University of
South-Eastern Norway

Introduction to Complex Exponentials
Direction Finding

TSE2280 Signal Processing

Lab 1, Spring 2025

1 Introduction

1.1 Background and Motivation

The lab demonstrates concepts from Chapters 2 and 3 in the course textbook *DSP First* [1], by illustrating how sinusoidal signals are described with complex amplitudes called *phasors*.

The first part of the lab gives training in sinusoidal signals and complex exponentials. The last part applies this to find the direction to a sound source from the phase difference at two microphones. This principle is the basis for steering and focusing antennas without moving them, such as *phased arrays* used in radar, wifi, and 5G, and sonar and medical ultrasound transducers.

The lab is based on *Lab P-3: Introduction to Complex Exponentials – Direction Finding* [?] that accompanies the textbook. The original lab has been converted from Matlab to Python and some contents has been modified.

1.2 Software Tools: Python with Spyder and JupyterLab

The programs are written in Python. The module NumPy [2, 3] is used to represent signals and Matplotlib [4, 5] to plot graphs. Including Python's cmath and math modules can make the code easier to read, but be aware that cmath and math only handle scalars. Note also that cmath always returns a complex number, even when the imaginary part is zero. The recommended setup for importing the modules is shown in Table 1.

The recommended Python programming environment is *Spyder* [6], which is included in the *Anaconda* [7] package management. However, any Python package management and programming environment can be used.

All code files shall be included with the lab report. We recommend collecting everything, text, code, and figures, into a single JupyterLab Notebook [?], but a pdf with separate Python files is also acceptable.

2 Theory

2.1 Sinusoidal Signals and Phasors

Recall from the lectures that a sinusoidal signal $x(t)$ is written as

$$x(t) = A\cos(\omega t + \phi) = \mathrm{Re}\left\{Ae^{j(\omega t+\phi)}\right\} = \mathrm{Re}\left\{Xe^{j\omega t}\right\} \quad , \qquad X = Ae^{j\phi} \tag{1}$$

where A is the amplitude, $\omega = 2\pi f$ is the angular frequency, f is the frequency, and ϕ is the phase. X is a *phasor* or *complex amplitude* that includes both the amplitude and the phase of the signal. Analysis of sinusoidal signals like $x(t)$ can be simplified by manipulating complex phasors instead of working with the amplitude and phase separately. See Chapter 2 in the textbook [1] for details.

Table 1. Recommended format for importing Python modules. NumPy is used to manipulate signals as arrays, Matplotlib to plot results. The real-valued mathematics library math is loaded for simple access to the trigonometric functions and π. The complex mathematics library cmath is included for simpler access to selected complex-valued functions. Note that math and cmath work only on scalars, not on arrays.

```
import numpy as np
import matplotlib.pyplot as plt
from math import pi, cos, sin, tan  # For readability, also covered by NumPy
from cmath import exp, sqrt
```

2.2 Adding Sinusiods Using Complex Exponentials

Consider a signal that is the sum of sinusoids with equal frequency f_0, but where the amplitudes A_k and phases ϕ_k of the individual signals can be different,

$$x_s(t) = \sum_{k=1}^{N} A_k \cos(2\pi f_0 t + \phi_k) \, . \tag{2}$$

The resulting signal can be found by representing the individual components as complex vectors. This is called *phasor summation* and is in general easier than using trigonometric identities,

$$x_s(t) = \mathrm{Re}\left\{\sum_{k=1}^{N} A_k e^{j\phi_k} e^{j2\pi f_0 t}\right\} = \mathrm{Re}\left\{\sum_{k=1}^{N} X_k e^{j2\pi f_0 t}\right\} \quad , \quad X_k = A_k e^{j\phi_k} \, . \tag{3}$$

The factor $e^{j2\pi f_0 t}$ in (3) is equal for all the individual signals. Hence, the amplitude A_k and phase ϕ_k of the summed signal $x_s(t)$ can be found by summing the complex amplitudes,

$$x_s(t) = \mathrm{Re}\left\{X_s e^{j\omega t}\right\} = A_s \cos(2\pi f_0 t + \phi_s) \, , \qquad X_s = \sum_{k=1}^{N} X_k = A_s e^{j\phi_k} \, . \tag{4}$$

The resulting signal will have the same frequency f_0 and period $T_0 = 1/f_0$ as the original signals.

