フーリエ変換・窓関数・高速フーリエ変換

24c1121 茂木一葉

2025-12-30

1. 目的 (Objective)

本レポートの目的は 2 つ.

- 信号処理論において学んだ，フーリエ変換・窓関数・高速フーリエ変換を線形代数の視点から捉え直すこと (理論).
- 離散フーリエ変換・高速フーリエ変換の計算量の差について記述する (アルゴリズム).

2. 理論 (Theory)

本実験で使用する基本原理について，線形代数的な観点から記述する.

2.1 フーリエ変換における基底変換

長さ N の任意の離散信号 $\mathbf{x} \in \mathbb{C}^N$ は,「時刻 $t = k$ のとき値が 1」であるインパルス信号（標準基底）$\mathbf{e}_k$ を用いた線形結合で表せる.

$$\mathbf{x} = x[0]\mathbf{e}_0 + x[1]\mathbf{e}_1 + \cdots + x[N-1]\mathbf{e}_{N-1}$$

フーリエ変換の本質は，この「時間ごとのインパルス」という基底を,「周波数ごとの正弦波ベクトル（複素正弦波)」という別の直交基底に取り替える座標変換である.

信号 $\mathbf{x}$ の中に特定の周波数成分（基底ベクトル $\mathbf{u}_k$）がどれだけ含まれているかを知るためには，信号と基底の内積を取ればよい．複素ベクトル空間におけるエルミート内積を以下で定義する.

$$\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{n=0}^{N-1} a[n] \cdot \overline{b[n]}$$

もし基底ベクトル $\mathbf{u}_k$ が正規化（長さ 1）されていれば，信号 $\mathbf{x}$ との内積は，$\mathbf{x}$ の $\mathbf{u}_k$ 方向への射影成分（影の長さ）を与える.

$$c_k = \langle \mathbf{x}, \mathbf{u}_k \rangle$$

連続時間におけるフーリエ係数 $\int x(t)e^{-i\omega t}dt$ は，無限次元ベクトル $x(t)$ と基底ベクトル $e^{i\omega t}$ の内積操作そのものである．

2.2 リーマン和による連続への移行

離散和 $\sum$ から積分 $\int$ への移行において，微小時間 dt は内積の定義において不可欠な役割を担う．無限次元の関数空間において，単に $f(t)\cdot\overline{g(t)}$ を無限個足し合わせれば値は発散しうるためである．

区間 T を N 等分した幅を $\Delta t = T/N$ とする．連続信号の内積は，この微小幅 Δt を重みとしたリーマン和の極限として定義される．

$$\langle f, g\rangle = \lim_{N\to\infty}\sum_{n=0}^{N-1} f(t_n)\cdot\overline{g(t_n)}\cdot\Delta t = \int_0^T f(t)\cdot\overline{g(t)}dt$$

2.3 三角関数の直交性と正規化

インパルス基底を三角関数系 $\{1, \cos t, \sin t, \cos 2t, ...\}$ に変換する場合，基底の「直交性」と「ノルム（長さ）」を確認する必要がある．三角関数系は区間 $[0, 2\pi]$ において互いに直交する．

2.1 節では基底ベクトルが正規化済みであると仮定したが，一般に基底ベクトル $\mathbf{v}$ の長さは 1 とは限らない．その場合，射影成分（フーリエ係数）を求めるには，基底自身の長さの二乗（ノルムの二乗）で除算する必要がある．

$$c_k = \frac{\langle \mathbf{x}, \mathbf{v}\rangle}{\langle \mathbf{v}, \mathbf{v}\rangle} = \frac{\langle \mathbf{x}, \mathbf{v}\rangle}{\|\mathbf{v}\|^2}$$

