Python for Vibration Analysts Without Coding Experience

A Step-by-Step Guide Using Google Colab and Gemini

by Gemini 2.5 & Akash

Part 1: Introduction

As a vibration analyst, you're skilled at interpreting waveforms and spectra to diagnose machinery health. Your tools might range from handheld meters to dedicated software or even spreadsheets. While effective, these tools can sometimes be limiting:

- **Repetitive Tasks:** Loading data, generating standard plots, and calculating basic metrics for many assets can be time-consuming.
- **Limited Customization:** Standard software might not offer the exact analysis or plot you need for a specific problem.
- **Handling Large Datasets:** Spreadsheets struggle with large data files from continuous monitoring.
- **Advanced Techniques:** Implementing newer or more complex analysis algorithms can be difficult or impossible.

Enter Python. Python is a powerful, versatile programming language widely used in science and engineering. For vibration analysis, it offers significant advantages:

- **Automation:** Write simple scripts (or have AI write them!) to process batches of data automatically.
- **Customization:** Perform *exactly* the analysis you need, tailored to your specific requirements.
- **Advanced Visualization:** Create publication-quality plots beyond standard templates.
- **Scalability:** Handle massive datasets efficiently.
- **Rich Ecosystem:** Access cutting-edge algorithms for signal processing, machine learning, and more through readily available libraries.

"But I Don't Know How to Code!"

We will use **Google Colaboratory (Colab)**, a free online tool, and an AI assistant like **Gemini**.

- **Colab** provides the environment to run Python code without installing anything.
- **Gemini** (or a similar AI) will act as your personal coder. You will describe the analysis task you want to perform in plain English (we call this a "prompt"), and the AI will generate the necessary Python code for you.

Your job shifts from *writing* code to *clearly describing your analysis needs* to the AI.

What You'll Achieve:

By the end of this guide, you will be able to use Google Colab and AI prompts to:

1. Load your vibration data (from uploads or Google Drive).
2. Visualize time waveforms.
3. Calculate standard time-domain metrics (RMS, Peak, Crest Factor).
4. Perform Fast Fourier Transforms (FFT) to analyze frequency content.
5. Plot frequency spectra and identify key peaks.
6. Apply basic digital filters to your data.

Let's begin!

Think of Google Colab and an AI assistant like Gemini as your new digital workshop for vibration analysis.

Google Colaboratory (Colab)

- **What it is:** Colab is a free service from Google that lets you write and execute Python code directly in your web browser. It's essentially a "Jupyter Notebook" hosted on Google's cloud servers.
- **Key Features:**
  - **No Installation:** Everything runs online; no need to install Python or libraries on your computer.
  - **Free Access:** Generous free tier, including access to computing resources.
  - **Notebook Format:** Combines live code, explanatory text, equations, and visualizations in one document. Perfect for analysis workflows.
  - **Easy Sharing:** Share your analysis notebooks just like Google Docs.
  - **Google Drive Integration:** Seamlessly access your data stored in Google Drive.

Gemini (Your AI Coding Assistant)

- **What it is:** Gemini is a powerful AI model developed by Google. It understands natural language and can generate text, translate languages, write different kinds of creative content, and, crucially for us, **write computer code**. Other similar AI tools also exist.
- **How we'll use it:** The core concept we'll use is **prompting**. You will formulate a request in English describing a specific task (like "plot the data in this column"). You can use an interface to Gemini (like the web interface, or potentially integrated Colab features) to get the Python code needed to accomplish that task. You then copy and paste this code into a Colab cell and run it.

How They Work Together:

1. **You (The Analyst):** Define the task (e.g., "Load my vibration data file").
2. **You (Prompting AI):** Write a clear instruction (prompt) for the AI describing the task.
3. **AI (Coder):** Generates the Python code required.
4. **You (Using Colab):** Paste the generated code into a "Code Cell" in your Colab notebook.
5. **Colab (Execution Engine):** Runs the Python code, performing the analysis and displaying results (tables, plots, values).

This process empowers you to leverage complex Python libraries without needing to learn their intricate syntax from scratch.

Let's get you set up with Colab.

Steps:

1. **Open your Web Browser:** Chrome is recommended, but others work too.
2. **Go to Google Colab:** Navigate to https://colab.research.google.com.
3. **Sign In:** You'll likely be prompted to sign in with your Google Account. If you don't have one, you'll need to create one (it's free).
4. **Welcome Screen:** You might see a welcome pop-up showing recent files, examples, etc. You can close this for now.
5. **Create a New Notebook:** Go to the File menu and select New notebook.

You now have a blank Colab notebook! Let's take a quick tour:

- **File Name:** At the top left, click UntitledX.ipynb to rename your notebook (e.g., "Motor_Bearing_Analysis.ipynb"). Notebooks are automatically saved to your Google Drive in a folder called 'Colab Notebooks'.
- **Menu Bar:** Contains standard options like File, Edit, View, Insert, Runtime, Tools, Help. We'll use File (New, Save) and Runtime (Run cells) most often.
- **Toolbar:** Quick access buttons below the menu bar. The most important are:
  - + Code: Adds a new Code cell (where you'll paste AI-generated Python code).
  - + Text: Adds a new Text cell (where you can write notes, headings, explanations using simple formatting called Markdown).
- **Main Work Area:** This is where your Code and Text cells live. You'll start with one empty Code cell.
- **Cell Execution:** Each Code cell has a ▶️ (Run) button on its left. Clicking this executes the code inside that specific cell. You can also press Shift + Enter to run the selected cell and move to the next one.
- **File Browser (Left Sidebar):** Click the folder icon (📁) on the far left to open the

file browser. This is where you can upload files or connect to Google Drive.

Code Cells vs. Text Cells

- **Code Cells:** Have a greyish background and a ▶️ (Run) button. This is where you put Python code. When you run it, the output (text, tables, plots) appears directly below the cell.
- **Text Cells:** Have a white background. Double-click to edit. You can use simple formatting (like # Heading 1, ## Heading 2, *italic*, **bold**, - bullet points). These are for documentation, notes, and structuring your analysis.

Try it:

1. In the first Code cell, type print("Hello Vibration World!").
2. Click the ▶️ button next to the cell (or press Shift + Enter).
3. You should see the text Hello Vibration World! printed below the cell. Congratulations, you've run your first piece of Python code!
4. Click + Text from the toolbar. Type # My Analysis Notes in the new cell. Click outside the cell (or press Shift + Enter) to see it formatted as a heading.

Now that you're familiar with the basic Colab environment, let's prepare your data.

Part 2: Getting Your

Data into Colab

Before we can analyze data, we need to make sure it's in a format Python can easily understand.

Common Formats:

The most common and easiest formats to work with are:

1. **CSV (Comma Separated Values):** A simple text file where data values are separated by commas. Often has a header row defining the column names. This is generally the **preferred** format.
2. **TXT (Text File):** Can also work, especially if columns are separated by tabs (Tab-Separated Values) or fixed-width spaces. CSV is usually more straightforward.

