APPLIED EVENT-RELATED POTENTIAL DATA ANALYSIS



Steven J Luck
University of California, Davis



Applied Event-Related Potential Data Analysis

Steven J Luck

University of California, Davis

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (https://LibreTexts.org) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden borne from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by NICE CXone Expert and were supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptions contact info@LibreTexts.org or visit our main website at https://LibreTexts.org.

The LibreTexts name, logo, and book covers are trademarked by LibreTexts, Inc. (a 501(c)(3) not-for-profit organization) and protected against unauthorized use.

This text was compiled on 09/04/2026

TABLE OF CONTENTS

Hardware and Software Requirements

Licensing

Preface

Acknowledgments

How to Cite This Book

1: First Steps

1.1: Getting Started

1.2: Installing the Software and Downloading the Data

1.3: Exercise- Loading an EEG File

1.4: Exercise - Viewing Continuous EEG Waveforms

1.5: Exercise - Viewing EEG Spectra

1.6: Exercise- Loading ERPs and Plotting ERP Waveforms

1.7: Exercise- Plotting ERP Scalp Maps

1.8: Finding the Right Routine in EEGLAB and ERPLAB

1.9: Understanding the Matlab Path

1.10: Key Takeaways and References

2: Processing the Data from One Participant in the ERP CORE N400 Experiment

2.1: Data For This Chapter

2.2: Design of the N400 Experiment

2.3: Exercise - Looking at the EEG and the Event Codes

2.4: Exercise- Filtering Out Low-Frequency Drifts from the EEG

2.5: Exercise- Creating an EventList

2.6: Exercise- Assigning Events to Bins with BINLISTER

2.7: Exercise- Epoching and Baseline Correction

2.8: Exercise- Artifact Detection

2.9: Exercise- Averaged ERPs

2.10: Exercise- Data Quality

2.11: Review of Processing Steps

2.12: A Simple Matlab Script

2.13: Key Takeaways and References

3: Processing Multiple Participants in the ERP CORE N400 Experiment

3.1: Data for This Chapter

3.2: Exercise- Preprocessing and Averaging the Data from 10 Participants

3.3: Exercise- Examining the Single-Participant ERPsets

3.4: Exercise- “Bad” Data

3.5: Exercise- Making a Grand Average

3.6: Exercise- Low-Pass Filtering
3.7: Exercise - Scoring N400 Amplitude
3.8: Exercise- Simple Statistical Analysis of N400 Data
3.9: Exercise- A More Complex Analysis
3.10: Exercise- ERP Channel Operations
3.11: Exercise- ERP Bin Operations
3.12: Review of Processing Steps
3.13: Matlab Scripts For This Chapter
3.14: Key Takeaways and References

4: Filtering the EEG and ERPs

4.1: Data for this Chapter
4.2: Classes of Filters
4.3: Exercise- Assessing the Frequency Content of the Noise
4.4: Exercise- Filtering the Artificial Waveforms
4.5: Exercise- The Impulse Response Function
4.6: Exercise- Applying the Impulse Response Function to a Series of Impulses
4.7: Background- Filtering with a Running Average
4.8: Background- Filtering with a Weighted Running Average
4.9: Exercise- Distortion of Onset and Offset Times by Low-Pass Filters
4.10: Exercise- High-Pass Filtering
4.11: Practical Advice
4.12: Exercise- Creating and Importing Artificial Waveforms
4.13: Matlab Script for this Chapter
4.14: Key Takeaways and References

5: Referencing and Other Channel Operations

5.1: Data for This Chapter
5.2: Background- Understanding Active, Reference, and Ground Electrodes
5.3: Exercise- Working with the Artificial Data
5.4: Exercise- Average Mastoids as the Reference
5.5: Exercise- Re-Referencing the N400 ERP CORE Data
5.6: Exercise- The Average Reference
5.7: What is the Best Reference Site?
5.8: Exercise- Current Density
5.9: Exercise- Global Field Power
5.10: Exercise- Referencing the EEG Data from the ERP CORE N400 Experiment
5.11: Exercise- Other Common Re-Referencing Scenarios
5.12: Matlab Script For This Chapter
5.13: Key Takeaways and References

6: Assigning Events to Bins, Averaging, Baseline Correction, and Assessing Data Quality

6.1: Data for This Chapter

6.2: Design of the ERP CORE Visual Oddball P3b Experiment
6.3: The Event Code Scheme
6.4: Overview of Bin Descriptor Files
6.5: Exercise - A Basic Assignment of Events to Bins
6.6: Exercise - Looking at the Averaged ERPs
6.7: Exercise - The Signal-to-Noise Ratio
6.8: Exercise - Response-Locked Averaging
6.9: Exercise - Comparing Correct and Error Trials
6.10: Exercise - Sequential Analysis of the P3b
6.11: Exercise - Combining Bins
6.12: Exercise - Overlap
6.13: Matlab Script For This Chapter
6.14: Key Takeaways and References

7: Inspecting the EEG and Interpolating Bad Channels

7.1: Data for This Chapter
7.2: Design of the Mismatch Negativity (MMN) Experiment
7.3: Video Demonstration- Performing an Initial Inspection of a Participant's EEG
7.4: The Fundamental Goal of EEG Preprocessing
7.5: Background- Interpolating Bad Channels
7.6: Exercise - Interpolating Bad Channels
7.7: Matlab Script For This Chapter
7.8: Key Takeaways and References

8: Artifact Detection and Rejection

8.1: Data for This Chapter
8.2: Overview
8.3: Background- Why Do We Reject Artifacts?
8.4: Background- The General Approach
8.5: Exercise- Simple Blink Detection
8.6: Exercise- Adjusting the Threshold
8.7: An Iterative Approach to Setting Parameters
8.8: Exercise- Data Quality and Confounds
8.9: Exercise- Better Blink Detection
8.10: Exercise- Detecting Eye Movements
8.11: Exercise- Deciding on a Threshold for Eye Movements
8.12: Exercise- Commonly Recorded Artifactual Potentials (C.R.A.P.)
8.13: Using Artifact Detection to Avoid Changes to Visual Inputs
8.14: The ERP CORE N2pc Experiment
8.15: Exercise- Visualizing the Eye Movements
8.16: Exercise- Using the Averaged HEOG to Visualize Consistent Eye Movements
8.17: Exercise- A Two-Stage Strategy for Eliminating Small But Consistent Eye Movements

8.18: Matlab Script For This Chapter
8.19: Key Takeaways and References

9: Artifact Correction with Independent Component Analysis

9.1: Data for this Chapter
9.2: Exercise- A First Pass at ICA-Based Blink Correction
9.3: Exercise- Evaluating the Impact of Artifact Correction
9.4: Background- A Quick Conceptual Overview of ICA
9.5: Exercise- Making ICA Work Better
9.6: Exercise- Transferring the Weights and Assessing the ICs
9.7: Exercise- Deciding Which ICs to Exclude
9.8: Exercise- Deleting C.R.A.P. Prior to ICA
9.9: General Recommendations
9.10: Matlab Scripts For This Chapter
9.11: Key Takeaways and References

10: Scoring and Statistical Analysis of ERP Amplitudes and Latencies

10.1: Data for This Chapter
10.2: Design of the Flankers Experiment
10.3: Exercise- Examining the Grand Averages
10.4: Exercise- A First Pass at Scoring and Statistical Analysis
10.5: Exercise- Simplifying the Statistical Analysis
10.6: Exercise- Peak Amplitude
10.7: Exercise- Peak Latency
10.8: Exercise- Fractional Area Latency
10.9: Exercise- Quantifying Onset Latency
10.10: Exercise- Collapsing Across Channels and Correlating Latencies with Response Times
10.11: Matlab Scripts For This Chapter
10.12: Key Takeaways and References

11: EEGLAB and ERPLAB Scripting

11.1: Data for This Chapter
11.2: Expected Background Knowledge
11.3: Bugs as an Opportunity for Growth
11.4: Design of the N170 Experiment
11.5: Exercise- The Matlab Command Line and the EEG Variable
11.6: Exercise- The ALLEEG Variable and Redrawing the GUI
11.7: Exercise- EEG.history and eegh
11.8: Exercise- From the Command Line to a Script
11.9: Exercise- Using a Variable for the Path
11.10: Exercise- Loops
11.11: Exercise- Looping Through Data from Multiple Participants
11.12: Rapid Cycling Between Coding and Testing

11.13: Exercise- Referencing with a Script
11.14: Exercise- Improving the Referencing Script
11.15: Exercise- Preprocessing the EEG and Using a Spreadsheet to Store Subject-Specific Information
11.16: Exercise- Building an Entire EEG Processing Pipeline
11.17: Exercise- Averaging with a Custom aSME Time Window
11.18: Exercise- Scoring Amplitudes and Latencies and Performing Statistical Analyses
11.19: Key Takeaways and References

12: Appendix 1: A Very Brief Introduction to EEG and ERPs

13: Appendix 2: Troubleshooting Guide

13.1: A2.1 The First Step
13.2: A2.2 Some Basic Solutions
13.3: A2.3 Taking a Scientific Approach
13.4: A2.4 Deciphering Matlab's Error Messages
13.5: A2.5 Debugging Scripts by Performing Experiments and Collecting Data
13.6: A2.6 Avoiding Bugs in Your Scripts with Good Programming Practices
13.7: A2.7 References

14: Appendix 3: Example Processing Pipeline

Index

Glossary

Detailed Licensing

This page titled Applied Event-Related Potential Data Analysis (Luck) is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

Hardware and Software Requirements

The core of this book is a large set of data processing and analysis exercises. They are designed to run on your own computer using two free, open source Matlab toolboxes, EEGLAB and ERPLAB. However, you will need a Matlab license to run these toolboxes. Check with your institution's Information Technology office to see if they provide free or reduced-cost Matlab licenses. It may also be possible for you to purchase a student license. In particular, you will need:

- Matlab version 2017a or higher
    - The Signal Processing Toolbox is required. If you don't have it, contact your institution's IT support department for assistance.
    - The Statistics and Machine Learning Toolbox is recommended.
    - To see what toolboxes are already installed in Matlab, type **ver** on the Matlab command line.
    - Some parts of the software will run under Octave, a free Matlab simulator. However, Octave will not run the graphical user interface, which is necessary for most of the exercises. You should therefore use Matlab rather than Octave.
- EEGLAB version 2022.0 or higher (see Chapter 1 for installation instructions)
- ERPLAB version 9.0 or higher (see Chapter 1 for installation instructions)

EEGLAB and ERPLAB will run on any reasonably recent version of Mac OS, Windows, or Linux. These packages are designed for professional users, and they work best on a computer with a lot of RAM, a large hard drive, and a high-resolution external monitor. However, they will function adequately on a midrange laptop. Absolute minimum specifications are:

- 4 MB RAM (but 8 MB will work much better)
- A screen height of at least 720 pixels (but 1080p or higher will work much better)
- Read and write permission in the folder used to store the EEGLAB and ERPLAB code

This page titled Hardware and Software Requirements is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

Licensing

A detailed breakdown of this resource's licensing can be found in ***Back Matter/Detailed Licensing***.

This page titled Licensing is shared under a license and was authored, remixed, and/or curated by LibreTexts.

Preface

When I was a new assistant professor, journal editors kept sending me ERP papers to review that were just awful. I think the editors didn't want to waste the time of more established researchers with these terrible manuscripts. The papers were generally written by researchers who were relatively new to the technique and had not "grown up" in an ERP lab. Because they had not received much ERP training, they made all kinds of errors in the design, the analysis, and the interpretation of the experiments.

That experience made me realize how fortunate I had been to be trained in Steve Hillyard's lab, which traced its roots to the very first published ERP experiment (Davis, 1939). I decided to write a book that distilled everything I had learned in the Hillyard lab so that everyone could benefit from this knowledge (Luck, 2005). I also started running in-person training workshops called ERP Boot Camps along with Emily Kappenman. Emily and I also put together an edited volume on ERP components (Luck & Kappenman, 2012). To make it easier for researchers to analyze their data using the methods we were promoting, I worked with Javier Lopez-Calderon to release ERPLAB Toolbox, a free ERP data processing package (Lopez-Calderon & Luck, 2014). And then I wrote a second edition of my ERP book (Luck, 2014), taking advantage of everything I had learned from the ERP Boot Camp about how to explain ERP methodology.

Although these efforts have reached thousands of researchers, I always felt that something was missing. The books and workshops provided a mix of theory and practical advice, but they were necessarily quite broad given the wide range of researchers who use ERPs. It can be very difficult to take these broad ideas and apply them to the analysis of actual experiments. So, what was missing from our books and our workshops was the opportunity to solve all the problems that arise when you're analyzing real data. We tried having data analysis tutorials in a few of the early ERP Boot Camps, but we found that we spent 98% of our time teaching the participants how to run the software. Also, we ended up teaching them how to analyze data from just a single experiment, which was often quite different from their own research interests.

The present book is designed to fill this gap. It includes tons of example data sets and exercises, all of which run on the free ERPLAB Toolbox package. These examples use data from the ERP CORE (Kappenman et al., 2021), which has data from six classic ERP paradigms. As a result, the examples in this book cover a broad range of paradigms and components with real data.

I've published this book on the free LibreTexts platform so that anyone in the world can learn about ERP data analysis for free. I was extremely fortunate to have great opportunities to learn about ERPs as a student, and I'm trying to "pay it forward" with this book. If you teach the ERP technique to other people (either in courses or in your lab), you should feel free to remix and reuse the book in any way that you find helpful.

Funding

Preparation of this book was made possible by grant R01MH087450 from the National Institute of Mental Health.

References

Davis, P. A. (1939). Effects of acoustic stimuli on the waking human brain. *Journal of Neurophysiology*, *2*, 494–499.

Kappenman, E. S., Farrens, J. L., Zhang, W., Stewart, A. X., & Luck, S. J. (2021). ERP CORE: An Open Resource for Human Event-Related Potential Research. *NeuroImage*, *225*, 117465. https://doi.org/10.1016/j.neuroimage.2020.117465

Lopez-Calderon, J., & Luck, S. J. (2014). ERPLAB: An open-source toolbox for the analysis of event-related potentials. *Frontiers in Human Neuroscience*, *8*, 213. https://doi.org/10.3389/fnhum.2014.00213

Luck, S. J. (2005). *An Introduction to the Event-Related Potential Technique*. MIT Press.

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Luck, S. J., & Kappenman, E. S. (2012). *The Oxford Handbook of Event-Related Potential Components*. Oxford University Press.

This page titled Preface is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

Acknowledgments

This book simply would not have been possible if it had not been for the hard work and creativity of the wonderful people who have worked in my lab over the years.

I'd like to start with a shout-out to Emily Kappenman. Emily and I have worked together for over 15 years on the ERP Boot Camp and many other efforts designed to promote rigorous and high-impact ERP research. Emily spearheaded the ERP CORE, which is one of the two main pillars of this book. Jaclyn Farrens also put years of work into the CORE, and the CORE was pushed across the finish line with help from Wendy Zhang and Andrew Stewart.

The other pillar of this book is ERPLAB Toolbox. The first several releases of ERPLAB were written entirely by Javier Lopez-Calderon, and his fingerprints can still be seen throughout the software. After Javier moved on, Andrew Stewart took over programming duties, and he wrote the initial code for the data quality metrics. After Andrew's departure, my lab manager Aaron Simmons filled in for a while, and it's amazing how much he added while simultaneously running my lab. Aaron made a ton of changes that will make your life easier as you go through the exercises in this book and then apply ERPLAB to your own data. Guanghui Zhang has recently become the primary ERPLAB developer, and I can't wait for the world to see the new version that he's working on now.

I'd also like to thank the other members of my lab who provided feedback on drafts of the book, including (alphabetically) Brett Bahle, Carlos Carrasco, John Kiat, Lara Krisst, Orestis Papaioannou, and Kurt Winsler. I received additional feedback from several members of the UCD EEG Research Group.

In addition, I'd like to acknowledge the National Institute of Mental Health, which has funded ERPLAB for many years (grant R01MH087450) and made this book possible.

Finally, I'd like to thank Lisa and Ruthie, who did their best to keep me sane during the writing process.

This page titled Acknowledgments is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

How to Cite This Book

Cite as: Luck, S. J. (2022). *Applied Event-Related Potential Data Analysis*. LibreTexts. https://doi.org/10.18115/D5QG92

This page titled How to Cite This Book is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

- Acknowledgments by Steven J Luck is licensed CC BY 4.0.

1: FIRST STEPS

Learning Objectives

In this chapter, you will learn to:

- Install the EEGLAB and ERPLAB and download the data for the exercises
- Load EEG and ERP data
- Plot EEG waveforms and spectra
- Plot ERP waveforms and scalp maps
- Update the Matlab PATH when you install or update EEGLAB/ERPLAB

This chapter is designed to provide key background knowledge you'll need for the rest of the book. We'll begin by describing the goals of this book and the background knowledge you should have. Then we'll show you how to install the software and download some data, followed by a few exercises so that you can learn the basics of the EEGLAB and ERPLAB software packages that will be used throughout the book.

1.1: Getting Started

1.2: Installing the Software and Downloading the Data

1.3: Exercise- Loading an EEG File

1.4: Exercise - Viewing Continuous EEG Waveforms

1.5: Exercise - Viewing EEG Spectra

1.6: Exercise- Loading ERPs and Plotting ERP Waveforms

1.7: Exercise- Plotting ERP Scalp Maps

1.8: Finding the Right Routine in EEGLAB and ERPLAB

1.9: Understanding the Matlab Path

1.10: Key Takeaways and References

This page titled 1: First Steps is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

1.1: Getting Started

This section of Chapter 1 provides some important background information that you need to know to make effective use of this book. The whole book will make much more sense if you read this section first!

The Goal of This Book

The primary goal of this book is to help students and researchers learn how to process and analyze event-related potentials (ERPs). My other ERP book (Luck, 2014) focuses on providing a conceptual understanding of ERPs, and the present book focuses on applying those concepts to real data. Theory is important, but there is no substitute for loading up real data—with all its warts and complexities—and figuring out how to go from a gigabyte of raw EEG files to a set of statistical analyses and figures that are ready for publication.

At its essence, this book is a set of data processing and analysis exercises that are wrapped in explanatory text. ERP analysis involves a million decisions, such as whether to filter before versus after artifact rejection and what measurement window to use for quantifying the amplitude of an ERP component. The exercises in this book are designed to give you experience making choices that will lead to the most robust and valid conclusions. In theory, you could read the book without doing the exercises, but that would be like trying to learn painting from a textbook without ever picking up a paintbrush. So fire up your computer and get ready to process some data!

I waited until now to write this book because I needed two things: 1) Free software that anyone can use to do the exercises, and 2) a large public dataset with multiple different ERP paradigms. Both are now available. The free software consists of ERPLAB Toolbox and its companion EEGLAB Toolbox, and the large public dataset is the ERP CORE (Compendium of Open Resources and Experiments).

ERPLAB and EEGLAB

ERPLAB (Lopez-Calderon & Luck, 2014) is a Matlab toolbox that my lab produces with a grant from the National Institutes of Health (NIH). ERPLAB works in tandem with another NIH-supported Matlab toolbox called EEGLAB, which is developed under the leadership of Arno Delorme and Scott Makeig at UCSD (Delorme & Makeig, 2004). EEGLAB takes care of several important EEG preprocessing steps, and ERPLAB allows you to create and analyze averaged ERP waveforms. The good news is that both of these toolboxes are free. The bad news is that Matlab is not free, and you will need it to run EEGLAB and ERPLAB. However, most institutions provide reduced-cost Matlab licenses, and the student version is even less expensive. Matlab has become the *lingua franca* of cognitive neuroscience, and it's well worth the investment.

The ERP Core

The ERP CORE is a set of six classic ERP paradigms that have been optimized to isolate seven widely-studied ERP components (N170, mismatch negativity, N400, P3b, N2pc, error-related negativity, and lateralized readiness potential). Emily Kappenman and I created the ERP CORE to provide a set of "reference" data that could be used by a large set of researchers for a wide range of purposes. The public resource includes the experimental control scripts, data from 40 neurotypical young adults, and the EEGLAB/ERPLAB processing scripts. If you'd like to know how to obtain a robust N400, you can download our N400 experimental control script to see how it's done. If you'd like to see how to professionally analyze the error-related negativity (ERN), you can download our ERN processing scripts. If you've just put together your own EEG recording system and you'd like to see if everything is working, you can run or more of our paradigms and compare your data with our data (including quantitative metrics of data quality).

The ERP CORE is particularly useful for this book because it provides data from many different paradigms, and yet the data are similarly formatted for each paradigm. That way you can see how to process many different types of data, but you won't have to deal with superficial differences between data sets (e.g., differences in file naming conventions). Although you can download the ERP CORE files directly from the ERP CORE site, you should instead download the data using the links provided within this book for each exercise, which provide a more streamlined set of files.

Scripting

EEGLAB has a graphical user interface (GUI) that allows you to process data by pointing and clicking, and ERPLAB works as a plugin to EEGLAB. Many people use EEGLAB and ERPLAB entirely through the GUI. With the exception of the last chapter, all the exercises in this book use the GUI.

However, you can achieve a lot of additional power and flexibility by writing Matlab *scripts*, which are text files that specify each processing operation with a line of code. Scripts allow you to automate the EEG and ERP processing steps, which is a huge time-saver (especially when your mentor or a reviewer makes you reprocess all of your data). If you already know how to write Matlab scripts, then you'll find it straightforward to write scripts with the EEGLAB and ERPLAB routines. If you don't know Matlab but you have some significant experience in one or more other programming languages, you'll be able to pick up Matlab pretty quickly (although it has a few quirks that you'll need to learn).

If you don't have much programming experience, EEGLAB and ERPLAB provide a good starting point for you to learn. Every operation that you perform in the EEGLAB/ERPLAB GUI corresponds to a line of code, and every time you perform an EEGLAB or ERPLAB operation in the GUI, that line of code is saved to a *history*. You can grab these lines of code from the history, paste them into a text file, and voila! You have a script!

To get real power and flexibility, however, you also need to learn a little bit about the Matlab programming language. Chapter 11 is devoted to teaching you how to write EEGLAB/ERPLAB scripts. It's designed for people at all levels of prior programming experience. However, it does assume that you know some basic programming concepts (e.g., variables, loops). If you want to learn scripting—which is an incredibly useful skill—I recommend taking a Matlab course and/or working through one or more Matlab books. I particularly recommend a book called *Matlab for Behavioral Scientists* (Rosenbaum et al., 2014) and the online Introduction to Programming with MATLAB course from Coursera.

I've also provided example scripts at the end of each chapter, showing you how to implement the GUI steps from that chapter in a script. I find that it's much easier to start with an example script and modify it than to write a script from scratch.

Although the chapter on scripting is the last chapter of the book, you might want to read the first half sooner than that so that you understand the essence of EEGLAB/ERPLAB scripting. The last half of Chapter 11 uses processing steps that are covered in Chapters 2-10, so you should probably save that half until later.

Expected Background Knowledge

This book assumes that you already have some very basic knowledge about ERPs. If you don't, Appendix 1 provides a quick overview. Here are some things you'll need to know right away:

Generation of the EEG from postsynaptic potentials in cortical pyramidal cells

- The 10/20 system for electrode locations
- Creating averaged ERPs from single-trial EEG epochs
- Artifact rejection and artifact correction
- ERP peaks and components

All of these issues are briefly covered in Appendix 1, and you should read about them if they are not already familiar.

If you want additional background, I recommend the first 2 chapters in *An Introduction to the ERP Technique* (Luck, 2014) or a chapter I wrote for the *APA Handbook of Research Methods* (Luck, 2012). Or, better yet, you can take my free online course, Introduction to ERPs, which typically takes about 4 hours to complete. The first 2 "chapters" of the online course would be enough to get you started and should take you less than an hour.

Much of the theory behind the analysis approaches described in this book is described in *An Introduction to the ERP Technique* (Luck, 2014). If you want to become an ERP researcher, you need to understand the reasons behind the recommended processing steps, so I recommend going back and forth between that book and the present book. You'll see lots of places in the present book where I point you to the relevant chapters in *An Introduction to the ERP Technique*.

You will also want to consult the online documentation for EEGLAB and ERPLAB if you want to understand some of the options and parameters in the software. There are millions of details about the operation of the software that I didn't want to repeat in

this book.

Read This Now or You'll Be Sorry!

The original heading for this section was “Troubleshooting,” but that sounds way too boring, and I figured many people would skip it. But don’t skip it! This is the most important section of this chapter.

Unless you already have years of experience with EEGLAB and ERPLAB, you’re bound to run into a few problems when you try to complete the exercises in this book. EEGLAB and ERPLAB are professional-strength software packages designed for state-of-the-art research, and the datasets used in the exercises are large and complex. As a result, I can’t foresee every possible problem that might arise on your individual computer, and you’ll probably encounter error messages, results that don’t match what are shown in the book, etc.

When you encounter one of these problems, you’ll certainly be frustrated and you might even be tempted to curse my ancestors. But these problems are actually an important part of the learning process. When you’re analyzing your own data, you’ll run into many of the same problems. In fact, the problems will probably be worse with your own data, because the data used in our exercises have been carefully chosen to avoid many common problems.

So, when you run into a problem, try to look at it as an opportunity for growth. Of course, you might let out a few expletives and need to spend a minute doing deep breathing exercises before you remember that the error message on the screen is actually a gift in disguise. To reduce your blood pressure and help you learn the art of troubleshooting, we’ve provided a Troubleshooting Guide in Appendix 2. I recommend skimming it now and then returning to it when you inevitably run into problems.

Here's a related but more positive piece of advice: Play! I learned the important of playing in science from one of my undergrad mentors, Allen Neuringer. If you simply follow the exercises in this book exactly as written, you’ll certainly learn a lot. But if you want to really understand what you’re doing, you should spend considerable time playing around with things (e.g., the various options in the ERPLAB GUI). For example, there are many things that I say you shouldn’t do, such as applying a high-pass filter to averaged ERP waveforms. But don’t just take my word for it – try it and see what happens.

This page titled 1.1: Getting Started is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

1.2: Installing the Software and Downloading the Data

Minimum hardware and software requirements were listed previously. However, keep in mind that those are only the minimum. You will find the exercises to be much more pleasant if you have at least 8 GB of RAM and at least a 1080p screen. If you don't have much free disk space, you may need to keep the data from only one or two exercises on your computer at a given time.

Installing Matlab

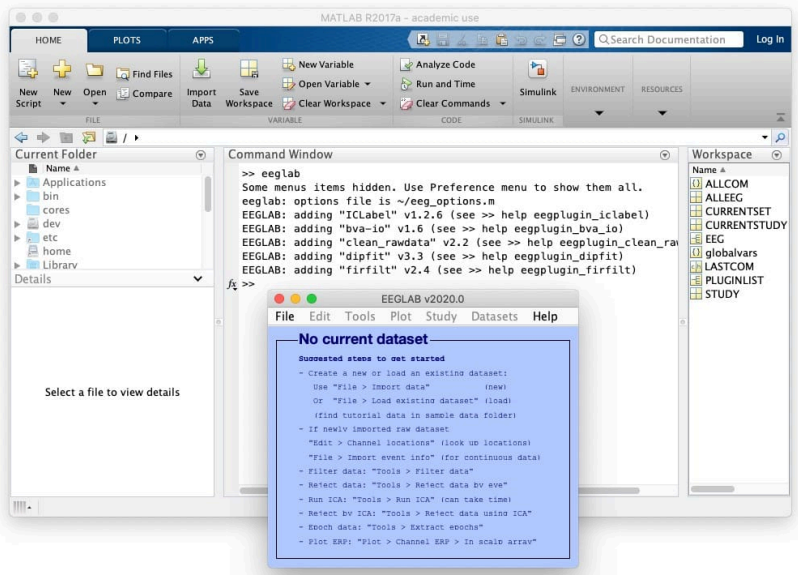
You will need Matlab version 2017a or later, including the Signal Processing Toolbox. The Statistics and Machine Learning Toolbox is recommended (but is not required for any of the exercises in this book). Once you have installed Matlab, you can see what toolboxes are installed by typing **ver** on the Matlab command line. If you don't have the necessary toolboxes, contact your institution's IT support department for assistance.

Installing EEGLAB and ERPLAB

New versions of Matlab, EEGLAB, and ERPLAB are released at least once per year, and these new versions can lead to changes in how things look (and occasionally changes in the results). To write this book, I mainly used Matlab 2017a or 2020b, EEGLAB 2020.0, and ERPLAB 8.23 on a MacBook Pro running macOS 11. However, we added some important features to ERPLAB as I was writing, so you should use EEGLAB 2022.0 or later and ERPLAB 9.0 or later (see the Hardware and Software Requirements section). You will probably use a different combination, so the screenshots and videos in this book may not exactly match what you see. Newer versions of Matlab, EEGLAB, and ERPLAB will probably work fine, but I would recommend against using older versions.

ERPLAB runs as an EEGLAB plugin, so you must install EEGLAB before installing ERPLAB. You can find the documentation for EEGLAB, including instructions for downloading and installing, at https://sccn.ucsd.edu/wiki/EEGLAB. Make sure to install version 2022.0 or later. Once you've installed it, you can launch EEGLAB by typing **eeglab** in the Matlab command window. Screenshot 1.1 shows what it should look like when EEGLAB has been launched.

Screenshot 1.1

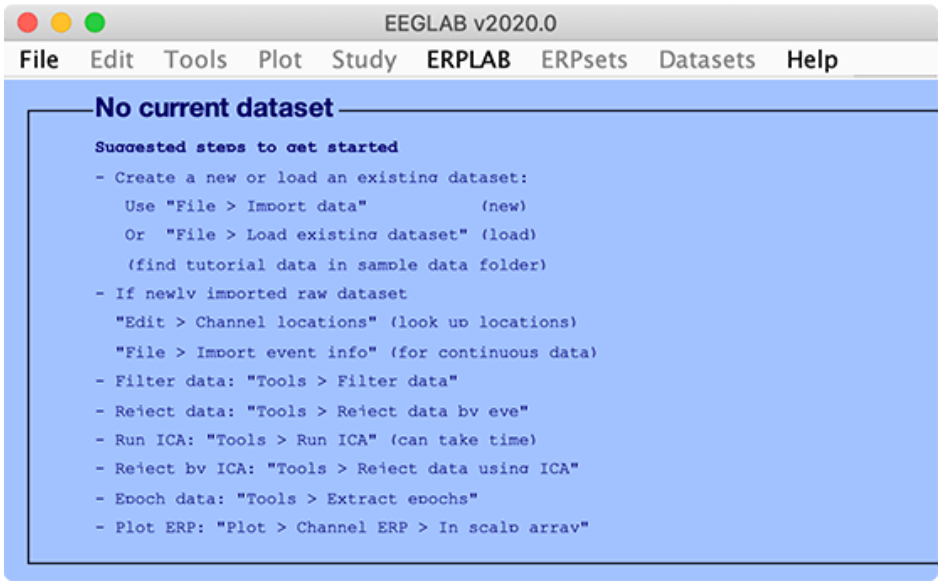


A key step in installing EEGLAB is to update the Matlab PATH (whether you are installing EEGLAB for the first time or upgrading to a newer version). If you don't know how to do this, or you can't get EEGLAB to launch, you should read the description of how the PATH works later in this chapter. If you are having trouble launching EEGLAB but the PATH isn't the problem, see the ERPLAB

FAQ page, the EEGLAB documentation or the Troubleshooting Guide in Appendix 2.

