to Canvas. Upload your Python code to GitHub.

4.4.2 Grading

The lab assignment will have the following grade breakdown:

- `compute_bn()`: 20pts.
- `fourier_series()`: 15pts.
- `compute_thd()`: 15pts.
- Clipped sine wave plot: 15pts.
- THD plot: 15pts.
- Report: 20pts.

Chapter 5

LTI system response to periodic signal

In this exercise, we will investigate the behavior of an LRC circuit system by calculating the output using Fourier series analysis and implementing the circuit in hardware. We will use Fourier series analysis to calculate the output of the LRC circuit system in response to a periodic input signal. By decomposing the input signal into its constituent sinusoidal components using Fourier series techniques, we can determine the amplitude and phase of each component. Using this information, we will evaluate the output of the LRC circuit system for each frequency component, providing insights into its frequency response characteristics.

In addition to theoretical analysis, we will implement the LRC circuit system in hardware. We will observe the output response firsthand by constructing the circuit and connecting it to a signal generator. This hands-on approach allows us to compare the theoretical calculations with the observed behavior of the circuit, providing a comprehensive understanding of its performance.

This exercise will give you valuable insights into the system's response to periodic input signals. By combining theoretical analysis with practical implementation, you will better understand Fourier series analysis, circuit behavior, and signal processing.

5.1 Procedure

5.1.1 Parts list

You will need the following supplies to complete the lab.

Item	**Quantity**
1 kΩ resistor	1
0.033 µF capacitor	1
1 mH inductor	1

5.1.2 Code

Clone the git repository from the provided link on Canvas. This repository will be empty as you place a single `*.py` file in your submission.

5.1.3 Fourier series

Derive the exponential Fourier series for the square wave that has an amplitude of 4 V (8 V peak-to-peak and no DC component seen in Fig. 5.1.

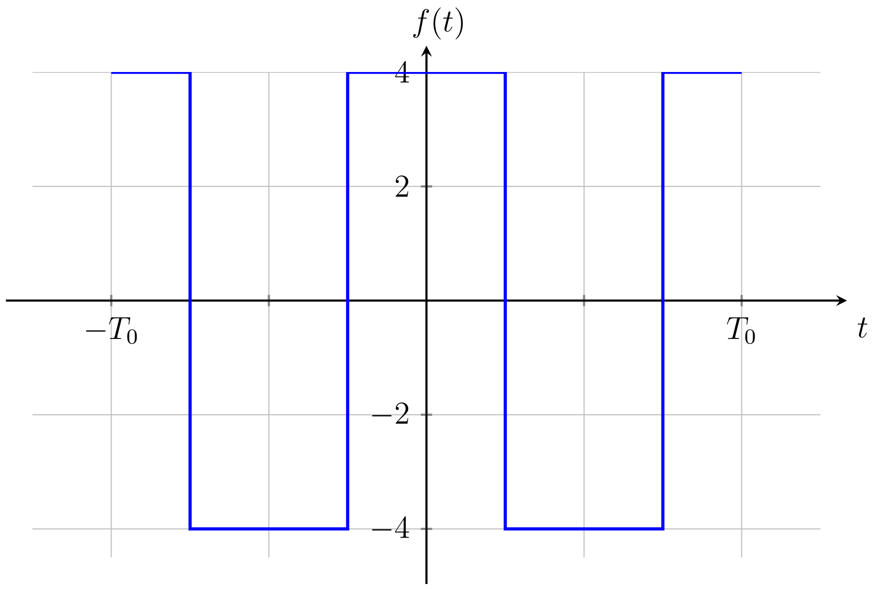

Figure 5.1: 4 V square wave.

Assume a frequency of 20 kHz. In Python script called `ece3210_lab05.py`, plot the square wave using $m = 35$ harmonics as you did in Chapter 4, except this time use the *exponential* Fourier series

$$f(t) \approx \sum_{n=-m}^{m} D_n e^{jn\omega_0 t}$$

where $\omega_0 = {}^{2\pi}/_{T_0} = 4 \times 10^4 \pi$ (for 20 kHz). In Python, you can make a number imaginary by adding an `1i` or `1j` to the end of it, for example, `2+3j`. All operators use complex arithmetic, including the function `exp()`.

Questions and tasks

- What is the complex exponential Fourier series representation of $f(t)$?

5.1.4 Circuit analysis

Find the ODE governing the circuit in Figure 5.2. From the ODE, derive the transfer function $H(s)$ from this ODE. In your `ece3210_lab06.py`, add a simulation of the system if the input $f(t)$ is the square wave in Fig. 5.1.

Recall from lecture that this is

$$y(t) = \sum_{-\infty}^{\infty} H(jn\omega_0) D_n e^{jn\omega_0 t}.$$

Use $R = 1\,\mathrm{k}\Omega$, $C = 0.033\,\mu\mathrm{F}$, and $L = 1\,\mathrm{mH}$ for the component values. Run your script and plot the reconstructed square wave and the system response. Please include the plot and your modified script in your Canvas submission.

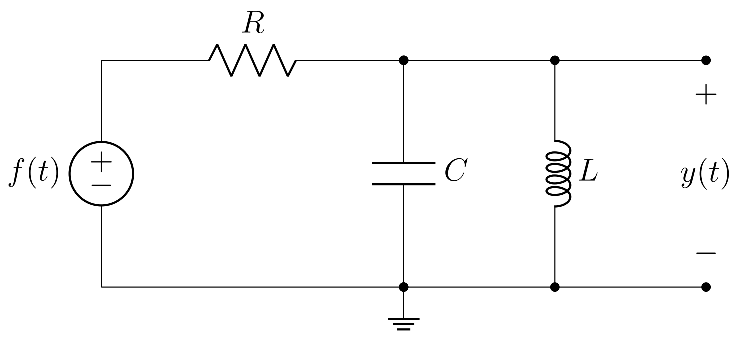

Figure 5.2: Series/parallel resonant circuit

Questions and tasks

- What ODE governs the circuit in Fig. 5.2?
- Plot the system response to the square wave input.

5.1.5 Hardware implementation

Connect the 1 kΩ resistor, the 0.033 µF capacitor, and the 1 mH inductor as shown in Figure 5.2. Connect the input to the function generator and attach a scope probe to the output. Configure the function generator to produce a 20 kHz square wave with a 4 V amplitude. Compare the scope output to your predicted response. It is unlikely they are the same due to part tolerances. The easiest (but not the most accurate) way to compensate is to adjust the frequency until the images are similar. Record your observations and include a conclusion in your Canvas submission summarizing what you have observed or discovered.

Questions and tasks

- Plot the circuit response to the square wave.
- How does this result compare with the theoretical response?

5.2 Deliverables

5.2.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). Include all requested figures. Submit the PDF to Canvas. Upload your code to GitHub.

5.2.2 Grading

The lab assignment will have the following grade breakdown:

- Derive Fourier series: 20pts.
- Python plot Fourier series: 20pts.
- Derive $H(s)$ from the circuit: 20pts.

- Simulation circuit response: 20pts.
- Hardware implementation circuit response: 20pts.
- Report: 20pts.

Chapter 6

Aliasing

During a telephone conversation, the digital telephone network transports streams of sampled voice signals between the phone exchanges to which the two connected handsets are attached. Voice signal transmission between the phone exchange and telephone handsets is typically analog. The required analog-to-digital conversion (sampling and quantization) and digital-to-analog conversion (reconstruction) are performed at the phone exchanges. The design is based on the fact that a significant fraction of human voice energy is typically below about 3.5 kHz. Assuming 4 kHz just to be safe, the Nyquist sampling rate is 8000 samples-per-second (i.e., $f_s = 2 \times 4\,\text{kHz}$), which is the standard sampling rate used in the telephone system. At each sampling instant, an 8-bit sample is taken, which results in a total bit rate of $64\,\text{kbit}\,\text{s}^{-1}$ (i.e., $8\,\text{bit} \times 8\,\text{kHz}$) for each one-way voice stream. (In this lab, we will not look at quantization effects, and we will consider that the number of bits-per-sample is large enough for the resulting errors to have negligible effects.)

In each telephone exchange, the incoming analog voice signal is first low-pass filtered to reduce the frequency components above 3.5 kHz so that significant aliasing does not occur. Following this, the signal is sampled, as discussed above, and transported to its destination telephone. In the first part of the lab, we will consider what would happen if this pre-filtering operation were not performed, and therefore, aliasing may occur.

6.1 Procedure

6.1.1 Aliasing of a sinusoid

We will consider the effects of aliasing on a sinusoid sampled and transmitted through the phone system. (You can assume that the person on one end of the phone conversation hums various sinusoids into the handset.) We are interested in what emerges at the other telephone. We can model the "transmitted" signal by looking at a simple sinusoid

$$x(t) = A \sin(2\pi f_0 t + \phi).$$

We can sample at periodic intervals of the sampling period, $T = {}^1/_{f_s}$, where f_s is the sampling rate, and ϕ can be any phase. The sampled signal is given by

$$\begin{aligned} x[k] &= A\sin(2\pi f_0 kT + \phi) \\ &= A\sin\left(2\pi k\left(\frac{f_0}{f_s}\right) + \phi\right). \end{aligned}$$

Write a Python script (`ece3210_lab06.py`) to do the following (place it in the provided GitHub repository). Writing a function to generate the sinusoid described above might be helpful to save yourself from some code redundancy.

In your script, set $f_s = 8\,\text{kHz}$ and $f_0 = 300\,\text{Hz}$, and generate a `numpy` array for $x(kT)$ over the interval $t \in [0\,\text{ms}, 10\,\text{ms}]$. Plot $x(kT)$ using the `matplotlib` stem command (please find the documentation on the `matplotlib` website for usage–it is essentially the same as `plt.plot()` which we have used before). You will need to choose A and ϕ.

When this sampled signal is transported to its destination, it is converted into analog form and transmitted to the destination telephone. For simplicity, we will assume that this digital-to-analog conversion is done by connecting the voice sample values in straight lines (essentially linear interpolation). This can be seen in Python by using the `plt.plot()` command rather than `plt.stem()`. In practice, a smoother reconstruction is performed by analog filtering at the output.