Essential Structure:

For basic time-series vibration analysis, your data file should ideally contain at least two columns:

1. **Time:** A column representing the time elapsed for each measurement, usually in seconds.
2. **Amplitude:** A column (or multiple columns for multi-axis sensors) representing the vibration measurement (e.g., in g's, mm/s, mils).

(Example CSV Data Snippet - motor_data_bearing1.csv)

Timestamp,Accel_X (g),Accel_Y (g),Speed (RPM)
0.0000,0.012,-0.005,1780
0.0010,-0.005,0.015,1780
0.0020,0.030,-0.022,1781
0.0030,0.015,0.008,1781
0.0040,-0.022,-0.011,1780
... more rows ...

In this example:

- The first row is the **header**, defining column names.
- Each subsequent row is a data point.

- Values are separated by commas.
- We might be interested in "Timestamp", "Accel_X (g)", and "Accel_Y (g)".

CRITICAL: Know Your Sample Rate (Fs)!

The **Sample Rate** (or Sampling Frequency, Fs) is how many data points are recorded per second (measured in Hertz, Hz). This is **essential** for accurate Frequency Domain (FFT) analysis.

- **How to find it?** Check the documentation for your data acquisition system, the settings used during recording, or sometimes the file metadata.
- **If you don't know** Fs**, your frequency analysis will be meaningless!**
- You can sometimes *calculate* it if you have accurate time stamps: Fs=1/(Time difference between consecutive samples). For the example above, the time difference is 0.001 s, so Fs=1/0.001=1000 Hz (or 1 kHz).

Make sure your data file is ready and you know its sample rate. In the next sections, we'll bring this file into Colab.

Method 1: Uploading Data Directly (Temporary Storage)

This is the quickest way to get a single file into Colab for immediate analysis, but the data disappears when your Colab session ends (usually after a period of inactivity or if you close the browser tab for too long).

Steps:

1. **Open File Browser:** In your Colab notebook, click the folder icon (📁) in the left sidebar.
2. **Click Upload:** Click the "Upload to session storage" button (looks like a page with an upward arrow).
3. **Select File:** Your computer's file dialog will open. Navigate to and select your vibration data file (e.g., motor_data_bearing1.csv). Click "Open" or "Upload".
4. **Warning:** You might see a warning that uploaded files will be deleted when the runtime is recycled. Click "OK".
5. **Verify:** Your file should now appear in the file browser list in the left sidebar.

Pros:

- Very simple and fast for single files.
- No setup required beyond clicking.

Cons:

- **Data is temporary:** Files are lost when the Colab session ends. Not suitable for

long-term projects or if you need to close and reopen the notebook later.
- Can be slow for very large files.

File Path: When uploaded this way, the file usually resides directly in the root working directory. Its path will simply be its name, e.g., 'motor_data_bearing1.csv'.

Method 2: Connecting to Google Drive (Persistent Storage)

This is the recommended method for most projects. It connects your Colab notebook directly to your Google Drive, allowing you to access files stored there. The data persists between sessions.

Steps:

1. **Add a Code Cell:** If you don't have one, click + Code.
2. **Enter Mount Code:** Type or paste the following standard code snippet into the cell:
   ```
   from google.colab import drive
   drive.mount('/content/drive')
   ```
3. **Run the Cell:** Click the ▶️ button or press Shift + Enter.
4. **Authorize Access:**
   - You'll see output with a URL (like https://accounts.google.com/o/oauth2/auth?...). Click this link.
   - A new browser tab will open asking you to choose the Google Account whose Drive you want to connect. Select the correct account.
   - You'll be asked to grant permission for "Google Drive for desktop" (or similar) to access your Google Account. Review the permissions and click "Allow" or "Continue".
   - You'll be given an authorization code. Copy this code.
5. **Enter Code in Colab:** Go back to your Colab notebook tab. Paste the copied authorization code into the input box that appeared below the code cell and press Enter.
6. **Confirmation:** You should see a message like Mounted at /content/drive.
7. **Browse Drive:** Now, in the left sidebar's file browser (📁), you should see a drive folder. Expand it, then expand MyDrive. This is the root of your Google Drive. You can navigate through your folders here to find your data file.

Finding Your File Path:

- Navigate through the drive/MyDrive/ folders in the file browser until you find your data file (e.g., inside a folder named VibrationData).

- Right-click on the file (e.g., motor_data_bearing1.csv).
- Select Copy path.
- The path will look something like /content/drive/MyDrive/VibrationData/motor_data_bearing1.csv. This is the path you'll need for loading the data.

Pros:

- Data persists between sessions.
- Good for larger files and organized projects.
- Access any file in your Google Drive.

Cons:

- Requires one-time authorization per session (though often quicker after the first time).
- Slightly more setup than direct upload.

Your First AI Prompt: Loading Data

Now that your data file is accessible (either uploaded or in Drive), let's tell the AI assistant (like Gemini) to load it into a usable format. In Python, data is often loaded into a structure called a **DataFrame**, which is like a smart spreadsheet table. The most common library for this is pandas.

Goal: Read the CSV file into a pandas DataFrame.

Key Python Library: pandas

Steps:

1. **Formulate Your Prompt:** Think about what information the AI needs:
    - What library to use (pandas).
    - What function to use (read_csv).
    - The **exact path** to your file (e.g., 'motor_data_bearing1.csv' if uploaded, or '/content/drive/MyDrive/VibrationData/motor_data_bearing1.csv' if from Drive).
    - The name you want to give the DataFrame (e.g., df is common).
    - Any special conditions (e.g., does the file have a header row?).
    - What you want to see as output (e.g., the first few rows).
2. **Example Prompt (using Drive path):**
   Prompt for AI:

Using the pandas library, load the CSV file located at '/content/drive/MyDrive/VibrationData/motor_data_bearing1.csv' into a pandas DataFrame called `df`.
Assume the first row of the CSV is the header.
After loading, display the first 5 rows of the DataFrame `df`.
Also, import the pandas library first.

3. **Get Code from AI:** Use your chosen AI interface (Gemini web, integrated tool) with the prompt above. The AI should generate Python code.
4. **Example AI-Generated Code (Conceptual):**
   ```
   # Import the pandas library
   import pandas as pd

   # Define the file path (replace with your actual path)
   file_path = '/content/drive/MyDrive/VibrationData/motor_data_bearing1.csv'
   # Or if uploaded directly: file_path = 'motor_data_bearing1.csv'

   # Load the CSV file into a DataFrame
   # Assumes the first row is the header (this is the default for read_csv)
   df = pd.read_csv(file_path)

   # Display the first 5 rows
   print("First 5 rows of the DataFrame:")
   print(df.head())
   ```
   Self-Correction Note: The AI might generate df.head() directly without print(), which also works in Colab to display the output nicely formatted.
5. **Paste and Run in Colab:**
   - Copy the code generated by the AI.
   - Go to your Colab notebook.
   - Click + Code to add a new cell.
   - Paste the code into the cell.
   - **Crucially, double-check the file_path variable in the code and make sure it exactly matches the path to *your* file.**
   - Run the cell (▶️ or Shift + Enter).
6. **Check the Output:** Below the cell, you should see the first 5 rows of your data printed in a table format. If you see an error (like FileNotFoundError), double-check the file path you provided in the code.