Once you have installed EEGLAB, you can download and install ERPLAB (version 9.0 or later). The ERPLAB documentation, including installation instructions, can be found at https://github.com/lucklab/erplab/wiki. There are two ways to install ERPLAB. If you just want the latest major release, you can launch EEGLAB and use its built-in extensions manager by going to **File > Manage EEGLAB extensions** in the EEGLAB GUI. It will pop up a window with a list of extensions; select **ERPLAB** and click **install/update**. If you want an earlier version of ERPLAB (or if you're a do-it-yourselfer by nature), you can download ERPLAB at https://github.com/lucklab/erplab. Additional ERPLAB installation information can be found at https://github.com/lucklab/erplab/wiki/Installation. Once ERPLAB has been installed and is running within EEGLAB, the EEGLAB GUI should include an ERPLAB menu, as in Screenshot 1.2.

Screenshot 1.2



ERPLAB Installation Problems

ERPLAB is installed inside the plugins folder within the EEGLAB folder (e.g., eeglab2020_0 > plugins > ERPLAB8.30). The most common installation problem is that ERPLAB is not located in this place. You should see a file named eegplugin_erplab.m inside that folder (and not inside another folder). ERPLAB has a Frequently Asked Questions page with information about solving such problems.

Another problem that occasionally arises (especially when ERPLAB is being used on a multiuser computer) is that ERPLAB must have write access to the folder that contains the ERPLAB software. This is necessary so that ERPLAB can update a file named **memoryerp.erpm**, which stores various user settings. To avoid this problem, you should install EEGLAB and ERPLAB inside a folder that you own (e.g., your Documents folder) rather than installing it in a folder that is used by multiple people.

Downloading Data for the Exercises

The data for the exercises are stored in a cloud storage system, and the master folder can be accessed at https://doi.org/10.18115/D50056. This master folder contains a set of subfolders with the data for each chapter. Unfortunately, the cloud storage system may be blocked in some countries, but I have faith that clever readers will still find a way to access the data. The data have been released with a Creative Commons license that permits them to be copied, used, remixed, and reposted as long as the original source is cited (see the **license.txt** file in each subfolder). Note that some of the folders are quite large and may take quite a while to download.

Free Data!

Once you've finished using this book to learn how to analyze ERP data, you should feel free to download the original data from the ERP CORE. The CORE is a very rich dataset, and you could use it to for novel analyses that might be publishable. Wouldn't it be nice to publish an ERP paper without having to collect the data yourself?

This page titled 1.2: Installing the Software and Downloading the Data is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

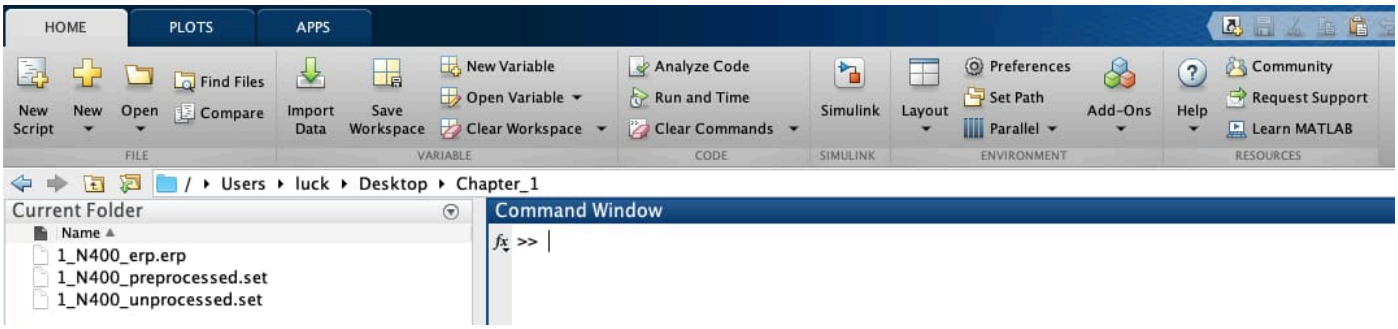
1.3: Exercise- Loading an EEG File

We're now going to start a quick tour of EEGLAB and ERPLAB. In the first exercise, you'll look at the EEG data and the averaged ERPs from one subject in an N400 experiment that will be described in more detail in the next chapter.

If Matlab is not already running, launch it. If EEGLAB is already running, quit from EEGLAB and launch it again by typing **eeglab** in the Matlab command window. (It's usually a good idea to quit and restart EEGLAB before starting an exercise. That way, EEGLAB will be in its default state, and what you see will better match the screenshots in this book.)

Now download the Chapter 1 folder from the master folder at https://doi.org/10.18115/D50056. Note that an underscore is used instead of a space in the folder name. Spaces can sometimes confuse Matlab, so it's best to use underscores or dashes instead of spaces in your folder names and file names. But don't use any other non-alphanumeric characters. Set Chapter_1 to be Matlab's *current folder*. The current folder is part of Matlab's path, and it's the first place Matlab will look for files and code. Screenshot 1.3 shows what it should look like once you've set the current folder. You'll see the path to the folder and the contents of the folder.

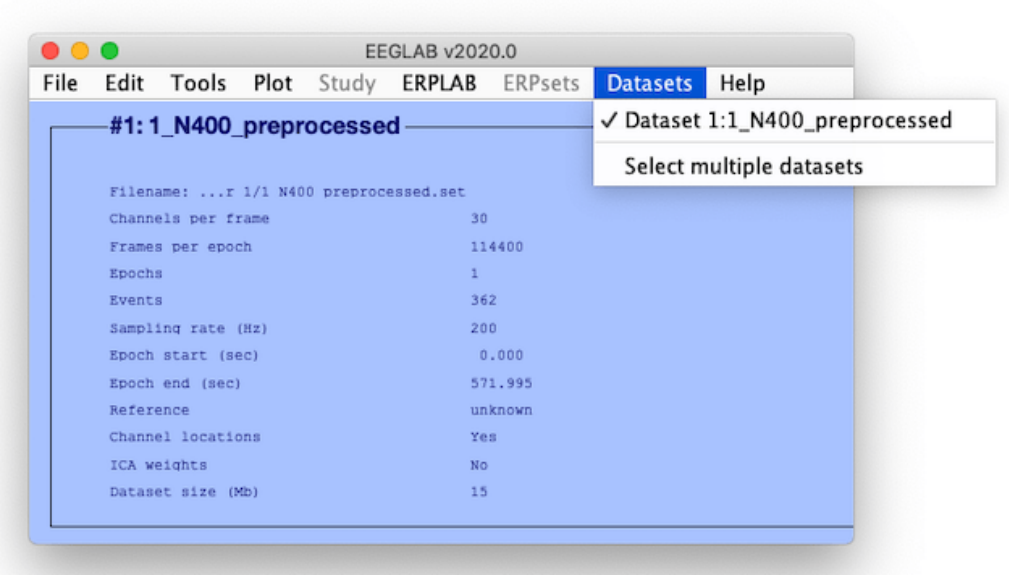
Screenshot 1.3



There are various ways of setting the current folder, including double-clicking on a folder name within the current folder. Play around with this a bit (Googling it if necessary using a search phrase such as "matlab current folder GUI"). You'll be changing the current folder frequently, so you'll want to know how to do it efficiently.

Once you have the correct current folder, go to the **EEGLAB** window and select **File > Load existing dataset**. It will bring up a dialog box, which should show the contents of the Chapter_1 folder. Select and open the file named **1_N400_preprocessed.set**. This file is called a **dataset**, which means that it contains the EEG data for one participant. You can see that it loaded correctly by looking in EEGLAB's **Datasets** menu (see Screenshot 1.4).

Screenshot 1.4



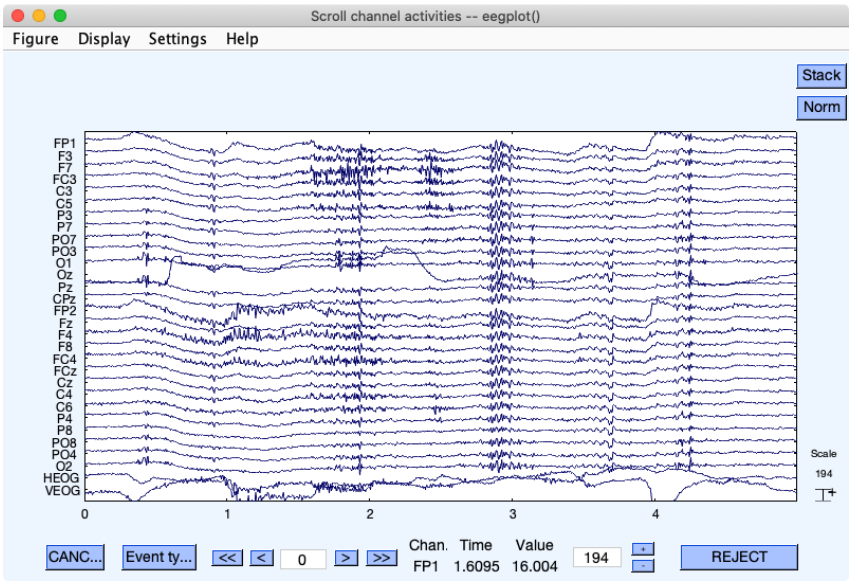
This page titled 1.3: Exercise- Loading an EEG File is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

1.4: Exercise - Viewing Continuous EEG Waveforms

Now let's take a look at the EEG data that we just loaded. In the EEGLAB window, go to **Plot > Channel data (scroll)**. This will be your main way of visualizing EEG data. You should see a new window that looks something like Screenshot 1.5. This window is showing the EEG data from all of the electrode sites for the first five seconds of the recording. If you click the **>>** button near the bottom of the window, it will scroll rightward to show you the next 5-second period. Using the <<, <, >, and >> buttons, scan through the file and see what the EEG looks like

If you type a number into the text box between the **<** and **>** buttons, it will go to that time point. Try typing **20** into that box (and hit Enter/Return). You should now see the data from 20-25 seconds.

Screenshot 1.5

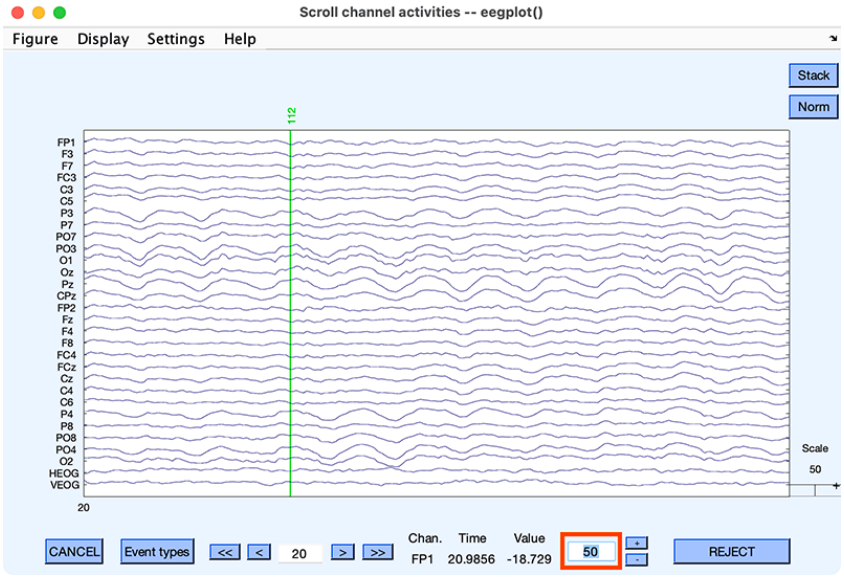


Now we're going to zoom in a little closer. You should be looking at the data from 20-25 seconds; if not, type **20** into the box between the < and > buttons. To zoom in on a shorter time period, go to the menus in the plot window and selection **Settings > Time range to display**, type a **1** into the text box, and click **OK**. You'll now be looking at the time period from 20-21 seconds instead of 20-25 seconds.

Now let's adjust the vertical zoom. If you look near the right edge of the window, you should see a vertical scale marker. On my computer, it says 194 µV (but it might be different on your computer). This means that the vertical space indicated by the marker corresponds to 194 µV. To zoom in closer, we need to do something a little counterintuitive: We need to use a smaller number of µV for that marker. For example, if we use 50 µV for that same vertical space, a smaller voltage deflection will now extend over a larger vertical range. To make this change, enter **50** into the vertical scale text box (indicated by the red box in Screenshot 1.6). Now you're zoomed in both in time and in amplitude.

If you look at the occipital and parietal electrode sites (e.g., O1), you can see an oscillating wave. Count the number of positive peaks. You should see 11 peaks in this 1-second period. 11 peaks per second means that it's an 11 Hz oscillation. This is called the *alpha* wave, which was the very first feature of the human EEG to be described (Berger, 1929). Alpha waves are oscillations at approximately 10 Hz that are largest over posterior electrode sites and are particularly large when the participant is zoned out or has closed eyes. I like to think of alpha waves as indicating that the participant's attention is internally focused instead of externally focused. In most experiments, you want the subject to be alert and paying attention externally to the task, so you don't want to see alpha waves. However, most ERP experiments are pretty boring, so the participants are often a little sleepy and generate quite a bit of alpha. When a participant produces a lot of alpha, we often give them a break and offer a caffeinated beverage.

Screenshot 1.6



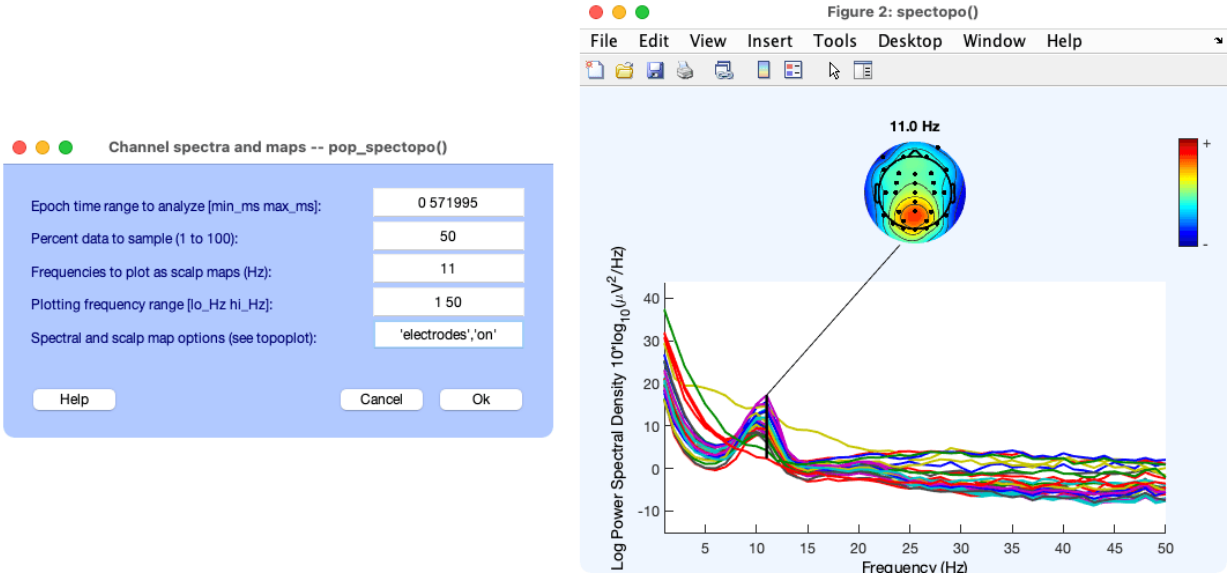
This page titled 1.4: Exercise - Viewing Continuous EEG Waveforms is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

1.5: Exercise - Viewing EEG Spectra

Now let's take a closer look at the alpha wave. Close the plotting window that shows the EEG, go to the main EEGLAB GUI, and select **Plot > Channel spectra and maps**. Adjust the bottom three text boxes so that they match the values shown in the left portion of Screenshot 1.7 and then click **OK**. You should now see a window like that shown on the right side of Screenshot 1.7.

This routine performs a Fourier transform of the EEG, converting it from the time domain to the frequency domain (see Chapters 1 and 12 in Luck, 2014). The waveforms at the bottom of the window show the strength of each frequency in the EEG data, with one waveform for each channel. You can see that most of the channels have a peak around 11 Hz, which is the alpha wave you saw previously. The topograph map at the top shows how the strength at 11 Hz varies across the scalp (using interpolation to estimate the values in between the electrode sites). You can see a big peak over the posterior electrode sites, where the alpha is largest.

Screenshot 1.7



Try playing around with the settings for this routine (e.g., getting scalp maps for different frequencies). When you're done, close the plotting window to prepare for the next exercise.

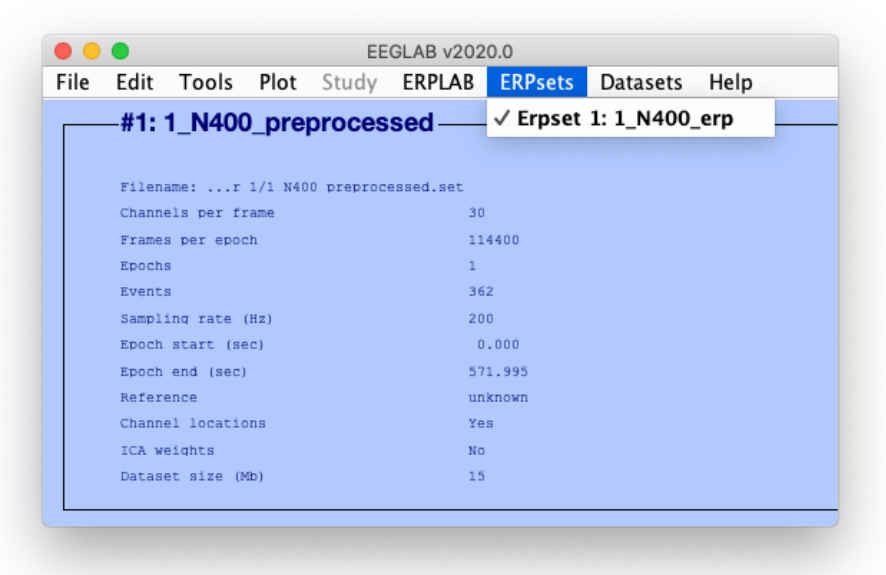
This page titled 1.5: Exercise - Viewing EEG Spectra is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

1.6: Exercise- Loading ERPs and Plotting ERP Waveforms

Now we're going to load some averaged ERP data and plot the waveforms. Take a look at the **ERPLAB** menu inside the EEGLAB window. ERPLAB is an EEGLAB plugin, and the **ERPLAB** menu is the main way you'll access the ERPLAB functions.

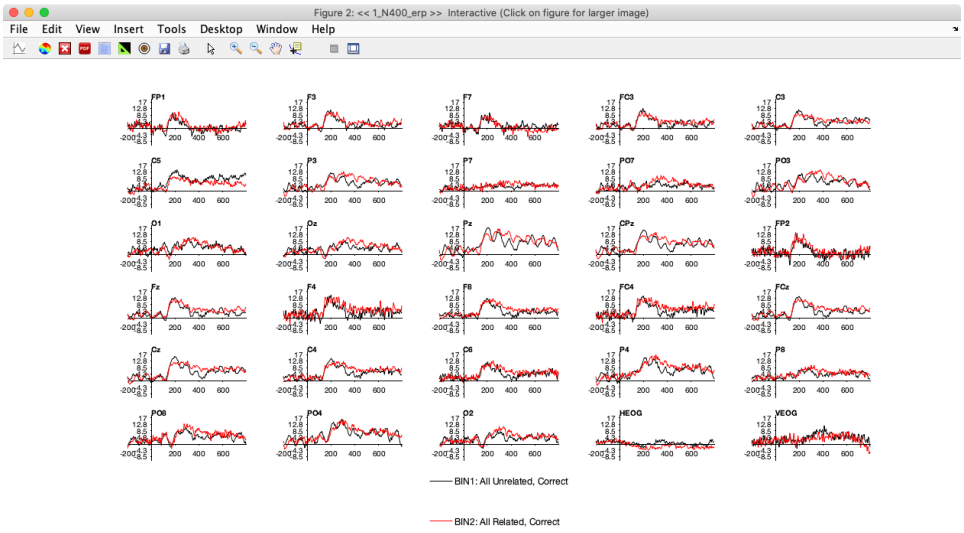
Select **ERPLAB > Load existing ERPset**, which will bring up a dialog box showing you the contents of the current folder. Select and open the file named **1_N400_erp.erp**. An ERPset is the ERPLAB equivalent of EEGLAB's datasets. Instead of holding EEG data, an ERPset contains averaged ERPs. I previously created this ERPset by averaging together the trials in the EEG dataset that you looked at in the previous exercise. We'll see how this averaging process works later. The currently loaded ERPsets are listed in the ERPsets menu (see Screenshot 1.8).

Screenshot 1.8



To visualize the ERP waveforms in an ERPset, go to **ERPLAB > Plot ERP > Plot ERP waveforms**. It will bring up a large and complicated window that allows you to control many different plotting parameters. Click the **RESET** button at the bottom window so that it uses its default parameters, and the click the **PLOT** button in the lower right corner of the window. You should see something like Screenshot 1.9. (Hint: If things don't look right when you plot a set of ERP waveforms, trying clicking the **RESET** button to get rid of any custom settings that you might have been using the last time you plotted some waveforms.)

Screenshot 1.9



This ERPset contains ERPs for two different **bins** (experimental conditions), labeled “All Unrelated, Correct” and “All Related, Correct”. There is a separate waveform for each electrode site for each of the two bins.

If you click an electrode label (like **FP1** for the waveforms in the upper left corner), it will pop up a new window with a blown-up version of the waveforms for that electrode site. The plotting GUI is a little twitchy and a little slow. If it doesn’t pop up the window after a second or two, try again (but wait a second before trying again or you’ll end up with multiple windows open). Also, you only need to click once; if you double-click the electrode label, you may end up with an extra copy of the blown-up window. Play around with this a bit, and close the plotting windows when you’re done.

The GUI that controls the plotting has a lot of options. For example, you select the **show standard error** option, it will show the standard error of the mean at each time point as a semitransparent cloud around the waveform. Play around with the options in this window so that you get a sense of what you can do.

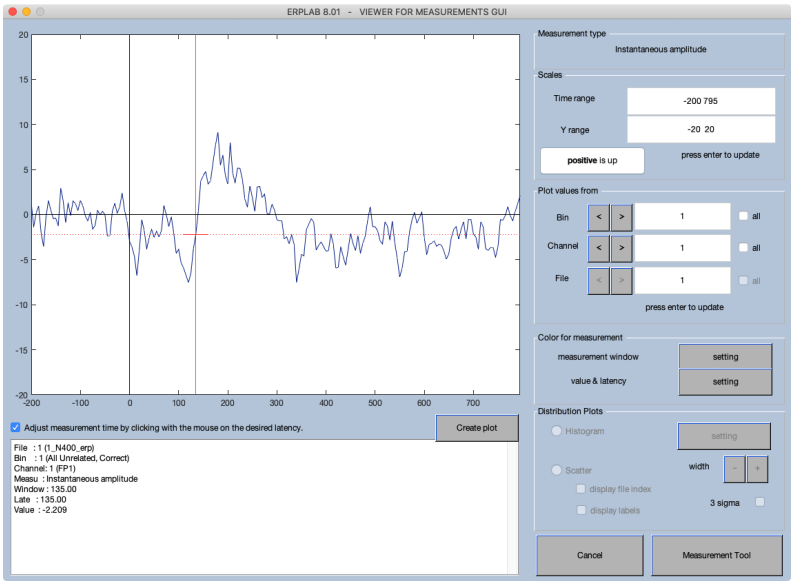
Making High-Quality Figures

Although ERPLAB’s tool for plotting ERP waveforms has a lot of options, it’s not designed to create publication-quality figures. To make nice figures for papers, posters, or presentations, you can plot the waveforms with this tool and then select **File > Save As...** from the plotting window. I recommend saving the plot in a vector format, such as PDF or SVG. In my lab, we save the files in SVG format and then load these files into a graphics program called **Affinity Designer** (which is like Adobe Illustrator, but easier to use and much less expensive).

Save As... gives you a lot of other file format options, including several bitmap formats (e.g., .bmp, .jpg, .tif). I recommend against using bitmap formats, because they’re difficult to edit and look terrible when blown up. If you don’t know the difference between vector and bitmap formats, you should search for “vector versus bitmap” in your favorite search engine. It’s an important distinction when you’re trying to plot ERP waveforms.

You can also plot the waveforms by going to **ERPLAB > ERP Viewer** (see Screenshot 1.10). Give it a try. This tool ordinarily shows you one waveform at a time (although you can overlay multiple waveforms by checking the boxes labeled **all**. You can scan through different bins and channels by clicking the arrow buttons next to **Bin** and **Channel**. If you click the box labeled **Adjust measurement time by clicking with the mouse on the desired latency** (near the left side of the window), you can click a point on the waveform and see the amplitude and latency at that point. This tool also provide a convenient way of overlaying waveforms from different files.

Screenshot 1.10



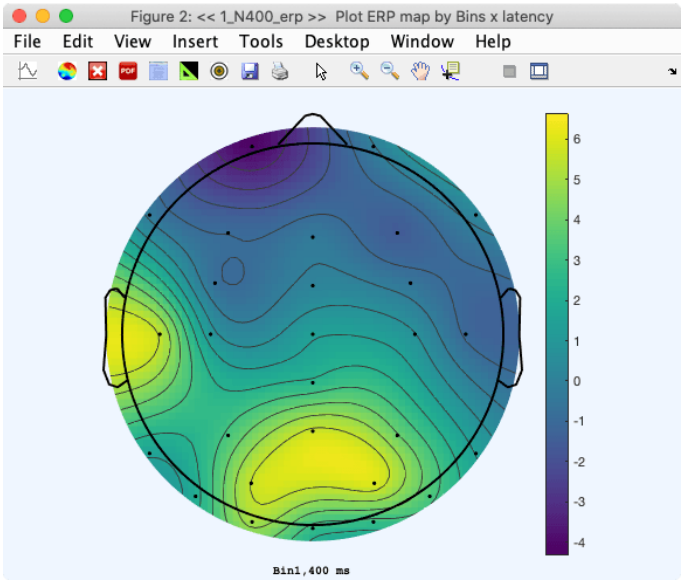
This page titled 1.6: Exercise- Loading ERPs and Plotting ERP Waveforms is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

1.7: Exercise- Plotting ERP Scalp Maps

As a last step in our quick tour of EEGLAB and ERPLAB, we're going to plot a *topographic map* (aka *scalp map*) of the ERPs.

Select **ERPLAB > Plot ERP > Plot Scalp maps**, which will bring up a dialog box that lets you control the plotting parameters. Enter **1** into the text box at the upper left labeled **Bin(s) to plot** (so that it plots bin 1) and enter **400** into the text box labeled **Latencies to plot in ms** (so that it plots the voltage at 400 ms). Then check the box labeled **display color scale box** near the right edge of the window. Finally, click the **PLOT** button in the lower right corner. You should see something like Screenshot 1.11.

Screenshot 1.11



Each little dot on the head is one of the electrode sites. The coloring indicates the voltage at each point on the scalp (using interpolation to fill in the values between electrodes). The scale along the right side indicates what amplitude (in microvolts) is indicated by a particular color.

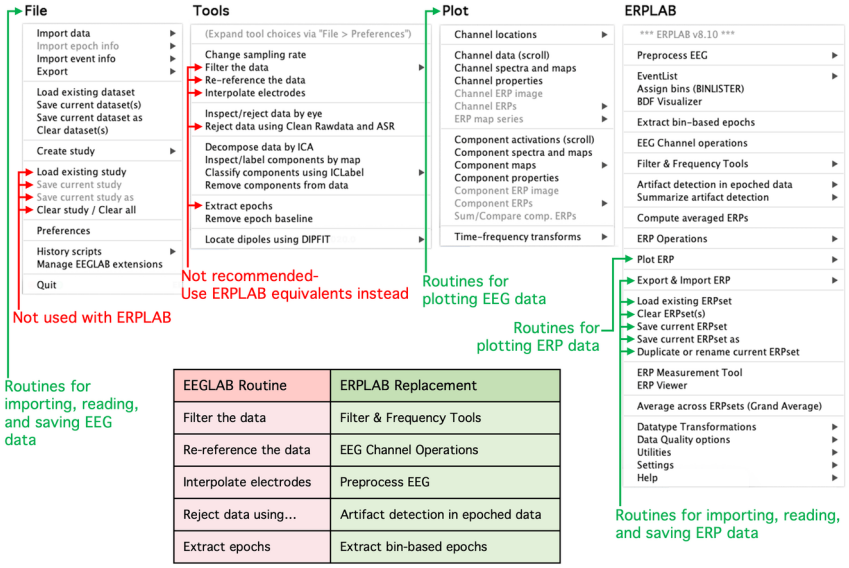
This page titled 1.7: Exercise- Plotting ERP Scalp Maps is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck.

1.8: Finding the Right Routine in EEGLAB and ERPLAB

The fact that ERPLAB works as an EEGLAB plugin has many advantages. However, it occasionally produces some complications. For example, when you are first learning to use EEGLAB and ERPLAB, you will probably find that you occasionally look in the wrong menu. For example, it's natural to go to the **File** menu when you want to load an ERP data file or the **Plot** menu when you want to plot an ERP waveform. However, all ERP-related routines are in the ERPLAB menu (e.g., **ERPLAB > Load Existing ERPset** and **ERPLAB > Plot > Plot ERP Waveforms**).

To make things even more complicated, we've added several routines for processing EEG data to the ERPLAB menu (e.g., **ERPLAB > EEG Channel Operations**). We created these routines for several reasons. In some cases, EEGLAB just didn't have a piece of functionality that we needed. In other cases, we wanted to create improved versions of functionality that was already present in EEGLAB. And in a few cases, the needs of typical ERP analysis required somewhat different versions of existing EEGLAB functions. To keep ERPLAB modular, all of these new or updated EEG functions are accessed from the ERPLAB menu.

