ChatGPT

EEG Wavefield Viewer – High-Resolution Implementation

EEG Wavefield Viewer is an interactive Python application for visualizing EEG data as a dynamic "wavefield" across the scalp. The updated implementation (improved from `eeg_wavefield_viewer_ui2.py`) supports high-DPI rendering and various interactive controls for exploring the data in time and different visualization modes. Key features include smooth high-resolution graphics, playback controls (both via UI buttons and keyboard shortcuts), a time slider, vector field overlays, and two visualization modes (Phase and Huygens).

Features and Controls

- **High-DPI Smooth Rendering:** The application uses Matplotlib with an adjustable figure DPI (default 150) for crisp, high-resolution output on modern displays. You can increase the DPI (e.g., `--dpi 200`) or grid resolution (e.g., `--grid 512`) for a very detailed wavefield image without sacrificing smoothness in animations.

- **Playback Controls (Play/Pause, Rewind, Fast-Forward):** You can start and stop the time animation using the **Play/Pause** button on the UI or the spacebar key. While playing, the wavefield frames advance at the specified frame rate (FPS) synchronized to the data's sampling rate, ensuring the playback corresponds to real time (or slow motion if FPS is lower than the sampling rate). The viewer also provides **rewind/forward** controls:

- Click the **"⟲ 1s"** or **"⟲ 5s"** buttons (or press the left arrow for 1s, Shift+left for 5s) to jump backward in time.
- Click **"1s ⟳"** or **"5s ⟳"** (or press the right arrow for +1s, Shift+right for +5s) to jump forward in time. These allow quick navigation through the recording [1] [2].

- The **Home** and **End** keys jump to the beginning or end of the recording, respectively.

- **Frame-by-Frame Stepping:** When paused, you can examine the data frame-by-frame. Use the **',' (comma)** key to step one frame backward and the **'.' (period)** key to step one frame forward [3]. Each "frame" corresponds to a small time increment determined by the FPS (for example, at 120 FPS on a 512 Hz EEG, one frame is ~4.27 ms of data). This fine-grained control helps inspect subtle changes between successive moments.

- **Time Slider:** The interactive time slider at the bottom of the window shows the current time (in seconds) and allows random access within the recording [4]. As the playback runs, the slider **moves in sync** with the animation, updating its position continuously. You can also click or drag the slider to seek to any time; the display will update to that point in time immediately. The slider's label shows the exact time value, and the title above the plot also reflects the current mode, frequency band, and time (e.g., "EEG Wavefield — phase | 4–12 Hz | t=10.00s").

- **Phase vs. Huygens Modes:** Two visualization modes can be toggled with the **'m'** key (or via the `--mode` command-line option):

- **Phase Mode:** Interpolates the instantaneous phase measured at each electrode (computed via the analytic signal/Hilbert transform) over the scalp. This mode displays a continuous **phase map** (using a cyclic colormap, e.g. Matplotlib's *twilight*, where –π to π phase wraps in color) and an amplitude overlay showing the signal envelope [5] [6]. Phase mode essentially shows the observed phase of brain rhythms across space.

- **Huygens Mode:** Reconstructs a wavefield by treating each electrode as a point source emitting waves (using Huygens' principle). In this mode, the app computes a superposition of spherical wavelets from each electrode (considering the current amplitude and phase), generating a *propagating wave* visualization [7] [8]. This can illustrate how waves might spread over the cortex. You can adjust the phase propagation speed via the `--c` parameter (phase velocity scale) if needed. The color mapping in Huygens mode similarly represents phase (angle of the complex field) and amplitude (magnitude).

- **Vector Field Overlay:** For additional insight, you can overlay a vector field indicating the spatial **phase gradient** (flow of the waves) by pressing **'v'** or using the `--vectors` flag [9]. This draws semi-transparent arrows on the wavefield showing the direction of increasing phase (i.e., indicative of wavefront propagation direction) and relative speed (arrow length). Toggling vectors **on** can help visualize how the wave patterns move across the scalp. Press 'v' again to hide the vectors. The vector overlay updates in real-time with the animation and in both visualization modes.