2.3 Harmonics and Periodic signals

Consider another signal $x_h(t)$ where the frequencies f_k of the individual cosine-waves are different, but integer multiples of a fundamental frequency f_0,

$$f_k = k f_0 \quad , \qquad k = 0, 1, 2, \dots \, . \tag{5}$$

The individual signals $\cos(2\pi k f_0 t + \phi_k)$ are called *harmonics*. The summed signal $x_h(t)$ can be written as

$$x_h(t) = \sum_{k=1}^{N} A_k \cos(2\pi k f_0 t + \phi_k) = \mathrm{Re}\left\{\sum_{k=1}^{N} X_k e^{j2\pi k f_0 t}\right\} \, . \tag{6}$$

The period T_0 of the fundamental frequency f_0 is

$$T_0 = \frac{1}{f_0} \, , \tag{7}$$

while the periods T_k of the harmonics are

$$T_k = \frac{1}{f_k} = \frac{1}{k f_0} = \frac{T_0}{k} \, . \tag{8}$$

The signal with frequency $k f_0$ will repeat itself after the period $T_k = T_0/k$. It will therefore have repeated itself k times after the fundamental period T_0. Hence, all the frequency components $k f_0$ will also be periodic with period T_0, and the summed signal $x_h(t)$ will be periodic with period T_0 given by the fundamental frequency.

3 Programming Tips

3.1 Complex Numbers in Python with NumPy

Python can handle complex numbers like any other numbers. The module cmath contains elementary mathematical functions for use on *scalar* complex numbers, while NumPy includes mathematical functions for use on *arrays* of complex numbers.

Basic operations on complex numbers in Python are listed in Table 2. A complex number in Python has three public members, `real`, `imag`, and `conjugate()`, while other operations on can be found in cmath or NumPy. Note that the format of the functions calls depends on how the modules NumPy and cmath were imported. The list in Table 2 assumes the mudules were imported as in Table 1.

Table 2. Overview of complex number operations in Python. Details are found in the documentation for cmath and NumPy. The code assumes the modules cmath and NumPy are imported as described in Table 1. Note that the operations on scalar complex numbers are a mix of internal members (`z.real`, `z.imag`, `z.conjugate()`), built-in functions (`abs`) and functions from the cmath module (`phase`, `exp`). The NumPy functions can be used on both scalars and arrays.

Scalar	Array	Description	Mathematical notation
`z = complex(2,3)`		Creates a complex number.	$z = x + jy = 2 + 3j$.
`z = 2 + 3j`		Same as above	
`1j`		Imaginary unit, i or j.	$i = j = \sqrt{-1}$
`z.conjugate()`	`np.conj(z)`	Complex conjugate.	$z^* = x - jy$
`abs(z)`	`np.abs(z)`	Absolute value.	$\|z\| = \sqrt{x^2 + y^2}$
`phase(z)`	`np.angle(z)`	Phase in radians.	$\angle z$
`z.real`	`np.real(z)`	Real part.	$\text{Re}\{z\} = x$
`z.imag`	`np.imag(z)`	Imaginary part.	$\text{Im}\{z\} = y$
`exp(1j*theta)`	`np.exp(1j*theta)`	Complex exponential.	$e^{j\theta} = cos\theta + j\sin\theta$

3.2 Displaying Phasors

The Python file `zplot.py` contains functions to plot complex numbers as phasors, called Argand-diagrams. This is done by the function `zplot.phasor()`, see description in Table 3.

3.3 Vectorization

The NumPy module allows mathematical operations to be used on arrays. This is convenient when defining signals as function of time. In the cosine-function $x(t) = A\cos(2\pi ft + \phi)$, the amplitude A and phase ϕ are scalars, while the time t is a vector spanning the time interval to be investigated. Vectors covering a defined interval can be created in in NumPy by one of the following methods.

1) `t = np.arange(t_start, t_end, dt)`

   Specify start, stop, and step between the samples in the time vector. `t_start` is the first point in the vector, `t_end` marks the end of `t`, and `dt` is the interval between the time points. Note that `t_end` is not included in the time vector, `t` ends on the last point before `t_end`.

2) `t = np.linspace(t_start, t_end, n_points)`

   Specify start, stop, and total number of points in the time vector. `t_start` is the first point in `t`, `t_end` marks the end of `t`, and `n_points` is number of points in the vector. Note again that `t_end` is not included in `t`.

3) `t = np.logspace(t_start, t_end, n_points)`

   This is the same as `linspace`, but the numbers are evenly spaced on a logarithmic scale. The start and end points are specified by their logarithms, `start=2` means the first value is 10^2=100.

