torsional

February 9, 2026

[ ]:
```
import labtools as lbts
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import find_peaks, argrelmin
import os
import argparse
import importlib, labtools
importlib.reload(labtools)
from __future__ import annotations
from pathlib import Path
import re
import pandas as pd
```

[ ]:
```
## First do step one of the investigation, find the natural frequency of
## oscillations. First, mainly find the period of oscillations while only
## damped but not driven, solve for the \beta value (as defined in lab manual)
## get natural frequency and make conclusion of oserved underdamping. \beta␣
 ↪might be useful later
```

[ ]:
```
## Take and process data from not driven but damped oscillator
## sort it out into what we need.



nodrR1 =np.genfromtxt("nodrR1.csv", delimiter = ",",comments = "%" ,skip_header␣
 ↪=0)
nodrR2 =np.genfromtxt("nodrR2.csv", delimiter = ",",comments = "%" ,skip_header␣
 ↪=0)
nodrR3 = np.genfromtxt("nodrR3.csv", delimiter = ",", comments ="%",skip_header␣
 ↪=0)
nodrR4 = np.genfromtxt("nodrR4.csv", delimiter = ",", comments = "%",␣
 ↪skip_header =0)
nodrR5 = np.genfromtxt("nodrR5.csv", delimiter = ",", comments = "%",␣
 ↪skip_header =0)
```

```
raw_runs= [nodrR1,nodrR2, nodrR3, nodrR4, nodrR5]

for i in range(len(raw_runs)):
  #print(raw_runs[i].shape)
  None


sliced_runs = []
for i in range(len(raw_runs)):
  a= np.asarray(raw_runs[i][0:1475,:])
  sliced_runs.append(a)

  #plt.plot(a[:,0],a[:,1])
  #plt.xlabel("time")
  #plt.ylabel("driving")
  #plt.title(f"run number {i+1}")
  #plt.show()

  #plt.plot(a[:,0], a[:,2])
  #plt.xlabel("time")
  #plt.ylabel("pos vol")
  #plt.title(f"run number {i+1}")
  #plt.show()

## VERY IMPORTANT
## column 1 is drive wave
## column 2 is motion wave
## this goes for all the files

## time to find out natural frequency
#print(type(sliced_runs))
#print(len(sliced_runs))
#for i in range(len(sliced_runs)):
  #print(f"{sliced_runs[i].shape} is the shape of run number {i}")

## Step number 1, find out the time intervals at which peaks and troughs
## occur, also take the opportunity to extract the voltage signals (y values)
## at these moments
## code below is for peak-to-peak analysis

lisfinxmxvals = []
lisfinymxvals = []
for i in range(len(sliced_runs)):
  #print(sliced_runs[i].shape)
  plt.plot(sliced_runs[i][:,0], sliced_runs[i][:,2])
  plt.xlabel("time")
  plt.ylabel("pos vol")
```

```
  plt.title(f"run number {i+1}")
  plt.show()
  indpeakrun = find_peaks(sliced_runs[i][:,2],height=1.45,distance=15)[0]
  #print(f"{sliced_runs[i][:,2].shape} is the shape of displacement run number
 ↪{i}")
  #print(f"{indpeakrun.shape} is the shape of indpeakrun {i}")
  #print(type(indpeakrun))
  #print(type(indpeakrun))
  #print(type(indpeakrun[0]))
  lisxvals = []
  lisyvals = []
  #print(len(indpeakrun))
  for j in range(len(indpeakrun)):
    pos = indpeakrun[j]
    #print((pos))
    #print(sliced_runs[i].shape)
    xval = sliced_runs[i][:,0][pos]
    yval = sliced_runs[i][:,2][pos]
    lisxvals.append(xval)
    lisyvals.append(yval)
  runxval = np.asarray(lisxvals)
  runyval = np.asarray(lisyvals)
  lisfinxmxvals.append(runxval)
  lisfinymxvals.append(runyval)

#print(type(lisfinxvals))
#print(type(lisfinxvals[0]))
#print(len(sliced_runs[0][:,2]))
#print(sliced_runs[0][:,2].shape)
#print(sliced_runs[0][:,2].size)