- **Accurate Timing and Data Sync:** The playback is synchronized to the data's sampling rate and the chosen FPS. Each video frame advances the EEG by the appropriate number of samples so that, for example, at 120 FPS on a 240 Hz dataset, one second of real time corresponds to one second of data. This ensures the **temporal dynamics** are faithfully represented (no unintended speed-up or slow-down, unless you intentionally choose an FPS lower than the data rate). The implementation uses the data's sample rate (Hz) and frame rate to calculate how many samples to skip per frame, including fractional steps if needed, so that timing stays on track. The animation will automatically stop when the end of the recording is reached.

- **Screenshot Capture:** You can save a high-resolution screenshot of the current frame at any time by pressing **'s'** [10]. The app will save a PNG image (at the current figure DPI) with a filename like `wavefield_<TIMESTAMP>ms.png` indicating the time in the recording. This is useful for capturing interesting wave patterns or moments for reports or analysis.

Code Implementation

Below is the complete Python code for the improved EEG Wavefield Viewer. It reads EEG data from a CSV (with header row of channel names) or EDF file, applies preprocessing (bandpass filter and optional notch filter and surface Laplacian), computes the analytic signal for phase/amplitude, and sets up an interactive Matplotlib figure with all the described functionality. The code uses Matplotlib's interactive widgets (Slider and Buttons) and event connections for keyboard controls. It leverages NumPy and SciPy for signal processing, and `scipy.interpolate.griddata` for smooth spatial interpolation of values across a 2D grid representing the scalp.

```
# eeg_wavefield_viewer_ui2.py — High-Resolution EEG Wavefield Viewer
import argparse, sys, math, time, os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider, Button
from scipy.signal import butter, filtfilt, hilbert, detrend, iirnotch
from scipy.interpolate import griddata

# ----------------------------
# Electrode layout (approximate 10-20/10-10 coordinates on a unit circle
projection)
# ----------------------------
COORDS = {
    "Fp1":(-0.5,1.0),"Fpz":(0.0,1.05),"Fp2":(0.5,1.0),
    "AF7":(-0.7,0.9),"AF3":(-0.35,0.92),"AFz":(0.0,0.95),
    "AF4":(0.35,0.92),"AF8":(0.7,0.9),
    "F7":(-0.85,0.75),"F5":(-0.55,0.78),"F3":(-0.3,0.8),"F1":(-0.1,0.82),
    "Fz":(0.0,0.85),"F2":(0.1,0.82),"F4":(0.3,0.8),"F6":(0.55,0.78),"F8":(0.85,
0.75),
    "Ft7":(-0.98,0.58),"Ft8":(0.98,0.58),
    "FC5":(-0.6,0.55),"FC3":(-0.35,0.58),"FC1":(-0.15,0.6),"FCz":(0.0,0.62),
    "FC2":(0.15,0.6),"FC4":(0.35,0.58),"FC6":(0.6, 0.55),
    "T7":(-1.05,0.40),"C5":(-0.65,0.38),"C3":(-0.35,0.40),"C1":(-0.15,0.42),
    "Cz":(0.0,0.45),"C2":(0.15,0.42),"C4":(0.35,0.40),"C6":(0.65,0.38),"T8":
(1.05,0.40),
    "T9":(-1.12,0.35),"T10":(1.12,0.35),
    "CP5":(-0.6,0.25),"CP3":(-0.35,0.28),"CP1":(-0.15,0.30),"CPz":(0.0,0.32),
    "CP2":(0.15,0.30),"CP4":(0.35,0.28),"CP6":(0.6,0.25),
    "P7":(-0.9,0.08),"P5":(-0.6,0.10),"P3":(-0.35,0.12),"P1":(-0.15,0.13),
    "Pz":(0.0,0.14),"P2":(0.15,0.13),"P4":(0.35,0.12),"P6":(0.6,0.10),"P8":(0.9,
0.08),
    "Po7":(-0.8,-0.02),"Po3":(-0.4,0.0),"Poz":(0.0,0.0),"Po4":(0.4,0.0),"Po8":
(0.8,-0.02),
    "O1":(-0.35,-0.25),"Oz":(0.0,-0.27),"O2":(0.35,-0.25)
}
ALIASES = {
    "FT7":"Ft7","FT8":"Ft8","PO7":"Po7","PO3":"Po3","PO4":"Po4","PO8":"Po8",
    "T7.":"T7","T8.":"T8","T9.":"T9","T10.":"T10","FCz":"FCz","Fcz":"FCz",
    "Pz.":"Pz","Fz.":"Fz","Cz.":"Cz","Oz.":"Oz"
}
def _map_label(lbl):
    """Normalize channel label to match COORDS keys (handle aliases and stray
characters)."""
    k = ALIASES.get(lbl.strip(), lbl.strip())
    return k.replace('..','').replace('.','')
```

```
# ----------------------------
# Data Loading (CSV or EDF)
# ----------------------------
def load_csv(path, fs=None):
    """Load EEG data from CSV. Assumes first row is headers with channel
names."""
    df = pd.read_csv(path)
