Introduction to Signal Processing with FFT

Jupyter Notebook Exercises in Python - with Worked Solutions

Prepared by Mr. J. Heystek Grobler

Contents

1 Introduction

The Fast Fourier Transform (FFT) is a powerful tool in signal processing. It allows us to analyze signals not only in the time domain but also in the frequency domain.

In this document, we explore:

- How to generate signals
- How to apply the FFT
- How to interpret frequency spectra
- The effects of noise
- Simple frequency-domain filtering
- A radio-astronomy application of the FFT

Each section includes an explanation, example Python code, an exercise, and a worked solution (memo) so that the material can be studied independently or used as a self-contained practical.

2 Generating a Simple Signal

We begin with a signal composed of two sine waves at 5 Hz and 20 Hz.

```
import numpy as np
import matplotlib.pyplot as plt

Fs = 200             # Sampling frequency
T = 1/Fs
t = np.arange(0, 1, T)

f1, f2 = 5, 20
signal = np.sin(2*np.pi*f1*t) + 0.5*np.sin(2*np.pi*f2*t)

plt.plot(t, signal)
plt.title("Time-domain Signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid(True)
plt.show()
```

3 Applying the FFT

The FFT transforms the signal into the frequency domain.

```
n = len(signal)
fft_result = np.fft.fft(signal)
fft_freq = np.fft.fftfreq(n, T)

mask = fft_freq >= 0
fft_result = np.abs(fft_result[mask]) * (2/n)
fft_freq = fft_freq[mask]

plt.stem(fft_freq, fft_result, basefmt=" ")
plt.title("Frequency Spectrum")
plt.xlabel("Frequency (Hz)")
```

```
plt.ylabel("Magnitude")
plt.grid(True)
plt.show()
```

Observation

You should see peaks at 5 Hz and 20 Hz.

Exercise 1

Task: Add a sine wave at 50 Hz with amplitude 0.3.
Expected result: The frequency spectrum should now show three peaks at 5 Hz, 20 Hz, and 50 Hz.

Memo — Exercise 1 Solution

```
import numpy as np
import matplotlib.pyplot as plt

# Sampling parameters
Fs = 200  # Sampling frequency
T = 1/Fs
t = np.arange(0, 1, T)

# Frequencies and amplitudes
f1, f2, f3 = 5, 20, 50
signal = np.sin(2*np.pi*f1*t) + 0.5*np.sin(2*np.pi*f2*t) + 0.3*
    np.sin(2*np.pi*f3*t)

# Plot new signal
plt.figure(figsize=(10,4))
plt.plot(t, signal)
plt.title("Time-domain Signal with 5 Hz, 20 Hz, and 50 Hz
    Components")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid(True)
plt.show()

# FFT
n = len(signal)
fft_result = np.fft.fft(signal)
fft_freq = np.fft.fftfreq(n, T)

mask = fft_freq >= 0
fft_result = np.abs(fft_result[mask]) * (2/n)
fft_freq = fft_freq[mask]

# Plot frequency spectrum
plt.figure(figsize=(10,4))
plt.stem(fft_freq, fft_result, basefmt=" ")
plt.title("Frequency Spectrum")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.grid(True)
plt.show()
```

Result: adding the 50 Hz term with amplitude 0.3 introduces a third, smaller peak in the spectrum at 50 Hz, alongside the original peaks at 5 Hz and 20 Hz (whose relative heights follow their amplitudes 1, 0.5 and 0.3).

4 Noise and the FFT

Real-world signals often contain noise.

```
noisy_signal = signal + 0.5*np.random.randn(len(t))

fft_noisy = np.fft.fft(noisy_signal)
fft_noisy = np.abs(fft_noisy[mask]) * (2/n)

plt.stem(fft_freq, fft_noisy, basefmt=" ")
plt.title("Noisy Frequency Spectrum")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.grid(True)
plt.show()
```

Exercise 2

Task: Try different noise strengths (0.2, 0.5, 1.0).
Expected result: Peaks at 5 Hz and 20 Hz become harder to distinguish under heavy noise, but they are still visible.

