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

November 12, 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, you will put theory into practice by tackling real-world problems. You will gain insights into the design process, from conceptualizing an idea to developing algorithms and implementing them in software or hardware. Along the way, you will 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 You 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 You 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, you 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 LaTeX template that you can download on Canvas. LaTeX is 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 LaTeX documents, but I recommend using an online editor like Overleaf. Overleaf is a cloud-based LaTeX editor 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 LaTeX documents. You can also download the LaTeX source files and compile them on your local machine if you prefer.

If you choose to use Overleaf, you will need to create an account. Once you have an account, you can upload the technical memo template files as a zip file. To do this, click on the `New Project` button and select `Upload Project`. You can then select the zip file containing the template files. Overleaf will extract the files and create a new project for you. After you create the project in Overleaf, you can edit the files in the web editor. The editor provides syntax highlighting, autocompletion, and other features to make writing LaTeX documents easier. You can also compile the document by clicking on the `Recompile` button. Overleaf will show you the output of the compilation in a preview window. You can also download the compiled PDF document by clicking on the `Download PDF` button. There is a bit of a learning curve to using LaTeX , but Overleaf provides a lot of documentation and tutorials to help you get started. You can also find many resources online, such as the Overleaf documentation and the LaTeX project documentation.

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 (Chapter 8)
2. Hardware (Chapters 2, 3, 4, 5, 6, 7, 10, 11)

The software lab is designed to reinforce the ideas you learned in class, particularly 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 kΩ resistor	1
0.033 µF capacitor	1
1 mH inductor	1

1.1.2 Repository

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

```
ece3210-lab01
```

```
├── ece3210_intro_sol.pyc
└── test_intro.py
```

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 $f(t)$.

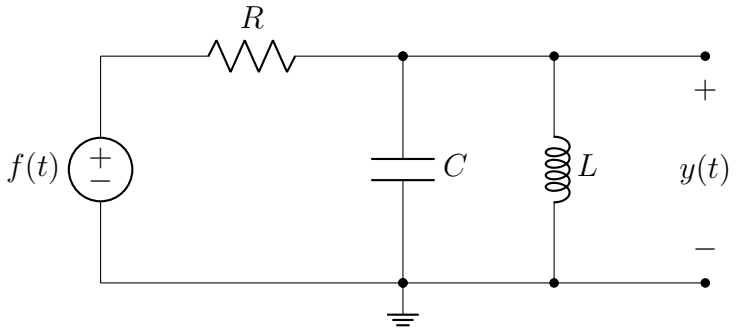


Figure 1.1: Series/parallel resonant circuit

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.

1.3 Computation

> In high schools and colleges, we teach 19th century mathematics to solve 19th physics problems.
>
> —*Steven Boyd, Stanford EE 364a, Winter 2012*

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* 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:

```
y = [4.75, 9, 12.5, 14.5]
y_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.)

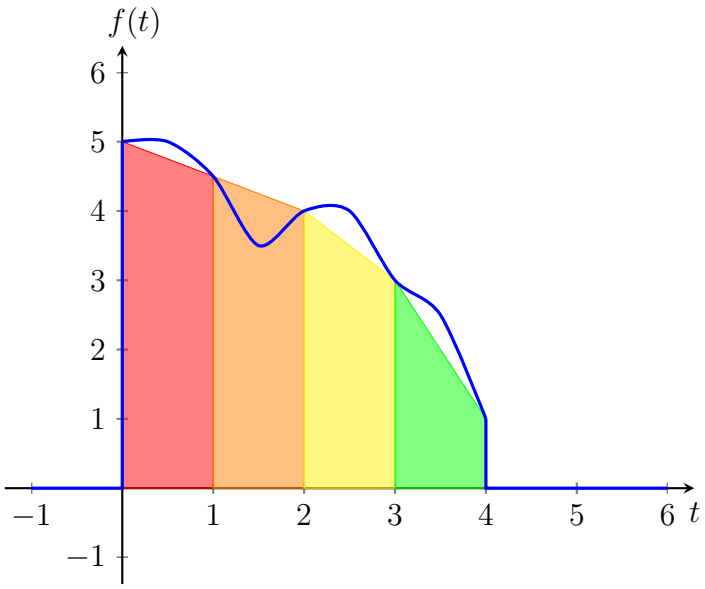


Figure 1.2: Some $f(t)$ and trapezoids for numerical integration

In a file called `ece3210_intro.py`, write a function called `py_cumtrapz()` that should have the following form.

```
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 Analytical 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_intro.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. Please note that the function you are integrating is defined as $f(t) = 5e^{-t}u(t)$, so you will need to define your array for $f(t)$ to capture the behavior

of the unit step function.

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_intro.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
```

When you push your repository to GitHub, the tests will run automatically. If your code does not pass the tests, you will see a red "X" next to your commit. You can click on this to see what tests failed and why. If the test passes, you will see a green checkmark next to your commit. You can click on the "X" or checkmark, you can select `Details` which brings you to a page with more information about the test run. On this page, you can see details about each test as well as the grades you will receive for each test. These scores are what will determine your coding grade for the lab.

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:

```
ece3210-lab01
├── ece3210_intro.py
└── ece3210_intro_sol.pyc
```

```
└── test_intro.py
```

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: 10pts.
- Frequency response plot: 20pts
- Correct `py_cumtrapz()`: 30pts
- Correct `analytical_integral()`: 15pts
- Numerical integration plot: 10pts
- Report: 15pts

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 kΩ resistor	1
0.033 µF capacitor	1
1 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 $R = 1\,\text{k}\Omega$, $C = 0.033\,\mu\text{F}$, and $L = 1\,\text{mH}$. Derive a differential equation governing this circuit. In a Python file called `ece3210_impulse.py`, write a function called `ode_coeffs()` that returns the coefficients of the differential equation in standard form

$$\frac{d^2y}{dt^2} + a_1\frac{dy}{dt} + a_0 y(t) = b_2\frac{d^2x}{dt^2} + b_1\frac{dx}{dt} + b_0 x(t).$$

```
def ode_coeffs(R, L, C):
    """
    Calculate the coefficients for a second-order ODE representing an RLC
   circuit.
```

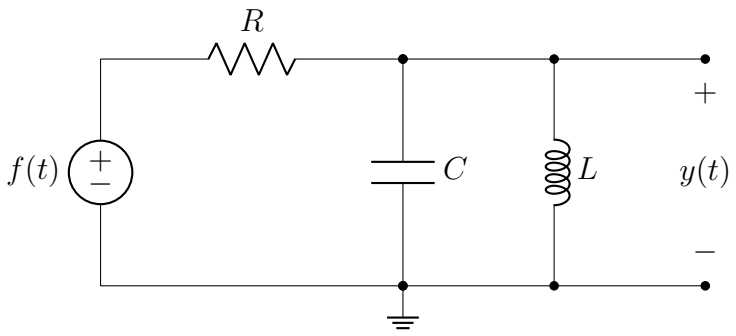


Figure 2.1: Series/parallel resonant circuit

```
    Parameters
    ----------
    R : float
        Resistance in ohms.
    L : float
        Inductance in henries.
    C : float
        Capacitance in farads.

    Returns
    -------
    a_1 : float
        Coefficient for the first derivative term.
    a_0 : float
        Coefficient for the zeroth derivative term.
    b_2 : int
        Coefficient for the second derivative input term (always 0).
    b_1 : float
        Coefficient for the first derivative input term.
    b_0 : int
        Coefficient for the zeroth derivative input term (always 0).
    """

    # ASSIGN VARIABLES HERE

    return a_1, a_0, b_2, b_1, b_0
```

Next, 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. Write a function in your `ece3210_impulse.py` file called `impulse_function(t)` that computes $h(t)$ for given time values `t`. The function definition looks like the following:

```
def impulse_response(t):
    """
    Compute the impulse response h(t) of a series RLC circuit.
    Parameters
    ----------
    t : array_like or float
        Time(s) at which to evaluate the impulse response.
    Returns
    -------
```

```
    h_t : ndarray or float
        The impulse response of the RLC circuit evaluated at time(s) `t`.
    Notes
    -----
    The circuit parameters are:
        R = 1 kOhm (resistance)
        C = 0.033 uF (capacitance)
        L = 1 mH (inductance)
    The impulse response is calculated using the roots of the characteristic
   equation
    for the series RLC circuit and is valid for underdamped cases.
    Examples
    --------
    >>> import numpy as np
    >>> t = np.linspace(0, 0.01, 100)
    >>> h = impulse_response(t)
    """
```

You are provided with unit tests for both `ode_coeffs()` and `impulse_response()` functions to help you verify your derivation. A successful test call is:

```
$ python test_impulse.py

Testing for impulse response function
.
Testing for ODE coefficients
.
----------------------------------------------------------------------
Ran 2 tests in 0.011s

OK
```

In the `main()` function in your `ece3210_impulse.py` file, generate a plot of $h(t)$ over the interval $[0,\ 300\,\mu s]$. 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?

2.1.4 Hardware implementation

The goal of this laboratory procedure is to verify that the impulse response computed in the preliminary section matches physical reality on a breadboard. 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 demonstration purposes. Using the function generator, create a square wave with an amplitude of 10 V, an offset of 5 V, a frequency of 1 kHz, and a pulse width of 1 µs. This will create a pulse that is 10 V tall and 1 µs wide[1]. You will need to set the output load to `High-Z`. This pulse approximates a $10^{-5}\delta(t)$ because the area under the pulse is $10\,V \times 1\,\mu s = 1 \times 10^{-5}\,V\,s$ (as opposed to $1\,V \cdot s$

[1]You can achieve a similar pulse using the Pulse waveform.

like a true $\delta(t)$ would be).

Save the first 300 µs after $t = 0$ worth of data from the oscilloscope as described in Sec. 0.2.1. If you were to plot this against the theoretical response, you would notice that the measured data has a *much* lower amplitude than the theoretical response. That is because we essentially used a $10^{-5}\delta(t)$ input to the system. We can fix this by multiplying the measured response by 10^5. Add this scaled measured data on the same plot of the theoretical $h(t)$ that you made earlier. How do the two plots compare? How do they differ?

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

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:

```
ece3210-lab02
├── ece3210_impulse.py
├── ece3210_impulse_sol.pyc
└── test_impulse.py
```

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

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:

- ODE: 10pts.
- Derivation: 30pts.
- Python plot: 20pts.
- Circuit plot: 20pts.
- Report: 20pts.

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 kΩ resistor	1
0.047 µF 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 $R = 10\,\text{k}\Omega$ and $C = 0.047\,\mu\text{F}$. 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 `convolution.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:

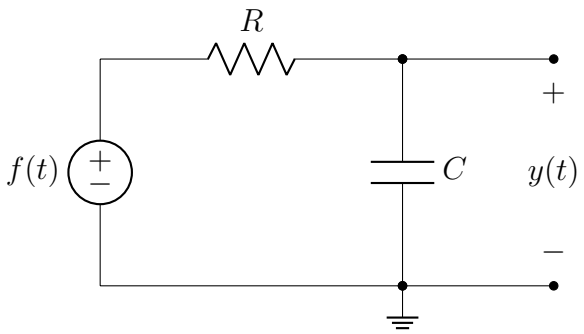


Figure 3.1: Basic RC circuit

```
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 the `convolution.py` file, 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 the `convolution.py` file, 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_convolution.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_convolution.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 `convolution.py` file, make sure you put it in a `main` section so that it will not run when the tests are run.

Because the impulse response is so small there will likely be a bias added to the signal. You can remove this bias by subtracting the steady-state portion of the signal (after the decay) of the signal from the signal itself. For example, if your measured impulse response is stored in a NumPy array called `measured_impulse`, you can remove the bias like this:

```
steady_state_value = np.mean(measured_impulse[idx])  # whatever indices give
    some values in the steady state
corrected_impulse = measured_impulse - steady_state_value
```

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 `convolution.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 `convolution.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.

Your complete repository should contain the following files:

```
ece3210-lab03
├── convolution.py
├── convolution_sol.pyc
├── ramp_wave.arb
└── test_convolution.py
```

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 sinusoid is used an input to some system, 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*.

Harmonic distortion is important in many applications, as it can degrade sound quality in audio systems and lead to poor power quality in electrical systems. In this exercise, we will analyze harmonic distortion by examining 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
220 kΩ 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.