    # Columns named 'time', 'sample', etc. are ignored; all others treated as
channels
    cols = [c for c in df.columns if c.lower() not in
("time","t","sample","index")]
    data = df[cols].to_numpy(dtype=float)
    ch_names = [_map_label(c) for c in cols]
    return data, ch_names, fs  # fs may be provided via argument

def load_edf(path):
    """Load EEG data from an EDF file using pyedflib (must be installed
separately)."""
    try:
        import pyedflib
    except ImportError:
        print("pyedflib not installed; please `pip install pyedflib` or use a
CSV file.")
        sys.exit(1)
    f = pyedflib.EdfReader(path)
    n = f.signals_in_file
    ch_names = [_map_label(f.getLabel(i)) for i in range(n)]
    # Read all channel signals and stack into array of shape (samples, channels)
    sig = np.vstack([f.readSignal(i) for i in range(n)]).T
    fs = float(f.getSampleFrequency(0))
    f.close()
    return sig, ch_names, fs

# ----------------------------
# Preprocessing Helpers
# ----------------------------
def bandpass(X, fs, lo, hi, order=4):
    """Apply a Butterworth band-pass filter to the signals (along time axis)."""
    b, a = butter(order, [lo/(fs/2), hi/(fs/2)], btype='band')
    return filtfilt(b, a, X, axis=0)

def notch(X, fs, f0=50.0, Q=30.0):
    """Apply a notch filter at frequency f0 (to remove mains interference)."""
    try:
        b, a = iirnotch(f0/(fs/2), Q)
        return filtfilt(b, a, X, axis=0)
    except Exception:
        # If filter design fails (e.g., if f0 is too high relative to fs), just
```

```
return original
        return X

def laplacian(X, ch_names):
    """Compute the surface Laplacian (spatial high-pass) to accentuate local
activity."""
    out = X.copy()
    coords = np.array([COORDS[c] for c in ch_names if c in COORDS])
    used_idx = [i for i, c in enumerate(ch_names) if c in COORDS]
    for idx, i in enumerate(used_idx):
        xi, yi = COORDS[ch_names[i]]
        d = np.linalg.norm(coords - np.array([xi, yi]), axis=1)
        # Consider neighbors within a certain radius (e.g., 0.5)
        nbrs = [used_idx[j] for j in np.where((d > 1e-6) & (d < 0.5))[0]]
        if len(nbrs) >= 3:
            # Subtract average of neighbor signals from the center (spatial
high-pass)
            out[:, i] = X[:, i] - np.mean(X[:, nbrs], axis=1)
    return out

def analytic(X):
    """Compute analytic signal per channel using Hilbert transform. Returns
complex, amplitude, phase."""
    Z = hilbert(X, axis=0)
    A = np.abs(Z)
    P = np.angle(Z)
    return Z, A, P

# ----------------------------
# Grid and Wavefield computation
# ----------------------------
def make_grid(n=256, margin=0.15):
    """Create a 2D grid (Xg, Yg) covering the head area for interpolation, with
a circular mask."""
    x = np.linspace(-1.1, 1.1, n)
    y = np.linspace(-0.9, 1.15, n)
    Xg, Yg = np.meshgrid(x, y)
    # Define a mask that roughly outlines an oval head shape (to clip outside
points)
    mask = ((Xg/1.1)**2 + ((Yg - 0.15)/1.0)**2) <= (1.0 - margin)
    return Xg, Yg, mask

def huygens(ch_names, amps, phases, Xg, Yg, k=40.0, eps=1e-2):
    """
    Reconstruct a wavefield using Huygens' principle.
    Each electrode acts as a point source emitting a spherical wave with
wavenumber k.
    amps and phases are arrays (per channel) giving current amplitude and phase.
```

```
    Returns a complex field F on the grid.
    """
    F = np.zeros_like(Xg, dtype=np.complex128)
    for a, ph, nm in zip(amps, phases, ch_names):
        if nm not in COORDS:
            continue
        sx, sy = COORDS[nm]
        r = np.hypot(Xg - sx, Yg - sy)
        # Add spherical wavelet from this source
        F += (a / np.sqrt(r + eps)) * np.exp(1j * (k * r + ph))
    return F

# ----------------------------
# Wavefield Viewer Class
# ----------------------------
class WavefieldViewer:
    def __init__(self, data, ch_names, fs, band=(4, 12), notch_hz=0.0,
                 use_laplacian=False, mode="phase", show_vectors=False,
