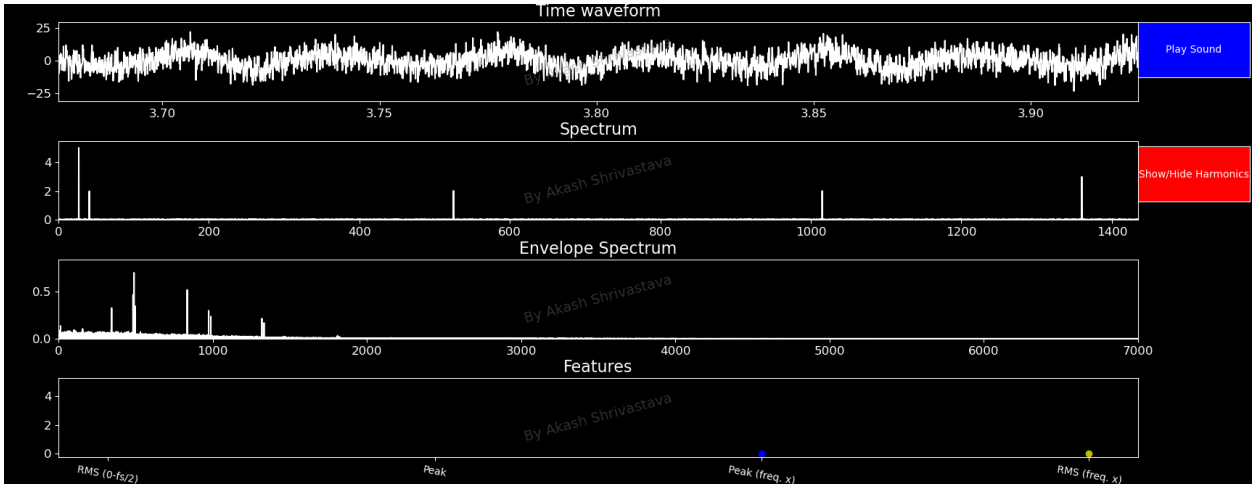
Python for Vibration Analysts: A Beginner's Guide



Introduction

Welcome to the world of Python for vibration analysis! If you're used to spreadsheets or specialized software, you might wonder why you should learn Python. Python is a versatile, powerful, and free programming language with a vast ecosystem of libraries specifically designed for scientific computing, data analysis, and visualization.

Why Python for Vibration Analysis?

1. **Automation:** Automate repetitive tasks like data loading, processing, and report generation.
2. **Customization:** Implement custom algorithms and analysis techniques not available in off-the-shelf software.
3. **Integration:** Combine vibration data with data from other sources (e.g., process parameters, maintenance logs).
4. **Advanced Analysis:** Access sophisticated libraries for signal processing, machine learning, and statistical analysis.
5. **Visualization:** Create publication-quality plots and interactive dashboards.
6. **Cost-Effective:** Python and its core scientific libraries are open-source and free to use.
7. **Large Community:** Benefit from extensive documentation, tutorials, and community support.

This guide is designed for vibration analysts with little or no programming experience. We'll start with installation and gradually build up to handling vibration data, plotting, and performing Fast Fourier Transforms (FFTs).

Chapter 1: Setting Up Your Python Environment

Before you can start analyzing data, you need to install Python and the necessary tools.

1.1 Installing Python (The Easy Way: Anaconda)

While you can install Python directly from python.org, for scientific computing, it's highly recommended to use the **Anaconda Distribution**. Anaconda bundles Python with many essential data science libraries and tools, simplifying installation and package management.

1. **Download:** Go to the Anaconda Distribution download page

(https://www.anaconda.com/download).
2. **Select OS:** Choose the installer for your operating system (Windows, macOS, Linux). Download the **Python 3.x** version (latest is usually best).
3. **Install:** Run the downloaded installer. Follow the on-screen instructions.
    - **Important (Windows):** During installation, you might be asked "Add Anaconda to my PATH environment variable." While the installer advises against it, for beginners using the command prompt directly, it *can* be convenient. However, the recommended way to access Anaconda tools is via the **Anaconda Navigator** or **Anaconda Prompt** (installed with Anaconda). Stick to the defaults if unsure.
    - Accept the default installation location unless you have a specific reason to change it.
4. **Verify:** Once installed, you can open the **Anaconda Navigator** (a graphical interface) or the **Anaconda Prompt** (a command-line interface).

1.2 Your Development Environment: IDEs and Notebooks

You need a place to write and run your Python code. Here are popular choices included with Anaconda or easily installable:

1. **Jupyter Notebook/JupyterLab:**
    - **What:** Web-based interactive environments where you can mix code, text, equations, and visualizations in a single document ("notebook").
    - **Pros:** Excellent for exploratory analysis, learning, and sharing results. Allows running code in small chunks (cells).
    - **Cons:** Can become messy for large projects. Version control can be tricky.
    - **Access:** Launchable from Anaconda Navigator or by typing jupyter notebook or jupyter lab in the Anaconda Prompt.
2. **Spyder:**
    - **What:** An Integrated Development Environment (IDE) specifically designed for scientific computing with Python.
    - **Pros:** Looks similar to MATLAB or RStudio. Includes an editor, console, variable explorer, and plotting pane. Good for developing scripts.
    - **Cons:** Can feel slightly less interactive than Jupyter for pure exploration.
    - **Access:** Included with Anaconda, launchable from Anaconda Navigator or the Start Menu/Applications folder.
3. **Visual Studio Code (VS Code):**
    - **What:** A popular, free, general-purpose code editor with excellent Python support (via extensions).
    - **Pros:** Highly customizable, supports many languages, integrates well with version control (Git). Can work seamlessly with Jupyter Notebooks.

