International Journal of Medical Science and Clinical Invention 13(06): 7888-7911, 2026
DOI:10.18535/ijmsci/v13i.06.01
https://valleyinternational.net/index.php/ijmsci

A Shape-sorting Tool to Maximise the Accuracy of Electrophysiological Estimation of Visual Acuity

A.M. Mackay[1], A. Sadler[1]

1. Flex Clin Sci Ltd, 45B Bewsey Street, Warrington, Cheshire, UK, WA2 7JQ

Abstract

Morphology of the Visual Evoked Potential (VEP) varies with age, pathology and the physical properties of the stimulus. For example, hydrocephalus can displace and compress the tissue of the visual brain, causing delays and distortion of measurements made from the standard Oz site. Steady-state stimuli enable quantitative analysis of VEPs by Fourier Transform and bespoke statistics, however, their rapidity increases the likelihood of standing waves and delayed far-field responses at the measurement electrodes, mimicking a contemporaneous response. This communication describes the development of new software that considers the morphology of an automatically detected VEP, compares it to a set of templates, then proposes its most likely origin. This tool could be used to reject non-spatial and other artefactual responses in exported data files, or added to existing real-time VEP VA tests enabling the stimulus presentation software to rapidly converge on the individuals threshold.

Keywords: Cross-correlation, Normalisation, Objective, VEP, Visual Acuity.

Introduction

The morphology of the normal VEP varies with stimulus pattern, [1] luminance, [2] field size, [3] onset vs reversal presentation, [4] contrast, [5] and colour. [6; 7] Maturity of the visual system influences VEP amplitude, which peaks during adolescence, [8; 9] though there is a lack of literature covering the teenage years in detail. Reductions in response amplitude are caused by fatigue, [10] uncorrected, or inadequately corrected refractive errors, [11] and voluntary movements when stimulation frequency falls between seven and 12 Hz. [12; 13] Prolonged latency has been observed in hypertension.[14]. Yet, there is a paucity of reports describing how such factors influence morphology.

In health, occipital ssVEPs are a combination of the V1/V2 response to visual stimulation (the near field) [15], responses from distant sources generated by an earlier visual stimulus (the far field) [16] and EEG standing waves [17] when the stimulation rate matches the brain's resonant frequency. [18] This resonance happens between eight and 12 Hz for visual stimuli, [19] is dependent on head size, [20] and is most prominent for full-field luminance flicker stimuli.[idem]

Travelling waves propagate from the occiput to the pre-frontal cortex, [21] and while they form the primary response at Oz, they may also contribute, to the signal at the reference electrode Fz, modulating the signal subject to individual anatomy, and recording duration.

Numerous clinical textbooks have been written about the effect of specific conditions on electrophysiological recordings, however, we have an incomplete understanding of many complex conditions- leading to the design of this project. For example, Cortical Visual Impairment (CVI) can cause hydrocephalus that displaces and compresses the tissue of the visual brain and causes a range of functional deficits.[22]. While imaging can reveal the extent of the anatomical damage, functional deficits require electrophysiological investigation with techniques that maintain attention.[23; 24] Prior imaging will guide re-positioning of VEP electrodes, and facilitate more accurate measurements. [25] Considered inversely, the vector linking a VEP from the standard O_z-F_z montage and an adjusted position provides an index of how specific anatomy changes the VEP. Creating a bank of such data could be used to retrospectively correct standard VEPs once neuro-anatomy is revealed by imaging.

While continuous, simultaneous EEG-MRI measurements are possible in specialist university labs, [26] the aim of this study is to develop a tool that can be used by healthcare professionals in a range of settings. The objectives are: 1) to create a database of VEP templates given what is already known about stimulus and clinical conditions; 2) to create an interface allowing users to add waveforms and detailed labels to this database; 3) to create software to allow threshold VEPs to be compared to the waveforms in the database.

Methods

Normal and Clinical VEPs were digitised and stored in Microsoft Excel Spreadsheets. Software was then developed in Python to read in the files in .csv format, normalise for amplitude then latency, and store three respective templates. Next, code was written to compare the morphology of a new, test waveform to a set of normal templates and calculate the index of cross-correlation for each comparison. (Equation 1). Dynamic Type Warping (DTW) was employed to represent these differences graphically. Algorithms were then developed to calculate the difference between concurrent standard VEP recording

and VEPs recorded from a site shifted anatomically given prior imaging. This applied to recordings that are normalised for amplitude only. The residual waveform is stored alongside metrics describing the displacement of the visual cortex (vector of translation) and the degree of hydrocephalus (qDESH and Intracranial pressure (ICP)) if known. A user-friendly interface was added to facilitate use by a range of professional groups.

Equation 1: $Cxy[t] = \sum x[t]y[t]$

Results

The code is included as Appendices 1-3 and was effective at reading in test waveforms, comparing them, and illustrating their simularity in numerical and graphical form. This provides the basis for a retrospective clinical analysis and the potential for real-time analysis if incorporated into existing software, e.g. the Step VEP.

Discussion

Where both Clinical VA and VEP VA estimations are available in the same patient, the relationship and agreement between the two measures has been described extensively [27][30] to support its diagnostic use when other tests are not possible, or unavailable. A reduction in Step VEP stimulus reversal rate has previously been proposed (in this journal) to optimise this relationship. [31] Here we have described analyses to allow further improvements in accuracy by rejection of responses that are unlikely to be spatial in origin, and by identifying and correcting for anatomical irregularity. The latter should increase the sensitivity of VEP VA assessment.

The effects of excessive noise cancellation and distortion secondary to traveling waves has been considered, but implementation of code to address this was beyond the scope of this study. This could lead to our code rejecting genuine waveforms that have been modulated (a redution of specificity). Evidence of an interaction between the VEP and the ongoing alpha-rhythm has been proven by other authors [32] and scaling multifocal VEP responses given underlying EEG activity reduced inter-subject variability. [33] Similar local scaling could be also be attempted in real-time VEP VA assessements, and may further increase sensitivity and specificity in response detection, enabling the software to display the most appropriate range of stimuli for the individual.

Acknowledgement

Thank you to Helen Sadler for recruiting her son, Alex, to expedite the technical aspects of this work. Also thanks to my parents for their continuing support. Finally, I would like to express my gratitude to local friends Geoff Boyle, and Keith and Karen Barlow, for providing coffee, curry, wine, and excellent company.

References

1. Yadav NK, Ludlam DP, Ciuffreda KJ. Effect of different stimulus configurations on the visual evoked potential (VEP). Doc Ophthalmol. 2012;124(3):177-196.
2. Fimreite V, Ciuffreda KJ, Yadav NK. Effect of luminance on the visually-evoked potential in visually-normal individuals and in mTBI/concussion. Brain Inj. 2015;29(10):1199-1210.
3. American Clinical Neurophysiology Society. Guideline 9B: Guidelines on visual evoked potentials. J Clin Neurophysiol. 2006;23(2):138-156. doi: 10.1097/00004691-200604000-00011
4. Šuštar Habjan M, Bach M, van Genderen MM, et al. ISCEV standard for clinical visual evoked potentials (2025 update). Doc Ophthalmol. 2025;151(2):97-112.
5. Souza GS, Gomes BD, Saito CA, da Silva Filho M, Silveira LC. Spatial luminance contrast sensitivity measured with transient VEP: comparison with psychophysics and evidence of multiple mechanisms. Invest Ophthalmol Vis Sci. 2007;48(7):3396-3404. 6. Chu L., Fernandez-Vargas J., Kita K., Yu W., Chen W., Hosoda K., Menegatti E., Shimizu M., Wang H. Influence of Stimulus Color on Steady State Visual Evoked Potentials. In: Advances in Intelligent Systems and Computing. ? EDs
6. Kaneko S, Kuriki I, Andersen SK, Peterzell DH. Individual variability in steady-state VEP responses for hues sweeping around cardinal color axes: Clues to cortical color coding?. J Vis. 2025;25(12):2.
7. Mackay A.M, Bradnam, M.S, Hamilton, R. Rapid detection of threshold VEPs, Clin. Neurophysiol. 2003; 114(6): 1009-1020.
8. Mahajan Y, McArthur G. Maturation of visual evoked potentials across adolescence. Brain Dev. 2012;34(8):655-666.
9. Balakrishnan A, Natarajan N. A comparative study on visual evoked potential in normotensive and hypertensive individuals. Natl J Physiol Pharm Pharmacol. 2018; 8(10): 1437-1440.
10. Cao T, Wan F, Wong CM, da Cruz JN, Hu Y. Objective evaluation of fatigue by EEG spectralanalysis in steady-state visual evoked potential-based brain-computer interfaces. Biomed Eng Online. 2014;13(1):28. 11. Kothari, R, Bokariya, P, Singh, S, Narang, P Singh, R. Refractive errors and their effects on visual evoked potentials. Journal of Clinical Ophthalmology and Research 2(1):3-6.
11. Lin Y-P, Wang Y, Wei C-S, and Jung T-P (2014) Assessing the quality of steady-state visual-evoked potentials for moving humans using a mobile electroencephalogram headset. Front. Hum.Neurosci. 8:182.
12. Rahman M, Karwowski W, Fafrowicz M and Hancock PA (2019) Neuroergonomics Applications of Electroencephalography in Physical Activities: A Systematic Review. Front. Hum. Neurosci.13:182.

13. Balakrishnan A, Natarajan N. A comparative study on visual evoked potential in normotensive and hypertensive individuals. Natl J Physiol Pharm Pharmacol 2018;8(10):1437-1440.

14. Zeki, S. et al. (1991). A direct demonstration of functional specialization in human visual cortex. J. Neurosci. 11:641-649.