You have successfully loaded your vibration data into Colab using an AI prompt!

Part 3: Basic Data

Exploration & Time Domain Analysis

Understanding Your Loaded Data

Now that the data is in the DataFrame df, let's ask the AI to help us understand its structure and basic properties.

Goal: Get information about columns, data types, and summary statistics.

Key Python Library: pandas (already imported, functions are called on the DataFrame df)

Prompts & Expected Code/Output:

1. **Show Column Names and Data Types:**
    - **Prompt:**
      Show the column names and their data types for the DataFrame `df`. Also show the number of non-null entries for each column.

    - **Expected AI Code:**
      print("DataFrame Info:")
      df.info()

    - **Expected Output:** A summary listing each column name, the count of non-missing values, and the data type (e.g., float64 for numbers, object for text, int64 for integers). This helps verify if columns were read correctly (e.g., numbers are numeric, not text).
2. **Calculate Summary Statistics:**
    - **Prompt:**
      Calculate basic descriptive statistics (count, mean, standard deviation, min, 25th percentile, median, 75th percentile, max) for all numerical columns in the DataFrame `df`.

    - **Expected AI Code:**

print("\nDescriptive Statistics:")
print(df.describe())

- **Expected Output:** A table showing these statistics for each numeric column. This gives a quick overview of the range and central tendency of your measurements (e.g., average acceleration, max RPM).

3. **Show Number of Rows and Columns:**
   - **Prompt:**
     Show the total number of rows and columns in the DataFrame `df`.

   - **Expected AI Code:**
     print(f"\nDataFrame shape (rows, columns): {df.shape}")

   - **Expected Output:** Something like DataFrame shape (rows, columns): (50000, 4), indicating 50,000 data points and 4 columns.

Interpreting the Results:

- Use df.info() to confirm your Time and Amplitude columns have numeric data types (float64 or int64). If they are object, there might be non-numeric characters in the data that need cleaning (a more advanced topic).
- Use df.describe() to check for reasonable ranges (e.g., min/max amplitude, average speed). Outliers or unexpected values might indicate data issues.
- Use df.shape to confirm the amount of data loaded matches your expectations.

(Page 10: Plotting the Time Waveform)

The most fundamental vibration plot is the time waveform. Let's ask the AI to plot amplitude against time.

Goal: Visualize the raw vibration signal over time.

Key Python Libraries: matplotlib.pyplot (for plotting), pandas (to access data columns)

Steps:

1. **Identify Columns:** From the previous step (df.info() or df.head()), note the exact names of your time column (e.g., 'Timestamp') and the amplitude column you want to plot (e.g., 'Accel_X (g)'). Column names are case-sensitive!
2. **Formulate Prompt:**
   Prompt for AI:

Using the matplotlib.pyplot library (import it as plt), create a line plot using data from the DataFrame `df`.
Plot the column named 'Accel_X (g)' on the y-axis against the column named 'Timestamp' on the x-axis.
Add the title 'Vibration Time Waveform (X-Axis)' to the plot.
Label the x-axis 'Time (s)'.
Label the y-axis 'Amplitude (g)'.
Display the plot.
Make sure to import matplotlib.pyplot as plt first.

3. **Get Code from AI:** The AI should generate code using matplotlib.
4. **Example AI-Generated Code (Conceptual):**

```
# Import the plotting library
import matplotlib.pyplot as plt

# Create the plot
plt.figure(figsize=(12, 4)) # Optional: Adjust figure size for better viewing
plt.plot(df['Timestamp'], df['Accel_X (g)'])

# Add title and labels
plt.title('Vibration Time Waveform (X-Axis)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude (g)')

# Add grid for easier reading (optional)
plt.grid(True)

# Display the plot
plt.show()
```

5. **Paste and Run in Colab:** Paste the code into a new Code cell and run it.
6. **Interpret the Plot:** Below the cell, you should see the time waveform graph. Look for:
    - Overall amplitude levels.
    - Any obvious impacts or transient events.
    - Periodic patterns.
    - The general shape of the vibration.

Refining Your Plots

The default plot might be okay, but often you'll want to customize it. You can ask the AI to make changes.

Goal: Modify plot appearance (size, colors, limits, etc.).

Key Python Library: matplotlib.pyplot

Example Refinement Prompts:

- **Change Figure Size:**
  Prompt: Regenerate the previous time waveform plot, but make the figure size 15 inches wide and 5 inches tall.

  (AI should modify or add plt.figure(figsize=(15, 5)))
- **Change Line Color and Style:**
  Prompt: Regenerate the previous time waveform plot, but use a red dashed line for the data.

  (AI should modify plt.plot(...) to plt.plot(..., color='red', linestyle='--'))
- **Add Grid Lines:**
  Prompt: Regenerate the previous time waveform plot and add grid lines to both axes.

  (AI should add plt.grid(True))
- **Limit Axes:** Sometimes you want to zoom in on a specific time range or amplitude range.
  Prompt: Regenerate the previous time waveform plot, but limit the x-axis (Time) to show only data between 0.5 seconds and 1.5 seconds. Also limit the y-axis (Amplitude) from -0.5 g to 0.5 g.

  (AI should add plt.xlim(0.5, 1.5) and plt.ylim(-0.5, 0.5) before plt.show())
- **Plot Only a Subset:** Maybe you only want to plot the first N points.
  Prompt: Using matplotlib, plot only the first 1000 data points from the 'Timestamp' and 'Accel_X (g)' columns of the DataFrame `df`. Add appropriate title and labels.

  (AI might use slicing like df['Timestamp'][:1000] or df['Timestamp'].head(1000))

Process: Use these prompts like before: get the code from the AI, paste it into a *new* Colab cell (or modify the existing one), run it, and observe the changes in the plot.

Iterate until the plot looks how you want it.

(Page 12: Calculating Time Domain Metrics)

Beyond visualization, we need numbers. Let's calculate standard time-domain metrics.

Goal: Compute RMS, Peak, Peak-to-Peak, and Crest Factor.

Key Python Libraries: numpy (for numerical calculations), pandas (to access data)

Concepts:

- **Peak:** The maximum absolute value in the signal (max positive or max negative magnitude).
- **Peak-to-Peak (Pk-Pk):** The difference between the maximum positive value and the minimum negative value.
- **RMS (Root Mean Square):** A measure of the overall energy or intensity of the signal. Calculated as the square root of the mean of the squared signal values.
- **Crest Factor:** The ratio of the Peak value to the RMS value. Indicates how "peaky" the signal is (high crest factor suggests impacts).

