Audio Steganography Implementation Using Complex Algebra Based on Fast Fourier Transform Approach

Anindya Naufal Pinasthika 13524013[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13524013@mahasiswa.itb.ac.id, [2]naufalpinasthika@gmail.com

***Abstract*—In the world of information security, many ways a piece of software can be represented such a way its intended messages are concealed within another file or images. This is commonly referred to as steganography. One type of steganography is audio steganography, which is an act to insert a hidden message into an audio file. This paper aims to use complex algebra in a form of Fast Fourier Transform (FFT) as one of many ways to implement an audio steganography.**

***Keywords*—Steganography, audio, Fast Fourier Transform(FFT), complex algebra**

I. Introduction

Steganography is the art of hiding information and an effort to conceal the existence of the embedded information or message into a medium. It serves as a better way of securing message than cryptography which only conceals the content of the message but not the existence of the message. Original message is being hidden within a carrier such that the changes occurred in the carrier are not observable[1].

Steganography allows a person to communicate to other people in such a way that it's hard to decipher its messages because the messages themselves are hidden in plain sight. Furthermore, there is no conventional way to track its messages unlike other forms of message encryption.

Steganography comes with many forms, notably such as video, text, images, network, etc. One of the other types of steganography that will be covered in this paper is audio steganography, which is an act on concealing a hidden message onto audio format such that its audio will sound the same even after concealing and encrypting the messages.

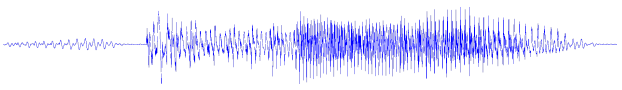

Fig 1.1 : Sample audio in waveform that contains no hidden messages
Source : writer's archive

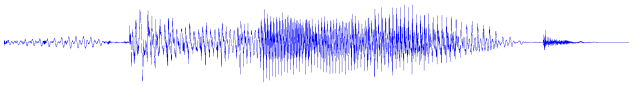

Fig 1.2 : Sample audio in waveform that contains the messages "hellow"
Source : writer's archive

In Fig1.1 and Fig1.2, while we can see that its visual waveform representation may be different, its audio property when a person listens to its sample will be the exact same. The only difference to both of the samples is the latter one contains a message "hellow" as a form of the hidden message that wants to be concealed from.

There are a lot of techniques to perform an audio steganography, but one of the famous one that will be covered on this paper is called Fast Fourier Transform (FFT), which uses complex algebra concept to manipulate the sound wave so that it can encrypt and decrypt a hidden message.

II. Fundamental Theorem

A. Complex Algebra

A complex number is a type of number system that can be represented by the structure $\alpha + \beta i$ with i having a property

$$i^2 = -1$$

Or

$$i = \sqrt{-1}$$

and will be interpreted as "imaginary" number with an i symbol [3].

The complex number $\alpha + \beta i$ can also be represented in a new plane that's called "complex plane" where x-axis will be interpreted as real number plane, where the normal real number α lies (1, 2, $\frac{1}{2}$, -3, etc.) and y-axis will be interpreted as "imaginary" plane where all the number βi resides. A complex plane can be sometimes referred to as the Argand plane. Suppose given the ordered pair (α, β) and plotted as a point in Argand plane as in Fig 2.1 below. Thus, the complex number

$$i = 0 + 1i$$

is identified with the point (0, 1) [2].

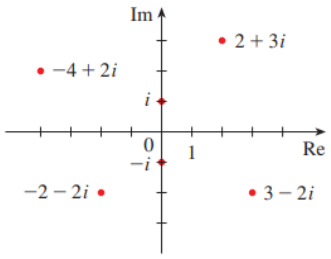

Fig 2.1 :Complex numbers as points in the Argand plane
Source: J. Stewart, *Essential Calculus: Early Transcendentals*, 2nd ed., Belmont, CA: Brooks-Cole, 2007, Section 12 (Complex Numbers).

After we define such that i with the property

$$i^2 = -1$$

if the imaginary unit is combined with two real numbers α and β by the processes of addition and multiplication, we can construct a complex number $\alpha + \beta i$. α are the real part of the imaginary while β are imaginary part of the complex number. If $\alpha = 0$, the number will be represented as purely imaginary, whereas if $\beta = 0$, it will be represented as purely real number. Zero is the only number which is at once real and purely imaginary. Two complex numbers are equal if and only if they have the same real part and the same imaginary part [4].