15. Kimura, J. Electrodiagnosis in diseases of nerve and muscle. Principles and Practice. Fourth Edition. Oxford University Press. 2013. ISBN 978-0-19-973868-7.

16. Nunez, PL. Srinivasan, R. A theoretical basis for standing and traveling brain waves measured with human EEG with implications for an integrated consciousness. Clin. Neurophysiol. 2006; 117(11):2424-2435.

17. Cabral, J., Castaldo, F., Vohryzek, J. et al. Metastable oscillatory modes emerge from synchronization in the brain spacetime connectome. Commun Phys 5, 184 (2022).

18. Burkitt GR, Silberstein RB, Cadusch PJ, Wood AW. Steady-state visual evoked potentials and travelling waves. Clin Neurophysiol. 2000;111(2):246-258.

19. Nunez PL, Reid L, Bickford RG. The relationship of head size to alpha frequency with implications to a brain wave model. Electroencephalogr Clin Neurophysiol. 1978;44(3):344-352.

20. Thorpe SG, Nunez PL, Srinivasan R. Identification of wave-like spatial structure in the SSVEP: comparison of simultaneous EEG and MEG. Stat Med. 2007;26(21):3911-3926.

21. Bennett RG, Tibaudo ME, Mazel EC, Y N. Implications of cerebral/cortical visual impairment on life and learning: insights and strategies from lived experiences. Front Hum Neurosci. 2025;18:1496153.

22. Mackay, AM. Bradnam, MS. Hamilton, R. Elliot, AT. Dutton, GN. Real-Time Rapid AcuityAssessment Using VEPs: Development and Validation of the Step VEP Technique. Invest. Ophthalmol. Vis. Sci. 2008;49(1):438-441.

23. Tyler CW, Apkarian P, Levi DM, Nakayama K. Rapid assessment of visual function: an electronic sweep technique for the pattern visual evoked potential. Invest Ophthalmol Vis Sci. 1979;18(7):703-713.

24. Whittingstall K, Wilson D, Schmidt M, Stroink G. Correspondence of visual evoked potentials with FMRI signals in human visual cortex. Brain Topogr. 2008;21(2):86-92.

25. Bullock M, Jackson GD and Abbott DF. Artifact Reduction in Simultaneous EEG-fMRI: A Systematic Review of Methods and Contemporary Usage. Front. Neurol. 2021; 12:622719

26. Mackay AM. Step VEP visual acuity in a pediatric neuro-ophthalmological cohort. Int J Clin Exp Ophthalmol. 2022; 6: 026-030.

27. Mackay AM. VEP visual acuity in children with cortical visual impairment. Int J Clin Exp Ophthalmol. 2022; 6: 031-034.

28. Bach M, Maurer JP, Wolf ME. Visual evoked potential-based acuity assessment in normal vision, artificially degraded vision, and in patients. Br J Ophthalmol. 2008;92(3):396-403.

29. Hamilton R, Bach M, Heinrich SP, et al. VEP estimation of visual acuity: a systematic review. Doc Ophthalmol. 2021;142(1):25-74.

30. Mackay AM. Adjustments to Stimulation Frequency and Duration of STEP VEPs in Paediatric CVI. International Journal of Medical Science and Clinical Invention. 2023; 10 (7):6801-6803.

31. Risner ML, Aura CJ, Black JE, Gawne TJ. The Visual Evoked Potential is independent of surface alpha rhythm phase. Neuroimage. 2009;45(2):463-469.

32. Klistorner AI, Graham SL. Electroencephalogram-based scaling of multifocal visual evoked potentials: effect on intersubject amplitude variability. Invest Ophthalmol Vis Sci. 2001;42(9):2145-2152.

Appendix 1

VEP Analysis - MainApplication

A comprehensive tool for VEP waveform generation, analysis, and comparison

==========

FILE STRUCTURE

==========

vep-analysis/

├── vep_main.py # This file - main application

├── vep_generator.py # VEP generator with clinical norms

├── vep_comparison.py # Comparison functionality

├── data/ # For storing CSV files and results

│ ├── csv_files/

│ ├── results/

│ └── templates/

└── generate_test_csv.py # CSV generator

==========

MENU TREE:

==========

Main Menu:

├── 1. Generate VEP Waveforms

│ ├── 1. Clean VEP (ideal reference)

│ ├── 2. VEP with physiological variability

│ ├── 3. Averaged VEP (100 trials)