Over the interval from $t \in [0\,\text{ms}, 10\,\text{ms}]$, plot $x(kT)$ using the plot command when $f_0 = 300\,\text{Hz}$, $500\,\text{Hz}$, $700\,\text{Hz}$, and $900\,\text{Hz}$. Rather than plotting your graphs separately, plot all four sinusoids on separate subplots in a 2×2 grid (again, the `matplotlib` documentation for subplot is thorough). Label each sub-plot with its frequency value f_0 and include the plot in your Canvas submission.

Play each sinusoid using the command `sounddevice` module. You will need to install this, which is easy through Anaconda. Install this by

```
$ pip install sounddevice
```

Alternatively, you could install it through

```
$ conda install -c conda-forge python-sounddevice
```

An example usage of soundevice is given below.

```
# Use the sounddevice module
# http://python-sounddevice.readthedocs.io/en/0.3.10/

import numpy as np
import sounddevice as sd
import time

# Samples per second
sps = 32000

# Frequency / pitch
freq_hz = 700.0

# Duration
```

```
duration_s = 3.0

# Attenuation so the sound is reasonable
atten = 0.3

# NumpPy magic to calculate the waveform
each_sample_number = np.arange(duration_s * sps)
waveform = np.sin(2 * np.pi * each_sample_number * freq_hz / sps)
waveform_quiet = waveform * atten

# Play the waveform out the speakers
sd.play(waveform_quiet, sps)
time.sleep(duration_s)
sd.stop()
```

Based on the information given for this problem, `fs=8000` is your sampling frequency. To make these sinusoids easier to hear, make $T = 2\,\mathrm{s}$ rather than $10\,\mathrm{ms}$. Describe what you hear. Choose a different value for ϕ and repeat this experiment. Can you hear the difference? Explain why or why not.

Concatenate four one-second tone segments above four frequencies into a single vector. The `numpy` command `np.hstack` might be helpful here. Play this signal. Describe what you hear, and explain why you would expect this.

Now repeat the last task, but this time use sinusoids of frequencies $f_0 = 7700\,\mathrm{Hz}$, $7500\,\mathrm{Hz}$, $7300\,\mathrm{Hz}$, and $7100\,\mathrm{Hz}$. Describe what you hear and explain why this is precisely what you would expect.

Questions and tasks

- Plot $x(kT)$ from $t \in [0\,\mathrm{ms}, 10\,\mathrm{ms}]$ with the specified f_s and f_0.
- Make a 2×2 grid of four plots at $f_0 = 300\,\mathrm{Hz}$, $500\,\mathrm{Hz}$, $700\,\mathrm{Hz}$, and $900\,\mathrm{Hz}$.
- Describe what you hear when you play the specified waveforms with different phase values and f_0 values.
- Describe what you hear when you play $f_0 = 7700\,\mathrm{Hz}$, $7500\,\mathrm{Hz}$, $7300\,\mathrm{Hz}$, and $7100\,\mathrm{Hz}$.
- Why the performance of a generic telephone system would degrade significantly if anti-aliasing pre-filtering were not used. Explain how this filtering prevents these negative effects. What would happen in the above experiments if this filtering were in place?

6.1.2 Aliasing of a Chirp Signal

In this section, we will illustrate the aliasing of a chirp signal. A chirp is a signal whose frequency is a linear function of time. We will use the following chirp signal

$$c(t) = A\cos\bigl(\pi\mu t^2 + 2\pi f_1 + \phi\bigr). \tag{6.1}$$

Recall that $\cos(2\pi f_0 t)$ has a frequency of $2\pi f_0$ radians per second, or f_0 cycles-per-second (Hz). Also, note that $\theta = 2\pi f_0 t$ is the phase of $\cos(2\pi f_0 t)$, and that $2\pi f_0 = \frac{d\theta}{dt}$. By taking the derivative of the signal phase in Eq. (6.1), it can be seen that the instantaneous frequency of the chirp signal is given by $f(t) = \mu t + f_1$. The frequency increases linearly with time, starting at f_1 Hz.

Generate samples of this signal in your Python code. Choose $f_1 = 100\,\text{Hz}$ and set $\mu = 2000$. Start using a sampling frequency of $f_s = 32\,\text{kHz}$. Sample $c(t)$ for 8 seconds. Plot the first 2000 samples to see what the sampled signal looks like. Then play the sampled signal using the `sounddevice` module. Describe and explain what you see and hear in both cases.

Repeat the above experiment but use a sampling frequency of 16 kHz. Explain what you hear now. Using the theory you know, explain in detail what you have heard. Try the same thing for 8 kHz, which indicates what would happen if this signal were sent through the digital telephone network without anti-aliasing pre-filtering. What would you hear over a telephone connection that includes anti-aliasing filtering? Experiment with other f_1, f_s, and μ values. In all cases, explain what you hear using sampling theory.

Questions and tasks

- Plot the first 2000 samples of the sampled chirp signal.
- What do you hear with a $f_s = 32\,\text{kHz}$, 16 kHz, 8 kHz? Why does it sound this way?

6.2 Deliverables

6.2.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). Include all requested figures. Submit the PDF to Canvas. Upload your code to GitHub.

6.2.2 Grading

The lab assignment will have the following grad breakdown.

- Signal stem plot: 10pts.
- Plot reconstructed signals (Aliasing Part (b)): 10pts.
- Explanation of phase impacts on sound: 10pts.
- Concatenate four tone segments: 10pts.
- Concatenate four tone segments with different sampling rates: 10pts.
- Explain point of anti-aliasing filter: 10pts.
- Plot chirp signal (and explain sound): 10pts.
- Explain resampled chirp signal: 10pts.
- Write-up: 20pts.

Chapter 7

Shazam

When you hear a song you like when you are out and about it can be frustrating when you cannot identify the title or artist. One way to find out what's playing on the radio is to use an application called Shazam that will record a snippet of the recording and can identify it against an extensive database. Interestingly, even in the age of increasing AI sophistication, Shazam uses a relatively simple algorithm based on the Fast Fourier Transform (FFT) to identify the song. In this chapter, we will explore the FFT and how it can be used to identify songs.

The basic procedure goes as follows:

1. Construct a database of features for each full-length song.
2. When a clip (hopefully part of one of the songs in the database) is recorded, compute the features for the clip.
3. Search the database for the song that has the most similar features.

Like Shazam, the features for each song (and clip) will be pairs of proximate peaks in the spectrogram of the clip. We start by finding the peaks in the (log) spectrogram; plotting the peaks gives a "constellation map." Each peak has a time frequency location (t, f) and a magnitude A. We then form pairs of peaks that are within a pre-specificied time and frequency distance of each other and record the details of these pairs in the database. For example, if we record a pair (f_1, t_1) and (f_2, t_2), we might record $(f_1, f_2, t_1, t_2 - t_1, \text{song\_id})$. We will discard the amplitudes because the amplitude of a peak may not be consistent across recordings. Full details can be found in [1].

Each song is "fingerprinted" by a table of extracted features, e.g.,

f_1	f_2	t_1	$t_2 - t_1$	song_id
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
f_i	f_j	t_i	$t_j - t_i$	song_id
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
f_m	f_n	t_m	$t_n - t_m$	song_id

If we are given a clip, we can extract the features and construct a similar table, e.g.,

f_1	f_2	t_1	$t_2 - t_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
f_i	f_j	t_i	$t_j - t_i$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
f_m	f_n	t_m	$t_n - t_m$

We do not know the clip's start time within its corresponding song. The times t_j appear in the clip table relative to the clip's start. The clip itself starts at some offset t_0 from the beginning of the music. Finding a match to the clip in the database is a matter of matching the constellation map of the clip to the constellation maps of the songs by effectively sliding the former over the latter until a position is found in which a significant number of points match.

Using pairs of peaks as features gives us three quantities that we expect to be independent of the unknown offset time: $(f_1, f_2, t_2 - t_1)$. The song with the most triples $(f_1, f_2, t_2 - t_1)$ in common with the clip table is likely to be the source of the clip.

7.1 Getting started

First, clone the repository in GitHub. This will contain a file `audio_utils.py` that will provide you with functions you will use throughout the project. Next, download the dataset from Canvas, which is stored in a a `.h5` file, a common file format for storing large datasets. You can use the `h5py` library to read the file. The dataset contains roughly 50 songs, each processed and cropped into 90 second clips. The dataset is stored in a dictionary with the `.wav` file names as keys and the clips as values stored as arrays of floats. You can read this file using the following code:

```python
import h5py

h5_file = h5py.File("songs.h5", "r")

for song in list(h5_file.keys()):
    signal = h5_file[song][:]
```

Also download the `songs_test.zip` file from Canvas. This contains a set of test clips that you will use to test your Shazam implementation. There are 15 clips in total, selected randomly from the dataset. Each clip is 10 seconds long.

All of your work will be in a file called `shazam.py`.

7.2 Building the database

7.2.1 Preprocessing

The first step is to preprocess the songs. The audio is recorded in stereo (two channels); we can average the two channels to get a single channel. We will also downsample the audio to 8 kHz to reduce the amount of data we need to process. We will also subtract the average of the signal to remove any DC offset.

These steps are already implemented in the `audio_utils.record_audio()` and `audio_utils.wav_processin` functions. Please look at both functions to get a sense of what is happening.

7.2.2 Spectrogram

The first step in the Shazam process is to compute the spectrogram of the clip. The spectrogram is a 2D representation of the audio signal, where the x-axis is time, the y-axis is frequency, and the color represents the amplitude and frequency of the signal at that time. The spectrogram is computed using the Short-Time Fourier Transform (STFT), which is a series of FFTs computed on overlapping windows of the signal. This allows us to see how the signal's frequency content changes over time.

Essentially, starting at time t, we take a window of the signal from t to $t+T$, where T is the window size. We then compute the FFT of this window, which gives us the signal's frequency content at that time. We then slide the window over by a certain amount and repeat the process. This provides us with a series of FFTs that we can stack on top of each other to form the spectrogram.

Consider a signal that is a sine wave where the frequency changes over time (from 300 Hz to 600 Hz to 900 Hz). Such a signal might look like Fig. 7.1.