Table 3. Function `phasor()` to show complex numbers as phasors in the complex plane (Argand-diagrams). The function is found in the file `zplot.py` included for this lab.

```
def phasor(zk, labels=[], include_sum=False, include_signal=False, frequency=1):
    """Show complex amplitudes and resulting signals.

    Parameters
    ----------
    zk : Complex or list of complex
        Complex numbers to show

    labels=[] : List of strings, optional
        List of labels to mark phasors

    include_sum=False : Boolean, optional
        Show the sum of all numbers as a phasor

    include_signal=False : Boolean, optional
        Include a plot of signals as function of time

    frequency=1 : Float, optional
        Frequency used to plot the signals

    Returns
    --------
    ax : List of Matplotlib Axes
        Handle to axes containing the plots
    """
```

3.4 Comparing Graphs

We sometimes want to compare graphs. Two useful methods to do this in Matplotlib are

1) Use `subplot` to stack the graphs vertically or horizontally.
2) Plot the two curves in the same graph, the first with a solid line ('-') and the second with a dashed line ('- -'). Even if the curves are very similar, the first one will be visible behind the second.

Example code for this is shown in Table 4, more information is found in the documentation for Matplotlib.

Table 4. Code for stacking graphs using `subplot` (Example 1) and for using dashed lines to distinguish between similar graphs (Example 2).

```
# Example 1: Using subplot to stack plots
plt.subplot(2, 1, 1)     # Two plots stacked vertically, use first (upper) graph
plt.plot(t, x1)          # Plot first result (x1) as function of t

#-- Formatting of the first subplot comes here (labels, axis limits etc.) ---

plt.subplot(2, 1, 2)     # Two plots stacked vertically, use second (lower) graph
plt.plot(t, x2)          # Plot second result (x2) as function of t

#-- Formatting of the second subplot comes here ---

# Example 2: Showing two similar graphs in one plot
plt.plot(t, x1, '-')     # Plot first result (x1) with a solid line
plt.plot(t, x2, '--')    # Plot second result (x2) with a dashed line

#-- Formatting of the graph comes here ---
```

4 Training Exercises

Reporting

Collect answers and code in a JupyterLab notebook. Export this to pdf and upload it to Canvas.

You may prefer to do some of the coding in a development tool with better options for testing and debugging, such as Spyder. If so, the code can be pasted into JupyterLab, or the .py-files from from Spyder can be loaded into JupyterLab.

4.1 Complex Numbers

1) Load `zplot` and enter the two complex numbers $z_1 = 2e^{j\pi/3}$ and $z_2 = -\sqrt{2} + 5j$.

   Use Python to find the real and imaginary parts, magnitude, and phase of z_1 and z_2.

   Display z_1 and z_2 and the sum $z_1 + z_2$ as phasors with `zplot.phasor`. The input to `zplot.phasor` is specified as a `list`, this is made by enclosing the numbers in square brackets, e.g., `[z1, z2]`.

2) Find the complex conjugate z^* and inverse $1/z$ for z_1 and z_2 and plot them together with z_1 and z_2 using `zplot`. Recall what you have learned about complex numbers in math courses. Are the results as expected?

3) Calculate the product $z_1 z_2$ and ratio z_1/z_2 and plot them using `zplot`. Are these results as expected?

4) Calculate the products of the conjugates, $z_1 z_1^*$ and $z_2 z_2^*$. Plot them in the same diagram as z_1 and z_2 and explain the result.

5) Calculate the sums $z_1 + z_1^*$ and differences $z_1 - z_1^*$ of the conjugates and plot them in the same diagram as z_1. Do the same for z_2. Explain these results.

4.2 Python Function to Generate a Sinusoid Signal

1) Write a function (`def` in Python) that generates a single sinusoid, $x(t) = A\cos(2\pi f t + \phi)$ from the four input arguments amplitude A, frequency f, phase ϕ and duration.

   The function shall return the signal $x(t)$ and the time vector t with the time-points where the signal is evaluated.

   The function shall generate exactly 32 values of the sinusoid per period.

   A skeleton for this function `make_cos` with the recommended function call and documentation string is listed in Table 5.

2) Demonstrate that your function works by plotting the output for the following parameters:

$$A = 10^4 \qquad f = 1.5\,\mathrm{MHz} \qquad \phi = -45^\circ \qquad \text{Duration } 10^{-6}\,\mathrm{s}$$

   Note that the phase must be converted to radians before calculating the result.