交流成分 $\cos(nt), \sin(nt)$ の区間 $[0, 2\pi]$ におけるノルムの二乗は以下のようになる．

$$\|\cos(nt)\|^2 = \int_0^{2\pi}\cos^2(nt)dt = \pi \quad (\sin(nt)\text{も同様})$$

よって長さは $\sqrt{\pi}$ である．一方，直流成分（1）のノルムの二乗は，

$$\|1\|^2 = \int_0^{2\pi} 1^2 dt = 2\pi$$

となり，長さは $\sqrt{2\pi}$ である．これらを射影公式に代入することで，実フーリエ級数の係数公式が導かれる．

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos(nt)dt \quad (n \geq 1)$$

$$a_0 = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)\cdot 1 dt$$

2.4 オイラーの公式と複素指数関数基底

実フーリエ級数では $\cos, \sin$ という 2 種類の基底を管理する必要があるが，オイラーの公式 $e^{i\theta} = \cos\theta + i\sin\theta$ を用いることで，これらを「複素平面上を回転するベクトル」として統合できる．

基底を $u_n(t) = e^{int}$ とすれば，区間 $[0, 2\pi]$ におけるノルムの二乗は，

$$\|e^{int}\|^2 = \int_0^{2\pi} e^{int}\overline{e^{int}}dt = \int_0^{2\pi} 1dt = 2\pi$$

となり，全ての整数 n において一定値 2π をとる．これにより，複素フーリエ係数 c_n の公式は統一的に記述できる．

$$c_n = \frac{1}{2\pi}\int_0^{2\pi} f(t)e^{-int}dt$$

負の周波数 n まで拡張することで，$\cos(nt) = \frac{e^{int}+e^{-int}}{2}$ のように，正負の回転の合成としてすべての実信号を表現可能となる．

2.5 離散化と直交性

連続信号 $x(t)$ を N 点にサンプリングしたベクトル $\mathbf{x} = [x[0], \dots, x[N-1]]$ を考える．同様に基底関数 $e^{i\omega t}$ も離散化する．周波数 k の基底ベクトル $\mathbf{w}_k$ の第 n 成分は以下のように書ける．

$$w_k[n] = e^{i\frac{2\pi}{N}kn} \quad (n = 0, 1, \dots, N-1)$$

異なる周波数 k と l $(k \neq l)$ を持つ 2 つの基底ベクトルの内積を計算する．

$$\langle \mathbf{w}_k, \mathbf{w}_l \rangle = \sum_{n=0}^{N-1} e^{i\frac{2\pi}{N}kn} \cdot e^{-i\frac{2\pi}{N}ln} = \sum_{n=0}^{N-1} e^{i\frac{2\pi}{N}(k-l)n}$$

ここで $r = e^{i\frac{2\pi}{N}(k-l)}$ と置くと，この式は初項 1，公比 r の等比級数の和となる．$k \neq l$ のとき $r \neq 1$ であるが，

$$r^N = \left(e^{i\frac{2\pi}{N}(k-l)}\right)^N = e^{i2\pi(k-l)} = 1$$

となるため，等比級数の和の公式より，

$$\text{Sum} = \frac{1-r^N}{1-r} = \frac{1-1}{1-r} = 0$$

よって，離散化しても異なる周波数の基底ベクトルは互いに直交することが示された．

2.6 DFT 行列

信号ベクトル $\mathbf{x}$ を直交基底 $\mathbf{w}_k$ の線形結合で表した際の係数（スペクトル）を $\mathbf{X}[k]$ とする．

$$\mathbf{X}[k] = \langle \mathbf{x}, \mathbf{w}_k \rangle = \sum_{n=0}^{N-1} x[n] e^{-i\frac{2\pi}{N}kn}$$

これを行列形式で記述すると，DFT 行列 $\mathbf{F}$ が導かれる．回転因子 $W_N = e^{-i\frac{2\pi}{N}}$ を用いると，

$$\begin{bmatrix} X[0] \\ X[1] \\ \vdots \\ X[N-1] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & W_N & W_N^2 & \dots & W_N^{N-1} \\ 1 & W_N^2 & W_N^4 & \dots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & \dots & W_N^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[N-1] \end{bmatrix}$$

この行列 $\mathbf{F}$ は，信号空間を時間軸から周波数軸へ回転させる座標変換行列である．$\mathbf{F}$ はヴァンデルモンド行列の一種であり，かつ定数倍の違いを除いてユニタリ行列の性質を持つ．