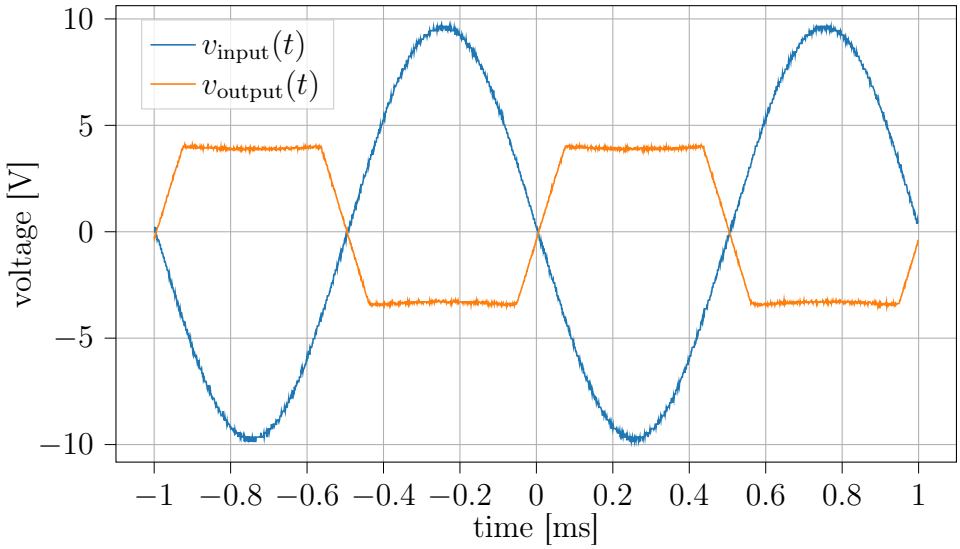


Figure 4.1: Clipping of a sinusoidal signal

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.

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

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.

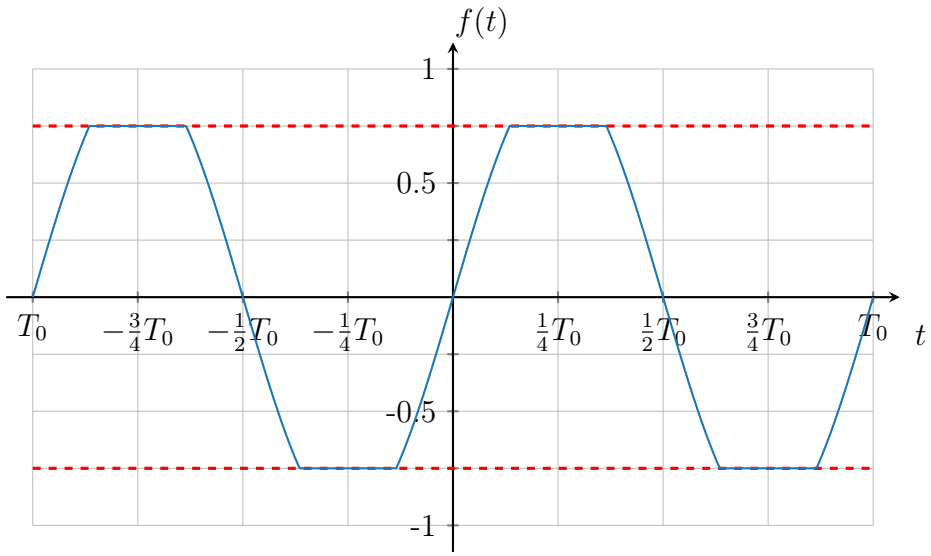


Figure 4.2: Clipped sine example

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:

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

    Parameters
    ----------
    n : int
        Harmonic number
    level : float
        clipping level
    omega_0 : float
```

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

```
        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

$$\text{THD} = \frac{\sqrt{b_2^2 + b_3^2 + b_4^2 + \dots}}{b_1}. \tag{4.2}$$

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:

```
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 with 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_thd.py

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

OK
```

Questions and tasks

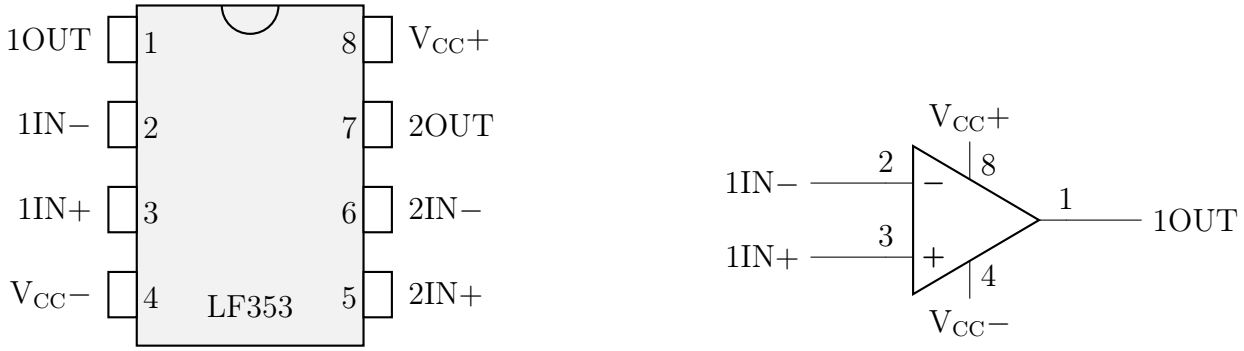
- Include a derivation for the b_n coefficients in your report.
- Include a plot of the clipped sine waves showing at least one period of the signal.
- What is the relationship between the clipping level and the THD?
- What is the relationship between the number of harmonics and the THD?

4.3 Experimental system responses

4.3.1 Experimental setup

Next, we will build the circuit seen in Fig. 4.4. Make sure that you have the correct resistor values and that the op-amp is connected correctly. The LF353 pinout and circuit schematic (with pins) are seen in Fig. 4.3. Please ensure that the power supply is connected properly. If it is

not connected properly you risk damaging the components. Set the rail voltage to the op-amp to $\pm 5\,\mathrm{V}$.



(a) LF353 op-amp pinout

(b) LF353 circuit schematic

Figure 4.3: LF353 op-amp pinout and circuit schematic

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 $6\,\mathrm{V_{pp}}$. You will notice that there is little clipping at this lower voltage.

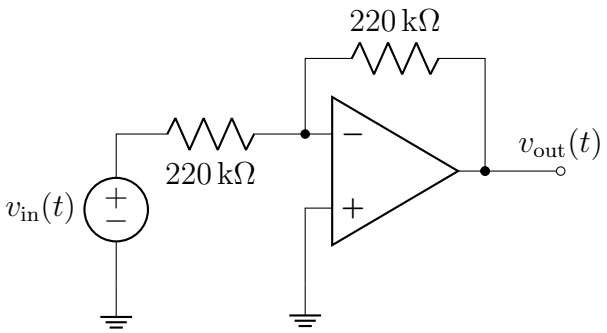


Figure 4.4: Clipping circuit

4.3.2 Data collection

Decrease the time resolution on the oscilloscope to $2\,\mathrm{ms\,div^{-1}}$. 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 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 $2\,\mathrm{V_{pp}}$ and retake the same measurements. Keep incrementing the amplitude until the function generator maxes out at $20\,\mathrm{V_{pp}}$.

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:

```
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.

Your complete repository should contain the following files:

```
ece3210-lab04
├── clipped_sine.py
├── clipped_sine_sol.pyc
├── data_thd.py
└── test_thd.py
```

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.

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).

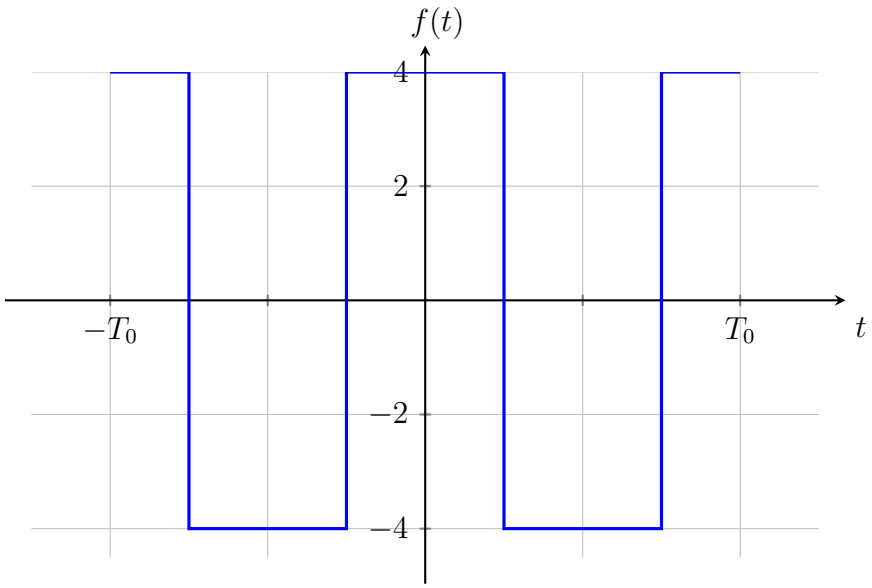


Figure 5.1: 4 V square wave.

In a Python file called `periodic_input.py`, write a function `f(t, omega_0, m)` that implements the exponential Fourier series for the square wave as seen in Fig. 5.1. The function should take in time `t`, fundamental frequency `omega_0`, and number of harmonics `m` as arguments. The function should return the value of the Fourier series at time `t`. The function definition should look like this:

```
def f(t, omega_0, m):
    """
    Compute the Fourier series approximation of a periodic signal.

    This function calculates a truncated Fourier series representation of a
   periodic
    signal using complex exponential form. The signal appears to be a square
   wave
    or similar periodic function based on the Fourier coefficients.

    Parameters
    ----------
    t : float or array_like
        Time values at which to evaluate the Fourier series (seconds).
    omega_0 : float
        Fundamental angular frequency of the periodic signal (rad/s).
    m : int
        Number of harmonics to include in the approximation. The series will
        include harmonics from -m to +m, for a total of 2m+1 terms.

    Returns
    -------
    complex or ndarray
        Complex-valued Fourier series approximation evaluated at time(s) t.
```

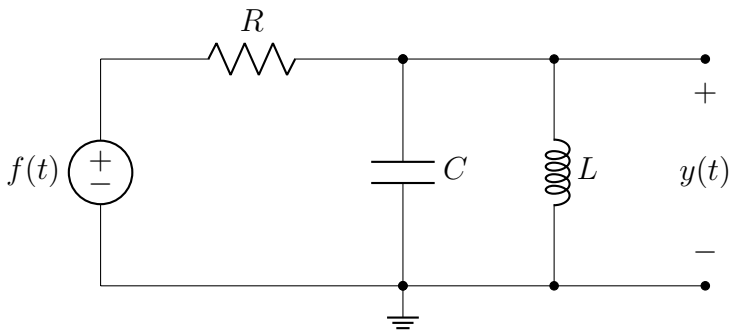


Figure 5.2: Series/parallel resonant circuit

```
        If t is a scalar, returns a complex number. If t is an array,
        returns a complex array of the same shape.

    """
```

Assume a frequency of 20 kHz and using $m = 100$ harmonics, plot the Fourier series approximation of the square wave. Make sure your plot has contains two periods of the square wave.

Questions and tasks

- What is the complex exponential Fourier series representation of $f(t)$?
- Generate a plot of the Fourier series approximation of the square wave.

5.1.4 Circuit analysis and simulation

Find the ODE governing the circuit in Figure 5.2. From the ODE, derive the transfer function $H(s)$ from this ODE.

Just to verify your result, write a function in your `periodic_input.py` called `H(s, R, L, C)` that computes the transfer function $H(s)$ for the circuit in Fig. 5.2. The function definition should look like this:

```
def H(s, R, L, C):
    """
    Compute the transfer function of a series RLC circuit (bandpass filter).

    This function calculates the transfer function H(s) for a series RLC
   circuit
    configured as a bandpass filter. The transfer function represents the ratio
    of output voltage (across the resistor) to input voltage.

    Parameters
    ----------
    s : complex or array_like
        Complex frequency variable.
        Units: rad/s
    R : float
        Resistance value in ohms.
    L : float
        Inductance value in henries.
```

```
    C : float
        Capacitance value in farads.

    Returns
    -------
    complex or ndarray
        Complex transfer function H(s) evaluated at the given s value(s).
        Dimensionless (voltage ratio).

    """
```

From lecture we learned that the output of an LTI system to a periodic input can be found by evaluating the system's frequency response at the harmonics of the input signal and scaling by the Fourier series coefficients

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

Modify your `periodic_input.py` file to include a function called `y(t, omega_0, m, R, L, C)` that computes the output of the circuit in Fig. 5.2 to the square wave input. The function definition should look like this:

```
def y(t, omega_0, m, R, L, C):
    """
    Compute the system response to a periodic input signal.

    This function calculates the output of a series RLC bandpass filter when
    excited by a periodic input signal. The computation is performed in the
    frequency domain by multiplying each Fourier component of the input by
    the corresponding transfer function value.

    Parameters
    ----------
    t : float or array_like
        Time values at which to evaluate the system response (seconds).
    omega_0 : float
        Fundamental angular frequency of the input periodic signal (rad/s).
    m : int
        Number of harmonics to include in the input signal approximation.
        The computation includes harmonics from -m to +m.
    R : float
        Resistance value of the RLC circuit in ohms.
    L : float
        Inductance value of the RLC circuit in henries.
    C : float
        Capacitance value of the RLC circuit in farads.