Steps:

1. **Identify Amplitude Column:** Note the exact name of the amplitude column you want to analyze (e.g., 'Accel_X (g)').
2. **Formulate Prompt:**
   Prompt for AI:

   Using the numpy library (import it as np), calculate the following time-domain metrics for the data in the 'Accel_X (g)' column of the DataFrame `df`:
   1. Maximum Peak value (absolute maximum)
   2. Peak-to-Peak value (max value - min value)
   3. Root Mean Square (RMS) value
   4. Crest Factor (Peak / RMS)

   Print each calculated metric with a descriptive label and format the numbers to 4 decimal places. Make sure to import numpy as np first.

3. **Get Code from AI:**
4. **Example AI-Generated Code (Conceptual):**
   # Import numpy
   import numpy as np

```
# Extract the signal data into a numpy array for easier calculation
signal = df['Accel_X (g)'].to_numpy()

# Calculate metrics
peak_value = np.max(np.abs(signal))
pk_pk_value = np.max(signal) - np.min(signal)
rms_value = np.sqrt(np.mean(signal**2))

# Avoid division by zero for crest factor if RMS is zero
if rms_value == 0:
    crest_factor = np.nan # Not a Number, or could be set to 0 or infinity
depending on convention
else:
    crest_factor = peak_value / rms_value

# Print the results
print("Time Domain Metrics:")
print(f"  Peak Value:      {peak_value:.4f} g")
print(f"  Peak-to-Peak:    {pk_pk_value:.4f} g")
print(f"  RMS Value:       {rms_value:.4f} g")
print(f"  Crest Factor:    {crest_factor:.4f}")
```

5. **Paste and Run in Colab:** Paste into a new Code cell and run.
6. **Interpret the Output:** Below the cell, you'll see the calculated values. Compare these to expected levels or historical trends for your equipment. A high Crest Factor (> 3-5 is often a rule of thumb, but depends heavily on the machine) can indicate impacting or bearing faults.

(Page 13: Handling Multiple Channels)

Your data likely has multiple vibration axes (e.g., X, Y, Z). The AI can handle this easily.

Goal: Plot and calculate metrics for multiple amplitude columns.

Key Libraries: pandas, numpy, matplotlib.pyplot

Example Prompts:

1. **Plotting Multiple Channels (Overlay):**
   Prompt: Using matplotlib, plot both 'Accel_X (g)' and 'Accel_Y (g)' columns from

DataFrame `df` against the 'Timestamp' column on the SAME axes. Add a legend to identify the lines. Include title and axis labels. Make the figure size (14, 5).

(AI should generate code with two plt.plot() calls and plt.legend())

2. **Plotting Multiple Channels (Subplots):**
   Prompt: Using matplotlib, create two subplots stacked vertically.
   In the top subplot, plot 'Accel_X (g)' vs 'Timestamp' from DataFrame `df`. Title it 'X-Axis Vibration'.
   In the bottom subplot, plot 'Accel_Y (g)' vs 'Timestamp' from DataFrame `df`. Title it 'Y-Axis Vibration'.
   Add appropriate x and y labels to both. Share the x-axis. Make the overall figure size (12, 8).

   (AI should use plt.subplot() or plt.subplots())
3. **Calculating Metrics for Multiple Channels:** You can ask for metrics for each channel individually, or ask the AI to loop through them.
   Prompt: For EACH of the columns 'Accel_X (g)' and 'Accel_Y (g)' in DataFrame `df`, calculate and print the RMS and Peak values. Label the output clearly for each axis. Use numpy. Format numbers to 4 decimal places.

   (AI might generate separate calculations or use a loop)

Process: As before, use the prompts, get the code, paste, run, and interpret. This allows you to compare vibration levels and characteristics across different sensor axes.

Part 4: Frequency

Domain Analysis (FFT) with AI

Introduction to FFT

The time waveform shows *when* vibration occurs, but the **Fast Fourier Transform (FFT)** shows *what frequencies* are present in the signal. This is crucial for diagnosing specific machine faults, as different components (bearings, gears, shafts) generate vibration at characteristic frequencies.

Key Concepts:

- **FFT:** An efficient algorithm to compute the Discrete Fourier Transform (DFT). It decomposes a time signal into its constituent sine wave frequencies.
- **Frequency Spectrum:** The output of the FFT, typically plotted as Amplitude vs. Frequency. Peaks in the spectrum indicate dominant frequencies in the original signal.
- **Sample Rate (**Fs**):** How many data points per second (Hz). **Absolutely critical** for correct frequency scaling in the FFT.
- **Number of Samples (N):** The total number of data points in the time block being analyzed. Affects frequency resolution.
- **Frequency Resolution (**Δf**):** The spacing between frequency lines in the FFT spectrum. Δf=Fs/N. A smaller Δf (better resolution) requires a longer time block (larger N).
- **Nyquist Frequency (**Fnyquist**):** The maximum frequency that can be reliably detected by the FFT. Fnyquist=Fs/2. Any frequencies in the signal above this will be aliased (appear incorrectly at lower frequencies).

Goal: Transform the time-domain signal into the frequency domain and plot the spectrum.

Key Python Libraries: numpy (for FFT calculation), matplotlib.pyplot (for plotting)

Prompting for FFT Calculation

Let's ask the AI to perform the core FFT calculation.

Steps:

1. **Define Parameters:** You MUST know your sample rate (Fs). Let's assume Fs=1000 Hz for our example data (based on the 0.001s time step).
2. **Identify Signal Column:** Choose the amplitude column (e.g., 'Accel_X (g)').
3. **Formulate Prompt:**
   Prompt for AI:

   1. Extract the data from the 'Accel_X (g)' column of DataFrame `df` into a numpy array called `signal`.
   2. Get the total number of samples in `signal` and store it in a variable `N`.
   3. Assume the sample rate is 1000 Hz and store it in a variable `Fs`.
   4. Using numpy's FFT function (`np.fft.fft`), calculate the Fast Fourier Transform of the `signal`. Store the complex result in a variable called `fft_raw`.
   5. Make sure to import numpy as np first.
   Print a message confirming the calculation is done, showing N and Fs.

4. **Get Code from AI:**
5. **Example AI-Generated Code (Conceptual):**

```
import numpy as np

# 1. Extract signal
signal = df['Accel_X (g)'].to_numpy()

# 2. Get number of samples
N = len(signal)

# 3. Define sample rate (IMPORTANT: Replace with your actual Fs)
Fs = 1000 # Hz

# 4. Calculate FFT
fft_raw = np.fft.fft(signal)

# 5. Confirmation message
print(f"FFT calculation complete.")
print(f"  Number of samples (N): {N}")
print(f"  Sample Rate (Fs):      {Fs} Hz")
# Optional: print first few complex values to see the raw output
```

```
# print(f"  First 5 raw FFT values: {fft_raw[:5]}")
```

6. **Paste and Run in Colab:** Paste, ensure Fs is correct for *your* data, and run.
7. **Output:** You'll see the confirmation message. The fft_raw variable now holds the complex FFT result, which isn't directly interpretable yet.