To help you find what you're looking for and use the appropriate routines, Figure 1.1 shows the key menus and indicates which EEGLAB functions should be replaced by ERPLAB functions.



EEGLAB Routine	ERPLAB Replacement
Filter the data	Filter & Frequency Tools
Re-reference the data	EEG Channel Operations
Interpolate electrodes	Preprocess EEG
Reject data using...	Artifact detection in epoched data
Extract epochs	Extract bin-based epochs

Figure 1.1. Key menus from the EEGLAB GUI. You will use the ERPLAB menu for all ERP-related operations and the other menus for EEG-related operations. For example, you will use the **File** menu to import and open EEG files ("datasets"), but ERP files ("ERPsets") are loaded from the ERPLAB menu. Similarly, you will use the **Plot** menu to plot EEG data, but **ERPLAB > Plot** to plot ERP data. Also, some EEGLAB functions cannot be used in conjunction with ERPLAB (the **study** functions and the **Extract epochs** function), and other functions will work but are not recommended because analogous functions are available in the ERPLAB menu (e.g., filtering and re-referencing).

This page titled 1.8: Finding the Right Routine in EEGLAB and ERPLAB is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

1.9: Understanding the Matlab Path

Now that you've had a chance to play around with EEGLAB and ERPLAB a bit, I want to discuss a vitally important concept, the *PATH*. Many of the problems that Matlab beginners encounter are a result of not understanding and appropriately setting the PATH. If you're already quite familiar with the Matlab PATH, you can skip this section.

Simply put, the Matlab's PATH defines where it will look for code and other files. When you type **eeglab** into the Matlab command window to launch EEGLAB, the only reason that Matlab knows where to find the EEGLAB code on your computer is that you (or someone) set the PATH to include the location of the files containing the EEGLAB code. And if you're reading this section because you're having a problem getting EEGLAB to run, there's a good chance that you're having a PATH problem.

As you know, your computer's file system is divided into a set of folders (also called *directories*). The folders are hierarchically organized: You have a main folder for each drive, and those folders contain sub-folders, which contain sub-sub-folders, etc. The location of a file can be expressed concisely as something like **D:\books\ERP_Analysis_Book\Exercises\Chapter_2\1_N400_erp.erp** (on a Windows system) or **/Users/luck/Documents/books/ERP_Analysis_Book/Exercises/Chapter_2/1_N400_erp.erp** (on a Mac or Linux system). To make life complicated, DOS and Windows use backslashes to separate the names of the folders, whereas Mac and Linux use forward slashes.

The location of a single file, when expressed this way, is the *path* to the file. But note that I'm using lower case to refer to the path for a single file. I use upper case to refer to Matlab's PATH, which consists of a list of multiple paths. This list defines where Matlab will look for code and other files. But note: Matlab will first look in the current folder (indicated by the blue arrow in Figure 1.2) before looking in the folders defined by the PATH.

Let's take a look at the PATH on your computer. In the main Matlab window, click on the icon labeled **Set Path** (circled in red Figure 1.2, but note that it might be somewhere else in your version of Matlab). This should cause a new window appear, like the one shown at the bottom of Figure 1.2.

You should see the main folder for EEGLAB in the PATH. If you recently installed EEGLAB, it should be at the top. If you're having trouble getting EEGLAB to launch, make sure you have the EEGLAB path in your PATH. To add the EEGLAB folder to your PATH, click on **Add Folder...**, navigate to the location of the EEGLAB folder that you downloaded when you installed EEGLAB, and add it to the PATH. It's also possible that the path is incorrect (e.g., because the folder was moved after the PATH was set). If you already have EEGLAB in your PATH but you don't know how to verify that the path is correct, you can delete the existing EEGLAB path (including all the subfolders) using the **Remove** button and then add the path again using **Add Folder...**

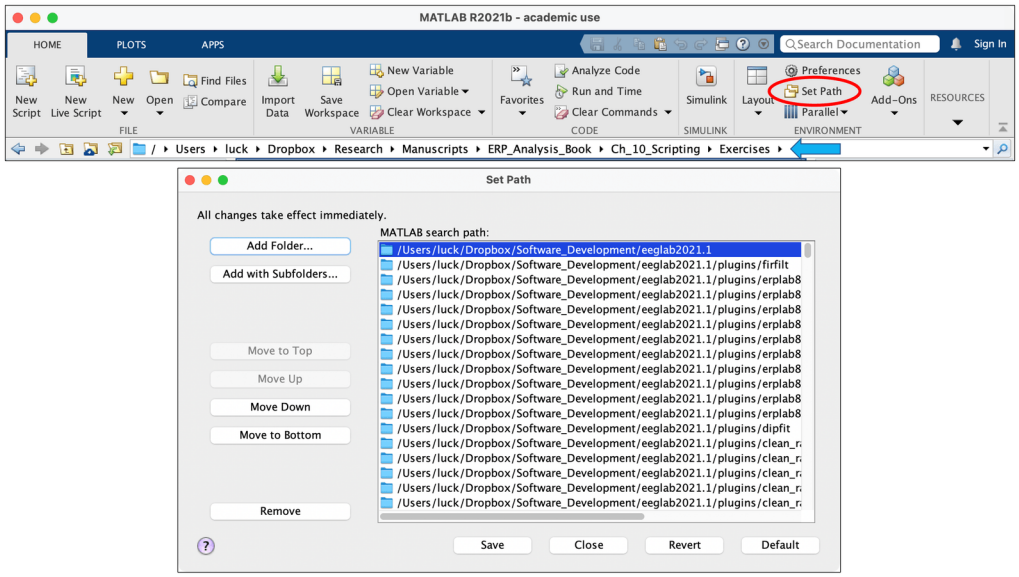


Figure 1.2. Setting the PATH in Matlab. To set the PATH, you click on the **Set Path** icon in the main Matlab window (circled in red here). This brings up the dialog box. Note that the current folder is indicated by the blue arrow. You can click on **Add Folder...** to add a new folder to the PATH. You can also remove items from the PATH using the **Remove** button. Make sure to click **Save** so that the PATH is saved for future Matlab session. Note that this might look different on your computer.

When you're done updating the PATH, click **Save** and then **Close**. If you click the **Close** button without saving, the updated PATH will work, but it will be forgotten when you quit from Matlab. You need to click **Save** before you click **Close** if you want Matlab to remember the path when you launch it again in the future.

Now let's talk in a little more detail about how the PATH works. A complicated package like EEGLAB or ERPLAB is divided into tons of individual files (often called **.m files** because the filename ends in **.m**). When you type **eeglab** in the Matlab command window, Matlab searches the path for a file named **matlab.m**, and then it executes the code in this file. The code in the **eeglab.m** file calls many other functions, such as **eeg_checkset()**. The code for a given function is typically stored in a separate .m file. When Matlab needs to call **eeg_checkset()**, it therefore looks for it in a file named **eeg_checkset.m**. Where does Matlab look to find this file? In the folders defined by the PATH, of course. The **eeg_checkset.m** file is actually located in a subfolder within the main EEGLAB folder. This is why there are actually many EEGLAB subfolders in your PATH.

When you add a new folder to the PATH, you would ordinarily click on **Add with Subfolders...** to make sure that all the subfolders within the new folder are in your path. However, EEGLAB is smart enough that it will automatically add the subfolders when you click **Add Folder...**

Here's the key takeaway from what we've discussed so far: Almost every command and function in Matlab is stored in a .m file, and when you try to execute a command or function, Matlab searches the PATH to find it. If Matlab tells you that it can't find a function or a .m file, this almost always means that you don't have the PATH set correctly.

Another problem that can arise is that there might be two different .m files with the same name in your path. For example, if you install a new version of EEGLAB on your computer, and you add the folder containing this version to your PATH without removing the old path, Matlab might execute the wrong version. So, make sure to clean out the old folder (and any subfolders) using the **Remove** button when installing a new version.

It's also possible that you have another package (e.g., FieldTrip or PsychToolbox) in your PATH that defines a function with the same name as one of the EEGLAB functions. For example, both EEGLAB and FieldTrip could have a function named **PlotEEGWaveforms**, which would be stored in a file named **PlotEEGWaveforms.m**. You would then have two identical files in your PATH. Which file will Matlab use when it tries to call the **PlotEEGWaveforms** function? The answer is simple: It searches the folder in the order that they're listed in the PATH, and it stops searching once it finds a match.

Yet another common problem is that the path to a folder may have one or more spaces, such as **D:\books\ERP Analysis Book\Exercises**. That's usually not a problem, but sometimes Matlab will interpret such a path as being three separate things (**D:\books\ERP** and **Analysis** and **Book\Exercises**). If this happens, you can usually solve the problem either by using underscores instead of spaces or by enclosing the path in single quotes (e.g., **'D:\books\ERP Analysis Book\Exercises'**).

If you want more information, Matlab's online documentation provides a detailed description of how the PATH works.

This page titled 1.9: Understanding the Matlab Path is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

1.10: Key Takeaways and References

Key Takeaways

- This book focuses on applied ERP data analysis, using the open source EEGLAB and ERPLAB analysis packages. The packages are free, but they require Matlab to run.
- Most of the analysis examples are from the ERP CORE, a set of 6 standardized ERP experiments.
- You will almost certainly run into technical problems, but a major goal of the book is for you to learn to solve these problems. Appendix 2 provides a Troubleshooting Guide that will teach you some useful principles for solving technical problems.
- Because ERPLAB is an EEGLAB plugin, you may have trouble figuring out where to find the routines you need and determining whether to use an EEGLAB routine or an ERPLAB routine. You can refer to Figure 1.1 to help you find what you're looking for.

References

Berger, H. (1929). Ueber das Elektrenkephalogramm des Menschen. *Archives Fur Psychiatrie Nervenkrankheiten*, *87*, 527–570.

Delorme, A., & Makeig, S. (2004). EEGLAB: an open source toolbox for analysis of single-trial EEG dynamics including independent component analysis. *Journal of Neuroscience Methods*, *134*, 9–21.

Lopez-Calderon, J., & Luck, S. J. (2014). ERPLAB: An open-source toolbox for the analysis of event-related potentials. *Frontiers in Human Neuroscience*, *8*, 213. https://doi.org/10.3389/fnhum.2014.00213

Luck, S. J. (2012). Event-related potentials. In H. Cooper, P. M. Camic, D. L. Long, A. T. Panter, D. Rindskopf, & K. J. Sher (Eds.), *APA Handbook of Research Methods in Psychology: Volume 1, Foundations, Planning, Measures, and Psychometrics* (pp. 523–546). American Psychological Association.

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Rosenbaum, D. A., Vaughan, J., & Wyble, B. (2014). *MATLAB for Behavioral Scientists* (2nd Edition). Routledge.

This page titled 1.10: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2: PROCESSING THE DATA FROM ONE PARTICIPANT IN THE ERP CORE N400 EXPERIMENT

Learning Objectives

In this chapter, you will learn to:

- Load **EEG** data
- Create an **EVENTLIST**, which stores information about events such as stimuli and responses
- Assign events to bins with **BINLISTER**
- A *bin* is a set of ERP waveforms, one for each channel, that were created by averaging together a set of trials
- Parse the continuous EEG into a series of discrete epochs (e.g., from -200 to +800 ms relative to stimulus onset) and perform baseline correction
- Perform simple artifact rejection
- Compute and plot averaged ERP waveforms
- Filter EEG and ERP data
- Make difference waves

This chapter goes through a simple example experiment so you can see the basic steps involved in processing the data from a single participant. The experiment is from the ERP CORE (Kappenman et al., 2021), and it is designed to isolate the N400 component that is elicited when a word is semantically unrelated to a previous word (e.g., SHOE when it's preceded by TREE instead of by SOCK).

We'll go through the most basic EEG processing steps (e.g., epoching, baseline correction, artifact detection) and create averaged ERPs for a single participant. In the next chapter, we'll process additional participants, measure the N400 amplitude, and do a simple statistical analysis. The details of these steps will be described in later chapters, along with important details about how Matlab, EEGLAB, and ERPLAB work. These two chapters are designed for you to get the big picture of how the data are processed and learn the basics of using EEGLAB and ERPLAB. You can also find an overview of an entire EEG processing pipeline in Appendix 3.

If you're already experienced with ERPLAB, you can just skim through this chapter. But if you don't have much ERPLAB experience, you'll want to download the data and do each data processing exercise. Remember, you'll probably run into some error messages or other technical snags, but don't get discouraged. An important implicit goal of this book is for you to learn how to troubleshoot technical problems. See the troubleshooting tips in Appendix 2 if you have problems.

2.1: Data For This Chapter

2.2: Design of the N400 Experiment

2.3: Exercise - Looking at the EEG and the Event Codes

2.4: Exercise- Filtering Out Low-Frequency Drifts from the EEG

2.5: Exercise- Creating an EventList

2.6: Exercise- Assigning Events to Bins with BINLISTER

2.7: Exercise- Epoching and Baseline Correction

2.8: Exercise- Artifact Detection

2.9: Exercise- Averaged ERPs

2.10: Exercise- Data Quality

2.11: Review of Processing Steps

2.12: A Simple Matlab Script

2.13: Key Takeaways and References

This page titled 2: Processing the Data from One Participant in the ERP CORE N400 Experiment is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.1: Data For This Chapter

This chapter, like every subsequent chapter in this book, is filled with exercises. Processing real data is the best way to learn ERP data analysis! Your first step in each chapter will therefore be to download the data for the exercises in that chapter.

The N400 data we'll be using for the exercises in this chapter can be found in the Chapter_2 folder in the master folder: https://doi.org/10.18115/D50056. In this chapter, we'll be looking at the data from only one of the 40 participants (Subject #6). Go ahead and download this folder and all its contents (and make sure to name the folder **Chapter_2**).

I recommend organizing the data for the various chapters of the book on your computer just as they're organized online, with a separate folder for each chapter.

This page titled 2.1: Data For This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.2: Design of the N400 Experiment

Before we look at the data, let's consider the design of the ERP CORE N400 experiment. As illustrated in Figure 2.1.A, the experiment involved a sequence of trials, each of which consisted of a *prime* word followed by a *target* word. The participants' task was to press one of two buttons on each trial to indicate whether the target was semantically related or semantically unrelated to the preceding prime word. For example, they would press the *related* button for CHAIR preceded by TABLE and the *unrelated* button for SPIDER preceded by RAKE. Additional methodological details can be found in Kappenman et al. (2021).

Hundreds of previous studies have shown that a word will elicit a larger N400 component if it is unrelated to a previously established semantic context than if it is related to that context (Kutas, 1997; Swaab et al., 2012), so we expected to see a larger N400 on unrelated trials than on related trials. Figure 2.1.B shows the ERP waveforms, recorded at the CPz electrode site (where the N400 is typically largest) and averaged over all 40 of the original participants (a *grand average*). In these waveforms, the N400 is a negative-going wave for the unrelated targets that is present from approximately 200-600 ms and is added onto the positive voltage that is ordinarily present during this time period.

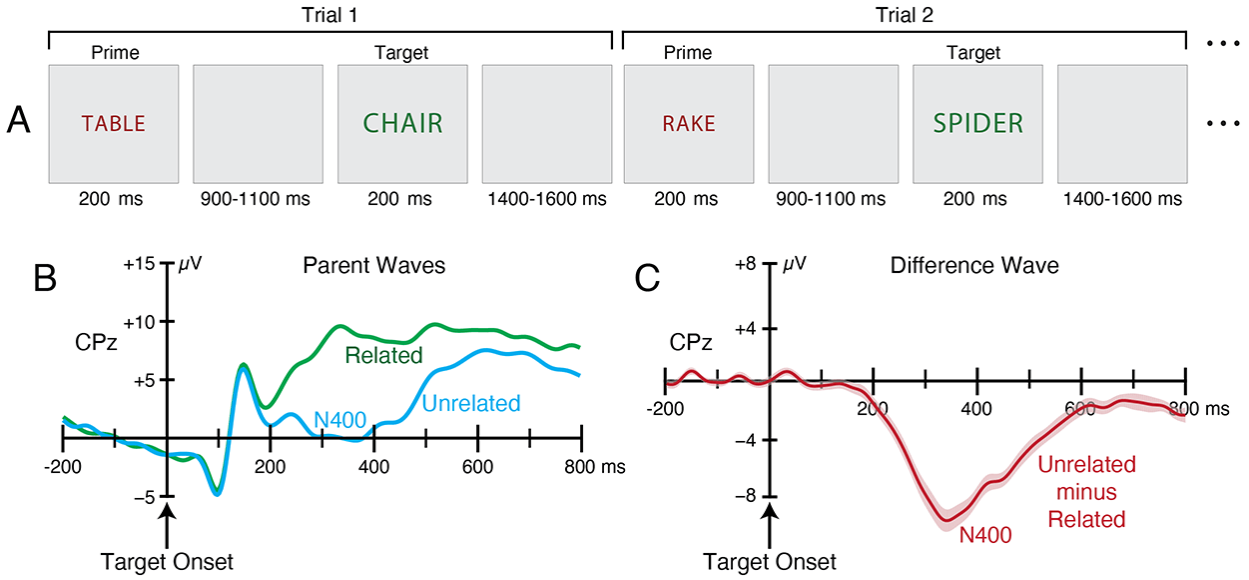


Figure 2.1. Experimental paradigm and results from the ERP CORE N400 experiment. (A) On each trial, subjects saw a prime word drawn in red and a target word drawn in green. The task was to press one of two buttons following each target word to indicate whether the target was semantically related or semantically unrelated to the preceding prime word. (B) ERP waveforms elicited by the target words, averaged over all 40 participants in the original study. (C) Difference wave created by subtracting the waveform on the related trials from the waveform on the unrelated trials.

It is often useful to subtract away everything that is in common to two conditions and focus on the difference in brain activity between conditions. To do this, we compute a *difference wave*, which is simply the difference in voltage between the two conditions at each time point. Figure 2.1.C shows the unrelated-minus-related difference wave for our N400 experiment. It allows us to see the brain's differential processing of unrelated versus related words. Because the difference deviates from zero at approximately 200 ms, we can conclude that the brain has determined whether the target is related or unrelated to the prime by this time.

This page titled 2.2: Design of the N400 Experiment is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.3: Exercise - Looking at the EEG and the Event Codes

In our first exercise in this chapter, we'll take a look at the EEG and the event codes for one participant (Participant #6). You should always scan through a participant's EEG and event codes before starting to process the data. Many things can go wrong during an EEG recording, and you want to make sure that there aren't any problems that you need to address before you go further. I provide a detailed example of how to visually inspect a participant's EEG in the chapter on artifact rejection (Chapter 7).

Remember, event codes are stored in the EEG file during data collection and indicate what event happened (e.g., which particular stimulus or response) and the time of occurrence (see Figure A1.1 in Appendix 1). In the system we used for collecting the ERP CORE data, event codes were integers between 1 and 255 (see the box below for more information about event codes). Table 1 lists the event codes for the N400 experiment. An event code was produced for each prime word and each target word, and different codes were used depending on whether the target was semantically related or unrelated to the prime on that trial. Note that we used two different lists of words (for counterbalancing purposes), and that was also indicated by the event codes. Each subject saw a total of 120 trials, 60 on which the target word was related to the prime word and 60 in which they were unrelated. Thus, there were 30 occurrences of each stimulus event code (30 related and 30 unrelated for each of the two lists).

Table 2.1. Event codes for the ERP CORE N400 experiment.

	Word Type	Relatedness	Word List	Event Code	Occurrences
Stimuli	Prime	Related	List 1	111	30
	Prime	Related	List 2	112	30
	Prime	Unrelated	List 1	121	30
	Prime	Unrelated	List 2	122	30
	Target	Related	List 1	211	30
	Target	Related	List 2	212	30
	Target	Unrelated	List 1	221	30
	Target	Unrelated	List 2	222	30
Responses	Correct			201	Variable
	Incorrect			202	Variable

Event Codes

In most EEG systems, one computer is used for presenting stimuli, and a different computer is used for recording the EEG. In addition, the experimenter often uses general-purpose stimulus presentation software that is not made by the manufacturer of the EEG recording system. To ensure compatibility, the method typically used to pass event codes from the stimulus presentation computer to the EEG recording computer is based on a communications protocol that has been around since 1970 (the Centronics-style parallel port). The protocol requires that the event codes be whole numbers between 1 and 255. That's plenty for some experiments, but it's woefully inadequate for others. In our N400 experiment, for example, it would have been nice for each event code to indicate the actual word that was presented. The ERP CORE online resource therefore provides text files (in CSV format) with the actual words used for each participant.

Some EEG systems come with stimulus presentation software, and they use custom protocols to allow for richer event codes (which might be text instead of 8-bit integers). However, this is not a very general solution.

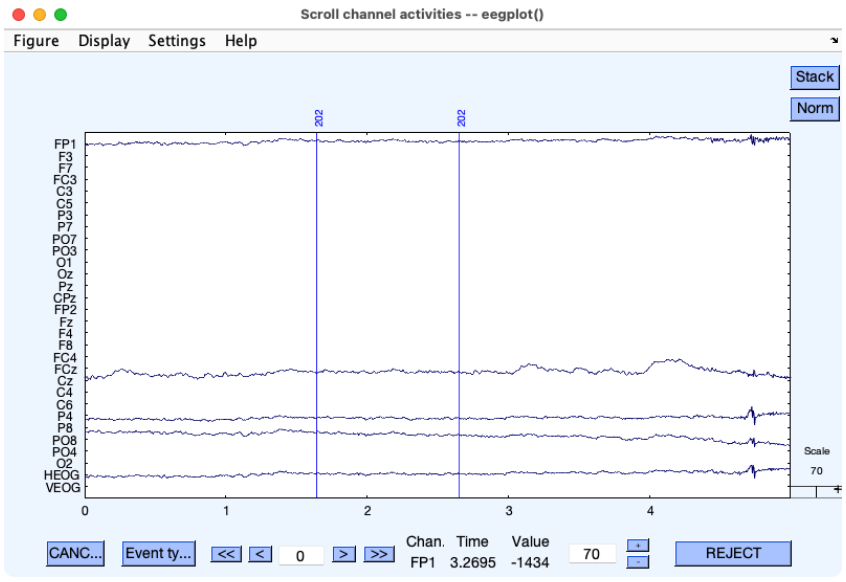
There is now a movement to use a newer standardized protocol called Lab Streaming Layer. This will make it possible to use an Ethernet cable instead of a parallel port and send much richer event codes, while still providing a standardized protocol that any software package can implement. I'm looking forward to the day when both my EEG recording system and our various stimulus presentation programs can use this newer protocol.

In the exercises for this chapter, we're going to look at the data from Subject 6, who has particularly nice data.

If Matlab isn't already running, launch it now and start EEGLAB (by typing **eeglab** in the Matlab command window). In Matlab, set the current folder to be the **Chapter_2** folder. On the left side of the Matlab window, you should see the contents of this folder, including a file named **6_N400_preprocessed.set**. That file contains the EEG from Subject 6. A few minor preprocessing steps have already been conducted to make the exercises in this chapter a little easier.

In the EEGLAB GUI, select **File > Load existing dataset** and select the file **6_N400_preprocessed.set** (be careful that you don't accidentally select **Load existing study** instead). Then select **Plot > Channel data (scroll)**. You should see a plot that looks like Screenshot 2. 1. You can see the EEG waveforms for only a few of the channels. This is because the voltages are out of the range of the plotting display for most of the channels, which is a result a DC voltage offset that arises mainly from the skin (see Chapter 5 in Luck, 2014). That is, the voltage recorded in our EEG electrodes is the sum of the EEG plus any voltage offset, and the voltage offsets are often so large that they make the signal go beyond the range of values shown in the plot. You didn't experience this when you did the exercises in Chapter 1 because the low frequencies had already been filtered out in the data in used in those exercises, which minimizes the DC offset.

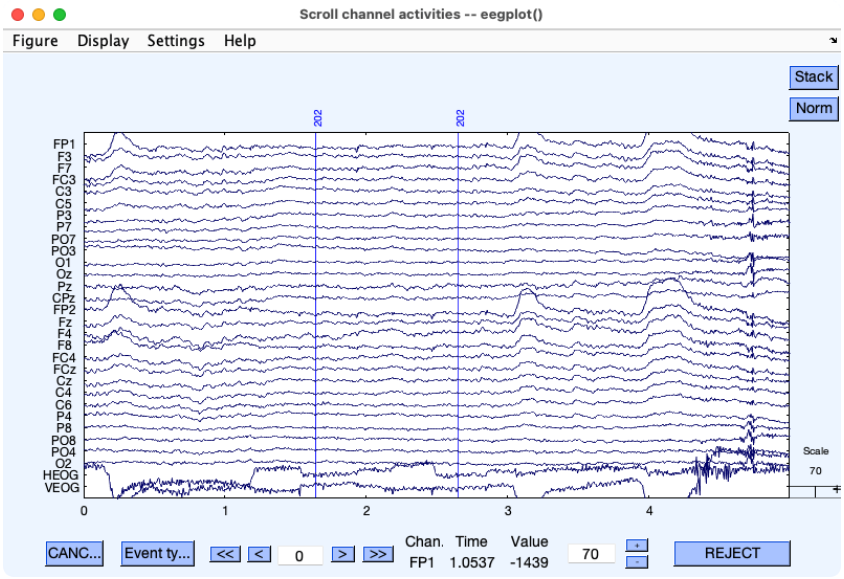
Screenshot 1.1



To see all of the channels, you can tell the plotting window to subtract the DC offset (i.e., to subtract the mean voltage across time

points from the voltage at each time point, separately for each channel). In the EEG plotting window, select **Display > Remove DC offset**, and then you'll see all the channels, as shown in Screenshot 2.2.

Screenshot 2.2



Someday, when you're loading your own EEG data into EEGLAB, you might see a completely blank screen when you try to plot the EEG. I'm hoping that you'll remember that this means you need to remove the DC offset.

Now let's look at the event codes in the EEG plot. Do you see the two vertical lines with a label of **202** at the top of each line? Those are event codes. If you look at Table 2.1, you'll see that event code 202 corresponds to an incorrect behavioral response. In this task, there are some instruction screens at the beginning, and the participants are required to press a response button to go to the next screen. Those responses generated an event code 202.

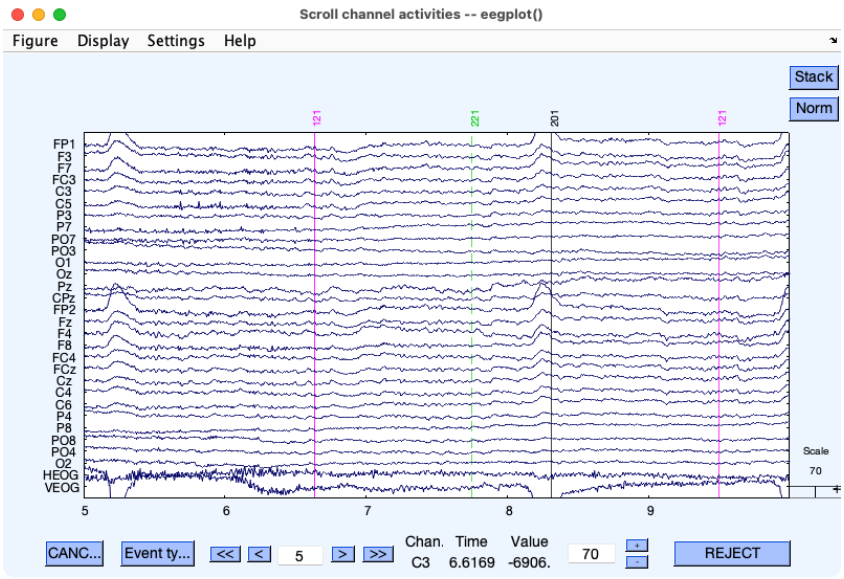
Pro tip: Starting the recording several seconds before the first trial

You might expect that the EEG recording would begin only moments before the first trial. Why waste disk space with all that extra EEG? However, there is a technical reason why you should start the recording several seconds before the first trial. Specifically, this can help you avoid artifacts that filters can produce at the beginning and end of the waveform. If you have some "extra" EEG at the beginning and end of the recording, the filter artifacts occur during these time periods that you don't care about rather than distorting the data during the first and last trials of the recording. See Chapter 7 in Luck (2014) for a more detailed explanation.

Click the **>>** button near the bottom of the plotting window to scroll forward 5 seconds in time. You should now see an event code 121 at approximately 6.6 seconds, an event code 221 at approximately 7.7 seconds, and event code 201 at approximately 8.3 seconds (see Screenshot 2.3). Look up these event codes in Table 2.1. What happened at these three time points?

Event code 121 was a prime word that began approximately 6.6 after the start of the recording. Event code 221 was a target word that was unrelated to the prime word, and it began approximately 1.1 seconds after the prime word. Event code 201 was a correct response, and it occurred approximately 0.6 seconds after the target word. So, this was the first trial in the experiment, and the subject correctly determined that the target was unrelated to the prime with a response time of approximately 600 ms. You can get approximate timing information like this by hovering the mouse over the plotting window. The time corresponding to the location of the mouse pointer is shown at the bottom of the plotting window. Later, we'll discuss how you can determine these times more precisely.

Screenshot 2.3



If you look near the right edge of the plotting window, you'll see the event code for the prime word on the second trial. You can click the > button 4 times to scroll over 4 seconds, and then you'll be able to see the event codes for the prime, the target, and the response on this second trial. Here are some questions you should try to answer:

- Given the experimental design shown in Figure 2.1, what is the shortest amount of time you should ever see between the prime word and the target word? What is the longest time?
- Similarly, what are the longest and shortest times between the target word on one trial and the prime word on the next trial?
- What was the response time for the second trial (approximately)?

If possible, keep the EEG plotting window open for the next exercise.