The the complex structure

$$\alpha + \beta i$$

can be used to do basic algebraic operations (addition, multiplication, etc.) and thus, can be represented as "complex algebra".

The sum and difference of two complex numbers are defined by adding or subtracting their real parts and their imaginary parts where we will do the operation respected to their status (a's position as a real number will only be operated by c who's also a real number, so on and so forth. While b having an imaginary status will only do operation with d respecting to their imaginary status)

$$(a + bi) + (c + di) = (a + c) + (b + d)i \quad (1)$$
$$(a + bi) - (c + di) = (a - c) + (b - d)i \quad (2)$$

For instance, given an example of 2 complex number such that the result of the addition is as follows:

$$(1 - i) + (4 + 7i) = (1 + 4) + (-1 + 7)i$$
$$= 5 + 6i$$

The product of complex numbers as defined by definition can also perform commutative and distributive method (which will be later useful for multiplication and division):

$$(a + bi)(c + di) = ac + adi + bci + bdi^2$$

Since that i have the property:

$$i^2 = -1$$

This becomes:

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i \quad (3)$$

Division of complex numbers will be performed the way we do square roots (by rationalizing its denominator of a rational expression). For the complex number

$$z = \alpha + \beta i$$

We define its complex conjugate (or "inverse" of the α and βi operand):

$$\bar{z} = \alpha - \beta i$$

If the operand in the denominator turns out to be:

$$z = \alpha - \beta i$$

We will satisfy the consistency of conjugate rule:

$$\bar{z} = \alpha + \beta i$$

To find the quotient of two complex numbers we multiply numerator and denominator by the complex conjugate of the denominator. For example if we express

$$\frac{-1+3i}{2+5i}$$

we multiply numerator and denominator by the complex conjugate of $2 + 5i$ to be $2 - 5i$, and we take advantage of the operation

$$\frac{-1+3i}{2+5i} = \frac{-1+3i}{2+5i} \cdot \frac{2-5i}{2-5i} = \frac{13+11i}{2^2+5^2} = \frac{13}{29} + \frac{11}{29}i$$

into a complex algebra result.

B . Fast Fourier Transform

Fast Fourier Transform (FFT) is an algorithm which gives a way we can multiply a certain 2 polynomial $A(x)$ and $B(x)$ in efficiently, rather than multiply it distributively (which takes $O(n^2)$ complexity), we can use an ingenious way in which turning the polynomial expression into value expression to multiply. The result of the multiplied value then can be expressed again as a polynomial function which will only take $O(n \log n)$ complexity. The heart of FFT lies within turning the polynomial function's coefficient into a value representation in the form of a complex value. The same can be done where we are gonna turn the value and turn it back to polynomial function to get the final result.

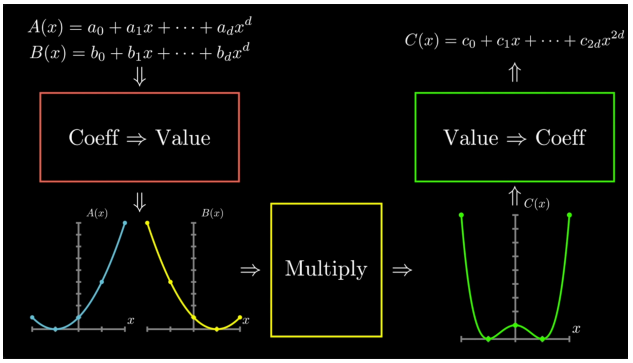

Fig 2.1 : Visualization process of FFT algorithm on polynomial multiplication.
Source : https://www.youtube.com/watch?v=h7apO7q16V0&t=1477s

The implementation of FFT is as follow:

```
import cmath
import math

def fft(a, invert):
    n = len(a)
    if n == 1:
        return
```

```
    half = n // 2
    a0 = [0] * half
    a1 = [0] * half

    for i in range(half):
        a0[i] = a[2 * i]
        a1[i] = a[2 * i + 1]

    fft(a0, invert)
    fft(a1, invert)

    angle = 2 * math.pi / n * (-1 if
invert else 1)
    w = 1 + 0j
    wn = cmath.exp(1j * angle)

    for i in range(half):
        a[i] = a0[i] + w * a1[i]
        a[i + half] = a0[i] - w *a1[i]

        if invert:
            a[i] /= 2
            a[i + half] /= 2

        w *= wn
```