    Returns
    -------
    complex or ndarray
        Complex-valued system response y(t) evaluated at time(s) t.
        If t is a scalar, returns a complex number. If t is an array,
        returns a complex array of the same shape.

    """
```

On the same plot where you plotted the Fourier series approximation of $f(t)$, plot the system response $y(t)$ using $R = 1\,\text{k}\Omega$, $C = 0.033\,\mu\text{F}$, and $L = 1\,\text{mH}$ for the component values.

Questions and tasks

- What ODE governs the circuit in Fig. 5.2?
- What is the system output given this square wave input $f(t)$?
- Plot the system response to the square wave input.

5.1.5 Testing

Unit tests are provided in your repository to test if your functions are numerically correct. A successful run of the tests will look like this:

```
$ python test_periodic_input.py

Testing for transfer function H(s)
.
Testing for Fourier series derivation
.
Testing for system response y(t)
.
----------------------------------------------------------------------
Ran 3 tests in 0.194s

OK
```

5.1.6 Hardware implementation

Connect the $1\,\text{k}\Omega$ resistor, the $0.033\,\mu\text{F}$ capacitor, and the $1\,\text{mH}$ 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 $20\,\text{kHz}$ square wave with a $4\,\text{V}$ amplitude. Record two periods of the input and output on the oscilloscope. You may need to adjust the time base and voltage scale to get a good view of the signals. Save the data and add the recorded data to your plot of $f(t)$ and $y(t)$. Ideally, the recorded data for the circuit output should match your computed response $y(t)$.

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.

Your complete repository should contain the following files:

```
ece3210-lab05
├── periodic_input.py
├── periodic_input_sol.pyc
└── test_periodic_input.py
```

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

DSB-SC Modulation

In communications systems, information is often transmitted from location to the other using electromagnetic waves. Usually, the signal that contains the information we want to transmit are at low frequencies, which are not suitable for transmission. To overcome this limitation, we use modulation techniques to shift the baseband signal to a higher frequency range, allowing it to be transmitted more effectively over long distances. We will not get into the specifics of transmission in this course; for now, we will focus on the modulation technique itself.

In this lab exercise, we will explore Double Sideband Suppressed Carrier (DSB-SC) modulation, a technique that allows efficient transmission of information by modulating a message signal $m(t)$ onto a carrier signal. There are several modulation techniques, but DSB-SC is one of the simplest and most widely used. It is a form of amplitude modulation where the message signal $m(t)$ is multiplied by some carrier signal, resulting in a modulated signal that contains both upper and lower sidebands.

Here, we will analyze this approach mathematically and implement it in hardware. We will generate a modulated signal using a message signal and a carrier signal, and then demodulate it to recover the original message. This exercise will provide insights into the principles of modulation and demodulation, as well as the practical aspects of implementing these techniques in a laboratory setting.

6.1 Background

Consider some message signal $m(t)$ that we wish to transmit. We can modulate this signal onto a carrier signal which is usually a high-frequency sinusoidal signal. The modulated signal $x_c(t)$ can be written as

$$x_c(t) = m(t) \cdot \cos(\omega_c t),$$

where ω_c is the carrier frequency. From class, we know that the Fourier transform of this modulated signal is given by

$$X_c(j\omega) = \frac{1}{2}M(j(\omega - \omega_c)) + \frac{1}{2}M(j(\omega + \omega_c)).$$

Suppose the message signal $m(t)$ is bandlimited to $[-\omega_m, \omega_m]$, then the modulated signal $x_c(t)$ will have a spectrum that is shifted to the right and left of the carrier frequency ω_c. The resulting spectrum will be non-zero in the range $[\omega_c - \omega_m, \omega_c + \omega_m]$. This means that the modulated signal contains two sidebands, one above the carrier frequency and one below it. The carrier

frequency itself is suppressed, hence the name Double Sideband Suppressed Carrier (DSB-SC) modulation.

This represents a significant advantage over traditional amplitude modulation (AM) techniques, where the carrier frequency is transmitted along with the sidebands, because suppressing the carrier allows for more efficient use of bandwidth and power. Because the spectrum is non-zero only in the range $[\omega_c - \omega_m, \omega_c + \omega_m]$, we can transmit the modulated signal using a bandwidth of $2\omega_m$.

To recover the original message signal $m(t)$ from the modulated signal $x_c(t)$, we can use a demodulation technique. One common method is to multiply the modulated signal by a local oscillator signal at the carrier frequency, which effectively shifts the spectrum back to baseband. The demodulated signal can then be filtered to recover the original message.

Mathematically, suppose we receive the modulated signal $x_c(t)$ and we want to demodulate it. We can multiply it by a local oscillator signal $\cos(\omega_c t)$

$$\begin{aligned} x_d(t) &= x_c(t) \cdot \cos(\omega_c t) \\ &= m(t) \cdot \cos(\omega_c t) \cdot \cos(\omega_c t) \\ &= \frac{1}{2} m(t) \cdot (1 + \cos(2\omega_c t)) \\ &= \frac{1}{2} m(t) + \frac{1}{2} m(t) \cdot \cos(2\omega_c t). \end{aligned}$$

This has the spectrum

$$X_d(j\omega) = \frac{1}{2} M(j\omega) + \frac{1}{4} M(j(\omega - 2\omega_c)) + \frac{1}{4} M(j(\omega + 2\omega_c)).$$

We notice that the first term is the original message signal $m(t)$, while the second and third terms are shifted versions of the message signal to $\omega - 2\omega_c$ and $\omega + 2\omega_c$, respectively. We can filter out these unwanted terms using a low-pass filter to recover the original message signal.

A high-level block diagram of this system is shown in Fig. 6.1. This representation also includes a "transmission" block, which represents the channel through which the modulated signal is transmitted. In practice, this could be a physical medium such as a wire or wireless channel. However, we are going to ignore the transmission block in this lab exercise and focus on the modulation and demodulation process.

6.2 Procedure

6.2.1 PCB and low-pass filter

We will implement the modulation in hardware, by using the AD633 multiplier IC, which is a four-quadrant analog multiplier. This IC can be used to multiply the message signal $m(t)$ with the carrier signal $\cos(\omega_c t)$ to produce the modulated signal $x_c(t)$. The AD633 is expensive, so you are provided a PCB that contains the AD633 and other components to make it easier to use. The PCB is shown in Fig. 6.2. This PCB has 14 input sockets, though only 13 are used. They are labeled in the figure, and are seen on the silkscreen on the PCB itself. The inputs are as follows (from top to bottom):

- `+17V`: This is the positive power supply to the board. The AD633 requires a dual power supply, so this should be connected to +17V.

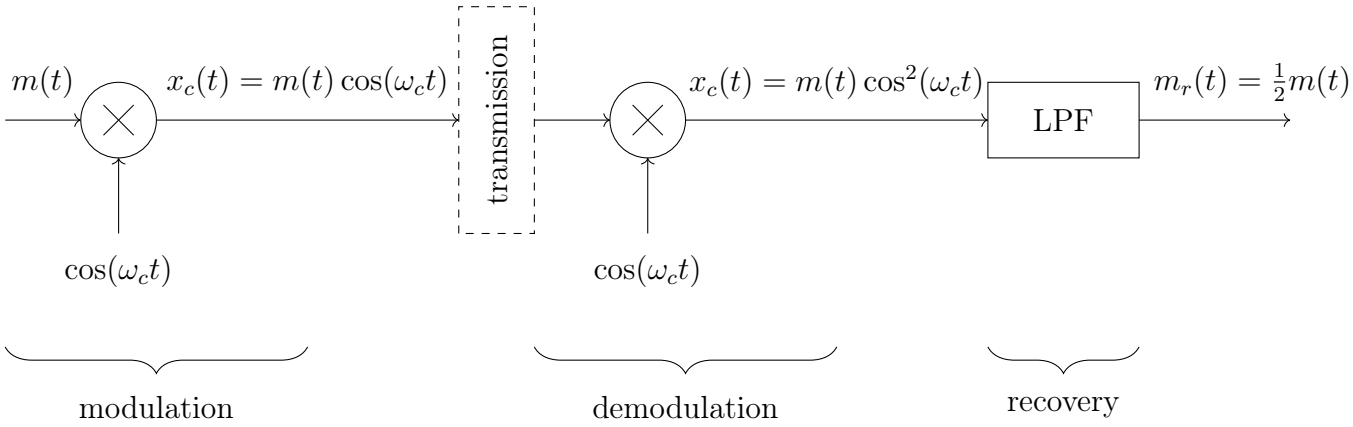


Figure 6.1: Block diagram of DSB-SC modulation and demodulation.

- `-17V`: This is the negative power supply to the board. The AD633 requires a dual power supply, so this should be connected to -17V.
- `GND`: This is the ground connection for the board.
- `x1(t)`: This is the input for the first signal to be multiplied (the message signal $m(t)$).
- `y1(t)`: This is the input for the second signal to be multiplied (the carrier signal $\cos(\omega_c t)$).
- `out1(t)`: This is the output of the first multiplier (the modulated signal $x_c(t)$).
- `x2(t)`: This is the input for the first signal to be multiplied (the message signal $m(t)$).
- `y2(t)`: This is the input for the second signal to be multiplied (the carrier signal $\cos(\omega_c t)$).
- `out2(t)`: This is the output of the second multiplier (the modulated signal $x_c(t)$).
- `x_filt(t)`: This is the input for the low-pass filter (not used in this lab).
- `y_filt(t)`: This is the output of the low-pass filter (not used in this lab).
- `CLK`: This is the clock signal for the AD633 (not used in this lab).
- `+5V out`: This is the +5V output from the voltage regulator (not used in this lab).
- (unconnected)

You will need a low-pass filter to recover the original message signal $m(t)$ from the demodulated signal $x_d(t)$[1]. Consider the active low-pass filter design in Fig. 6.3. This is a third-order Chebyshev low-pass filter with a cutoff frequency of 4.7 kHz. Build this filter using components in the lab. You are welcome to use any op-amp you like, but I prefer the LF353, which is a dual op-amp that is easy to use. Please read the datasheet for whatever op-amp you choose to use, and make sure you understand how to use it. After building the filter, measure its frequency response using the oscilloscope. You should see a cutoff frequency of around 4.7 kHz with some oscillation in the

[1]The PCB has a *very* good low-pass filter that would work for this lab. However, to drive this filter we would need to generate a clock signal to set the corner frequency, which would require a channel from the function generator. Each lab station has three function generator channels (two on the function generator and one on the oscilloscope), and we will use all three for other purposes in this lab. The filter described in Fig. 6.3. are sufficient.

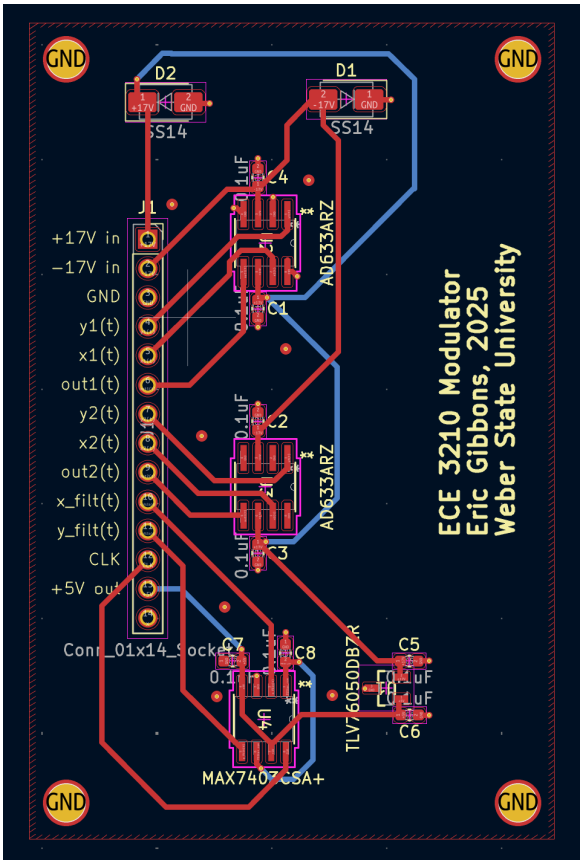


Figure 6.2: PCB for DSB-SC modulation and demodulation using the AD633 multiplier IC.

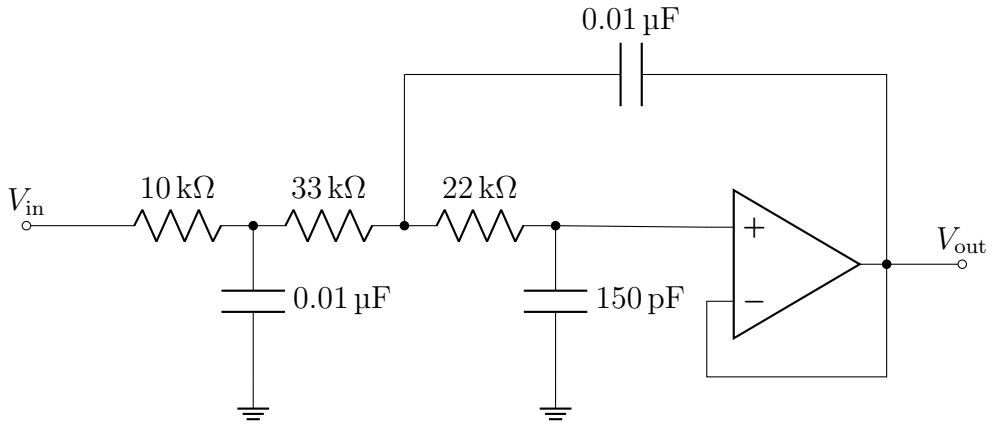


Figure 6.3: Active low-pass filter design for DSB-SC signal recovery.

passband before the cutoff[2]. Include the frequency response plot in your lab report.

Questions and tasks

- Include a plot of the frequency response of the low-pass filter in your report.

6.2.2 Coherent demodulation

The demodulation described in Section 6.1 is known as coherent detection. This technique is effective when the carrier frequency is known and can be accurately generated. However, it requires precise synchronization between the local oscillator and the carrier frequency, which can be challenging in practice. For our first experiment, we are going to assume that we have perfect synchronization, and we will implement the coherent demodulation technique in hardware.

For the message signal $m(t)$, we will use the function generator. For your message signal $m(t)$, use the waveform generator in the oscilloscope to generate a $5\,V_{pp}$ sine wave with a frequency of 1 kHz. For the carrier signal $\cos(\omega_c t)$, we will use the first channel of the function generator. Set this to a frequency of 9 kHz and a $20\,V_{pp}$ amplitude. Make sure the output load is set to `High-Z`. Connect the outputs of the function generators to the inputs of the first AD633 multiplier PCB, which is shown in Fig. 6.2.

Measure the output of the first AD633 multiplier PCB using the oscilloscope. You should see a modulated signal that is a sine wave with a frequency of 9 kHz and an amplitude that varies with the message signal $m(t)$. You will notice that signal will "move" on the oscilloscope. This is because the waveform generatoron the oscilloscope and the standalone waveform generator are not synchronized. That is fine. You can just push `Stop` on the top right corner of the oscilloscope to get a freeze-frame view. The output of the AD633 will be the modulated signal $x_c(t)$. Record this waveform. Include this in your report. Measure the frequency response using the FFT[3] function of the oscilloscope. Set the center to be 25 kHz and the span to be 25 kHz with the amplitude set to `Vrms`. You will likely need to decrease the time resolution on the oscilloscope—which increases the time duration of the signal shown—in order to get better frequency resolution. A time resolution of 5 ms per division should be sufficient. You should see two peaks at 8 kHz and 10 kHz, which

[2]We will learn how to design these filters later in the semester

[3]The FFT is just a computational method to compute the Fourier transform.

correspond to the upper and lower sidebands of the modulated signal. Record this frequency response.

Next, connect the output of the first multiplier to the input of the second multiplier. For the other input of second multiplier, use the second channel of the function generator to generate a $20\,V_{pp}$, 9 kHz waveform (same settings for the first channel). You need to make sure both channels are in sync. In the sinusoid waveform menu, selct `Phase` and then `Sync Internal`, which just forces both channels to be in phase. Measure the output of the second AD633 multiplier PCB. For the local oscillator signal, use the function generator again, using the same carrier signal as before. Record the output and plot this in your lab report. Measure the frequency response of the output of the second AD633 PCB using the FFT function of the oscilloscope. Measure the FFT from 0 Hz to 30 kHz. You should see a peak at 1 kHz, which corresponds to the original message signal $m(t)$. You should see peaks at 18 kHz. That corresponds to the second harmonic of the carrier frequency. This is expected, as the demodulation process produces a second harmonic of the carrier frequency. The output of the second AD633 PCB is the demodulated signal $x_d(t)$.

We will use a low-pass filter to recover the original message signal $m(t)$ from the demodulated signal $x_d(t)$. Connect the output of the second AD633 PCB to the input of the low-pass filter. The output of the low-pass filter will be the recovered message signal $m(t)$. Measure the output of the low-pass filter using the oscilloscope. You should see a sine wave with a frequency of 1 kHz. Record this waveform and include it in your report.

Try a more complicated message signal $m(t)$. Use a triangle wave signal $m(t)$ with a frequency of 500 Hz and an amplitude of $5\,V_{pp}$. Set the carrier frequency to 9 kHz again. Repeat the modulation and demodulation process as before. Measure the output of the first AD633 PCB, the second AD633 PCB, and the low-pass filter. Record these waveforms and include them in your report. Measure the frequency response of the output of the second AD633 PCB and the low-pass filter using the FFT function of the oscilloscope.

Questions and tasks

- Include plots of the original message signal $m(t)$, modulated signal $x_c(t)$, the demodulated signal $x_d(t)$, and the recovered message signal in your report. Do this for the sine wave and triangle wave message signals.
- Include the frequency response of the sine message signal $M(\omega)$, modulated signal $X_c(\omega)$, the demodulated signal $X_d(\omega)$, and the recovered message signal $M_r(\omega)$ in your report. Do this just for the sine wave.

6.2.3 Coherent demodulation with a phase shift

In the previous section, we assumed that the local oscillator signal was perfectly synchronized with the carrier frequency. In practice, this is often not the case, and there may be a phase shift between the local oscillator and the carrier frequency. This can lead to distortion in the recovered message signal $m(t)$. In this section, we will explore how to handle this phase shift in the demodulation process.

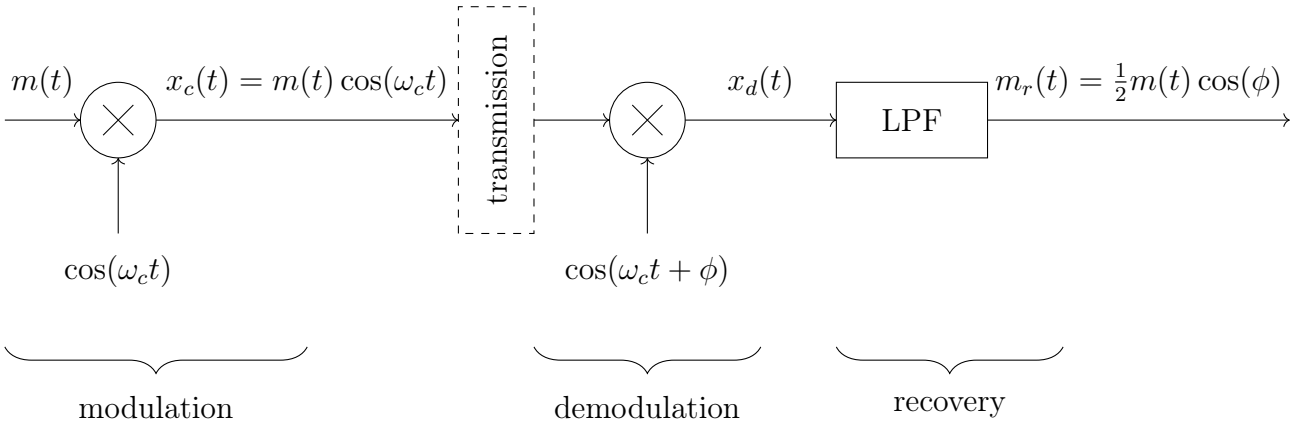


Figure 6.4: Block diagram of DSB-SC modulation and demodulation with a phase shift.

In class, we learned that the demodulated signal can be written as

$$\begin{aligned} x_d(t) &= x_c(t) \cdot \cos(\omega_c t + \phi) \\ &= m(t) \cdot \cos(\omega_c t) \cdot \cos(\omega_c t + \phi) \\ &= m(t) \cdot \left(\frac{1}{2}\cos(\phi) + \frac{1}{2}\cos(2\omega_c t + \phi) \right) \\ &= \frac{1}{2} m(t) \cdot \cos(\phi) + \frac{1}{2} m(t) \cdot \cos(2\omega_c t + \phi). \end{aligned}$$

This means that the recovered message signal $m(t)$ will be scaled by a factor of $\frac{1}{2}\cos(\phi)$, and there will be a second harmonic term $\frac{1}{2}m(t) \cdot \cos(2\omega_c t + \phi)$ that will be present in the demodulated signal. The second harmonic term can be filtered out using a low-pass filter, but the scaling factor will affect the amplitude of the recovered message signal. A block diagram of this system is shown in Fig. 6.4.

We want to observe this in hardware. We can use the same setup as before (using the 1 kHz sine wave as the input message signal $m(t)$), but we will need a second carrier signal that is phase-shifted by some angle ϕ. This phase shift will allow us to observe the effect of the phase shift on the recovered message signal $m(t)$. Adjust the waveform settings for the second channel of the function generator to introduce a phase shift ϕ between the two carrier signals. Adjust the phase to be 20°. After you have the phase set, push `Sync Internal` again to make sure both channels are synchronized. Measure the output of the second AD633 multiplier PCB. Record this waveform and include it in your report. Add the reconstructed message signal from Sec. 6.2.2 for comparison.

Questions and tasks

- Include the time-domain plot of the recovered message signal $m(t)$ in your report. Plot this waveform along with the recovered message signal from the previous section (no phase shift) for comparison. How are they different?

6.2.4 Non-coherent demodulation

In the previous sections, we assumed that the modulator and demodulator's carrier frequencies were perfectly synchronized. In practice, this is often not the case, and there may be a mismatch

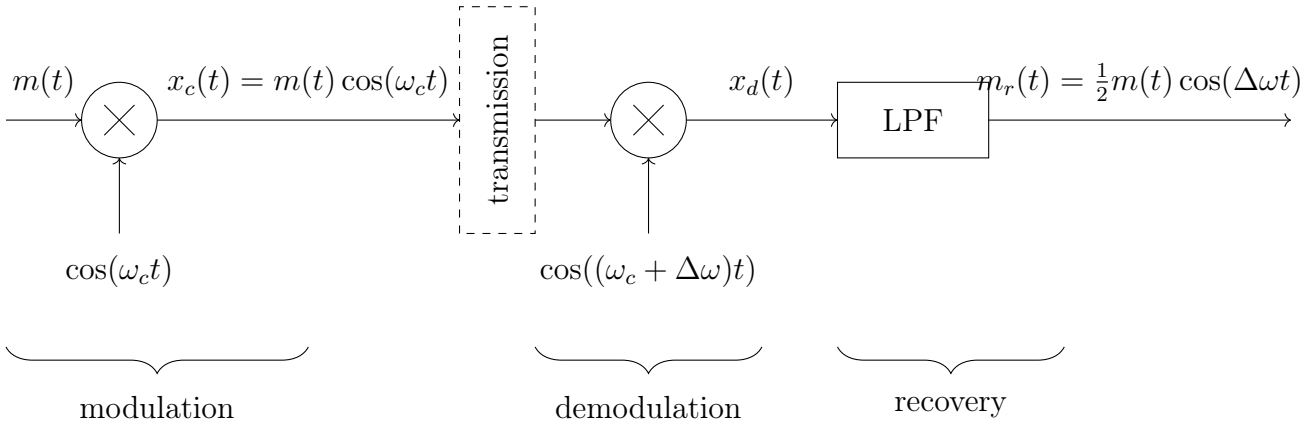


Figure 6.5: Block diagram of DSB-SC modulation and demodulation with a frequency mismatch.

between the modulator and demodulator oscillators. This can lead to distortion in the recovered message signal $m(t)$.

Consider the case where the receiving oscillator is not synchronized with the carrier frequency. In this case, the demodulated signal can be written as

$$\begin{aligned} x_d(t) &= x_c(t) \cdot \cos((\omega_c + \Delta\omega)t) \\ &= m(t) \cdot \cos(\omega_c t) \cdot \cos((\omega_c + \Delta\omega)t) \\ &= m(t) \cdot \left(\frac{1}{2}\cos(\Delta\omega t) + \frac{1}{2}\cos((2\omega_c + \Delta\omega)t)\right) \\ &= \frac{1}{2}m(t) \cdot \cos(\Delta\omega t) + \frac{1}{2}m(t) \cdot \cos((2\omega_c + \Delta\omega)t). \end{aligned}$$

We notice that after the demodulation process and filtering process, the message signal $m(t)$ is still modulated by some low-frequency cosine signal $\cos(\Delta\omega t)$. This means that the recovered message signal will be distorted, and the distortion will depend on the frequency mismatch $\Delta\omega$. This is a significant limitation of the coherent demodulation technique, as it requires precise synchronization between the modulator and demodulator oscillators. In this section, we will explore this effect in hardware. We will use the same setup as before, but we will introduce a frequency mismatch between the modulator and demodulator oscillators. A block diagram of this system is shown in Fig. 6.5.

For the message signal $m(t)$, we will use the function generator. For your message signal $m(t)$, generate a 5 Vpp sine wave with a frequency of 1 kHz. Generate this signal using the function generator coming out of the oscilloscope. Use the same settings as before for the first channel of the function generator (9 kHz, 20 V_{pp}) to generate the carrier signal. For the second carrier signal, on the second channel of the function generator, generate a sine wave with an amplitude of 10 V_{pp}, but this time set the frequency to 9.5 kHz to introduce a frequency offset of $\Delta\omega = 500$ Hz. Set the phase of this second carrier signal to be 0° for simplicity. Make sure you select `Sync Internal` in the `Phase` submenu to keep both channels synchronized.

Measure the frequency response of the output of the second AD633 multiplier using the FFT function of the oscilloscope. Measure the output of the reconstruction low-pass filter. Record the waveform and include it in your report. You should see a *very* distorted sine wave. Record this reconstructed message signal $m_r(t)$ waveform and include this plot it in your report.

Questions and tasks

- Measure the output Fourier transform of the output of the second AD633 multiplier as you set $\Delta\omega = 500\,\text{Hz}$
- Measure and plot the reconstructed message signal $m_r(t)$.
- Measure and plot the Fourier transform of the demodulated signal $X_d(\omega)$.

6.3 Deliverables

6.3.1 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.

6.3.2 Grading

The lab assignment will have the following grade breakdown:

- Low-pass filter frequency response plot: 10pts.
- Coherent demodulation sine wave time-domain plot (original $m(t)$, modulated $x_c(t)$, demodulated $x_d(t)$, and recovered message $m_r(t)$ signals): 20pts.
- Coherent demodulation sine wave frequency response plot (original $M(\omega)$, modulated $X_c(\omega)$, demodulated $X_d(\omega)$, and recovered message $M_r(\omega)$ signals): 20pts.
- Coherent demodulation triangle wave time-domain plot (original $m(t)$, modulated $x_c(t)$, demodulated $x_d(t)$, and recovered message $m_r(t)$ signals): 10pts.