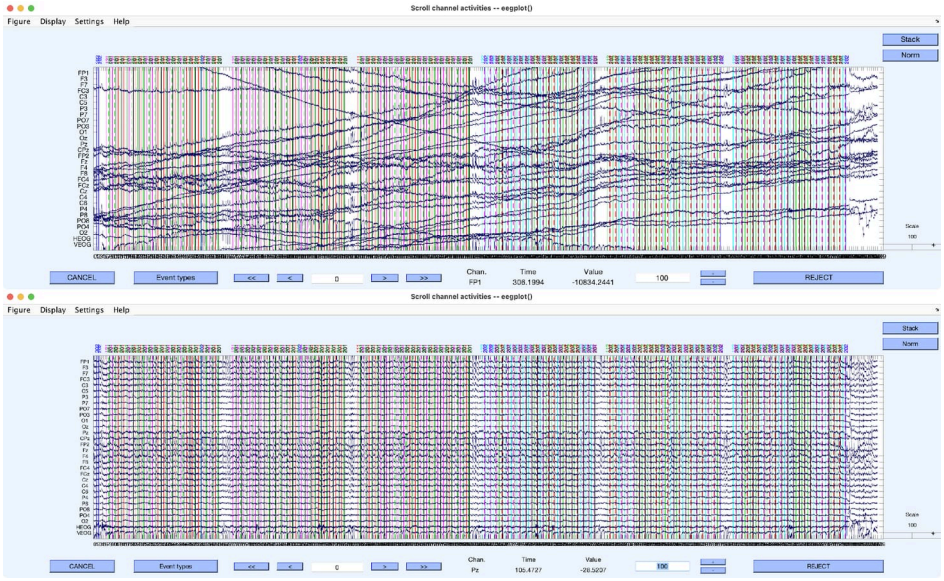
This page titled 2.3: Exercise - Looking at the EEG and the Event Codes is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.4: Exercise- Filtering Out Low-Frequency Drifts from the EEG

The skin is an electrical organ, and it produces slow drifts in voltage that are picked up by our EEG electrodes. These skin potentials can cause the voltage to gradually change by hundreds of microvolts over a period of a few minutes. To get a better look at the skin potentials, go to the EEG plotting window that you opened in the previous exercise (or open the window again), and change the settings as follows:

- Go to time zero by putting a zero in the text box between the < and > buttons in the plot window.
- Set the vertical gain to 100 (using the little text box near the bottom of the plot window).
- Set the time range to 400 seconds using **Settings > Time range to display** in the plot window.
- Stretch the window as wide as you can so that it looks something like the top window in Screenshot 2.4.

Screenshot 2.4



Looks pretty weird, doesn't it? The first thing you should look at is the event codes (the vertical lines). The N400 experiment lasted about 6 minutes, and you're looking at the entire recording, so there are lots of event codes. Notice that there are 6 clusters of event codes, separated by gaps of approximately 7 seconds. The 120 trials in this experiment were divided into 6 blocks of 20 trials each, with a short rest break after each block. I find that participants can maintain their attention better if we use a large number of short blocks, each followed by a brief break, so this experiment was broken up into short blocks that lasted only about a minute each.

Now take a look at the EEG waveforms. You can now see that the voltage is gradually drifting over time. It drifts upward in some channels and downward in others. Most of the channels change by well over 100 μV over the course of this 400-second period. These drifts are mainly caused by electrical potentials in the skin that are picked up by the EEG electrodes (see Chapter 5 in Luck, 2014 for more details).

These drifts can make it difficult to obtain reliable ERP differences across conditions, and it's usually a good idea to filter them out. To accomplish this, we apply a *high-pass filter*, which filters out low frequencies and passes high frequencies. Here, we'll use the filter settings that I recommend for most studies of cognitive and affective processes, which has a half-amplitude cutoff at 0.1 Hz and a slope of 12 dB/octave. If you don't know what these parameters mean, don't worry – we'll cover them in Chapter 4. You can also find a broad conceptual overview of filters in Chapter 7 of Luck (2014) and a more detailed mathematical treatment in Chapter 12 of Luck (2014).

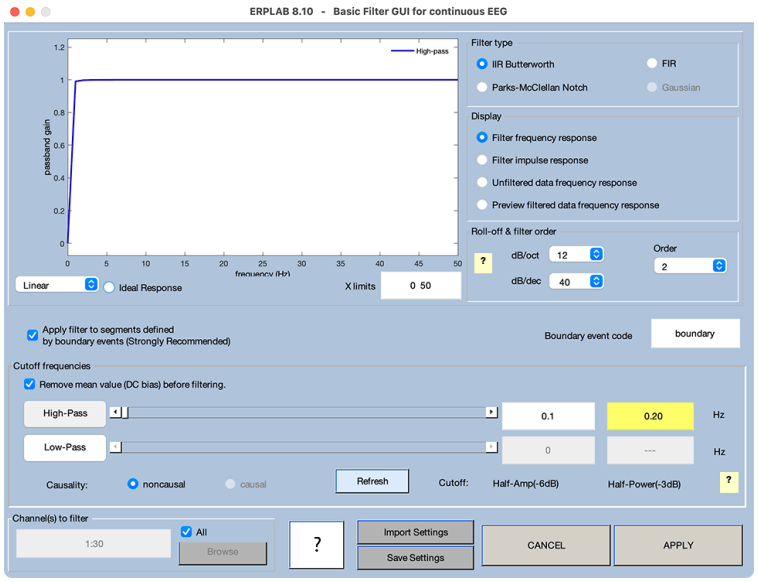
Now let's get rid of these drifts with a high-pass filter. Leave the current plotting window open, and go to **EEGLAB > ERPLAB > Filter & Frequency Tools > Filters for EEG data**. You'll see ERPLAB's filtering GUI, which is big and complicated (because filters have a lot of different options). We'll explain these options in a later chapter, but for this exercise you should just make sure that everything is set to match Screenshot 2.5. Most importantly, make sure the **High-Pass** button is selected with a half-amplitude cutoff of 0.1 Hz, and the **Low-Pass** button is not selected (these button are a slightly darker gray when selected).

Getting an Error Message?

Did you get an error message when you launched the filtering tool? If so, the message probably said that you're missing the Signal Processing Toolbox. This toolbox comes from the makers of Matlab and is required for certain ERPLAB processes, such as filtering. Depending on your institution's Matlab license, it may be free or it may require an extra fee.

You can see what toolboxes are installed by typing ver on the Matlab command line. If you don't have the Signal Processing Toolbox and you don't know how to get it and/or install it, contact your institution's IT support department for assistance.

Screenshot 2.5



Saving the New Dataset

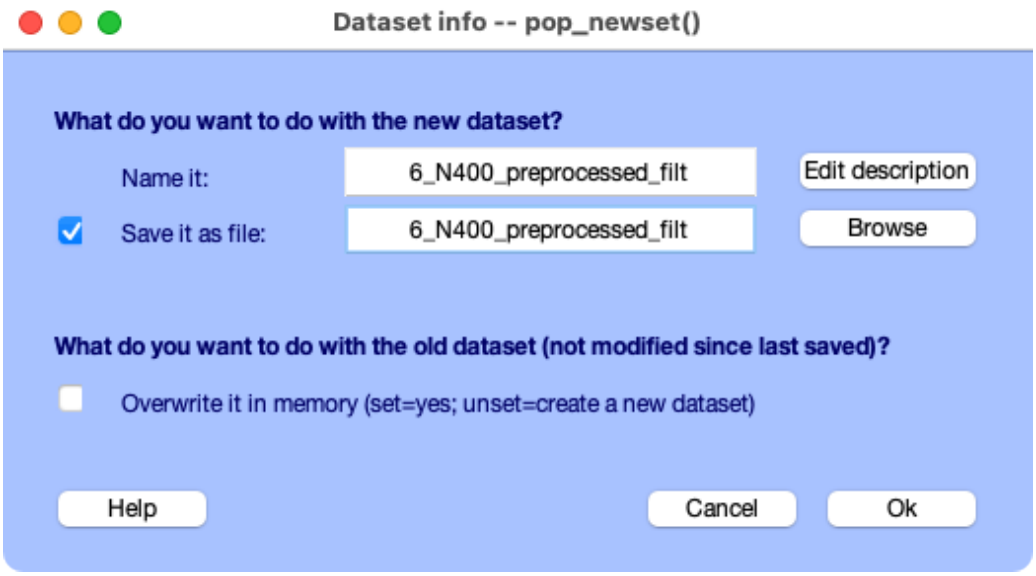
Once all the parameters are set, click the **APPLY** button. You'll then see the window shown in Screenshot 2.6, which asks **What do you want to do with the new dataset?** In EEGLAB and ERPLAB, most operations that modify a dataset will actually create a new dataset. That way, if you make a mistake or change your mind, you can easily go back to the previous dataset. These datasets are stored in memory, where they're listed in the **Datasets** menu, and you can also save them to your hard drive if you want. The top text box in Screenshot 2.6 allows you to specify the name of the dataset (which will be the name shown in the Datasets menu). You can use any name you like, but ERPLAB will give you a suggestion (which is the name of the original dataset with a suffix that indicates the nature of the processing step, such as **_filt** for filtering).

If you want to save the dataset as a file on your hard drive, check the box next to **Save it as a file** and type in the name that will be used for this file. The name of the dataset in memory doesn't have to be the same as the filename, but it can be confusing if the name in memory is different from the filename. I usually just select the name of the dataset from the top text box, copy it into the clipboard, and the paste it into the second text box. Note that if you don't save the dataset as a file now, you can save it later with **EEGLAB > File > Save current dataset as**. You'll need the new filtered dataset for the next exercise, so you should save it as a file if you're not going to do the next exercise right away.

Once you have everything set in this window, click **OK**. You've now created a new dataset with the filtered data. The previous dataset was named **6_N400_preprocessed**, and the new one should be named **6_N400_preprocessed_filt**. If you look in the

Datasets menu in the main EEGLAB GUI, you should see both of these datasets listed, with the new one checked.

Screenshot 2.6



Looking at the Filtered Data

Now that you've saved the dataset, plot the filtered data with **EEGLAB > Plot > channel data (scroll)**. Set it up like the plotting window that shows the unfiltered data (remove the DC offset, set the vertical gain to 100, set the time range to 400 second, and stretch the window to the same width). It should look something like the bottom window in Screenshot 2.4. The slow drifts are now gone, and the data look much more orderly. We've now gotten rid of a major source of artifactual activity from the EEG, which improve our ability to obtain robust, reliable ERP effects.

Now change the time period to display to be 10 seconds instead of 400 seconds for both the unfiltered and filtered data. Because you're still removing the DC offset and the drifts are slow, the filtered and unfiltered data don't look as radically different with this 10-second time scale as they did with the 400-second time scale. But if you look carefully (especially at the PO3 channel), you'll see that there is some drift in the unfiltered data that is absent in the filtered data. You can also see that all the faster deflections in the data are present in both the filtered and unfiltered waveforms. So, the high-pass filter has largely eliminated the slow drifts but has had minimal effect on the other features of the EEG. We'll take a closer look at filters in Chapter 4.

You'll need the filtered dataset for the next exercise. If you're going to do the next exercise right away, just leave EEGLAB open (but you can close the two plotting windows). If you're not going to do it right away, you can save the filtered dataset as a file on your hard drive by selecting **EEGLAB > File > Save current dataset as**.

This page titled 2.4: Exercise- Filtering Out Low-Frequency Drifts from the EEG is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.5: Exercise- Creating an EventList

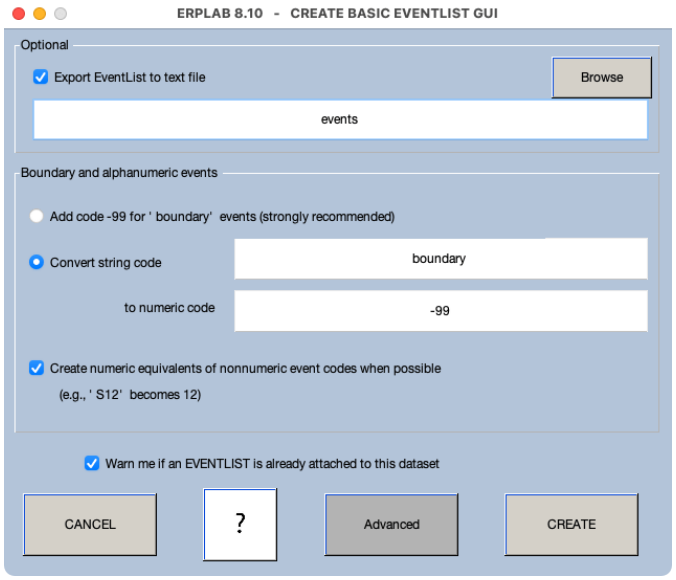
Our next step is to add something called an *EventList* to the EEG dataset. An EventList is a simple, easy-to-access list of all the event codes in a dataset. EEGLAB stores event codes in a Matlab data structure that's a little bit difficult for beginners to access, so we added the EventList structure when we created ERPLAB. After all, *event* is in the name of the ERP technique, so we wanted to make it easy to see and manipulate the event codes.

If you don't still have the filtered dataset from the previous exercise loaded in EEGLAB, go ahead and load it now (6_N400_preprocessed_filt).

In the EEGLAB GUI, select **ERPLAB > EventList > Create EEG EVENTLIST**. In the GUI that appears, check the box labeled **Export EventList to text file** and enter **events** in the text box to indicate the filename. When the EventList is created, it's attached to the EEG dataset, and the **Export EventList to text fil**e option also saves it as a text file so that you can easily see the event codes.

Make sure that the rest of the GUI matches the settings shown in Screenshot 2.7 and then click the **CREATE** button.

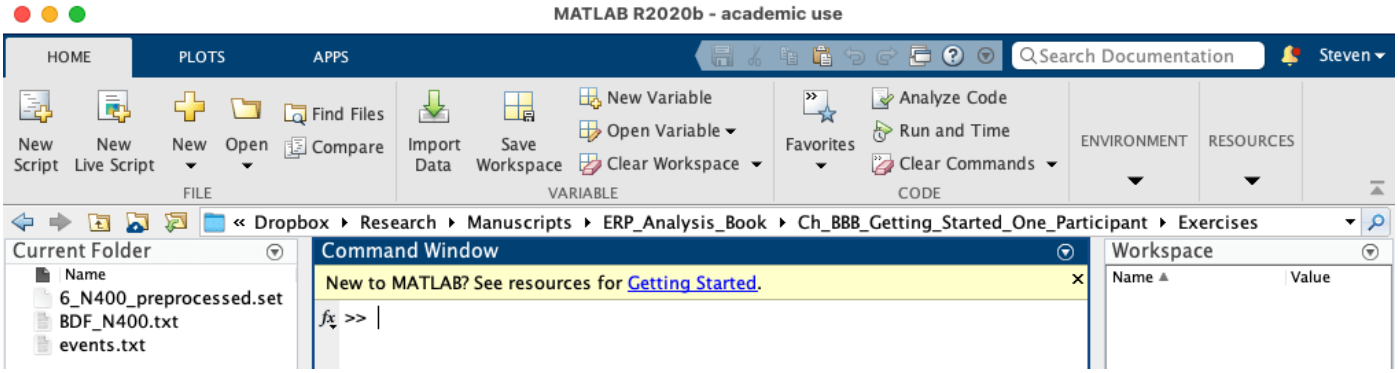
Screenshot 2.7



You'll then see a new window that asks **What do you want to do with the new dataset?** Just accept the default settings by clicking **OK**. You've now created a new copy of the dataset that has the EventList attached. The starting dataset was named **6_N400_preprocessed_filt**, and the new one should be named **6_N400_preprocessed_filt_elist**. You can see the loaded datasets in the **Datasets** menu. You will need the new dataset for the next exercise, so make sure you keep EEGLAB open or save the dataset to your hard drive (using **EEGLAB > File > Save current dataset as**).

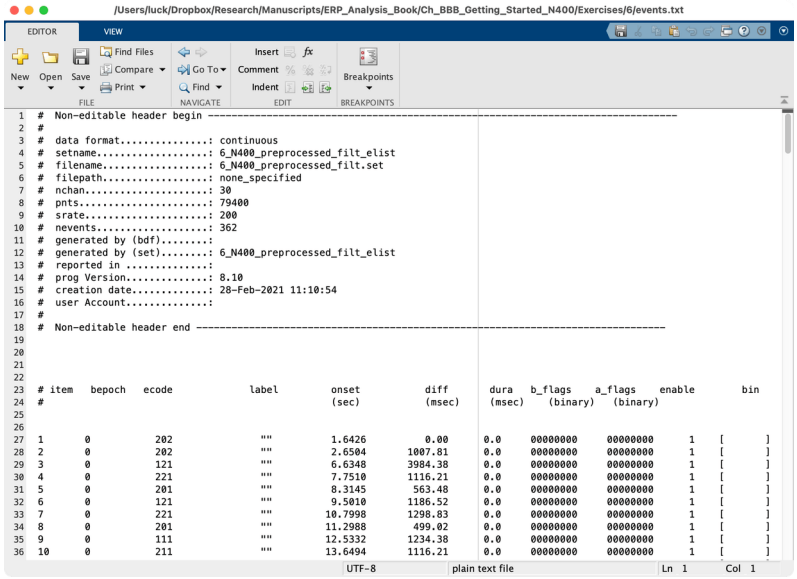
It's a little tricky to look at the EventList directly, but we can look at the copy you saved as a text file, which should be named **events.txt**. You should be able to see this file listed in the **Current folder** section of the main Matlab GUI (see the lower left corner of Screenshot 2.8).

Screenshot 2.8



You can look at this file in virtually any text editor, but the easiest thing to do is to use Matlab's built-in text editor. You can just double-click the filename in the **Current folder** box to open the file. It should look something like Screenshot 2.9.

Screenshot 2.9



At the top of the file, you'll see a bunch of general information about the EEG recording, such as the number of channels and the sampling rate. Then you'll see a list of the event codes, with one per line. Each line contains several columns, many of which we will discuss later. For now, the key columns are **item** (which indicates the ordering of the event codes), **ecode** (which is the actual event code), **onset** (which is the time of the event code relative to the beginning of the recording), and **diff** (which is the amount of time between the current event code and the previous event code, in milliseconds rather than seconds). In principle, event codes can have a duration (listed in the **dura** column), but this feature is not used by ERPLAB and event codes are typically considered to be instantaneous (i.e., a duration of 0).

If you look at the **diff** column, you can now see exactly how much time elapsed between the onset of each target word and the preceding prime word, and you can see the response times. For example, the behavioral response on the first trial (item #5, event code 201) was 563.48 ms after the preceding target word.

You can actually modify the text file and import it back into EEGLAB/ERPLAB to modify the event codes in an EEG dataset (using **ERPLAB > EventList > Import EEG EVENTLIST from text file**). For example, imagine that the subject was confused about the task between event codes 120 and 160. You could set the **enable** column to 0 for these events, which would cause them to be ignored by all ERPLAB processes. Similarly, if you used an eye tracker during the experiment, you could add event codes corresponding to fixations. If you know how to program in Matlab, you can directly modify the EVENTLIST structure rather than modifying the text file and importing it.

Our last step will be to verify that we have the current number of occurrences of each event code. As shown in Table 2.1, there should have been 30 occurrences of each of stimulus event codes. The number of occurrences of each response event code varies across subjects depending on the number of errors. However, there were 120 trials, so there should be 120 total response event codes (plus the two at the beginning during the instruction phase).

To see the number of occurrences, select **EEGLAB > ERPLAB > EventList > Summarize current EEG event codes**. It will plot a table in the Matlab command window. Do we have the correct number of occurrences of each event code type? On what proportion of trials did this participant make a correct response?

Pro tip: Checking the number of event codes for every participant

In my experience, the most common error in ERP experiments is that there is some kind of problem with the event codes. If you are designing and running your own experiments, perhaps the single most important thing you can do to avoid trouble later is to make sure that your event codes are correct. A given experiment might have 1000+ event codes, so it is difficult to check every single one. However, if you set up your experiment properly, you can at least make sure that you have the right number of occurrences of each event code (for the stimuli).

The first step is to know exactly how many occurrences there should be for each code. And I do mean "exactly" rather than "approximately." You can have major problems with your event codes and end up with approximately the right number. Then you can simply run your stimulus presentation program while recording the EEG but without a participant. You will then import your data into EEGLAB, add the EventList, and check the number of occurrences of each event code using **EEGLAB > ERPLAB > EventList > Summarize current EEG event codes**. You should also "spot check" a few dozen randomly selected event codes to make sure that they are correct with respect to the stimulus that was actually presented.

You should also check the number of occurrences of each event code for every participant you run (immediately after the recording session). Lots of things can go wrong with event codes, and this will allow you to catch a problem before you've wasted weeks of time collecting data that turn out to be useless. Checking the number of each event code is easy, and **you absolutely must do it** to avoid problems.

This page titled 2.5: Exercise- Creating an EventList is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.6: Exercise- Assigning Events to Bins with BINLISTER

What is a Bin?

When we create averaged ERPs, we often want to combine trials with different event codes into the same average. For example, the N400 experiment used separate event codes to indicate which of the two word lists was used for a given trial, but we don't really need to make this distinction in our data analysis, so we'll just average together the trials with the different word lists. Also, we won't be making averages for the response event codes, but we want to use those codes so that we exclude trials with incorrect responses from our averages. Thus, we need a sophisticated way of indicating which events should be combined together when we make our averages.

We call this process *assigning events to bins*, and ERPLAB accomplishes this with a routine called *BINLISTER*. A *bin* is a set of averaged ERP waveforms—one for each electrode site—that were created by averaging together a specific set of trials. In our N400 experiment, for example, we will create four bins:

1. Primes followed by a related target
2. Primes followed by an unrelated target
3. Targets preceded by a related prime and followed by a correct response
4. Targets preceded by an unrelated prime and followed by a correct response

We recorded from 30 channels, so each bin will have 30 averaged ERP waveforms, one for each channel. For all 30 channels in a given bin, we averaged together the same set of trials. Together with some header information, the bins for a given participant are stored together in an *ERPset*. Figure 2.2 shows how the ERPset will be structured for the average we will be creating in this chapter.

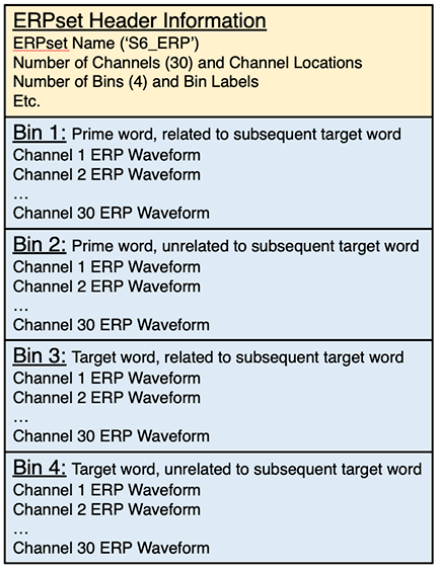


Figure 2.2. Structure of the ERPset that we will create in these exercises.

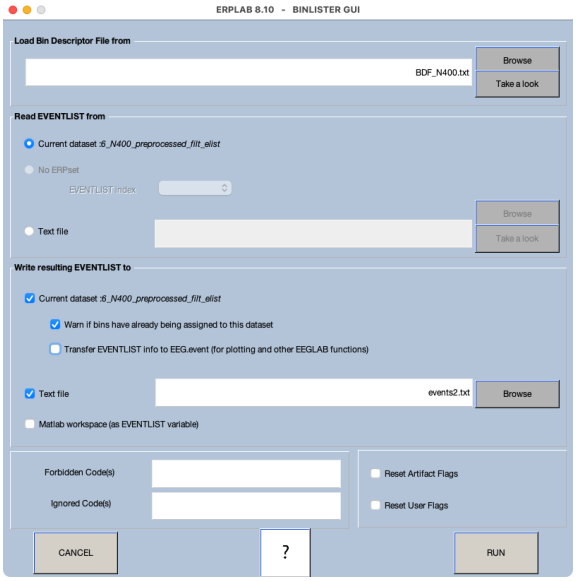
Running BINLISTER

Now that you see how everything fits together, let's actually use BINLISTER to assign the event codes to bins for our example subject. We'll start with the dataset that you created in the previous exercise that has the EventList attached (**6_N400_preprocessed_filt_elist**). If you still have it loaded, make sure that it is active (checked) in the **Datasets** menu (Dataset 1:6_N400_preprocessed_elist). If you don't still have it loaded but you saved it as a file, open the file (using **EEGLAB > File > Load existing dataset**).

Go to **EEGLAB > ERPLAB > Assign bins (BINLISTER)**, which will pop up a GUI window. In the text box at the top (under **Load Bin Descriptor File from**), you will enter **BDF_N400.txt**, which is a text file that tells BINLISTER how to assign events to bins for this particular experiment. We'll take a look at this file later.

BINLISTER adds new information to the EventList, and we want to save a copy of the updated EventList as a text file for this exercise so that we can easily look at it. To accomplish this, check the box next to **Text File** in the section of the GUI labeled **Write resulting EVENTLIST to** (not in the section labeled **Read EVENTLIST from**). Then type **events2.txt** in the text box to the right (see Screenshot 2.10). This will be the name of the text file that will be used to store a copy of the updated EventList. Make sure that everything else is set up like the screenshot and then click **RUN**.

Screenshot 2.10

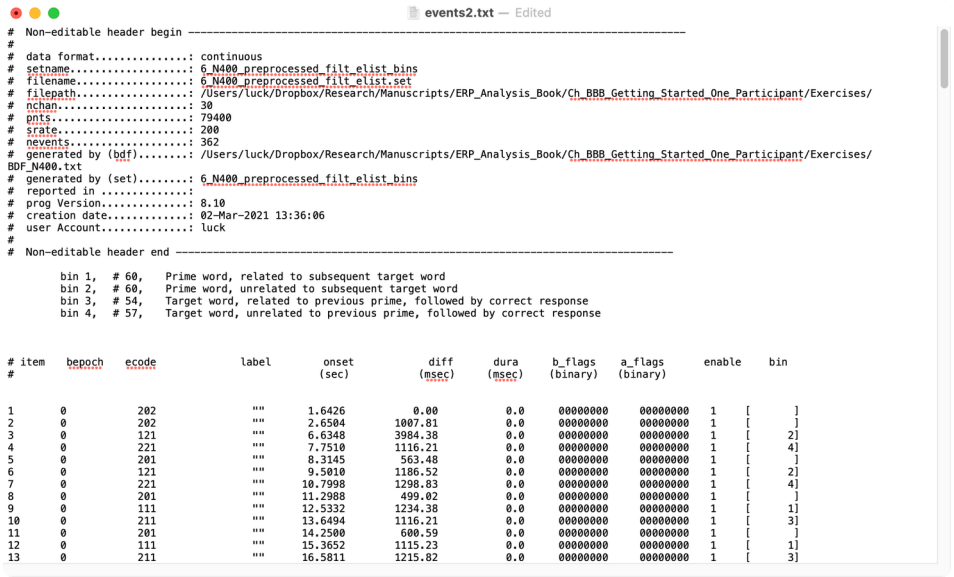


As in the previous exercises, you'll see a new window that asks **What do you want to do with the new dataset?** Just accept the default settings by clicking **OK**. You've now created a new copy of the dataset with the updated EventList attached. The new dataset should be named **6_N400_preprocessed_filt_elist_bins**. You will need the new dataset for the next exercise, so save the dataset to your hard drive if you're not going to do the next exercise right away.

Looking at the Bin Assignments in the EventList

Now open the text file that contains a copy of the updated EventList, which should be named **events2.txt**. You can just double-click it in the Current Folder panel of the main Matlab GUI to open it in the Matlab text editor. It should look something like Screenshot 2.11.

Screenshot 2.11



events2.txt — Edited

```
#  Non-editable header begin ------------------------------------------------------------------------------
#
#  data format.............: continuous
#  setname.................: 6_N400_preprocessed_filt_elist_bins
#  filename................: 6_N400_preprocessed_filt_elist.set
#  filepath................: /Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Ch_BBB_Getting_Started_One_Participant/Exercises/
#  nchan...................: 30
#  pnts....................: 79400
#  srate...................: 200
#  nevents.................: 362
#  generated by (bdf)......: /Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Ch_BBB_Getting_Started_One_Participant/Exercises/
BDF_N400.txt
#  generated by (set)......: 6_N400_preprocessed_filt_elist_bins
#  reported in ............:
#  prog Version............: 8.10
#  creation date...........: 02-Mar-2021 13:36:06
#  user Account............: luck
#
#  Non-editable header end --------------------------------------------------------------------------------

        bin 1,   # 60,    Prime word, related to subsequent target word
        bin 2,   # 60,    Prime word, unrelated to subsequent target word
        bin 3,   # 54,    Target word, related to previous prime, followed by correct response
        bin 4,   # 57,    Target word, unrelated to previous prime, followed by correct response


# item    bepoch    ecode             label        onset          diff      dura     b_flags    a_flags      enable     bin
#                                                   (sec)         (msec)    (msec)   (binary)   (binary)


1       0           202               ""       1.6426          0.00          0.0      00000000     00000000    1     [       ]
2       0           202               ""       2.6504       1007.81          0.0      00000000     00000000    1     [       ]
3       0           121               ""       6.6348       3984.38          0.0      00000000     00000000    1     [      2]
4       0           221               ""       7.7510       1116.21          0.0      00000000     00000000    1     [      4]
5       0           201               ""       8.3145        563.48          0.0      00000000     00000000    1     [       ]
6       0           121               ""       9.5010       1186.52          0.0      00000000     00000000    1     [      2]
7       0           221               ""      10.7998       1298.83          0.0      00000000     00000000    1     [      4]
8       0           201               ""      11.2988        499.02          0.0      00000000     00000000    1     [       ]
9       0           111               ""      12.5332       1234.38          0.0      00000000     00000000    1     [      1]
10      0           211               ""      13.6494       1116.21          0.0      00000000     00000000    1     [      3]
11      0           201               ""      14.2500        600.59          0.0      00000000     00000000    1     [       ]
12      0           111               ""      15.3652       1115.23          0.0      00000000     00000000    1     [      1]
13      0           211               ""      16.5811       1215.82          0.0      00000000     00000000    1     [      3]
```

If you compare it with the original version of the EventList (in **events.txt**), you'll see two main differences. First, the new version has a list of the four bins that we've created, which includes the number of trials and the label for each bin. For example, you can see that there were 60 instances of Bin 1, which was labeled "Prime word, related to subsequent target word". The second change is that there are now numbers in the **bin** column for the individual events (at the far right of the window). These numbers indicate which bin (if any) a given event code has been assigned to. When an event is assigned to a bin, this means that the event is the *time-locking event* for that trial. In other words, the time of the event code will be time zero in our averaged ERP waveforms.

Consider, for example, item #3 on line 33. It has an event code of 121, which means that it is a prime word that will be followed by an unrelated target word, with the word taken from the first word list (see Table 2.1). It was assigned to Bin 2 (see the far right column in **events2.txt**). When we make an averaged ERP waveform for Bin 2, we'll use this event code as time zero in the waveform. Now look at item #4 on the next line. It has an event code of 221, and it's the target word that followed the prime word from item #3. It was assigned to Bin 4. The response event codes (e.g., items 1, 2, and 5) were not assigned to a bin because we are not making any averaged ERP waveforms in which the response is at time zero in this experiment.

