Fourier Transform: Numerical Implementation

CH481

October 8, 2025

1 Introduction

The Fourier Transform (FT) is a fundamental tool that converts a time-domain signal into its frequency-domain representation. This allows us to understand the spectral content of signals.

2 Numerical Fourier Transform using Integration

In practice, the continuous-time integral cannot always be computed analytically. We can use **numerical integration** in Python to approximate the Fourier transform:

$$F(\omega) \approx \sum_{n=0}^{N-1} f(t_n) e^{-i\omega t_n} \Delta t \tag{1}$$

where t_n are uniformly spaced time points and Δt is the time step.

2.1 Procedure in Python

1. Define the time grid t_n covering the duration of the signal.
2. Evaluate the signal $f(t_n)$ at all time points.
3. Choose a range of angular frequencies ω_k for which the Fourier transform is desired.
4. Compute the integral numerically using methods such as the `trapezoidal rule`:

$$F(\omega_k) = \texttt{np.trapz}(f(t_n) \cdot e^{-i\omega_k t_n}, t_n)$$

5. Normalize and plot the magnitude $|F(\omega_k)|$ and optionally the phase $\phi(\omega_k)$.

3 Code Explanation and Steps

3.1 1. Importing Libraries

We import the necessary Python libraries:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from IPython.display import HTML
import sympy as sp
```

Listing 1: Importing required Python libraries

- `numpy`: For numerical operations and arrays.
- `matplotlib.pyplot`: For plotting.
- `FuncAnimation`: For animation.
- `IPython.display.HTML`: To render animation in Jupyter Notebook.
- `sympy`: To parse symbolic user input safely.

—

3.2 2. User Input for Function

The user can input any function of time t, e.g., sine, Gaussian wave packet:

```python
print("Enter any time-dependent function using variable 't'.")
print("Examples:")
print("  sin(2*pi*5*t)")
print("  exp(-t**2)*cos(2*pi*5*t)")
print("  exp(-t**2/(2*0.1**2))*cos(2*pi*5*t)  # Gaussian wave packet")
func_str = input("Enter function f(t): ")

# Parse the function safely
t_sym = sp.Symbol('t', real=True)
expr = sp.sympify(func_str)
f = sp.lambdify(t_sym, expr, modules=["numpy"])
```

Listing 2: User input and parsing

Explanation: - `sympify` converts string input to a symbolic expression. - `lambdify` converts the symbolic expression to a numerical function for evaluation on arrays.

—

3.3 3. Define Time and Frequency Grids

We define uniform grids for time and angular frequency:

```python
t = np.linspace(-3, 3, 200)       # time grid
omega = np.linspace(-100, 100, 200)  # angular frequency grid
freq = omega / (2*np.pi)           # frequency in Hz
```

Listing 3: Time and frequency grids

Explanation: - `t` is the time axis where the function is evaluated. - `omega` is the angular frequency axis for FT calculation. - `freq` converts angular frequency to Hz for plotting.

—

3.4 4. Numerical Fourier Transform via Integration

We perform the Fourier transform numerically using the trapezoidal rule:

```
def numerical_ft(f_t, t, omega):
    F_w = np.zeros_like(omega, dtype=complex)
    for i, w in enumerate(omega):
        integrand = f_t * np.exp(-1j * w * t)
        F_w[i] = np.trapz(integrand, t)
    return F_w
```

Listing 4: Numerical Fourier Transform function

Explanation: - For each frequency ω_i, multiply the function by $e^{-i\omega_i t}$ and integrate using `np.trapz`. - Returns a complex-valued array representing the Fourier transform.

—

3.5 5. Animation Setup

Create the figure and axes for time and frequency domain plots:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
plt.tight_layout(pad=3)

# Time domain plot
line1, = ax1.plot([], [], color='navy', lw=2)
ax1.set_xlim(t[0], t[-1])
ax1.set_ylim(-1.2, 1.2)
ax1.set_title("Time Domain Signal")
ax1.set_xlabel("Time (s)")
ax1.set_ylabel("f(t)")
ax1.grid(True)

# Frequency domain plot
line2, = ax2.plot([], [], color='darkred', lw=2)
ax2.set_xlim(freq[0], freq[-1])
ax2.set_ylim(0, 1.1)
ax2.set_title("Frequency Domain: |F(w)|")
ax2.set_xlabel("Frequency (Hz)")
ax2.set_ylabel("Magnitude")
ax2.grid(True)
```

Listing 5: Animation setup

Explanation: - Creates two subplots: one for the time-domain signal, one for the magnitude of its Fourier transform. - Axis limits and grid are set for clarity.

—

3.6 6. Precompute Frames for Animation

We scale the time axis to show dynamic stretching/compression and precompute all frames:

```
scales = np.linspace(0.5, 2.0, 60)  # scaling factors
frames_data = []

for scale in scales:
    t_scaled = t / scale
    f_t = f(t_scaled)
    F_w = numerical_ft(f_t, t, omega)
    # Normalize for plotting
    frames_data.append((f_t / np.max(np.abs(f_t)), np.abs(F_w) /
       np.max(np.abs(F_w))))
```

Listing 6: Precompute animation frames

Explanation: - The `scales` array stretches or compresses the time axis. - Each frame contains the normalized time-domain signal and Fourier magnitude spectrum.

—

3.7 7. Define Animation Functions

```
def init():
    line1.set_data([], [])
    line2.set_data([], [])
    return line1, line2

def update(frame):
    f_t_norm, F_w_norm = frames_data[frame]
    line1.set_data(t, f_t_norm)
    line2.set_data(freq, F_w_norm)
    ax1.set_title(f"Time Domain (scale = {scales[frame]:.2f})")
    return line1, line2
```

Listing 7: Animation functions

Explanation: - `init` clears the lines. - `update` sets the data for each frame of the animation.

—

3.8 8. Run the Animation

```
ani = FuncAnimation(fig, update, frames=len(scales),
                    init_func=init, blit=True, interval=100)

HTML(ani.to_jshtml())
```

Listing 8: Create and display animation

Explanation: - `FuncAnimation` generates the animation. - `to_jshtml()` embeds the animation directly in a Jupyter Notebook.