Memo — Exercise 2 Solution

```
import numpy as np
import matplotlib.pyplot as plt

# Sampling setup
Fs = 200
T = 1/Fs
t = np.arange(0, 1, T)

# Base clean signal: 5 Hz + 20 Hz
f1, f2 = 5, 20
signal = np.sin(2*np.pi*f1*t) + 0.5*np.sin(2*np.pi*f2*t)

# Try different noise strengths
noise_levels = [0.2, 0.5, 1.0]

for noise_amp in noise_levels:
    noisy_signal = signal + noise_amp * np.random.randn(len(t))

    # FFT
    n = len(noisy_signal)
    fft_result = np.fft.fft(noisy_signal)
    fft_freq = np.fft.fftfreq(n, T)

    mask = fft_freq >= 0
    fft_result = np.abs(fft_result[mask]) * (2/n)
    fft_freq = fft_freq[mask]

```

```
    # Plot
    plt.figure(figsize=(10,4))
    plt.stem(fft_freq, fft_result, basefmt=" ")
    plt.title(f"Frequency Spectrum with Noise Amplitude = {
        noise_amp}")
    plt.xlabel("Frequency (Hz)")
    plt.ylabel("Magnitude")
    plt.grid(True)
    plt.show()
```

Result: at noise amplitude 0.2 the 5 Hz and 20 Hz peaks remain sharp and clearly dominant; at 0.5 the noise floor rises noticeably but the two peaks are still the tallest features; at 1.0 the noise floor is comparable to the signal peaks, so the peaks are still visible but far less distinct.

5 Filtering

We can filter out high-frequency noise by removing FFT components above 30 Hz.

```
fft_noisy = np.fft.fft(noisy_signal)
freqs = np.fft.fftfreq(n, T)
cutoff = 30
mask = np.abs(freqs) <= cutoff
fft_filtered = fft_noisy * mask
reconstructed = np.fft.ifft(fft_filtered).real

plt.plot(t, noisy_signal, label="Noisy")
plt.plot(t, reconstructed, label="Filtered")
plt.legend()
plt.title("Signal Filtering with FFT")
plt.show()
```

Exercise 3

Task: Reconstruct the signal using inverse FFT after filtering out frequencies above 30 Hz.
Expected result: The reconstructed signal should closely resemble the original clean signal.

Memo — Exercise 3 Solution

```
import numpy as np
import matplotlib.pyplot as plt

# --- Sampling / signal setup ---
Fs = 200              # sampling frequency (Hz)
T = 1/Fs
t = np.arange(0, 1, T)
n = len(t)

# Clean signal (ground truth)
f1, f2 = 5, 20
clean = np.sin(2*np.pi*f1*t) + 0.5*np.sin(2*np.pi*f2*t)

# Add noise (choose strength)
```

```
noise_amp = 0.8
noisy = clean + noise_amp * np.random.randn(n)

# --- FFT of noisy signal ---
fft_noisy = np.fft.fft(noisy)
freqs = np.fft.fftfreq(n, T)

# --- Ideal low-pass filter in frequency domain: keep |f| <= 30 Hz ---
cutoff = 30.0
filter_mask = np.abs(freqs) <= cutoff

fft_filtered = fft_noisy * filter_mask  # zero-out components above cutoff

# --- Reconstruct signal with inverse FFT ---
reconstructed = np.fft.ifft(fft_filtered).real  # take real part (original is real)

# --- Plots: time domain (clean / noisy / reconstructed) ---
plt.figure(figsize=(12, 6))
plt.plot(t, clean, label='Clean (original)', linewidth=1)
plt.plot(t, noisy, label=f'Noisy (noise amp={noise_amp})', alpha=0.7)
plt.plot(t, reconstructed, label='Reconstructed (low-pass, 30 Hz)', linewidth=1.5)
plt.xlim(0, 0.5)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Time-domain: clean vs noisy vs reconstructed')
plt.legend()
plt.grid(True)
plt.show()

# --- Plots: frequency domain (magnitude) before & after ---
mag_noisy = np.abs(fft_noisy) * (2/n)
mag_filtered = np.abs(fft_filtered) * (2/n)
pos_mask = freqs >= 0

plt.figure(figsize=(12,4))
plt.stem(freqs[pos_mask], mag_noisy[pos_mask], basefmt=" ", label='Noisy spectrum')
plt.stem(freqs[pos_mask], mag_filtered[pos_mask], basefmt=" ",
         linefmt='C1-', markerfmt='C1o', label='Filtered spectrum')
plt.xlim(0, 100)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude (approx)')
plt.title('Frequency spectrum: before and after filtering')
plt.legend()
plt.grid(True)
plt.show()

# --- Simple quantitative metrics ---
mse_noisy = np.mean((clean - noisy)**2)
mse_recon = np.mean((clean - reconstructed)**2)
print(f"MSE (clean vs noisy):         {mse_noisy:.4f}")
```

```
print(f"MSE (clean vs reconstructed): {mse_recon:.4f}")
```