2.7 スペクトル漏れと窓関数

DFT は信号が周期的である（$x[0]$ と $x[N-1]$ が滑らかに接続される）ことを暗に仮定している．しかし，実際の信号を有限区間で切り出すと，始端と終端に不連続なズレが生じることが多い．DFT はこの不連続性を表現するために，本来存在しない高周波成分まで動員して波形を合成しようとする．これが目的の周波数以外にエネルギーが拡散する「スペクトル漏れ」である．

これを抑制するために「窓関数」を用いる．ハニング窓（Hanning Window）は，信号の両端を滑らかにゼロに減衰させることで不連続性を解消する．

$$w[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right)$$

これにより，振幅分解能は多少犠牲になるが，スペクトル漏れによる信号の汚染を防ぐことができる．

2.8 逆変換（IDFT）とユニタリ性

周波数領域 $\mathbf{X}$ から時間領域 $\mathbf{x}$ を復元する逆変換は，行列 $\mathbf{F}$ の逆行列を用いて記述される．

$$\mathbf{x} = \mathbf{F}^{-1}\mathbf{X}$$

通常，逆行列の計算は計算コストが高いが，DFT 行列の持つユニタリ性（に近い性質）により，以下の関係が成り立つ．行列 $\mathbf{F}$ とそのエルミート共役（随伴行列）$\mathbf{F}^\dagger$ の積を考える．

$$(\mathbf{F}^\dagger \mathbf{F})_{mn} = \sum_{k=0}^{N-1} e^{i\frac{2\pi}{N}k(m-n)} = N\delta_{mn}$$

ここで δ_{mn} はクロネッカーのデルタである．これより $\mathbf{F}^\dagger\mathbf{F} = N\mathbf{I}$（$\mathbf{I}$ は単位行列）となるため，

$$\mathbf{F}^{-1} = \frac{1}{N}\mathbf{F}^\dagger$$

すなわち，逆 DFT 行列は，元の行列の複素共役を取り，係数 $1/N$ を掛けるだけで求まる．これが IDFT の計算原理である．

3. 方法 (Methods)

3.1 実験環境・使用機器 (Environment & Equipment)

- Language: Rust (1.92.0)
- Libraries: Standard Library only (std::ops, std::f64)
- Visualization: Python (Matplotlib)