                 fps=30, grid=256, phase_c=1.0, dpi=150):
        """
        Initialize the EEG Wavefield viewer.
        - data: 2D NumPy array of shape (n_samples, n_channels)
        - ch_names: list of channel name strings corresponding to columns in
data
        - fs: sampling rate of the data in Hz
        - band: (low, high) frequency range for bandpass filtering
        - notch_hz: notch filter frequency (50 or 60 Hz typical), 0 to disable
        - use_laplacian: if True, apply surface Laplacian spatial filter
        - mode: "phase" or "huygens" for initial visualization mode
        - show_vectors: if True, show phase gradient vectors initially
        - fps: playback speed in frames per second
        - grid: resolution of the interpolation grid (pixels per side)
        - phase_c: phase velocity scale factor for Huygens mode (c in Huygens
k=2πf_c/c)
        - dpi: figure DPI for high-resolution rendering
        """
        self.fs = float(fs)
        self.mode = mode
        self.show_vectors = show_vectors
        self.grid_n = grid
        self.phase_c = phase_c
        self.fps = int(fps)
        # Save band range and compute parameters for Huygens mode
        self.band = (float(band[0]), float(band[1]))
        self.k = 2 * np.pi * (0.5 * (self.band[0] + self.band[1])) / max(1e-6,
self.phase_c)

# Determine playback step: how many samples to advance per frame (may be
```

```
fractional)
        self.play_step = self.fs / self.fps
        # For manual frame-stepping, use nearest integer step
        self.hop = max(1, int(round(self.play_step)))

        # Filter and preprocess the data for visualization
        # Keep only channels that have known coordinates for mapping
        keep_idx = [i for i, nm in enumerate(ch_names) if nm in COORDS]
        self.ch = [ch_names[i] for i in keep_idx]
        X = detrend(data[:, keep_idx], axis=0, type='constant')  # remove DC
offset for each channel
        if notch_hz > 0:
            X = notch(X, self.fs, notch_hz)
        X = bandpass(X, self.fs, self.band[0], self.band[1])
        if use_laplacian:
            X = laplacian(X, self.ch)

        # Compute analytic signal for amplitude and phase
        _, self.A, self.P = analytic(X)
        self.n_samples = self.A.shape[0]
        self.duration = self.n_samples / self.fs

        # Determine robust amplitude range for colormap scaling (5th to 95th
percentile)
        self.amp_lo = np.percentile(self.A, 5)
        self.amp_hi = np.percentile(self.A, 95)

        # Generate the interpolation grid and head mask
        self.Xg, self.Yg, self.mask = make_grid(n=self.grid_n)

        # Playback state
        self.playing = False
        self.index = 0         # current sample index in the data
        self.play_index = 0.0  # float index for precise timing during playback
        self.timer = None      # Matplotlib timer for animation
        self.quiv = None       # store quiver plot for vectors

        # Build the figure and UI elements
        self._build_figure(dpi)

    def _build_figure(self, dpi):
        """Construct the Matplotlib figure, axis, and all UI widgets (slider,
buttons)."""
        self.fig = plt.figure(figsize=(9.5, 8.3), dpi=dpi)
        gs = self.fig.add_gridspec(12, 6, left=0.05, right=0.98, top=0.96,
bottom=0.09,
                                   hspace=0.6, wspace=0.3)
        ax = self.fig.add_subplot(gs[:10, :6])
```

```
        self.ax = ax
        ax.set_xticks([]); ax.set_yticks([])
        # Title showing mode, band, and time
        self.title = ax.set_title(self._title_text(0.0), fontsize=12)

        # Draw scalp outline as a circle for reference
        scalp_circle = plt.Circle((0, 0.15), 1.0, edgecolor=(0.65, 0.66, 0.74),
                                  facecolor='none', lw=1.0)
        ax.add_patch(scalp_circle)
        # Plot electrode locations as small white dots
        xs = []; ys = []
        for nm in self.ch:
            if nm in COORDS:
                x, y = COORDS[nm]
                xs.append(x); ys.append(y)
