university of groningen

faculty of science and engineering

Code Documentation

Signals and Systems (for AI)

WBAI016-05.2024-2025

Title of Deliverable:	Lab 3
Group Number:	75
Author(s):	Marinus v/d Ende Matthijs Prinsen
Delivery Date:	January 17, 2025

Overview

This document was written for **Lab 3** of the course **Signals and Systems**. It contains explanations of design choices and documentation for the implementations for the questions *2.3: Inverse DFT using Vandermonde matrix* and *2.5: z-domain to time domain*.

Inverse DFT with Vandermonde matrix

For this question, we were meant to modify the code from the previous question (Discrete Fourier transform using Vandermonde matrix) in order to calculate the inverse DFT.

The input for this question was a single signal $X[k]$ and the output is the complex form of $x[n]$.

We are meant to use the Vandermonde matrix when making this computation. All without **numpy**.

Solution Description

In our code implementation, we have two functions to solve the problem: `build_vandermonde` and `dot_product`. The Vandermonde matrix is constructed by calculating the real and imaginary parts of the signal using **Euler's formula** and storing these values (denoted as ω) in the matrix as tuples:

$$\omega = e^{-j\frac{2\pi}{N}kn} = \cos\left(\frac{2\pi}{N}kn\right) + j\sin\left(\frac{2\pi}{N}kn\right) \tag{1}$$

This process creates a Vandermonde matrix structured as follows:

$$W = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix} \tag{2}$$

The pseudocode for building this Vandermonde matrix is as follows:

Algorithm 1 Build Vandermonde Matrix

```
Require: $length \geq 0$
Ensure: $vandermonde$ is a $length \times length$ matrix
 1: $vandermonde \leftarrow []$                              ▷ Initialize an empty matrix
 2: for $n \leftarrow 0$ to $length - 1$ do
 3:     $row \leftarrow []$                                   ▷ Initialize an empty row
 4:     for $k \leftarrow 0$ to $length - 1$ do
 5:         $\omega \leftarrow \frac{2\pi nk}{length}$
 6:         $real \leftarrow \cos(\omega)/length$             ▷ Normalize by the length
 7:         $imag \leftarrow \sin(\omega)/length$
 8:         append $(real, imag)$ to $row$
 9:     end for
10:     append $row$ to $vandermonde$
11: end for
12: return $vandermonde$
```

After we have constructed our Vandermonde matrix, we do the dot product of the matrix and the input signal. This is computationally equivalent to taking the sum of each element of the input signal and multiplying it by omega(ω). This, in a coding context, is more efficient than first constructing a Vandermonde matrix at $O(n^2)$ time complexity, then having to loop through that matrix again at $O(n^s)$ complexity, leaving us with a grand time complexity of $O(2n^2)$. However, the question was answered using the Vandermonde matrix as instructed.
The pseudocode for the dot product is as follows:

Algorithm 2 Dot Product with Vandermonde Matrix

```
Require: $x$ is a vector of size $length$, $W$ is a $length \times length$ Vandermonde matrix
Ensure: $X$ is a vector of $length$ tuples $(real, imag)$
 1: $X \leftarrow []$                                       ▷ Initialize an empty result vector
 2: for $row \in W$ do
 3:     $real\_sum \leftarrow 0$
 4:     $imag\_sum \leftarrow 0$
 5:     for $k \leftarrow 0$ to $length - 1$ do
 6:         $real\_sum \leftarrow real\_sum + row[k].real \times x[k]$
 7:         $imag\_sum \leftarrow imag\_sum + row[k].imag \times x[k]$
 8:     end for
 9:     append $(real\_sum, imag\_sum)$ to $X$
10: end for
11: return $X$
```

Functions

build_vandermonde

```
build_vandermonde(length: int) → list[list[tuple[float, float]]]
```

Description: This function constructs a Vandermonde matrix for the Inverse Discrete Fourier Transform (IDFT). The matrix is built row by row, where each element is a complex exponential normalized by the input signal length.

Inputs:

- `length` (*int*): The size of the Vandermonde matrix (number of rows and columns).

Outputs:

- `vandermonde` (*list[list[tuple[float, float]]]*): A 2D list representing the Vandermonde matrix, where each element is a tuple containing the real and imaginary parts of the matrix element.

Notes:

- The complexity of this function is $O(n^2)$, which is not optimal for large matrices.
- Each element in the Vandermonde matrix is normalized by the input length to ensure proper scaling.

```
dot_product
```

```
dot_product(x:  list[int], W: list[list[tuple[float, float]]]) →
list[tuple[float, float]]
```

Description: This function computes the dot product between a vector and the Vandermonde matrix to obtain the IDFT output. Each result is represented as a tuple containing the real and imaginary parts of the complex sum.

Inputs:

- `x` (*list[int]*): A 1D list representing the input signal in the frequency domain.
- `W` (*list[list[tuple[float, float]]]*): A 2D list representing the Vandermonde matrix.

Outputs:

- `X` (*list[tuple[float, float]]*): A list of tuples representing the IDFT result, where each tuple contains the real and imaginary parts of the transformed signal.