#print(sliced_runs[0].size)
#print(sliced_runs[0].shape)
#print((lisfinxmxvals[0]))
#print(lisfinymxvals[0])
#print(lisfinxmxvals[1])
#print(lisfinymxvals[1])

## Now extract the same thing, but go trough to trough

lisfinxmnvals = []
lisfinymnvals = []
for i in range(len(sliced_runs)):
  #print(sliced_runs[i].shape)
  #plt.plot(sliced_runs[i][:,0], sliced_runs[i][:,2])
  #plt.xlabel("time")
  #plt.ylabel("pos vol")
  #plt.title(f"run number {i+1}")
```

```
  #plt.show()
  indpeakrun = argrelmin(sliced_runs[i][:,2],order=15)[0]
  #print(f"{sliced_runs[i][:,2].shape} is the shape of displacement run number␣
↪{i}")
  #print(f"{indpeakrun.shape} is the shape of indpeakrun {i}")
  #print(type(indpeakrun))
  #print(type(indpeakrun))
  #print(type(indpeakrun[0]))
  lisxvals = []
  lisyvals = []
  #print(len(indpeakrun))
  for j in range(len(indpeakrun)):
    pos = indpeakrun[j]
    #print((pos))
    #print(sliced_runs[i].shape)
    xval = sliced_runs[i][:,0][pos]
    yval = sliced_runs[i][:,2][pos]
    lisxvals.append(xval)
    lisyvals.append(yval)
  runxval = np.asarray(lisxvals)
  runyval = np.asarray(lisyvals)
  mask = runyval < 1.392
  filrunxvals = runxval[mask]
  filrunyvals = runyval[mask]
  lisfinxmnvals.append(filrunxvals)
  lisfinymnvals.append(filrunyvals)

for i in range(len(lisfinxmxvals)):
  print(f"this thing has a period of {np.diff(lisfinxmxvals[i])}")

for i in range(len(lisfinxmnvals)):
  print(f"this thing has a period of {np.diff(lisfinxmnvals[i])}")
#print(lisfinxmnvals[0])
#print(lisfinymnvals[0])
#print(lisfinxmnvals[1])
#print(lisfinymxvals[0])

## data sorted succesfully. now proceed to find damped freq.
## and \beta.
```

[ ]:
```
## finding the damped oscillator angular frequency

damped_freqsmx = []
for i in range(len(lisfinxmxvals)):
  runfreqs = (2*np.pi)/np.diff(lisfinxmxvals[i])
  damped_freqsmx.append(runfreqs)
```

```
damped_freqmn = []
for i in range(len(lisfinxmnvals)):
  runfreqs = (2*np.pi)/np.diff(lisfinxmnvals[i])
  damped_freqmn.append(runfreqs)

mx_means = []
mx_uncs = []
for i in range(len(damped_freqsmx)):
  a=lbts.find_plottabl_stuff(damped_freqsmx[i])
  mx_means.append(a[0])
  mx_uncs.append(a[1])

mx_means = np.asarray(mx_means)
mx_uncs = np.asarray(mx_uncs)

mn_means = []
mn_uncs = []
#print(damped_freqmn)
for i in range(len(damped_freqmn)):
  b = lbts.find_plottabl_stuff(damped_freqmn[i])
  mn_means.append(b[0])
  mn_uncs.append(b[1])

mn_means = np.asarray(mn_means)
mn_uncs = np.asarray(mn_uncs)

#print(mn_means)
#print(mn_uncs)

finfreqmx = lbts.weighted_mean(mx_means, mx_uncs)
finfreqmn = lbts.weighted_mean(mn_means, mn_uncs)

fin_mean = np.asarray([finfreqmx[0],finfreqmn[0]])
fin_uncs = np.asarray([finfreqmx[1],finfreqmn[1]])

FIN_dmpfreqmean = lbts.weighted_mean(fin_mean,fin_uncs)[0]
FIN_dmpfrequnc = lbts.weighted_mean(fin_mean, fin_uncs)[1]

print(FIN_dmpfreqmean)
print(FIN_dmpfrequnc)
print(FIN_dmpfreqmean/(2*np.pi))
```

[ ]:
```
## Before doing any ratio division of any sort (to solve for beta), we
## have to calibrate remove the voltage offset from signals.

raw_offset = np.genfromtxt("offset_calibration.csv",delimiter = ",", comments =␣
 ↪"%",skip_header=0)
```

```
mean_offset = np.mean(raw_offset[:,2],axis=0)
mea_offset_unc = lbts.SEM(raw_offset[:,2],axis =0)

print(mean_offset)
print(mea_offset_unc)
```

[ ]:
```
## DO NOT RUN THIS CELL WITHOUT RUNNING ALL ONES ABOVE AS WELL IN ORDER
## VERY IMPORTANT!!!!!!!!!

## solve for beta
## removing the voltage offset


for i in range(len(lisfinymxvals)):
  lisfinymxvals[i] += -mean_offset

#print(lisfinymxvals)

for i in range(len(lisfinymnvals)):
  lisfinymnvals[i]+= -mean_offset
betamxs = []
for i in range(len(lisfinymxvals)):
  a= np.log(((lisfinymxvals[i][1:])/
                                                  lisfinymxvals[i][0:-1]))
  betamx = (1/np.diff(lisfinxmxvals[i])) * a
  #print(a<0)
  betamxrun = np.mean(betamx)
  betamxs.append(betamxrun)
betamxs = np.asarray(betamxs)

betamns = []
for i in range(len(lisfinymnvals)):
  betamn = (1/np.diff(lisfijupytxmnvals[i]))* (np.log((lisfinymnvals[i][1:])/
                                                 (lisfinymnvals[i][0:-1])))
  betamnrun = np.mean(betamn)
  betamns.append(betamnrun)
betamns= np.asarray(betamns)

#print(np.mean(betamxs))
#print(lbts.SEM(betamxs))
#print(np.mean(betamns))
#print(lbts.SEM(betamns))

finbeta = lbts.weighted_mean(np.asarray([np.mean(betamxs),np.mean(betamns)]),
                             np.asarray([lbts.SEM(betamxs),lbts.
 ↪SEM(betamns)]))[0]
finbetaunc = lbts.weighted_mean(np.asarray([np.mean(betamxs),np.mean(betamns)]),
```

```
                                np.asarray([lbts.SEM(betamxs),lbts.
 ↪SEM(betamns)]))[1]

print(finbeta)
print(finbetaunc)
## note that here beta was kept negative (decay nature)
```

[ ]:
```
## calculate nartural freq, with uncertianty
## observs that beta <0 and beta < freq damped

omega0 = np.sqrt(finbeta**2+FIN_dmpfreqmean**2)

unc_omega0 = np.sqrt((finbetaunc**2)*((finbeta**2)/
 ↪(FIN_dmpfreqmean**2+finbeta**2))+
                     (FIN_dmpfrequnc**2)* FIN_dmpfreqmean**2/
 ↪(FIN_dmpfreqmean**2+finbeta**2))

print(omega0)
print(unc_omega0)
```

[ ]:
```
TWOPI = 2*np.pi

## functions to transcribe the csvs into data

def parse_frequency_from_name(name: str) -> float:
    m = re.search(r'(\d+(?:\.\d+)?)\s*(mHz|Hz)', name, re.IGNORECASE)
    if not m:
        raise ValueError(f"Could not parse frequency from filename: {name}")
    val = float(m.group(1))
    unit = m.group(2).lower()
    return val/1000.0 if unit == "mhz" else val

def find_data_start(path: Path, npeek: int = 200) -> int:
    with open(path, "r", errors="ignore") as f:
        lines = []
        for _ in range(npeek):
            line = f.readline()
            if not line:
                break
            lines.append(line)
    for i, line in enumerate(lines):
        parts = [p.strip() for p in line.strip().split(",")]
        if len(parts) < 3:
            continue
        try:
            float(parts[0]); float(parts[1]); float(parts[2])
            return i
```

```
        except Exception:
            continue
    return 0

def parse_fs_from_header(path: Path, fallback: float = 200.0) -> float:
    with open(path, "r", errors="ignore") as f:
        for _ in range(140):
            line = f.readline()
            if not line:
                break
            m = re.search(r'Acquisition rate:\s*([0-9.+-eE]+)\s*Hz', line)
            if m:
                return float(m.group(1))
    return float(fallback)

def read_three_col_csv_with_fs(path: Path) -> tuple[pd.DataFrame, float]:
    skip = find_data_start(path)
    fs = parse_fs_from_header(path)
    df = pd.read_csv(path, header=None, skiprows=skip)
    if df.shape[1] < 3:
        raise ValueError(f"{path.name}: expected >=3 columns, got {df.