Calculating Frequency Bins and Magnitude

The raw FFT output contains complex numbers and includes negative frequencies (a mathematical artifact). We need to process it to get a usable spectrum: frequency values (x-axis) and corresponding magnitudes (y-axis), usually only for positive frequencies up to the Nyquist frequency.

Goal: Calculate the frequency axis and the amplitude spectrum magnitude.

Key Python Libraries: numpy

Steps:

1. **Formulate Prompt:** We need the results from the previous step (fft_raw, N, Fs). Prompt for AI:

   Using the results from the previous FFT calculation (`fft_raw`, `N`, `Fs`):
   1. Calculate the corresponding frequency bins using `np.fft.fftfreq`. Store this in `freq_bins`.
   2. Calculate the single-sided amplitude spectrum magnitude. This involves:
      a. Taking the absolute value of `fft_raw`.
      b. Normalizing by the number of samples `N`.
      c. Multiplying by 2 (to account for discarding negative frequencies).
      d. Store the result in `magnitude`.
   3. Keep only the positive frequency components: Find the index corresponding to the Nyquist frequency (Fs/2). Keep only the `freq_bins` and `magnitude` values up to this index. Store these positive-frequency results in `freq` and `magnitude_positive`.

   Print the number of points in the final `freq` and `magnitude_positive` arrays.

2. **Get Code from AI:**
3. **Example AI-Generated Code (Conceptual):**
   ```
   import numpy as np
   # Assume fft_raw, N, Fs are available from the previous cell
   ```

```
# 1. Calculate frequency bins for the full spectrum
freq_bins = np.fft.fftfreq(N, d=1/Fs)

# 2. Calculate magnitude (single-sided)
# Take absolute value, normalize by N, multiply by 2 for single-sided
magnitude = 2.0/N * np.abs(fft_raw)

# 3. Keep only positive frequencies (up to Nyquist)
# Find the index corresponding roughly to Nyquist
# We can take the first N//2 points for both freq and magnitude
half_N = N // 2
freq = freq_bins[:half_N]
magnitude_positive = magnitude[:half_N]

# Correct the DC component (0 Hz) which shouldn't be doubled
if N > 0:
  magnitude_positive[0] = magnitude_positive[0] / 2.0

print(f"Calculated single-sided spectrum.")
print(f"  Number of frequency points: {len(freq)}")
print(f"  Maximum frequency (approx): {freq[-1]:.2f} Hz")
```

Self-Correction Note: The exact way to handle the single-sided spectrum (normalization, DC component) can vary slightly. The AI should handle this based on standard practice (numpy conventions). Using N//2 is a common way to get the positive frequencies.

4. **Paste and Run in Colab:** Paste and run the code.
5. **Output:** Confirmation message and details about the resulting frequency (freq) and magnitude (magnitude_positive) arrays, ready for plotting. The maximum frequency should be close to Fs/2.

(Page 17: Plotting the Frequency Spectrum)

Now we can visualize the results of the FFT.

Goal: Plot Amplitude Magnitude vs. Frequency.

Key Python Libraries: matplotlib.pyplot

Steps:

1. **Formulate Prompt:** We need the freq and magnitude_positive arrays from the previous step.
   Prompt for AI:

   Using matplotlib.pyplot (as plt), create a line plot:
   1. Plot `magnitude_positive` on the y-axis against `freq` on the x-axis.
   2. Add the title 'Frequency Spectrum (FFT)'.
   3. Label the x-axis 'Frequency (Hz)'.
   4. Label the y-axis 'Amplitude (g)'.
   5. Add grid lines.
   6. Set the figure size to (14, 5).
   7. Optionally, limit the x-axis to a relevant range if the full Nyquist range is too wide (e.g., `plt.xlim(0, 200)` to show 0-200 Hz).
   Display the plot.

2. **Get Code from AI:**
3. **Example AI-Generated Code (Conceptual):**
   import matplotlib.pyplot as plt
   # Assume freq and magnitude_positive are available from previous cell

   plt.figure(figsize=(14, 5))
   plt.plot(freq, magnitude_positive)
   plt.title('Frequency Spectrum (FFT)')
   plt.xlabel('Frequency (Hz)')
   plt.ylabel('Amplitude (g)')
   plt.grid(True)

   # Optional: Limit x-axis (adjust range as needed)
   # plt.xlim(0, 200)
   # plt.ylim(0, max(magnitude_positive)*1.1) # Optional: Adjust y-limit

   plt.show()

4. **Paste and Run in Colab:** Paste, adjust plt.xlim if desired, and run.
5. **Interpret the Plot:** This is the core diagnostic plot! Look for peaks:
   - **Identify Frequencies:** Note the frequency (x-axis value) of significant peaks.
   - **Compare to Fault Frequencies:** Relate these peak frequencies to known

fault frequencies for your equipment (e.g., running speed, bearing defect frequencies, gear mesh frequencies, blade pass frequencies).
    - **Amplitude:** The height of the peak (y-axis value) indicates the severity of vibration at that frequency. Track changes over time.

Prompting for Peak Identification

Manually reading peaks from the plot is okay, but we can ask the AI to find the most significant ones automatically.

Goal: Identify the frequencies and amplitudes of the highest peaks in the spectrum.

Key Python Libraries: scipy.signal (for peak finding), numpy

Steps:

1. **Formulate Prompt:** We need freq and magnitude_positive. We also need to decide how many peaks to find or set criteria (e.g., minimum height).
   Prompt for AI:

   Using the `scipy.signal.find_peaks` function:
   1. Find the indices of peaks in the `magnitude_positive` array. You might need to specify a minimum height threshold (e.g., only find peaks higher than 0.1 g) or a minimum distance between peaks to avoid finding multiple points on the same broad peak. Let's start by finding peaks with a minimum height of 0.05.
   2. Get the frequencies and magnitudes corresponding to these peak indices from the `freq` and `magnitude_positive` arrays.
   3. Store the peak frequencies and magnitudes.
   4. Print a table or list showing the top 10 highest peaks found, sorted by magnitude in descending order (highest first). Display both frequency (Hz) and magnitude (g), formatted to 2 decimal places.
   Make sure to import `find_peaks` from `scipy.signal` and `numpy as np`.