3.2 実装の設計

信号を複素数ベクトル構造体 `Signal` として定義し，演算子オーバーロードにより内積及びアダマール積を実装した．

3.3 使用したコード

- Rust により記述されたメインプログラム

```rust
use std::ops::{Add, Mul};
use std::f64::consts::{PI};

fn main() {
    // println!("Hello, world!");

    println!("k,mag_rect,mag_hann");

    // 信号生成：周波数 2.5(非整数)->スペクトル漏れが発生するはず
    let mut sig_data = Vec::new();
    let n = 32;
    for i in 0..n {
        let t = i as f64;
        sig_data.push(Complex::new((2.0 * PI * 2.5 * t / n as f64).sin(), 0.0)); // n as f64 を 8 とす
    }
    let raw_signal = Signal::new(sig_data);

    // 窓なし->DFT
    let spec_rect = raw_signal.dft();
```

```
    // ハニング窓を適用->DFT
    let hanning_window = Signal::create_hanning(n);
    let windowed_signal = raw_signal.clone() * hanning_window;
    let spec_hann = windowed_signal.dft();

    // データ出力
    for k in 0..n {
        let mag_r = spec_rect.data[k].norm();
        let mag_h = spec_hann.data[k].norm() * 2.0; //2.0はハニング窓による減衰の補正
        println!("{},{:.6},{:.6}", k, mag_r, mag_h);
    }
}

#[derive(Debug, Clone, Copy)]
struct Complex {
    re: f64,
    im: f64,
}

impl Mul for Complex {
    type Output = Complex;

    fn mul(self, other: Complex) -> Complex {
        Complex {
            re: self.re * other.re - self.im * other.im,
            im: self.re * other.im + self.im * other.re,
        }
    }
}

impl Add for Complex {
    type Output = Complex;
    fn add(self, other: Complex) -> Complex {
        Complex {
            re: self.re + other.re,
            im: self.im + other.im,
        }
    }
}

impl Complex {
    fn new(re: f64, im: f64) -> Complex {
        Complex {
            re,
```

```
            im,
        }
    }

    fn conjugate(&self) -> Complex {
        Complex {
            re: self.re,
            im: -1.0 * self.im,
        }
    }

    fn norm(&self) -> f64 {
        (self.re * self.re + self.im * self.im).sqrt()
    }
}

#[derive(Debug, Clone)]
struct Signal {
    data: Vec<Complex>,
}

// アダマール積
impl Mul for Signal {
    type Output = Signal;

    fn mul(self, other: Signal) -> Signal {
        if self.data.len() != other.data.len() {
            panic!("Vector dimension mismatch in Hadamard product");
        }

        let mut new_data = Vec::with_capacity(self.data.len());
        for i in 0..self.data.len() {
            new_data.push(self.data[i] * other.data[i]);
        }
        Signal::new(new_data)
    }
}

impl Signal {
    fn new(data: Vec<Complex>) -> Signal {
        Signal {
            data,
        }
    }
```

```rust
fn inner_product(&self, other: &Signal) -> Result<Complex, String> {
    if self.data.len() != other.data.len() {
       return Err("Vector sizes differ".to_string());
    }

    let mut sum = Complex::new(0.0, 0.0);

    for i in 0..self.data.len() {
        let term = self.data[i] * other.data[i].conjugate();
        sum = sum + term;
    }
    Ok(sum)
}

fn basis(n: usize, k: usize) -> Signal {
    // 周波数 k の基底ベクトル e_k を生成
    let mut data = Vec::with_capacity(n);
    for i in 0..n {
        let theta = 2.0 * PI * (k as f64) * (i as f64) / (n as f64);
        data.push(Complex::new(theta.cos(), theta.sin()));
    }
    Signal::new(data)
}

// DFT: X[k] = <x, e_k>
// 信号 x を基底 e_k に射影する
fn dft(&self) -> Signal {
    let n = self.data.len();
    let mut spectrum_data = Vec::with_capacity(n);

    for k in 0..n {
        let e_k = Signal::basis(n, k);
        let x_k = self.inner_product(&e_k).unwrap(); // <x, e_k>
        spectrum_data.push(x_k);
    }

    Signal::new(spectrum_data)
}

// ハニング窓の生成
// w[n] = 0.5 - 0.5 * cos(2 π n / (N-1))
fn create_hanning(n: usize) -> Signal {
    let mut hanning = Vec::with_capacity(n);
```

```
        for i in 0..n {
            hanning.push(Complex::new(0.5 - 0.5 * (2.0 * PI * i as f64 / (n as f64 - 1.0)).cos(), 0.0
        }
        Signal::new(hanning)
    }
}
```

- Python により記述された，出力結果をビジュアル化するためのプログラム

```
import sys
import pandas as pd
import matplotlib.pyplot as plt

# 標準入力からデータを読み込み
try:
    df = pd.read_csv(sys.stdin)
except Exception as e:
    print("Error reading CSV:", e)
    sys.exit(1)

# プロット設定
plt.figure(figsize=(10, 6))

# 矩形窓 (Rectangular) - 青色・破線
plt.plot(df['k'], df['mag_rect'], label='Rectangular (No Window)',
         marker='o', linestyle='--', color='blue', alpha=0.6)

# ハニング窓 (Hanning) - 赤色・実線
plt.plot(df['k'], df['mag_hann'], label='Hanning Window',
         marker='s', linestyle='-', color='red', linewidth=2)

# グラフ装飾
plt.title('Spectrum Leakage Analysis: Rectangular vs Hanning')
plt.xlabel('Frequency Index k')
plt.ylabel('Magnitude (Linear)')
plt.grid(True)
plt.legend()
plt.xticks(df['k'][::2]) # 目盛りを適度に間引く

# 保存または表示
plt.savefig('spectrum_analysis.png')
print("Graph saved to 'spectrum_analysis.png'")
# plt.show()
```