How BINLISTER Works

How did BINLISTER know what bins we wanted and which event codes should be assigned to each bin? To accomplish this, BINLISTER used a set of abstract bin descriptions that are stored in the BDF_N400.txt file that you entered at the top of the BINLISTER GUI (see Screenshot 2.10). The *BDF* in the filename stands for *bin descriptor file*, because it contains abstract descriptors for each bin. Here's what is in that file:

Bin 1
Prime word, related to subsequent target word
.{111;112}

Bin 2
Prime word, unrelated to subsequent target word
.{121;122}

Bin 3
Target word, related to previous prime, followed by correct response
.{211;212}{t<200-1500>201}

Bin 4
Target word, unrelated to previous prime, followed by correct response

.{221;222}{t<200-1500>201}

The details of the bin descriptor file syntax will be described in a later section, but some aspects of the syntax are fairly obvious here. Each bin is described with three lines. The first line just gives the bin number. The second number is a text string that describes the contents of the bin (and can be whatever you want). The third line is the actual *bin descriptor*. Each bin descriptor contains one or more event codes inside some curly brackets. For example, Bin 2 contains **121;122** inside the curly brackets, indicating that event codes 121 or 122 should be used as the time-locking event for this bin. Bins 3 and 4 contain a second set of curly brackets that indicate what must follow the time-locking event code. The text **t<200-1500>201** means that the time-locking event code must be followed by an event code 201 with a delay of 200-1500 ms. For example, an event code of 221 will be assigned to Bin 4 if and only if it is followed by a correct response (event code 201) within 200 to 1500 ms. Responses that are faster than 200 ms are probably bogus (because no one can respond that quickly), and responses that are later than 1500 ms probably indicate that the subject was zoning out.

BINLISTER reads in the bin descriptor file and then goes through all of the event codes in the EventList. When it finds an event code that matches the bin descriptor for a given bin, it puts that bin number into the **bin** column for that event code. Note that a given event code can be assigned to more than one bin. For example, we could have one bin for target words that are followed by a correct response and a separate bin for target words irrespective of whether the response was correct.

Now that we've discussed how the four bins were defined in this experiment, you should go through several events in the **events2.txt** file and make sure that you understand why each event was assigned to a given bin. In particular, you should find the target words that were followed by incorrect responses (event code 202) and verify that they were not assigned to a bin.

This page titled 2.6: Exercise- Assigning Events to Bins with BINLISTER is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.7: Exercise- Epoching and Baseline Correction

The next step in processing Subject 6's data is quite simple, and it involves two operations that are done together in a single step. One of these operations is extracting a fixed-length *epoch* for each event that has been assigned to a bin. If you're not quite sure what this means, see Figure A1.3 in Appendix 1). As shown in Figure 2.1, we will be looking at the data from 200 ms prior to stimulus onset through 800 ms after stimulus onset. Consequently, for each trial we need to extract an epoch that starts 200 ms before the time-locking event code and extends until 800 ms after the event code.

The second operation will be *baseline correction*, which is a way of dealing with the DC voltage offsets. As shown in Figure 2.3, we use the average voltage during the prestimulus period as an estimate of the offset, and we shift the whole waveform upward or downward by this amount. Ordinarily, this is done on epoched EEG data, but ERPLAB also allows you to perform correction on averaged ERP waveforms as well.

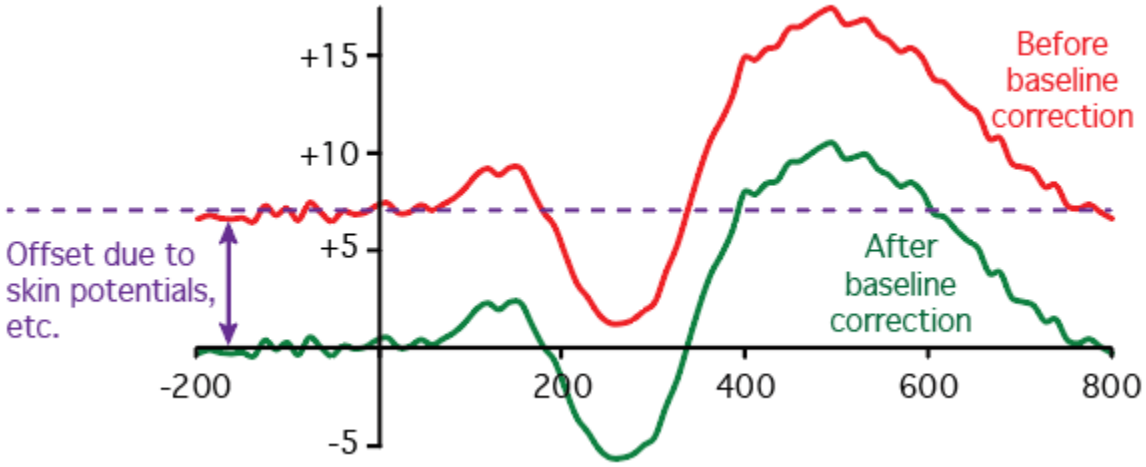
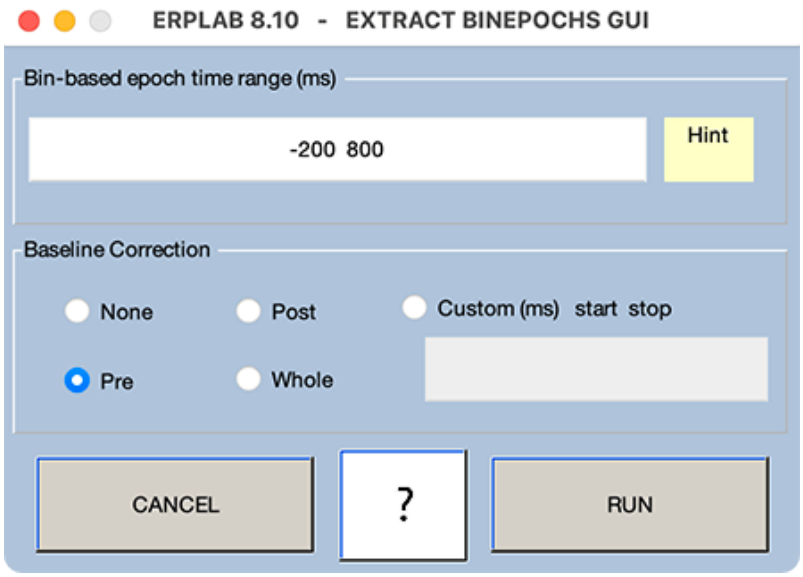


Figure 2.3. Baseline correction procedure. The mean voltage during the prestimulus period (from -200 to 0 ms in this example) is used as an estimate of the voltage offset. This value is simply subtracted from each point in the waveform to create the corrected waveform.

To perform these steps on our example data, make sure that the dataset created during the previous exercise (**6_N400_preprocessed_filt_elist_bins**) is loaded in EEGLAB. Then select **EEGLAB > ERPLAB > Extract bin-based epochs**. In the window that pops up, enter **-200 800** as the time range and select **Pre** for baseline correction (as in Screenshot 2.12). The time window values tell the routine that you want the epochs to start 200 ms before stimulus onset and extend for 800 ms after stimulus onset, for a total epoch length of 1000 ms. The baseline correction parameter tells the routine that you want to use the entire prestimulus interval (-200 to 0 ms) as the baseline period. Once you've set these parameters, click **RUN**. As usual, accept the default settings when asked **What do you want to do with the new dataset?** The new dataset should be named **6_N400_preprocessed_filt_elist_bins_be**. Save it to your hard drive if you're not going to do the next exercise right away.

Screenshot 2.12



Looking at the Epoched Dataset

Now let's see what happened when you ran this routine by selecting **EEGLAB > Plot > Channel data (scroll)**. You should set the vertical scale to 100 in the plotting window, but you don't need to remove the DC offset (because it was removed by the baseline correction procedure). The result should look like the plot in Figure 2.4. By default, the EEG plotting window shows 5 individual epochs, but it looks a lot like one continuous EEG waveform for each channel until you look closely. There is a dashed vertical line between each epochs, 200 ms before the event codes that were used for time locking (because we specified a 200-ms prestimulus period when we epoched the data).

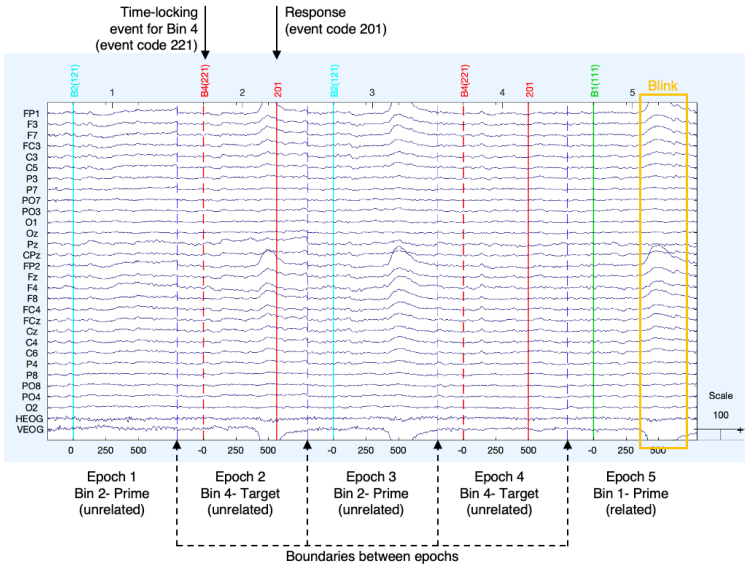


Figure 2.4. Plot of epoched EEG data. It looks like continuous data, but the dashed horizontal lines indicate boundaries between epochs.

For example, look at the event code labeled **B4(221)**, which corresponds to item #4 in the EventList (line 34) shown in Screenshot.2.10. This event code corresponds to a target word, and it was assigned to Bin 4 when you ran BINLISTER. The event code label now contains both the bin number (B4) and the original event code (221). If you look at the labels on the X axis, you'll see that this event is at time zero, because it's the time-locking event for this epoch. When we create our averaged ERPs, all the epochs labeled B1 will be averaged together to form Bin 1 (whether the actual event code was 111 or 112), all the epochs labeled

B2 will be averaged together to form Bin 2 (whether the actual event code was 121 or 122), etc.

The next event code is labeled **201**. This was the response to the target word. We're not using it as the time-locking point for any bins, so it does not have a bin number. However, the information about the event is still present, so you can see the event code in the EEG plotting window.

Scroll through the epochs using the **>>** button at the bottom of the plotting window. Notice that there is usually a response event code following the target words (Bins 3 and 4) but not following the prime words (Bins 1 and 2). Sometimes the response event code is missing; this happens when the response was later than 800 ms and therefore fell outside of the time window of the epochs. You might find it useful to open the **events2.txt** file and compare the contents of that file to what you're seeing in the epoched EEG data.

You'll see a large voltage deflection in many of the epochs, which is large and positive in the Fp1 and Fp2 channels and negative in the VEOG channel. These voltage deflections were produced by eyeblinks—this participant blinked a lot! Most of the blinks occurred in the epochs for the prime words, not the target words. This is fortunate, because we mainly care about the ERPs elicited by the target words in this experiment.

Don't make this mistake!

EEGLAB also has a routine for extracting epochs from the continuous EEG (**EEGLAB > Tools > Extract epochs**), but do not use it!!! The EEGLAB routine knows nothing about bins, so the epochs won't contain the bin information that you'll need for the rest of the ERPLAB steps.

But what if you already have a dataset that has been epoched using the EEGLAB routine, followed by many other processing steps that you don't want to repeat? Or what if you've imported epoched data from another analysis system into EEGLAB and you'd like to process it in ERPLAB?

Fortunately, there is a trick for solving this problem. You can convert the epoched data back into continuous data using **EEGLAB > ERPLAB > Utilities > Convert an epoched dataset into a continuous one**.

This page titled 2.7: Exercise- Epoching and Baseline Correction is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

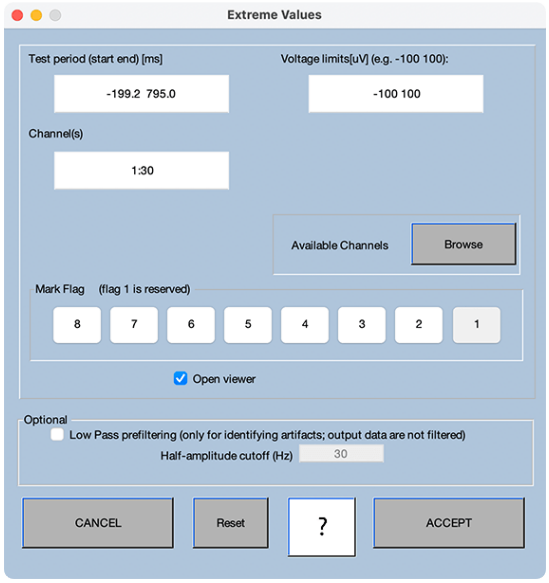
2.8: Exercise- Artifact Detection

The eyeblink artifacts you saw in the previous exercise are huge relative to the EEG, and these artifacts can be a real problem. They can make it difficult to see the actual brain activity, and we need a way to deal with them. In the published version of the ERP CORE experiments, we used a method called *Independent Component Analysis* (ICA) to estimate and remove the artifactual voltages, leaving behind the uncontaminated EEG. ICA is both slow and complicated, so we won't use it in this chapter (but we'll cover it in detail in Chapter 9). Instead, we'll use a cruder approach called *artifact rejection*. In this approach, we use a simple algorithm to identify which epochs are contaminated by eyeblinks. We'll then mark these epochs by setting a *flag* in the EventList. Later, when we make the averaged ERPs, we'll simply leave out the epochs in which this flag has been set.

There is a lot to know about artifacts and artifact rejection, and this topic will be covered in detail in Chapter 8. For now, we'll take a very simple approach in which we'll check every epoch to see if the voltage exceeds ±100 μV in any channel. If the voltage exceeds this range in a given epoch, we'll flag that epoch for rejection. We call this stage of the process *artifact detection* rather than *artifact rejection*, because we're simply marking the epochs with artifacts so that they will be excluded when we get to the averaging step.

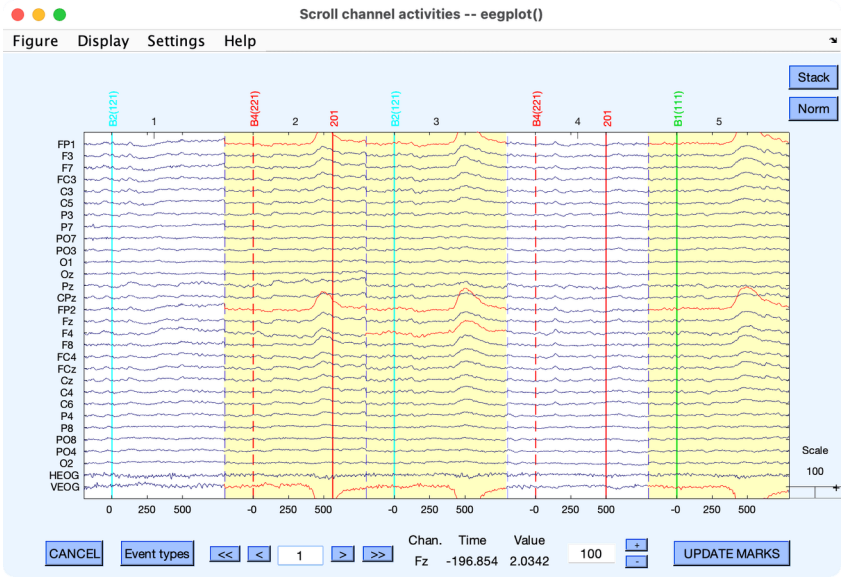
For this exercise, make sure that the dataset created during the previous exercise (**6_N400_preprocessed_filt_elist_bins_be**) is loaded in EEGLAB. Then select **EEGLAB > ERPLAB > Artifact detection in epoched data > Simple voltage threshold**. In the window that pops up, enter **-100 100** as the voltage limits, make sure that only the Flag 1 button is selected (slightly darker gray), and make sure that the other parameters match those shown in Screenshot 2.13).

Screenshot 2.13



When you click the **ACCEPT** button, ERPLAB will test every epoch for artifacts, and then two windows will pop up. One is the usual window asking what you would like to do with the new dataset. The other is the usual EEG plotting window, but now any epoch with an artifact is highlighted with a yellow background (see Screenshot 2.14). The idea is that you'll first use the EEG plotting window to make sure that ERPLAB did an adequate job of detecting artifacts. Then, if everything looks fine, you'll click **OK** in the "save" window to keep the new dataset. Often, however, your visual inspection of the EEG will indicate that some adjustments need to be made to the artifact detection parameters. For example, you might see that some blinks were missed because they were too small. You might then reduce the voltage limits (e.g., setting them to ±90 instead of ±100 in the window shown in Screenshot 2.13) and run the artifact detection procedure again. Chapter 8 describes this process in detail.

Screenshot 2.14



Verifying that the Epochs with Artifacts Have Been Flagged

Go ahead and take a look at the EEG in the plotting window, using a vertical scale of 100 µV as shown in Screenshot 2.14. In the first screen of data, you can see that the second, third, and fifth epochs are marked as containing artifacts. The individual channels that exceeded our ±100 µV limits are drawn in red, and the epochs containing an artifact in one or more channels have a yellow background. We will exclude an entire epoch from our averages even if it contains an artifact in only one channel. The reason is that the artifact may not be easily visible in all channels in the raw EEG, but it might still be large enough to distort our data. Also, it would be a little weird if our averaged ERPs were based on different trials for different channels.

Now scroll through all of the data in the EEG plotting window and check to see if there are any epochs 1) that contain large artifacts that are not marked for rejection or 2) that do not contain large artifacts and are nonetheless marked for rejection.

When I go through the data, it looks pretty good, but I did find a few epochs that contain smallish eyeblink artifacts but were not marked for rejection (e.g., epochs 9, 154, and 201). They all contain voltage deflections in the VEOG channel that have the same basic shape as the eyeblinks that were flagged for rejection, along with an opposite-polarity deflect in the Fp1 and Fp2 electrodes. As will be discussed in Chapter 8, this pattern is characteristic of eyeblinks. So, the very simple approach that we've used to detect eyeblinks in this exercise is pretty good but not perfect. We'll talk about better approaches in Chapter 8.

This participant blinked a lot, more than is typical, so a lot of trials will be excluded from our averaged ERPs. This will in turn reduce the signal-to-noise ratio of the averaged ERPs, making it more difficult to precisely quantify the N400 amplitude. To see exactly how many trials were marked for rejection, go to the main Matlab GUI and look in the command window. You'll see that the artifact detection routine produced a table showing the number and percentage of accepted and rejected trials for each bin, as well as the total across bins. (Don't worry about the columns labeled F2, F3, etc., which will be discussed in Chapter 8). You'll see that 38.5% of trials were rejected across bins. Ordinarily, my lab "throws out" any participant for whom more than 25% of trials were rejected (see Chapter 6 in Luck, 2014). However, this experiment was designed to be analyzed using artifact correction instead of artifact rejection (see Chapter 9), so we didn't actually need to exclude this participant. By the way, you can print this table of values at a later time if you'd like by selecting **EEGLAB > ERPLAB > Summarize artifact detection**.

Now that you've looked through the epochs and the number of trials with artifacts, you can go to the window that asks **What do you want to do with the new dataset?** and click **OK** to save this dataset as **6_N400_preprocessed_filt_elist_bins_be_ar**. Save the dataset to your hard drive if you're not going to do the next exercise right away.

This page titled 2.8: Exercise- Artifact Detection is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

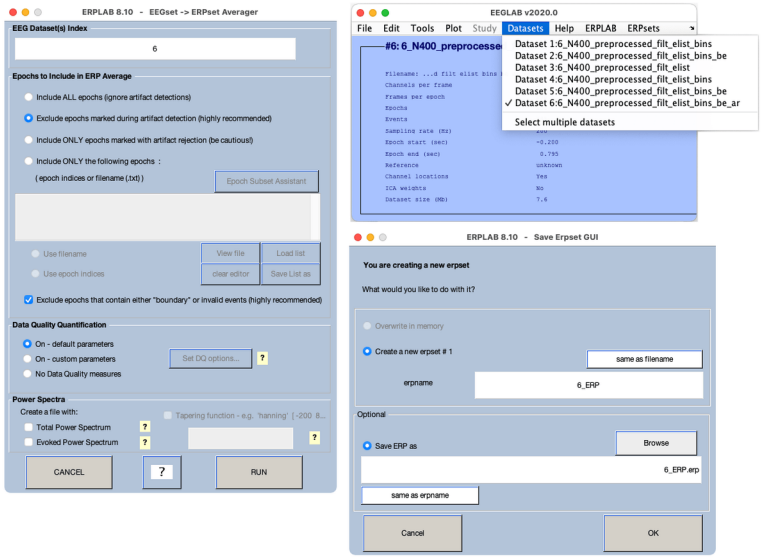
2.9: Exercise- Averaged ERPs

We're now finally ready to create the averaged ERPs for this participant. First make sure that the dataset from the previous exercise is loaded (**6_N400_preprocessed_filt_elist_bins_be_ar**). Then select **EEGLAB > ERPLAB > Compute averaged ERPs**. A window will pop up that allows you to specify a variety of options for the averaging process, which is shown in Screenshot 2.15.

Averaging Options

The first and most essential value is an index for the dataset that contains the epoched EEG to be averaged. You can see the list of datasets that are currently loaded into EEGLAB in the Datasets menu (see the upper left portion of Screenshot 2.15). The currently active dataset is indicated with a check mark (and is ordinarily the most recently loaded or created dataset), and by default this is the dataset that will be used for averaging.

Screenshot 2.15



The datasets that are loaded depend on what you've done since launching EEGLAB. And your **Datasets** menu will look different from mine, because mine has a bunch of datasets that were created as I was trying out different things while creating the exercises in this chapter. We want to average the epoched dataset in which the epochs with artifacts have been marked (**6_N400_preprocessed_filt_elist_bins_be_ar**). For me, this is **Dataset 6** in the **Datasets** menu, but it is probably a different dataset in your **Datasets** menu. In the GUI for the averaging routine, make sure that the right dataset number is listed in the text field at the top (**EEG Dataset(s) Index**).

Dealing with multiple datasets for a given participant

Imagine that you are running an EEG recording session and a fire alarm starts ringing halfway through the session. You would need to stop recording, disconnect the participant, and head to safety. But after 5 minutes, the alarm stops and you're allowed back into the lab. You reconnect the participant and resume the recording, but with the data in a new file. How do you average together the trials from the two files?

Another possibility is that you run an experiment with 10 different trial blocks, and you create a different EEG recording file for each block. Again, you need to be able to average the data across multiple blocks.

> One way to accomplish this is to combine the datasets into a single dataset with **EEGLAB > Edit > Append datasets**. But if you want to keep the datasets separate (e.g., so you don't overload your computer's memory with huge datasets), you can actually specify more than one dataset in the text box at the top of the averaging GUI. The datasets that you specify will be treated like a single large dataset during the averaging process.

The next section of the averaging GUI controls how artifacts are treated during the averaging process. The default option, which you should make sure is selected, is **Exclude epochs marked during artifact detection**. We also provide options for including all of the epochs (whether or not they are marked for artifacts) or for including only the marked epochs; these options are used only rarely.

You should also make sure that there is a check mark in the box labeled **Exclude epochs that contain either "boundary" or invalid events**. A boundary is a special event code that indicates a discontinuity in the EEG. For example, imagine that the data collection was temporarily paused 300 ms after the onset of an event because the participant asked for a short break, and then it was restarted again a minute later. A boundary code would be inserted into the data at the time of the pause. We wouldn't want to include that trial, and so we exclude trials with boundary events. Another possibility is that the **enable** flag in the EventList was set to zero for the time-locking event (e.g., because you realized that the participant had fallen asleep during the last part of the experiment and you therefore disabled the events during that period after the session was over). These trials should also be excluded.

ERPLAB can calculate some measures of data quality during the averaging process, and the **Data Quality** section of the averaging GUI allows you to control this process. Just leave it set to **On – default parameters**. ERPLAB can also compute power spectra during the averaging process, but you should leave the **Power Spectra** options off for the present exercise.

Creating and Saving an ERPset

Once you have everything set properly in the averaging GUI, you can click **RUN** to create the averaged ERPs. You'll then see a window that allows you to save the averaged ERPs, which are stored in an ERPset (as illustrated earlier in Figure 2.2). You should name the ERPset **6_ERP** (because this is the ERP data from Participant 6). That's the name that will show up in the ERPsets menu. You can also save the ERPset on your hard drive as a file. To do this, activate the **Save ERP as** button. The name of the ERPset in memory does not need to be the same as the name of the file, but it's usually a good idea to use the same name for both. You can accomplish this by clicking the **same as erpname** button, which will put **6_ERP** into the text box for the filename. Once you have everything set, you can click the **OK** button. If you look in the ERPsets menu in the main EEGLAB GUI, you'll see that the new ERPset is now listed as **ERPset 1: 6_ERP**.

If you look in the Matlab command window, you'll see that the averaging routine printed a bunch of information when it finished. Here's the last part of what it printed:

```
TOTAL:
The dataset 6_N400_preprocessed_filt_elist_bins_be_ar has a 38.5 % of discarded trialsSummary per bin:
Bin 1 was created with a 41.7 % of rejected trials
Bin 1 was created with a 0.0 % of invalid trials
Bin 2 was created with a 55.0 % of rejected trials
Bin 2 was created with a 0.0 % of invalid trials
Bin 3 was created with a 29.6 % of rejected trials
Bin 3 was created with a 0.0 % of invalid trials
Bin 4 was created with a 26.3 % of rejected trials
Bin 4 was created with a 0.0 % of invalid trials
-------------------------------------------------
Data Quality measure of aSME
Median value of 1.0008 at elec FP2, and time-window 0:100ms, on bin 1, Prime word, related to subsequent target word
Min value of 0.16593 at elec Oz, and time-window -200:100ms, on bin 1, Prime word, related to subsequent target word
Max value of 3.5935 at elec F4, and time-window 600:700ms, on bin 3, Target word, related to previous prime, followed by correct response
```

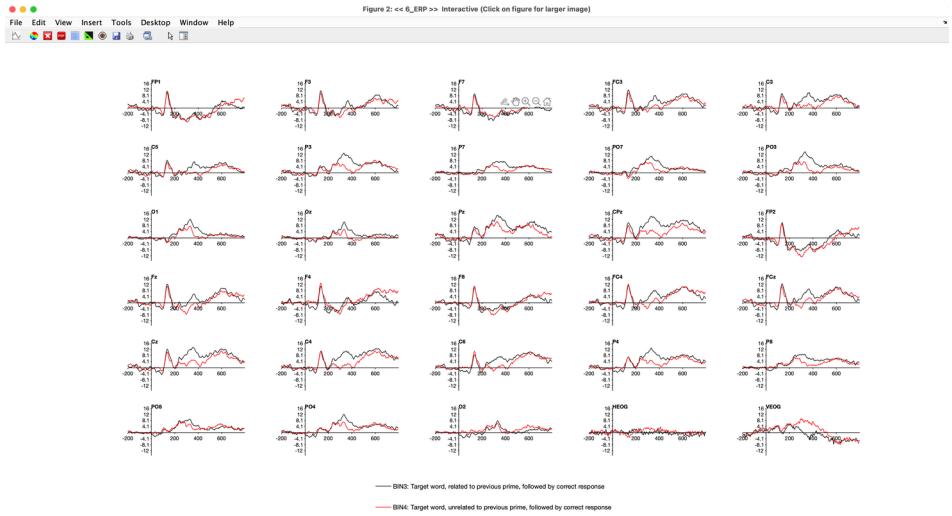
You should always look at this information in the command window. First, it allows you to verify that the expected number of trials were rejected because of artifacts. Second, it allows you to see if there were any invalid trials (e.g., trials on which the **enable** flag in the EventList was set so zero). Third, it provides a summary of some data quality metrics (which will be described in a later

section).

Viewing the Averaged ERP Waveforms

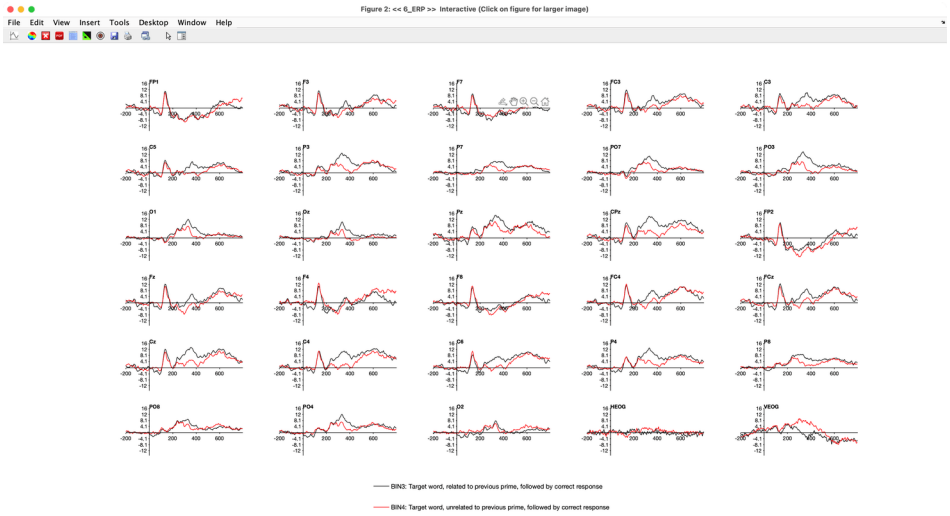
Now let's plot the averaged ERP waveforms. Select **EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms**, and a big complicated window will pop up with lots of options for controlling the plotting. Click the **RESET** button near the bottom to reset it to the default parameters. To keep things simple, we'll start by looking at the ERPs to the prime words, which are stored in Bin 3 (for targets that were related to the preceding prime) and Bin 4 (for targets that were unrelated to the preceding prime). To specify that we want to plot just Bins 3 and 4, uncheck the button in the top left of the GUI labeled **all bins** and type **3 4** into the text box underneath (as in Screenshot 2.16). You could instead click the **Browse** button to see a list of the bins.

Screenshot 2.16



Now click **PLOT** to see the waveforms. It should look something like Screenshot 2.17. If you're not used to looking at ERP waveforms, the plot may look like a chimpanzee threw a plate of spaghetti on the wall, but once you gain some expertise it will be easy for you to comprehend what you're seeing. I always recommend starting by looking at the prestimulus baseline period. Note that it's relatively flat compared to the poststimulus period. In theory, the prestimulus period should contain only random noise in the single-trial EEG epochs, and if we average together enough epochs, the noise will "average out" to zero. It never actually reaches a perfectly flat line with a finite number of trials. (This would be a good time to remind yourself of how many trials were in each bin.) Also, there may be a tilt in the waveform during the prestimulus period as a result of overlapping ERP activity from the previous stimulus or anticipatory activity.

