Aim

To analyze and process neural signals through spectral analysis using both simulated and real data, and to explore various techniques in signal processing.

Objective

1. Generate and visualize sine waves with different frequencies and amplitudes.
2. Add noise to the generated signals and analyze their impact.
3. Compute and analyze the power spectrum using FFT.
4. Compare different approaches to power spectrum computation on real neural data.
5. Implement the Fourier Transform from scratch.
6. Explore the effects of zero-padding on frequency resolution.
7. Apply frequency-domain filtering to remove noise.

Summary

This project involves generating sine waves and analyzing their spectral properties using FFT. It includes adding noise to the signals and examining its impact on frequency reconstruction. The power spectrum of real neural data is computed and compared using different averaging methods. Additionally, a manual implementation of the Fourier Transform is created to validate the FFT results. The project also explores zero-padding to improve frequency resolution and applies frequency-domain filtering to remove 50 Hz line noise.

Tools and Libraries Used

- **MATLAB**: Primary environment for coding and data analysis.
- **MATLAB's Signal Processing Toolbox**: For FFT and plotting functions.

Procedure and Code Explanation

1. Generate Sine Waves

Generate 10 seconds of data at 1 kHz, comprising 4 sine waves with different frequencies and amplitudes. Plot the individual sine waves, the summed sine waves with little noise, and with a lot of noise.

CODE:

```
srate  = 1000;  % Sampling rate

frex   = [ 3  10 15 30 ];  % Frequencies of the sine waves

amplit = [ 5  15  5  7 ];  % Amplitudes of the sine waves

phases = [  pi/8  pi  pi/2  -pi/4 ];  % Phases of the sine waves

time   = -1:1/srate:1;  % Time vector
```

```
% Create sine waves

sine_waves = zeros(length(frex), length(time));

for fi = 1:length(frex)

   sine_waves(fi,:) = amplit(fi) * sin(2*pi*time*frex(fi) + phases(fi));

end


% Generate noise

littleNoise = randn(1, length(time)) * 10;

lotsOfNoise = randn(1, length(time)) * 50;


% Plot constituent sine waves

figure(1), clf

for snum = 1:4

   subplot(4,1,snum)

   plot(time, sine_waves(snum,:))

   title(['Sine wave component with frequency ' num2str(frex(snum)) ' Hz'])

end

xlabel('Time (s)'), ylabel('Amplitude (arb.)')


% Plot summed sine waves with noise

figure(2), clf

subplot(211)

plot(time, sum(sine_waves, 1) + littleNoise)

title('Time series with LITTLE noise')


subplot(212)

plot(time, sum(sine_waves, 1) + lotsOfNoise)

title('Time series with A LOT of noise')
```

2. Compute Power Spectrum Using FFT

Compute the power spectrum of the simulated time series (with different noise levels) using FFT and plot the results.

CODE:

```
figure(3), clf


for noisei = 1:2

  % FFT

  if noisei == 1

    f = fft(sum(sine_waves, 1) + littleNoise) / length(time);

  else

    f = fft(sum(sine_waves, 1) + lotsOfNoise) / length(time);

  end


  % Compute frequencies in Hz

  hz = linspace(0, srate/2, floor(length(time)/2)+1);


  % Plot

  subplot(2,1,noisei)

  plot(hz, 2*abs(f(1:length(hz))), 'k-o')

  xlabel('Frequencies (Hz)'), ylabel('Amplitude')

  set(gca, 'xlim', [0 35], 'ylim', [0 max(amplit)*1.2])


  if noisei == 1

    title('FFT with LITTLE noise')

  else

    title('FFT with LOTS OF noise')

  end

end
```

3. Analyze Real Neural Data

Compute the power spectrum of data from electrode 7 in the laminar V1 dataset. Compare the results of averaging power spectra of individual trials and averaging trials before computing the power spectrum.

CODE:

```
% Load the LFP data
load v1_laminar.mat

% Pick which channel
chan2use = 7;

% FFT of all trials individually
powspectSeparate = fft(squeeze(csd(chan2use,:,:))) / length(timevec);
powspectSeparate = mean(2 * abs(powspectSeparate), 2); % average over trials, not over frequency!

% FFT of all trials after averaging together
powspectAverage  = fft(squeeze(mean(csd(chan2use,:,:), 3))) / length(timevec);
powspectAverage  = 2 * abs(powspectAverage);

% Frequencies in Hz
hz = linspace(0, srate/2, floor(length(timevec)/2) + 1);

% Plot
figure(4), clf
set(gcf, 'name', ['Results from electrode ' num2str(chan2use)])
subplot(211)
plot(hz, powspectSeparate(1:length(hz)))
set(gca, 'xlim', [0 100])
xlabel('Frequency (Hz)'), ylabel('Amplitude')
title('Averaging done after FFT on each trial')
```

```
subplot(212)
plot(hz, powspectAverage(1:length(hz)))
xlabel('Frequency (Hz)'), ylabel('Amplitude')
set(gca, 'xlim', [0 100])
title('FFT done on trial average')
```

4. Fourier Transform from Scratch

Implement the discrete-time Fourier transform manually and compare it with MATLAB’s FFT function.