    - **Cons:** Requires installing the Python extension separately. Might feel overwhelming initially due to its many features.
    - **Access:** Download and install from https://code.visualstudio.com/. Install the official Python extension from Microsoft within VS Code.

Recommendation for Beginners: Start with **Jupyter Notebook/Lab** for interactive exploration and learning, or **Spyder** if you prefer a more traditional IDE layout.

1.3 Installing Essential Libraries

Anaconda comes with many libraries, but you might need to install or update some. You do this using the pip package installer (or conda, Anaconda's package manager). Open the **Anaconda Prompt** (or your system terminal if Anaconda is in your PATH) and type:

```
pip install numpy pandas matplotlib scipy
```

- **numpy:** The fundamental package for numerical computing (arrays, linear algebra, etc.).
- **pandas:** Powerful library for data manipulation and analysis (DataFrames).
- **matplotlib:** The most widely used library for creating static, animated, and interactive visualizations.
- **scipy:** Builds on NumPy, providing modules for optimization, integration, interpolation, signal processing (including FFTs), and more.

You might also want seaborn for more advanced statistical plots:

```
pip install seaborn
```

Now your environment is ready!

Chapter 2: Python Fundamentals for Analysts

Let's cover the absolute basics of Python you'll need. We'll keep it focused on what's necessary for data analysis.

2.1 Basic Syntax and Variables

Python code is read line by line. Indentation (whitespace at the beginning of a line) is crucial and defines code blocks (unlike curly braces {} in other languages).

```
# This is a comment. Python ignores lines starting with #.
```

```
# Variables store data. Use descriptive names.
sensor_name = "Pump Bearing 1A" # A string (text)
sampling_rate = 25600           # An integer (whole number)
duration_seconds = 10.5         # A float (decimal number)
is_running = True               # A boolean (True or False)

# Print output to the console
print(sensor_name)
print("Sampling Rate:", sampling_rate, "Hz")
```

2.2 Basic Operations

Python supports standard arithmetic operations:

```
# Arithmetic
total_samples = sampling_rate * duration_seconds
print("Total Samples:", total_samples)

# String concatenation
location = "North Plant"
full_id = location + " - " + sensor_name
print(full_id)
```

2.3 Data Structures: Lists

Lists are ordered collections of items, enclosed in square brackets []. They are very flexible.

```
# A list of bearing fault frequencies (floats)
fault_frequencies = [145.6, 198.2, 250.0]
print("First fault frequency:", fault_frequencies[0]) # Access items by index (starts at 0)

# Add an item
fault_frequencies.append(310.5)
print("Updated list:", fault_frequencies)

# Length of the list
```

```
num_faults = len(fault_frequencies)
print("Number of faults:", num_faults)
```

2.4 Control Flow (Briefly)

- **if/elif/else:** Execute code based on conditions.

```
temperature = 75 # degrees C

if temperature > 80:
   print("ALERT: High Temperature!")
elif temperature > 65:
   print("WARNING: Elevated Temperature.")
else:
   print("Temperature OK.")
```

- **for loop:** Iterate over sequences (like lists).

```
# Print each fault frequency
for freq in fault_frequencies:
   print("Checking frequency:", freq)
```

2.5 Functions

Functions are reusable blocks of code. You define them with def and call them by name.

```
# Define a function to calculate RMS
import numpy as np # We'll import numpy properly later

def calculate_rms(signal_data):
  """Calculates the Root Mean Square of a signal."""
  # Ensure signal_data is a NumPy array for efficient calculation
  signal_array = np.array(signal_data)
  rms_value = np.sqrt(np.mean(signal_array**2))
  return rms_value

# Example usage (assuming some data)
vibration_data = [0.1, -0.2, 0.3, -0.1, 0.2, 0.0]
```

```
rms = calculate_rms(vibration_data)
print("RMS Value:", rms)
```

2.6 Importing Libraries

To use functions from libraries like NumPy or Pandas, you need to import them, usually at the beginning of your script or notebook.

```
import numpy as np        # np is the conventional alias for numpy
import pandas as pd       # pd is the conventional alias for pandas
import matplotlib.pyplot as plt # plt is the conventional alias for matplotlib's plotting module
```

Now you can use functions from these libraries, e.g., np.array(), pd.read_csv(), plt.plot().

2.7 The Importance of Comments

Use the # symbol to add comments to your code. Explain *why* you are doing something, not just *what* the code does (which should ideally be clear from the code itself). **Good comments make your code understandable** later, or by others.

Chapter 3: Working with Vibration Data (NumPy & Pandas)

Raw vibration data often comes in formats like CSV (Comma Separated Values). We need tools to load, handle, and manipulate this data efficiently.