- Phase shift recovered message signal time-domain plot $m_r(t)$ (also include the reconstructed signal from the coherent demodulation system): 10pts.
- Frequency mismatch recovered message $m_r(t)$ signal time-domain plot: 15pts.
- Frequency mismatch demodulated signal $X_d(\omega)$ Fourier transform plot: 15pts.

Chapter 7

Sampling and Aliasing

Sampling is one of the most fundamental concepts in digital signal processing and forms the bridge between the analog and digital worlds. In this laboratory exercise, you will explore the practical implementation of sampling theory, investigating how continuous-time signals can be accurately converted to discrete-time representations and subsequently reconstructed without loss of information.

The Nyquist sampling theorem states that a bandlimited continuous-time signal can be perfectly reconstructed from its samples if the sampling frequency is at least twice the highest frequency component present in the signal. This critical frequency, known as the Nyquist frequency, establishes the minimum sampling rate required to avoid aliasing——a phenomenon where high-frequency components are incorrectly represented as lower-frequency components in the sampled signal.

In this lab, you will implement a practical sampling system using electronic hardware. The sampling process will be realized by multiplying an analog voltage signal with a periodic square wave train having a short duty cycle. This multiplication effectively creates a series of narrow pulses whose amplitudes correspond to the instantaneous values of the input signal at the sampling instants. The resulting sampled signal will then be processed through a low-pass filter designed with a cutoff frequency at the Nyquist frequency to reconstruct the original continuous-time signal.

Through this hands-on approach, you will gain insight into the critical design considerations for sampling systems, including the trade-offs between sampling rate, filter design, and signal fidelity. You will also observe firsthand the effects of violating the sampling theorem and witness how aliasing can corrupt signal information when insufficient sampling rates are employed.

7.1 Background

To understand the sampling process mathematically, we must examine what occurs in both the time and frequency domains when a continuous-time signal is sampled. Consider a continuous-time signal $x(t)$ that we wish to sample at regular intervals T_s (the sampling period). The sampling process can be mathematically modeled as the multiplication of the continuous signal with an impulse train:

$$x_s(t) = x(t) \cdot \sum_{n=-\infty}^{\infty} \delta(t - nT_s). \tag{7.1}$$

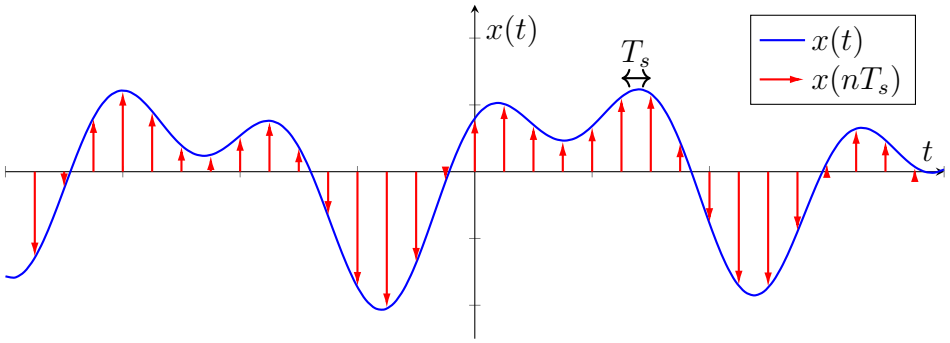


Figure 7.1: Sampling of a continuous-time signal $x(t)$ at intervals T_s.

This multiplication results in a sampled signal $x_s(t)$ that consists of weighted impulses occurring at the sampling instants $t = nT_s$, where the weight of each impulse equals the value of the original signal at that instant

$$x_s(t) = x(nT_s) = \sum_{n=-\infty}^{\infty} x(nT_s)\delta(t - nT_s).$$

A representation of this process in the time domain is shown in Figure 7.1.

In the frequency domain, the effects of sampling become even more apparent. The Fourier transform of the impulse train used for sampling is itself an impulse train in the frequency domain with period $\omega_s = 2\pi/T_s$ (the sampling frequency in rad/s)

$$\mathcal{F}\left\{\sum_{n=-\infty}^{\infty} \delta(t - nT_s)\right\} = \frac{2\pi}{T_s}\sum_{k=-\infty}^{\infty} \delta(\omega - k\omega_s).$$

Since multiplication in the time domain corresponds to convolution in the frequency domain, the spectrum of the sampled signal $X_s(\omega)$ is given by the convolution of the original signal's spectrum $X(\omega)$ with the frequency-domain impulse train. This convolution operation results in

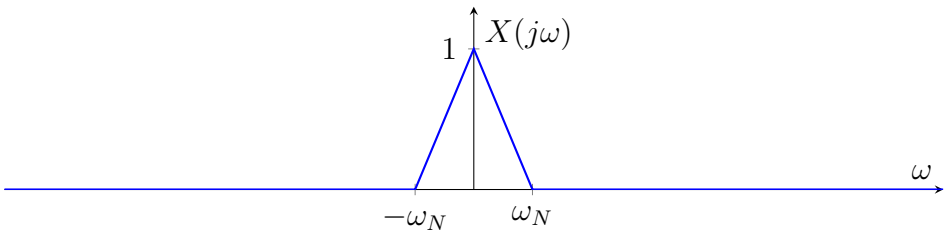
$$X_s(\omega) = \frac{1}{2\pi}X(\omega) * \frac{2\pi}{T_s}\sum_{k=-\infty}^{\infty} \delta(\omega - k\omega_s) \tag{7.2}$$

$$= \frac{1}{T_s}\sum_{k=-\infty}^{\infty} X(\omega - k\omega_s). \tag{7.3}$$

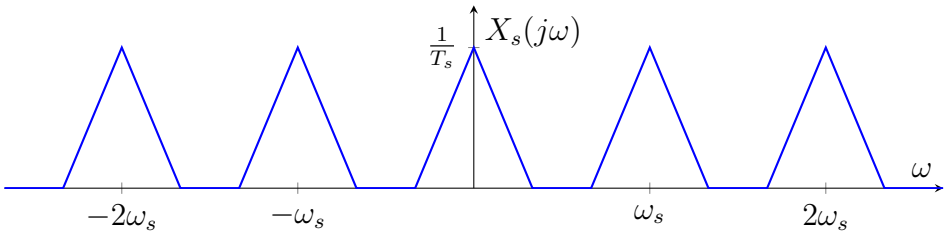
Suppose that the original signal $x(t)$ has a Fourier transform $X(j\omega)$, which is

This is depicted in Fig. 7.2 where—without the loss of generality—the some original, bandlimited signal's spectrum $X(\omega)$ is shown in Figure 7.2a and the spectrum of the sampled signal $X_s(\omega)$ is shown in Figure 7.2b. The sampling frequency ω_s is defined as $\omega_s = 2\pi/T_s$, where T_s is the sampling period.

Eq. 7.3 shows that sampling causes the original spectrum to be replicated (with scaling factor $1/T_s$) at integer multiples of the sampling frequency. If the original signal $x(t)$ is bandlimited



(a) Fourier transform of the original signal $x(t)$, $X(\omega)$.



(b) Fourier transform of the sampled signal $x_s(t)$, $X_s(\omega)$, showing spectral replicas.

Figure 7.2: Frequency domain representation of the impacts of sampling in the time domain.

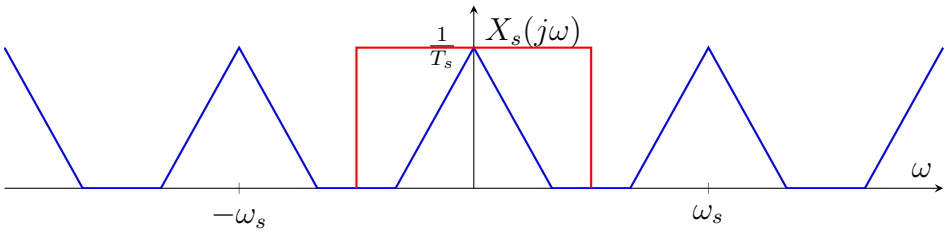


Figure 7.3: Reconstruction of the continuous-time signal $x(t)$ from its samples $x(nT_s)$ using a low-pass filter.

from frequency $-\omega_N$ to ω_N, then perfect reconstruction is possible provided that $\omega_s > 2\omega_N$ (the sampling theorem condition). However, if this condition is violated, the replicated spectra will overlap, causing aliasing where high-frequency components are indistinguishable from low-frequency components in the sampled signal.

To recover the original signal, we can use a low-pass filter (LPF) with cutoff frequency ω_N to remove the unwanted high-frequency replicas. This process, known as reconstruction or interpolation, aims to estimate the continuous-time signal $x(t)$ from its samples $x(nT_s)$ by filling in the gaps between the samples. This is seen in the frequency domain as Fig. 7.3.

In this lab, we are going to observe these processes in hardware by implementing a quasi-sampling system using a square wave train to sample an analog signal. The sampled signal will then be passed through a low-pass filter to attempt to reconstruct the original signal. This will allow us to see the effects of sampling and reconstruction in practice, as well as the potential pitfalls of aliasing if the sampling rate is not chosen appropriately.

7.2 Building the circuit

7.2.1 Sampling in hardware

To implement sampling in hardware, we will use the AD633 multiplier PCBs we used in Chapter 6. Power these PCBs to $\pm 17\,\text{V}$ using your power supply, do not forget the ground connection. Generate a $1\,\text{V}_{\text{pp}}$ triangle wave at 500 Hz using function generator to represent $x_s(t)$. Generate a square wave at 10 kHz with a 5% duty cycle that is $10\,\text{V}_{\text{pp}}$ and a 5 V offset to represent the sampling signal. Make sure you have your function generator to `High Z` output mode. This will need to be connected to the multiplier PCB as the carrier input. The multiplier will then output a sampled signal that is a series of narrow pulses whose amplitudes correspond to the instantaneous values of the input signal at the sampling instants.

Use an oscilloscope to measure your input signal $x(t)$ and the output of the multiplier $x_s(t)$. You should see that the output is a series of narrow pulses that correspond to the input signal at the sampling instants. The width of these pulses will depend on the duty cycle of the square wave used for sampling.

7.2.2 Signal conditioning

We will use a low-pass filter to attempt to reconstruct the original signal from the sampled signal. To do this, we are going to need a very sharp low-pass filter. To simplify the lab, we are going to use an IC that implemenets an 8^{th} order elliptic switched capacitor low-pass filter. Before we use this IC chip, the signal needs to range from 0.25 V to 5 V, so we are going to need to bias the signal between these two levels.

To do this, add an op-amp buffer with a gain of 1 (i.e., voltage follower) to the output of the multiplier. This will allow us to provide proper impedance isolation between the multiplier and subsequent filtering stages, providing a high input impedance to the multiplier output, and preventing loading effects that could distort the sampled pulses. Additionally, the buffer provides a low output impedance that can effectively drive the switched capacitor filter, ensuring that the delicate timing and amplitude characteristics of the sampled signal are preserved through the signal chain. The op-amp should be powered by 15 V and $-15\,\text{V}$ supplies.

From the voltage follower, we will add a capacitor and then a voltage divider between 5 V and ground to bias the signal to the desired range. The capacitor will be used to block any DC offset from the multiplier output, allowing only the AC component of the sampled signal to pass through. The voltage divider will then set the DC level of the signal to be between 0 V and 5 V, as desired. We can get the 5 V from the 5 V pin on the PCB. This entire stage is shown in Fig. 7.4. You will notice that this biasing will effectively act as a high-pass filter, removing any offset from the sampled signal and allowing only the AC component to pass through. In practice, you might want to preserve the DC level of the sampled signal, but for this lab, we are only interested in the AC component.

Once you have this stage built, connect the output of the AD633 multiplier to the input of the biasing stage. You should see that the output of the biasing stage is a signal that is between the desired 0.25 V and 5 V range.

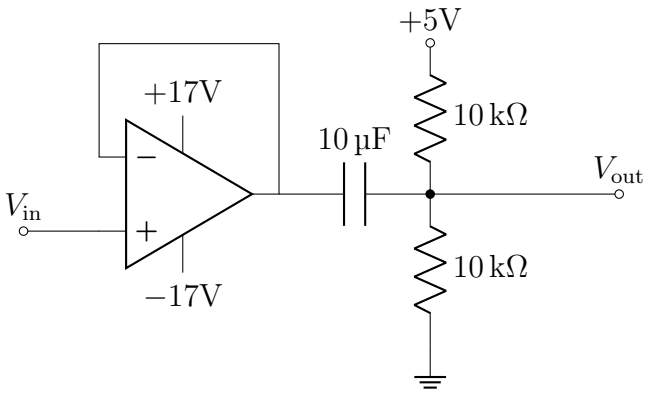


Figure 7.4: Biasing and filtering stage for the sampled signal.

7.2.3 Low-pass filtering

Next, we will connect the output of the biasing stage to the input of the low-pass filter. This filter will help to smooth out the sampled signal and remove any high-frequency noise that may be present. The output of the low-pass filter will be the reconstructed signal $x_r(t)$. This is a switched-capacitor filter, which uses capacitors and electronic switches (typically MOSFET transistors) to emulate the behavior of resistors and inductors, allowing for precise filter characteristics that can be controlled by an external clock frequency. This type of filter is particularly well-suited for our application since it can be easily tuned while minimizing component count. To tune this filter, we will add a clock signal to the filter. This clock signal is a square wave ranging from 0 V to 5 V with a duty cycle of 50%. The cutoff frequency f_c is tuned by adjusting the clock frequency at a 1:100 ratio. In other words, if we want a 7 kHz cutoff frequency, we would set the clock frequency to 700 kHz. We are sampling our signal at 10 kHz, so we will want to adjust f_c to the Nyquist frequency. Choose the appropriate frequency for our clock. We should verify we have wired everything correctly. Run a frequency sweep through our low-pass filter using the oscilloscope. Does this match the desired specifications for this sampling scheme? Record this frequency response and include it in your lab report.

Because the MAX 7403 is a single-supply chip, the output of the low-pass filter will be biased to 2.5 V. We will need to remove this bias with a simple RC high-pass filter. Additionally, because sampling reduces the amplitude of the signal by a factor of $\frac{1}{T_s}$, we should also add a gain stage to amplify the signal back to something that is more visible on the oscilloscope. This can be done with a simple op-amp amplifier. You can choose the gain factor. The output of this stage will be the final reconstructed signal $x_r(t)$. An example of this output is seen in Fig. 7.5.

Questions and tasks

- What frequency did you choose for the clock signal? Why did you choose this frequency?

7.3 Measurements

7.3.1 Simple reconstruction

Using the 1 V_{pp}, 500 Hz triangle wave as the input signal, we can now observe the effects of sampling and reconstruction. Connect the output of the inverting amplifier to an oscilloscope and observe the

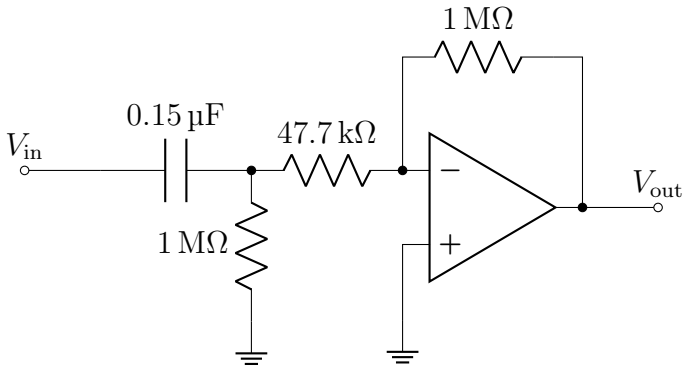


Figure 7.5: Final reconstruction stage for the sampled signal.

reconstructed signal $x_r(t)$. You should see a smooth waveform that closely resembles the original triangle wave, albeit with some distortion due to the sampling process and the characteristics of the low-pass filter.

Measure the original signal $x(t)$, the sampled signal $x_s(t)$, and the reconstructed signal $x_r(t)$. Plot all three signals in the time domain. Similarly, measure each of the three signals in the frequency domain. Measure the FFT using a `Span` of 50 kHz, a `Center` of 25 kHz, Vrms, and a time resolution of 5 ms per division. Plot all three FFTs on the same plot. Notice that the Fourier transform of $x(t)$ is repeated at integer multiples of f_s with $X_s(j\omega)$, as depicted in Fig. 7.2b.

This approach does have a limitation with this sampling approach in that the $x(t)$ of a triangle function is not bandlimited (very few signals are in practice). In a perfect world, we would first filter this signal using an anti-aliasing filter, which is a low-pass filter that helps suppress the signal beyond the Nyquist frequency. Here, we choose a triangle function because its Fourier transform is roughly a $\text{sinc}^2()$, which rolls off by a factor of $\frac{1}{\omega^2}$. By comparison, a square wave would have more issues because its Fourier transform is a simple sinc() which rolls off by a factor of $\frac{1}{\omega}$.

Questions and tasks

- Create a plot in the time-domain of $x(t)$, $x_s(t)$, and $x_r(t)$. Plot on the same axes as long as the vertical scale is roughly the same. If not, use subplots.
- Create a plot a frequency-domain of $X(j\omega)$, $X_s(j\omega)$, and $X_r(j\omega)$. The vertical scaling is likely going to be different, so plot these on separate axes, but you can consolidate using a subplot.

7.3.2 Square wave sampling

Eq. 7.1 models as the sampling process as a multiplication of the original signal and a train of delta functions. In practice, the delta function is not realizable as it is a theoretical construct. Instead, we use a narrow pulse train to approximate the sampling process. The pulse train can be generated using a square wave oscillator, which produces a series of rectangular pulses at the sampling frequency. We can describe these rectangular pulses as $p_{T_s}(t)$ where T_s is the period of the square wave with a fundamental frequency of $\frac{2\pi}{T_s}$. The Fourier series representation for this pulse is

$$p_{T_s}(t) = C_0 + \sum_{n=1}^{\infty} C_n \cos\left(\frac{2\pi n}{T_s}t + \theta_n\right)$$

The sampled signal can then be represented as $x_s(t) = x(t) \cdot p_{T_s}(t)$ which is

$$\begin{aligned} x_s(t) &= x(t) \cdot p_{T_s}(t) \\ &= x(t) \cdot \left(C_0 + \sum_{n=1}^{\infty} C_n \cos\left(\frac{2\pi n}{T_s}t + \theta_n\right)\right) \\ &= C_0 x(t) + \sum_{n=1}^{\infty} C_n x(t) \cos\left(\frac{2\pi n}{T_s}t + \theta_n\right) \end{aligned}$$

Based on the principles of modulation (see Chapter 6), we can see that this is equivalent to modulating the original signal $x(t)$ with train of cosine functions of increasing frequencies. This modulation process effectively shifts the frequency content of the original signal to higher frequencies $n\omega_s$, which is similar to what we see in Eq. 7.3. The key difference is that the additional copies of the original signal spectrum are not just shifted, but also scaled by the coefficients C_n and $x(t)$. If you were to work out the coefficients in C_n, you would find that they depend on the duty cycle of the square wave used for sampling.

Adjust the duty cycle of the square wave used for sampling to see how it affects the sampled signal. Collect $x_s(t)$, the FFT of $x_s(t)$, and the reconstructed signal $x_r(t)$ for duty cycles as 5%, 20%, and 50%. Plot $x_s(t)$, $X_s(j\omega)$, and $x_r(t)$ for each duty cycle. Notice the impact on the peaks of the additional copies of $X(j\omega)$ that show up in the frequency domain. How much does the peak amplitude of the first harmonic replica drop relative to the central replica peak for each duty cycle? Collect this data in a table.

Questions and tasks

- Plot (on the same axes) $x_s(t)$ in the time domain for each duty cycle tested.
- Plot (on the same axes) $X_s(j\omega)$ in the frequency domain for each duty cycle tested.
- Plot $x_r(t)$ for the 50% duty cycle.
- Collect the peak amplitudes of the first harmonic replica relative to the central replica peak for each duty cycle. Report this as a table in your lab report.

7.3.3 Aliasing

Change the input signal $x(t)$ to be a $1\,V_{pp}$ sine wave with a frequency of 1 kHz. Record $x(t)$, $x_s(t)$, and $x_r(t)$ in the time domain. Also record the FFT of $x_s(t)$. Increase the frequency of the sine wave to 2 kHz and repeat the recording process. Record $x(t)$, $x_s(t)$, and $x_r(t)$ in the time domain. Also record the FFT of $x_s(t)$. Keep adjusting the frequency by 1 kHz increments until you reach 10 kHz. What do you observe? What happens to $x_r(t)$ as the frequency of $x(t)$ goes beyond 5 kHz? How does this relate to the Nyquist sampling theorem? What happens to the FFT of $x_s(t)$ as the frequency of $x(t)$ goes beyond 5 kHz? How does this relate to aliasing?

For each frequency, plot $x(t)$ and $x_r(t)$ in the time domain. Plot $X_s(j\omega)$ in the frequency domain for each frequency. I would strongly recommend plotting the various waveforms on the same time and frequency axes. What patterns do you see?

- Plot $x(t)$ and $x_r(t)$ in the time domain for each frequency.
- Plot $X_s(j\omega)$ in the frequency domain for each frequency.

7.4 Deliverables

7.4.1 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.

7.4.2 Grading

The lab assignment will have the following grade breakdown:

- Report document (format, clarity, conclusions): 15pts.
- Time-domain plots of $x(t)$, $x_s(t)$, $x_r(t)$ for the triangle waveform (feel free to plot on the same plot as long as the scaling is roughly the same): 15pts.
- Frequency-domain plots (FFT) of $X(\omega)$, $X_s(\omega)$, $X_r(\omega)$ for the triangle waveform using correct FFT settings (Span=50 kHz, Center=25 kHz, Vrms, 5 ms/div) (would recommend using subplots to simplify): 15pts.
- Duty-cycle study (5%, 20%, 50%): 25pts total.
  - Time-domain plots of $x_s(t)$ for each duty cycle on the same axes: 5pts.
  - Frequency-domain plots $X_s(\omega)$ for each duty cycle (recommend using subplots to simplify): 5pts.
  - Reconstructed signal $x_r(t)$ for the 50% duty cycle: 5pts.
  - Table reporting peak amplitude of first harmonic replica relative to central replica for each duty cycle (i.e., 1st harmonic amplitude / central amplitude): 10pts.
- Aliasing sweep (1 kHz to 10 kHz in 1 kHz steps): 30pts total.
  - Time-domain plots of $x(t)$ and $x_r(t)$ for each tested frequency (or representative subset with commentary) on the same plot: 15pts.
  - Frequency-domain plots $X_s(\omega)$ for each tested frequency (or representative subset) on the same plot: 15pts.

Chapter 8

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.,

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

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

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

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.

8.1 Getting started

First, clone the repository from 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 `.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:

```
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`.

8.2 Building the database

8.2.1 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.

8.2.2 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. 8.1.

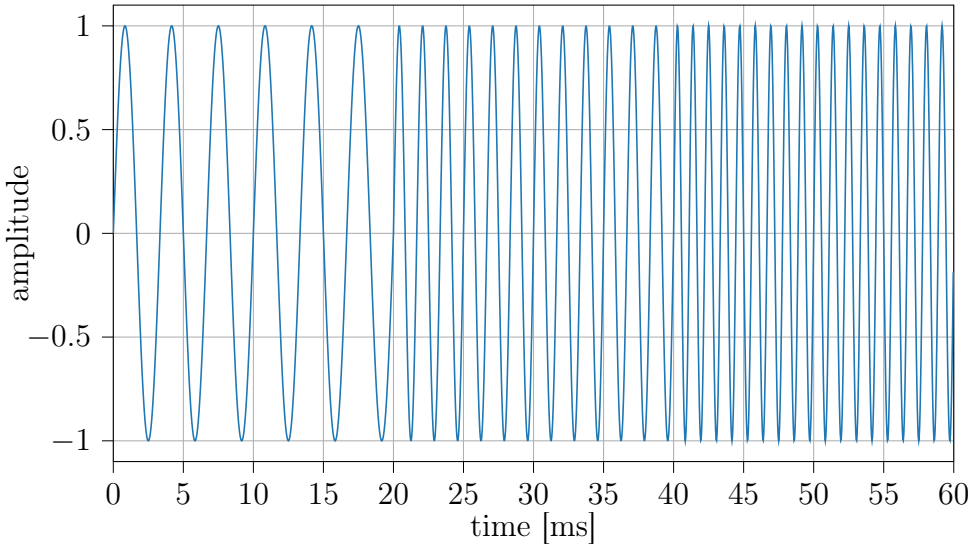


Figure 8.1: A signal that is a sine wave where the frequency changes over time.