2. **Get Code from AI:**
3. **Example AI-Generated Code (Conceptual):**
   ```
   import numpy as np
   from scipy.signal import find_peaks
   # Assume freq and magnitude_positive are available

   # Define peak finding parameters (adjust as needed)
   min_height = 0.05 # Minimum amplitude to be considered a peak
   ```

```
min_distance = 5 # Minimum samples between peaks (adjust based on Fs/N)

# 1. Find peak indices
peak_indices, properties = find_peaks(magnitude_positive, height=min_height,
distance=min_distance)

# 2. Get corresponding frequencies and magnitudes
peak_frequencies = freq[peak_indices]
peak_magnitudes = magnitude_positive[peak_indices]

# 3. Store and Sort
# Combine frequencies and magnitudes, then sort by magnitude descending
peaks = sorted(zip(peak_frequencies, peak_magnitudes), key=lambda item:
item[1], reverse=True)

# 4. Print top 10 peaks
print("\nTop 10 Peaks Found (Frequency, Magnitude):")
print("----------------------------------------")
for i, (p_freq, p_mag) in enumerate(peaks[:10]):
    print(f"  {i+1}. {p_freq:.2f} Hz, {p_mag:.2f} g")

if not peaks:
    print("  No peaks found above the threshold.")
```

4. **Paste and Run in Colab:** Paste, potentially adjust min_height or min_distance based on your spectrum's appearance, and run.
5. **Interpret the Output:** You get a clean list of the most dominant frequencies and their amplitudes, making it much easier to focus your diagnostic efforts.

Part 5: Basic

Filtering

Why Filter?

Sometimes, raw vibration signals contain noise or components that aren't relevant to your specific analysis. Digital filters allow you to selectively remove or isolate certain frequency ranges.

Common Goals:

- **Noise Reduction:** Remove high-frequency noise unrelated to machine operation (using a Low-Pass filter).
- **Isolate Low Frequencies:** Focus on running speed and its harmonics, removing high frequencies (using a Low-Pass filter).
- **Isolate High Frequencies:** Focus on bearing or gear mesh frequencies, removing low-frequency components like unbalance (using a High-Pass filter).
- **Focus on a Specific Band:** Isolate frequencies within a known range, like a specific bearing defect frequency band (using a Band-Pass filter).

Types of Filters (Simplified):

- **Low-Pass:** Allows frequencies *below* a cutoff frequency (fc) to pass, attenuates frequencies *above* fc.
- **High-Pass:** Allows frequencies *above* a cutoff frequency (fc) to pass, attenuates frequencies *below* fc.
- **Band-Pass:** Allows frequencies *between* two cutoff frequencies (flow, fhigh) to pass, attenuates frequencies outside this band.

Key Python Library: scipy.signal (contains functions for filter design and application)

(Page 20: Prompting for a Filter Design & Application)

Let's ask the AI to design and apply a common type of filter, the Butterworth filter.

Goal: Apply a low-pass filter to the signal.

Steps:

1. **Define Filter Parameters:**
    - Filter Type: e.g., 'lowpass'
    - Cutoff Frequency (fc): The frequency where the filter starts working (e.g., 100

Hz).
    - Filter Order: Controls how sharply the filter cuts off (e.g., 4th order is common). Higher order = sharper cutoff, but can introduce phase distortion.
    - Sample Rate (Fs): Still essential! (e.g., 1000 Hz).
2. **Identify Signal:** We need the original time-domain signal array (e.g., signal created earlier from df['Accel_X (g)']).
3. **Formulate Prompt (Low-Pass Example):**
   Prompt for AI:

   Using the `scipy.signal` library:
   1. Design a 4th-order Butterworth digital filter.
   2. Make it a 'lowpass' filter with a cutoff frequency of 100 Hz.
   3. Use the sample rate `Fs` (which we defined earlier as 1000 Hz).
   4. Apply this filter to the numpy array `signal` (which contains the original time data). Use the `filtfilt` function for zero-phase filtering.
   5. Store the filtered signal in a new numpy array called `filtered_signal`.
   Make sure to import the necessary functions from `scipy.signal` (like `butter`, `filtfilt`).
   Print a message confirming the filtering is complete.

4. **Get Code from AI:**
5. **Example AI-Generated Code (Conceptual):**
   ```
   import numpy as np
   from scipy.signal import butter, filtfilt
   # Assume signal and Fs are available from previous cells

   # Filter parameters
   filter_order = 4
   cutoff_freq = 100 # Hz
   filter_type = 'lowpass'

   # Normalize cutoff frequency (required by scipy)
   # Cutoff freq is normalized to the Nyquist frequency (Fs/2)
   nyquist_freq = 0.5 * Fs
   normalized_cutoff = cutoff_freq / nyquist_freq

   # 1. Design the filter coefficients (b, a)
   b, a = butter(filter_order, normalized_cutoff, btype=filter_type, analog=False)
   ```

```
# 4. Apply the filter using filtfilt (zero-phase)
filtered_signal = filtfilt(b, a, signal)

# 5. Confirmation
print(f"Filtering complete. Filtered signal stored in 'filtered_signal'.")
print(f"  Type: {filter_type}, Order: {filter_order}, Cutoff: {cutoff_freq} Hz")
```

6. **Paste and Run in Colab:** Paste, verify parameters, and run.
7. **Output:** Confirmation message. The filtered_signal array now holds the low-pass filtered version of your original signal.

Adapting for Other Filter Types:

- **High-Pass:** Change filter_type = 'highpass' in the code/prompt.
- **Band-Pass:** Change filter_type = 'bandpass' and provide the cutoff frequency as a list/tuple: cutoff_freq = [low_cut, high_cut] (e.g., cutoff_freq = [500, 800] for 500-800 Hz band). The AI should adjust the prompt/code accordingly.

(Page 21: Visualizing Filtered Data)

How do we know the filter worked? By comparing the original and filtered signals in both the time and frequency domains.

Goal: Plot original vs. filtered time waveforms and FFTs.

Key Libraries: matplotlib.pyplot, numpy

Prompts:

1. **Compare Time Waveforms:**
   Prompt: Using matplotlib, create two subplots side-by-side (figure size 16x5).
   Left subplot: Plot the original `signal` array against time (use `df['Timestamp']` or create a time vector `np.arange(N)/Fs`). Title it 'Original Signal'.
   Right subplot: Plot the `filtered_signal` array against the same time vector. Title it 'Filtered Signal (Low-Pass 100Hz)'.
   Add appropriate axis labels and grid lines to both.

2. **Compare FFT Spectrums:** (This requires recalculating the FFT for the filtered signal)
   Prompt:
   1. Calculate the FFT for the `filtered_signal` array (using `np.fft.fft`, same `N` and `Fs` as before). Store the raw result.

2. Process this raw FFT to get the single-sided positive frequency axis (`freq_filtered`) and magnitude (`magnitude_filtered_positive`), similar to how we did it for the original signal.
3. Using matplotlib, create one plot (figure size 14x5).
4. On these axes, plot BOTH the original spectrum (`magnitude_positive` vs `freq`) AND the filtered spectrum (`magnitude_filtered_positive` vs `freq_filtered`).
5. Use different colors (e.g., blue for original, red for filtered) and add a legend.
6. Title the plot 'Original vs Filtered Spectrum'. Add axis labels and grid lines.
7. Optionally limit the x-axis (e.g., `plt.xlim(0, Fs/2)`).

Process:

- Use the prompts to get the code for plotting.
- For the FFT comparison, the AI needs to generate code that first recalculates the FFT for filtered_signal and then plots both original and filtered magnitudes on the same axes.
- Run the code and examine the plots.