│ └── 4. Custom VEP parameters

│ ├── 1. Modify P100 parameters

│ └── 2. Modify all component parameters

```
│
│ Sub-menu for morphology:
│ ├── 1. Ideal (symmetric Gaussian)
│ └── 2. Asymmetric Gaussian (realistic rise/fall)
│
├── 2. Export Template VEP Data
├── 3. Load and Examine CSV File
│ ├── [List of available CSV files]
│ ├── Browse for file...
│ └── Load physiological measurement system file (optimized)
├── 4. Compare CSV to Synthetic VEP
| └── Shape-based Comparison (correlation analysis)
| ├── Normalization: max_abs, p100_peak, rms, max_pos
| └── Reference: clean, variable, averaged
├── 5. View Analysis History
├── 6. Settings and Configuration
│ ├── 1. Sampling Rate
│ ├── 2. Duration
│ ├── 3. Data Directory
│ ├── 4. Default Normalization
│ └── 5. Default Reference
├── 7. Troubleshoot CSV Format
├── 8. VEP Generator Demonstration
├── 9. Help and Documentation
└── 0. Exit
"""
import os
import sys
import glob
from pathlib import Path
import warnings
warnings.filterwarnings('ignore')
try:
from vep_generator import VEPGenerator
from vep_comparison import VEPComparator
except ImportError as e:
print(f"Error importing modules: {e}")
print("Make sure vep_generator.py and vep_comparison.py are in the same directory")
sys.exit(1)
def get_script_directory():
"""Get the directory where this script is located."""
return os.path.dirname(os.path.abspath(__file__))
class VEPAnalysisApp:
"""
Main application class for VEP analysis.
Provides a menu-driven interface for various VEP analysis functions.
"""
def __init__(self):
"""Initialize the application with default settings."""
self.generator = None
self.comparator = None
self.current_csv_file = None
self.results_history = []
# Get script directory for proper path handling
self.script_dir = get_script_directory()
# Default settings with paths relative to script location
self.settings = {
'sampling_rate': 1000,
'duration': 0.5,
'data_dir': os.path.join(self.script_dir, 'data'),
'results_dir': os.path.join(self.script_dir, 'data', 'results'),
'default_norm_method': 'max_abs',
'default_reference_type': 'clean'
}
# Create directories if they don't exist
self._setup_directories()
# Initialize generator and comparator
self._initialize_tools()
def _setup_directories(self):
"""Create necessary directories for data and results."""
dirs_to_create = [
self.settings['data_dir'],
self.settings['results_dir'],
os.path.join(self.settings['data_dir'], 'csv_files'),
os.path.join(self.settings['data_dir'], 'templates')
]
for directory in dirs_to_create:
Path(directory).mkdir(parents=True, exist_ok=True)
print(f"Data directories verified in: {self.settings['data_dir']}")
def _initialize_tools(self):
"""Initialize VEP generator and comparator with current settings."""
self.generator = VEPGenerator(
sampling_rate=self.settings['sampling_rate'],
duration=self.settings['duration']
)
self.comparator = VEPComparator(self.generator)
def display_main_menu(self):
```

```
"""Display the main application menu."""
print("\n" + "="*60)
print(" VEP ANALYSIS")
print("="*60)
print("1. Generate VEP Waveforms")
print("2. Export Template VEP Data")
print("3. Load and Examine CSV File")
print("4. Compare CSV to Synthetic VEP")
print("5. View Analysis History")
print("6. Settings and Configuration")
print("7. Troubleshoot CSV Format")
print("8. VEP Generator Demonstration")
print("9. Help and Documentation")
print("0. Exit")
print("="*60)
if self.current_csv_file:
print(f"Current CSV: {os.path.basename(self.current_csv_file)}")
# Show relative path for cleaner display
rel_data_dir = os.path.relpath(self.settings['data_dir'], os.getcwd())
print(f"Data Directory: {rel_data_dir}")
def run(self):
"""Main application loop."""
print("Welcome to the VEP Analysis!")
print(f"Working directory: {self.script_dir}")
while True:
try:
self.display_main_menu()
choice = input("\nSelect option (0-9): ").strip()
if choice == '0':
print("Thank you for using VEP Analysis!")
break
elif choice == '1':
self.generate_vep_waveforms()
elif choice == '2':
self.export_template_data()
elif choice == '3':
self.load_csv_file()
elif choice == '4':
self.compare_csv_to_synthetic()
elif choice == '5':
self.view_history()
elif choice == '6':
self.settings_menu()
elif choice == '7':
self.troubleshoot_csv()
elif choice == '8':
self.vep_demonstration()
elif choice == '9':
self.show_help()
else:
print("Invalid choice. Please try again.")
input("\nPress Enter to continue...")
except KeyboardInterrupt:
print("\n\nExiting...")
break
except Exception as e:
print(f"An error occurred: {e}")
input("Press Enter to continue...")
def generate_vep_waveforms(self):
"""Generate and display VEP waveforms with morphology options."""
print("\n" + "="*50)
print("GENERATE VEP WAVEFORMS")
print("="*50)
print("VEP Types:")
print("1. Clean VEP (ideal reference)")
print("2. VEP with physiological variability")
print("3. Averaged VEP (100 trials)")
print("4. Custom VEP parameters")
vep_choice = input("\nSelect VEP type (1-4): ").strip()
if vep_choice not in ['1', '2', '3', '4']:
print("Invalid choice.")
return
# For options 1-3, ask for morphology type
if vep_choice in ['1', '2', '3']:
print("\nMorphology Options:")
print("1. Ideal (symmetric Gaussian)")
print("2. Asymmetric Gaussian (realistic rise/fall)")
morph_choice = input("Select morphology (1-2): ").strip()
morphology_map = {
'1': 'gaussian',
'2': 'asymmetric_gaussian'
}
morphology = morphology_map.get(morph_choice, 'asymmetric_gaussian')
print(f"\nGenerating {['Clean', 'Variable', 'Averaged'][int(vep_choice)-1]} VEP with {morphology} morphology...")
try:
if vep_choice == '1':
```

```
# Clean VEP
vep = self.generator.generate_baseline_vep(add_variability =False, morphology=morphology)
title = f'Clean VEP ({morphology.replace("_", " ").title()} Morphology)'
self.generator.plot_vep(vep, title=title)
elif vep_choice == '2':
# VEP with variability
vep = self.generator.generate_baseline_vep(add_variability=True, morphology=morphology)
title = f'VEP with Variability ({morphology.replace("_", " ").title()} Morphology)'
self.generator.plot_vep(vep, title=title)
elif vep_choice == '3':
# Averaged VEP
n_trials = int(input("Enter number of trials (default 100): ") or "100")
trials, averaged = self.generator.generate_trial_batch(
n_trials=n_trials,
add_noise=False,
morphology=morphology
)
title = f'Averaged VEP - {n_trials} trials ({morphology.replace("_", " ").title()})'
# Show comparison plot
self.generator.plot_trials_comparison(trials, averaged, morphology)
elif vep_choice == '4':
# Custom parameters
self.custom_vep_parameters()
except ValueError as e:
print(f"Invalid input: {e}")
except Exception as e:
print(f"Error generating VEP: {e}")
def custom_vep_parameters(self):
"""Generate VEP with custom clinical parameters."""
print("\n" + "="*40)
print("CUSTOM VEP PARAMETERS")
print("="*40)
print("Choose parameter modification:")
print("1. Modify P100 parameters")
print("2. Modify all component parameters")
choice = input("Select option (1-2): ").strip()
try:
if choice == '1':
self._modify_p100_parameters()
elif choice == '2':
self._modify_all_parameters()
else:
print("Invalid choice.")
except Exception as e:
print(f"Error in custom parameters: {e}")
def _modify_p100_parameters(self):
"""Modify P100 component parameters."""
print("\nModify P100 Parameters (press Enter to keep current):")
current_p100 = self.generator.clinical_norms['P100']
# P100 latency
lat_input = input(f"P100 latency mean (ms) [current: {current_p100['latency_mean']}]: ")
if lat_input:
self.generator.clinical_norms['P100']['latency_mean'] = float(lat_input)
# P100 amplitude
amp_input = input(f"P100 amplitude mean (µV) [current: {current_p100['amplitude_mean']}]: ")
if amp_input:
self.generator.clinical_norms['P100']['amplitude_mean'] = float(amp_input)
# Variability
sd_input = input(f"P100 latency SD (ms) [current: {current_p100['latency_sd']}]: ")
if sd_input:
self.generator.clinical_norms['P100']['latency_sd'] = float(sd_input)
# Select morphology
print("\nMorphology:")
print("1. Gaussian 2. Asymmetric")
morph_choice = input("Select morphology (1-2): ").strip()
morphologies = {'1': 'gaussian', '2': 'asymmetric_gaussian'}
morphology = morphologies.get(morph_choice, 'asymmetric_gaussian')
# Generate and plot
vep = self.generator.generate_baseline_vep(add_variability=True, morphology=morphology)
title = f'Custom P100 VEP - Lat:{self.generator.clinical_norms["P100"]["latency_mean"]}ms, Amp:{self.generator.clinical_norms["P100"]["amplitude_mean"]}µV'
self.generator.plot_vep(vep, title=title)
def _modify_all_parameters(self):
"""Modify all component parameters."""
print("\nModify All Component Parameters:")
print("(press Enter to keep current values)")
```

```
for comp_name in ['N75', 'P100', 'N135']:
print(f"\n{comp_name} Component:")
current = self.generator.clinical_norms[comp_name]
# Latency
lat_input = input(f" Latency mean (ms) [current: {current['latency_mean']}]: ")
if lat_input:
self.generator.clinical_norms[comp_name]['latency_mean'] = float(lat_input)
# Amplitude
amp_input = input(f" Amplitude mean (µV) [current: {current['amplitude_mean']}]: ")
if amp_input:
self.generator.clinical_norms[comp_name]['amplitude_mean'] = float(amp_input)
# Select morphology
print("\nMorphology:")
print("1. Gaussian 2. Asymmetric")
morph_choice = input("Select morphology (1-2): ").strip()
morphologies = {'1': 'gaussian', '2': 'asymmetric_gaussian'}
morphology = morphologies.get(morph_choice, 'asymmetric_gaussian')
# Generate and plot
vep = self.generator.generate_baseline_vep(add_variability=True, morphology=morphology)
self.generator.plot_vep(vep, title=f'Custom All Components VEP ({morphology.replace("_", " ").title()})')
def export_template_data(self):
"""Export template VEP data for reference."""
print("\n" + "="*50)
print("EXPORT TEMPLATE VEP DATA")
print("="*50)
output_dir = os.path.join(self.settings['data_dir'], 'templates')
print("Generating template VEP data...")
try:
# Generate different types of VEP data with different morphologies
templates = {}
for morphology in ['gaussian', 'asymmetric_gaussian']:
morph_name = morphology.replace('_', '-')
templates[f'clean-vep-{morph_name}.csv'] = self.generator.generate_baseline_vep(
add_variability=False, morphology=morphology)
templates[f'variable-vep-{morph_name}.csv'] = self.generator.generate_baseline_vep(
add_variability=True, morphology=morphology)
# Generate averaged VEP
_, avg_vep = self.generator.generate_trial_batch(
n_trials=100, add_noise=False, morphology=morphology)
templates[f'averaged-vep-{morph_name}.csv'] = avg_vep
# Export to CSV files
import pandas as pd
time_ms = self.generator.times * 1000
for filename, vep_data in templates.items():
filepath = os.path.join(output_dir, filename)
df = pd.DataFrame({
'Time_ms': time_ms,
'Amplitude_uV': vep_data
})
df.to_csv(filepath, index=False)
print(f"Exported: {filename}")
# Show relative path for cleaner display
rel_output_dir = os.path.relpath(output_dir, os.getcwd())
print(f"\nTemplate files saved to: {rel_output_dir}")
print(f"Generated {len(templates)} template files with 3 morphology types")
except Exception as e:
print(f"Error exporting templates: {e}")
def load_csv_file(self):
"""Load and examine a CSV file."""
print("\n" + "="*50)
print("LOAD AND EXAMINE CSV FILE")
print("="*50)
# Show available CSV files
csv_pattern = os.path.join(self.settings['data_dir'], "**/*.csv")
csv_files = glob.glob(csv_pattern, recursive=True)
if csv_files:
print("Available CSV files:")
for i, file in enumerate(csv_files, 1):
rel_path = os.path.relpath(file, self.settings['data_dir'])
print(f"{i:2d}. {rel_path}")
print(f"{len(csv_files)+1:2d}. Browse for file...")
print(f"{len(csv_files)+2:2d}. Load physiological measurement system file (optimized)")
else:
print("No CSV files found in data directory.")
print("1. Browse for file...")
print("2. Load physiological measurement system file (optimized)")
print("\nTip: Run option 2 (Export Template Data) to create test files!")
try:
```

```