Screenshot 2.17



In this particular example, the prestimulus baseline period looks quite good. The residual noise after averaging is relatively small compared to the ERPs in the poststimulus period, and there is no obvious tilt. I wish they always looked this good. I chose Participant 6 for this chapter because the data were very nice. In later exercises, you'll see participants with much noisier data.

Now take a look at the poststimulus period. Start by looking at the CPz channel, which is the channel where the N400 is typically largest and is the channel shown in the grand average in Figure 2.1. (If you want a zoomed-in view, you can single-click the channel label to get a new window that shows only this channel.) You can see that there is a large broad positive voltage for the targets that were related to the preceding prime word, extending from approximately 200 ms until the end of the epoch at 800 ms. For the targets that were unrelated to the preceding prime, the voltage is more negative (less positive) during much of this period. The existing evidence indicates that both the related and unrelated target words produce essentially the same activity, including the broad positive voltage, but that the unrelated words also produce an additional negative voltage (the N400) that is added onto the broad positive voltage. So, even though the voltage remains above zero for the unrelated targets, the difference between the related and unrelated targets appears to be largely the result of the addition of an N400 component for the unrelated targets. This N400 activity appears to reflect the additional work the brain must do to process a word that is not related to the concepts that were already active when the word was presented (for a review of the N400 and other language-related ERP components, see Swaab et al., 2012).

Now take a look at the other channels. You'll see that waveforms in channels near CPz (e.g., Cz, C3, C4, Pz, P3, C4, P4) look quite similar to the CPz waveforms, but the waveforms in more distant channels (e.g., Fp1, Fp2, Oz) look quite different. This is because the resistance of the skull is high (especially relative to the underlying cortex and overlying scalp), which causes the voltages to spread widely before they reach the electrodes.

If this is the first time you've ever created averaged ERPs, I hope you have a real sense of accomplishment and awe. You are looking at voltages created by neurons in the brain of a living human being who looked at pairs of words and decided whether the second word in each pair (the target) was related or unrelated to the first word (the prime). A tremendous amount of brain power and knowledge was needed for this participant to take the light emitted by the pixels on the computer screen, organize this light into letters and words, recognize the words, access their meanings, and compare them. And you are looking at the actual voltages created by the neurons as the brain carried out these processes. That is, the ERPs are the extracellular voltages produced by cortical pyramidal neurons as a result of neurotransmission, which (amazingly!) are able to pass through the brain, meninges, skull, and scalp to our recording electrodes. I've been recording and analyzing ERPs for almost 40 years, and this still gives me chills!!!

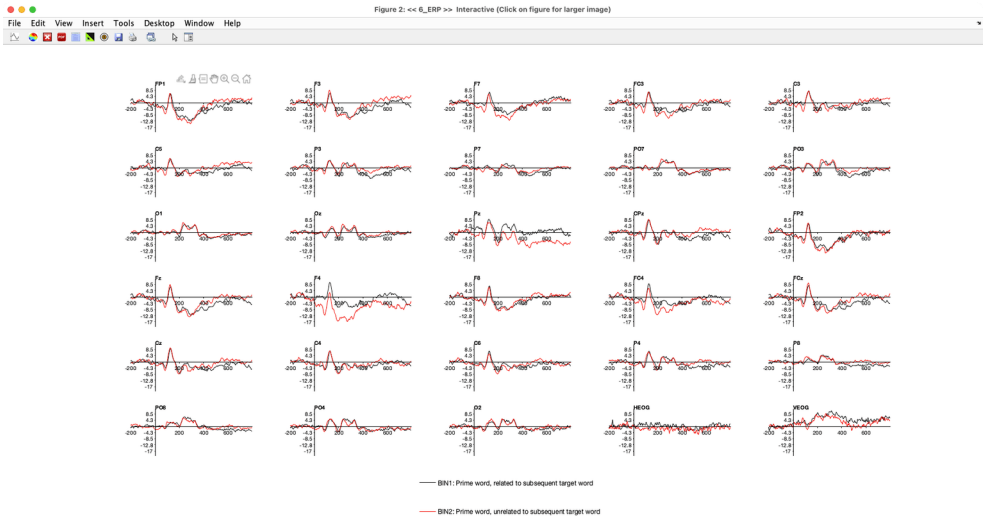
Viewing the Prime Words

Now let's look at the averaged ERP waveforms for the prime words. Select **EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms**,

and set it up just as you did to look at the target words except specify **1 2** in the **Bins to plot** section. Bin 1 is the ERP for prime words that are followed by a related target word, and Bin 2 is the ERP for prime words that are followed by an unrelated target word. Unless the participant had ESP, these ERPs should be equivalent except for random noise. How could the brain response to a given word vary according to the nature of a word that presented later in time?

Click the **PLOT** button to see the waveforms for Bins 1 and 2. If you look at the CPz channel, where the N400 effect was largest for the target words, you'll see that the waveforms for the two prime bins are pretty similar until about 400 ms poststimulus (see Screenshot 2.18). Then, the waveform becomes slightly more negative for primes followed by related words than for primes followed by unrelated words. Logically, this small difference must just be random noise in the data.

Screenshot 2.18



Now look at the F4 channel. You should see a large difference between Bins 1 and 2, beginning right around the onset of the prime word (0 ms). This absolutely must be noise, because it takes at least 50 ms for visual information to reach the cortex and generate an ERP. A similar but somewhat smaller early effect can be seen at the Pz electrode site.

Noise is an inevitable fact of life in ERP studies. After all, we're trying to measure voltages produced by tiny neurons in the cerebral cortex with electrodes placed on the skin, and there is a big thick skull between the neurons and the skin. Also, the brain signals are only a few millionths of a volt once they reach the scalp, where they're mixed with other signals such as skin potentials, muscle activity, and induced voltages from computers and other electrical devices in the recording environment. When you read journal articles, you don't usually get to see the single-participant data. Instead, you see grand averages, which have much less noise. Even without noise, the ERP waveforms from different people often look quite different from each other (probably due to individual differences in how the cortex is folded up in the brain). So, don't be surprised when the single-participant data you see in this book or in your own studies look quite different from the grand average waveforms in published papers.

This page titled 2.9: Exercise- Averaged ERPs is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.10: Exercise- Data Quality

Now that you've seen how noise can distort averaged ERP waveforms, let's look at how we can quantify the noise level (or, to put it in more positive terms, the *data quality*). When you averaged the data, recall that there was a **Data Quality Quantification** section in the averaging GUI, and you left it set to **On - default parameters** (see Screenshot 2.15). Here, we'll look at the data quality metrics that were created by default.

Recall that the averaging routine printed some text to the command window, ending with the following:

Data Quality measure of aSME
Median value of 1.0008 at elec FP2, and time-window 0:100ms, on bin 1, Prime word, related to subsequent target word
Min value of 0.16593 at elec Oz, and time-window -200:100ms, on bin 1, Prime word, related to subsequent target word
Max value of 3.5935 at elec F4, and time-window 600:700ms, on bin 3, Target word, related to previous prime, followed by correct response

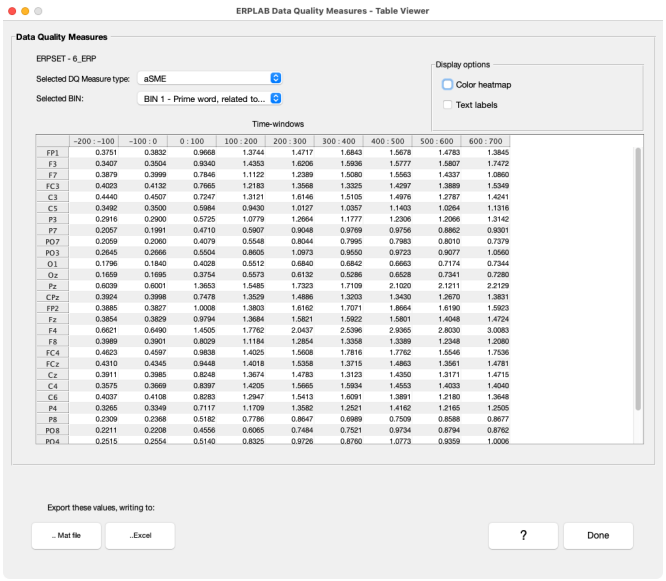
This is a summary of a large set of data quality measures that are computed by default when you average, using a metric of data quality called the *standardized measurement error* (SME; the specific version of SME used here is the *analytic SME* or aSME). You can read a full description of the SME metric in Luck et al. (2021), and you watch a short video overview here along with a corresponding infographic here. Briefly, the SME values provided by default give you the *standard error of measurement* for the mean voltage within a set of time ranges (e.g., 0-100 ms, 100-200 ms, etc.). The larger the SME value, the less precisely the voltage in that part of the waveform is likely to reflect the true voltage for that participant (i.e., the voltage that would be obtained if there were no noise or we could average over an infinite number of epochs). The averaging routine computes a separate aSME value for each combination of bin, channel, and time period.

The summary printed in the command window shows the best (minimum), worst (maximum), and median aSME values. You can see that the best aSME value was 0.16593 μV at the Oz electrode site in Bin 1 during the first portion of the baseline period (-200 to -100 ms). The worst aSME value was 3.5935 μV at the F4 electrode site in Bin 3 near the end of the epoch (from 600 to 700 ms). The median was 1.0008 μV, so the worst value was more than three times as large as the "typical" value.

Do you remember that we saw big differences between primes that were followed by related versus unrelated words starting at time zero in the F4 channel, which logically must have been noise? It's no accident that the aSME value was largest for the same channel.

Let's take a look at all the aSME values that were computed when the averaged ERPs were created. Make sure that the ERPset from the previous exercise is still loaded (by checking the **ERPsets** menu). Now select **EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table**. You'll see a table of aSME values like that shown in Screenshot 2.19. Each row is a different channel and each column is a different 100-ms time range. You can select which bin is shown with a popup menu near the top of the window. We're currently looking at Bin 1.

Screenshot 2.19



A huge amount of information is shown in this table. To help you find the cases with the worst data quality (the largest values), select the **Color heatmap** option. Now the cells of the table are colored according to the magnitude of the aSME values.

Notice that the values tend to get larger during later time windows. This is because the baseline correction procedure brings all the single-trial EEG epochs toward 0 μV during the prestimulus period, and noise will cause the voltage to drift away from this baseline over time. The more random variation there is over trials, the harder it is to precisely measure the amplitude, so this drift causes larger aSME values.

You should also notice that the values tend to be largest in the F4 channel and second largest in the Pz channel. These are the same channels where we saw the greatest noise-related differences between Bins 1 and 2 in the averaged ERP waveforms (Screenshot 2.18). Look at the other 3 bins as well (using the **Selected BIN** popup menu). These channels are also noisy in those bins, indicating that these channels were just generally noisy. If you go back and look at the EEG epochs, it's not obvious that these channels are noisier than the others. That's the value of having an quantitative metric of data quality: It's possible to objectively determine which channels (or which participants) have unusually noisy data. In later chapters, we'll discuss what to do when a channel or participant is particularly noisy.

The data quality metrics are computed by default when you create averaged ERPs. Often, however, you want to assess the data quality prior to averaging (e.g., to determine whether a specific artifact rejection procedure will increase or decrease your data quality). To accomplish this, you select the appropriate EEG dataset (which must be epoched) and then select **EEGLAB > ERPLAB > Compute data quality metrics (without averaging)**.

This page titled 2.10: Exercise- Data Quality is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.11: Review of Processing Steps

You did a lot of exercises in this chapter, and they're probably beginning to blur together in your mind. So let's review the steps:

- Load the EEG dataset
  - EEGLAB > File > Load existing dataset
- Look at the dataset to verify that the EEG and event codes looked okay
  - EEGLAB > Plot > Channel data (scroll)
- Filter out the low-frequency drift in the EEG
  - EEGLAB > ERPLAB > Filter & Frequency Tools > Filters for EEG data
- Add an EventList to the dataset
  - EEGLAB > ERPLAB > EventList > Create EEG EVENTLIST
- Verify that we had the correct number of occurrences of each event code
  - EEGLAB > ERPLAB > EventList > Summarize current EEG event codes
- Use BINLISTER to assign events to bins
  - EEGLAB > ERPLAB > Assign bins (BINLISTER)
- Extract fixed-length epochs from the continuous EEG and perform baseline correction
  - EEGLAB > ERPLAB > Extract bin-based epochs
- Test each epoch for artifacts using a simple voltage threshold, verify that the appropriate epochs were marked, and see how many artifacts were detected
  - EEGLAB > ERPLAB > Artifact detection in epoched data > Simple voltage threshold
- Average together the epochs without artifacts, separately for each bin
  - EEGLAB > ERPLAB > Compute averaged ERPs
- Plot the averaged ERP waveforms
  - EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms
- Examine the data quality of the averaged ERPs
  - EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table

Although this may seem like a lot, it's the minimal set of steps for going from raw EEG to an averaged ERP in ERPLAB for most studies. There are many additional steps that are often included in what we call the *EEG preprocessing pipeline*, and there are also many steps after averaging. Indeed, several preprocessing steps had already been applied to the EEG data used in this chapter's exercises. That's why I decided to write a whole book! I hope this chapter has given you a clear overview of the basics of EEG preprocessing. The next chapter will show the basic steps that follow the creation of the averaged ERP waveforms.

This page titled 2.11: Review of Processing Steps is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.12: A Simple Matlab Script

How long did it take you to run all of the steps in this chapter? Quite a while, I suspect. Now imagine that there are twice as many steps (as will typically be the case), and that one of them takes 20 minutes to run (which is typical for the artifact correction process that we typically use). And imagine that you've collected data from 40 participants and need to repeat the sequence of steps for each of them. Now imagine that a reviewer insists that you change one of the steps, so now you need to run them all again. Does this sound like fun? I didn't think so. Also, when you do each step "by hand" using the EEGLAB/ERPLAB GUI, you're likely to make occasional errors.

To make your life easier and to reduce errors, you can automate the EEGLAB and ERPLAB processing steps using a Matlab *script*. A script is basically a set of commands that you could type in the Matlab command window but that you inside save as a text file with a **.m** extension (sometimes called a *.m file*). To make things more efficient, a script can include more sophisticated programming elements like loops, if/then statements, and variables. If you already know how to program in a typical programming language, you'll find these aspects of Matlab scripting to be pretty straightforward. If you don't already have much programming background, this book will teach you the basics.

Fortunately, EEGLAB and ERPLAB have a *history* feature that makes scripting a lot easier. Every time you do something in the EEGLAB/ERPLAB GUI, the equivalent script command is saved in the history. For example, when you ran the artifact detection step, the following was saved in EEGLAB's history:

pop_artextval(EEG , 'Channel',1:30, 'Flag',1, 'Threshold', [-100 100], 'Twindow', [-200 795]);

This makes it really easy to see how to create a line of script that corresponds to a step you carried out in the GUI. In fact, when I write EEGLAB/ERPLAB scripts, I often do the steps first in the GUI, look at the history, and then copy the relevant commands from the history into my .m file. In a later chapter, we'll discuss how to access the history and build your own scripts. For now, let's run an example script just so that you can see the basic idea.

In the folder for this chapter, there is a file named **preprocess_EEG.m**. You should be able to see it in the **Current Folder** pane of the Matlab GUI. Double-click the filename to open it in the Matlab text editor. This text editor recognizes certain aspects of the Matlab scripting syntax and uses different coloring for different parts of the file (even though the actual file is just plain text without any colors specified, as you can verify by opening it in some other text editor). Lines that begin with a percent symbol are comments, and by default they're colored in green. I added some comments at the top of the script that explain how to run it, and I also put a comment above each line of code to explain what that line does. For example, near the bottom you will see the **pop_artextval** command for doing artifact detection, and there is a comment above that line that explains what it does.

This script does all the basic EEG preprocessing steps that we covered in this chapter. Read through the comments in the script file to see the steps. The last step prior to averaging was artifact detection, and the dataset created by this step is saved to a file. The filename has **_from_script** appended to it.

To run the script, first quit EEGLAB to make sure that we don't have any conflicts. Then, run the script. There are several ways to do this, including a Run button in the tool bar at the top of the Matlab text editor window for the preprocess_EEG.m file. With some luck, you'll see a bunch of text appear in the Matlab command window, with information about each step that is running. When everything is done, you'll see a prompt in the command window.

If you see an error message or have some other kind of problem, then you're about to learn some something useful! Start by reading the error message and trying to figure out how to solve the problem. If that doesn't work, see if there is someone around who has experience running Matlab scripts and can give you some help. And if you can't find someone to help, it's time to consult the troubleshooting tips in Appendix A2. You might also want to read the first half of Chapter 11 to learn some of the basic concepts of EEGLAB/ERPLAB scripting.

Once you've gotten the script to run, you should see that it has created a new dataset file named **6_N400_preprocessed_filt_elist_bins_be_ar_from_script** in the same folder with all the other files from this chapter. Launch EEGLAB and open this file using **EEGLAB > File > Load existing dataset**. Take a look at the EEG using **EEGLAB > Plot > Channel data (scroll)**. It should look just like the EEG data that you saw after going through all the preprocessing steps with the GUI.

I hope you can now see how much faster it is to process a participant's data with a script than by doing it with the GUI. However, this doesn't mean that you won't need the GUI once you've learned EEGLAB/ERPLAB scripting. You should use the GUI the first time you go through a participant's data so that you can check for problems and errors. You'll also need to use the GUI to determine the appropriate artifact detection parameters (and artifact correction parameters once you learn about that). But once you've processed the participant's data manually and determined the right parameters, you can run a script using those parameters to reprocess the participant's data. That way you can avoid any errors you might make when you process the data manually. And when Reviewer 2 asks you to change your preprocessing pipeline, you'll be able to reprocess all the data quite easily and won't be as grumpy about making the change.

This page titled 2.12: A Simple Matlab Script is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

2.13: Key Takeaways and References

Key Takeaways

- Several *EEG preprocessing* steps are necessary prior to creating the averaged ERPs for a given participant (and this chapter did not even cover all of them).
- After some of these preprocessing steps, it is important to look at the EventList information and the EEG waveforms to verify that there are no problems or errors.
- You should also look at the data quality metrics that are computed during the averaging process to determine if there are problematic bins or channels (or a more general problem with the data quality for a given participant).
- In this experiment, the data analyses focus on the target words to see if the response to the target depends on whether the target word was related or unrelated to the preceding prime word. However, owing to noise, there were some differences in the ERP elicited by the prime words that were followed by either related or unrelated targets. Does this make it difficult to interpret the ERPs elicited by the target words? How does baseline correction help us avoid problems of this sort?

References

Kappenman, E. S., Farrens, J. L., Zhang, W., Stewart, A. X., & Luck, S. J. (2021). ERP CORE: An Open Resource for Human Event-Related Potential Research. *NeuroImage*, *225*, 117465. https://doi.org/10.1016/j.neuroimage.2020.117465

Kutas, M. (1997). Views on how the electrical activity that the brain generates reflects the functions of different language structures. *Psychophysiology*, *34*, 383–398.

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Luck, S. J., Stewart, A. X., Simmons, A. M., & Rhemtulla, M. (2021). Standardized Measurement Error: A Universal Measure of Data Quality for Averaged Event-Related Potentials. *Psychophysiology*. https://doi.org/10.1111/psyp.13793

Swaab, T. Y., Ledoux, K., Camblin, C. C., & Boudewyn, M. (2012). Language-related ERP components. In S. J. Luck & E. S. Kappenman (Eds.), *The Oxford Handbook of Event-Related Potential Components* (pp. 397–439). Oxford University Press.

This page titled 2.13: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3: PROCESSING MULTIPLE PARTICIPANTS IN THE ERP CORE N400 EXPERIMENT

Learning Objectives

In this chapter, you will learn to:

- Run a script for making averaged ERPs for multiple participants
- Create a grand average (an average of the single-participant averages)
- Visualize the grand average waveforms and noise metrics
- Apply low-pass filters to the grand average to minimize high-frequency noise
- Quantify N400 amplitude from single-participant averaged ERP waveforms
- Conduct a simple statistical analysis on the N400 amplitudes
- Conduct a more complex statistical analysis
- Run a script that automates the steps in this chapter

In the previous chapter, we processed the data from a single participant in the ERP CORE N400 experiment, up to the stage of creating and viewing the averaged ERP waveforms. In this chapter, we'll process nine more participants and do the steps necessary to get to the stage of quantifying N400 amplitude and conducting a simple statistical analysis.

Between the previous chapter and the current chapter, you'll be able to see most of the major steps required to go from raw data to a final conclusion. The details will be spelled out in subsequent chapters, and our goal here is for you to see the big picture. Also, to make things go quickly, we'll be looking at only 10 of the 40 participants in the actual study. If you're already experienced with ERPLAB, you can just skim this chapter.

Don't forget to consult the troubleshooting tips in Appendix 2 if you run into error messages or other problems.

3.1: Data for This Chapter

3.2: Exercise- Preprocessing and Averaging the Data from 10 Participants

3.3: Exercise- Examining the Single-Participant ERPsets

3.4: Exercise- "Bad" Data

3.5: Exercise- Making a Grand Average

3.6: Exercise- Low-Pass Filtering

3.7: Exercise - Scoring N400 Amplitude

3.8: Exercise- Simple Statistical Analysis of N400 Data

3.9: Exercise- A More Complex Analysis

3.10: Exercise- ERP Channel Operations

3.11: Exercise- ERP Bin Operations

3.12: Review of Processing Steps

3.13: Matlab Scripts For This Chapter

3.14: Key Takeaways and References

This page titled 3: Processing Multiple Participants in the ERP CORE N400 Experiment is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.1: Data for This Chapter

The N400 data we'll be using for the exercises in this chapter can be found in the Chapter_3 folder in the master folder: https://doi.org/10.18115/D50056. In this chapter, we'll be looking at the data from only 10 of the 40 participants (Subjects 1, 6, 7, 12, 15, 16, 21, 22, 34, 38).

This page titled 3.1: Data for This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.2: Exercise- Preprocessing and Averaging the Data from 10 Participants

Before you start this exercise, quit from EEGLAB if it's already running and then restart it so that you're starting fresh. You might even want to restart Matlab. Set **Chapter_3** (the folder with the data you downloaded for this chapter) to be Matlab's current folder.

For this chapter, you'll need averaged ERPs from 10 participants. You have three options for obtaining these ERPsets:

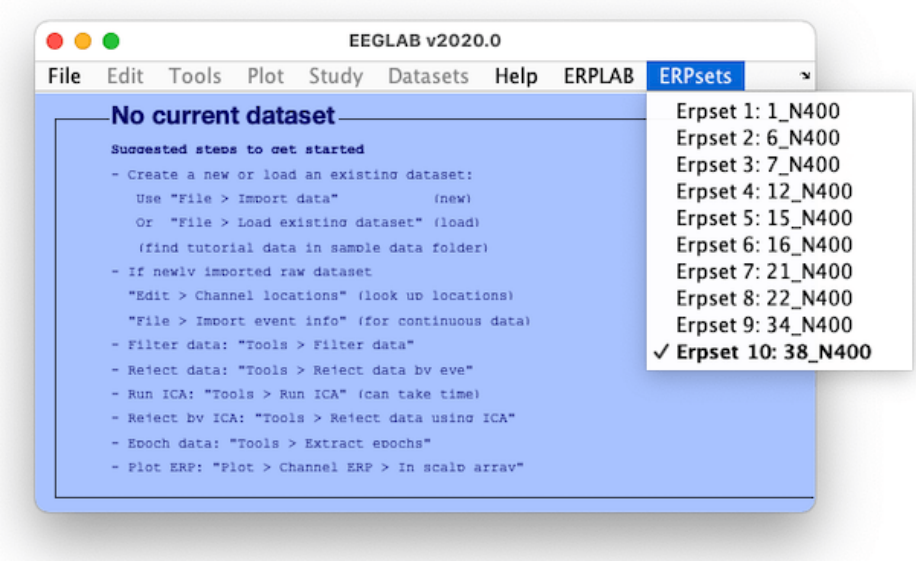
- You can go through all the steps from Chapter 2 for each participant using the EEGLAB/ERPLAB GUI. The **Chapter_3** folder contains a folder for each participant containing the raw EEG data, just like the data you processed in Chapter 2. This will take quite a bit of time, but you'll really learn the preprocessing steps well by repeating them for each participant.

- You can run the script named **preprocess_and_average.m** (which is in the Chapter_3 folder). It will run all the processing steps automatically and create the ERPset for each participant. The script file contains instructions for running it. The advantage of this approach is that you'll get more comfortable running Matlab scripts (which will become increasingly useful as you progress through the book). If you'd like, you can process some or all of the participants manually first and then run this script. When you're processing your own data, this is the right approach: You first process the data manually to make sure everything is OK, and then you run a script to reprocess the data to avoid any errors in the manual processing.

- There is a subfolder named **Pre-made_ERPsets** inside the **Chapter_3** folder that contains—you guessed it!—a pre-made ERPset for each participant. You can just move those ERPsets into the single-participant folders inside the **Chapter_3** folder. This is the quick-and-easy approach, but you won't learn as much.

No matter which way you do it, make sure that you end up with an ERPset file named *N*_**N400.erp** inside folder *N* for each participant (where *N* is the participant number). Then, load all 10 of these ERPsets into EEGLAB/ERPLAB (either with the script or using **EEGLAB > ERPLAB > Load existing ERPset**). The ERPsets menu should now look like Screenshot 3.1.

Screenshot 3.1



This page titled 3.2: Exercise- Preprocessing and Averaging the Data from 10 Participants is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.3: Exercise- Examining the Single-Participant ERPsets

Now let's take a look at the averaged ERP data from each participant. Select **ERPset 1** from the **ERPsets** menu so that we can examine the data from Subject 1. Go to the Matlab command window and type **ERP** (followed by Return or Enter). You should see something like the information shown in Screenshot 3.2.

Screenshot 3.2

```
>> ERP

ERP =

  struct with fields:

       erpname: '1_N400'
      filename: '1_N400.erp'
      filepath: '/Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Ch_3_Getting_Started_10_Participants/Exercises/1'
     workfiles: {'1_N400_preprocessed_filt_elist_bins_be_ar_from_script.set'}
       subject: ''
         nchan: 30
          nbin: 4
          pnts: 200
         srate: 200
          xmin: -0.2000
          xmax: 0.7950
         times: [1×200 double]
       bindata: [30×200×4 double]
      binerror: [30×200×4 double]
      datatype: 'ERP'
       ntrials: [1×1 struct]
     pexcluded: 29.6000
        isfilt: 0
      chanlocs: [1×30 struct]
           ref: 'common'
      bindescr: {1×4 cell}
         saved: 'yes'
       history: [3×2167 char]
       version: '8.12'
    splinefile: ''
     EVENTLIST: [1×1 struct]
   dataquality: [1×3 struct]
```

ERP is the name of a Matlab variable that stores the current ERPset (the one you selected in the **ERPsets** menu). When you typed the name of it in the command window, Matlab printed out the contents of the variable. It's a complicated variable with many different fields, including the **erpname** field (the name of the ERPset), the **nchan** field (which stores the number of channels), the **chanlocs** field (which stores the names and 3-D locations of the electrodes), and the **bindata** field (which stores the actual ERP data, in binary format). There is also an **EEG** variable that stores the current EEG dataset (if one is loaded).

The available variables can be seen in the Workspace pane of the Matlab GUI. You can also see the contents of a variable like **ERP** or **EEG** by double-clicking the name of the variable in the Workspace pane. This causes the variable to be shown in a separate window. You can then double-click on the fields of the **ERP** variable to see those fields in more detail. For example, try double-clicking the **chanlocs** field to see what information it holds.

Now type **ERP.ntrials** in the command window. Matlab will print out information about the number of accepted trials, the number of rejected trials, and the number of invalid trials for each bin. You should see that there were 47 accepted trials and 13 rejected trials for Bin 1. You can also get a slightly nicer table with the same information by selecting **EEGLAB > ERPLAB > Summarize artifact detection > Summarize ERP artifacts in a table**. You'll want to know this information before looking at the ERP waveforms for a given participant. For example, you'll want to know if there were any bins without a reasonable number of accepted trials.

Now plot the ERP waveforms with **EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms**. You can use the default settings, except just plot Bins 3 and 4 (the related and unrelated targets). Just as in the grand averages shown in Figure 2.1 in the previous chapter, you should see that the unrelated targets elicited a more negative voltage around 400 ms than the related targets. However, this participant's waveforms are a bit noisier than those of those of the participant we looked at in Chapter 2 (Subject 6). If you don't remember what Subject 6's waveforms looked like, you can select that participant's ERPset in the **ERPsets** menu and plot the waveforms.

To look at the data quality (the analytic standardized measurement error or aSME), select **EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table**.

Now repeat this sequence of steps to look at the data from all 10 participants. Here are some questions you should answer:

- Do you see the N400 effect (more negative for unrelated than related targets) in each participant's waveforms?
- Are there any participants who have an unusually small or large numbers of trials rejected?
- Are there any participants with appreciably worse aSME values?

These 10 participants were chosen because they all have pretty good data. They all have an N400 effect, and they all have pretty similar levels of data quality. If we looked at all 40 participants in the full study, we'd see a wider range of effects, numbers of rejected trials, and data quality.

This page titled 3.3: Exercise- Examining the Single-Participant ERPsets is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.4: Exercise- “Bad” Data

So far, we've been looking at really clean data. However, the reality of ERP research (and most areas of human neuroscience) is that you often get some participants with really noisy data. And in some areas, noisy data is the norm (and large numbers of participants are needed to make up for it. For example, imagine trying to record the EEG from wiggly 2-year-olds. You'd get all kinds of movement artifacts, and they won't sit through an hour of data collection the way a paid adult will. But you'll also see some noisy in studies of calm, compliant adults. So, no matter what kind of ERP research you're interested in, you'll probably need to learn to deal with noisy data.

In this exercise, we'll look at one of the 40 participants in the full N400 study whose data were problematic (Subject 30). This participant wasn't horrible—all of our participants were college students who were pretty compliant with our instructions, and we know a lot of tricks for optimizing the data quality in EEG recordings (see Farrens et al., 2019 for a detailed description of our EEG recording protocol). However, the data from this participant were problematic in a way that we often see in our college student population.

You can find this participant's data in the folder named **Bad_Subject** inside the **Chapter_3** folder. I've already preprocessed the EEG and made the averaged ERPs, so you don't need to go through those steps. The folder contains the original EEG dataset file, the EEG dataset file after all preprocessing steps (including artifact detection), and the averaged ERPset file.