In the world of FFT implementation there are 2 types of FFT. The normal FFT is where we are going to "transform" from a polynomial coefficient and turn them into a complex number value. The second type of FFT is Inverse Fast Fourier Transformation (IFFT) where we are doing the exact opposite where we are turning the complex values and turning them into the final polynomial result. With that in mind, the implementation of FFT starts from giving an input "a" as an array of polynomial coefficients (with the rules that the size of the array must be power of 2). For example, if we were to turn

$$A(x) = 5 + 3x + 2x^2 + x^3$$

Then the given input is

$$a = [5, 3, 2, 1]$$

The core FFT algorithm is a recursive function where the initial base case is to return the initial value if the length of the array goes to 1 (if its [5], return 5). The next step is to divide the function into even (a_0) and odd (a_1) indexes.

From the previous example, in the first recursive iteration:

$$a_0 = [5, 2]$$

$$a_1 = [3, 1]$$

When the second recursion starts we get:

$$a_0 = [5]$$

$$a_1 = [2]$$

$$a_0 = [3]$$

$$a_1 = [1]$$

We combine it via recursive backtrack (when n == 2) as:

$$(a_0 = 5 + 2x)$$

$$(a_1 = 3 + x)$$

After getting the base polynom we can construct the w such that:

$$w = e^{\frac{2\pi i}{2}}$$

We construct the angle ($\frac{2\pi}{n}$) which already been expressed as $\frac{2\pi}{2}$ and we will get $180°$. The w will go around the circle from

$$1 \rightarrow -1$$

When calculating each of the a variable respectively.

We then calculate the value (when w = 1 and w = -1) :

$$a_0 = a_o[i] + w * a_1$$

$$a_0 = 5 + 1 * 2 = 7$$

$$a_0 = 5 - 1 * 2 = 3$$

$$a_0 = [7, 3]$$

Using the same steps for a_1:

$$a_1 = [4, 2]$$

The final backtracking (n == 4)makes so that angle are:

$$\frac{2\pi}{4} = 90°$$

Which makes so that w circle around:

$$1 \rightarrow i \rightarrow -1 \rightarrow -i$$

So that the calculation are follows in the even part (using w = 1 and w = -1):

$$a_0 = 7 + 1 * 4 = 11$$

$$a_0 = 7 - 1 * 4 = 3$$

The odd will be a slightly different as we are gonna use imaginary part (w = i and w = -i):

$$a_1 = 7 + i * 2 = 3 + 2i$$

$$a_1 = 7 - 1 * 2 = 3 - 2i$$

The final placement will be placed so that:

$$a = [11, 3 + 2i, 3, 3 - 2i]$$

For the inverse part of FFT (IFFT), the only difference is to inverse the angle. The rest of the process will stay the same afterwards. We can summarize it by giving another argument "invert" which tells us if we are doing FFT or IFFT, pass invert = 1 to do IFFT, pass invert = 0 to do FFT.

III. Implementation

A. Initial Tools

We will be using Python as the main programming language for simplicity and readability. Python also gives a useful number of libraries that will be important when processing the audio file. The initial audio will be using is .wav file and transform them (encoding a message or decoding a message) via python script. The initial implementation defines libraries and a set of variables:

```
import numpy as np
import scipy.io.wavfile
import sys
SEGMENT_SIZE = 4096
START_INDEX = 200
```

```
MARKER_START = "@@@"
MARKER_END = "###"
```

We will only be using the np library to do the FFT function. The rest of libraries (scipy and sys) will be used only for processing audio and argv on the Python.

The marker is used as a “mark” to identify in the program later where we will put our hidden messages on. Because of the audio format the data (in binary) will consist of “random” bits that will be represented by something that the computer will be hard to read. For example, a certain audio .wav file may produce the sound of a wind, but when the data converts to binary and later the string format, it will look like

“D¥µ§¶®.wlz”

Inserting a hidden messages so that the overall string will look like

“D¥this_is_a_secret_messageµ§¶®.wlz”

Maybe works at first glance, but the program will be having a hard time to pinpoint “where” is the location of “this_is_a_secret_message”. Using the marker that will look like

“`”

Will help the program identify hidden messages by simply identify if there are strings that start from @@@ and end to ###.