- Rust により記述された逆 DFT の精度確認用プログラム

```rust
use std::ops::{Add, Mul, Sub}; // Sub を追加
use std::f64::consts::PI;

fn main() {
    // 1. 信号生成 (Original Signal)
    let n = 8; // 検証用なので少なめで OK
    let mut sig_data = Vec::new();
    for i in 0..n {
        let t = i as f64;
        // 複雑な信号を作る: 2.5Hz + 1.0Hz の合成波
        let val = (2.0 * PI * 2.5 * t / n as f64).sin() + 0.5 * (2.0 * PI * 1.0 * t / n as f64).cos()
        sig_data.push(Complex::new(val, 0.0));
    }
    let original = Signal::new(sig_data);

    // 2. DFT (Analysis)
    // 時間領域 -> 周波数領域
    let spectrum = original.dft();

    // 3. IDFT (Synthesis)
    // 周波数領域 -> 時間領域
    // スペクトルから元の波形を復元する
    let reconstructed = spectrum.idft();

    // 4. 検証 (Verification)
    // 元の信号と復元信号の差（誤差）を確認する
    println!("Index | Original (Re) | Recon (Re) | Error");
    println!("------+---------------+------------+-------------");

    let mut max_error = 0.0;
    for i in 0..n {
        let orig_re = original.data[i].re;
        let recon_re = reconstructed.data[i].re;
        let error = (orig_re - recon_re).abs();

        if error > max_error {
            max_error = error;
        }

        println!("{:5} | {:13.8} | {:10.8} | {:.4e}", i, orig_re, recon_re, error);
    }

    println!("-------------------------------------------------");
    println!("Max Reconstruction Error: {:.4e}", max_error);
```

```
    if max_error < 1e-10 {
        println!("Result: SUCCESS. F^(-1) * F = I is proved.");
    } else {
        println!("Result: FAILED. Check the implementation.");
    }
}

// --- 構造体定義 ---

#[derive(Debug, Clone, Copy)]
struct Complex {
    re: f64,
    im: f64,
}

impl Complex {
    fn new(re: f64, im: f64) -> Complex {
        Complex { re, im }
    }
    fn conjugate(&self) -> Complex {
        Complex { re: self.re, im: -1.0 * self.im }
    }
    fn norm(&self) -> f64 {
        (self.re * self.re + self.im * self.im).sqrt()
    }
}

impl Mul for Complex {
    type Output = Complex;
    fn mul(self, other: Complex) -> Complex {
        Complex {
            re: self.re * other.re - self.im * other.im,
            im: self.re * other.im + self.im * other.re,
        }
    }
}

impl Add for Complex {
    type Output = Complex;
    fn add(self, other: Complex) -> Complex {
        Complex {
            re: self.re + other.re,
            im: self.im + other.im,
```

```
        }
    }
}

// Sub トレイトの実装（誤差計算用）
impl Sub for Complex {
    type Output = Complex;
    fn sub(self, other: Complex) -> Complex {
        Complex {
            re: self.re - other.re,
            im: self.im - other.im,
        }
    }
}

#[derive(Debug, Clone)]
struct Signal {
    data: Vec<Complex>,
}

impl Signal {
    fn new(data: Vec<Complex>) -> Signal {
        Signal { data }
    }

    fn inner_product(&self, other: &Signal) -> Result<Complex, String> {
        let mut sum = Complex::new(0.0, 0.0);
        for i in 0..self.data.len() {
            sum = sum + self.data[i] * other.data[i].conjugate();
        }
        Ok(sum)
    }

    fn basis(n: usize, k: usize) -> Signal {
        let mut data = Vec::with_capacity(n);
        for i in 0..n {
            let theta = 2.0 * PI * (k as f64) * (i as f64) / (n as f64);
            data.push(Complex::new(theta.cos(), theta.sin()));
        }
        Signal::new(data)
    }

    fn dft(&self) -> Signal {
        let n = self.data.len();
```

```
        let mut spectrum_data = Vec::with_capacity(n);
        for k in 0..n {
            let e_k = Signal::basis(n, k);
            let x_k = self.inner_product(&e_k).unwrap();
            spectrum_data.push(x_k);
        }
        Signal::new(spectrum_data)
    }

    // IDFT: x[n] = (1/N) * sum( X[k] * e_k )
    // スペクトル成分を重みとして，基底ベクトルを足し合わせる
    fn idft(&self) -> Signal {
        let n = self.data.len();
        // ゼロで初期化した信号ベクトルを作成
        let mut reconstructed_data = vec![Complex::new(0.0, 0.0); n];

        for k in 0..n {
            let X_k = self.data[k];        // スペクトル係数
            let e_k = Signal::basis(n, k); // 基底ベクトル（波）

            // 線形結合: vec += X_k * e_k
            for i in 0..n {
                // X_k * e_k[i]
                let term = X_k * e_k.data[i];
                reconstructed_data[i] = reconstructed_data[i] + term;
            }
        }

        // 最後に 1/N で正規化
        for i in 0..n {
            reconstructed_data[i].re /= n as f64;
            reconstructed_data[i].im /= n as f64;
        }

        Signal::new(reconstructed_data)
    }
}

// 必要ないが一応残しておく
impl Mul for Signal {
    type Output = Signal;
    fn mul(self, other: Signal) -> Signal {
         let mut new_data = Vec::with_capacity(self.data.len());
        for i in 0..self.data.len() {
```

```
            new_data.push(self.data[i] * other.data[i]);
        }
        Signal::new(new_data)
    }
}
```