max_choice = len(csv_files) + 2 if csv_files else 2
choice = input(f"\nSelect file (1-{max_choice}): ").strip()
if csv_files and choice.isdigit() and 1 <= int(choice) <= len(csv_files):
filepath = csv_files[int(choice)-1]
# Use standard loading
real_data = self.comparator.load_csv_data(filepath)
elif choice == str(len(csv_files)+1 if csv_files else 1):
filepath = input("Enter full path to CSV file: ").strip()
if not os.path.exists(filepath):
print(f"File not found: {filepath}")
return
real_data = self.comparator.load_csv_data(filepath)
elif choice == str(len(csv_files)+2 if csv_files else 2):
# Optimized loading for physiological measurement systems
filepath = input("Enter path to physiological measurement CSV: ").strip()
if not os.path.exists(filepath):
print(f"File not found: {filepath}")
return
print("Loading with physiological measurement system optimizations...")
real_data = self.load_physiological_csv(filepath)
else:
print("Invalid choice.")
return
self.current_csv_file = filepath
print(f"\nFile loaded successfully!")
print(f"Duration: {real_data['duration']:.3f}s")
print(f"Samples: {real_data['n_samples']}")
print(f"Sampling Rate: {real_data['sampling_rate']:.1f}Hz")
# Plot the loaded data
import matplotlib.pyplot as plt
time_ms = real_data['time'] * 1000
plt.figure(figsize=(12, 6))
plt.plot(time_ms, real_data['amplitude'], 'b-', linewidth=2)
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude (µV)')
plt.title(f'Loaded VEP Data: {os.path.basename(filepath)}')
plt.grid(True, alpha=0.3)
plt.show()
except Exception as e:
print(f"Error loading CSV file: {e}")
def load_physiological_csv(self, filepath):
"""
Optimized loading for physiological measurement system CSV files.
Handles formats like: "1ms; 0.00001V"
"""
print("Detecting physiological measurement format...")
# First, examine the file structure
with open(filepath, 'r') as f:
first_lines = [f.readline().strip() for _ in range(5)]
print("First few lines:")
for i, line in enumerate(first_lines, 1):
print(f" {i}: {line}")
# Detect separator and format
separators = [';', ',', '\t']
separator = ';' # Default for physiological systems
for sep in separators:
if sep in first_lines[0]:
separator = sep
break
print(f"Using separator: '{separator}'")
# Check if first line looks like header or data
has_header = not any(char.isdigit() for char in first_lines[0].split(separator)[0])
print(f"Has header: {has_header}")
try:
# Load with detected parameters
real_data = self.comparator.load_csv_data(
filepath=filepath,
time_col=0, # First column is time
data_col=1, # Second column is voltage
time_unit='ms', # Assume milliseconds
has_header=has_header
)
print("Successfully loaded physiological measurement data")
return real_data
except Exception as e:
print(f"Error with optimized loading: {e}")
# Fallback to standard loading
print("Trying standard loading...")
return self.comparator.load_csv_data(filepath)
def compare_csv_to_synthetic(self):
"""Compare loaded CSV file - choose comparison type."""
if not self.current_csv_file:
print("No CSV file loaded. Please load a file first (option 3).")
return
```

```
print("\n" + "="*50)
print("VEP ANALYSIS OPTIONS")
print("="*50)
print(f"Current file: {os.path.basename(self.current_csv_file)}")
# Only shape-based comparison is implemented in this version
self.shape_based_comparison()
def shape_based_comparison(self):
"""Original shape-based comparison method."""
print("\n" + "="*50)
print("SHAPE-BASED COMPARISON")
print("="*50)
# Comparison options
print("\nNormalization methods:")
print("1. Maximum absolute value")
print("2. P100 peak (90-120ms region)")
print("3. RMS value")
print("4. Maximum positive value")
norm_methods = {
'1': 'max_abs',
'2': 'p100_peak',
'3': 'rms',
'4': 'max_pos'
}
norm_choice = input("Select normalization method (1-4): ").strip()
norm_method = norm_methods.get(norm_choice, 'max_abs')
print("\nReference waveform types:")
print("1. Clean (no variability)")
print("2. With physiological variability")
print("3. Averaged (100 trials)")
ref_types = {
'1': 'clean',
'2': 'variable',
'3': 'averaged'
}
ref_choice = input("Select reference type (1-3): ").strip()
ref_type = ref_types.get(ref_choice, 'clean')
try:
# Load the current CSV data
real_data = self.comparator.load_csv_data(self.current_csv_file)
# Perform comparison
print(f"\nPerforming shape-based comparison...")
print(f"Normalization: {norm_method}")
print(f"Reference: {ref_type}")
comparison = self.comparator.compare_with_reference(
real_data,
norm_method=norm_method,
reference_type=ref_type,
plot=True
)
# Store result in history
result_summary = {
'file': self.current_csv_file,
'analysis_type': 'shape_based',
'norm_method': norm_method,
'reference_type': ref_type,
'quality': comparison['quality_assessment']['overall'],
'shape_score': comparison['quality_assessment']['shape_score'],
'pearson_r': comparison['metrics']['pearson_r'],
'rmse': comparison['metrics']['rmse']
}
self.results_history.append(result_summary)
# Display summary
print(f"\n" + "="*40)
print("SHAPE-BASED COMPARISON SUMMARY")
print("="*40)
print(f"Quality: {comparison['quality_assessment']['overall']}")
print(f"Shape Score: {comparison['quality_assessment']['shape_score']:.3f}")
print(f"Pearson r: {comparison['metrics']['pearson_r']:.3f}")
print(f"RMSE: {comparison['metrics']['rmse']:.3f}")
print(f"Cosine Similarity: {comparison['metrics']['cosine_similarity']:.3f}")
# Save results option
save_choice = input("\nSave detailed results to file? (y/n): ").strip().lower()
if save_choice == 'y':
self.save_comparison_results(comparison)
except Exception as e:
print(f"Error performing shape-based comparison: {e}")
def view_history(self):
"""Updated view history to handle both analysis types."""
print("\n" + "="*50)
print("ANALYSIS HISTORY")
print("="*50)
if not self.results_history:
print("No analysis history available.")
return
```

```
# Group by analysis type for better organization
shape_based = [r for r in self.results_history if r.get('analysis_type') == 'shape_based']
if shape_based:
print("\nSHAPE-BASED COMPARISONS:")
print(f"{'#':<3} {'File':<25} {'Quality':<10} {'Score':<6} {'r':<6} {'RMSE':<6}")
print("-" * 60)
for i, result in enumerate(shape_based, 1):
filename = os.path.basename(result['file'])[:24]
print(f"{i:<3} {filename:<25} {result['quality']:<10} "
f"{result['shape_score']:<6.3f} {result['pearson_r']:<6.3f} "
f"{result['rmse']:<6.3f}")
def settings_menu(self):
"""Display and modify application settings."""
print("\n" + "="*50)
print("SETTINGS AND CONFIGURATION")
print("="*50)
while True:
print(f"\nCurrent Settings:")
print(f"1. Sampling Rate: {self.settings['sampling_rate']} Hz")
print(f"2. Duration: {self.settings['duration']} s")
# Show relative path for cleaner display
rel_data_dir = os.path.relpath(self.settings['data_dir'], os.getcwd())
print(f"3. Data Directory: {rel_data_dir}")
print(f"4. Default Normalization: {self.settings['default_norm_method']}")
print(f"5. Default Reference: {self.settings['default_reference_type']}")
print(f"0. Back to main menu")
choice = input("\nSelect setting to change (0-5): ").strip()
if choice == '0':
break
elif choice == '1':
try:
new_rate = int(input(f"Enter sampling rate (current: {self.settings['sampling_rate']}): "))
self.settings['sampling_rate'] = new_rate
self._initialize_tools() # Reinitialize with new settings
print("Sampling rate updated.")
except ValueError:
print("Invalid input.")
elif choice == '2':
try:
new_duration = float(input(f"Enter duration in seconds (current: {self.settings['duration']}): "))
self.settings['duration'] = new_duration
self._initialize_tools() # Reinitialize with new settings
print("Duration updated.")
except ValueError:
print("Invalid input.")
elif choice == '3':
print("Data directory is fixed relative to script location.")
print(f"Current absolute path: {self.settings['data_dir']}")
elif choice == '4':
print("1=max_abs, 2=p100_peak, 3=rms, 4=max_pos")
norm_choice = input("Select default normalization: ")
norm_map = {'1': 'max_abs', '2': 'p100_peak', '3': 'rms', '4': 'max_pos'}
if norm_choice in norm_map:
self.settings['default_norm_method'] = norm_map[norm_choice]
print("Default normalization updated.")
elif choice == '5':
print("1=clean, 2=variable, 3=averaged")
ref_choice = input("Select default reference: ")
ref_map = {'1': 'clean', '2': 'variable', '3': 'averaged'}
if ref_choice in ref_map:
self.settings['default_reference_type'] = ref_map[ref_choice]
print("Default reference updated.")
def troubleshoot_csv(self):
"""Help troubleshoot CSV format issues."""
print("\n" + "="*50)
print("CSV FORMAT TROUBLESHOOTING")
print("="*50)
filepath = input("Enter path to problematic CSV file: ").strip()
if not os.path.exists(filepath):
print(f"File not found: {filepath}")
return
try:
# Examine file structure
print(f"\nExamining: {filepath}")
print("-" * 40)
# Read first few lines
with open(filepath, 'r') as f:
lines = f.readlines()[:10]
print("First 10 lines:")
for i, line in enumerate(lines, 1):
print(f"{i:2d}: {line.strip()}")
```

```
# Analyze separators
print(f"\nSeparator Analysis:")
first_line = lines[0].strip()
separators = [',', ';', '\t', '|']
for sep in separators:
count = first_line.count(sep)
if count > 0:
parts = first_line.split(sep)
print(f" '{sep}': {count} occurrences -> {len(parts)} parts")
if len(parts) == 2:
print(f" Good for 2-column data")
print(f" Part 1: '{parts[0]}'")
print(f" Part 2: '{parts[1]}'")
# Try pandas analysis with different methods
import pandas as pd
print(f"\nPandas Reading Tests:")
methods = [
("Default", {}),
("Semicolon + Python engine", {"sep": ";", "engine": "python"}),
("Semicolon + Default engine", {"sep": ";"}),
("No header + Semicolon", {"sep": ";", "header": None}),
]
for method_name, kwargs in methods:
try:
df = pd.read_csv(filepath, **kwargs)
print(f" {method_name}: Success - Shape {df.shape}")
if df.shape[1] >= 2:
print(f" Columns: {list(df.columns)}")
print(f" Sample data: {df.iloc[0].tolist()}")
else:
print(f" Warning: Only {df.shape[1]} column(s)")
except Exception as e:
print(f" {method_name}: Failed - {str(e)[:50]}...")
# Manual parsing test for semicolon data
if ';' in first_line:
print(f"\nManual Semicolon Parsing Test:")
try:
parsed_data = []
for i, line in enumerate(lines[:5]):
line = line.strip()
if ';' in line:
parts = line.split(';')
if len(parts) >= 2:
parsed_data.append([parts[0], parts[1]])
print(f" Row {i+1}: '{parts[0]}' | '{parts[1]}'")
if parsed_data:
print(f" Manual parsing successful for {len(parsed_data)} rows")
else:
print(f" No valid rows found")
except Exception as e:
print(f" Manual parsing failed: {e}")
# Unit detection
print(f"\nUnit Detection:")
time_units = ['ms', 's', 'samples']
voltage_units = ['V', 'mV', 'uV', 'µV']
text_content = ''.join(lines[:5])
detected_time_units = [unit for unit in time_units if unit in text_content]
detected_voltage_units = [unit for unit in voltage_units if unit in text_content]
print(f" Time units detected: {detected_time_units}")
print(f" Voltage units detected: {detected_voltage_units}")
# Provide specific recommendations
print(f"\n" + "="*40)
print("RECOMMENDATIONS:")
print("="*40)
if ';' in first_line:
print("SEMICOLON FORMAT DETECTED:")
print(" Your file uses semicolons (common in physiological systems)")
print(" Solutions:")
print(" 1. Try the 'Load physiological measurement system file' option")
print(" 2. Use manual parsing in the comparison tool")
if any(unit in text_content for unit in time_units + voltage_units):
print("UNITS IN DATA DETECTED:")
print(" Your data contains units (ms, V, etc.) embedded in values")
print(" The system should automatically remove these")
print("\nIf issues persist:")
print(" • Try creating new CSV with Export Template Data (option 2)")
print(" • Use Load physiological measurement system file (option 3.2)")
print(" • Check that your CSV has exactly 2 columns")
except Exception as e:
print(f"Error examining file: {e}")
print("\nBasic checks:")
print("• File exists and is readable")
print("• Contains numeric data")
```