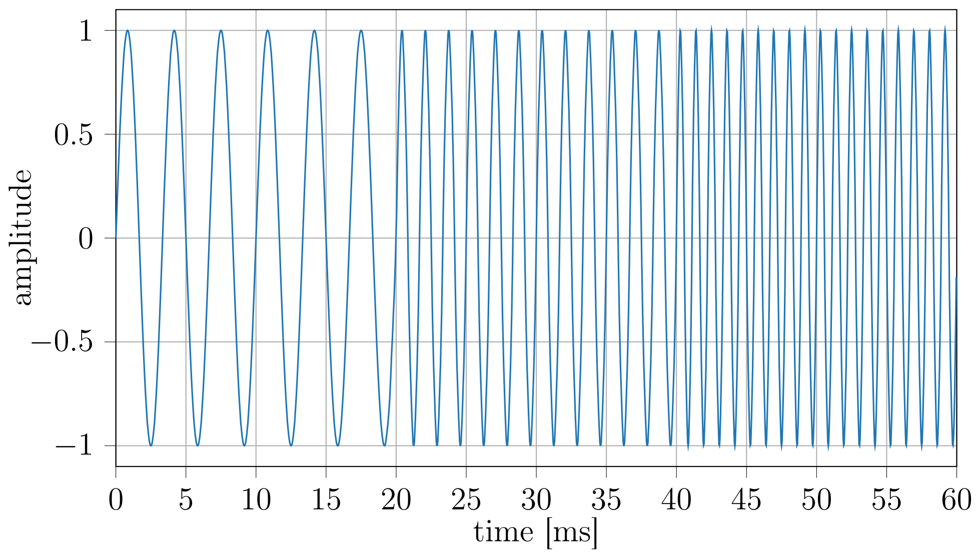

Figure 7.1: A signal that is a sine wave where the frequency changes over time.

We can compute the spectrogram and display its magnitude as a 2D image. The spectrogram of the signal in Fig. 7.1 is shown in Fig. 7.2. In this image, we notice that the signal's frequency content changes over time, which is what we would expect.

We can also use a "real" signal, such as a clip from a song. The spectrogram of a clip from the song "Cruel Summer" sampled at 44.1 kHz is shown in Fig. 7.3. Similar to what we saw earlier, we can see how the signal's frequency content changes over time.

We notice that this spectrogram has much more detail than the previous one. This is because the signal is more complex and has more frequency content. We can see that the signal has a lot of low-frequency content, which is likely the bass and drums, and some high-frequency content, which is likely the vocals and cymbals. This is what we would expect from a song. However, it is still hard to visualize because the peaks in the FFTs dominate the rest of the output. For this reason, we will use the logarithm of the magnitude of the FFTs, which is a more common way to visualize the spectrogram. The logarithm of the spectrogram of the same clip is shown in Fig. 7.4.

We will need to write a function that computes the spectrogram of a signal. The function should

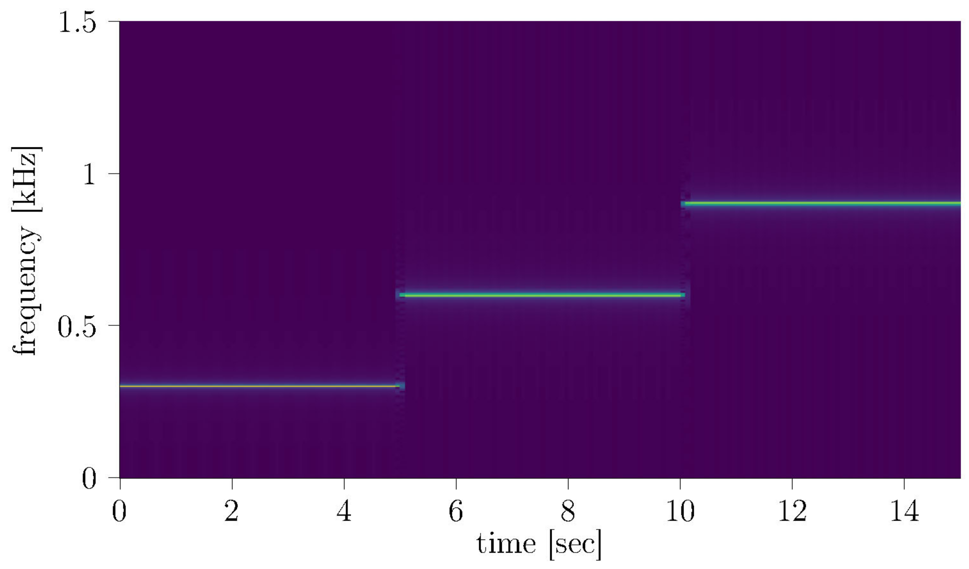

Figure 7.2: The spectrogram of the signal in Fig. 7.1.

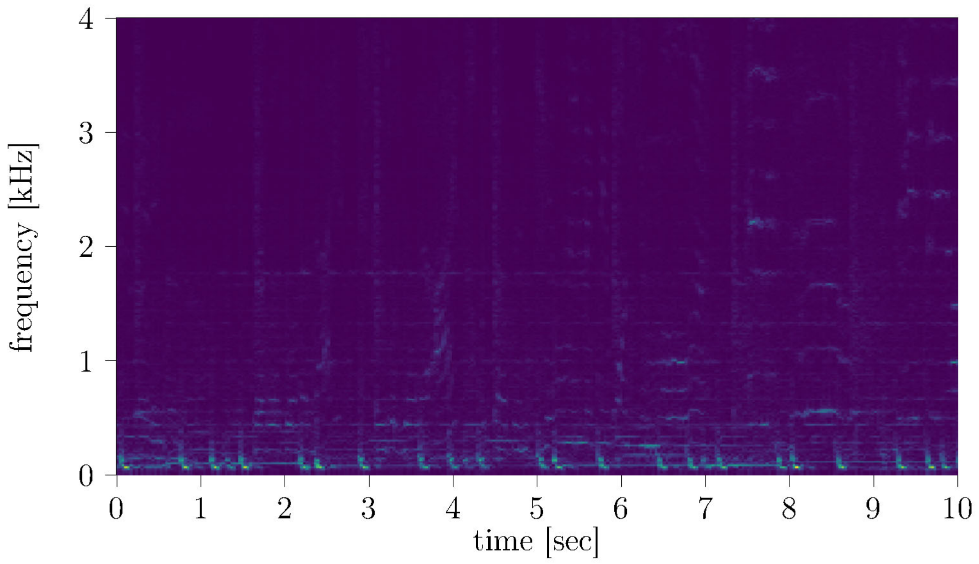

Figure 7.3: The spectrogram of a clip from the song “Cruel Summer”.

take in the signal `signal_in`, the sample rate of the signal (`fs`), the window size (`nperseg`), the overlap between each segment (`noverlap`), and the number of points in the FFT (`nfft`) in case you want to zero-pad each window. The function will return the sample frequencies, the time samples, and the magnitude of the FFTs. Your function should look like:

```python
def compute_spectrogram(signal_in, fs=1.0, nperseg=512,
                        noverlap=None, nfft=None):
    """Compute the spectrogram of a signal with a rectangular window.

    Parameters
    ----------
    signal_in : ndarray
        1D array-like, the input signal.
    fs : float, optional
```

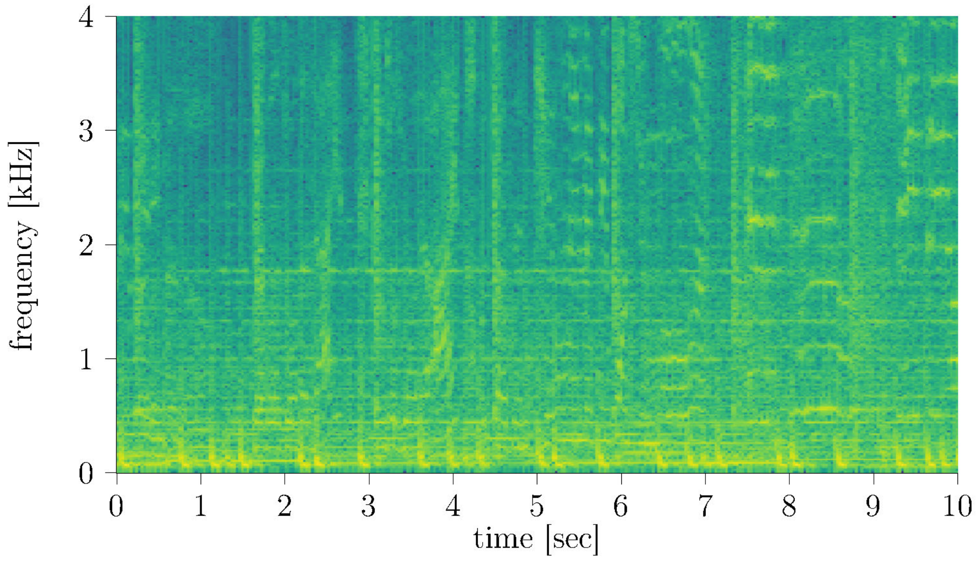

Figure 7.4: The logarithm of the spectrogram of a clip from the song "Cruel Summer".

```
        sampling rate in Hz, by default 1.0
    nperseg : int, optional
        length of each segment, by default 512
    noverlap : int, optional
        number of points shared between segments, by default None, which set to
    nperseg // 8
    nfft : int, optional
        length of the FFT used, by default None, which sets to nperseg

    Returns
    -------
    ndarray
        1D array of sample frequencies
    ndarray
        1D array of segment times
    ndarray
        2D array, the spectrogram of the signal
    """
```

We are assuming that the input signal is a 1D array of real-valued floats. Because these are real-valued, we can use the `rfft` function in `numpy` to compute the FFTs. This function will return the positive half of the FFT, which is all we need for the spectrogram (due to conjugate symmetry of the Fourier transform). For example, if we set `nfft=512`, the function will return an array of size 257, which corresponds to the positive half of the FFT. The sample frequencies will be the first 257 frequencies in the FFT, and the time samples will be the center of each window. The magnitude of the FFTs will be the magnitude of the positive half of the FFT.