Interpretation:

- **Time Waveform Plot:** The filtered signal should look smoother if it was a low-pass filter (high frequencies removed) or might show different dominant patterns if high-pass/band-pass.
- **FFT Spectrum Plot:** This clearly shows the filter's effect.
  - Low-Pass: Magnitudes above the cutoff frequency (100 Hz in our example) should be significantly reduced in the filtered (red) spectrum compared to the original (blue).
  - High-Pass: Magnitudes *below* the cutoff should be reduced.
  - Band-Pass: Magnitudes *outside* the specified band should be reduced.

Part 6: Putting it

Together & Next Steps

Mini Case Study - Workflow Example

Let's walk through a typical analysis sequence using prompts. Assume we have bearing_fault_data.csv in Google Drive at /content/drive/MyDrive/TestData/ and the sample rate Fs is 2048 Hz.

Analysis Goal: Load data, view waveform, check metrics, view FFT, identify peaks, maybe apply a high-pass filter.

Sequence of Prompts & Actions:

1. **Mount Drive:**
   - *Action:* Run the Drive mount code cell. Authorize.
2. **Load Data:**
   - *Prompt:* "Using pandas, load '/content/drive/MyDrive/TestData/bearing_fault_data.csv' into DataFrame df. Assume header exists. Show first 5 rows."
   - *Action:* Get code, paste, run, verify output.
3. **Basic Info:**
   - *Prompt:* "Show info and descriptive statistics for DataFrame df."
   - *Action:* Get code, paste, run, check columns/types/ranges. Let's say the amplitude column is 'Acceleration (m/s^2)' and time is 'Time (s)'.
4. **Plot Time Waveform:**
   - *Prompt:* "Using matplotlib, plot 'Acceleration (m/s^2)' vs 'Time (s)' from df. Title 'Bearing Time Waveform'. Label axes appropriately. Figure size (14, 4)."
   - *Action:* Get code, paste, run, view waveform.
5. **Calculate Time Metrics:**
   - *Prompt:* "Using numpy, calculate Peak, RMS, and Crest Factor for 'Acceleration (m/s^2)' column in df. Print results formatted to 3 decimals."
   - *Action:* Get code, paste, run, note metrics (e.g., Crest Factor might be high).
6. **Calculate FFT:**
   - *Prompt:* "Extract 'Acceleration (m/s^2)' from df into numpy array signal. Get N.

Set Fs = 2048. Calculate FFT using np.fft.fft, store in fft_raw."
    - *Action:* Get code, paste, set Fs correctly, run.
7. **Process FFT for Plotting:**
    - *Prompt:* "Using fft_raw, N, Fs from previous step, calculate the single-sided positive frequency axis freq and magnitude magnitude_positive."
    - *Action:* Get code, paste, run.
8. **Plot FFT Spectrum:**
    - *Prompt:* "Using matplotlib, plot magnitude_positive vs freq. Title 'Bearing FFT Spectrum'. Label axes. Grid on. Figure size (14, 5). Limit x-axis from 0 to 1000 Hz."
    - *Action:* Get code, paste, adjust xlim, run, examine peaks.
9. **Identify Peaks:**
    - *Prompt:* "Using scipy.signal.find_peaks, find peaks in magnitude_positive with minimum height 0.1. Print top 5 peaks (frequency and magnitude), sorted by magnitude."
    - *Action:* Get code, paste, adjust height threshold if needed, run, note dominant frequencies.
10. **(Optional) Apply High-Pass Filter:** Maybe we want to remove running speed (e.g., 30 Hz) to focus on higher frequencies.
    - *Prompt:* "Using scipy.signal, design a 4th order Butterworth 'highpass' filter with cutoff 50 Hz for sample rate Fs=2048. Apply it to signal using filtfilt, store in filtered_signal_hp."
    - *Action:* Get code, paste, run.
11. **(Optional) View Filtered FFT:**
    - *Prompt:* "Calculate and process the FFT for filtered_signal_hp to get freq_hp and mag_hp. Plot original spectrum (magnitude_positive vs freq) and filtered spectrum (mag_hp vs freq_hp) on the same axes using matplotlib. Add legend, title, labels. Limit x-axis 0-1000 Hz."
    - *Action:* Get code, paste, run, compare spectra (low frequencies should be reduced in filtered plot).

This sequence demonstrates how you can chain prompts together to perform a complete analysis workflow without writing the Python code yourself.

Tips for Effective Prompting

Getting good code from the AI depends on giving good prompts.

- **Be Specific:** Don't just say "load the data." Say "Using pandas, load the CSV file at '/path/to/your/file.csv' into a DataFrame named df." Mention specific column names ('Timestamp', 'Accel_X (g)'), variable names (signal, Fs, fft_raw), and

function names (np.fft.fft, plt.plot, find_peaks) if you know them or if they were used in previous steps.
- **Provide Context:** Briefly explain the goal. Mention variables created in previous steps that are needed now (e.g., "Using the signal array and Fs variable created earlier...").
- **Break Down Complex Tasks:** Instead of asking for loading, plotting, FFT, and filtering all at once, ask for each step separately. This makes it easier for the AI and easier for you to check the results at each stage.
- **Specify Libraries:** Tell the AI which library to use (pandas, numpy, matplotlib.pyplot, scipy.signal). This avoids ambiguity.
- **State Assumptions:** Clearly state things like the sample rate (Fs = 1000), file format (CSV), or if a header row exists.
- **Ask for Output:** Tell the AI what you want to see: "Print the RMS value," "Display the plot," "Show the first 5 rows," "Print the top 10 peaks."
- **Iterate and Refine:** If the first prompt doesn't give exactly what you want, modify it and try again. Add more detail or clarify your request.
- **Ask for Explanations:** If the AI generates code you don't understand, ask it! "Explain the line b, a = butter(...)" or "What does filtfilt do?".

Think of it like giving instructions to a very capable but literal assistant – clarity and detail are key.

Basic Troubleshooting

Even with AI help, things can go wrong. Here are common issues and how to approach them:

- **FileNotFoundError:**
  - *Cause:* The Python code cannot find the file specified in the path.
  - *Fix:* Double-check the file path *in the code cell* matches the *exact* location and name of your file (case-sensitive!). Use the "Copy path" feature in Colab's file browser to be sure. Ensure your Drive is mounted if accessing from Drive. Ensure the file was uploaded correctly if using direct upload.
- **NameError: name '...' is not defined:**
  - *Cause:* The code is trying to use a variable (e.g., df, signal, Fs) that hasn't been created yet or wasn't created in a cell that has been run in the current session.
  - *Fix:* Make sure you have run the previous code cells that define the variable. Colab cells execute in the order you run them, not necessarily top-to-bottom. You might need to re-run earlier cells. Check for typos in variable names in your prompt or the AI's code.