We can compute the spectrogram and display its magnitude as a 2D image. The spectrogram of the signal in Fig. 8.1 is shown in Fig. 8.2. 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. 8.3. 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. 8.4.

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

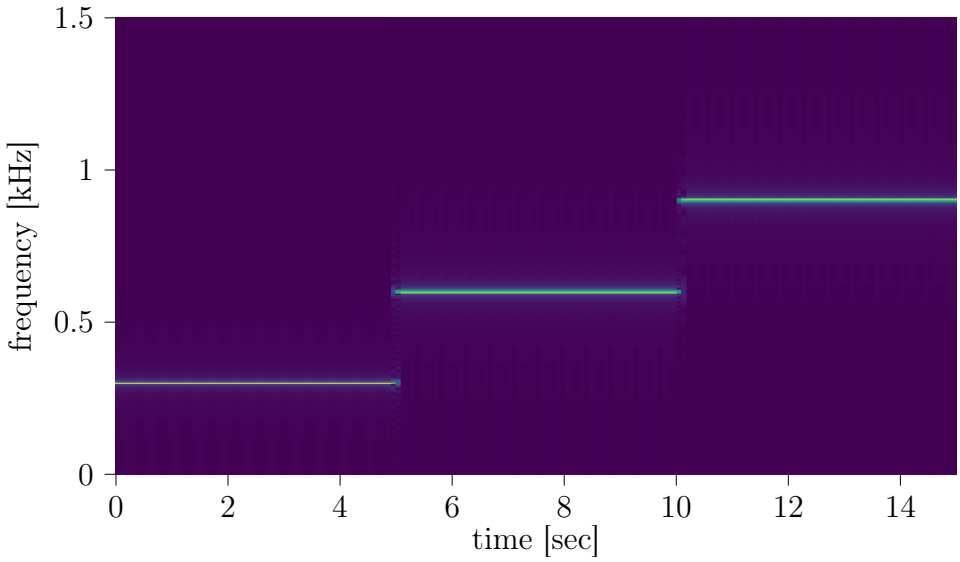


Figure 8.2: The spectrogram of the signal in Fig. 8.1.

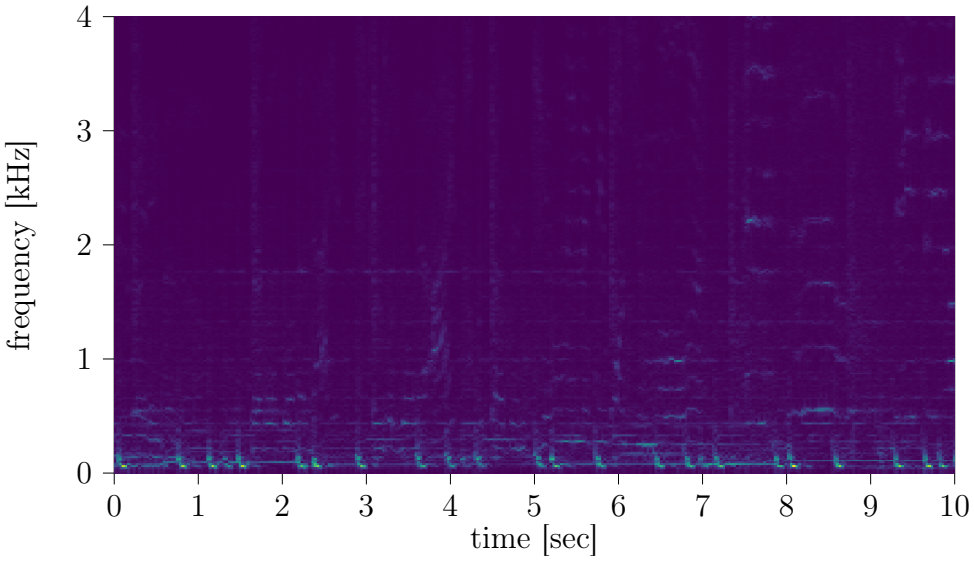


Figure 8.3: The spectrogram of a clip from the song "Cruel Summer".

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:

```
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
```

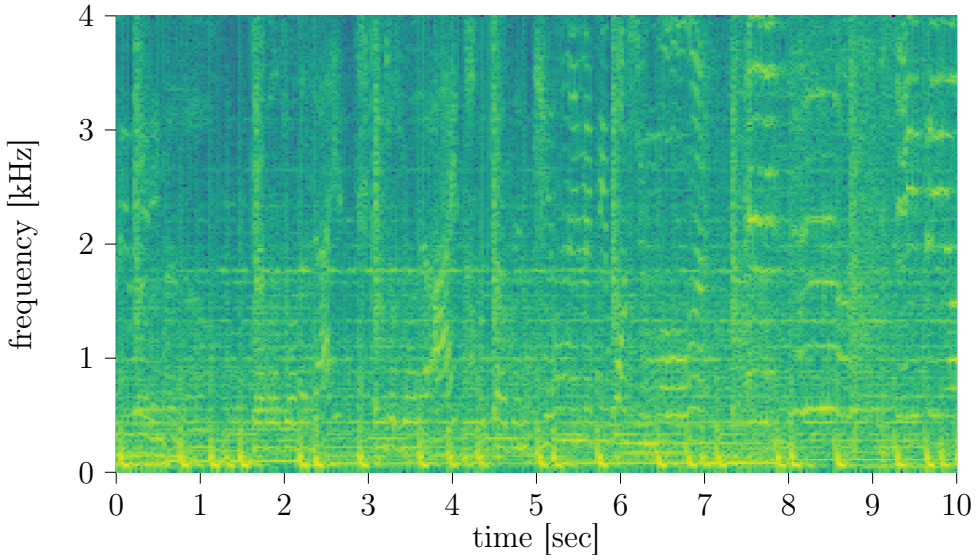


Figure 8.4: The logarithm of the spectrogram of a clip from the song "Cruel Summer".

```
        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. 8.3, 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 compute 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.

8.2.3 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. 8.4, 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. 8.5.

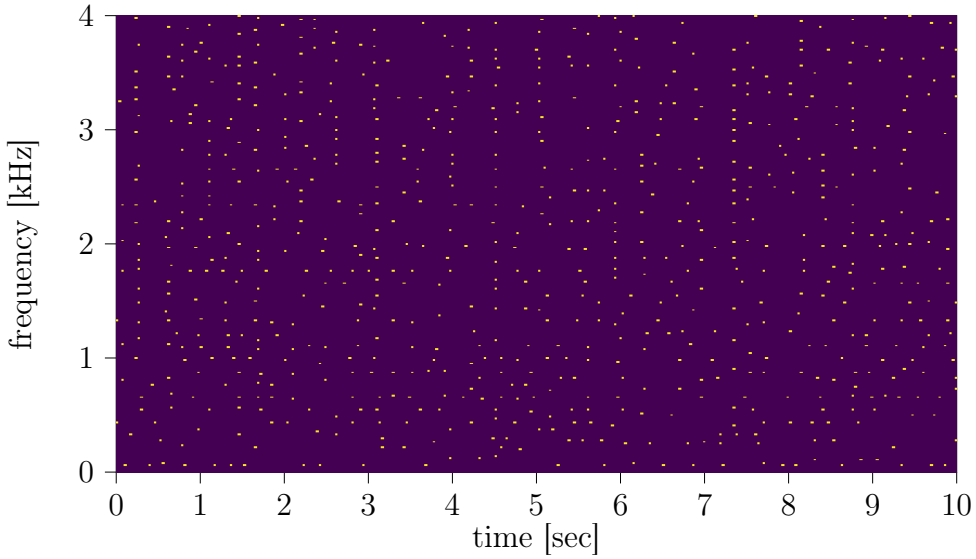


Figure 8.5: The peaks in the spectrogram of a clip from the song "Cruel Summer".

8.2.4 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. 8.6.

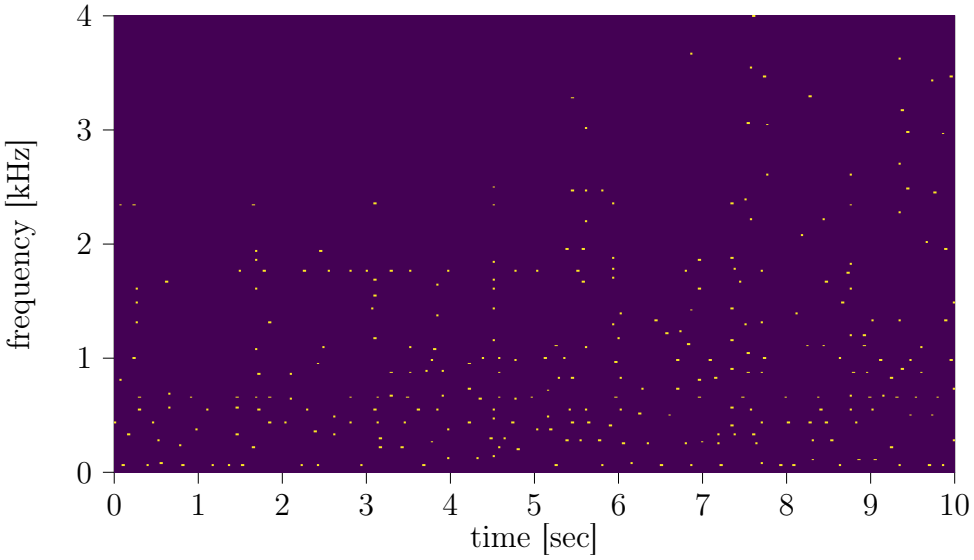


Figure 8.6: The peaks in the spectrogram of a clip from the song "Cruel Summer" after thresholding at 30 peaks per second.

8.2.5 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. 8.7.

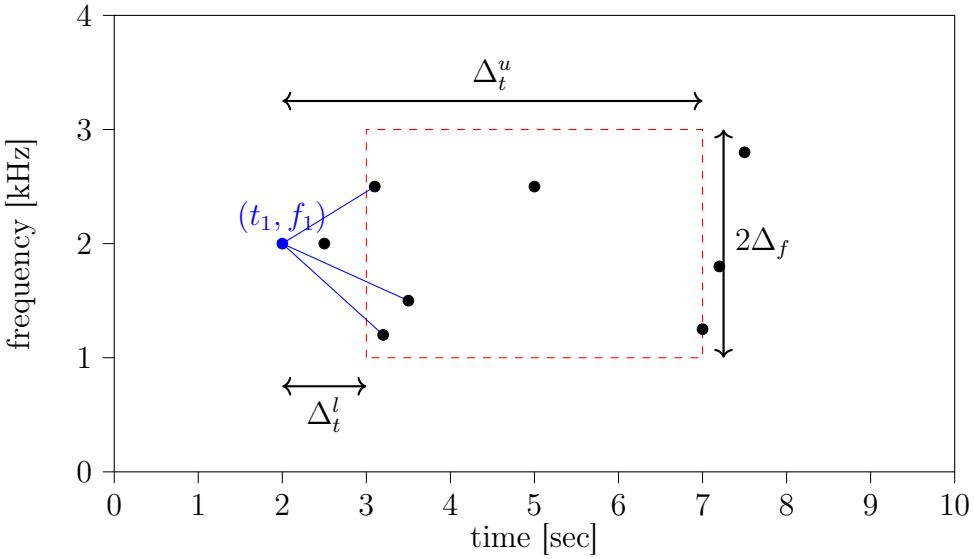


Figure 8.7: An example of the fan-out for a peak.

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:

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

8.2.6 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

$$h(f_1, f_2, t_2 - t_1) = (t_2 - t_1) \cdot 2^{16} + f_1 \cdot 2^8 + f_2. \tag{8.1}$$

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 is 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
```

8.3 Song identification

8.3.1 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.

8.3.2 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.