        ax.scatter(xs, ys, s=8, c='white', alpha=0.6, linewidths=0, zorder=3)

        # Phase image and amplitude image (overlayed) for wavefield
        self.im_phase = ax.imshow(np.zeros_like(self.Xg), origin='lower',
                                  extent=[self.Xg.min(), self.Xg.max(),
self.Yg.min(), self.Yg.max()],
                                  cmap='twilight', vmin=-np.pi, vmax=np.pi,
animated=True)
        self.im_amp = ax.imshow(np.zeros_like(self.Xg), origin='lower',
alpha=0.65,
                                extent=[self.Xg.min(), self.Xg.max(),
self.Yg.min(), self.Yg.max()],
                                cmap='magma', vmin=0, vmax=1, animated=True)

        # Time slider (for scrubbing through the recording)
        axt = self.fig.add_axes([0.08, 0.04, 0.72, 0.03])
        self.s_time = Slider(axt, "Time (s)", 0.0, self.duration, valinit=0.0)
        self.s_time.on_changed(self._on_slider)

        # Playback and skip buttons
        ax_play = self.fig.add_axes([0.82, 0.035, 0.06, 0.04])
        ax_back1 = self.fig.add_axes([0.08, 0.008, 0.06, 0.025])
        ax_back5 = self.fig.add_axes([0.145, 0.008, 0.06, 0.025])
        ax_fwd1  = self.fig.add_axes([0.725, 0.008, 0.06, 0.025])
        ax_fwd5  = self.fig.add_axes([0.79,  0.008, 0.06, 0.025])

        self.btn_play  = Button(ax_play, "Play")
        self.btn_back1 = Button(ax_back1, "⟲ 1s")
        self.btn_back5 = Button(ax_back5, "⟲ 5s")
        self.btn_fwd1  = Button(ax_fwd1, "1s ⟳")
        self.btn_fwd5  = Button(ax_fwd5, "5s ⟳")
```

```
        # Connect button events to their actions
        self.btn_play.on_clicked(lambda event: self.toggle_play())
        self.btn_back1.on_clicked(lambda event: self.seek_seconds(-1))
        self.btn_back5.on_clicked(lambda event: self.seek_seconds(-5))
        self.btn_fwd1.on_clicked(lambda event: self.seek_seconds(+1))
        self.btn_fwd5.on_clicked(lambda event: self.seek_seconds(+5))

        # Connect keyboard events
        self.fig.canvas.mpl_connect('key_press_event', self._on_key)

        # Set up a timer for animation (interval in milliseconds per frame)
        self.timer = self.fig.canvas.new_timer(interval=int(1000 / self.fps))
        self.timer.add_callback(self._tick)

        # Initial drawing of frame 0
        self._draw_index(self.index)

    def _title_text(self, t):
        """Format the title string showing current mode, band range, and
time."""
        mode_name = "Phase" if self.mode == "phase" else "Huygens"
        return f"EEG Wavefield — {mode_name.lower()} | {self.band[0]:g}–
{self.band[1]:g} Hz | t={t:0.2f}s"

    # --- Rendering and drawing functions ---
    def _amp_norm(self, a):
        """Normalize amplitude values to [0,1] for colormap, based on
precomputed 5-95% range."""
        return np.clip((a - self.amp_lo) / (self.amp_hi - self.amp_lo + 1e-9),
0, 1)

    def _draw_index(self, idx):
        """Update the display to show the data at sample index `idx`."""
        idx = int(np.clip(idx, 0, self.n_samples - 1))
        self.index = idx
        t = self.index / self.fs  # current time in seconds

        # Get amplitude and phase at this index for each channel
        amplitudes = self._amp_norm(self.A[idx, :])
        phases = self.P[idx, :]

        # Compute the wavefield on the grid based on current mode
        if self.mode == "phase":
            # Interpolate cosine and sine of phase to grid, then compute phase
angle
            R = griddata([COORDS[n] for n in self.ch], np.cos(phases), (self.Xg,
self.Yg), method='cubic')
            I = griddata([COORDS[n] for n in self.ch], np.sin(phases), (self.Xg,
```

```
self.Yg), method='cubic')
            phase_grid = np.arctan2(I, R)
            amp_grid   = griddata([COORDS[n] for n in self.ch], amplitudes,
(self.Xg, self.Yg), method='cubic')
        else:  # Huygens mode
            # Sum spherical waves from each electrode
            F = huygens(self.ch, amplitudes, phases, self.Xg, self.Yg, k=self.k)