Result: the reconstructed signal tracks the clean 5 Hz + 20 Hz signal much more closely than the noisy version, since both signal components lie below the 30 Hz cutoff and most of the noise energy above 30 Hz is removed. The mean-squared error against the clean signal drops substantially after filtering, and a small amount of ringing may appear near sharp transitions – an artifact of the ideal (brick-wall) filter.

Observation

Filtering removes high-frequency noise but may introduce small artifacts (ringing). This is common with ideal FFT filters.

6 Summary

- The FFT converts signals from the time domain to the frequency domain.
- Peaks in the spectrum reveal which frequencies are present.
- Noise spreads across the frequency spectrum.
- Filtering in the frequency domain can recover useful signals.

7 Radio Astronomy Example

Radio telescopes detect extremely weak signals from space, often buried under thermal noise. One classic task is to identify narrow-band emission lines (e.g., the neutral hydrogen line at 1420 MHz) or periodic pulsar signals.

7.1 Simulated Observation

We simulate an observation where:

- The sampling frequency is 1000 Hz (simplified).
- The telescope is observing for 1 second.
- A weak sinusoidal signal at 60 Hz represents the astronomical source.
- Gaussian noise represents the receiver and sky noise.

```
import numpy as np
import matplotlib.pyplot as plt

Fs = 1000       # sampling frequency (Hz)
T = 1/Fs
t = np.arange(0, 1, T)
n = len(t)

# Simulated astronomical signal: weak sinusoid at 60 Hz
astro_freq = 60
astro_amp = 0.05    # very weak amplitude
astro_signal = astro_amp * np.sin(2*np.pi*astro_freq*t)

# Strong Gaussian noise
noise = np.random.randn(n)
```

```python

# Total observed signal
observed = astro_signal + noise

plt.plot(t[:200], observed[:200])
plt.title("Simulated Radio Telescope Time Series (first 200 samples)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.show()
```

7.2 Frequency Analysis with FFT

In the time series the signal is invisible. The FFT can reveal the hidden tone.

```python
fft_result = np.fft.fft(observed)
freqs = np.fft.fftfreq(n, T)
mask = freqs >= 0
fft_mag = np.abs(fft_result[mask]) * (2/n)
fft_freqs = freqs[mask]

plt.stem(fft_freqs, fft_mag, basefmt=" ")
plt.xlim(0, 200)   # zoom into first 200 Hz
plt.title("Frequency Spectrum of Radio Observation")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.grid(True)
plt.show()
```

Observation

Although the signal amplitude is only 0.05 compared to unit-variance noise, a clear spectral peak emerges at 60 Hz. This is how astronomers detect hidden periodicities and spectral lines in noisy data.