8.3.3 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
```

8.4 Testing

You will be provided with 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_shazam.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: california_dreamin.wav
Test passed for: golden.wav
Test passed for: i_can_change.wav
Test passed for: burning_down_the_house.wav
Test passed for: my_girls.wav
Test passed for: blister_in_the_sun.wav
Test passed for: changes.wav
Test passed for: dont_stop_me_now.wav
Test passed for: this_charming_man.wav
Test passed for: cant_hold_us.wav
Test passed for: last_nite.wav
Test passed for: islands.wav
Test passed for: we_only_come_out_at_night.wav
Test passed for: zombie.wav
.
```

```
Testing --help output
.
Testing unknown argument handling
.
----------------------------------------------------------------------
Ran 4 tests in 15.934s

OK

Total points: 150
```

Other than that, you are free to design the software as you see fit. You will also be tested on whether your code can identify songs using a microphone[1].

8.5 Deliverables

8.5.1 Submission

Please submit your code in a single file called `shazam.py`. Please do not commit any song files or `.h5` files to your repository.

8.5.2 Grading

The lab assignment will have the following grade breakdown.

- `compute_spectrogram()`: 10pts.
- `.wav` file accuracy (5pts. each): up to 65pts.
- Microphone accuracy (10pts. each): up to 150pts.

[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 9

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.

9.1 Procedure

9.1.1 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. 9.2. With 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

$$H(s) = \frac{s^2 + \omega_0^2}{s^2 + (2\omega_0 \cos(\theta))s + \omega_0^2} \tag{9.1}$$

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

$$H(s) = \frac{s^2 + \omega_0^2}{s^2 + \frac{\omega_0}{Q}s + \omega_0^2} \tag{9.2}$$

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.

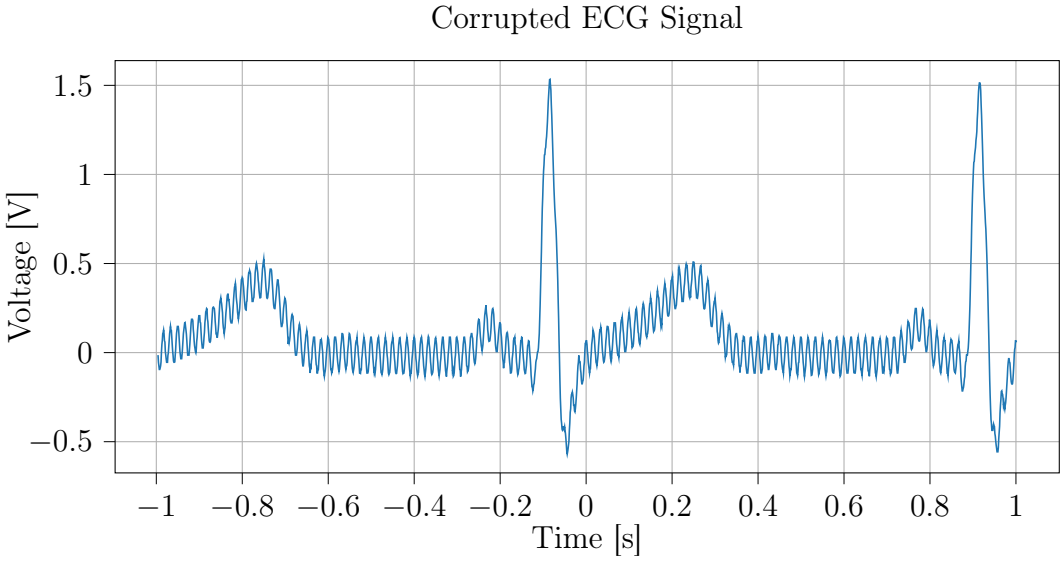


Figure 9.1: ECG signal with 60 Hz interference.

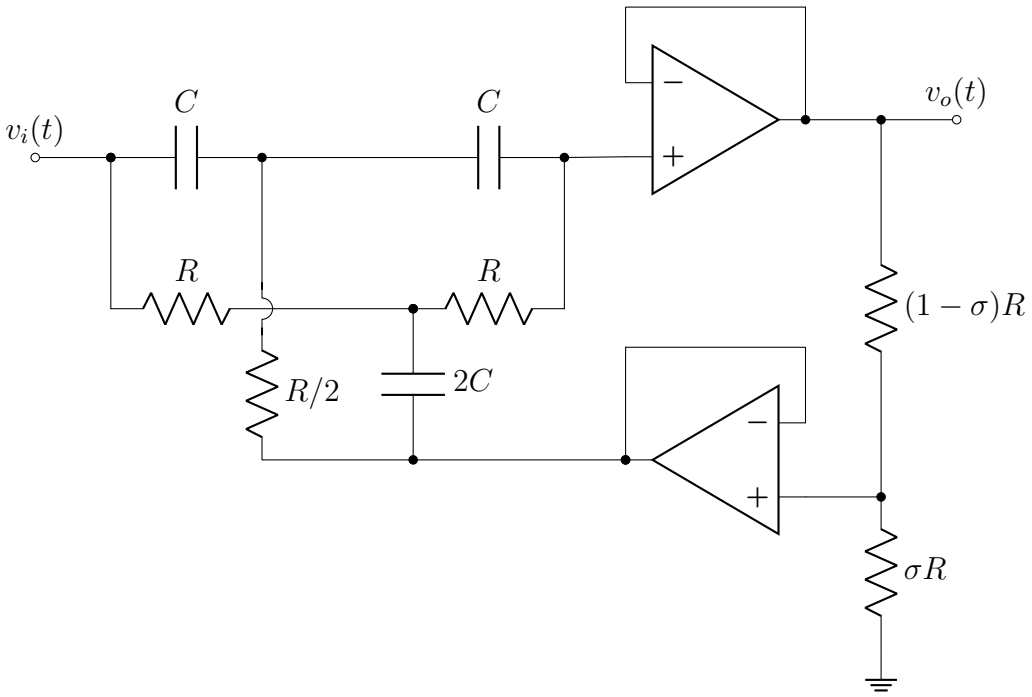


Figure 9.2: Notch filter circuit.

You will need to derive the transfer function for the circuit in Fig. 9.2. The derivation of this transfer function is very complicated. To make it easier, I would strongly recommend using 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 with 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

$$H(s) = \frac{V_o(s)}{V_i(s)}. \tag{9.3}$$

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:

```
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 the `notch_filter.ipynb` notebook, 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 test in the notebook.

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).

9.1.2 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.

9.2 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

- You do not need to include this plot. But it is a good exercise and a good way to debug your design.

9.3 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 $f \in [10\,\text{Hz}, 1000\,\text{Hz}]$. 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 $450\,\text{S}\,\text{s}^{-1}$. 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?

9.4 Deliverables

9.4.1 Submission

Fill out the notebook and upload it to GitHub. Make sure you include any CSV files that you used to plot the frequency response and ECG data. Make sure that your plots are properly labeled and include units on the axes.

9.4.2 Grading

The lab assignment will have the following grade breakdown:

- Correct transfer function `H(s)`: 25pts.
- Theoretical frequency response plot: 15pts.
- s-plane plot: 15pts.

- Measured frequency response: 20pts.
- ECG plot: 20pts.
- Additional questions: 5pts.

Chapter 10

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, which is 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 practice your circuit analysis strengthen your circuit analysis skills and develop practical skills in designing and analyzing filters for various applications such as audio processing, data acquisition, and communication systems.

10.1 Background

The Sallen–Key circuit (using an LM353 or LM124 etc. op-amp) seen in Fig. 10.1 can implement a 2$^{\text{nd}}$ order low pass filter. The transfer function is often given in a reduced form

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

10.2 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 the ratio of the actual damping to the critical damping. An under-damped system has $\zeta < 1$, an over-damped system has $\zeta > 1$, and a critically damped system has $\zeta = 1$.

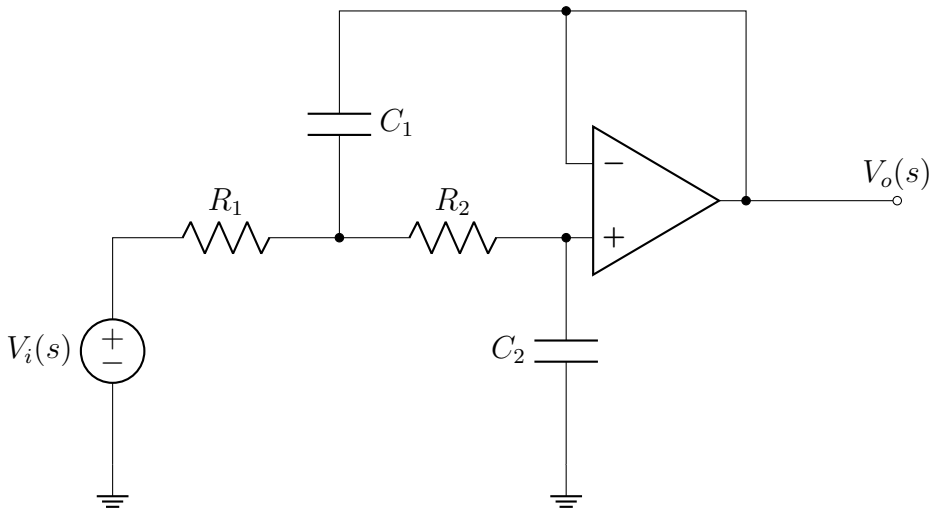


Figure 10.1: Sallen-Key 2nd order low pass filter.

Derive the transfer function $H(s)$ in terms of R_1, R_2, C_1, and C_2. In the notebook `sallen_key.ipynb`, implement your derived transfer function as a function 'H(s, R1, R2, C1, C2)', which has the function definition:

```
def H(s, R1, R2, C1, C2):
    """Derive transfer function for 2nd order filter circuit

    Parameters
    ----------
    s : complex or ndarray
        Complex frequency variable.
    R1 : float
        Resistance R1 in ohms.
    R2 : float
        Resistance R2 in ohms.
    C1 : float
        Capacitance C1 in farads.
    C2 : float
        Capacitance C2 in farads.

    Returns
    -------
    complex or ndarray
        The transfer function H(s).
    """
    # Your implementation here
```

Choose resistors and capacitors appropriately. Determine the poles and zeros of the transfer function and plot them on the s-plane. Plot the ideal frequency response of the filter from 0 Hz to 100 kHz. Plot the magnitude response in decibels and the phase in degrees.

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. Save the output as some form of text file (.csv, .txt, etc.) that you

can read into Python. Plot the simulated frequency response overlaid on the theoretical response you derived earlier. Make sure to plot both the magnitude and phase response.

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 $f \in [10\,\text{Hz}, 100\,\text{kHz}]$. Record this data from the oscilloscope and plot it overlaid against the derived frequency response and the SPICE simulation. Make sure to plot both the magnitude and phase response. Do these plots agree?

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, the SPICE simulation, and 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 theoretical, SPICE, and measured data should be overlaid on the same plot.

10.3 Deliverables

10.3.1 Submission

Fill out the provided Jupyter notebook `sallen_key.ipynb`. Make sure you also include the recorded data files from the oscilloscope and SPICE simulation. Submit your work via GitHub.

10.3.2 Grading

The lab assignment will have the following grade breakdown:

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

Chapter 11

Filter design

In the Shazam lab, we sample 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 a signal.

The design strategy for this filter is largely up to you. You 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 12 kHz to ∞.
- The passband gain is -2 dB (the passband gain needs to be between ± 2 dB).
- The minimum attenuation in the stop band is -20 dB.

11.1 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. Furthermore, 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. Make sure when designing the transfer function, you are designing it as an analog filter, not a digital filter. Plot the poles and zeros on the complex plane. Plot the theoretical frequency response of your designed filter from 0 Hz to 20 kHz. Plot the magnitude response in decibels and the phase in degrees.

Once you have a suitable transfer function and have verified that it matches or exceeds the specifications, start designing an appropriate circuit implementation. When designing your circuit, consider cascading multiple filter stages to achieve the desired filter order. Second-order Sallen-Key filter stages are a popular choice for implementing higher-order filters. The `scipy.signal` modules can return the transfer coefficients in this format.

- What is the transfer function $H(s)$ of your filter?
- Plot the poles and zeros of the transfer function on the s-plane.
- Include a sketch of your circuit. Explain what type of filter you designed and why you chose that design.

11.2 Procedure

Verify your design using your preferred SPICE simulator. Plot the simulated frequency response overlaid on the theoretical response you derived earlier. Make sure to plot both the magnitude and phase response.

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 work well).

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 $H(s)$ and SPICE simulation that 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 theoretical frequency response, the SPICE simulation, and 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 theoretical, SPICE, and measured data should be overlaid on the same plot.
- Does your hardware implementation match the theoretical and SPICE plots?
- You will be graded on whether your filter meets the specifications listed earlier. Please save your measured data as arrays according to the instructions in the notebook.

11.3 Deliverables

11.3.1 Submission

Fill out the Jupyter notebook `lpf.ipynb`. Your code and data files should be submitted to GitHub.

11.3.2 Grading

The lab assignment will have the following grade breakdown.

- Transfer function: 10pts.
- Plots of poles on complex plane: 10pts.
- Circuit diagram: 20 pts.

- Frequency response plots: 40pts.
- Measured data hitting specifications: 20pts.

Bibliography

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