Start by loading the averaged ERP data from this participant (**EEGLAB > ERPLAB > Load existing ERPset**) and plotting Bins 3 and 4 (**EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms**). You should see a very noisy waveform for the related target words, but the waveform for the unrelated target words is missing. If you look at the aSME data quality metric (**EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table**), you'll find an aSME value of 0 for every channel at every time point for Bin 3 (related targets), and a value of **NaN** for Bin 4 (unrelated targets). **NaN** is an abbreviation for *not a number*, and it's what Matlab uses when something can't be computed (e.g., when it requires dividing by zero).

Now plot the ERP waveforms for the prime words (Bins 1 and 2) and look at the aSME values for these words. The waveforms are noisy, and the aSME values are higher than those for the 10 participants you processed in the previous exercises. But at least it looks like there is valid data for these bins.

Your job now is to figure out what has gone wrong with Bins 3 and 4 for this participant. In Chapter 2, I made a point of describing several checks that you should perform while processing a participant's data (see summary of steps in Section 2.12). Section 3.4 of the present chapter describes some additional checks. Go through these checks to figure out what went wrong with this participant. Once you've done it, you can read the text box below to make sure your answer was correct (but no peaking until you've figured it out for yourself!).

I hope you've now figured out the problem with Subject 30. I included this example to drive home a point that I made in Chapter 2, namely that you really need to pay close attention when you're initially processing each participant's data. Don't just run a script and hope for the best. Look at the number of event codes, the number of accepted and rejected trials, the continuous EEG, and the epochs that were marked by the artifact detection process. If you don't, your data will be filled with C.R.A.P. (which is an acronym for *Commonly Recorded Artifactual Potentials*, but also refers to a variety of other problems, such as incorrect event codes). And as they say: *garbage in, garbage out*. So, if you want your experiments to yield robust, statistically significant, and accurate results, pay close attention to the data!

What's wrong with Subject 30?

If you load the ERPset for Subject 30 and look at ERP.ntrials, you'll see that there was only one accepted trial in Bin 3 and there were zero accepted trials in Bin 4. And if you load one of the EEG dataset files and look at the EEG, you'll see that this participant blinked a lot. In particular, the participant blinked right around the time of the buttonpress response (event code 201) on almost every trial. As a result, the ERP waveform for Bin 3 was based on an “average” of only one trial, and the aSME value was zero. Bin 4 had no trials, so no ERP waveform could be plotted for that bin, and the aSME value was not a number (NaN). Well over half the trials were also rejected in Bins 1 and 2, and the data were just generally noisy for this participant. That's why the aSME values were bad even for Bins 1 and 2.

When you loaded the ERPset for Subject 30 into ERPLAB, the fact that there were no trials in Bin 4 led to a warning message that was printed in red text in the command window (WARNING: bin #4 has flatlined ERPs). You can probably still see it if you scroll up. You probably didn't notice it when it first happened, because it probably scrolled off the screen before you could see it. When you run into a problem (like a bin that doesn't appear to plot properly), you should look at the command window (scrolling up if necessary) to see if any warning or error messages were printed. That can help you find problems like this.

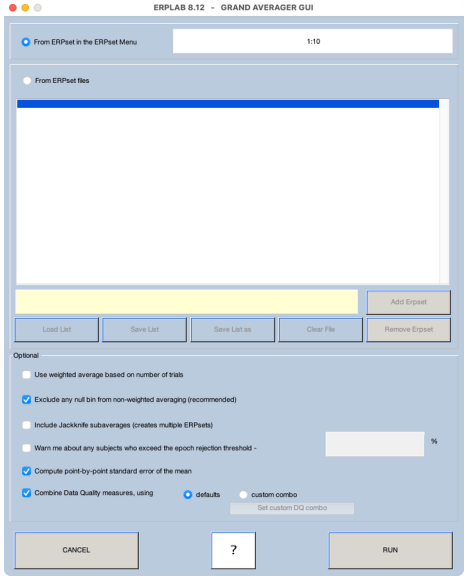
So, what can we do about this subject? In the published version of the N400 ERP CORE experiment, we used *artifact correction* instead of artifact rejection to deal with blinks. That is, we used a procedure called *independent component analysis* to estimate and remove the part of the signal that was caused by blinking. We rejected trials with blinks only if the blinks happened near time zero, indicating that the eyes were closed when the word was presented (which was rare). Consequently, we were able to include almost all the trials from every participant in our averaged ERP waveforms.

This page titled 3.4: Exercise- "Bad" Data is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.5: Exercise- Making a Grand Average

Although it's important to look at the single-participant ERP waveforms, it will be easier to see subtle effects by averaging the waveforms across participants to create *grand average* waveforms for each bin. To do this, make sure that the ERPsets from all 10 participants are loaded (which you can verify by looking at the **ERPsets** menu), and then select **EEGLAB > ERPLAB > Average across ERPsets (Grand Average)**. A new window will open that look something like Screenshot 3.3.

Screenshot 3.3

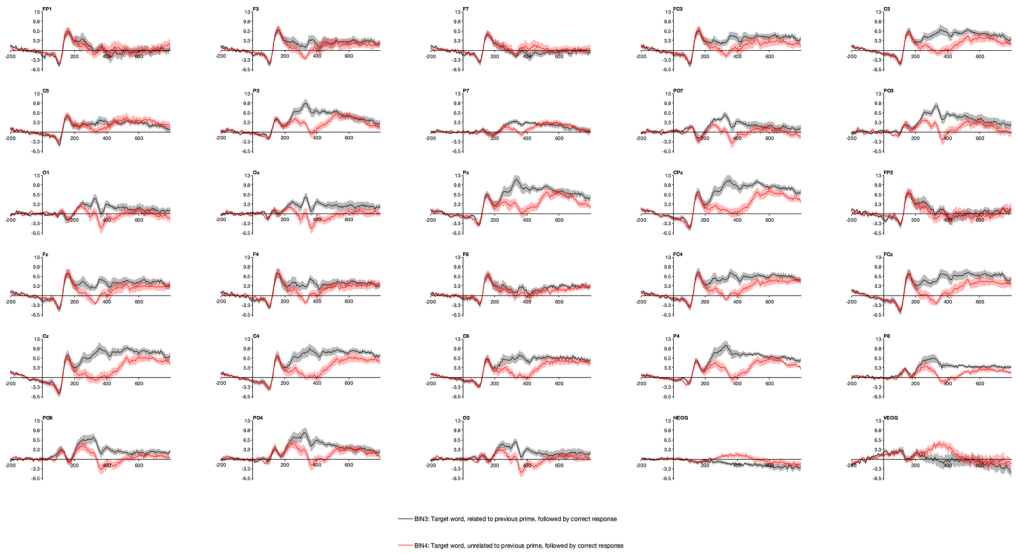


You need to specify which ERPsets will be averaged together. You can do this either by specifying a set of ERPsets that have already been loaded (using the ERPset numbers in the **ERPsets** menu) or by specifying the filenames for ERPsets that have been stored in files. In this exercise, we'll specify the ERPsets that have already been loaded. If you have only the 10 ERPsets from our 10 example participants in your ERPsets menu, you can specify **1:10** (as in Screenshot 3.3). In Matlab, you can indicate a list of consecutive numbers by providing the first and last numbers, separated by a colon. So, **1:10** is equal to **1 2 3 4 5 6 7 8 9 10**. If these aren't the right ERPsets (because you have others also loaded into ERPLAB), just provide a list of the ten numbers for the ERPsets you want to average together.

You can leave the other options set to their default values (making sure that they match Screenshot 3.3). Then click **RUN**. You'll then see the usual window for saving the new ERPset that you've created. Name it **Grand_N400** (and save it as a file so that you have it for the subsequent exercises). You should now see **Grand_N400** in the **ERPsets** menu.

Now, plot the ERPs from Bins 3 and 4 (using **EEGLAB > ERPLAB > Plot ERPs > Plot ERP waveforms**). But let's add something new: Tick the box labeled **show standard error** (and make sure that the transparency level is set to 0.7). You should see something like Screenshot 3.4, with a more negative voltage for the unrelated targets than for the related targets at CPz and surrounding electrode sites. The light shading around the waveforms is the standard error of the mean at each time point (see the box below for more information).

Screenshot 3.4



Now plot Bins 1 and 2. The waveforms for these bins should be lying right on top of each other, with any differences being small relative to the SEM. Remember, these are the bins for primes that are followed by related versus unrelated targets, and unless the participants have ESP, they can't differ as a function of something that happens later. As a result, any differences between them are simply a result of noise.

Finally, take a look at the aSME data quality values for the grand average (**EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table**). When you made the grand average, the default settings caused the aSME values from the individual participants to be combined together using something called the *root mean square* (RMS). This is like taking the average of the single-participant aSME, except that the RMS value is more directly related to the impact of each participant's data quality to the effect size (see Luck et al., 2021). These aSME values are like a standard error, but they're the standard error of the mean voltage over a 100-ms time window rather than the SEM at a single time point (see the box below). If you look at the table of aSME values, you'll see that the values in the N400 time range are around 1.5 μV. That's reasonably small relative to the large difference in mean voltage between the unrelated and related targets. In other words, the data quality is quite good for our goal of detecting differences between these two types of targets.

Plotting the Standard Error

Plotting the standard error of the mean (SEM) at each time point in an ERP waveform, as in Screenshot 3.4, can be helpful in assessing whether the differences between conditions are reasonably large relative to the variability among participants. These standard error values are just like the error bars that you might see in a bar graph. At each time point, the grand average ERP waveform is simply the mean of the voltage values across participants at that time point. The SEM is just the standard deviation (SD) of the single-participant values divided by the square root of the number of participants (which is exactly how the SEM is usually calculated in other contexts).

You can also see the SEM when you plot a single participant's averaged ERP waveforms. In this case, the waveform shows the mean across trials rather than the mean across participants, and the SEM reflects the variability across trials rather than the variability across time points.

Although the SEM can be useful, it has some downsides. First, imagine that the voltage at 400 ms is exactly 3μV more negative for unrelated targets than for related targets in every participant (i.e., the experimental effect is extremely consistent across participants). But imagine that the overall voltage at 400 ms is much more positive in some participants than others, leading to quite a bit of variability in the voltage for each condition. Because of this variability, the SEM for each waveform would be quite large at 400 ms. This would make it look like the difference in means between conditions was small relative to the SEM, even though the difference for each participant was extremely consistent. In behavioral research, this problem is addressed by using the within-subjects SEM (Cousineau, 2005; Morey, 2008). ERPLAB doesn't have this version of the

SEM built in, but you can achieve the same result by making a difference wave between the conditions (e.g., unrelated targets minus related targets) and getting the SEM of the difference wave. This is exactly what was done in the grand averages from the full study (see Figure 2.1C in Chapter 2).

Another downside of the SEM is that it can be very large if there is a lot of high-frequency noise in the data, even though this noise has minimal impact when we quantify the N400 as the mean voltage between 300 and 500 ms (as we will do later in this chapter). The aSME value provided in our Data Quality measures does not have this downside, because it provides the standard error of the mean voltage over a time period rather than the standard error of the values at individual time points. See Luck et al. (2021) for a more detailed discussion.

This page titled 3.5: Exercise- Making a Grand Average is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

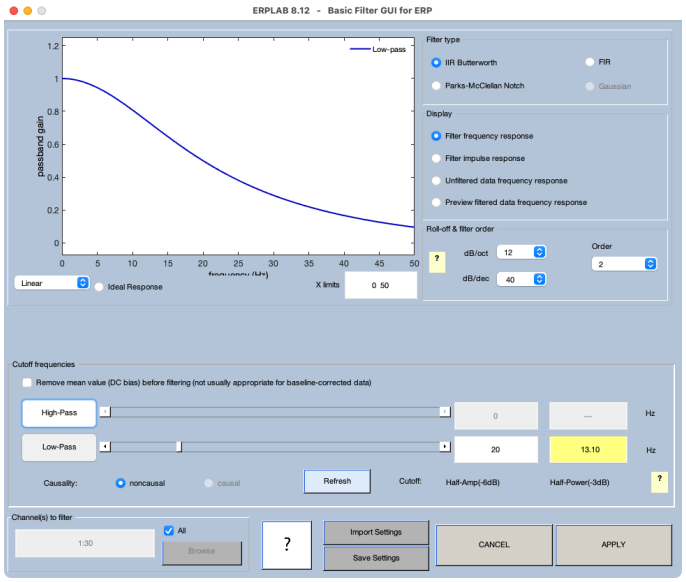
3.6: Exercise- Low-Pass Filtering

We've already filtered out the low-frequency voltage drifts in the continuous EEG data for each participant's data, but there is also some high-frequency noise (small but rapid deflections in the waveforms). In our lab, we take great pains to reduce the major sources of high-frequency noise (induced voltages from electrical devices in the environment and muscle activity). As a result, there isn't a lot of high-frequency noise in the grand averages shown in Screenshot 3.4. But there's a little, and you'll see a lot more in most experiments (especially in the single-participant waveforms). So, this exercise will show you how to filter out high-frequency noise using a *low-pass filter*. We'll apply it to the grand average ERP waveform, but you could instead apply to the single-subject ERPs, the epoched EEG data, or even the continuous EEG data (see Chapter 7 in Luck, 2014 for information about when different filters should be applied).

Make sure that the **Grand_N400** ERPset is still loaded in ERPLAB, and then select **EEGLAB > ERPLAB > Filter & Frequency Tools > Filters for ERP data**. You'll see a window that looks nearly identical to the filtering GUI you used to filter out low-frequency drifts in the continuous EEG data. Set it up as shown in Screenshot 3.5, which should mainly involve setting the low-pass cutoff to 20 Hz. Then click **APPLY** to run the filtering routine. You can name the new ERPset **Grand_N400_filt**.

Now plot the new ERPset. In the plotting GUI, notice that the option for plotting the standard error is grayed out. When you filter the data, the original standard error values are no longer valid—they're the standard error of the unfiltered mean voltage at each time point, not the standard error of the filtered values. If you want to see the standard error of the filtered data, you'd need to filter the single-participant ERPs prior to making the grand average.

Screenshot 3.5



Now compare the filtered waveforms to the original unfiltered waveforms. (If you don't still have the plot of the unfiltered waveforms, select **Grand_N400** from the **ERPsets** menu and run the plotting routine). You should see that the filtered waveforms look smoother than the unfiltered waveforms. In a later chapter, we'll take a closer look at filtering and see how filters can reduce noise but can also distort the data, and you'll learn how to select filters that make your data cleaner without producing significant distortions.

This page titled 3.6: Exercise- Low-Pass Filtering is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.7: Exercise - Scoring N400 Amplitude

Our next step is to quantify the amplitude of the N400 component for the unrelated and related target words so that we can perform statistical analyses (which we will do in the next exercise). We need N400 amplitude values from each individual participant, so make sure that you still have the single-participant ERPsets loaded in ERPLAB (you can check this in the **ERPsets** menu).

We'll quantify the amplitude of the N400 as the mean voltage between 300 and 500 ms. Let's be concrete about what this means. These data are sampled at 200 Hz, which means that there is one voltage value every 5 ms. This gives us 21 values between 300 and 500 ms (not 20, because we include the value at 300 ms and the value at 500 ms, plus the values between them). We calculate the mean voltage by summing together these 21 values and dividing by 21. It's that simple. If you want to know why we quantify the amplitude of an ERP component this way, see Chapter 9 in Luck (2014). We'll apply this procedure to the averaged ERP waveforms from each participant, once for the unrelated targets and once for the related targets. Initially, we'll just do this for the CPz electrode site.

Scoring Versus Measuring the Amplitude or Latency of an ERP Component

When researchers apply an algorithm to an ERP waveform to quantify the amplitude or latency of an ERP component, we often say that we're *measuring* the amplitude or latency. But I don't really like this terminology. An ERP component is a hypothetical entity in the brain, and we're obtaining a value from a scalp signal that typically consists of a mixture of many components. As a result, it's not really a measurement of the component. Also, it seems weird to use the term "measuring" when we're taking values that were already measured (the EEG voltages) and recombining them in a new way. So, it seems perfectly natural to say that we're "measuring the EEG," but it seems odd to say that we're "measuring the amplitude of the N400 component."

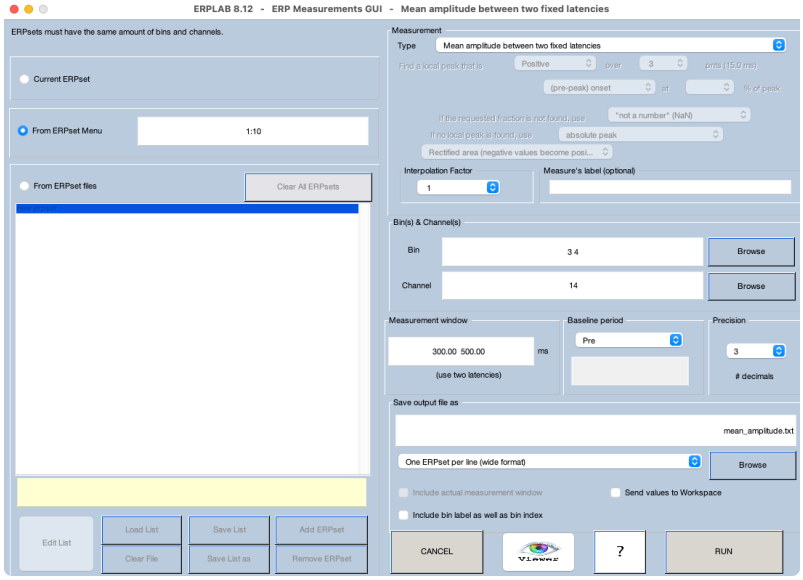
I prefer to use the term *score*. That is, we're "scoring the amplitude of the N400 component," and we obtain "N400 amplitude scores." I started using this terminology just a couple years ago, so you'll see the term *measurement* rather than *score* in many of my writings and in ERPLAB Toolbox.

To score the N400 amplitude as the mean voltage between 300 and 500 ms, select **EEGLAB > ERPLAB > ERP Measurement Tool**. You'll see the big complicated window shown in Screenshot 3.6. The left side of the window is used to indicate which ERPsets should be measured. Our ERPsets are loaded into ERPLAB already, so select **From ERPset Menu** and indicate the ERPset numbers for the 10 single-participant ERPsets. In the example shown in Screenshot 3.6, these are ERPsets 1-10, so we specify it as **1:10** (or, equivalently, **1 2 3 4 5 6 7 8 9 10**). If we didn't want to load the ERPsets into ERPLAB, we could instead provide a list of the filenames using **From ERPset files**.

The right side of the window is used to specify how we want to score the data. The measurement type should be set to **Mean amplitude between two fixed latencies**. In a later chapter, we'll go over the other scoring methods (including peak amplitude and peak latency, which are widely used but often inferior to the other options provided by ERPLAB). We want to obtain the scores from Bins 3 and 4 and from Channel 14 (CPz), so make sure those are specified in the **Bin(s) & Channel(s)** section. The **Measurement window** field should be **300 500** (with a space between the two numbers) to indicate the starting and ending latencies of the time window for computing the mean amplitude score.

The **Baseline period** should be set to **pre**, which indicates that the entire prestimulus period should be used as the baseline. You already baseline-corrected the data during the epoching process, but it doesn't hurt to re-baseline the data just to be sure.

Screenshot 3.6

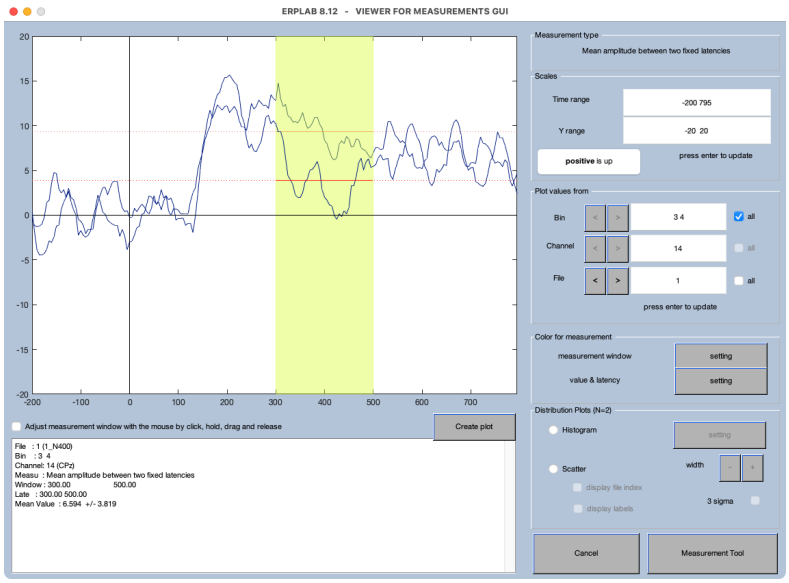


In the **Save output file as** section, you should use **mean_amplitude.txt** as the name of the file used to store the scores. Select **One ERPset per line (wide format)**, which will produce a format that works well with statistical packages that expect each line to be one participant and each column to be a different score from that participant.

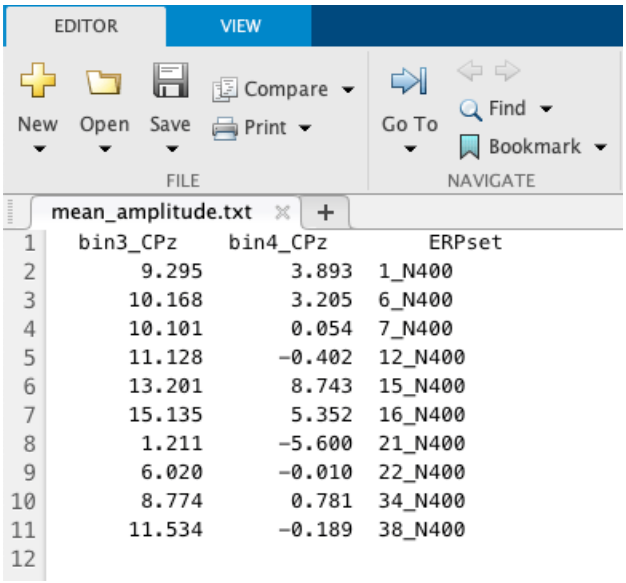
Most people would just click **RUN** at this point to obtain the amplitude scores. But there is another very important step, which is to click the **Viewer** button. This will allow you to see the scores for each ERP waveform, which is very important for making sure that the scoring procedure is working in a sensible manner. For example, a measurement window that seems appropriate when you're looking at grand average waveforms may not actually work well on the single-participant waveforms.

The Viewer is shown in Screenshot 3.7. I've set it to show all the bins. You can scroll through the different files (participants) to see how the scoring is working for all the waveforms. From my perspective, everything looks like it's working fine. Once you're done, click the **Measurement Tool** button to go back to the Measurement Tool. Then you can click **RUN** to obtain the amplitude scores and save them in the text file.

Screenshot 3.7



If you look in the Current Folder section of the Matlab GUI, you'll see that a file named **mean_amplitude.txt** has been created. Double-click on it to open it in the Matlab text editor. It should look something like this:



	bin3_CPz	bin4_CPz	ERPset
1			
2	9.295	3.893	1_N400
3	10.168	3.205	6_N400
4	10.101	0.054	7_N400
5	11.128	-0.402	12_N400
6	13.201	8.743	15_N400
7	15.135	5.352	16_N400
8	1.211	-5.600	21_N400
9	6.020	-0.010	22_N400
10	8.774	0.781	34_N400
11	11.534	-0.189	38_N400
12			

The first column has the amplitude scores from Bin 3 (related targets) and the second column has the scores from Bin 4 (unrelated targets). The third column is the name of the ERPset, which tells you which subject was measured on that line. You can see that every single participant has a more negative (less positive) voltage for the unrelated trials than for the related trials.

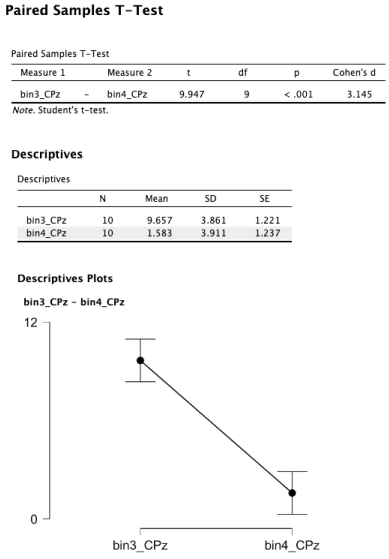
This page titled 3.7: Exercise - Scoring N400 Amplitude is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.8: Exercise- Simple Statistical Analysis of N400 Data

Now we're going to perform a simple statistical analysis of the N400 amplitude scores that you obtained in the previous exercise. We have two amplitude scores for each participant, one for related targets and one for unrelated targets, and we want to know if the scores are significantly different for these two experimental conditions. The simplest way to do this is with a paired *t* test.

I used the free JASP statistical package to run the *t* test, but you can use whatever package you find comfortable. Make sure you specify a paired *t* test rather than an independent-samples *t* test. The results are shown in Screenshot 3.8. Before you look at the *t* and *p* values, you should always look at the descriptive statistics. Once we get to more complex analyses, it will be really easy to make mistakes in the statistical analysis. The most common mistake is to incorrectly specify which variable is in which column of the data file. For example, you might think that the unrelated and related targets are stored in the first and second columns, respectively, reversing the actual order. This kind of error becomes both more likely and more likely to lead to incorrect conclusions when your design has several factors and each row of the data file has a dozen or more columns. By comparing the group means from the statistical analysis to the grand average waveforms, you can often detect these errors.

Screenshot 3.8



If you look at the group means in Screenshot 3.8, you'll see a mean of 9.657 µV for the related targets (Bin 3) and 1.583 µV for the unrelated targets (Bin 4). Those values at least approximately match what you can see for the CPz channel from 300-500 ms in the grand average waveforms shown in Screenshot 3.4.

Now that we've verified that the descriptive statistics look correct, we can look at the *t* and *p* values. The effect was significant at the $p < .001$ level, and the effect size (Cohen's *d*) was huge. The effect size of 3.145 indicates that the difference between the group means for related and unrelated targets was 3.145 times as large as the standard deviation of the scores. You won't find effects this large in most experiments, but the N400 ERP CORE experiment was carefully designed to maximize the experimental effects, and we chose a paradigm that was known to produce large effects. Also, I chose 10 participants with really clear effects for the exercises in this chapter; the effect size was "only" 2.33 in the full sample of 40 participants (but this was still a huge effect size).

Limits on Comparing Descriptive Statistics with Grand Average Waveforms

When we score the amplitude of an ERP component as the mean voltage in a fixed time window, we can directly compare the group mean values from the statistical analysis with the grand average ERP waveforms. This is because this scoring method is a *linear operation* (for a definition and more information, see the Appendix in Luck, 2014). The order of operations does

not matter for linear operations. This means that we can obtain our mean amplitude score from the single-subject waveforms and then compute the mean of these scores, and we will get exactly the same value that we would obtain by scoring the mean amplitude from the grand average waveform.

Unfortunately, most other scoring methods are not linear. For example, the peak amplitude in a given time window is not linear. If we obtain the peak amplitude from the single-subject waveforms and then compute the mean of these scores, the result will not be the same as the peak amplitude of the grand average waveform. However, you should still compare the group means from your statistical analysis with the grand average waveforms. If there is a large mismatch, then you may have made an error in specifying the order of variables in your statistical analysis, or your grand average waveform may not adequately represent what is happening at the single-subject level. In either case, you want to know!

This page titled 3.8: Exercise- Simple Statistical Analysis of N400 Data is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.9: Exercise- A More Complex Analysis

In this exercise, we're going to repeat the N400 analysis from the previous exercises, but we're going to make it more complex by measuring and analyzing the N400 at multiple electrode sites. We'll set this up with two electrode factors: laterality (left hemisphere, midline, and right hemisphere) and anterior-posterior (frontal, central, and parietal). That is, we'll obtain scores from F3, Fz, F4, C3, Cz, C4, P3, Pz, and P4. When we combine this with the relatedness factor, this will give us a factorial design with three total factors. We won't include CPz in these analyses because we don't have electrodes at CP3 and CP4 and we don't want an unbalanced design.

Launch the Measurement Tool again and set it up exactly as before (Screenshot 3.7) except for the list of channels. If you click the **Browse** button next to the text box for the channels, you'll be able to select the nine electrode sites that we want. After you've selected them, click **OK** to go back to the Measurement Tool. You should now see **2 5 7 13 16 17 21 22 24** in the text box. These are the channels we want. You should also change the name of the output file to be **mean_amplitude_multiple_channels.txt**. Use the **Viewer** to make sure that everything looks OK, and then click **RUN** in the Measurement Tool.

Now open the **mean_amplitude_multiple_channels.txt** file in the Matlab text editor. The text editor doesn't deal with the tabs very well, so you might want to import the file into Excel instead. Now we have 19 columns: 9 channels x 2 bins plus the ERPset column. Unfortunately, the channels are in the order that they appear in the ERPsets, which is not very convenient. If you're not sure whether the amplitude scores are correct, you can launch the Measurement Tool again and use the Viewer to see the single-subject scores.

Once you've verified that the scores are correct, you can enter the data into a statistical analysis. You should use a 3-way repeated-measures ANOVA with factors of relatedness, laterality, and anterior-posterior. I ran this ANOVA in JASP, and the results are shown in Screenshot 3.9.

Screenshot 3.9

Repeated Measures ANOVA

Within Subjects Effects

Cases	Sum of Squares	df	Mean Square	F	p
Related vs Unrelated	1200.196	1	1200.196	118.159	< .001
Residuals	91.417	9	10.157		
Left–to–Right Electrode Site	46.522	2	23.261	12.398	< .001
Residuals	33.772	18	1.876		
Anterior–to–Posterior Electrode Site	342.875	2	171.438	21.132	< .001
Residuals	146.028	18	8.113		
Related vs Unrelated × Left–to–Right Electrode Site	26.646	2	13.323	17.022	< .001
Residuals	14.088	18	0.783		
Related vs Unrelated × Anterior–to–Posterior Electrode Site	123.671	2	61.836	30.934	< .001
Residuals	35.981	18	1.999		
Left–to–Right Electrode Site × Anterior–to–Posterior Electrode Site	8.964	4	2.241	1.236	0.313
Residuals	65.259	36	1.813		
Related vs Unrelated × Left–to–Right Electrode Site × Anterior–to–Posterior Electrode Site	1.957	4	0.489	0.890	0.480
Residuals	19.797	36	0.550		

Note. Type III Sum of Squares
ᵃ Mauchly's test of sphericity indicates that the assumption of sphericity is violated (p < .05).