3.1 Introduction to NumPy Arrays

NumPy's core object is the ndarray (n-dimensional array). It's like a Python list but much more efficient for numerical operations, especially on large datasets. Vibration signals are naturally represented as NumPy arrays.

```
import numpy as np

# Create an array from a list
time_data = np.array([0.0, 0.1, 0.2, 0.3, 0.4])
amplitude_data = np.array([0.5, -0.2, 0.8, 0.1, -0.5])

print("Time Array:", time_data)
print("Amplitude Array:", amplitude_data)
```

```
print("Data type:", amplitude_data.dtype) # e.g., float64
print("Shape:", amplitude_data.shape)     # (5,) -> 5 elements, 1 dimension

# Element-wise operations are fast
amplitude_squared = amplitude_data ** 2
print("Squared Amplitudes:", amplitude_squared)

# Useful NumPy functions
mean_amplitude = np.mean(amplitude_data)
std_dev = np.std(amplitude_data)
max_amplitude = np.max(amplitude_data)

print(f"Mean: {mean_amplitude:.2f}, Std Dev: {std_dev:.2f}, Max: {max_amplitude:.2f}")
```

3.2 Introduction to Pandas DataFrames

Pandas provides the DataFrame, a 2-dimensional labeled data structure, like a spreadsheet or SQL table. It's excellent for handling tabular data, including time series.

- **Rows:** Often represent time points or individual measurements.
- **Columns:** Represent different variables (e.g., 'Timestamp', 'Sensor_X', 'Sensor_Y', 'Temperature').
- **Index:** A label for each row (can be timestamps, integers, etc.).

3.3 Reading CSV Files

This is a fundamental task. Let's assume you have a CSV file named vibration_data.csv with columns like 'Timestamp', 'X_Axis_g', 'Y_Axis_g'.

```
import pandas as pd

# Define the path to your file
# Note: Use forward slashes '/' or double backslashes '\\' in paths on Windows
# Example: file_path = "C:/Users/YourUser/Documents/Data/vibration_data.csv"
# Or: file_path = "C:\\Users\\YourUser\\Documents\\Data\\vibration_data.csv"
# Or use a relative path if the file is in the same directory as your script/notebook:
file_path = 'vibration_data.csv' # Make sure this file exists where Python can find it!

# --- Create a dummy CSV file for demonstration ---
# (You would normally skip this part and use your own file)
```

```
try:
    dummy_data = {
        'Timestamp': pd.to_datetime(['2023-10-26 10:00:00', '2023-10-26 10:00:01',
'2023-10-26 10:00:02', '2023-10-26 10:00:03']),
        'X_Axis_g': [0.1, -0.2, 0.3, 0.0],
        'Y_Axis_g': [0.5, 0.4, 0.6, 0.5]
    }
    dummy_df = pd.DataFrame(dummy_data)
    dummy_df.to_csv(file_path, index=False)
    print(f"Created dummy '{file_path}' for demonstration.")
except Exception as e:
    print(f"Could not create dummy file (maybe permissions?): {e}")
    # If dummy file creation fails, the rest might fail if 'vibration_data.csv' doesn't exist
# --- End of dummy file creation ---



try:
    # Read the CSV file into a Pandas DataFrame
    # Important parameters:
    #   - parse_dates: List of columns to attempt parsing as dates
    #   - index_col: Column to use as the DataFrame index (often 'Timestamp')
    df = pd.read_csv(file_path, parse_dates=['Timestamp'], index_col='Timestamp')

    print("Successfully loaded data!")

except FileNotFoundError:
    print(f"ERROR: File not found at '{file_path}'. Please check the path.")
    # Create an empty DataFrame to prevent later errors in the example
    df = pd.DataFrame()
except Exception as e:
    print(f"An error occurred while reading the CSV: {e}")
    df = pd.DataFrame()

# If the DataFrame was loaded successfully (or the dummy was created)
if not df.empty:
    ### 3.4 Basic Data Inspection

    # Display the first 5 rows
    print("\nFirst 5 rows (head):")
```

```
    print(df.head())

    # Display the last 5 rows
    print("\nLast 5 rows (tail):")
    print(df.tail())

    # Get concise summary (column types, non-null counts)
    print("\nDataFrame Info:")
    df.info()

    # Get descriptive statistics (count, mean, std, min, max, quartiles)
    print("\nDescriptive Statistics:")
    print(df.describe())

    ### 3.5 Selecting Data

    # Select a single column (returns a Pandas Series)
    x_axis_data = df['X_Axis_g']
    print("\nX-Axis Data (Series):")
    print(x_axis_data.head())

    # Select multiple columns (returns a DataFrame)
    xy_data = df[['X_Axis_g', 'Y_Axis_g']]
    print("\nXY Data (DataFrame):")
    print(xy_data.head())

    # Select rows based on index (e.g., specific time range - requires datetime index)
    # Note: This requires the index to be sorted, which read_csv usually handles if
index_col is set.
    # try:
    #     data_slice = df['2023-10-26 10:00:00':'2023-10-26 10:00:02']
    #     print("\nData Slice by Time:")
    #     print(data_slice)
    # except Exception as e:
    #     print(f"\nCould not slice by time (Index might not be DatetimeIndex or sorted):
{e}")



    # Select rows based on condition
```

```
    high_x_vibration = df[df['X_Axis_g'] > 0.1]
    print("\nRows with X_Axis_g > 0.1:")
    print(high_x_vibration)

    # Get underlying NumPy array (if needed for specific libraries)
    x_axis_numpy = df['X_Axis_g'].values
    print("\nX-Axis data as NumPy array:", x_axis_numpy)
else:
    print("\nSkipping data inspection and selection as DataFrame is empty.")

# Clean up the dummy file
import os
if os.path.exists(file_path) and "dummy_df" in locals():
    try:
        os.remove(file_path)
        print(f"\nRemoved dummy '{file_path}'.")
    except Exception as e:
        print(f"Could not remove dummy file: {e}")
```

Common read_csv Parameters:

- filepath_or_buffer: Path to the file.
- sep or delimiter: Character used to separate values (default is ,). Use \t for tab-separated.
- header: Row number to use as column names (default is 0, the first row). Use None if there's no header.
- names: List of column names to use, especially if header=None.
- skiprows: Number of lines to skip at the beginning of the file.
- nrows: Number of rows to read (useful for large files).