   Calculate the value of $x(t)$ at $t = 0$. Does this agree with the plot?

   What is the period of the signal? Does this agree with the plot?

4.3 Python Function to Generate a Sum of Sinusoid Signals

Signals are often described as a sum of sinusoids with different amplitudes, frequencies, and phases. It can therefore be convenient to have a function that generates a sum from several cosine-functions, each specified by its amplitude A_k, frequency f_k, and phase ϕ_k.

1) Write a function that generates a signal

$$x(t) = \sum_{k=1}^{N} A_k \cos(2\pi f_k t + \phi_k) = \sum_{k=1}^{N} X_k e^{j2\pi f_k t}. \tag{9}$$

   The arguments to the function are the frequencies f_k and complex amplitudes $X_k = A_k e^{j\phi}$, the sample rate f_s and the signal duration.

Table 5. Skeleton for a function to generate a cosine signal from amplitude, frequency, and phase. The first lines are the recommended function call and docstring. The last line specifies that the signal `x` and time vector `t` are returned.

```
def make_cos(A, f0, phase, duration):
"""Make a cosine-function from specified parameters.
Parameters
----------
A : float
    Amplitude
f0: float
    Frequency [Hz]
phase: float
    Phase [radians]
duration: float
    Duration of signal [seconds]

Returns
-------
x: 1D array of float
    Cosine-wave
t: 1D array of float
    Time vector [seconds]
"""

#-- Your code comes here ---
return x, t
```

The function shall return the summed signal $x(t)$ and the time vector t where the signal is evaluated.

The frequencies f_k and complex amplitudes X_k shall be specified as lists or NumPy arrays, and the function shall accept any number of frequency components.

Each frequency f_k has a complex amplitude X_k, so the vectors f_k and X_k must have equal length. The function shall return an error message if this is not fulfilled.

A skeleton of this function `make_summed_cos` with the recommended function call and documentation string is listed in Table 6.

2) Demonstrate that your function works by plotting the output for a signal that is the sum of the following components.

k	Frequency f_k [Hz]	Complex amplitude X_k
1	0	10
2	100	$14e^{-j\pi/3}$
3	250	$8j$

Set the sample rate to 10 000 Samples/s, the duration of the signal to 0.1 s, and the start time to 0 s. Plot the result with Matplotlib.

Make a new subplot and plot the individual frequency components of the signal in this subplot.

3) Measure the period T_0 of the signal from the graph. Compare this to the periods T_k of the individual frequency components f_k.

Explain how the period of the summed signal can be calculated from the periods of the individual components.

4) Generate the signal

$$x(t) = \mathrm{Re}\Big\{-2e^{j50\pi t} - e^{j50\pi(t-0.02)} + (2-3j)e^{j50\pi t}\Big\}$$

over a time range that covers 2 periods.

Plot the signal x as function of time t.

5) All frequency components in the signal above are equal. Hence, the amplitude and phase can be calculated by summing its complex amplitudes, *phasors*.

   Use the function `zplot.phasor` from earlier to plot the phasor diagram for this signal, and check that this agrees with the result from `make_summed_cos`.

   `zplot.phasor` has optional arguments that can be set to illustrate this better, see Table 3.

`include_sum = True`	Include the sum of all the phasors to the plot.
`include_signal = True`	Plot the signals corresponding to the phasors.
`frequency = <value>`	Frequency to use when plotting the signals.

Table 6. Skeleton for a function `make_summed_cos` to generate a signal by summing cosine-functions with different complex amplitudes and frequencies. The first lines are the recommended function call and docstring. The last line specifies that the signal `x` and time vector `t` are to be returned. Note how the start time `t_start` is specified as an optional argument with default value 0.

```python
def make_summed_cos(fk, Xk, fs, duration, t_start=0):
    """Synthesize a signal as a sum of cosine waves

    Parameters
    ----------
    fk: List of floats
        Frequencies [Hz]
    Xk: List of floats
        Complex amplitudes (phasors)
    fs: float
        Sample rate [Samples/second]
    duration: float
        Duration of signal [s]
    t_start : float, optional
        Start time, first point of time-vector [seconds]

    fk and Xk must have the same lengths.

    Returns
    -------
    x: 1D array of float
        Signal as the sum of the frequency components
    t: 1D array of float
        Time vector [seconds]
    """

    #-- Your code comes here ---

    return x, t
```

5 Lab Exercise: Direction Finding

The text in this exercise is taken from [?] and somewhat modified.