print("• Has consistent separator throughout")

print("• No extra quotes or special characters")

def vep_demonstration(self):

"""Run the VEP generator demonstration."""

print("\n" + "="*50)

print("VEP GENERATOR DEMONSTRATION")

print("="*50)

print("This will run the full VEP generator demonstration...")

proceed = input("Continue? (y/n): ").strip().lower()

if proceed == 'y':

try:

Import and run demonstration from original file

from vep_generator import demonstrate_vep_generator

demonstrate_vep_generator()

except Exception as e:

print(f"Error running demonstration: {e}")

def show_help(self):

"""Display comprehensive help and documentation."""

help_text = """

VEP ANALYSIS - HELP & DOCUMENTATION

OVERVIEW:

This application provides comprehensive tools for Visual Evoked Potential (VEP)

analysis, including synthetic waveform generation, data loading, and clinical

assessment capabilities.

MENU OPTIONS EXPLAINED:

═══

1. GENERATE VEP WAVEFORMS

═══

Creates synthetic VEP waveforms for testing and reference purposes.

Sub-options:

- Clean VEP: Perfect theoretical waveform with no noise or variability

- Variable VEP: Realistic waveform with physiological variability added

- Averaged VEP: Simulates multiple trials averaged together (reduces noise)

- Custom VEP: Modify clinical parameters (latencies, amplitudes) manually

Morphology Types:

- Gaussian: Symmetric mathematical model (ideal but unrealistic)

- Asymmetric Gaussian: More realistic with different rise/fall times

Use When: Testing analysis methods, creating reference data, education

═══

2. EXPORT TEMPLATE VEP DATA

═══

Generates and saves standard VEP waveforms as .csv files for testing.

What It Does:

- Creates clean, variable, and averaged VEPs for each morphology type

- Saves as properly formatted CSV files in data/templates/ directory

- Generates 9 template files total (3 types × 3 morphologies)

Use When: Need test data for analysis, validating CSV loading, demonstration

═══

3. LOAD AND EXAMINE CSV FILE

═══

Loads VEP data from CSV files and displays basic information.

Features:

- Lists all available CSV files in data directory

- Manual file browsing option

- Specialized loading for physiological measurement systems

- Automatic format detection and unit conversion

- Visual plot of loaded waveform

- Handles various separators (comma, semicolon, tab)

Supported Formats:

- Standard: "Time_ms, Amplitude_uV"

- Physiological: "1ms; 0.00001V"

- Time units: ms, s, sample indices (auto-detected)

- Voltage units: V, mV, µV (auto-converted to µV)

Use When: Loading real VEP recordings, examining data quality, preparation for analysis

4. COMPARE CSV TO SYNTHETIC VEP

Main analysis engine - compares loaded VEP data using two methods.

Analysis Options:

A) SHAPE-BASED COMPARISON (Research/Development):

- Correlation analysis against synthetic references
- Multiple normalization methods (max_abs, p100_peak, rms, max_pos)
- Shape similarity scoring (Pearson r, cosine similarity, RMSE)
- Good for waveform validation and research applications

B) CLINICAL ASSESSMENT (Medical/Diagnostic):

- Extracts N75, P100, N135 component parameters
- Compares against clinical normative ranges
- Calculates Z-scores and percentiles
- Provides clinical interpretation (normal/borderline/abnormal)
- Generates diagnostic recommendations
- Matches hospital/clinical workflow

C) BOTH ANALYSES:

- Runs complete evaluation with both methods
- Comprehensive report for research and clinical use

Use When: Analyzing patient VEP recordings, research studies, clinical diagnosis

5. VIEW ANALYSIS HISTORY

Displays summary of all previous analyses performed in current session.

What It Shows:

- Shape-based results: quality scores, correlation values, RMSE
- Clinical results: severity assessments, P100 parameters, Z-scores
- File names and analysis parameters used
- Organized by analysis type for easy review

Use When: Reviewing previous results, comparing multiple files, session summary

6. SETTINGS AND CONFIGURATION

Modify application parameters and default behaviors.

Configurable Settings:

- Sampling Rate: Hz for generated waveforms (default: 1000 Hz)
- Duration: Length of generated waveforms (default: 0.5 s)
- Data Directory: Location for CSV files (fixed relative to script)
- Default Normalization: Method for shape-based comparisons
- Default Reference: Type of synthetic reference waveform

Use When: Adapting to different recording systems, changing analysis defaults

7. TROUBLESHOOT CSV FORMAT

Diagnostic tool for problematic CSV files.

What It Does:

- Examines file structure and content
- Tests different separator detection methods
- Identifies encoding and formatting issues
- Shows parsing attempts with multiple methods
- Provides specific recommendations for fixes

Common Issues It Helps With:

- Wrong separator (semicolon vs comma)
- Units embedded in data ("1ms", "0.5V")
- Header detection problems
- Encoding issues
- Multiple columns or unexpected format

Use When: CSV files won't load properly, format errors, data import problems

8. VEP GENERATOR DEMONSTRATION

Runs comprehensive demonstration of VEP generation capabilities.

What It Shows:

- All morphology types with comparisons
- Effect of variability and noise
- Individual trials vs averaged responses
- Clinical parameter variations
- Full plotting and visualization suite

Use When: Learning about VEPs, understanding morphology differences, education

═══

9. HELP AND DOCUMENTATION

═══

This help system - comprehensive documentation and usage guide.

═══

0. EXIT

═══

Safely exits the application.

═══

TYPICAL WORKFLOWS:

FOR LEARNING/EDUCATION:

1 → 8 → 2 → 3 → 4 (Generate → Demo → Export → Load → Analyze)

FOR CLINICAL USE:

3 → 4 (Clinical Assessment) → 5 (Load patient data → Diagnose → Review)

FOR RESEARCH/DEVELOPMENT:

1 → 2 → 3 → 4 (Shape-based) → 6 (Generate → Export → Load → Analyze → Configure)

FOR TROUBLESHOOTING:

7 → 3 → 4 (Fix format → Load → Analyze)

═══

FILE LOCATIONS:

- Input CSV files: vep-analysis/data/csv_files/
- Template files: vep-analysis/data/templates/
- Analysis results: vep-analysis/data/results/
- All paths relative to script location

TECHNICAL REQUIREMENTS:

- Python 3.6+ with matplotlib, pandas, numpy, scipy
- CSV files: 2 columns (time, amplitude)
- Recommended sampling rate: 1000+ Hz
- Typical VEP duration: 200-500 ms

For technical details and source code documentation, see individual module files:

- vep_generator.py - Waveform generation
- vep_comparison.py - Shape-based analysis
- vep_clinical_assessor.py - Clinical assessment

═══

"""

print(help_text)

def save_comparison_results(self, comparison):

"""Save detailed comparison results to file."""

try:

import json

import datetime

timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

filename = f"vep_comparison_{timestamp}.json"

filepath = os.path.join(self.settings['results_dir'], filename)

Prepare data for JSON serialization

result_data = {

'timestamp': timestamp,

'file_analyzed': comparison['real_data']['filepath'],

'settings': {

'normalization_method': comparison['normalized']['method'],

'reference_type': comparison['reference']['type']

},

'metrics': comparison['metrics'],

'quality_assessment': comparison['quality_assessment']

}

with open(filepath, 'w') as f:

json.dump(result_data, f, indent=2)

Show relative path for cleaner display

rel_filepath = os.path.relpath(filepath, os.getcwd())

print(f"Results saved to: {rel_filepath}")

except Exception as e:

print(f"Error saving results: {e}")

Main execution

def main():

"""Main entry point for the VEP Analysis."""

```
try:
app = VEPAnalysisApp()
app.run()
except Exception as e:
print(f"Fatal error: {e}")
input("Press Enter to exit...")
if __name__ == "__main__":
main()
```