            phase_grid = np.angle(F)
            amp_grid   = np.abs(F)
            # Normalize amplitude grid for display (to 0–1 based on 99th
percentile within head)
            amp_max = np.nanpercentile(amp_grid[self.mask], 99)
            if amp_max <= 0:
                amp_max = 1e-9
            amp_grid /= amp_max

        # Mask out points outside the head region for cleanliness
        Pm = np.where(self.mask, phase_grid, np.nan)
        Am = np.where(self.mask, amp_grid,   np.nan)

        # Update the image data on the plot
        self.im_phase.set_data(Pm)
        self.im_amp.set_data(Am)

        # Update or remove vector field arrows
        if self.show_vectors:
            # Remove old arrows if they exist
            if self.quiv is not None:
                self.quiv.remove()
            # Compute spatial gradient of phase to indicate flow direction
            gy, gx = np.gradient(Pm)
            mag = np.hypot(gx, gy) + 1e-6  # magnitude of gradient (add tiny
constant to avoid division by zero)
            # Sample a subset of points for arrows to avoid overcrowding
            step = max(1, Pm.shape[0] // 20)
            xs = self.Xg[::step, ::step];  ys = self.Yg[::step, ::step]
            vx = -gx[::step, ::step] / mag[::step, ::step]
            vy = -gy[::step, ::step] / mag[::step, ::step]
            # Plot quiver (arrows) with a fixed scaling
            self.quiv = self.ax.quiver(xs, ys, vx, vy, color=(0.95, 0.95, 1,
0.65),
                                       scale=30, width=0.003)
        else:
            # If vectors are turned off, ensure no quiver arrows remain
            if self.quiv is not None:
                self.quiv.remove()
                self.quiv = None
```

```
        # Update the title time (mode and band remain constant in title)
        self.title.set_text(self._title_text(t))
        # Update time slider position (avoid recursion by checking difference
threshold)
        if abs(self.s_time.val - t) > (0.25 / self.fs):
            self.s_time.set_val(t)

        # Request a redraw of the canvas
        self.fig.canvas.draw_idle()

    # --- Playback and interaction handlers ---
    def _tick(self):
        """Timer callback for updating the frame during playback."""
        if not self.playing:
            return  # If not playing, do nothing
        # Advance the play index by the fractional frame step
        self.play_index += self.play_step
        next_index = int(self.play_index)
        if next_index >= self.n_samples:
            # If we reached or exceeded the end, show last frame and stop
playback
            next_index = self.n_samples - 1
            self._draw_index(next_index)
            if self.playing:
                self.toggle_play()  # stop the playback
            return
        # Otherwise draw the next frame
        self._draw_index(next_index)

    def toggle_play(self):
        """Play/Pause toggle for the animation."""
        self.playing = not self.playing
        self.btn_play.label.set_text("Pause" if self.playing else "Play")
        if self.playing:
            # Start playing from current position
            self.play_index = float(self.index)
            self.timer.start()
        else:
            # Pause playback
            self.timer.stop()

    def seek_seconds(self, secs):
        """Jump forward/backward by a given number of seconds (negative for
backward)."""

# If currently playing and a seek is requested, pause playback (to avoid
conflict)
        if self.playing:
```

```
            self.toggle_play()  # pause
        # Compute target index and draw that frame
        target_idx = self.index + int(secs * self.fs)
        self._draw_index(target_idx)

    def _on_slider(self, val):
        """Handler for time-slider movement: jump to the selected time."""
        # When the slider is moved, update to the closest sample corresponding
to slider value (val is time in sec)
        self._draw_index(int(val * self.fs))

    def _on_key(self, event):