4. 結果 (Results)

4.1 メインプログラムの出力結果

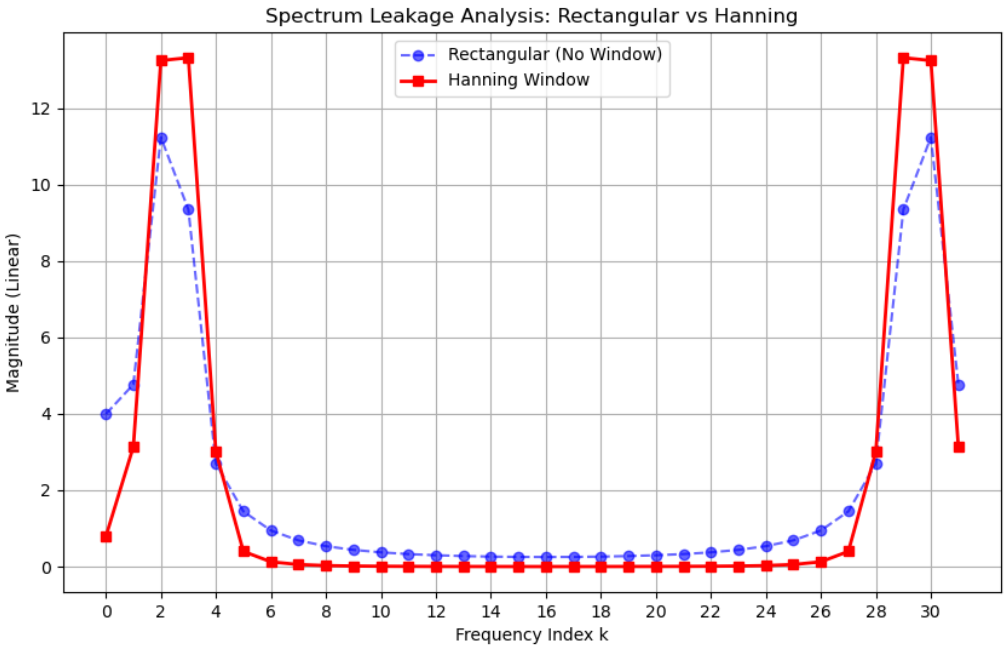

図 1　Result1

また，csv ファイルに結果を出力した（可読性のために空白を追加した）.

```
k, mag_rect, mag_hann
0, 3.992224, 0.787802
1, 4.768133, 3.137013
2, 11.234895,13.242323
3, 9.343534, 13.317712
4, 2.696554, 3.007223
5, 1.444447, 0.397724
6, 0.944233, 0.125344
7, 0.686336, 0.053566
8, 0.534511, 0.026927
9, 0.437690, 0.014963
10,0.372762, 0.008877
11,0.327930, 0.005489
```

```
12,0.296657, 0.003460
13,0.275125, 0.002163
14,0.261046, 0.001271
15,0.253071, 0.000605
16,0.250487, 0.000163
17,0.253071, 0.000605
18,0.261046, 0.001271
19,0.275125, 0.002163
20,0.296657, 0.003460
21,0.327930, 0.005489
22,0.372762, 0.008877
23,0.437690, 0.014963
24,0.534511, 0.026927
25,0.686336, 0.053566
26,0.944233, 0.125344
27,1.444447, 0.397724
28,2.696554, 3.007223
29,9.343534, 13.317712
30,11.234895,13.242323
31,4.768133, 3.137013
```

4.2 逆 DFT プログラムの出力結果

```
Index | Original (Re) | Recon (Re) | Error
------+---------------+------------+-------------
    0 |    0.50000000 | 0.50000000 | 9.9920e-16
    1 |    1.27743292 | 1.27743292 | 0.0000e0
    2 |   -0.70710678 | -0.70710678 | 5.5511e-16
    3 |   -0.73623682 | -0.73623682 | 6.6613e-16
    4 |    0.50000000 | 0.50000000 | 7.7716e-16
    5 |   -0.73623682 | -0.73623682 | 1.1102e-16
    6 |   -0.70710678 | -0.70710678 | 2.2204e-16