Appendix 2

```
Simple VEP Test CSV Generator
Creates clean, ideal VEP waveforms for testing using Gaussian components
Note: Component amplitudes are adjusted to compensate for Gaussian overlap.
When multiple Gaussians sum together, their tails interact, reducing peak values.
The adjusted amplitudes ensure the actual peaks match clinical expectations:
- P100 peak: ~10 µV
- N75 trough: ~-4 µV
- N135 trough: ~-8 µV
"""
import numpy as np
import os
from pathlib import Path
def create_data_folders():
"""Create the necessary data folder structure."""
script_dir = os.path.dirname(os.path.abspath(__file__))
folders = [
os.path.join(script_dir, 'data'),
os.path.join(script_dir, 'data', 'csv_files'),
os.path.join(script_dir, 'data', 'results')
]
for folder in folders:
Path(folder).mkdir(parents=True, exist_ok=True)
print(f"□ Data folders created/verified in: {os.path.join(script_dir, 'data')}")
return os.path.join(script_dir, 'data', 'csv_files')
def generate_gaussian_component(time_data, latency, amplitude, width):
"""
Generate a single Gaussian VEP component.
Parameters:
----------
time_data : array
Time points in seconds
latency : float
Component latency in seconds
amplitude : float
Peak amplitude in microvolts
width : float
Standard deviation in seconds
Returns:
--------
component : array
Gaussian component waveform
"""
return amplitude * np.exp(-0.5 * ((time_data - latency) / width) ** 2)
def generate_ideal_vep(time_data):
"""
Generate a clean, ideal VEP waveform using Gaussian components.
Parameters:
----------
time_data : array
Time points in seconds
Returns:
--------
vep : array
Clean VEP waveform in microvolts
"""
vep = np.zeros(len(time_data))
# N75 component - negative deflection at 75ms
n75 = generate_gaussian_component(
time_data,
latency=0.075, # 75ms
amplitude=-5.0, # -4 µV
width=0.015 # 15ms width
)
vep += n75
# P100 component - main positive peak at 100ms
p100 = generate_gaussian_component(
time_data,
latency=0.100, # 100ms
amplitude=14.5, # 10 µV (most prominent)
width=0.020 # 20ms width
)
vep += p100
# N135 component - negative deflection at 135ms
n135 = generate_gaussian_component(
time_data,
latency=0.135, # 135ms
```

```
amplitude=-11.0, # -8 µV
width=0.025 # 25ms width
)
vep += n135
return vep
def generate_delayed_p100_vep(time_data):
"""
Generate VEP with delayed P100 component (abnormal).
Parameters:
----------
time_data : array
Time points in seconds
Returns:
--------
vep : array
VEP with delayed P100 in microvolts
"""
vep = np.zeros(len(time_data))
# N75 - normal timing
n75 = generate_gaussian_component(
time_data,
latency=0.075,
amplitude=-4.0,
width=0.015
)
vep += n75
# P100 - DELAYED to 120ms (abnormal)
p100_delayed = generate_gaussian_component(
time_data,
latency=0.120, # 120ms (delayed by 20ms)
amplitude=10.0,
width=0.020
)
vep += p100_delayed
# N135 - shifted later due to P100 delay
n135_shifted = generate_gaussian_component(
time_data,
latency=0.155, # 155ms (also shifted)
amplitude=-8.0,
width=0.025
)
vep += n135_shifted
return vep
def generate_absent_p100_vep(time_data):
"""
Generate VEP with absent P100 component (severe abnormality).
Parameters:
----------
time_data : array
Time points in seconds
Returns:
--------
vep : array
VEP without P100 in microvolts
"""
vep = np.zeros(len(time_data))
# N75 - present but reduced
n75 = generate_gaussian_component(
time_data,
latency=0.075,
amplitude=-3.0, # Reduced amplitude
width=0.020 # Slightly wider
)
vep += n75
# P100 - ABSENT (no component added)
# N135 - present
n135 = generate_gaussian_component(
time_data,
latency=0.135,
amplitude=-7.0, # Slightly reduced
width=0.030 # Wider
)
vep += n135
return vep
def generate_reduced_amplitude_vep(time_data):
"""
Generate VEP with reduced overall amplitude.
Parameters:
----------
time_data : array
Time points in seconds
Returns:
--------
vep : array
Low amplitude VEP in microvolts
"""
# Generate normal VEP and reduce amplitude
vep = generate_ideal_vep(time_data)
return vep * 0.5 # 50% reduced amplitude
def generate_noisy_vep(time_data, noise_level=2.0):
```

```
"""
Generate ideal VEP with added noise for testing filtering.
Parameters:
----------
time_data : array
Time points in seconds
noise_level : float
Standard deviation of noise in microvolts
Returns:
--------
vep : array
VEP with noise in microvolts
"""
# Start with ideal VEP
vep = generate_ideal_vep(time_data)
# Add Gaussian noise
noise = np.random.normal(0, noise_level, len(time_data))
return vep + noise
def save_csv(filename, time_ms, amplitude_uv):
"""
Save VEP data to CSV file with headers.
Parameters:
----------
filename : str
Output filename
time_ms : array
Time points in milliseconds
amplitude_uv : array
Amplitude in microvolts
"""
with open(filename, 'w') as f:
# Write header
f.write("Time_ms,Amplitude_uV\n")
# Write data
for t, a in zip(time_ms, amplitude_uv):
f.write(f"{t:.3f},{a:.6f}\n")
print(f"  Saved: {os.path.basename(filename)}")
def create_test_files(output_dir):
"""
Create a set of test VEP CSV files.
Parameters:
----------
output_dir : str
Directory to save files
"""
# Setup time data
duration = 0.5 # 500ms
sampling_rate = 1000 # 1000 Hz
time_data = np.arange(0, duration, 1/sampling_rate)
time_ms = time_data * 1000
files_created = []
print("\nGenerating test VEP files:")
print("-" * 40)
# 1. Ideal VEP
print("1. Ideal VEP (clean reference)")
vep = generate_ideal_vep(time_data)
filename = os.path.join(output_dir, "ideal_vep.csv")
save_csv(filename, time_ms, vep)
files_created.append(filename)
# 2. Delayed P100
print("2. Delayed P100 (abnormal)")
vep = generate_delayed_p100_vep(time_data)
filename = os.path.join(output_dir,
"delayed_p100_vep.csv")
save_csv(filename, time_ms, vep)
files_created.append(filename)
# 3. Absent P100
print("3. Absent P100 (severe abnormality)")
vep = generate_absent_p100_vep(time_data)
filename = os.path.join(output_dir, "absent_p100_vep.csv")
save_csv(filename, time_ms, vep)
files_created.append(filename)
# 4. Reduced amplitude
print("4. Reduced amplitude VEP")
vep = generate_reduced_amplitude_vep(time_data)
filename = os.path.join(output_dir,
"reduced_amplitude_vep.csv")
save_csv(filename, time_ms, vep)
files_created.append(filename)
# 5. VEP with noise
print("5. VEP with noise (SNR testing)")
vep = generate_noisy_vep(time_data, noise_level=3.0)
filename = os.path.join(output_dir, "noisy_vep.csv")
save_csv(filename, time_ms, vep)
files_created.append(filename)
return files_created
def create_single_file(output_dir, file_type):
"""
Create a single test file of specified type.
Parameters:
----------
output_dir : str
```

```
Directory to save file
file_type : str
Type of file to create
"""
# Setup time data
duration = 0.5
sampling_rate = 1000
time_data = np.arange(0, duration, 1/sampling_rate)
time_ms = time_data * 1000
if file_type == "ideal":
vep = generate_ideal_vep(time_data)
filename = os.path.join(output_dir, "ideal_vep.csv")
elif file_type == "delayed":
vep = generate_delayed_p100_vep(time_data)
filename = os.path.join(output_dir, "delayed_p100_vep.csv")
elif file_type == "absent":
vep = generate_absent_p100_vep(time_data)
filename = os.path.join(output_dir, "absent_p100_vep.csv")
elif file_type == "reduced":
vep = generate_reduced_amplitude_vep(time_data)
filename = os.path.join(output_dir, "reduced_amplitude_vep.csv")
elif file_type == "noisy":
vep = generate_noisy_vep(time_data, noise_level=3.0)
filename = os.path.join(output_dir, "noisy_vep.csv")
else:
print("Unknown file type")
return None
save_csv(filename, time_ms, vep)
return filename
def main():
"""Main menu for VEP test file generation."""
print("=" * 50)
print("Simple VEP Test CSV Generator")
print("=" * 50)
print("Creates clean test VEP waveforms using Gaussian components")
# Create output directory
output_dir = create_data_folders()
while True:
print("\n" + "=" * 50)
print("Menu Options:")
print("-" * 50)
print("1. Generate all test files")
print("2. Generate ideal VEP only")
print("3. Generate delayed P100 VEP")
print("4. Generate absent P100 VEP")
print("5. Generate reduced amplitude VEP")
print("6. Generate noisy VEP")
print("0. Exit")
print("=" * 50)
choice = input("\nSelect option (0-6): ").strip()
if choice == "0":
print("\nExiting...")
break
elif choice == "1":
files = create_test_files(output_dir)
print(f"\n Created {len(files)} test files")
elif choice == "2":
filename = create_single_file(output_dir, "ideal")
print(f"\n Created ideal VEP file")
elif choice == "3":
filename = create_single_file(output_dir, "delayed")
print(f"\n Created delayed P100 VEP file")
elif choice == "4":
filename = create_single_file(output_dir, "absent")
print(f"\n Created absent P100 VEP file")
elif choice == "5":
filename = create_single_file(output_dir, "reduced")
print(f"\n Created reduced amplitude VEP file")
elif choice == "6":
filename = create_single_file(output_dir, "noisy")
print(f"\n Created noisy VEP file")
else:
print("Invalid choice. Please select 0-6.")
input("\nPress Enter to continue...")
if __name__ == "__main__":
main()
```