        """Handler for key press events (for keyboard shortcuts)."""
        key = (event.key or "").lower()
        if key == ' ':
            self.toggle_play()
        elif key == 'right':
            self.seek_seconds(+1.0)
        elif key == 'left':
            self.seek_seconds(-1.0)
        elif key == 'shift+right':
            self.seek_seconds(+5.0)
        elif key == 'shift+left':
            self.seek_seconds(-5.0)
        elif key == '.':
            # Step forward by one frame
            self._draw_index(self.index + self.hop)
        elif key == ',':
            # Step backward by one frame
            self._draw_index(self.index - self.hop)
        elif key == 'home':
            self._draw_index(0)
        elif key == 'end':
            self._draw_index(self.n_samples - 1)
        elif key == 'v':
            # Toggle vector field overlay
            self.show_vectors = not self.show_vectors
            if not self.show_vectors and self.quiv is not None:
                # Remove arrows if turning off
                self.quiv.remove()
                self.quiv = None
            self._draw_index(self.index)
        elif key == 'm':
            # Toggle visualization mode between phase and Huygens
            self.mode = "huygens" if self.mode == "phase" else "phase"
            self._draw_index(self.index)
        elif key == 's':
            # Save a screenshot of the current frame
```

```
            timestamp_ms = int(self.index / self.fs * 1000)
            fname = f"wavefield_{timestamp_ms:06d}ms.png"
            self.fig.savefig(fname, dpi=self.fig.dpi)
            print(f"Saved {fname}")

    def show(self):
        """Display the interactive plot (blocks until window is closed)."""
        plt.show()

# ----------------------------
# Command-line interface
# ----------------------------
def main():
    ap = argparse.ArgumentParser(description="Interactive EEG wavefield viewer
(high-res, with playback controls, Phase & Huygens modes)")
    ap.add_argument("-f", "--file", required=True, help="Path to EEG data file
(CSV with header or EDF format).")
    ap.add_argument("--fs", type=float, default=None,
help="Sampling rate of the data (required for CSV if not embedded).")
    ap.add_argument("--band", nargs=2, type=float, default=[4, 12],
help="Bandpass frequency range (low high) in Hz, e.g. --band 4 12")
    ap.add_argument("--notch", type=float, default=0.0, help="Notch filter
frequency (50 or 60 Hz common; 0 to disable).")
    ap.add_argument("--laplacian", action="store_true", help="Apply surface
Laplacian spatial filter to enhance local patterns.")
    ap.add_argument("--mode", choices=["phase", "huygens"], default="phase",
help="Initial mode: 'phase' (default) or 'huygens'.")
    ap.add_argument("--vectors", action="store_true", help="Show phase-gradient
flow vectors initially.")
    ap.add_argument("--fps", type=int, default=30, help="Frames per second for
playback.")
    ap.add_argument("--grid", type=int, default=256, help="Grid resolution
(pixels per side for interpolation).")
    ap.add_argument("--c", type=float, default=1.0, help="Phase velocity scale
(c) for Huygens mode (k = 2π f_c / c).")
    ap.add_argument("--dpi", type=int, default=150, help="Figure DPI for high-
resolution display.")
    args = ap.parse_args()

    # Load data from specified file
    path = args.file
    if path.lower().endswith(".edf"):
        data, ch_names, fs = load_edf(path)
    else:
        data, ch_names, inferred_fs = load_csv(path, args.fs)
        fs = args.fs or inferred_fs or 256.0  # if CSV has no explicit fs,
default to 256 Hz
```

```
    # Create and launch the viewer
    viewer = WavefieldViewer(
        data, ch_names, fs,
        band=tuple(args.band),
        notch_hz=args.notch,
        use_laplacian=args.laplacian,
        mode=args.mode,
        show_vectors=args.vectors,
        fps=args.fps,
        grid=args.grid,
        phase_c=args.c,
        dpi=args.dpi
    )
    viewer.show()

if __name__ == "__main__":
    main()
```