 ↪shape[1]}")
    df = df.iloc[:, :3].copy()
    df.columns = ["t_raw", "drive", "resp"]
    df = df.apply(pd.to_numeric, errors="coerce").dropna()

    # Reconstruct time using fs to avoid rounding artifacts in t_raw:
    N = df.shape[0]
    t = np.arange(N) / fs
    out = pd.DataFrame({"t": t, "drive": df["drive"].to_numpy(), "resp":␣
 ↪df["resp"].to_numpy()})
    return out, fs


## time domain functions

def lin_sine_fit(t: np.ndarray, y: np.ndarray, f_hz: float) -> dict:
    """TD fit: y = B sin( t) + D cos( t) + C (  fixed)."""
    w = TWOPI*f_hz
    X = np.column_stack([np.sin(w*t), np.cos(w*t), np.ones_like(t)])
    coef, *_ = np.linalg.lstsq(X, y, rcond=None)
    B, D, C0 = coef
    A = float(np.hypot(B, D))
    phi = float(np.arctan2(D, B) % TWOPI)
    return {"A": A, "phi": phi, "C": float(C0)}

def fft_projection_at_f(t: np.ndarray, y: np.ndarray, f_hz: float) -> dict:
```

```
    """Fourier method: single-frequency projection at exact f."""
    y0 = y - np.mean(y)
    w = TWOPI*f_hz
    N = len(y0)
    Z = (2.0/N) * np.sum(y0 * np.exp(-1j*w*t))
    return {"A": float(np.abs(Z)), "phi": float(np.angle(Z) % TWOPI)}

def circular_mean_sem(phases: list[float]) -> tuple[float, float]:
    phases = np.asarray(phases, dtype=float)
    z = np.exp(1j*phases)
    m = z.mean()
    mean_angle = float(np.angle(m) % TWOPI)
    R = abs(m)
    circ_std = float(np.sqrt(-2*np.log(max(R, 1e-12))))
    sem = circ_std/np.sqrt(len(phases)) if len(phases) > 1 else np.nan
    return mean_angle, sem

def chunk_indices_cycles(N: int, fs: float, f_hz: float, cycles_per_chunk:␣
 ↪float = 2.0, min_seconds: float = 2.0):
    """Split file into non-overlapping chunks so SEM can be computed."""
    chunk_s = max(min_seconds, cycles_per_chunk/max(f_hz, 1e-12))
    L = int(round(fs*chunk_s))
    L = max(L, 20)
    if N // L < 3:
        min_cycles = 1.5
        L_min = int(round(fs*(min_cycles/max(f_hz,1e-12))))
        L = max(L_min, N//3) if N//3 >= L_min else max(L_min, N//2)
    n_chunks = max(1, N // L)
    L = N // n_chunks
    starts = [i*L for i in range(n_chunks)]
    L_min = int(round(fs*(1.5/max(f_hz,1e-12))))
    return [(s, s+L) for s in starts if (s+L - s) >= L_min]

# ----------------------------
# Plot / table helpers
# ----------------------------
def phase_ticks(ax, axis='y'):
    ticks = [0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi]
    labels = ["0", r"$\pi/2$", r"$\pi$", r"$3\pi/2$", r"$2\pi$"]
    if axis == 'y':
        ax.set_yticks(ticks, labels)
        ax.set_ylim(0, 2*np.pi)
    else:
        ax.set_xticks(ticks, labels)
        ax.set_xlim(0, 2*np.pi)

def format_pm(val: float, err: float, sig: int = 2) -> str:
```

```
    if pd.isna(err) or err == 0:
        return f"{val:.6g}"
    e = int(np.floor(np.log10(abs(err)))) if err != 0 else 0
    decimals = max(0, -(e) + (sig-1))
    err_r = round(err, decimals)
    val_r = round(val, decimals)
    if abs(val_r) >= 1e4 or abs(val_r) < 1e-3:
        return f"{val_r:.{sig}e} \\pm {err_r:.{sig}e}"
    return f"{val_r:.{decimals}f} \\pm {err_r:.{decimals}f}"

def df_to_latex_table(df: pd.DataFrame, caption: str, label: str, col_spec:␣
 ↪str, headers: list[str]) -> str:
    lines=[]
    lines.append("\\begin{table}[h!]")
    lines.append("\\centering")
    lines.append(f"\\caption{{{caption}}}")
    lines.append(f"\\label{{{label}}}")
    lines.append(f"\\begin{{tabular}}{{{col_spec}}}")
    lines.append("\\hline")
    lines.append(" & ".join(headers) + " \\\\")
    lines.append("\\hline")
    for _, r in df.iterrows():
        lines.append(" & ".join(r.tolist()) + " \\\\")
    lines.append("\\hline")
    lines.append("\\end{tabular}")
    lines.append("\\end{table}")
    return "\n".join(lines)


##batch zoom selection, either coarse, medium or fine

def choose_one_file_per_frequency(files: list[Path]) -> list[Path]:
    """If duplicates exist at same frequency, prefer '(in)' filenames."""