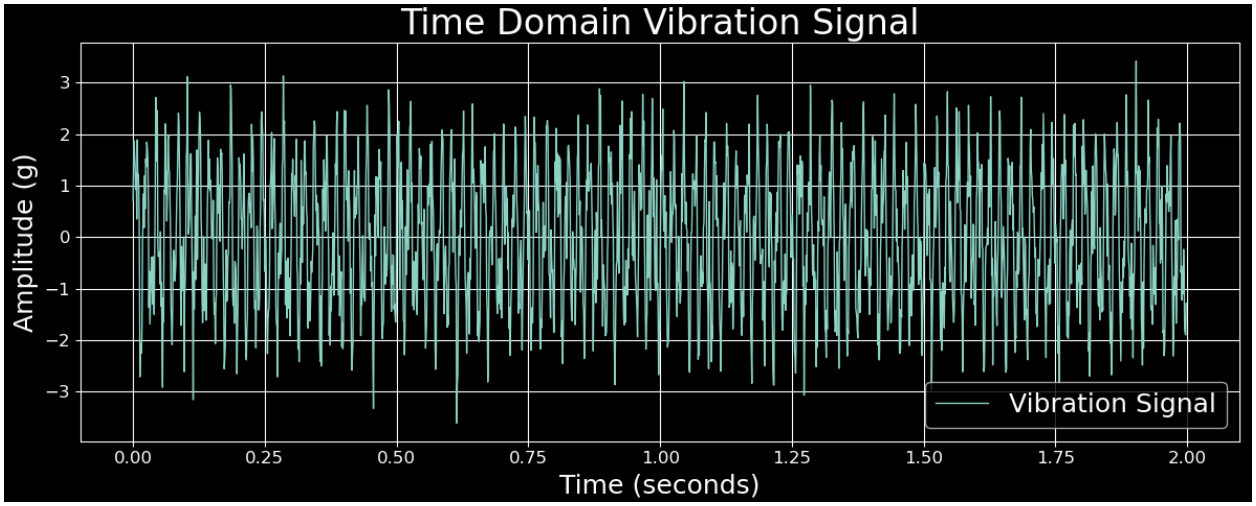
Chapter 4: Visualizing Vibration Data (Matplotlib & Seaborn)

Visualizing data is crucial for understanding trends, patterns, and anomalies. Matplotlib is the workhorse for plotting in Python.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns # Import Seaborn for correlation heatmap
```

```
# --- Generate Sample Data for Plotting ---
# (Replace this with loading your actual data using pd.read_csv)
sampling_rate = 1000 # Hz
duration = 2 # seconds
n_samples = int(sampling_rate * duration)
time = np.linspace(0, duration, n_samples, endpoint=False)

# Simulate a signal with two frequency components + noise
freq1 = 50 # Hz
freq2 = 120 # Hz
noise_level = 0.5
signal = 1.5 * np.sin(2 * np.pi * freq1 * time) + \
        0.8 * np.sin(2 * np.pi * freq2 * time) + \
        noise_level * np.random.randn(n_samples)

# Create a DataFrame
plot_df = pd.DataFrame({'Time': time, 'Amplitude_g': signal})
# Add another correlated signal for scatter/correlation plots
plot_df['Amplitude_Related'] = plot_df['Amplitude_g'] * 0.7 + 0.2 *
np.random.randn(n_samples)
plot_df['Temperature'] = 50 + 5 * np.sin(2 * np.pi * 1 * time) + 2 *
np.random.randn(n_samples) # Simulate temperature


# --- Plotting Starts Here ---

### 4.1 Plotting Time-Domain Signals

plt.figure(figsize=(12, 5)) # Create a figure object, set size (width, height in inches)

plt.plot(plot_df['Time'], plot_df['Amplitude_g'], label='Vibration Signal', linewidth=1)