CODE:

```
N      = 20;       % length of sequence
signal  = randn(1, N); % data
fTime   = (0:N-1) / N;  % "time" used in Fourier transform

% Initialize Fourier output matrix
fourierCoefs = zeros(size(signal));

% Loop over frequencies
for fi = 1:N
   % Create sine wave for this frequency
   fourierSine = exp(-1i * 2 * pi * (fi-1) .* fTime);
   % Compute dot product as sum of point-wise elements
   fourierCoefs(fi) = sum(fourierSine .* signal);
end

% Divide by N to scale coefficients properly
fourierCoefs = fourierCoefs / N;

% Plot results
figure(5), clf
```

```
subplot(211)
plot(signal)
title('Data')

subplot(212)
plot(abs(fourierCoefs) * 2, '*-')
xlabel('Frequency (a.u.)')

% Use the FFT function on the same data
fourierCoefsF = fft(signal) / N;

% Plot the results on top. Do they look similar? (Should be identical!)
hold on
plot(abs(fourierCoefsF) * 2, 'ro')
legend({'Manual FT', 'FFT'})
```

5. Zero-Padding and Interpolation

Compute the power spectrum with and without zero-padding to analyze the impact on frequency resolution.

CODE:

```
tidx = [dsearchn(timevec', 0) dsearchn(timevec', .5)];

% Set nfft to be multiples of the length of the data
nfft = 10 * (diff(tidx) + 1);

powspect = mean(abs(fft(squeeze(csd(7, tidx(1):tidx(2), :)), nfft, 1) / (diff(tidx) + 1)).^2, 2);
hz = linspace(0, srate/2, floor(nfft/2) + 1);

figure(6), clf
plot(hz, powspect(1:length(hz)), 'k-o', 'linew', 3)
```

```
xlabel('Frequency (Hz)'), ylabel('Power')

set(gca, 'xlim', [0 120])

title(['Frequency resolution is ' num2str(mean(diff(hz))) ' Hz'])
```

6. Frequency-Domain Filtering

Apply frequency-domain filtering to remove 50 Hz line noise from a 1/f noise signal.

CODE:

```
% Generate 1/f noise with 50 Hz line noise

srate = 1234;

npnts = srate * 3;

time  = (0:npnts-1) / srate;


% Key parameter of pink noise is the exponential decay (ed)

ed = 50; % try different values!

as = rand(1, npnts) .* exp(-(0:npnts-1) / ed);

fc = as .* exp(1i * 2 * pi * rand(size(as)));


signal = real(ifft(fc)) * npnts;


% Add 50 Hz line noise

signal = signal + sin(2 * pi * 50 * time);


% Compute its spectrum

hz = linspace(0, srate/2, floor(npnts/2) + 1);

signalX = fft(signal);


% Plot the signal and its power spectrum

figure(7), clf

subplot(211)

plot(time, signal, 'k')
```

```
xlabel('Time (s)'), ylabel('Activity')
title('Time domain')

subplot(212)
plot(hz, 2 * abs(signalX(1:length(hz))), 'k')
set(gca, 'xlim', [0 100])
xlabel('Frequency (Hz)'), ylabel('Power')
title('Frequency domain')

% Implement basic frequency-domain filter
signalX(hz > 45 & hz < 55) = 0;
filteredSig = real(ifft(signalX));

% Plot the filtered signal and its power spectrum
figure(8), clf
subplot(211)
plot(time, filteredSig, 'k')
xlabel('Time (s)'), ylabel('Activity')
title('Time domain after filtering')

subplot(212)
plot(hz, 2 * abs(signalX(1:length(hz))), 'k')
set(gca, 'xlim', [0 100])
xlabel('Frequency (Hz)'), ylabel('Power')
title('Frequency domain after filtering')
```

Highlights

- **Manual Fourier Transform**: Demonstrates a foundational understanding of the Fourier Transform by implementing it from scratch.
- **Noise Analysis**: Investigates how different levels of noise affect the spectral properties of signals.

- **Real Data Application**: Applies theoretical concepts to real neural data, providing practical insights.
- **Frequency-Domain Filtering**: Explores basic filtering techniques and their limitations, leading to a deeper understanding of signal processing.

Conclusion

The project effectively demonstrates various spectral analysis techniques on both simulated and real neural data. It highlights the impact of noise on frequency reconstruction, the benefits of zero-padding for frequency resolution, and the practical challenges of frequency-domain filtering. The manual implementation of the Fourier Transform reinforces the theoretical concepts, while the application to real data bridges the gap between theory and practice.