In the example in Fig. 7.3, we used parameters `fs=8000`, `nperseg=512`, `noverlap=256`, `nfft=512`. This means that each window is 512 samples long, and each window is shifted by 256 samples before we computed the next window. Because the `nperseg` and `nfft` are the same, we did not zero-pad the windows. Notice in this figure that the frequency content goes up to 4 kHz, which is half of the sample rate. This is because the Nyquist frequency is half the sample rate, and the FFT will only return the positive half of the FFT.

The behavior of your function should be very similar to the `scipy.signal.spectrogram()` function, though the `scipy` version is a bit more complicated because it allows for different window types and scaling options. I would strongly suggest comparing your frequency and time output arrays to the `scipy` version to verify they are correct. You will also have a unit test for your `compute_spectrogram` function.

7.2.3 Spectrogram local peaks

We will need to find the local peaks in the spectrogram. We will say that a location is a local peak if it is greater than its neighbors in a surrounding $g_s \times g_s$ grid. There are a lot of ways to do this, and it is pretty simple to do, but we will want an implementation that is rather fast, so you are provided one in the `audio_utils` module. The function is called `audio_utils.local_peaks()` and takes in the spectrogram, the size of the square grid, and a threshold. The function will return a boolean 2D array where the peaks are `True` and the non-peaks are `False`.

In the example in Fig. 7.4, we could use a grid size of 9×9. This means that a location is a peak if it is greater than its neighbors in a 9×9 grid. The peaks are shown in Fig. 7.5.

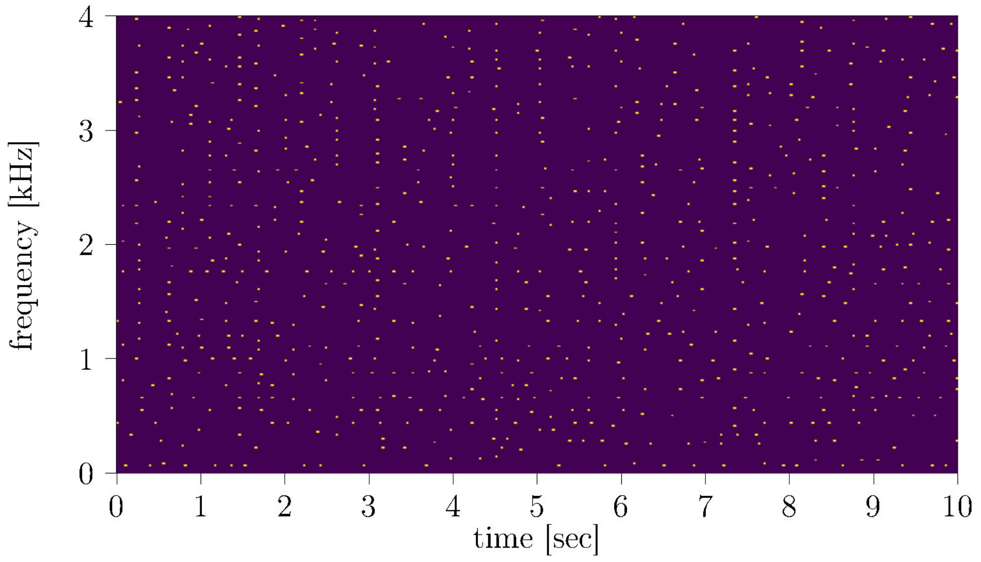

Figure 7.5: The peaks in the spectrogram of a clip from the song "Cruel Summer".

7.2.4 Thresholding

We want to use only the larger peaks. To select the larger peaks and also to control the average rate of peak section (peaks per second), we have to do some form of selection operation on the peaks. The simplest thing to do is apply a fixed threshold to the detected peaks and keep only those above the threshold. The threshold could be selected to yield (approximately) the desired number of peaks.

Assume that we want approximately 30 peaks per second. For example, if we have a 10-second clip, we will want to keep the 300 largest peaks. In the instance of the clip from "Cruel Summer," we would keep the 300 largest peaks. The peaks are shown in Fig. 7.6.

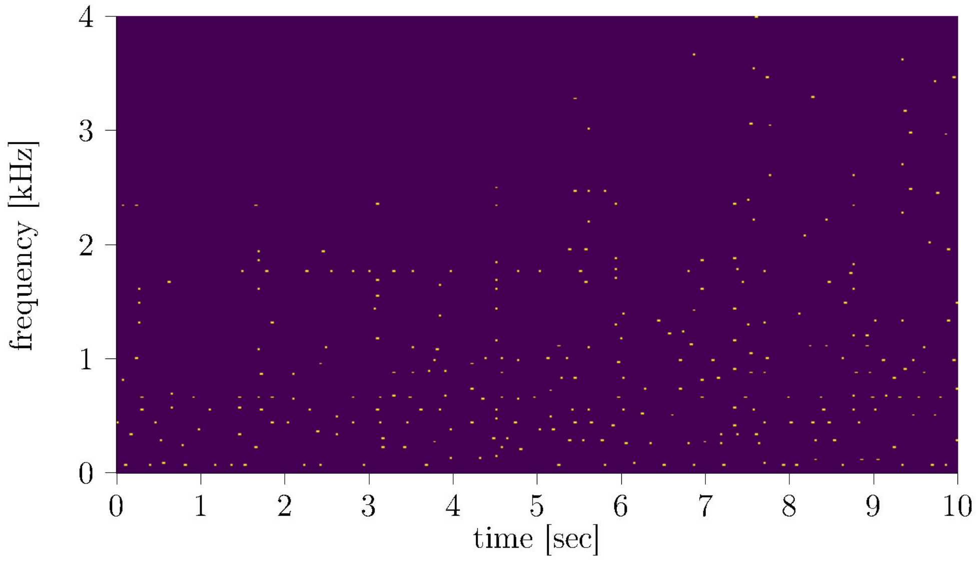

Figure 7.6: The peaks in the spectrogram of a clip from the song "Cruel Summer" after thresholding at 30 peaks per second.

7.2.5 Building the database

We want to select pairs of peaks and record the frequency of each peak, the time of the first peak and the time difference between the two peaks. A peak-pair must satisfy certain constraints: the second peak must fall within a given distance from the frequency of the first peak and the second peak must occur within a certain time interval after the first peak. We will also limit the number of pairs allowed to form from a given peak, say to 3 (this is called the "fan-out"). You can choose which three pairs you want to keep. An illustration of the fan-out is shown in Fig. 7.7.

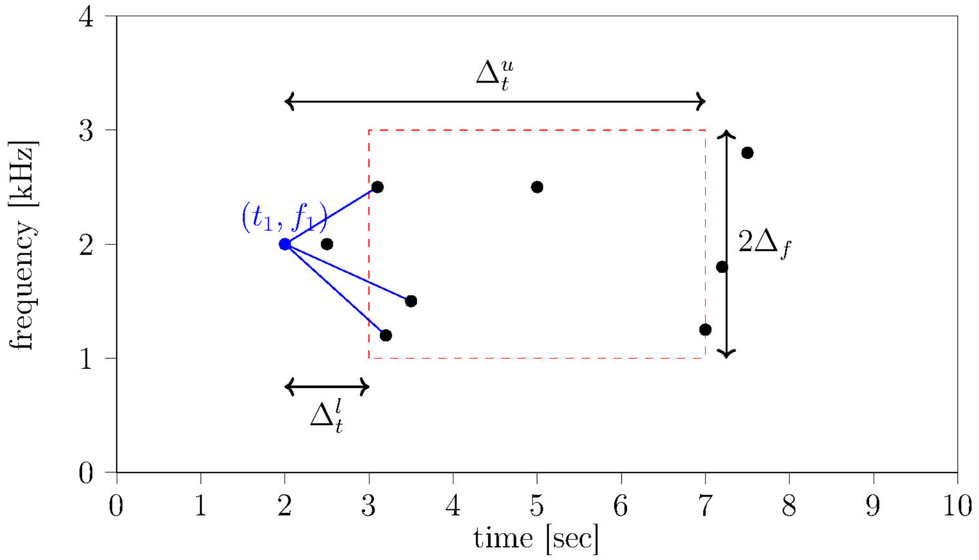

Figure 7.7: An example of the fan-out for a peak.

We will define the search region by restricting the time from Δ_t^l to Δ_t^u such that $t_1 + \Delta_t^l < t_2 < t_1 + \Delta_t^u$ and that in frequency $f_1 - \Delta_f < f_2 < f_1 + \Delta_f$. To keep the math simple, I suggest just letting the values of Δ_t^l, Δ_t^u, and Δ_f be integer values of the columns and rows you are searching over. If there are more than three peaks in the search region, you must select three. You can choose how you want to pick these three peaks. Regarding practical use, setting `delta_tl=3`, `delta_tu=6`, and `delta_f=9` seem to give decent results, but you are free to experiment.

Now, combine everything into a function and make a table that combines all these steps. It will take as input the song signal, and it will return an $n_{\text{pairs}} \times 4$ matrix, which contains in each row the 4-tuple corresponding to a peak pair:

$$\left|\begin{array}{c|c|c|c} f_1 & f_2 & t_1 & t_2 - t_1 \\ \vdots & \vdots & \vdots & \vdots \\ f_i & f_j & t_i & t_j - t_i \\ \vdots & \vdots & \vdots & \vdots \\ f_m & f_n & t_m & t_n - t_m \end{array}\right|$$

7.2.6 Hash table

Once we have the 4-tuples for a given song/clip, we will need to be able to store this information in a database. There are many ways of doing this, but using a hash table is the easiest. We will use a dictionary in Python to store the information. We will hash the 4-tuple into some integer that will serve as the dictionary key, song ID, and location t_1 as the value. We will use a simple hash function

$$h(f_1, f_2, t_2 - t_1) = (t_2 - t_1) \cdot 2^{16} + f_1 \cdot 2^8 + f_2. \tag{7.1}$$

To make this work, we will need f_1 and f_2 to take values from 0 to 255. However, the spectrogram returns a frequency axis that 257 elements long. We can remove the last row of the spectrogram (the highest frequency). We will use the row index of the peaks (which will now be restricted from 0 to 255) as f_1 and f_2. The time difference $t_2 - t_1$ will also be represented in the difference in column indices of the peak pair.