# Add labels and title
plt.xlabel("Time (seconds)")
plt.ylabel("Amplitude (g)")
plt.title("Time Domain Vibration Signal")
plt.legend() # Show the legend (using the 'label' provided in plot)
plt.grid(True) # Add a grid
plt.tight_layout() # Adjust plot to prevent labels overlapping
```

plt.show() # Display the plot

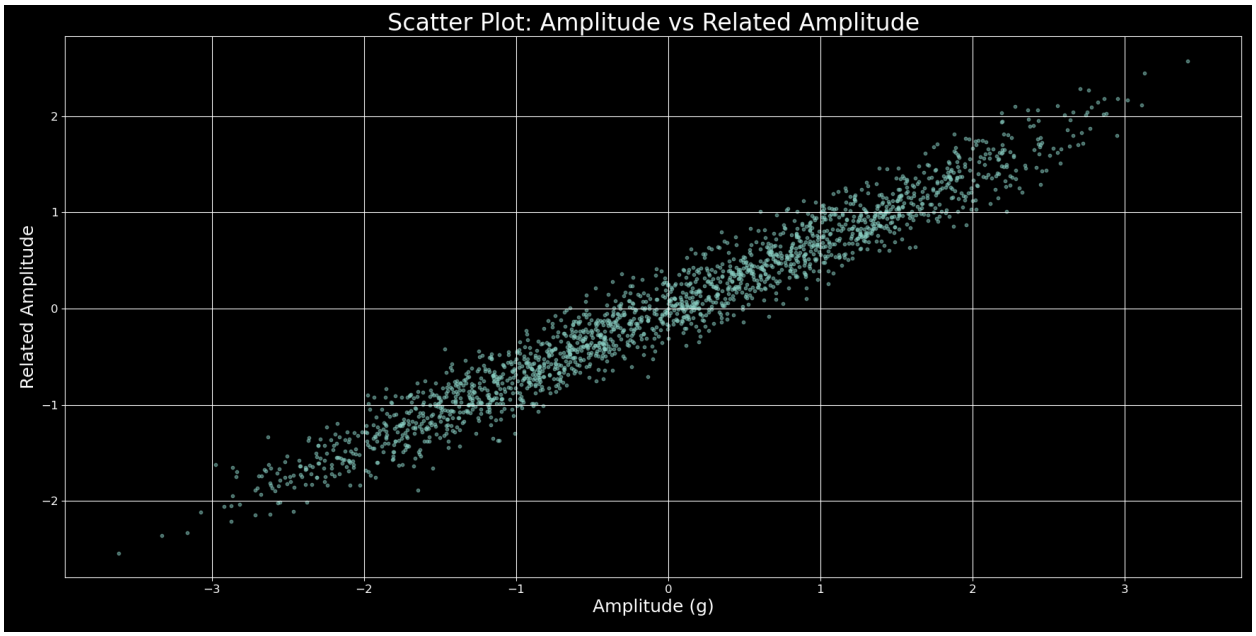


4.2 Creating Scatter Plots

# Useful for comparing two variables
plt.figure(figsize=(7, 6))

plt.scatter(plot_df['Amplitude_g'], plot_df['Amplitude_Related'], alpha=0.5, s=10) # s=size, alpha=transparency

plt.xlabel("Amplitude (g)")
plt.ylabel("Related Amplitude")
plt.title("Scatter Plot: Amplitude vs Related Amplitude")
plt.grid(True)
plt.tight_layout()
plt.show()

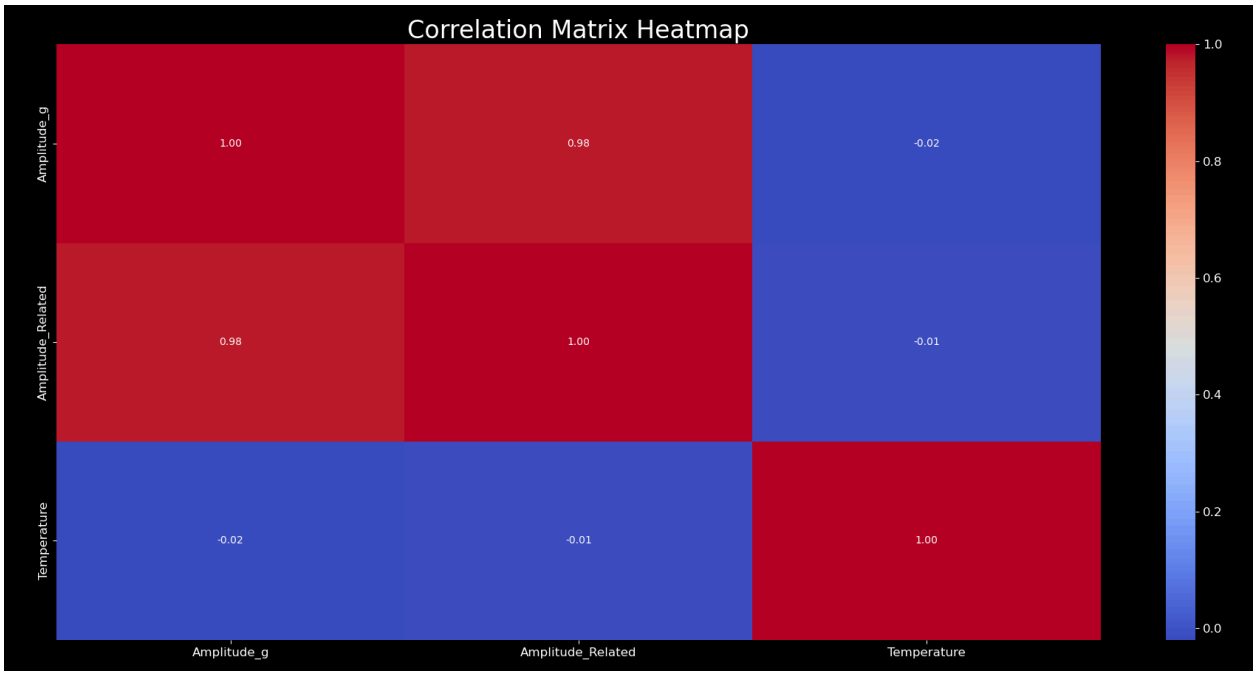


4.3 Understanding and Visualizing Correlation

Correlation measures the linear relationship between two variables (-1 to +1). Pandas DataFrames have a built-in `.corr()` method.