Notes:

- The function iterates over each row of the Vandermonde matrix and computes the dot product with the input signal vector.

Z-domain to time-domain

For this question we need to take the zeros of a system as $X(z)$, given as angles ϕ_k and converted to roots z_k, and then calculate the impulse response of the system in the time-domain $h[n]$.

Mathematical Foundations

The process of converting a z-domain representation of a system's impulse response to the time-domain coefficients involves the following steps:

1. **Convert Angles (θ_k) into Roots (z_k) using Euler's Formula:**

$$z_k = e^{j\theta_k} = \cos(\theta_k) + j\sin(\theta_k) \tag{3}$$

   For simplicity, the roots are denoted as z_k.

2. **Expand $H(z)$ Using Roots:** Substitute the roots into the polynomial representation of $H(z)$:

$$H(z) = G\prod_{k=1}^{M}(1 - z_k z^{-1}) \tag{4}$$

   where $G = 1$ in this assignment.

   To implement this in code, I examined various polynomial expansions to identify patterns in the coefficients. For example:

   - **Two roots**:

$$H(z) = (1 - z_1 z^{-1})(1 - z_2 z^{-1})$$

   Expanding:

$$H(z) = 1 - (z_1 + z_2)z^{-1} + (z_1 z_2)z^{-2}$$

   The corresponding impulse response $h[n]$:

$$h[n] = [1, -(z_1 + z_2), z_1 z_2]$$

   - **Three roots**:

$$H(z) = (1 - z_1 z^{-1})(1 - z_2 z^{-1})(1 - z_3 z^{-1})$$

   Expanding:

$$H(z) = 1 - (z_1 + z_2 + z_3)z^{-1} + (z_1 z_2 + z_2 z_3 + z_3 z_1)z^{-2} - (z_1 z_2 z_3)z^{-3}$$

   Resulting in:

$$h[n] = [1, -(z_1 + z_2 + z_3), (z_1 z_2 + z_2 z_3 + z_3 z_1), -(z_1 z_2 z_3)]$$

The Pattern

Assume 10 roots ($M = 10$) and a monic polynomial:

$$a_0 = 1$$

$$a_1 = -(r_1 + r_2 + \cdots + r_{10}) \quad \text{(sum of single roots)}$$

$$a_2 = \sum_{i<j} r_i r_j \quad \text{(sum of products of pairs of roots)}$$

$$a_3 = -\sum_{i<j<k} r_i r_j r_k \quad \text{(sum of all triplets of roots)}$$

$$a_4 = \sum_{i<j<k<l} r_i r_j r_k r_l \quad \text{(sum of all quadruplets of roots)}$$

and so on, following the observed pattern of increased combinations of roots. Note that all **odd** coefficients are negative.

Using Python's `itertools.combinations()` initially seemed promising, but the nested multiplications of combinations resulted in a time complexity of $O(2^n)$, prompting a search for a better solution.

The Better Solution

After extensive research, we discovered **Horner's Method**[1], which uses an iterative approach to calculate polynomials. This method reduces the time complexity to $O(n)$, where n is the degree of the polynomial.
Horner's method simplifies polynomial evaluation by restructuring it into a nested form. For example:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

can be rewritten as:

$$P(x) = a_0 + x(a_1 + x(a_2 + x(\cdots + x(a_{n-1} + x \cdot a_n))))$$

For finding coefficients from roots, we adapt Horner's method. Start with $P(x) = 1$, and for each root r_i, iteratively update the polynomial as:

$$P(x) = P(x) \times (x - r_i)$$

Algorithm Implementation

To implement this, we use the following steps:

1. Convert the angles to complex roots:

$$z_k = \cos(\theta_k) + j\sin(\theta_k)$$

2. Initialize an array of coefficients, where the index corresponds to the $-n^{th}$ power of the polynomial (z^{-n}).

3. Use Horner's method to iteratively calculate the coefficients:

   - Start with the first coefficient as 1.
   - Update coefficients iteratively:

$$\texttt{coeffs}[i] -= z_k \times \texttt{coeffs}[i-1]$$

Pseudo-code

Algorithm 3 Calculate Polynomial Coefficients Using Horner's Method

```
Require: Angles of roots angles, degree $M$
Ensure: Polynomial coefficients coefficients
 1: Convert angles to roots:
 2: roots $\leftarrow$ [complex(cos(angle), sin(angle)) for each angle in angles]
 3: Initialize coefficients:
 4: $n \leftarrow M + 1$                                    ▷ Number of coefficients
 5: coefficients $\leftarrow [0] * n$
 6: coefficients[0] $\leftarrow 1.0$                          ▷ Set first coefficient to 1.0
 7: for each root in roots do
 8:     for $i$ from $n-1$ to 1 step $-1$ do
 9:         coefficients[$i$] $\leftarrow$ coefficients[$i$] $-$ root $\times$ coefficients[$i-1$]
10:     end for
11: end for
12: Output coefficients: coefficients
```

Bibliography

[1] Wikipedia contributors, "Horner's method — Wikipedia, The Free Encyclopedia," 2025. [Online; accessed 17-January-2025].