We will need to store this data in a Python data container. I would strongly suggest using a dictionary with this hash index as the key for each entry. However, you will find that each hash index will not necessarily be unique. E.g., you might have multiple different peak pairs that return the same hash index. You will need to adapt your container to handle this. You will need to process all of the songs in the `.h5` file and store the data in this container. It will be very inefficient to create this database every time you call your function, so you will want to save this database to a file. I would suggest using a JSON file. Please see the `json` library for how to do this.

Your program will need to be able to handle different functions, whether it is building a database or identifying a song. You will need to build a command line interface that will allow you to select the function you want to run. You can use the `argparse` library to do this. Please see the documentation for this library to understand how to use it.

To build the library, usage should look like the following:

```
$ python shazam.py --build-db

database successfully created and saved at: hash_database.json
```

7.3 Song identification

7.3.1 Recording audio

Now that we have built the database, we can use it to identify songs. The first thing your program should do is load the database. Your program should have two options: reading data straight from a `.wav` file or recording audio from the microphone. Both approaches have functions in the `audio_utils` module to do this. In both instances, make sure you use an appropriate sample rate. To be consistent with the data in the database, you should use a sample rate of 8 kHz.

7.3.2 Fingerprinting the audio clip

Once you have the data from the microphone or the `.wav` file, you will need to compute the spectrogram of the clip. You will then need to find the peaks in the spectrogram and threshold them. You will then need to build the 4-tuples for the clip just like you did with the database audio. You will need to store this information in a table that is similar to the database table.

Compute the hash index for each 4-tuple and look up the hash index in the database. You will need to keep track of the number of times each song appears in the database as well as the t_1 value (the time location for the peak) for both the database song, t_1^s, and for the audio clip you are trying to identify, t_1^c.

The song that appears the most is the song that is most likely to be the song that was recorded. However, the hash index might sometimes appear in the recorded clip and be used in a different song. To determine which is a "true" match, compute $t_o = t_1^s - t_1^c$ for each song that appears in the database. The offsets t_0 should roughly be the same in the song that matches the clip. If the offsets differ, the song is likely not a match. You will need to determine which offsets are the same (within some error) and how many similar offsets are for each song for which there is a matching peak pair. The song with the most similar offsets is the song that is most likely to be the song that was recorded.

7.3.3 Identifying the song

Your program should have two options to identify the song. The first is the easiest and is to identify the song from a `.wav` file. To use this functionality, add the following command line argument using the `argparse` library:

```
$ python shazam.py --wav-input new_slang_crop.wav

your song is: new_slang.wav
```

In the `songs_test.zip` file there are 15 clips from the dataset. Using this command line option, you can use these clips to test your program.

The second option is to record audio from the microphone. To use this functionality, add the following command line argument using the `argparse` library:

```
$ python shazam.py --mic-input 10
* recording
* done recording

your song is: we_only_come_out_at_night.wav
```

This will record audio for 10 seconds and then identify the song. You can specify a default value for the duration when you set up `argparse`. The microphone functionality will come from the `pyaudio` library. The lab computers will have this installed, but if you want to install it on your personal machine you can install it using `pip`.

One benefit of using `argparse` is that it automatically builds a help menu. You can access this by running the program with the `--help` argument. For example, the help menu might look like:

```
$ python shazam.py --help
usage: shazam.py [--debug] [--build-db] [--wav-input WAVFILE] [--mic-input
    DURATION]

This program builds a Shazam-like database, and analyzes songs based on
either external wav files or microphone input.

options:
  -h, --help            show this help message and exit
  --debug               Enable debug mode
  --build-db            Build Shazam database
  --wav-input WAV_INPUT
                        Analyze a song based on an external wav file
  --mic-input MIC_INPUT
                        Duration of the song to analyze
```

7.4 Testing

You will be provided two unit tests to check the correctness of your code. The first will check the `compute_spectrogram()` function. The second test will test to see the output using the `songs_test.zip` files.

```
$ python test_lab07.py

Testing compute_spectrogram
Test passed
.
Testing song matching




database successfully created and saved at: hash_database.json
Test passed for: new_slang.wav
Test passed for: party_in_the_usa.wav
Test passed for: my_girls.wav
Test passed for: blinding_lights.wav
Test passed for: blister_in_the_sun.wav
Test passed for: green_light.wav
Test passed for: angeles.wav
Test passed for: cant_hold_us.wav
Test passed for: gimme_gimme_gimme_a_man_after_midnight.wav
Test passed for: last_nite.wav
Test passed for: smells_like_teen_spirit.wav
Test passed for: islands.wav
Test passed for: cruel_summer.wav
Test passed for: dont_look_back_in_anger.wav
Test passed for: zombie.wav
.
----------------------------------------------------------------------
Ran 2 tests in 9.715s

OK
```

Other than that, you are free to design the software as you see fit. You will also be tested on whether or not your code can identify songs using a microphone[1].

7.5 Deliverables

7.5.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise.

In the appendices, please include the requested plots. Your code should be submitted to GitHub.

7.5.2 Grading

The lab assignment will have the following grad breakdown.

- `compute_spectrogram()`: 25pts.
- Write-up: 20pts.
- `.wav` file accuracy (5pts. each): 45pts.
- Microphone accuracy (10pts. each): 110pts.

[1]This is considerably more difficult than identifying a song from a `.wav` file as the phase and timing uncertainty (and spectral reshaping due to the transfer functions of the speakers and microphone) is what makes you go to the magnitude of the spectrogram and look for relative maxima locations rather than the values themselves.

Chapter 8

ECG notch filtering

In medical instrumentation applications, it is often necessary to filter out unwanted noise from signals such as electrocardiograms (ECGs). One common type of noise is the power line interference, which typically occurs at a frequency of 60 Hz in the United States. This interference can obscure the desired signal and make it difficult to analyze the ECG data accurately.

In this lab, you will design a notch filter to remove the 60 Hz interference from an ECG signal. You will use the transfer function of the notch filter to analyze its frequency response and implement it in Python. By the end of this lab, you will have a better understanding of how to design and implement filters for real-world applications.

8.1 Procedure

8.1.1 Deriving the transfer function

We will start by using a common notch filter design, which is a second-order active filter. This topography is known to provide a narrow notch (high Q factor), which is ideal for removing the 60 Hz interference while preserving the majority of the rest of the signal.

A generic version of this filter is shown in Fig. 8.2. In this filter we see that there are three resistors. Two have the value R and one has the value $R/2$, which is to say it has half the resistance of the other two resistors. Similarly, there are three capacitors. Two are the same with a value of C and one has a value of $2C$. Your job is to derive the transfer function of this circuit.

The book gives the transfer function of a notch filter as

$$H(s) = \frac{s^2 + \omega_0^2}{s^2 + (2\omega_0 \cos(\theta))s + \omega_0^2} \tag{8.1}$$

The transfer function can also be expressed in terms of the quality factor Q as

$$H(s) = \frac{s^2 + \omega_0^2}{s^2 + \frac{\omega_0}{Q}s + \omega_0^2} \tag{8.2}$$

where $\omega_0 = 2\pi f_0$ is the center frequency of the notch filter, and Q is the quality factor of the filter. The quality factor is a measure of how selective the filter is at removing the unwanted frequency. A higher Q value means a narrower notch, while a lower Q value means a wider notch.

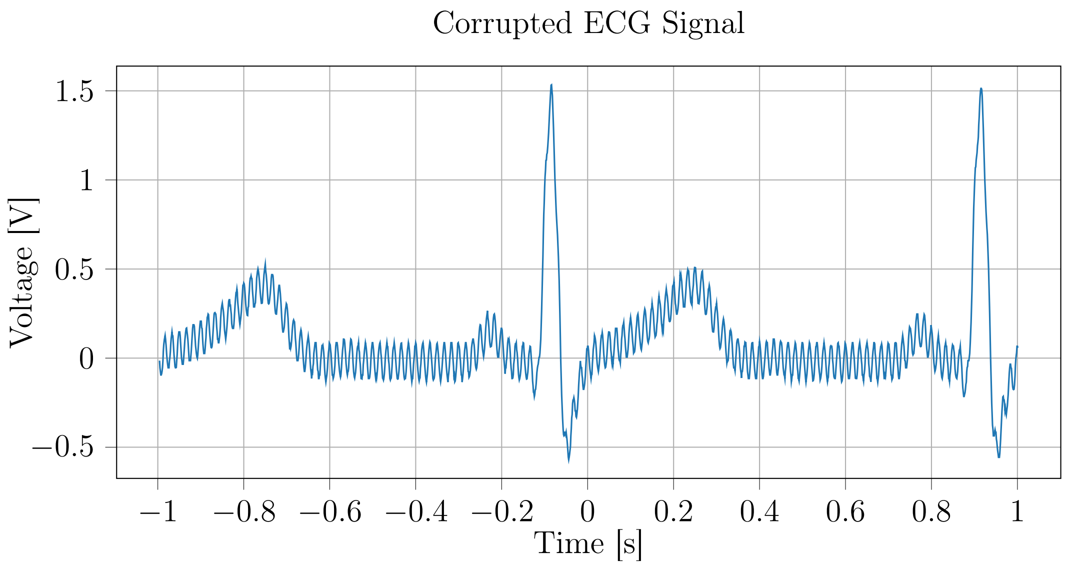

Figure 8.1: ECG signal with 60 Hz interference.