B. Audio Encoding

The way FFT will do on encoding hidden message in audio using the code follows:

```
def encode_process(filename):
    rate, channels = load_audio(filename)

    message = input("Enter message: ")
    full_message = MARKER_START + message
+ MARKER_END
    binary_message =
text_to_bits(full_message)

    msg_length = len(binary_message)
    max_capacity = (SEGMENT_SIZE // 2) -
START_INDEX

    if msg_length > max_capacity:
        print("Message too long.")
        return

    segment = channels[0][:SEGMENT_SIZE]

    spectrum = np.fft.fft(segment)
    magnitude = np.abs(spectrum)
    phase = np.angle(spectrum)

    for i in range(msg_length):
        bit = binary_message[i]
        freq_idx = START_INDEX + i
        mirror_idx = SEGMENT_SIZE -
freq_idx

        target_angle = np.pi/2 if bit ==
0 else -np.pi/2

        phase[freq_idx] = target_angle
        phase[mirror_idx] = -target_angle

        if magnitude[freq_idx] < 500:
            magnitude[freq_idx] = 500
        if magnitude[mirror_idx] < 500:
            magnitude[mirror_idx] = 500

    new_spectrum = magnitude * np.exp(1j
* phase)
    new_segment =
np.fft.ifft(new_spectrum).real

    channels[0][:SEGMENT_SIZE] =
new_segment.astype(np.int16)

    output_name = "steg_" + filename
    save_audio(output_name, rate,
channels)
    print(f"Saved to {output_name}")
```

The audio .wav file will be first processed via load_audio() function to get its rate (the speed of the wave audio in Hz) and channels (the raw audio data consist of roughly [left,right] sound). After processing the audio we will construct the hidden messages by concatenating the messages with `MARKER_START` and `MARKER_END`. If the messages were to look like:

“this_is_a_secret_message”

Then the constructed messages that will be encoded to the audio will look like:

“@@@this_is_a_secret_message###”

And will be converted to binary messages in array format ([0, 1, 0, 0, ...]) and will be checked if the given input messages are too long.

The next part is to transform the given audio into spectrum, magnitude, and phase respectively. This works because FFT can break down an audio format from only an amplitude over time into a frequency component (pitch, volume, phase, etc.)

The main part of the program is a for loop where we are gonna “inject” our messages into the audio. We first take the messages in binary format and take its bit (0 or 1) from index i and will insert it that starts from `START_INDEX`. The property of FFT in audio makes it so that after we manipulate a certain amount of bit in the freq_idx, we also have to change its index mirror (`SEGMENT_SIZE - freq_idx`). The current bit then will calculate its target_angle. The rules are follows:

$$bit\ 0\ =\ 90°$$
$$bit\ 1\ =-\ 90°$$

We then change the original phase from the audio and turn them into our deliberate phase (the messages that we are trying to encode).

The last part is to force the audio magnitude and its mirror to 500 to ensure our messages does not disappear when we are gonna reconstruct the audio. figures and tables may span both columns.

The final part is to reconstruct our new audio format

where now there is an encoded hidden message back to the audio where we transform the frequency back to its time domain. We then save the audio file into a brand new type of .wav with "steg_" to label that there is a hidden message within that audio .wav file.

The before and final audio result can be viewed on the github repository on the Attachment page.

C. Audio Decoding

Decoding a steganography audio will have to follow the assumption that it will follow the same rules as when the audio was encoded (which means the concealed messages must have the same `MARKER_START` and `MARKER_END`, safety max messages and index, etc.). The following audio decoding implementation in audio steganography using Python follows:

```
def decode_process(filename):
    rate, channels = load_audio(filename)

    segment = channels[0][:SEGMENT_SIZE]

    spectrum = np.fft.fft(segment)
    phase = np.angle(spectrum)

    extracted_bits = ""
    for i in range(2000):
        freq_idx = START_INDEX + i
        angle = phase[freq_idx]

        if angle > 0:
            extracted_bits += "0"
        else:
            extracted_bits += "1"

    raw_text =
bits_to_text(extracted_bits)

    if MARKER_START in raw_text and
MARKER_END in raw_text:
        content =
raw_text.split(MARKER_START)[1]
        clean_message =
content.split(MARKER_END)[0]
        print(f"Message:
{clean_message}")
    else:
        print("No hidden message found.")
```

The first step of the decoding process is the exact same when encoding an audio, which is where we will load the audio .wav file inload_audio() function. The next part is to take the data into a segment where it will take the first `SEGMENT_SIZE` available. We then perform another FFT to transform the time domain in audio into a spectrum domain where we get the spectrum and then phase data from its angle (we will not be extracting magnitude/volume as there is no purpose to that component when we are trying to decrypt the messages, the only component that we will need is only spectrum). We then loop from 0 to 2000 bits (this can be changed based on assumption that we take, if the string of the hidden message is very long, we can increase it to 10000). We then transform from the bit into string using bits_to_text(). By this process the result will still looks like

"D¥@@@this_is_a_secret_message###µ§¶®.wlz"

But then the ingenious part is to detect the "@@@" and "###". If it's exist in the right order we can parse the message that we want to get. If there are no hidden messages/the parse failed, we return "No hidden message found".