    by_f: dict[float, Path] = {}
    for p in files:
        f = parse_frequency_from_name(p.name)
        if f not in by_f:
            by_f[f] = p
        else:
            if "(in)" in p.name and "(in)" not in by_f[f].name:
                by_f[f] = p
    return [by_f[f] for f in sorted(by_f.keys())]

def select_files_in_range(folder: Path, fmin: float, fmax: float) -> list[Path]:
    all_csvs = sorted(folder.glob("*.csv"))
    in_range = []
    for p in all_csvs:
```

```
        try:
            f = parse_frequency_from_name(p.name)
        except Exception:
            continue
        if (fmin - 1e-12) <= f <= (fmax + 1e-12):
            in_range.append(p)
    return choose_one_file_per_frequency(in_range)


## all necessary computations for selected batch


def process_batch(files: list[Path]) -> pd.DataFrame:
    rows = []
    for path in files:
        f_hz = parse_frequency_from_name(path.name)
        df, fs = read_three_col_csv_with_fs(path)
        t = df["t"].to_numpy()
        drive = df["drive"].to_numpy()
        resp = df["resp"].to_numpy()

        chunks = chunk_indices_cycles(len(t), fs, f_hz)

        A_td_list, d_td_list = [], []
        A_f_list, d_f_list, H_list = [], [], []
        for a, b in chunks:
            fd = lin_sine_fit(t[a:b], drive[a:b], f_hz)
            fr = lin_sine_fit(t[a:b], resp[a:b], f_hz)
            A_td_list.append(fr["A"])
            d_td_list.append((fr["phi"] - fd["phi"]) % TWOPI)

            od = fft_projection_at_f(t[a:b], drive[a:b], f_hz)
            orr = fft_projection_at_f(t[a:b], resp[a:b], f_hz)
            A_f_list.append(orr["A"])
            d_f_list.append((orr["phi"] - od["phi"]) % TWOPI)
            H_list.append(orr["A"]/od["A"] if od["A"] > 0 else np.nan)

        A_td_mean = float(np.mean(A_td_list))
        A_td_sem  = float(np.std(A_td_list, ddof=1)/np.sqrt(len(A_td_list))) if␣
 ↪len(A_td_list) > 1 else np.nan
        d_td_mean, d_td_sem = circular_mean_sem(d_td_list)

        A_f_mean = float(np.mean(A_f_list))
        A_f_sem  = float(np.std(A_f_list, ddof=1)/np.sqrt(len(A_f_list))) if␣
 ↪len(A_f_list) > 1 else np.nan
        d_f_mean, d_f_sem = circular_mean_sem(d_f_list)
```

```
        H_mean = float(np.nanmean(H_list))

        rows.append({
            "file": path.name,
            "f_Hz": f_hz,
            "A_TD_V": A_td_mean,
            "A_TD_SEM_V": A_td_sem,
            "delta_TD_rad": d_td_mean,
            "delta_TD_SEM_rad": d_td_sem,
            "A_Fourier_V": A_f_mean,
            "A_Fourier_SEM_V": A_f_sem,
            "delta_Fourier_rad": d_f_mean,
            "delta_Fourier_SEM_rad": d_f_sem,
            "Hmag_VperV": H_mean
        })
    return pd.DataFrame(rows).sort_values("f_Hz").reset_index(drop=True)

def write_batch_outputs(outdir: Path, res: pd.DataFrame, title_suffix: str,␣
 ↪f_fmt: str):
    outdir.mkdir(parents=True, exist_ok=True)

    ## Save CSV
    (outdir/"extracted_summary.csv").write_text(res.to_csv(index=False),␣
 ↪encoding="utf-8")

    f = res["f_Hz"].to_numpy()

    ## Time domain analusis plots
    plt.figure()
    plt.errorbar(f, res["A_TD_V"], yerr=res["A_TD_SEM_V"], fmt='o-', capsize=3)
    plt.xlabel("Driving frequency f (Hz)")
    plt.ylabel("Steady-state response amplitude (V)")
    plt.title(f"Response amplitude vs driving frequency (Time-domain sine␣
 ↪fit){title_suffix}")
    plt.grid(True, alpha=0.3)
    plt.savefig(outdir/"TD_plot_amplitude_vs_frequency.png", dpi=200,␣
 ↪bbox_inches="tight")

    plt.figure()