Notes on the implementation: This code uses standard Python libraries and idioms for clarity and maintainability: - The data loading functions handle CSV and EDF formats, mapping channel names to coordinates (with some aliases for variations in naming). Only channels with known positions are visualized. - A bandpass filter (default 4–12 Hz, e.g. alpha band) and optional notch filter or Laplacian can be applied to focus on certain rhythms or remove noise. - The analytic signal via Hilbert transform provides instantaneous phase and amplitude for each channel, which are then used to construct the wavefield. - The **Matplotlib** UI elements (slider and buttons) and key events are connected to methods for seamless interaction. The use of Matplotlib's timer ensures smooth animation at the requested FPS. - The figure DPI and grid resolution can be increased for higher quality. Even at high grid sizes (e.g. 512x512) and FPS (e.g. 120), the application remains reasonably smooth by using NumPy vectorized operations for heavy computations (interpolation and summing waves).

README

Below is a minimal `README.md` content for the EEG Wavefield Viewer, summarizing its purpose, requirements, usage, and controls:

EEG Wavefield Viewer
A Python application for visualizing EEG recordings as dynamic wave propagation across the scalp. It displays an interpolated wavefield of EEG phase and amplitude over a 2D head surface in real time, with interactive controls for playback and visualization modes.

Installation and Requirements

- **Python 3.x** with the following libraries: `numpy`, `pandas`, `matplotlib` (for the UI and plotting), `scipy` (for signal processing and interpolation). If viewing EDF files, install `pyedflib` as well (`pip install pyedflib`).
- Ensure these packages are installed before running the app.

Usage

```
python eeg_wavefield_viewer_ui2.py -f <EEG_data_file> [--fps <FPS>] [--grid
<res>] [--mode phase|huygens] [--vectors] [--band lo hi] [--notch F] [--
laplacian] [--dpi <DPI>]
```

Example:

```
python eeg_wavefield_viewer_ui2.py -f subject1.edf --fps 120 --grid 512
```

This will open the viewer for `subject1.edf` with a 120 FPS animation on a 512×512 grid (high detail). For CSV data, use `-f data.csv --fs 256` to specify the sampling rate if not known.

Optional arguments allow bandpass filtering (e.g., `--band 1 30` for 1–30 Hz), notch filtering (`--notch 50` to remove 50 Hz), applying a surface Laplacian (`--laplacian`), starting in Huygens mode (`--mode huygens`), showing vectors initially (`--vectors`), or adjusting figure DPI for higher resolution output (`--dpi 200`, etc.).

Controls and Interface

- **Play/Pause:** Click the **Play** button or press **Space** to start/stop the animation. The data will play back at the specified FPS, synced to real time based on the EEG sampling rate.
- **Time Slider:** Drag or click on the **Time (s)** slider at the bottom to jump to a specific time in the recording. The current time (in seconds) is displayed next to it. The slider also moves automatically during playback to indicate progress.
- **Rewind/Fast-Forward:** Use the on-screen buttons ⟲ **1s**, ⟲ **5s** to jump backward by 1s or 5s, and **1s** ⟳, **5s** ⟲ to jump forward. You can also use the **Left/Right Arrow** keys for ±1 second, and **Shift+Arrow** for ±5 seconds jumps. The **Home** key jumps to start, **End** to the end of the data.
- **Frame Step:** When paused, press **',' (comma)** to step one frame backward or **'.' (period)** to step one frame forward. This lets you inspect the wavefield frame-by-frame.
- **Visualization Mode:** Press **'m'** to toggle between **Phase** mode and **Huygens** mode. Phase mode shows the instantaneous phase (color) and amplitude (brightness) interpolated from the EEG sensors. Huygens mode simulates wave propagation by treating each sensor as a wave source (the phase speed can be adjusted with the `--c` parameter). The mode is also indicated in the window title.
- **Vector Field:** Press **'v'** to toggle the display of flow vectors. When on, arrows will overlay the map, pointing in the direction of wave propagation (phase increasing direction) with length indicating relative speed. This can be useful to visualize the movement of phase patterns. Press 'v' again to hide the vectors.
- **Screenshot:** Press **'s'** at any time to save a PNG screenshot of the current frame. The image file will be saved in the working directory (e.g., `wavefield_010000ms.png` for a frame at 10.000 seconds).
- **Exit:** Close the figure window or press `Ctrl+W` / `Alt+F4` to exit the application.

The interface displays a schematic head outline with small white dots for electrode positions to help orient the wavefield. The phase map is shown with a cyclic colormap (wrapping from –π to π), and the amplitude is

overlaid with a semi-transparent magma (heat) colormap—brighter areas indicate higher signal amplitude. All controls are designed to help you explore the spatio-temporal patterns in the EEG data easily and in high fidelity. Enjoy visualizing your EEG waves!

---

1 2 3 4 5 6 7 8 9 10 eeg_wavefield_viewer_ui2.py

file://file-3noRXjkv5uKRt1LHCAUznV