Exercise 4

1. Try reducing the amplitude to 0.02 or 0.01. How low can the signal be before you can no longer see it?
2. Increase the integration time (observe for 2 seconds instead of 1). How does this affect your ability to detect the peak?
3. Replace the sinusoid with two nearby tones (e.g., 60 Hz and 65 Hz). Can you resolve them both?

Memo — Exercise 4 Solution

```python
import numpy as np
import matplotlib.pyplot as plt

# ======================
# PARAMETERS
# ======================
Fs = 1000        # Sampling frequency (Hz)
T = 1/Fs
duration = 1.0  # Observation time in seconds (try 2.0 later)
t = np.arange(0, duration, T)
```

```python
n = len(t)

# Astronomical signal parameters
astro_freq1 = 60    # first tone (Hz)
astro_freq2 = 65    # second tone, e.g., 65 Hz
astro_amp = 1       # amplitude (try 0.02 or 0.01)

# ======================
# SIGNAL GENERATION
# ======================
astro_signal = astro_amp * np.sin(2*np.pi*astro_freq1*t)
if astro_freq2 is not None:
    astro_signal += astro_amp * np.sin(2*np.pi*astro_freq2*t)

# Add Gaussian noise (variance ~ 1)
noise = np.random.randn(n)
observed = astro_signal + noise

# ======================
# PLOT: TIME DOMAIN
# ======================
plt.figure(figsize=(10,4))
plt.plot(t[:200], observed[:200])
plt.title("Simulated Radio Telescope Time Series (first 200
    samples)")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid(True)
plt.show()

# ======================
# FFT ANALYSIS
# ======================
fft_result = np.fft.fft(observed)
freqs = np.fft.fftfreq(n, T)

mask = freqs >= 0
fft_mag = np.abs(fft_result[mask]) * (2/n)
fft_freqs = freqs[mask]

plt.figure(figsize=(10,4))
plt.stem(fft_freqs, fft_mag, basefmt=" ")
plt.xlim(0, 200)
plt.title("Frequency Spectrum of Radio Observation")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.grid(True)
plt.show()
```

Discussion of results:

1. **Lowering the amplitude (0.02, 0.01):** with unit-variance noise the peak becomes progressively harder to distinguish from the noise floor. At 0.05 it is clearly visible; by 0.01 it is usually indistinguishable from the surrounding noise fluctuations in a single 1-second observation.
2. **Longer integration (2 s instead of 1 s):** doubling the observation time doubles the number of samples n, which sharpens the frequency resolution ($\Delta f = 1/(nT)$)

and increases the height of the signal peak relative to the (roughly constant, spread-out) noise floor, making a weak tone easier to detect.

3. **Two tones (60 Hz and 65 Hz):** whether the two peaks can be resolved depends on the frequency resolution $\Delta f = 1/(nT)$. With $Fs = 1000$ Hz and a 1-second window, $\Delta f = 1$ Hz, which is fine enough to separate 60 Hz and 65 Hz into two distinct peaks; with a shorter observation window the resolution is coarser and the two tones may blur into one.

7.3 Key Concept

Radio astronomy often relies on the fact that integrating longer (more samples) reduces noise variance while the signal peak remains. This is why telescopes observe for minutes or hours: the FFT spectrum gradually reveals hidden signals.

A Installing Jupyter Notebook on Ubuntu

This appendix explains how to install Jupyter Notebook on Ubuntu and all the Python libraries needed to run the notebook and exercises above (NumPy, Matplotlib, etc.). Two installation paths are covered — **pip + venv** (lightweight, recommended for most users) and **Anaconda / Miniconda** (larger download, good if you also want conda's environment management). Pick one; you do not need both.

A.1 Step 0: Check your Ubuntu and Python version

```
lsb_release -a
python3 --version
```

Ubuntu 20.04+ ships with Python 3.8+, which is enough for this practical.