    plt.errorbar(f, res["delta_TD_rad"], yerr=res["delta_TD_SEM_rad"],␣
 ↪fmt='o-', capsize=3)
    plt.xlabel("Driving frequency f (Hz)")
    plt.ylabel("Phase difference  (rad)")
    plt.title(f"Phase difference vs driving frequency (Time-domain sine␣
 ↪fit){title_suffix}")
    phase_ticks(plt.gca(), 'y')
```

```
    plt.grid(True, alpha=0.3)
    plt.savefig(outdir/"TD_plot_phase_vs_frequency.png", dpi=200,␣
↪bbox_inches="tight")

    plt.figure()
    plt.errorbar(res["delta_TD_rad"], res["A_TD_V"],
                 xerr=res["delta_TD_SEM_rad"], yerr=res["A_TD_SEM_V"],
                 fmt='o-', capsize=3)
    plt.xlabel("Phase difference   (rad)")
    plt.ylabel("Steady-state response amplitude (V)")
    plt.title(f"Response amplitude vs phase difference (Time-domain sine␣
↪fit){title_suffix}")
    phase_ticks(plt.gca(), 'x')
    plt.grid(True, alpha=0.3)
    plt.savefig(outdir/"TD_plot_amplitude_vs_phase.png", dpi=200,␣
↪bbox_inches="tight")

    ## Fast Fourier Transform plots
    plt.figure()
    plt.errorbar(f, res["A_Fourier_V"], yerr=res["A_Fourier_SEM_V"], fmt='s-',␣
↪capsize=3)
    plt.xlabel("Driving frequency f (Hz)")
    plt.ylabel("Steady-state response amplitude (V)")
    plt.title(f"Response amplitude vs driving frequency (Fourier projection at␣
↪f){title_suffix}")
    plt.grid(True, alpha=0.3)
    plt.savefig(outdir/"Fourier_plot_amplitude_vs_frequency.png", dpi=200,␣
↪bbox_inches="tight")

    plt.figure()
    plt.errorbar(f, res["delta_Fourier_rad"],␣
↪yerr=res["delta_Fourier_SEM_rad"], fmt='s-', capsize=3)
    plt.xlabel("Driving frequency f (Hz)")
    plt.ylabel("Phase difference   (rad)")
    plt.title(f"Phase difference vs driving frequency (Fourier projection at␣
↪f){title_suffix}")
    phase_ticks(plt.gca(), 'y')
    plt.grid(True, alpha=0.3)
    plt.savefig(outdir/"Fourier_plot_phase_vs_frequency.png", dpi=200,␣
↪bbox_inches="tight")

    plt.figure()
    plt.errorbar(res["delta_Fourier_rad"], res["A_Fourier_V"],
                 xerr=res["delta_Fourier_SEM_rad"], yerr=res["A_Fourier_SEM_V"],
                 fmt='s-', capsize=3)
    plt.xlabel("Phase difference   (rad)")
```

```
    plt.ylabel("Steady-state response amplitude (V)")
    plt.title(f"Response amplitude vs phase difference (Fourier projection at␣
 ↪f){title_suffix}")
    phase_ticks(plt.gca(), 'x')
    plt.grid(True, alpha=0.3)
    plt.savefig(outdir/"Fourier_plot_amplitude_vs_phase.png", dpi=200,␣
 ↪bbox_inches="tight")


## main function that calls all those above

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--data_folder", type=str, default=".", help="Folder␣
 ↪containing CSV files.")
    ap.add_argument("--output_folder", type=str, default="outputs", help="Where␣
 ↪to write batch outputs.")
    args, _ = ap.parse_known_args()


    folder = Path(args.data_folder)
    outroot = Path(args.output_folder)

    # Batch definitions
    batches = [
        ("batch1", 0.50, 1.20, " - coarse sweep", ".2f"),
        ("batch2", 0.80, 0.90, " - zoom near resonance", ".2f"),
        ("batch3", 0.860, 0.875, " - fine scan near resonance", ".3f"),
    ]

    for name, fmin, fmax, suffix, fmt in batches:
        files = select_files_in_range(folder, fmin, fmax)
        if not files:
            print(f"[WARN] {name}: no files found in range [{fmin}, {fmax}] Hz")
            continue
        res = process_batch(files)
        write_batch_outputs(outroot/name, res, suffix, fmt)
        print(f"[OK] {name}: processed {len(res)} frequencies -> {outroot/
 ↪name}")

    plt.show()

main()
```