Appendix 3

```
Basic Visual Evoked Potential (VEP) Waveform Generator
Using MNE-Python for neurophysiological signal synthesis
This implementation creates synthetic VEP waveforms based on clinical standards
with proper N75, P100, and N135 components.
"""
import numpy as np
import matplotlib.pyplot as plt
import mne #https://mne.tools/stable/index.html
from scipy import signal, stats
import warnings
warnings.filterwarnings('ignore')
```

```
class VEPGenerator:
"""
Generates synthetic Visual Evoked Potential waveforms
based on clinical specifications and ISCEV standards.
"""
def __init__(self, sampling_rate=1000, duration=0.5):
"""
Initialize VEP generator with basic parameters.
Parameters:
----------
sampling_rate : int
Sampling frequency in Hz (recommended: 500-1000 Hz)
duration : float
Total duration of VEP epoch in seconds
"""
self.sampling_rate = sampling_rate
self.duration = duration
self.times = np.arange(0, duration, 1/sampling_rate)
# Standard VEP component parameters (in seconds and microvolts)
# Based on research findings
'''
Stimulus → [0ms]
↓
Retinal response → [~20-30ms] (not seen in VEP)
↓
Optic nerve conduction → [~40-60ms]
↓
N75: Initial cortical arrival → [75ms] ← "Signal received"
↓
P100: Main cortical processing → [100ms] ← "Pattern detected!"
↓
N135: Higher-level processing → [135ms] ← "Analyzing meaning"
'''
self.components = {
'N75': {
'latency': 0.075, # 75 ms
'amplitude': -4.0, # -4 µV
'width': 0.015, # 15 ms width
'latency_range': (0.067, 0.110),
'amplitude_range': (-6, -2)
},
'P100': {
'latency': 0.100, # 100 ms
'amplitude': 10.0, # 10 µV (most prominent)
'width': 0.020, # 20 ms width
'latency_range': (0.090, 0.120),
'amplitude_range': (5, 20)
},
'N135': {
'latency': 0.135, # 135 ms
'amplitude': -8.0, # -8 µV
'width': 0.025, # 25 ms width
'latency_range': (0.121, 0.246),
'amplitude_range': (-10, -6)
}
}
self.clinical_norms = {
'N75': {
'latency_mean': 75, # ms
'latency_sd': 3.5, # ms
'latency_range': (65, 80), # Clinical limits
'amplitude_mean': -4.0, # µV
'amplitude_sd': 1.0, # µV
'amplitude_range': (-6, -2),
'rise_time': 12, # ms
'fall_time': 18, # ms
},
'P100': {
'latency_mean': 100, # ms - most important clinically
'latency_sd': 5.0, # ms
'latency_range': (95, 115), # >115ms is abnormal
'amplitude_mean': 10.0, # µV
'amplitude_sd': 3.0, # µV
'amplitude_range': (5, 20),
'rise_time': 15,
'fall_time': 25,
},
'N135': {
'latency_mean': 135,
'latency_sd': 6.0,
'latency_range': (120, 145),
'amplitude_mean': -7.0,
'amplitude_sd': 2.0,
'amplitude_range': (-12, -4),
'rise_time': 20,
'fall_time': 30,
}
}
```

```
def sample_clinical_parameters(self, component_name, add_variability=True):
"""
Sample parameters from clinical distributions.
Parameters:
----------
component_name : str
Name of VEP component ('N75', 'P100', 'N135')
add_variability : bool
Whether to add statistical variability
Returns:
--------
params : dict
Sampled parameters for the component
"""
norms = self.clinical_norms[component_name]
if add_variability:
# Sample from normal distributions
latency_ms = np.random.normal(norms['latency_mean'], norms['latency_sd'])
amplitude = np.random.normal(norms['amplitude_mean'], norms['amplitude_sd'])
# Add variability to rise/fall times (±20%)
rise_time_ms = norms['rise_time'] * np.random.uniform(0.8, 1.2)
fall_time_ms = norms['fall_time'] * np.random.uniform(0.8, 1.2)
# Clamp to clinical ranges
latency_ms = np.clip(latency_ms, norms['latency_range'][0], norms['latency_range'][1])
amplitude = np.clip(amplitude, norms['amplitude_range'][0], norms['amplitude_range'][1])
else:
# Use mean values for clean/reference waveform
latency_ms = norms['latency_mean']
amplitude = norms['amplitude_mean']
rise_time_ms = norms['rise_time']
fall_time_ms = norms['fall_time']
return {
'latency': latency_ms / 1000, # Convert to seconds
'amplitude': amplitude,
'rise_time': rise_time_ms / 1000, # Convert to seconds
'fall_time': fall_time_ms / 1000 # Convert to seconds
}
def generate_component(self, component_name, add_variability=True,
morphology='asymmetric_gaussian'):
"""
Generate a single VEP component using clinical parameters.
Parameters:
----------
component_name : str
Name of component ('N75', 'P100', 'N135')
add_variability : bool
Add physiological variability
morphology : str
Component morphology model ('gaussian', 'asymmetric_gaussian')
Returns:
--------
component : ndarray
Generated component waveform
"""
# Sample clinical parameters
params = self.sample_clinical_parameters(component_name, add_variability)
if morphology == 'asymmetric_gaussian':
component = self._generate_asymmetric_gaussian_component(params)
else: # Default Gaussian
width = (params['rise_time'] + params['fall_time']) / 4 # Approximate width
component = params['amplitude'] * np.exp(-0.5 * ((self.times - params['latency']) / width) ** 2)
return component
def _generate_asymmetric_gaussian_component(self, params):
"""
Generate asymmetric Gaussian component with realistic rise/fall times.
"""
latency = params['latency']
amplitude = params['amplitude']
rise_time = params['rise_time']
fall_time = params['fall_time']
component = np.zeros(len(self.times))
for i, t in enumerate(self.times):
if t <= latency:
# Rising phase - steeper
if t >= latency - 3*rise_time: # Start 3 rise-times before peak
sigma = rise_time / 2.5 # Steeper rise
```

```
component[i] = amplitude * np.exp(-0.5 * ((t - latency) / sigma) ** 2)
else:
# Falling phase - more gradual
sigma = fall_time / 2.0 # More gradual fall
component[i] = amplitude * np.exp(-0.5 * ((t - latency) / sigma) ** 2)
return component
def generate_baseline_vep(self, add_variability=False, morphology='asymmetric_gaussian'):
"""
Generate a complete VEP waveform using clinical norms.
Parameters:
----------
add_variability : bool
Add realistic inter-trial variability
morphology : str
Component morphology model
Returns:
--------
vep : ndarray
Complete VEP waveform in microvolts
"""
vep = np.zeros(len(self.times))
# Generate each component using clinical norms
for comp_name in ['N75', 'P100', 'N135']:
component = self.generate_component(comp_name, add_variability, morphology)
vep += component
# Optional: Add component interaction effects
if add_variability:
vep = self._apply_component_interactions(vep, comp_name, component)
return vep
def _apply_component_interactions(self, vep, comp_name, component):
"""
Apply realistic interactions between VEP components.
"""
# P100 amplitude affects N135 amplitude (refractory effects)
if comp_name == 'P100':
p100_amplitude = np.max(component)
# Store for N135 adjustment
self._last_p100_amplitude = p100_amplitude
elif comp_name == 'N135' and hasattr(self, '_last_p100_amplitude'):
# Larger P100 -> slightly larger N135 (within limits)
if self._last_p100_amplitude > 12: # Above average P100
enhancement_factor = 1.1
component_idx = np.argmin(component) # Find N135 peak (most negative)
vep[component_idx] *= enhancement_factor
return vep
def generate_ideal_vep(self, morphology='asymmetric_gaussian'):
"""
Generate the 'ideal' reference VEP using mean clinical values.
"""
return self.generate_baseline_vep(add_variability=False, morphology=morphology)
def add_eeg_background(self, vep, snr_db=10):
"""
Add realistic EEG background activity to VEP.
Parameters:
----------
vep : ndarray
Clean VEP waveform
snr_db : float
Signal-to-noise ratio in decibels
Returns:
--------
vep_with_noise : ndarray
VEP with added EEG background
"""
# Generate 1/f pink noise for EEG background
freqs = np.fft.fftfreq(len(self.times), 1/self.sampling_rate)
pink_spectrum = np.where(freqs > 0, 1/np.sqrt(freqs), 0)
# Generate random phase
random_phase = np.random.uniform(0, 2*np.pi, len(freqs))
pink_fft = pink_spectrum * np.exp(1j * random_phase)
# Convert to time domain
pink_noise = np.real(np.fft.ifft(pink_fft))
# Scale noise based on desired SNR
signal_power = np.mean(vep ** 2)
noise_power = np.mean(pink_noise ** 2)
snr_linear = 10 ** (snr_db / 10)
noise_scale = np.sqrt(signal_power / (snr_linear * noise_power))
scaled_noise = pink_noise * noise_scale
# Add alpha rhythm (8-12 Hz) common in visual areas
alpha_freq = np.random.uniform(8, 12)
alpha_amplitude = np.random.uniform(2, 5) # 2-5 μV
```

```
alpha = alpha_amplitude * np.sin(2 * np.pi * alpha_freq * self.times)
return vep + scaled_noise + alpha
def generate_trial_batch(self, n_trials=100, add_noise=False, morphology='asymmetric_gaussian'):
"""
Generate multiple VEP trials for averaging simulation.
Parameters:
----------
n_trials : int
Number of trials to generate
add_noise : bool
Add EEG background noise
Returns:
--------
trials : ndarray
Array of VEP trials (n_trials x n_samples)