A.2 Option A — pip and a virtual environment (recommended)

Step 1: Update system packages

```
sudo apt update && sudo apt upgrade -y
```

Step 2: Install Python, pip, and venv

```
sudo apt install -y python3 python3-pip python3-venv
python3 --version
pip3 --version
```

Step 3: Create a virtual environment

```
mkdir -p ~/fft-notebook
cd ~/fft-notebook
python3 -m venv venv
source venv/bin/activate
```

Your prompt should now start with `(venv)`. Run `source venv/bin/activate` again each time you open a new terminal for this project (and `deactivate` to leave it).

Step 4: Upgrade pip inside the environment

```
pip install --upgrade pip
```

Step 5: Install Jupyter Notebook and required libraries

```
pip install notebook numpy matplotlib scipy
```

Package	Purpose
`notebook`	The Jupyter Notebook application itself
`numpy`	Array math and the FFT functions (`np.fft.fft`, `np.fft.ifft`)
`matplotlib`	Plotting signals and frequency spectra
`scipy`	Optional — extra signal-processing tools (filters, windows, etc.)

Step 6: Launch Jupyter Notebook

```
jupyter notebook
```

This starts a local server and opens Jupyter in your default browser at `http://localhost:8888`, where you can open a notebook (e.g. `fft_signal_processing.ipynb`) and run the cells. To stop the server, return to the terminal, press `Ctrl+C`, then confirm with `y`.

A.3 Option B — Miniconda / Anaconda

Step 1: Download Miniconda

```
cd ~/Downloads
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-
    x86_64.sh
```

Step 2: Run the installer

```
bash Miniconda3-latest-Linux-x86_64.sh
```

Accept the license, confirm the install location (default is fine), and say **yes** when asked whether the installer should initialize conda by running `conda init`.

Step 3: Reload your shell

```
source ~/.bashrc
```

You should now see `(base)` at the start of your terminal prompt.

Step 4: Create a dedicated environment (optional but recommended)

```
conda create -n fft-notebook python=3.11 -y
conda activate fft-notebook
```

Step 5: Install Jupyter Notebook and required libraries

```
conda install -y notebook numpy matplotlib scipy
```

Step 6: Launch Jupyter Notebook

```
jupyter notebook
```

A.4 Verifying the installation

Once Jupyter opens in the browser:

1. Click **New** → **Notebook** (or open an existing `.ipynb` file).
2. In the first cell, run:

```
import numpy as np
import matplotlib.pyplot as plt
print(np.__version__)
```

3. If this runs without errors and prints a version number, everything is installed correctly.

You can also check versions directly from the terminal (with the venv or conda environment active):

```
python3 -c "import numpy, matplotlib, scipy; print(numpy.__version__,
    matplotlib.__version__, scipy.__version__)"
```

A.5 Common issues

Problem	Likely fix
`jupyter: command not found`	The virtual environment / conda environment isn't activated. Run `source venv/bin/activate` or `conda activate fft-notebook`.
`ModuleNotFoundError: No module named 'numpy'`	You installed packages in a different environment than the one Jupyter is running in. Re-run `pip install numpy matplotlib scipy` with the correct environment active.
Plots don't show in the notebook	Make sure you're calling `plt.show()`, and that you're running in Jupyter Notebook/Lab, not a plain terminal Python session.
Browser doesn't open automatically	Copy the `http://localhost:8888/?token=...` URL printed in the terminal and paste it into your browser manually.
Permission errors with `pip install`	Don't use `sudo pip install`. Use a virtual environment (Option A, Step 3) instead.

A.6 Quick reference — all commands in one block (Option A)

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y python3 python3-pip python3-venv

mkdir -p ~/fft-notebook && cd ~/fft-notebook
python3 -m venv venv
source venv/bin/activate

pip install --upgrade pip
pip install notebook numpy matplotlib scipy

jupyter notebook
```