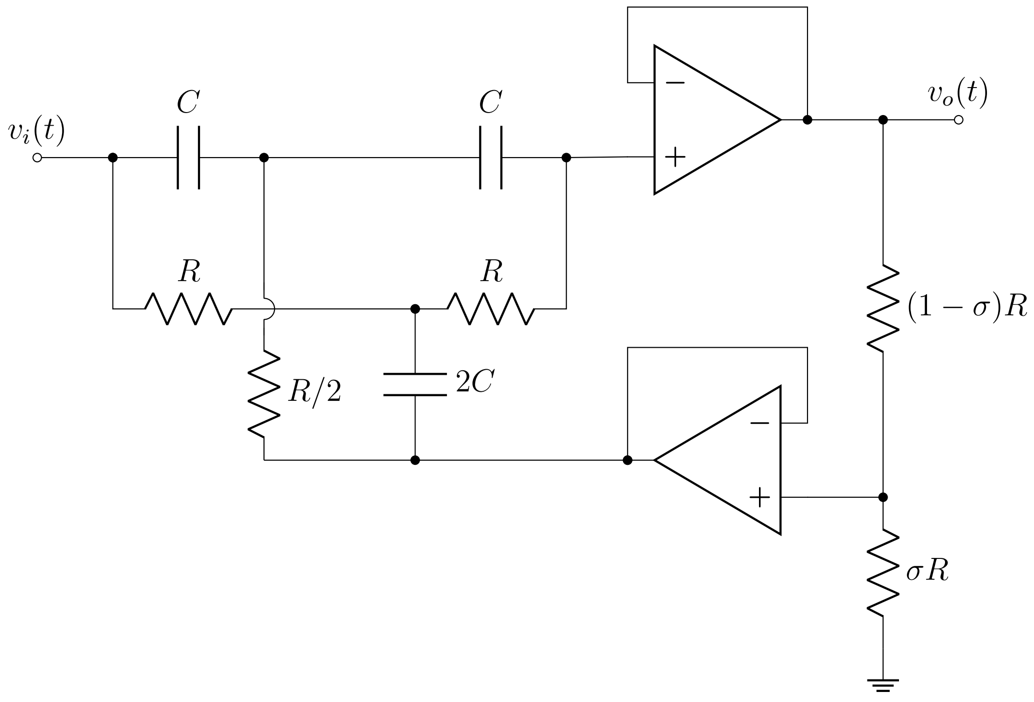

Figure 8.2: Notch filter circuit.

You will need to derive the transfer function for the circuit in Fig. 8.2. The derivation of this transfer function is very complicated. To make it easier, I would strongly use the `sympy` package in Python. This package is a symbolic math library that can help you derive the transfer function symbolically. I would recommend coming up either node voltage or mesh current equations, and then using the `sympy` package to solve for the output voltage V_o in terms of R, C, s, σ, and V_i.

Once you have the expression of V_o in terms of V_i, you can find the transfer function by dividing both sides by V_i and rearranging the equation. The transfer function will be in the form of

$$H(s) = \frac{V_o(s)}{V_i(s)}. \tag{8.3}$$

What is the transfer function in the form of R, C, σ, and s?

Using symbolic Python

Suppose we had a complicated system of equations that we wanted to solve. For example, suppose we had the following system of equations

$$\begin{aligned} 2Rx + Cy - sz &= 3 \\ Rx - 2Cy + sz &= -2 \\ 3Rx + Cy + 2sz &= 10 \end{aligned}$$

where R, C, and s are symbolic constants. We could use Cramer's rule to solve this system of equations, but it would be very tedious. Instead, we can use the `sympy` package to solve this system of equations. The `sympy` package is a symbolic math library that can help you solve systems of equations symbolically. The following code will do just that:

```python
import sympy as sp


# Define the variables
x, y, z = sp.symbols("x y z")
R, C, s = sp.symbols("R C s")  # Define symbolic constants

# Define the equations
eq1 = sp.Eq(2 * R * x + C * y - s * z, 3)
eq2 = sp.Eq(R * x - 2 * C * y + s * z, -2)
eq3 = sp.Eq(3 * R * x + C * y + 2 * s * z, 10)

# Solve the system of equations
solution = sp.solve([eq1, eq2, eq3], (x, y, z))

# Print the solution
print(solution)
```

Verifying the transfer function

In a file called `notch_filter.py`, write a function `H(s, R, C, sigma)` that computes the transfer function value at a given frequency s, resistor R, capacitor C, and σ. You can test whether your transfer function is correct by a provided unit test.

```
$ python test_notch.py

Testing import of the solution file
.
Testing transfer function
.
----------------------------------------------------------------------
Ran 2 tests in 0.066s

OK
```

One test will check whether the transfer function is correct. The second test checks if you imported the solution file (you should not import the solution file).

8.1.2 Finding component values

Once you have the transfer function, you can find the component values that will give you a notch at 60 Hz. To do this, you will need to set the center frequency f_0 to 60 Hz. Choose the σ value to give a decent Q factor. There is a tradeoff between the Q factor and the bandwidth of the notch. A higher Q factor will give you a narrower notch, but it will also make the filter more sensitive to changes in frequency (i.e., if the rejected frequency ω_0 shifts the filter might miss it). A lower Q factor will give you a wider notch, but it will also make the filter less sensitive to changes in frequency.

You will need to choose the component values R and C to give you the desired center frequency. The easiest way to approach this is to match the term in $H(s)$ that corresponds to ω_0^2. Choose a reasonable R or C value, and then solve for the other component value. Be mindful that there is a limited range of components that you can use, and you will need to choose values that are available in the lab. I would recommend starting by choosing a reasonable resistor value, and then calculating the capacitor value. You might need to place multiple capacitors in parallel to get the desired value.

Make a frequency response plot of the transfer function assuming ideal component values where you can get the null frequency exactly at 60 Hz. Also make a plot on the s-plane showing the poles and zeros of the transfer function.

Questions and tasks

- Derive the transfer function of the circuit $H(s)$ in terms of R, C, σ, and s.
- Find the component values that will give you a notch at 60 Hz. Choose a reasonable Q factor.
- Make a frequency response plot of the transfer function assuming ideal component values where you can get the null frequency exactly at 60 Hz. Also make a plot on the s-plane showing the poles and zeros of the transfer function.

8.2 Simulation

Using MultiSim or LTSpice (or whatever SPICE flavor you like) to simulate your circuit. Use the AC input, selecting `Simulate`→`Analyses`→`AC Analysis`, and assign V_{out} as the output. Make sure to set the AC input to 1 V so that you can compare the output voltage to the input voltage. You will need to set the frequency range to include 10 Hz to 1000 Hz with a logarithmic scale. This will give you a good idea of how the filter behaves around the notch frequency.

Questions and tasks

- Simulate the circuit in MultiSim or LTSpice. Include the frequency response plot in your write-up. Make sure the magnitude plot is in decibels and the phase plot is in degrees. Make sure to include the frequency range in your plot.

8.3 Building the circuit

Build your circuit on a breadboard. Use the LF353 op-amp. This is a low-power op-amp that is ideal for this application. It has a low noise figure and a low input bias current, which makes it ideal for use in low-level signal applications. The LF353 is also a dual op-amp, which means that you only need one chip to build the circuit.

As you build the circuit, I recommend check the output as a response to a sinusoidal input. You can use a function generator to generate a sinusoidal input at 60 Hz as an input. What output would you expect to see? What happens if you increase or decrease the input frequency? If the behavior does not match your expectations, check the circuit for errors. Make sure that all the components are connected correctly and that there are no shorts or opens in the circuit.

Once you are satisfied with the circuit, you can use the oscilloscope to measure the frequency response of the circuit. Use the built-in capabilities in the oscilloscope as described in Sec. 0.2.2. Sweep from $f \in [10\,\mathrm{Hz}, 1000\,\mathrm{Hz}]$. Record this data from the oscilloscope and plot it overlaid against the derived frequency response. Make sure to plot both the magnitude and phase response. In your write-up, comment on the similarities and differences between the actual and simulated data.

Next, let's see how well this notch performs in the presence of 60 Hz noise. Our function generator is capable of generating a cardiac ECG signal which we can corrupt by adding a 60 Hz noise signal. Make sure that Channel 1 is selected and select `Select Waveform`. Then select `Arb` on the side menu, then `Select Arb`, then `Select Internal`, then scroll to `Cardiac` which will provide a cardiac waveform. Hit the `Parameter` button on the function generator and change the sample rate of $450\,\mathrm{S\,s^{-1}}$. This would correspond to a heart rate of 60 bpm, which is reasonable. Adjust the amplitude to 1 Vpp.

Next, we will corrupt the cardiac signal with interference. On Channel 2, set up a 60 Hz sine wave with an amplitude of 100 mVpp. To combine the channels, go back to Channel 1. Hit `Setup`, then `Dual Channel`, and make sure that `Combine` is selected.

Record the filtered and pass-through data on a USB drive and plot both the input signal data and the filtered signal. Is the 60 Hz interference removed?

Questions and tasks

- Record the frequency response of the circuit using the oscilloscope. Make sure to include the magnitude and phase response in your write-up. Plot this data overlaid against the derived frequency response. Make sure to use proper labels and semi-log axes. The plots should match up, but there will be some differences due to component mismatches and tolerances.
- Plot both the input signal data and the filtered signal. Is the 60 Hz interference removed? What is the cutoff frequency of the filter?

8.4 Deliverables

8.4.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). Include all requested figures. Submit the PDF to Canvas. Upload your code to GitHub.

8.4.2 Grading

The lab assignment will have the following grade breakdown:

- Correct transfer function `H(s)`: 20pts.
- Theoretical frequency response: 10pts.
- *s*-plane plot: 10pts.
- SPICE simulation: 10pts.
- Measured frequency response: 20pts.
- ECG plot: 15pts.
- Report: 15pts.

Chapter 9

Second order low-pass filter system analysis

In electronics and circuit design, filters play a critical role in shaping and manipulating signals. Low-pass filters, in particular, are widely used to attenuate higher-frequency components and allow lower-frequency components to pass through. One popular circuit architecture for designing low-pass filters is the Sallen-Key architecture, known for its simplicity and versatility. In this laboratory exercise, we will explore the design and implementation of an under-damped low-pass filter using the Sallen-Key architecture.

This exercise aims to provide you with hands-on experience in designing and constructing an under-damped low-pass filter. Using the Sallen-Key architecture, you will learn filter design principles and gain insights into the trade-offs in selecting filter parameters. Additionally, students will understand the concept of damping and how it affects the filter's response. Through this exercise, you will strengthen your understanding of circuit theory and develop practical skills in designing and analyzing filters for various applications such as audio processing, data acquisition, and communication systems.