Why do humans have two ears? One answer is that the brain can process acoustic signals received at the two ears and determine the direction to the source of the acoustic energy. Using sinusoids, we can describe and analyze a simple scenario that explains this direction finding capability in terms of phase differences or time-delay differences. This principle is the basis for a wide range of other applications, such as radars that locate and track airplanes, 5G and wifi antennas, and phased array transducers for medical ultrasound imaging and sonar.

Direction Finding With Microphones

Consider a system consisting of two microphones that both hear the same source signal. The microphones are placed some distance apart, so the sound must travel different paths from the source to the receivers. When the travel paths have different lengths, the two signals will arrive at different times.

The direction to the source can be calculated from the time difference between the signals received by the two microphones. If the source signal is a sinusoid, the time difference can be calculated from the phase difference. The scenario is illustrated in Figure 1. A vehicle traveling along the roadway has a siren that transmits a tone with frequency f_v=400 Hz. The roadway forms the x-axis of a coordinate system. Two microphones are located some distance away and aligned parallel to the roadway. The distance from the road to the microphones is y_r=100 m and the microphone separation is d=0.40 m. The task is to process the signals from the microphones to find the direction to the vehicle, described by the angle θ in Figure 1.

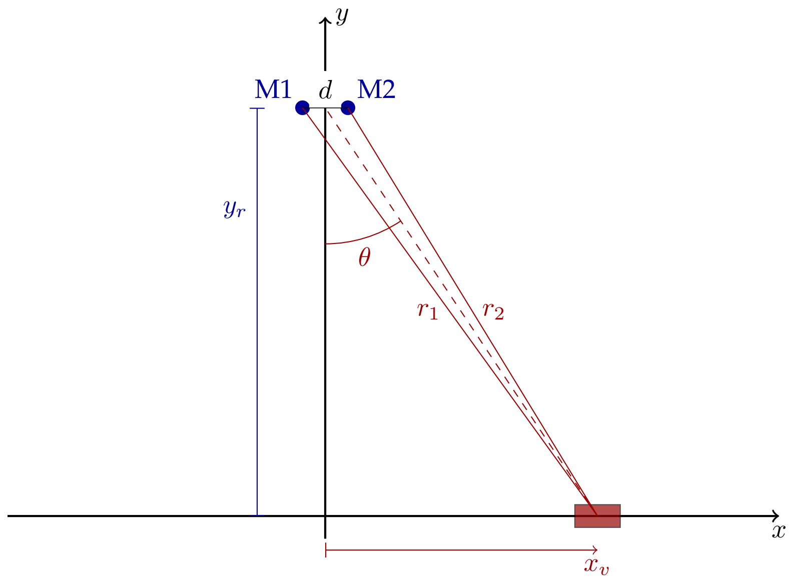

Figure 1. Direction finding using two microphones. A vehicle at position x_v travels along the x-axis while emitting a sound with frequency f_v=400 Hz. The sound is picked up by two microphones M1 and M2 positioned with spacing d=0.40 m. The difference in path length $\Delta r = r_1 - r_2$ causes a phase-shift between the signals received by the two microphones. This phase shift can be used to estimate the direction to the vehicle, specified by the angle θ.

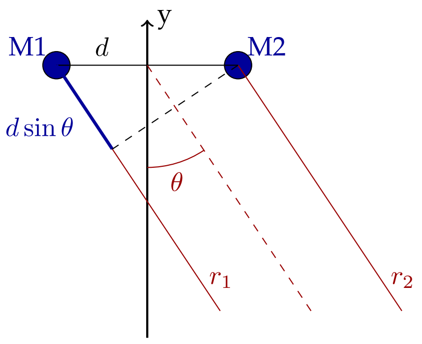

Figure 2. Zoomed-in version of Figure 1. When the distance to the source is very long compared to the spacing between the microphones, $r_1, r_2 \gg d$, the paths can be approximated as parallel. In this case, the difference in travel length to the two microphones is $r_2 - r_1 = \Delta r \approx d \sin\theta$.

1) The time from the sound is transmitted by the source until it is received by one of the microphones can be computed for the two propagation paths r_1 and r_2. The time is given by the distance from the vehicle location at coordinate $(x_v, 0)$ to either M1 at coordinate $(-\frac{1}{2}d, y_r)$ or M2 at $(+\frac{1}{2}d, y_r)$.