```
# Calculate the correlation matrix
correlation_matrix = plot_df[['Amplitude_g', 'Amplitude_Related', 'Temperature']].corr()
print("\nCorrelation Matrix:")
print(correlation_matrix)

# Visualize the correlation matrix using Seaborn's heatmap
plt.figure(figsize=(7, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
# annot=True: show values on heatmap
# cmap='coolwarm': color map choice
# fmt=".2f": format numbers to 2 decimal places

plt.title("Correlation Matrix Heatmap")
plt.tight_layout()
plt.show()
```

```
Correlation Matrix:
                  Amplitude_g  Amplitude_Related  Temperature
Amplitude_g          1.000000           0.976774    -0.020169
Amplitude_Related    0.976774           1.000000    -0.013405
Temperature         -0.020169          -0.013405     1.000000
```



Key Matplotlib Concepts:

- **plt.figure():** Creates a new figure window. figsize controls its size.
- **plt.plot(x, y):** Creates a line plot.
- **plt.scatter(x, y):** Creates a scatter plot.
- **plt.xlabel(), plt.ylabel(), plt.title():** Set labels and title.
- **plt.legend():** Displays the legend (requires label='...' in plot commands).
- **plt.grid():** Adds a grid.
- **plt.tight_layout():** Adjusts spacing.
- **plt.show():** Displays the plot(s). In Jupyter notebooks, plots often appear automatically, but plt.show() is good practice.
- **plt.savefig('filename.png'):** Saves the current figure to a file (call *before* plt.show()).

Seaborn: Builds on Matplotlib, providing higher-level functions for attractive statistical plots like heatmaps (sns.heatmap), distribution plots (sns.histplot,

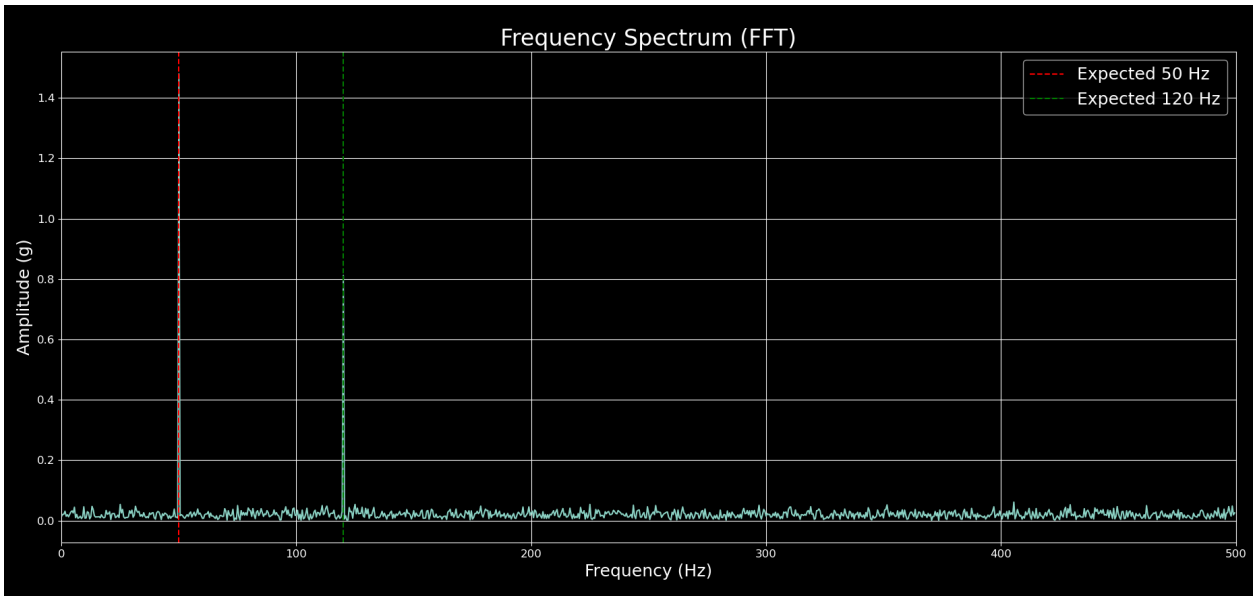
sns.kdeplot), and more.

Chapter 5: Frequency Analysis with FFT

The Fast Fourier Transform (FFT) is a cornerstone of vibration analysis, converting a time-domain signal into its frequency-domain representation, revealing the dominant frequencies present. SciPy provides the necessary tools.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq

# --- Use the Sample Data Generated in Chapter 4 ---
# signal = ... (our simulated vibration signal)
# sampling_rate = 1000 # Hz
# n_samples = len(signal)
# duration = n_samples / sampling_rate
# time = np.linspace(0, duration, n_samples, endpoint=False)

# --- FFT Calculation ---

# 1. Compute the FFT
# The output of fft is complex numbers (containing amplitude and phase info)
yf = fft(signal)

# 2. Compute the frequency bins
# fftfreq generates the frequency bins corresponding to the fft output
# It needs the number of samples (n_samples) and the sample spacing
(1/sampling_rate)
xf = fftfreq(n_samples, 1 / sampling_rate)

# --- Prepare for Plotting ---

# The FFT output is symmetric, so we only need the positive frequencies
# Select frequencies >= 0
positive_freq_indices = np.where(xf >= 0)
xf_positive = xf[positive_freq_indices]
yf_positive = yf[positive_freq_indices]

# Calculate the amplitude (magnitude) of the FFT results
```

```
# We take the absolute value of the complex numbers
# We normalize by n_samples and multiply by 2 (except for the DC component at 0 Hz)
# to get the correct amplitude for sinusoidal components.
yf_amplitude = np.abs(yf_positive) / n_samples
yf_amplitude[1:] = yf_amplitude[1:] * 2 # Double amplitudes for non-DC components


# --- Plotting the Frequency Spectrum ---

plt.figure(figsize=(12, 5))

plt.plot(xf_positive, yf_amplitude)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude (g)") # Or appropriate unit
plt.title("Frequency Spectrum (FFT)")

# Optional: Limit x-axis to relevant frequency range
plt.xlim(0, sampling_rate / 2) # Show up to Nyquist frequency

# Optional: Add grid and potentially log scale for y-axis if needed
plt.grid(True)
# plt.yscale('log') # Uncomment if amplitudes vary widely

# Optional: Mark expected frequencies
plt.axvline(freq1, color='r', linestyle='--', label=f'Expected {freq1} Hz')
plt.axvline(freq2, color='g', linestyle='--', label=f'Expected {freq2} Hz')
plt.legend()


plt.tight_layout()
plt.show()
```