9.1 Background

The Sallen–Key circuit (using an LM353 or LM124 etc. op-amp) seen in Fig. 9.1 can implement a 2nd order low pass filter. The transfer function is often given in a reduced form

$$H(s) = \frac{\omega_c^2}{s^2 + 2\zeta\omega_c s + \omega_c^2}.$$

9.2 Procedure

The task of this lab is to build a low-pass filter with a cutoff frequency of 10 kHz ($\omega_c = 2\pi \times 10000$ and a damping ratio of $\zeta = 0.5$ (under-damped). As a reminder, the cutoff frequency is defined as the frequency at which the gain is -3dB, or $|H(j\omega_c)|^2 = \frac{1}{2}$. The damping ratio ζ is defined as

Derive the transfer function $H(s)$ in terms of R_1, R_2, C_1, and C_2. Choose resistors and capacitors appropriately. Determine the poles and zeros of the transfer function and plot them on the s-plane.

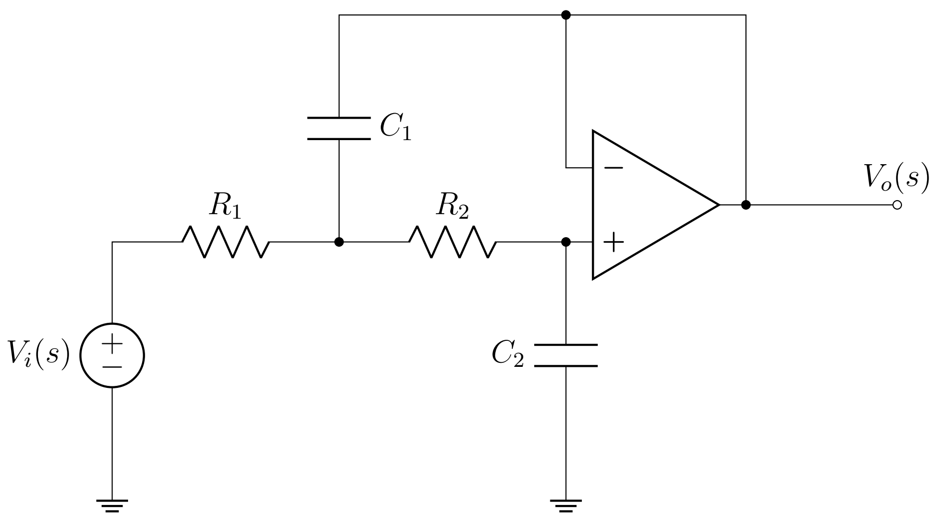

Figure 9.1: Sallen-Key 2nd order low pass filter.

Plot the ideal frequency response of the filter from 0 Hz to 50 kHz. Plot the magnitude response in decibels and the phase in degrees.

In a python file called `sallen_key.py`, implement the transfer function `H(s, R1, R2, C1, C2)`. Make sure the rest of your code is in a `main()` function so your code can be imported without executing. There is a test script called `test_sallen_key.py` that will test your code. You can run the test script with the command

```
$ python test_sallen_key.py

Testing import of the solution file
.
Testing transfer function
.
----------------------------------------------------------------------
Ran 2 tests in 0.060s

OK
```

Use MultiSim or LTSpice (or whatever SPICE flavor you like) to simulate your circuit. Use the AC input, selecting

`Simulate`→`Analyses`→`AC Analysis`,

and assign V_{out} as the output.

Build your circuit on a breadboard. With this lab, you have wide latitude with component selection, so choose whatever op-amp you feel most comfortable with (assuming we have it in stock), but the LF353 we used in prior labs will work great here. Measure the frequency response using the built-in capabilities in the oscilloscope as described in Sec. 0.2.2. Sweep from $f \in [50\,\text{Hz}, 50\,\text{kHz}]$. Record this data from the oscilloscope and plot it overlaid against the derived frequency response. Make sure to plot both the magnitude and phase response. In your write-up, comment on the similarities and differences between the actual and simulated data.

Questions and tasks

- Derive $H(s)$ symbolically in terms of circuit components R_1, R_2, C_1, and C_2.
- List your selected circuit component values to match the filter specifications.
- Plot the poles and zeros of the transfer function on the s-plane.
- Plot the theoretical frequency response with the measured frequency response. Make sure to use proper labels and semi-log axes. The phase and magnitude should be plotted on separate figures, but the measured and theoretical data should be overlaid on the same plot.
- Plot the SPICE simulation frequency response.

9.3 Deliverables

9.3.1 Submission

Write up the results of this lab in the technical memo format (see Canvas for a template example). Submit the PDF to Canvas. Upload your Python code to GitHub.

9.3.2 Grading

The lab assignment will have the following grade breakdown:

- Derivation of the transfer function: 10pts.
- Theoretical frequency response plots: 20pts.
- Pole/zero plot: 10pts.
- SPICE plots: 20pts.
- Measured frequency response plots: 20pts.
- Write-up: 40pts.

Chapter 10

Filter design

In the Shazam lab we sampled an audio signal at 48 kHz and digitally filtered and downsampled the signal to 8 kHz to reduce the amount of data that needed to be stored. If we had not used a digital low-pass filter, this would have led to aliasing in the downsampled signal. In this lab, we will design and implement an analog low-pass filter to remove the high frequency components of the audio signal before it is sampled.

The design strategy for this filter is largely up to you. You will can choose which type of filter to use, and how you want to implement it in hardware, so long as the filter hits the specifications. The specifications are as follows:

- The pass band is 0 Hz to 7.5 kHz.
- The stop band is 10 kHz to ∞.
- The maximum ripple in the pass band is 1.5 dB.
- The minimum attenuation in the stop band is -24 dB.

10.1 Design

The filter design is up to you. Be thoughtful in which filter you choose, and how you implement it. You can use the Butterworth, Chebyshev, or Elliptic filter design methods. You can implement the filter using op-amps, or you can use an active filter design. You can use a passive filter design, or you can use a combination of active and passive components. The choice is yours.

You are welcome to use the SciPy library to design your filter. You can use the `scipy.signal` module to design your filter, and the `scipy.signal.freqs` function to plot the frequency response of your filter. Make sure when designing the transfer function you are designing it as an analog filter, not a digital filter.

Once you have a suitable transfer function and have verified that it matches, or exceeds, the specifications, start designing an appropriate circuit implementation. Verify your design using your preferred SPICE simulator.

- What is the transfer function $H(s)$ of your filter?
- Plot the theoretical frequency response in Python.

- Include a sketch of your circuit.
- Plot the SPICE simulation frequency response.

10.2 Procedure

Build your circuit and connect its input $f(t)$, to the function generator. Configure the power supply to produce both a positive and negative voltage ($\pm 10\,\mathrm{V}$ will do nicely).

Generate a frequency response using the oscilloscope described in Sec. 0.2.2. Plot these results on the same plot as the theoretical frequency response of $H(s)$ you generated earlier. Do these plots agree? If they do not, check your circuit design and the connections and try again. (Small errors are normal due to the tolerances of the components.)

Record any additional observations, and write a conclusion in your memo summarizing what you have observed or discovered.

Questions and tasks

- Plot the circuit frequency response on the same Python plot as the theoretical frequency response.
- Does your hardware implementation match the theoretical and SPICE plots?

10.3 Deliverables

10.3.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise.

In the appendices, please include the requested plots. Your code should be submitted to GitHub.

10.3.2 Grading

The lab assignment will have the following grad breakdown.

- Plots of poles on complex plane: 10pts.
- Python plots: 15 pts.
- SPICE plots: 15pts.
- Circuit plots: 20pts.
- Write-up: 40pts.

Chapter 11

Discrete-Time Systems

Linear difference equations are fundamental in various areas of electrical engineering, providing mathematical models for discrete-time systems and digital signal processing. You will explore different techniques to solve and analyze these equations, gaining a deeper understanding of their behavior and practical applications.

Throughout the lab, you will tackle linear difference equations using three essential methods: recursive, convolutional, and Z-transform. The recursive method involves iterative calculations based on previous terms, simulating real-world systems with feedback loops. Conversely, the convolutional method relies on the convolution operation to compute the output response of a system given the input and impulse responses. Lastly, the Z-transform method provides a powerful tool for representing and analyzing discrete-time systems in the frequency domain.

By working through practical exercises and simulations, students will become proficient in applying these methods to solve and analyze linear difference equations. They will explore the advantages and limitations of each technique, gaining insights into their computational complexity, stability, and frequency response characteristics. Through hands-on experimentation, students will develop the skills to analyze and design discrete-time systems in various engineering applications.

Overall, this lab explores different approaches to solving linear difference equations in electrical engineering. You will acquire a versatile toolkit for understanding and manipulating discrete-time systems, enabling you to tackle real-world engineering problems involving digital signal processing, communication systems, and control systems in further courses.

11.1 Preliminary

In class, we talked about linear difference equations in the form

$$(E^n + a_1 E^{n-1} + \cdots + a_{n-1})y[k] = (b_0 E^m + b_1 E^{m-1} + \cdots + b_{m-1})f[k].$$

By taking the z-transform (and assuming that all initial conditions were set to zero), we were easily able to determine the transfer function in the form

$$H[z] = \frac{b_0 z^m + b_1 z^{m-1} + \cdots + b_{m-1}}{z^n + a_1 z^{n-1} + \cdots + a_{n-1}}.$$

We learned various methods to solve for $y[k]$ given some input $f[k]$. We will explore these in this lab. First, we must implement these routines given arbitrary $H[z]$ and input $f[k]$. We will implement this algorithm in a Python file called `ltid.py`. To make things interesting, to get full points on this assignment, you can only use one (1) `for` or `while` loop in the `ltid.py` file. This will ensure you write optimized NumPy code, which is generally good practice. Additionally, you cannot use `scipy` libraries for this project. Tests in `test_lab10.py` ensure you are on the right path.

11.2 Recursive Solution

First, we implement a recursive solver. Given some array `a` which is $[a_0, a_1, \dots, a_{n-1}]$, `b` which is $[b_0, b_1, \dots, b_{m-1}]$, and some arbitrary input $f[k]$, you need to numerically compute the output. We are interested in the zero-state solution, so we don't have to worry about initial conditions. While we generally assume that $a_0 = 1$, it still needs to be a part of the array `a`. The function definition will look like the following:

```
def recu_solution(f, b, a):
    """Computes the recursion solution for some system defined by a
    generic transfer function

        H[z] = b[0]*z^m + b[1]*z^{m-1} + ... + b[m-1]
               ---------------------------------------
               a[0]*z^n + a[1]*z^{n-1} + ... + a[n-1]

    Parameters
    ----------
    f : ndarray
        input function samples (starting at n=0)
    b : ndarray
        difference equation coefficients b[0], b[1],...,b[m-1] with
        length m
    a : ndarray
        difference equation coefficients a[0], a[1],...,a[n-1] with
        length n (in most cases a[0]=1)

    Raises
    ------
    ValueError
        checks if len(b) > len(a)

    Returns
    -------
    y : ndarray
        output array y
    """
```

You will also need to make sure that $m < n$ is similar to the systems that we looked at in class. You will need to raise a `ValueError` if $m > n$. You can assume that the counter $f[k]$ starts at $k = 0$.[1]

[1]This type of algorithm is VERY common in DSP filter implementations. There is a non-trivial chance you will code something similar at some point in your career.