    7 |    1.27743292 | 1.27743292 | 4.4409e-16
-----------------------------------------------
Max Reconstruction Error: 9.9920e-16
Result: SUCCESS. F^(-1) * F = I is proved.
```

5. 考察 (Discussion)

結果より，以下の知見が得られた.

5.1 矩形窓とハニング窓における結果の違い

4.1 節の結果を見ると，理論通りの結果が得られていることが即座に理解できる．

ハニング窓適用前，すなわち矩形窓のグラフでは，周波数が非整数のために「スペクトル漏れ」が発生していることが分かる．ピークにおけるエネルギーは全周波数帯に散り，本来関係のない中心 $k = 16$ においても，0.25 程度エネルギーが残ってしまっている．

それに対して，ハニング窓適用後のグラフでは，目的の周波数付近で即座にエネルギーが小さくなっている．そのためピーク帯におけるエネルギーは，矩形窓のピークにおける値よりも大きくなっている（補整を考慮してもエネルギーの散らばらなさによる影響が優勢である）．また，理論通り，ピークの山が潰れていることが分かる．

以上より，ハニング窓の適用は理論通り，周波数分解能を低下させるが，スペクトル漏れによる信号の汚染を防ぐことができていることが分かった．

5.2 逆 DFT 行列による逆行の正確性

4.2 節の結果には，実際に与えた信号の実部と，逆 DFT 行列により求められた逆算した信号，そしてその差が記載されている．差の最大誤差を見ると `9.9920e-16` であり，この 10^{-16} という数値はコンピュータが表現できるゼロの限界値「マシンイプシロン」である．すなわち，与えた信号ベクトルと，逆算した信号ベクトルは限りなく近い．

以上より，導出した逆 DFT 行列は正しく機能していることが分かった．

5.3 計算量の観点: DFT と FFT

本実装では，定義通りのりさんフーリエ変換（DFT）を採用した．

DFT は N 次元ベクトルに対する $N \times N$ 行列の乗算であるため，その計算量オーダーは $O(N^2)$ となる．

$$\mathbf{X} = F_N \mathbf{x} \quad \Rightarrow \quad N^2 \text{ complex multiplications}$$

一方，実用上で広く用いられる高速フーリエ変換（FFT）は，行列 F_N の対称性を利用して，再帰的に計算を行うアルゴリズムであり，計算量は $O(N \log N)$ に削減される．

$$\frac{O(N^2)}{O(N \log N)} = \frac{N}{\log N}$$

$N = 1024$ の場合，この比は約 100 倍となるが，本実験の様な小規模な N においては，行列演算としての構造的明瞭さを優先し，DFT の実装を採用した．

6. 結論 (Conclusion)

以上の結果より，本実験の目的は達成されたと結論づけられる．