IV. Result Analysis

A. Encoding

Given the audio voice-note.wav the encoding process, we will run python script:

```
python3 audio_steg.py voice-note.wav
encode
```

The result will be shown as follows:

```
Thertaselette@DESKTOP-G3N13UL:~/tubes/makalah
_algeo$ python3 audio_steg.py sine.wav encode
Enter message: this_is_a_secret_message
Saved to steg_voice-note.wav
```

The result will produce another audio steg_sine.wav and will successfully hear the exact same as the initial audio without steganography.

We can confirm this by decoding the steg_voice-note.wav messages:

```
Thertaselette@DESKTOP-G3N13UL:~/tubes/makalah
_algeo$ python3 audio_steg.py steg_voice-note.wav
decode
Message: this_is_a_secret_message
```

B. Decoding

The process of testing the decoding messages can be done by first given an already steg_[audio_name].wav using the same base of assumption where they need to have the exact same "mark".

Supposed that we already been given steg_voice.wav, we will run the python script:

```
python3 audio_steg.py steg_voice.wav
decode
```

The result will be shown as follows:

```
Thertaselette@DESKTOP-G3N13UL:~/tubes/makalah
_algeo$ python3 audio_steg.py steg_voice.wav decode
Message: Algeo{th1s_1s_my_h1993n_m3ss4ges_:D}
```

We can show the hidden messages that were hidden in the steg_voice.wav are:

```
Algeo{th1s_1s_my_h1993n_m3ss4ges_:D}
```

V. Conclusion

Fast Fourier Transform is one of the most important algorithms in modern times. As there are so many use cases and implementations that comes from this algorithm. And FFT would be impossible which if not the result of complex numbers taking a role in its algorithms. In many such real life cases of FFT, one of it is audio manipulation, where we can manipulate a certain part of the audio, one of which is audio steganography, where we can conceal a hidden message inside an audio that can be useful in digital forensic, digital security, and many more.

VI. Attachment

[1] Github Repository: https://github.com/Naufal-Pinasthika/Audio-Steganography
[2] Video Presentation: https://drive.google.com/file/d/1DGonG_OCsoyaIBncntoZod0x-KN6u5td/view?usp=sharing
[3] Audio link that are used in this paper: https://drive.google.com/drive/folders/1ru5W7hKSX1XTKR9deOJHXTwfZNI12OeF?hl=id

VII. Acknowledgment

The author is deeply thankful to Allah SWT for providing strength, determination, and opportunity to authors in persuading academic journey in Informatics Engineering Institute Teknologi Bandung and bring Linear Algebra and Geometry paper to completion. The author also expresses deep appreciation to Dr. Ir. Rinaldi Munir, M.T., lecturer on IF2123 Linear Algebra and Geometry course, for his dedicated guidance and support, which has inspired author in many such ways. The author would like to express immense gratitude to the authors family and friends which help authors in their journey.

References

[1] Kumar, Arvind & Km, Pooja. (2010). Steganography- A Data Hiding Technique. International Journal of Computer Applications. 9. 10.5120/1398-1887. Accessed Dec. 21, 2025.
[2] Stewart, J., *Essential Calculus: Early Transcendentals*, 2nd ed. Belmont, CA: Brooks-Cole, 2007. Accessed Dec. 21, 2025.
[3] Munir, Rinaldi, *Aljabar Kompleks*, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-25-Aljabar-Kompleks-2025.pdf. Accessed Dec. 21, 2025.
[4] L. V. Ahlfors, *Complex Analysis: An Introduction to the Theory of Analytic Functions of One Complex Variable*, 3rd ed. New York: McGraw-Hill, 1979. Accessed Dec. 24, 2025.
[5] cp-algorithms.com, "Fast Fourier Transform (FFT) — cp-algorithms," [Online]. Available: https://cp-algorithms.com/algebra/fft.html. Accessed Dec. 24, 2025.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Desember 2025

Anindya Naufal Pinasthika 13524013