Questions and tasks

- Make sure your code passes the test.

11.2.1 Determining the system impulse response

We can compute the impulse response from a difference equation. There are many approaches to doing this. Choose one and implement it in your `ltid.py` file. The function definition should look like the following:

```python
def find_impulse_response(b,a,N):
    """Finds the N first samples in the time-domain impulse
    response for the transfer function of the form

        H[z] = b[0]*z^m + b[1]*z^{m-1} + ... + b[m-1]
               --------------------------------------
               a[0]*z^n + a[1]*z^{n-1} + ... + a[n-1]

    Parameters
    ----------
    b : ndarray
        difference equation coeffiencts b[0], b[1],...,b[m-1] with
        length m
    a : ndarray
        difference equation coefficients a[0], a[1],...,a[n-1] with
        length n (in most cases a[0]=1)
    N : int
        number of samples for h[n]

    Raises
    ------
    ValueError
        checks that len(b) > len(a)

    Returns
    -------
    h : ndarray
        the first N samples of h[n]
    """
```

The `b` and `a` arguments are the same as the `recu_solution()` function. The `N` argument specifies how many samples you will generate with your impulse function. Oftentimes, an impulse function $h[k]$ will be as long as $k \to \infty$, so we must truncate $h[k]$ to N samples here.

Don't focus too much on an analytical solution. Remember that

$$h[k] = \mathcal{H}\{\delta[k]\}$$

which makes solving for $h[k]$ using your `recu_solution()` rather simple.

Questions and tasks

- Make sure your code passes the test.

11.2.2 System frequency response

We are often interested in determining the system's frequency response. One method we discussed is computing the DTFT, but as we discussed in class, this can be problematic because it would require us to take an infinite summation. Here, we want to write a routine that would compute the DTFT for a general system, so we won't know *a priori* if the summation will converge. An alternative approach is to work with the transfer function itself, similar to what we did in the continuous domain. In the discrete-time framework, if we know $H[z]$, we can let $z \to e^{j\Omega}$.

Write a function in `ltid.py` that computes $H(e^{j\Omega})$ over $\Omega \in [0, 2\pi]$. The function definition should look like the following:

```python
def frequency_response(b,a,N=1000):
    """Computes the frequency response (i.e., the equivalent of
    the DTFT) for the transfer function

        H[z] = b[0]*z^m + b[1]*z^{m-1} + ... + b[m-1]
               --------------------------------------
               a[0]*z^n + a[1]*z^{n-1} + ... + a[n-1]

    Parameters
    ----------
    b : ndarray
        difference equation coeffiencts b[0], b[1],...,b[m-1] with
        length m
    a : ndarray
        difference equation coefficients a[0], a[1],...,a[n-1] with
        length n (in most cases a[0]=1)
    N : int
        number of points in frequency Omega to compute the
        frequency response

    Raises
    ------
    ValueError
        checks if len(b) > len(a)

    Returns
    -------
    H : ndarray
        the complex frequency response (H(e^{j\Omega}) )
    Omega : ndarray
        the frequencies associated with H(e^{j\Omega}) )
    """
```

This function is set up very similarly to the other functions. The `N=1000` argument allows you to specify the number of points on $\Omega \in [0, 2\pi]$. The syntax of including the equal sign in the argument indicates that 1000 is the default value so you don't need to explicitly pass that argument when you use the function. However, if you were interested in using a different value for `N`, you could change that when you call the function.

The function will return the complex-valued array $H(e^{j\Omega})$ at frequencies defined by a second array Ω.

Questions and tasks

- Make sure your code passes the test.

11.3 Testing your methods

11.3.1 Looking a specific difference equation

Consider a linear and time-invariant difference equation defined as

$$(E^2 + 0.15E - 0.76)y[k] = (E - 0.25)f[k].$$

We will solve for $y[k]$ using numerical recursion, convolution in time with $h[k]$, and pre-determined z-transform. We will use two different input functions

$$f_1[k] = u[k]$$
$$f_2[k] = 0.98^k u[k].$$

For the recursive solution, compute the first 128 samples of $y_1[k]$ and $y_2[k]$ using your `recu_solution()` function.

Next, compute $y_1[k] = h[k] * f_1[k]$ and $y_2[k] = h[k] * f_2[k]$. You may use the `np.convolve()` function to make the project a bit easier once you have $h[k]$ from your `find_impulse_response()` routine. Use the first 128 samples for both $h[k]$ and $f_i[k]$ (i.e., $k = 0, 1, \ldots, 127$).

Lastly, you will need to generate the z-transform solution. To do this, use pencil and paper to work out the solution for $y_1[k]$ and $y_2[k]$ via the z-transform and write a function that returns both solutions as arrays. Yes, you will need to hardcode the solution. The functions definition will look like the following:

```
def find_z_transform(k):
    """Returns the pre-determined z-transform solutions for y_1[n]
    and y_2[n] for the difference equation

        (E^2 + 0.15*E - 0.76)*y[n] = (E - 0.25)*f[n]

    for some inputs f_1[n] = u[n] and f_2[n] = 0.98^n * u[n].

    Parameters
    ----------
    k : ndarray
        sample locations to compute y_1[n] and y_2[n]

    Returns
    -------
    y_1: ndarray
       the solution for y_1[n] where input is f_1[n]
    y_2: ndarray
       the solution for y_2[n] where input is f_2[n]
    """
```

Include the brief details of this derivation in your lab write-up.

Now that you have determined $y_1[k]$ and $y_2[k]$ via three different methods make a figure in your lab report with six subplots arranged in a 2×3 grid. The first row should show the three solutions of $y_1[k]$ on separate subplots, and the second row should show the three solutions of $y_2[k]$ on separate subplots. Since we are working with discrete-time signals, you should represent the signals as stem plots. Please plot just the first 50 samples of each of the signals. Your plots should be appropriately labeled. You may create the array of subplots in Python or LaTeX. Just make it look presentable.

Questions and tasks

- Make sure your code passes the test.
- Include 2×3 grid of stem plots of the various solutions.

11.3.2 Solution accuracy

Your solutions should match visually, but let's check more thoroughly. We can assume that the z-transform solution is the most accurate because the recursive approach could be subject to floating point arithmetic errors, and in the convolution case, we have to truncate $h[k]$. Thus, we will treat the z-transforms as the reference standard. To compute the error, we can compute the absolute difference between $y_{i,z\text{-transform}}$ and one of the other methods. For example, to compute the error in $y_1[k]$ for the recursive approach, we can compute

$$\text{Error}_{1,\text{recu}} = \sum_k |y_{1,z\text{-transform}}[k] - y_{1,\text{recu}}[k]|$$

Using the recursion method, find the error for each $y_i[k]$. Additionally, vary M (the length of $h[k]$) using values $M \in [16, 32, 64, 96, 128, 192, 256]$. For each value of M, find the error in the convolution method and $y_i[k]$. Make a table of these values. You should have 16 values in your table. What trend do you see?

Questions and tasks

- Make a table of the errors.
- What trend do you see?

11.3.3 Frequency response

Using your `frequency_response()` function, determine the frequency response of your system on $\Omega \in [0, 2\pi]$. Plot the magnitude and phase of $H(e^{j\Omega})$ on separate plots. These plots should neither be in dB nor logarithmic axes.

Let's look at the frequency response of the truncated $h[k]$. Compute the M-point DFT of $h[k]$ for $M \in [16, 32, 64, 96, 128, 192, 256]$. You may use the `np.fft.fft()` function to compute these DFTs. Plot the magnitude and phase on the same two plots above (don't use a stem or scatter plot to plot these lines; a normal `plt.plot()` should be sufficient). Ensure the two plots have legends to tell which line is which (each value of M should be a different color). Once again, combine the magnitude and phase plots as subplots in one figure in your report. Comment in the report what trend you notice as M increases. At what point is that length sufficient to represent the true frequency response of the system? Does this match the errors you found in the convolution solutions?

Questions and tasks

- Plot the truncated $h[k]$ frequency responses.
- When does M become sufficiently large such that the truncation effects are negligible?

11.4 Tests

Your code will be subject to several unittests. I strongly encourage you to look at what each test is checking. Most of them are looking for numerical accuracy. A few will check that you are raising exceptions when necessary. And one test will ensure you aren't using too many loops.

You might consider moving the code to generate the figures needed for the report to a separate Python script called `ece3210_lab10.py` and importing the `ltid` functions there. This will allow you to use a loop for cycling through different values of M in the convolution solution portion to generate your error table and frequency response plots. Alternatively, you could move that code to a `main()` in `ltid.py`; you would be penalized if you used additional loops there to generate your plots.

11.5 Deliverables

11.5.1 Submission

Please write up the results of this lab in the technical memo format (see Canvas for a template example). In this memo, please give a brief overview of the lab exercise.

In the appendices, please include the requested plots. Your code should be submitted to GitHub.

11.5.2 Grading

The lab assignment will have the following grade breakdown.

- `recu_solution()`: 30pts.
- `find_impulse_response()`: 15pts.
- `frequency_response()`: 15pts.
- `find_z_transform`: 20pts.
- Stem plots: 20pts.
- Error table: 15pts.
- Frequency plots: 15pts.
- Write-up: 20pts.

You may use only one loop (either `for` or `while`) in the `ltid.py` file. Each additional loop will incur a 20pt. penalty.

Bibliography

[1] Avery Wang et al. “An industrial strength audio search algorithm.” In: *Ismir*. Vol. 2003. Washington, DC. 2003, pp. 7–13.