averaged : ndarray
Averaged VEP across all trials
"""
trials = np.zeros((n_trials, len(self.times)))
for i in range(n_trials):
# Generate VEP with variability
vep = self.generate_baseline_vep(add_variability=True, morphology=morphology)
if add_noise:
# Add background with varying SNR
snr = np.random.uniform(5, 15) # Variable SNR
vep = self.add_eeg_background(vep, snr_db=snr)
trials[i, :] = vep
# Calculate averaged VEP (standard clinical practice)
averaged = np.mean(trials, axis=0)
return trials, averaged
def apply_clinical_filters(self, signal_data):
"""
Apply standard clinical filtering to VEP signal.
Parameters:
----------
signal_data : ndarray
Raw VEP signal
Returns:
--------
filtered : ndarray
Filtered VEP signal
"""
# Design filters based on ISCEV standards
# High-pass: 1 Hz
# Low-pass: 100 Hz
nyquist = self.sampling_rate / 2
# Butterworth filters (standard in clinical practice)
b_high, a_high = signal.butter(2, 1/nyquist, btype='high')
b_low, a_low = signal.butter(4, 100/nyquist, btype='low')
# Apply filters
filtered = signal.filtfilt(b_high, a_high, signal_data)
filtered = signal.filtfilt(b_low, a_low, filtered)
return filtered
def detect_components(self, vep_data):
"""
Detect actual component peaks/troughs in VEP waveform.
Returns:
--------
components : dict
Detected latencies and amplitudes for each component
"""
time_ms = self.times * 1000
components = {}
# N75: Find minimum in 60-85ms window
n75_window = (time_ms >= 60) &amp;amp; (time_ms <= 85)
if np.any(n75_window):
n75_idx = np.where(n75_window)[0][np.argmin(vep_data[n75_window])]
components['N75'] = {
'latency': time_ms[n75_idx],
'amplitude': vep_data[n75_idx],
'index': n75_idx
}
# P100: Find maximum in 85-120ms window
p100_window = (time_ms >= 85) &amp;amp; (time_ms <= 120)
if np.any(p100_window):
p100_idx = np.where(p100_window)[0][np.argmax(vep_data[p100_window])]
components['P100'] = {
'latency': time_ms[p100_idx],
'amplitude': vep_data[p100_idx],
'index': p100_idx
}
# N135: Find minimum in 120-150ms window
n135_window = (time_ms >= 120) &amp;amp; (time_ms <= 150)
if np.any(n135_window):
n135_idx = np.where(n135_window)[0][np.argmin(vep_data[n135_window])]
```

```
components['N135'] = {
'latency': time_ms[n135_idx],
'amplitude': vep_data[n135_idx],
'index': n135_idx
}
return components
def plot_vep(self, vep_data, title='VEP Waveform', show_clinical_ranges=True,
mark_actual_components=True):
"""
Create clinical-style VEP plot with clinical range highlighting.
Parameters:
----------
vep_data : ndarray
VEP signal to plot
title : str
Plot title
show_clinical_ranges : bool
Whether to show clinical normal/abnormal ranges
mark_actual_components : bool
True - detect and mark actual peaks/troughs
False - mark at expected times (75, 100, 135ms)
"""
fig, ax = plt.subplots(1, 1, figsize=(12, 6))
# Time domain plot
time_ms = self.times * 1000 # Convert to milliseconds
# Add clinical range shading BEFORE plotting the waveform
if show_clinical_ranges:
# P100 normal range (95-115ms) - most clinically important
ax.axvspan(95, 115, alpha=0.2, color='green',
label='P100 Normal Range (95-115ms)')
# N75 normal range (65-80ms)
ax.axvspan(65, 80, alpha=0.15, color='blue',
label='N75 Normal Range (65-80ms)')
# N135 normal range (120-145ms)
ax.axvspan(120, 145, alpha=0.15, color='orange',
label='N135 Normal Range (120-145ms)')
# Plot the waveform on top
ax.plot(time_ms, vep_data, 'b-', linewidth=2, zorder=5)
ax.axhline(y=0, color='k', linestyle='-', alpha=0.3)
# Component markers
if mark_actual_components:
# Detect and mark actual components
detected = self.detect_components(vep_data)
if 'N75' in detected:
ax.axvline(x=detected['N75']['latency'], color='darkblue',
linestyle='--', alpha=0.7, linewidth=1.5, label='N75')
ax.text(detected['N75']['latency']-5, ax.get_ylim()[1]*0.9,
f"N75\n{detected['N75']['latency']:.1f}ms",
ha='center', fontsize=8, color='darkblue')
if 'P100' in detected:
ax.axvline(x=detected['P100']['latency'], color='darkgreen',
linestyle='--', alpha=0.7, linewidth=2, label='P100')
ax.text(detected['P100']['latency']+5, ax.get_ylim()[1]*0.9,
f"P100\n{detected['P100']['latency']:.1f}ms",
ha='center', fontsize=8, color='darkgreen')
if 'N135' in detected:
ax.axvline(x=detected['N135']['latency'],
color='darkorange',
linestyle='--', alpha=0.7, linewidth=1.5, label='N135')
ax.text(detected['N135']['latency']-5, ax.get_ylim()[1]*0.9,
f"N135\n{detected['N135']['latency']:.1f}ms",
ha='center', fontsize=8, color='darkorange')
else:
# Mark at expected times
ax.axvline(x=75, color='darkblue', linestyle='--', alpha=0.5,
label='N75')
ax.text(80, ax.get_ylim()[1]*0.95, 'N75',
ha='center', fontsize=8, color='darkblue')
ax.axvline(x=100, color='darkgreen', linestyle='--',
alpha=0.7, label='P100')
ax.text(105, ax.get_ylim()[1]*0.95, 'P100',
ha='center', fontsize=8, color='darkgreen')
ax.axvline(x=135, color='darkorange', linestyle='--',
alpha=0.5, label='N135')
ax.text(140, ax.get_ylim()[1]*0.95, 'N135',
ha='center', fontsize=8, color='darkorange')
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Amplitude (μV)')
ax.set_title(title)
ax.grid(True, alpha=0.3)
ax.legend(loc='upper right', fontsize=9)
ax.set_xlim([0, 250])
plt.tight_layout()
plt.show()
def plot_trials_comparison(self, trials, averaged, morphology, show_clinical_ranges=True):
"""Plot individual trials with averaged response and clinical ranges."""
fig, axes = plt.subplots(1, 1, figsize=(12, 8))
```

```
time_ms = self.times * 1000
# Add clinical range shading first
if show_clinical_ranges:
# P100 normal range
axes.axvspan(95, 115, alpha=0.2, color='green',
label='P100 Normal Range (95-115ms)', zorder=1)
# N75 normal range
axes.axvspan(65, 80, alpha=0.1, color='blue',
label='N75 Normal Range (65-80ms)', zorder=1)
# N135 normal range
axes.axvspan(120, 145, alpha=0.1, color='orange',
label='N135 Normal Range (120-145ms)', zorder=1)
# Plot x number of trials
x = 25
for i in range(x):
axes.plot(time_ms, trials[i, :], alpha=0.3, color='gray',
linewidth=0.8, zorder=2)
# Plot averaged response on top
axes.plot(time_ms, averaged, 'b-', linewidth=3,
label=f'Averaged (n={len(trials)})', zorder=3)
# Add component markers
axes.axvline(x=75, color='darkblue', linestyle=':',
alpha=0.8, label='N75')
axes.axvline(x=100, color='darkgreen', linestyle=':',
alpha=0.8,
linewidth=2, label='P100')
axes.axvline(x=135, color='darkorange', linestyle=':',
alpha=0.8, label='N135')
axes.set_title(f'Individual Trials ({x} shown) with Clinical
Ranges - {morphology.replace("_", "
").title()}')
axes.set_xlabel('Time (ms)')
axes.set_ylabel('Amplitude (μV)')
axes.legend(loc='upper right', fontsize=9)
axes.grid(True, alpha=0.5)
axes.set_xlim([0, 250])
plt.tight_layout()
plt.show()
# Example usage and demonstration
def demonstrate_vep_generator():
"""
Demonstrate the VEP generator with various examples.
"""
print("=" * 60)
print("VEP Waveform Generator Demonstration")
print("=" * 60)
# Initialize generator
generator = VEPGenerator(sampling_rate=1000,
duration=0.5)
time_ms = generator.times * 1000 # Time in milliseconds
# 1. Generate single clean VEP
print("\n1. Generating single clean VEP...")
clean_vep =
generator.generate_baseline_vep(add_variability=False,
morphology='asymmetric_gaussian')
# clean_vep = generator.generate_reference_vep()
generator.plot_vep(clean_vep, title='Clean VEP Waveform
(No Variability) - Asymmetric Gaussian')
# 2. Generate VEP with EEG background
print("\n2. Adding EEG background noise...")
noisy_vep = generator.add_eeg_background(clean_vep,
snr_db=10)
generator.plot_vep(noisy_vep, title='VEP with EEG
Background (SNR=10dB)')
# 3. Apply clinical filtering
print("\n3. Applying clinical filters...")
filtered_vep = generator.apply_clinical_filters(noisy_vep)
# Plot filtered vs unfiltered
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(time_ms, noisy_vep, 'b-', alpha=0.5,
label='Unfiltered')
ax.plot(time_ms, filtered_vep, 'r-', linewidth=2,
label='Filtered (1-100 Hz)')
ax.set_xlabel('Time (ms)')
ax.set_ylabel('Amplitude (μV)')
ax.set_title('Effect of Clinical Filtering')
ax.legend()
ax.grid(True, alpha=0.3)
plt.show()
# 4. Generate and average multiple trials
print("\n4. Generating 100 trials and averaging...")
trials, averaged_vep =
generator.generate_trial_batch(n_trials=100,
add_noise=True)
generator.plot_trials_comparison(trials, averaged_vep,
morphology='asymmetric_gaussian')
return generator
# Run demonstration if script is executed directly
if __name__ == "__main__":
generator, raw = demonstrate_vep_generator()
print("\n VEP Generator demonstration complete!")
print("Generator object and MNE Raw object are available
for further use.")
```