   The speed of sound in air is c=340 m/s. Write mathematical expressions for the time t_1 it takes for the sound to travel from the the source to M1 and for t_2 from the source to to M2. Plot t_1 and t_2 as functions of the vehicle position x_v from −400 m to 400 m.

   Can you see any difference between t_1 and t_2 in the graph?

2) In the simplest model, the signals received by the microphones, $s_1(t)$ at M1 and $s_2(t)$ at M2, are delayed copies of the transmitted signal $s(t)$,

$$s_1(t) = s(t - t_1) \qquad\qquad s_2(t) = s(t - t_2) \tag{10}$$

where $s(t)$ is the signal transmitted from x_v.

Assume that the tone $s(t)$ emitted from the source is a sinusoid with zero phase, frequency f_v=400 Hz, and amplitude A=1. The phases ϕ_1 and ϕ_2 of the received signals $s_1(t)$ and $s_2(t)$ can be found from the delays t_1 and t_2.

Find the phases ϕ_1 and ϕ_2.

Plot the phases ϕ_1 and ϕ_2 as functions of x_v from -400 m to 400 m in one subplot.

Plot the phase difference $\Delta\phi = \phi_2 - \phi_1$ as function of x_v in another subplot.

Comment the results.

3) The received signals can be represented as phasors.

Use `zplot` to show the signals received by M1 and M2 when the vehicle is at positions x_v=-400 m, -100 m, 0 m, 100 m, and 400 m.

Set optional argument `include_signal=True` to display the signals, and set the argument `frequency` to the correct frequency.

Calculate the phase differences by hand and compare this with the plots.

4) The next step is to convert the relative time-shifts into a direction θ.

The distance from the microphones to the source is much larger than the distance between the microphones, making the the paths to M1 and M2 almost parallel. This is illustrated in Figure 2, where we have zoomed in on the microphones in Figure 1.

Figure 2 shows that the difference Δr in propagation distance for paths r_1 and r_2 can be approximated to

$$\Delta r = r_1 - r_2 \approx d \sin\theta \, . \tag{11}$$

This is called the *far field approximation* and is often used to find the beam pattern from antennas, loudspeakers, ultrasound transducers, and other transmitters or receivers for waves.

Calculate the propagation time difference $\Delta t = t_2 - t_1$ from the approximation (11). Use this to find the approximated phase difference $\Delta\phi_F$ between the two received signals. Plot $\Delta\phi_F$ in the same graph as the correct value $\Delta\phi$ found previously.

Comment the result.

5) The last step is to write a Python function to process the received signals and find the direction θ.

Show first that the phase difference between two phasors X_1 and X_2 is given by

$$\Delta\phi = -\angle(X_1 X_2^*)$$

where the superscript * denotes the complex conjugate and $\angle$ is the phase of a complex number.

Calculate the complex amplitudes $X_1 = A_1 e^{j\phi_1}$ and $X_2 = A_2 e^{j\phi_2}$ received at M1 and M2 as function of vehicle position x_v. Assume no propagation loss, so that the amplitudes are unchanged, $A_1 = A_2 = A$.

Use this and earlier results to write a Python-function that will compute the direction θ from the complex amplitudes.

Run this function for the vehicle moving from $x_v = -400$ m to 400 m and plot the angle θ_F calculated from the phase-shifts using the far-field approximation. Use degrees [°] when plotting the angle.

Compare θ_F to the true angle calculated from the actual position of the vehicle.

Comment the result.

Concluding Remarks

This exercise has illustrated how the direction to a sound source can be estimated using two receivers separated by a distance much smaller than the distance to the source. This principle is the basis for how sound and radar beams can be steered and focused without moving parts in *phased arrays*. Such arrays are used to steer medical ultrasound beams by GE Vingmed Ultrasound and in underwater sonars made by Kongsberg Discovery.

References

[1] J. H. McClellan, R. Schafer, and M. Yoder, *DSP First*, 2nd ed. United Kingdom: Pearson Education Limited, 2016.

[2] NumPy. (2024) NumPy ver. 2.2. [Online]. Available: https://numpy.org/doc/2.2

[3] C. R. Harris et al., "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[4] Matplotlib. (2024) Matplotlib 3.10.0. [Online]. Available: https://matplotlib.org/stable

[5] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: https://doi.org/10.1109/MCSE.2007.55

[6] P. Raybaut. (2024) Spyder. [Online]. Available: https://docs.spyder-ide.org/

[7] Anaconda, Inc. (2024) Anaconda. [Online]. Available: https://www.anaconda.com