- **KeyError: 'ColumnName':**
  - *Cause:* Trying to access a column in a DataFrame (df['ColumnName']) using a name that doesn't actually exist in the DataFrame's headers.
  - *Fix:* Check the exact column names using df.head() or df.info(). Pay attention to capitalization, spaces, and special characters. Correct the column name in your prompt or the AI's code.
- **IndexError: index ... is out of bounds:**
  - *Cause:* Trying to access an element in a list or array using an index number that is too large (e.g., asking for the 100th element of a 50-element array). Often happens in plotting or processing loops if indexing is wrong.
  - *Fix:* This usually indicates a logic error in the code. Ask the AI to review the specific line causing the error and fix the indexing logic.
- **Incorrect Plot / Weird Results:**
  - *Cause:* Could be many things: incorrect data loaded, wrong columns used for plotting, incorrect sample rate (Fs) used for FFT, inappropriate filter parameters.
  - *Fix:* Go back step-by-step. Verify data loading (df.head(), df.info()). Double-check column names in plotting/calculation prompts. **Confirm your Sample Rate (Fs) is correct.** Re-examine filter parameters (cutoff frequency, order). Simplify the analysis (e.g., plot a smaller section of data) to isolate the issue.

Asking the AI for Help:

If you get an error message you don't understand:

1. **Copy the Error:** Select and copy the entire error message shown in the Colab output.
2. **Prompt the AI:**
   Prompt: I ran the following code:

   [Paste the code that caused the error here]

   And I got this error:

   [Paste the full error message here]

   Can you explain what this error means and how to fix the code?

The AI can often diagnose the problem and suggest corrected code.

Where to Go From Here?

You've learned how to use Colab and AI prompts to perform fundamental vibration analysis! This opens the door to many possibilities:

- **Explore More scipy.signal:** Ask the AI about:
  - **Windowing:** Applying windows (Hanning, Hamming, Flat Top) before FFT to reduce spectral leakage. (Prompt: "Apply a Hanning window to the signal before calculating the FFT.")
  - **Spectrograms:** Visualizing how frequency content changes over time. (Prompt: "Using scipy.signal.spectrogram, calculate and plot a spectrogram for the signal array with sample rate Fs.")
  - **Other Filter Types:** Explore Chebyshev or Elliptic filters.
- **Advanced Plotting:** Use libraries like plotly for interactive plots (zooming, panning). (Prompt: "Regenerate the FFT plot using the plotly.graph_objects library to make it interactive.")
- **Saving Results:**
  - **Saving Plots:** (Prompt: "Save the last generated matplotlib plot as a PNG file named 'spectrum.png' to my Google Drive in the '/content/drive/MyDrive/Results/' folder.")
  - **Saving Data:** (Prompt: "Save the calculated peak frequencies and magnitudes (the peaks variable) to a CSV file named 'peak_results.csv' in my Google Drive.")
  - **Saving Filtered Data:** (Prompt: "Create a new DataFrame containing the 'Timestamp' column from df and the filtered_signal array. Save this DataFrame to a CSV file named 'filtered_data.csv' in Google Drive.")
- **Automation:** For repetitive tasks on many files, you can ask the AI to help structure code that loops through files in a directory. (This requires understanding basic loop concepts).
- **Specialized Libraries:** Explore Python libraries built specifically for vibration or rotating machinery analysis (e.g., pyvib, vibration-toolbox), although these might require more understanding of Python concepts.
- **Learning Basic Python:** If you find yourself wanting more control or wanting to understand the AI's code better, consider learning Python fundamentals (variables, data types, lists, loops, functions). Many free online resources exist (like the official Python tutorial, W3Schools, Codecademy).

The combination of Colab and AI provides a powerful, low-barrier entry point. Keep

experimenting with prompts and exploring the capabilities!

Glossary

- **AI Assistant (e.g., Gemini):** A tool that understands natural language prompts and can generate code, text, or other content.
- **Amplitude:** The magnitude or intensity of the vibration signal (e.g., in g, mm/s, mils).
- **Colab (Google Colaboratory):** A free, cloud-based environment for running Python code in interactive notebooks.
- **CSV (Comma Separated Values):** A common, plain-text file format for storing tabular data.
- **Crest Factor:** The ratio of the Peak amplitude to the RMS amplitude of a signal.
- **Cutoff Frequency (**fc**):** The frequency at which a filter starts to significantly attenuate signals.
- **DataFrame (pandas):** A primary data structure in the pandas library, representing data in a labeled, 2D table format (like a spreadsheet).
- **FFT (Fast Fourier Transform):** An efficient algorithm for converting a time-domain signal into its frequency-domain components.
- **Filter (Digital):** An algorithm applied to a signal to remove or enhance specific frequency ranges.
- **Frequency (Hz):** Hertz, cycles per second. The rate of oscillation.
- **Frequency Resolution (**Δf**):** The spacing between frequency lines in an FFT spectrum (Δf=Fs/N).
- **Frequency Spectrum:** A plot showing the amplitude (or power) of different frequency components present in a signal.
- **Gemini:** Google's large language model AI assistant.
- **Google Drive:** Google's cloud storage service, easily integrated with Colab.
- **Header:** The first row in a data file (like CSV) that contains the names of the columns.
- **Jupyter Notebook:** An open-source web application that allows you to create and share documents containing live code, equations, visualizations, and narrative text. Colab is based on this.
- **Magnitude:** The size or amplitude, often used for the y-axis of an FFT spectrum.
- **matplotlib:** A fundamental Python library for creating static, animated, and interactive visualizations.
- **Mount (Drive):** The process of connecting your Google Drive to your Colab session so you can access files.
- **Notebook (.ipynb):** The file format used by Jupyter and Colab, containing code cells, text cells, and outputs.

- **numpy:** A fundamental Python library for numerical computing, especially array manipulation.
- **Nyquist Frequency (**Fnyquist**):** The highest frequency that can be accurately represented in a sampled signal (Fnyquist=Fs/2).
- **pandas:** A powerful Python library for data manipulation and analysis, providing structures like the DataFrame.
- **Peak:** The maximum absolute amplitude value in a signal.
- **Peak-to-Peak (Pk-Pk):** The difference between the maximum positive and minimum negative values in a signal.
- **Prompt:** The natural language instruction given to an AI assistant.
- **Python:** A high-level, interpreted, general-purpose programming language widely used in data science and engineering.
- **RMS (Root Mean Square):** A statistical measure of the magnitude of a varying quantity; represents the effective amplitude or energy of the signal.
- **Sample Rate (**Fs**):** The number of data points (samples) recorded per unit of time (usually seconds), measured in Hz.
- **scipy:** A Python library used for scientific and technical computing; scipy.signal contains tools for signal processing like filtering and FFT analysis.
- **Session (Colab):** The temporary connection to Google's servers that runs your notebook. Uploaded files are lost when the session ends.
- **Signal:** A function that conveys information about the behavior or attributes of some phenomenon (e.g., vibration over time).
- **Time Domain:** Representation of a signal as its amplitude changes over time.
- **Time Waveform:** A plot of a signal's amplitude versus time.