--- Interpretation ---
The plot shows peaks at the dominant frequencies present in the signal.
In our example, we expect to see peaks around 50 Hz and 120 Hz,
corresponding to the sine waves we added. The height of the peak
roughly corresponds to the amplitude of that frequency component.
The rest of the plot shows the noise floor.

print(f"\nFFT Analysis Complete. Expected peaks near {freq1} Hz and {freq2} Hz.")

Key FFT Steps:

1. **Get Time Data:** You need your vibration signal (signal) and the sampling_rate.
2. **Compute FFT:** Use scipy.fft.fft(signal).
3. **Compute Frequencies:** Use scipy.fft.fftfreq(n_samples, 1 / sampling_rate).
4. **Get Positive Frequencies:** The FFT output is mirrored; usually, you only need the first half (positive frequencies).
5. **Calculate Amplitude:** Take the absolute value (np.abs()) of the positive frequency FFT results. Normalize by the number of samples (n_samples) and multiply by 2 (except the 0 Hz component) to get meaningful amplitude units corresponding to the original signal.
6. **Plot:** Plot the calculated amplitude against the positive frequencies.

Interpretation: Peaks in the FFT plot indicate the frequencies with the most energy in the signal. These often correspond to machine running speeds, bearing defect

frequencies, gear mesh frequencies, resonance, etc. The height of the peak relates to the intensity of vibration at that frequency.

Chapter 6: Basic Signal Processing Examples

Python allows you to easily calculate common vibration metrics.

6.1 Calculating Overall RMS

Root Mean Square (RMS) is a common measure of the overall energy or level of a vibration signal.

```
import numpy as np

# Assuming 'signal' is your NumPy array of vibration data from previous examples
# signal = ...

# Calculate RMS using NumPy
rms_value = np.sqrt(np.mean(signal**2))

print(f"\nOverall RMS: {rms_value:.4f} g") # Use appropriate units

# If using Pandas DataFrame 'plot_df'
rms_pandas = np.sqrt(np.mean(plot_df['Amplitude_g']**2))
print(f"Overall RMS (from DataFrame): {rms_pandas:.4f} g")
```

6.2 Peak and Peak-to-Peak Values

```
# Assuming 'signal' is your NumPy array
peak_value = np.max(np.abs(signal))
max_val = np.max(signal)
min_val = np.min(signal)
peak_to_peak = max_val - min_val

print(f"Peak Value: {peak_value:.4f} g")
print(f"Peak-to-Peak Value: {peak_to_peak:.4f} g")

# From DataFrame
peak_pandas = plot_df['Amplitude_g'].abs().max()
peak_to_peak_pandas = plot_df['Amplitude_g'].max() - plot_df['Amplitude_g'].min()
print(f"Peak Value (from DataFrame): {peak_pandas:.4f} g")
```

```
print(f"Peak-to-Peak Value (from DataFrame): {peak_to_peak_pandas:.4f} g")
```

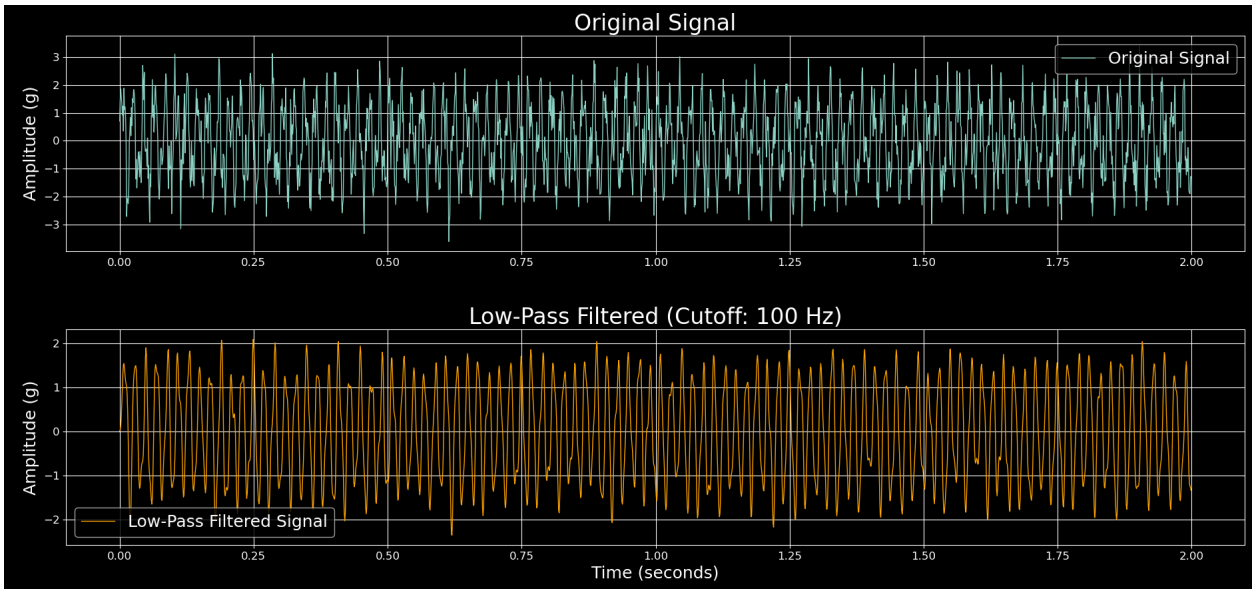
6.3 Simple Filtering (Conceptual)

Filtering (e.g., high-pass, low-pass, band-pass) removes unwanted frequency components. scipy.signal offers powerful filtering capabilities. Designing filters correctly requires understanding filter types (Butterworth, Chebyshev, etc.), order, and cutoff frequencies.

Here's a conceptual example using a Butterworth low-pass filter:

```
from scipy import signal as sig

# Filter design parameters
cutoff_freq = 100 # Hz (Want to keep frequencies below this)
filter_order = 4
fs = sampling_rate # Your sampling rate

# Design the Butterworth filter
# 'sos' output format is generally preferred for numerical stability
sos = sig.butter(filter_order, cutoff_freq, btype='low', analog=False, output='sos',
fs=fs)

# Apply the filter to the signal
# Use sosfilt for the SOS format
filtered_signal = sig.sosfilt(sos, signal) # 'signal' is your original time data

# --- Plot original vs filtered ---
plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1) # 2 rows, 1 column, plot 1
plt.plot(time, signal, label='Original Signal', linewidth=1)
plt.title("Original Signal")
plt.ylabel("Amplitude (g)")
plt.grid(True)
plt.legend()

plt.subplot(2, 1, 2) # 2 rows, 1 column, plot 2
plt.plot(time, filtered_signal, label='Low-Pass Filtered Signal', color='orange',
```

```
linewidth=1)
plt.title(f"Low-Pass Filtered (Cutoff: {cutoff_freq} Hz)")
plt.xlabel("Time (seconds)")
plt.ylabel("Amplitude (g)")
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()
```




```
# You could then perform FFT on the 'filtered_signal'
# to see the effect in the frequency domain.
```

Note: Filter design is a complex topic. Always validate your filter's effect (e.g., by checking its frequency response using scipy.signal.sosfreqz) and understand its impact on phase.

Conclusion and Next Steps

Congratulations! You've taken the first steps into using Python for vibration analysis. We've covered:

- Setting up your Python environment using Anaconda.
- Basic Python syntax relevant to data handling.
- Loading and inspecting vibration data from CSV files using Pandas.
- Visualizing time-domain signals and correlations using Matplotlib and Seaborn.
- Performing and interpreting Fast Fourier Transforms (FFT) using SciPy.

- Calculating basic metrics like RMS and Peak values.

Where to Go From Here?

1. **Practice:** Apply these techniques to your own vibration data. Start small and gradually tackle more complex datasets.
2. **Pandas Mastery:** Explore more Pandas features for data cleaning, merging datasets, resampling time series, and handling different file formats (Excel, HDF5).
3. **Advanced Signal Processing:** Dive deeper into scipy.signal for more sophisticated filtering, windowing (for FFT), spectral density estimation (Welch's method), and envelope analysis.
4. **Interactive Plotting:** Explore libraries like Plotly or Bokeh for creating interactive plots that can be embedded in web pages or dashboards.
5. **Machine Learning:** Investigate libraries like Scikit-learn (sklearn) for applying machine learning techniques (classification, clustering, anomaly detection) to vibration data for predictive maintenance.
6. **Specific Applications:** Look for libraries or examples related to specific analyses like bearing diagnostics, gear analysis, or modal analysis.
7. **Online Resources:** Utilize resources like Stack Overflow, the official documentation for NumPy, Pandas, Matplotlib, and SciPy, and numerous online tutorials and courses.

Python offers a vast and powerful toolkit for the modern vibration analyst. While the learning curve exists, the ability to automate, customize, and perform advanced analysis makes it a valuable skill to develop. Good luck!