Between Subjects Effects

Cases	Sum of Squares	df	Mean Square	F	p
Residuals	1950.368	9	216.708		

Note. Type III Sum of Squares

Descriptives

Descriptives

Related vs Unrelated	Left–to–Right Electrode Site	Anterior–to–Posterior Electrode Site	Mean	SD	N
Related	Left	Central	5.764	3.545	10
		Frontal	2.220	4.496	10
		Parietal	7.274	2.808	10
	Midline	Central	8.409	3.795	10
		Frontal	3.970	3.709	10
		Parietal	8.992	3.725	10
	Right	Central	8.329	4.367	10
		Frontal	3.669	3.682	10
		Parietal	8.387	3.328	10
Unrelated	Left	Central	0.656	3.894	10
		Frontal	0.272	4.662	10
		Parietal	2.059	3.378	10
	Midline	Central	0.895	3.643	10
		Frontal	0.472	3.768	10
		Parietal	2.434	4.185	10
	Right	Central	1.405	3.812	10
		Frontal	0.627	3.681	10
		Parietal	1.716	3.751	10

Again, start by looking at the descriptive statistic and make sure they match the grand average waveforms in Screenshot 3.4. For example, in both cases the amplitude for the related trials increases from the frontal to the central and parietal channels, and it tends to be larger for the midline and right-hemisphere channels than for the left-hemisphere channels. You can also see the basic N400 effect in both the grand average waveforms and the group means: the voltage is more negative (less positive) for the unrelated targets than for the related targets.

If you look at the *F* and *p* values, you'll see that the main effect of relatedness (related vs. unrelated) was significant at the $p < .001$ level. The laterality and anterior-posterior main effects were also significant, and these factors both interacted significantly with relatedness. That is, the difference between related and unrelated words was largest at the sites where the voltage was largest. This pattern of interactions is exactly what would be expected given the multiplicative relationship between the magnitude of an

internal ERP generator and the observed scalp distribution (see Chapter 10 in Luck, 2014).

You've now completed a fairly sophisticated analysis of the N400 experiment. Congratulations! That was a lot of steps, and it took us two chapters to get to this point.

However, I should note that I don't generally recommend scoring a component from multiple sites and including electrode site factors in the statistical analysis. The reasoning is described in the text box below. Sometimes it is justifiable, such as when your scientific hypothesis leads to a prediction of different effects over the left and right hemispheres. But unless you have a real reason to compare the effects across electrode sites, it's usually better to limit your analysis to a single site or create a waveform that averages across multiple sites. We'll explore the latter option in the next exercise.

Minimizing the Number of Factors in an Analysis

The problem with including one or more electrode site factors is that it leads to a large number of statistical tests, increasing the likelihood that you'll get one or more significant effects that are a result of random noise in the data. Look at the table of statistics at the top of Screenshot 3.9—how many p values do you see? Seven!

Ordinarily, you would expect a 5% probability that an effect will be significant ($p < .05$) when the null hypothesis is true. However, if the null hypothesis were true for all seven of these tests, the chance that one or more would be significant ($p < .05$) would be greater than 30%!

As we increase the number of factors in an ANOVA, the number of main effects and interactions skyrockets, and the odds that one or more will be significant by chance becomes extremely high (Cramer et al., 2015; Frane, 2021). For example, in a 5-way ANOVA, you are more likely than not to obtain a significant-but-bogus-effect. As a result, it is difficult to trust the results of such analyses. My general advice is therefore to minimize the number of factors (see Luck & Gaspelin, 2017 for a detailed discussion).

This page titled 3.9: Exercise- A More Complex Analysis is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.10: Exercise- ERP Channel Operations

In this exercise, we will take a look at two kinds of processing operations that are commonly applied to ERPs in which either the bins or the channels are mathematically recombined. For example, we could average together two bins or two channels. These kinds of operations are carried out with **ERP Bin Operations** and **ERP Channel Operations**, respectively.

Let's start by looking at ERP channel operations. In the previous exercise, I noted that it is often better to average across channels rather than to include channels as a factor in the statistical analysis. In the present exercise, we'll average across the nine channels that we used in the previous exercise (F3, Fz, F4, C3, Cz, C4, P3, Pz, and P4). Ordinarily, you would perform this averaging process on the single-subject waveforms, then obtain the N400 amplitude scores, and then conduct the statistical analysis on these scores. To make this exercise quick, however, we will instead average across channels in the grand average ERP waveform. This will allow us to visualize the results of averaging but not perform a statistical analysis on the averaged data.

Before we look at how the averaging process is implemented in ERPLAB, let's think for a minute about how you would compute the average of these nine channels by hand. The average of nine values is just the sum of those nine values divided by nine. So, to compute the average, you would use this equation:

$$\frac{F3 + Fz + F4 + C3 + Cz + C4 + P3 + Pz + P4}{9} \tag{3.10.1}$$

This is how you perform channel operations in ERPLAB. That is, you specify an equation that describes exactly what you want to compute, and ERPLAB computes it for you. The only difference is that you need to use the channel numbers rather than the channel names.

Let's give it a try. If necessary, load the grand average you made earlier (**Grand_N400**) into ERPLAB and make it the active ERPset. Now select **EEGLAB > ERPLAB > ERP Operations > ERP Channel operations**.

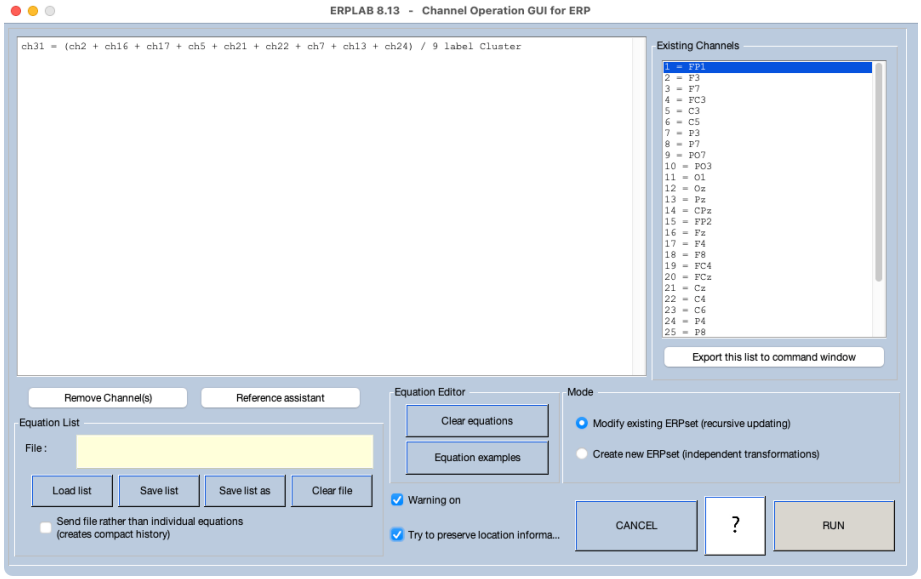
You'll see a window like the one shown in Screenshot 3.10. The panel on the right side gives you a list of the available channels and the channel numbers. The panel on the left is a text window that allows you to write one or more equations. You can see that we currently have 30 channels, so the new channel we will create will be channel 31. We'll call this new channel **Cluster**. To make this happen, we use the equation:

ch31 = (ch2 + ch16 + ch17 + ch5 + ch21 + ch22 + ch7 + ch13 + ch24) / 9 label Cluster

It's just like the previous equation, except that we use channel numbers instead of channel names, and we add **label Cluster** to the end of the equation to indicate the name of the new channel. Type this equation into the text box. Make sure that all the other parts of the window match Screenshot 3.10, especially **Modify existing ERPset**, and then click **RUN**.

Because we're modifying an existing ERPset rather than creating a new ERPset, you won't see a window for saving the ERPset. However, if you look at the ERPsets menu, you'll see that the name of the current ERPset has been changed from **Grand_N400** to **Grand_N400_chop** to indicate that Channel Operations (abbreviated as "chop") has been performed. If you want to save the changed ERPset, you can select **EEGLAB > ERPLAB > Save Current ERPset as**.

Screenshot 3.10



Now let's take a look at the result of this operation. Select **EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms** and plot Bins 3 and 4. At the bottom of the plot, you'll see the new channel, labeled **Cluster**. And you'll see that it looks like what you'd expect for the average of the 9 individual channels. If you performed this operation on the single-subject data, you could use the **ERP Measurement Tool** to score the N400 amplitude from this new channel. Note that there is also an **EEG Channel Operation**s routine in ERPLAB that works in the same way except that it operates on EEG data (continuous or epoched) rather than on averaged ERP data.

This page titled 3.10: Exercise- ERP Channel Operations is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.11: Exercise- ERP Bin Operations

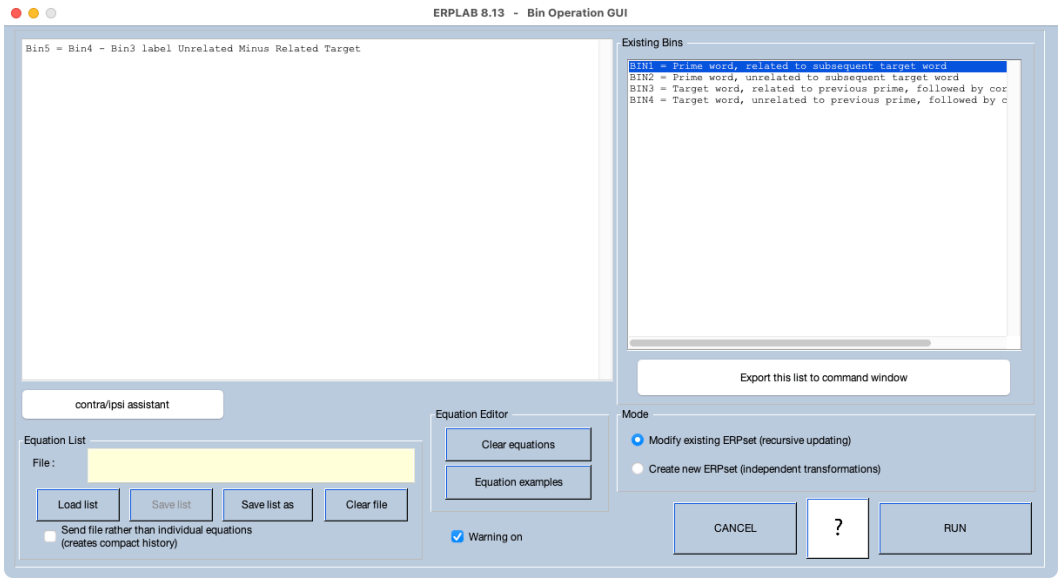
Now that we've seen how to recombine channels with **ERP Channel Operations**, let's see how to recombine bins using **ERP Bin Operations**. One of the most common reasons to recombine bins is to make a *difference wave*, in which we subtract one bin from another. In an N400 experiment, for example, we can isolate the brain activity that differs between related and unrelated target words by constructing an unrelated-minus-related difference wave.

Our grand average ERP waveform currently has 4 bins, with Bin 3 being related targets and Bin 4 being unrelated targets. To create an unrelated-minus-related difference wave as Bin 5, we use the following equation:

Bin5 = Bin4 - Bin3 label Unrelated Minus Related Target

Make sure that the grand average from the previous exercise (**Grand_N400**) is still loaded and is the active ERPset. Now select **EEGLAB > ERPLAB > ERP Operations > ERP Bin operations**. You'll see a new window like the one shown in Screenshot 3.11. The available bins are listed in the panel along the right side of the window, and there is a text box for writing equations. Type the above equation into that box. Note that there is no space between "Bin" and the bin number. And make sure you use a minus sign rather than a dash. Make sure the rest of the window is set up as shown in the screenshot, and click **RUN**.

Screenshot 3.11



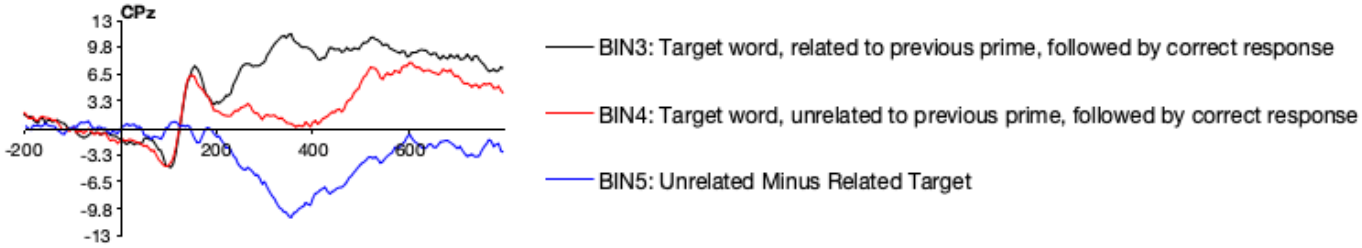
Because we're modifying an existing ERPset rather than creating a new ERPset, you won't see a window for saving the ERPset. However, if you look in the Matlab Command Window, you'll see something like this:

%Equivalent command:
ERP = pop_binoperator(ERP, { 'Bin5 = Bin4 - Bin3 label Unrelated Minus Related Target'});

This is the command that you would include in a script to achieve the same result. You'll learn more about scripting later, but I thought it would be good for you to see that the equivalent Matlab command is printed in the Command Window every time you perform an operation in the ERPLAB GUI.

Now let's see what the difference wave looks like. Select **EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms** and plot Bins 3, 4, and 5. Screenshot 3.12 shows the waveforms from the CPz site. You can see that the voltage in Bin 5 at a given time point is the difference between the voltages for Bins 3 and 4 at that time point. Note that the difference wave subtracts away any brain activity that is identical for the related and unrelated targets, such as the negativity at ~100 ms and the positivity at ~180 ms. All that remains is the brain activity that differentiates between the related and unrelated targets. This is an excellent way of isolating the brain activity of interest from the ERP waveforms. If you read my general book on ERPs (Luck, 2014), you'll see that I'm a big fan of using difference waves to isolate specific brain responses.

Screenshot 3.12



Lost Information as a Result of Bin and Channel Operations

The original EEG data files that we used when we began processing this experiment contained information about the 3-dimensional locations of the individual electrode sites. When you use **Channel Operations** to create a new channel, ERPLAB has no way of knowing the 3-dimensional location that should be used for this channel. And if you modify a channel, ERPLAB isn't smart enough to know whether the original channel location is still valid. To avoid making assumptions that might turn out to be incorrect, ERPLAB discards the channel information for any new or changed channels when you perform **Channel Operations**. (If you check the **Try to preserve location information** box in the GUI, ERPLAB will make a guess about the locations for changed locations, which usually works pretty well.)

The data quality information (including the SEM at each time point) is also lost when you perform Channel Operations. ERPLAB is not smart enough to know how the data quality should be updated for new or modified channels. Similarly, ERPLAB isn't smart enough to estimate the data quality when you perform **Bin Operations**.

The bottom line is that if information about channel locations or data quality is missing, it is likely a result of Bin Operations or Channel Operations.

This page titled 3.11: Exercise- ERP Bin Operations is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.12: Review of Processing Steps

To review, here are the steps we carried out in this chapter:

- Processed the single-participant to get an ERPset for each of our 10 participants
  - I provided 3 different ways of doing this
- Examined the number of accepted and rejected trials, ERP waveforms, and data quality measures for each participant to check for problems
  - EEGLAB > ERPLAB > Summarize artifact detection > Summarize ERP artifacts in a table (or type ERP.ntrials in the Matlab command window)
  - EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms
  - EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table
- Made a grand average from our 10 participants
  - EEGLAB > ERPLAB > Average across ERPsets (Grand Average)
- Applied a low-pass filter to the grand average to attenuate the high-frequency noise
  - EEGLAB > ERPLAB > Filter & Frequency Tools > Filters for ERP data
- Obtained N400 amplitude "scores" (mean voltage from 300-500 ms) for the related and unrelated targets in each participant's averaged ERP waveforms
  - EEGLAB > ERPLAB > ERP Measurement Tool
  - Used the Viewer to see the measurements for each waveform
- Performed a simple statistical analysis with the data from a single channel and a more complex analysis with the data from multiple channels
  - The N400 amplitude scores were exported into a text file and then imported into a statistical package
    - ERPLAB does not perform statistical analyses—we did not want to "reinvent the wheel," and it is difficult to anticipate every possible statistical analysis someone would want to perform
  - We also compared the table of means from the statistical analyses with our grand average ERP waveforms to make sure that the analysis was performed correctly
- Created a new "cluster" channel that was an average of 9 of the original channels
  - EEGLAB > ERPLAB > ERP Operations > ERP Channel operations
- Created a difference wave by subtracting one bin from another
  - EEGLAB > ERPLAB > ERP Operations > ERP Bin operations

This page titled 3.12: Review of Processing Steps is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.13: Matlab Scripts For This Chapter

I've provide two scripts in the data folder for this chapter. One is called **preprocess_and_average.m**, and it does the preprocessing and averaging for all 10 participants. The other is called **postprocessing.m**, and it carries out all the remaining steps in this chapter. These scripts are reasonably simple, and I've put lots of explanatory comments in them, so they're a good way to start learning scripting. Give them a try!

This page titled 3.13: Matlab Scripts For This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

3.14: Key Takeaways and References

Key Takeaways

- There are many steps between putting electrodes on the participant's head prior to the recording session and seeing the *p* value from the statistical analysis after the whole study is complete. As a result, there are many opportunity for some kind of problem or error that can lead you to draw the wrong conclusion from the study. As a result, it's crucially important that you constantly check for problems and errors.
- Some problems can be detected during preprocessing and averaging. For example, the response button may have been broken, leading to no event codes for responses. Or the participant's blinks may have been smaller than typical, causing many blinks to escape the artifact detection procedure. You can often catch these errors by checking the number of rejected and accepted trials and by examining the continuous and epoched EEG data.
- The process of scoring the amplitude or latency of an ERP component can also go awry for some participants, so you should use the Viewer tool to examine the scores alongside the waveforms for each participant.
- Once you've entered the scores into a statistical analysis, you should compare the descriptive statistics from your statistical package with the grand average waveforms to make sure that the analysis was performed correctly. It's really easy to accidentally put the scores in the wrong order.

References

Cousineau, D. (2005). Confidence intervals in within-subjects designs: A simpler solution to Loftus and Masson's method. *Tutorials in Quantitative Methods for Psychology*, *1*, 42–45.

Cramer, A. O. J., van Ravenzwaaij, D., Matzke, D., Steingroever, H., Wetzels, R., Grasman, R. P. P. P., Waldorp, L. J., & Wagenmakers, E.-J. (2015). Hidden multiplicity in exploratory multiway ANOVA: Prevalence and remedies. *Psychonomic Bulletin & Review*, *23*, 640–647.

Farrens, J. L., Simmons, A. M., Luck, S. J., & Kappenman, E. S. (2019). Electroencephalogram (EEG) Recording Protocol for Cognitive and Affective Human Neuroscience Research. *Protocol Exchange*. https://doi.org/10.21203/rs.2.18328/v3

Frane, A. V. (2021). Experiment-Wise Type I Error Control: A Focus on 2 × 2 Designs. *Advances in Methods and Practices in Psychological Science*, *4*(1), 2515245920985137. https://doi.org/10.1177/2515245920985137

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Luck, S. J., & Gaspelin, N. (2017). How to get statistically significant effects in any ERP experiment (and why you shouldn't). *Psychophysiology*, *54*, 146–157.

Luck, S. J., Stewart, A. X., Simmons, A. M., & Rhemtulla, M. (2021). Standardized Measurement Error: A Universal Measure of Data Quality for Averaged Event-Related Potentials. *Psychophysiology*.

Morey, R. D. (2008). Confidence intervals from normalized data: A correction to Cousineau (2005). *Tutorials in Quantitative Methods for Psychology*, *4*, 61–64.

This page titled 3.14: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4: FILTERING THE EEG AND ERPS

Learning Objectives

In this chapter, you will learn to:

- Compare the frequency content of an ERP waveform with the frequency response function of a filter to predict how well the filter will attenuate the noise in the data
- Determine the *impulse response function* of a filter and conceptualize filtering as a process that replaces each point in the unfiltered waveform with a scaled copy of this function
- Think of an ERP waveform as a series of *impulses*, one at each time point
- Predict how a filter will distort an ERP waveform on the basis of the filter's impulse response function
- Select filter parameters that provide the best balance between noise reduction and distortion of the waveform
- Create artificial waveforms and filter them to see how a filter might be distorting your data

You *must* use filters in ERP experiments. At a minimum, your amplifier includes an *antialiasing* filter that must be used prior to digitizing the EEG. In almost all ERP experiments, additional filtering is important for reducing sources of noise that would otherwise create large measurement error and reduce your statistical power. However, when filters are misused, they can dramatically distort your data, leading to incorrect conclusions. As a result, it's vitally important that you understand how filters work and the conditions under which they can produce significant distortion of your ERP waveforms.

For most ERP researchers, there is no topic more boring than filtering. At the core of filtering is a mathematical operation called *convolution*. Even the word "convolution" sounds complicated and boring!

However, you can get a reasonable understanding of filtering by seeing how convolution works visually, without ever seeing an equation. This chapter takes you through a set of exercises that will show you how convolutions are used for filtering without any equations. If you want a more detailed description of filtering, you should read Chapter 7 in Luck (2014). If you want to understand the math, you can read Chapter 12 in Luck (2014), which is available for free online.

4.1: Data for this Chapter

4.2: Classes of Filters

4.3: Exercise- Assessing the Frequency Content of the Noise

4.4: Exercise- Filtering the Artificial Waveforms

4.5: Exercise- The Impulse Response Function

4.6: Exercise- Applying the Impulse Response Function to a Series of Impulses

4.7: Background- Filtering with a Running Average

4.8: Background- Filtering with a Weighted Running Average

4.9: Exercise- Distortion of Onset and Offset Times by Low-Pass Filters

4.10: Exercise- High-Pass Filtering

4.11: Practical Advice

4.12: Exercise- Creating and Importing Artificial Waveforms

4.13: Matlab Script for this Chapter

4.14: Key Takeaways and References

This page titled 4: Filtering the EEG and ERPs is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.1: Data for this Chapter

The data we'll be using for the exercises in this chapter can be found in the Chapter_4 folder in the master folder: https://doi.org/10.18115/D50056.

All of the exercises in this chapter involve artificial data rather than real EEG or ERP signals. This is because we don't know the true waveform with real data. With real data, the waveform consists of the sum of an unknown ERP waveform and unknown noise, so when you apply a filter, you don't know what the result should look like if the filter is working properly. With artificial data, we can create a true waveform and add known noise to it. We can then see how well we can recover the true waveform by filtering the data. In other words, artificial waveforms give us *ground truth*.

Once you understand how filters work, they're pretty easy to implement using ERPLAB. You've already seen how to filter both EEG and ERP data in the previous chapters, so this chapter will focus on helping you understand how filters work rather than applying them to real data. All of the exercises use ERPsets rather than EEG datasets, but the general principles are the same for EEG and averaged ERPs.

This page titled 4.1: Data for this Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.2: Classes of Filters

There are four main classes of filters used in EEG/ERP research. They're typically named in terms of the frequencies that they pass, not the frequencies that are filtered out (much as an air filter passes air and filters out dust).

- A *low-pass* filter passes low frequencies and filters out high frequencies (e.g., muscle activity).
- A *high-pass* filter passes high frequencies and filters out low frequencies (e.g., gradual drifts resulting from skin potentials).
- A *bandpass* filter passes an intermediate band of frequencies and filters out the lower and higher frequencies. A bandpass filter is the same as filtering twice, once with a low-pass filter and once with a high-pass filter.
- A *notch* filter passes all frequencies except for a narrow band (e.g., 60 Hz). A notch filter is typically used during an EEG recording when AC electrical devices produce so much contamination of the EEG that it's hard to see the signal.

I don't ordinarily recommend applying notch filters (unless they are necessary during the recording process); it's usually better to use a low-pass filter that attenuates all the high frequencies. However, if you don't want to use a low-pass filter with a cutoff of 20 or 30 Hz (e.g., because you are interested in relatively high-frequency activity), a very sophisticated line noise filtering approach (Mitra & Pesaran, 1999) is available in EEGLAB as the *cleanline* plugin (see Bigdely-Shamlo et al., 2015 for important details about implementing this tool). Another tool called Zapline is can also be used for this purpose (de Cheveigné, 2020; Klug & Kloosterman, 2022), but it is newer and hasn't yet accumulated a strong track record.

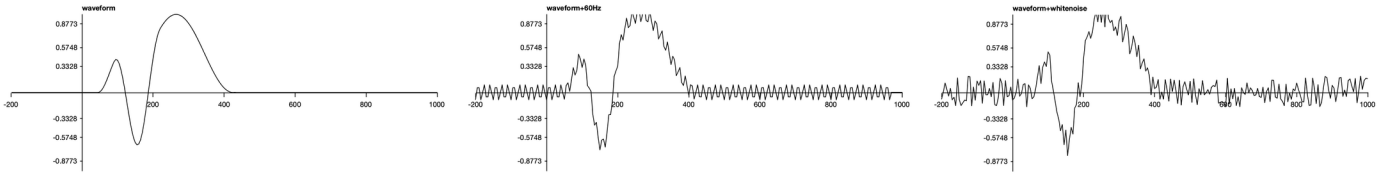
This page titled 4.2: Classes of Filters is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.3: Exercise- Assessing the Frequency Content of the Noise

We're going to start by thinking about filtering as a frequency-domain operation, in which we suppress some frequencies and pass others. If you don't already know how filtering works in the frequency domain, I recommend that you read the first 10 pages in Chapter 7 of Luck (2014) before you go any further.

If EEGLAB is running, quit it and restart it so that everything is fresh. Set **Chapter_4** to be Matlab's current folder. Load the ERPset file named **waveforms.erp** (**EEGLAB > ERPLAB > Load existing ERPset**) and plot the waveforms (**EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms**). It should look something like Screenshot 4.1.

Screenshot 4.1



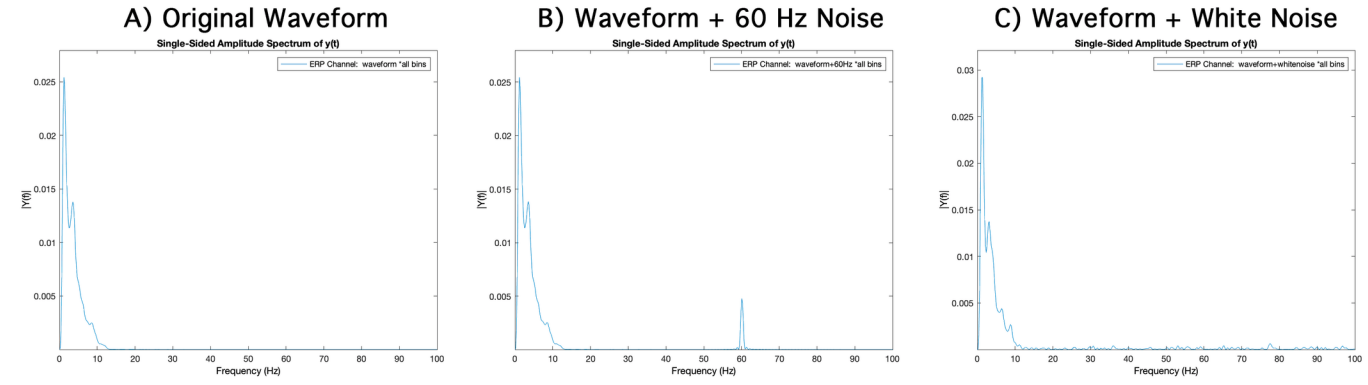
You can see that we have three channels. I created the waveforms in Excel. The first channel is an artificial waveform that I created by summing together three simulated ERP components, each of which was one cycle of a cosine function. The second channel is the sum of the first channel and a 60 Hz sine wave (like the line noise that is often picked up from electrical devices in the recording environment). The third channel is the sum of the first channel and some random noise (similar to the noise that is produced by tonic muscle activity and picked up by our EEG electrodes).

Line Noise

AC electrical lines run at 60 Hz in North America and some other parts of the world. Other regions use 50 Hz. We often call this the *line frequency* to be agnostic about whether it is 50 or 60 Hz. The noise produced by this signal is called *line noise*.

Before we filter the data, let's perform a Fourier transform on these waveforms so that we can see their frequency content. To do this for the original waveform, select **ERPLAB > Filter & Frequency Tools > Plot amplitude spectrum for ERP data**. In the window that pops up, specify **channel 1** and **bin 1**. For the **Frequency range to plot**, set **F1** to **0** and **F2** to **100**. You should see something like Screenshot 4.2.A. The X axis is the frequency, and the Y axis is the amplitude at this frequency. This plot tells us that we could reconstruct the original time-domain waveform by summing together a set of sinusoids with the set of amplitudes shown at each frequency in the plot. We'd also need to know the phase at each frequency to reconstruct the original waveform, but phase information isn't usually shown with ERP data. You can get a quick introduction to the Fourier transform in Chapter 6 of my online Introduction to ERPs course (or just watch this YouTube video). You can find a more detailed treatment in Chapters 7 and 12 of Luck (2014).

Screenshot 4.2



As you can see from the plot, the original ERP waveform mostly consists of relatively low frequencies. This is fairly typical of the waveforms you would see in most perceptual, cognitive, and affective experiments. In low-level sensory experiments, you might see more high-frequency activity.

Now repeat the process with Channel 2, which should produce something like Screenshot 4.2.B. It's the same as the amplitude spectrum for the original waveform, except that there is also activity at 60 Hz. This is because I created Channel 2 by summing together the original waveform and a 60-Hz waveform. Now do the same thing for Channel 3. As shown in Screenshot 4.2.C, you can see some activity at all frequencies. Also, the low-frequency activity is slightly different from the original waveform, because the noise extends down to these frequencies. This broad band of frequencies occurred because I added *white noise* to the original waveform, and white noise consists of equal amount of all frequencies (just as white light consists of approximately equal amounts of all wavelengths in the visible spectrum).

When you're first starting out in ERP research, you should plot Fourier transforms like these prior to filtering so that you have a good idea of the frequency content of the noise in your data. This can help you figure out where the noise is coming from (because different sources of noise have different frequency content). By knowing where the noise is coming from, you may be able to eliminate it in future recordings. It's better to reduce the noise before it contaminates your data rather than relying on filters and other signal processing techniques. For reasons described in Luck (2014), I call this *Hansen's Axiom*: "There is no substitute for clean data." As we will see later in this chapter, filters reduce the temporal precision of your data. And isn't temporal precision one of the most important features of the ERP technique?

This page titled 4.3: Exercise- Assessing the Frequency Content of the Noise is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

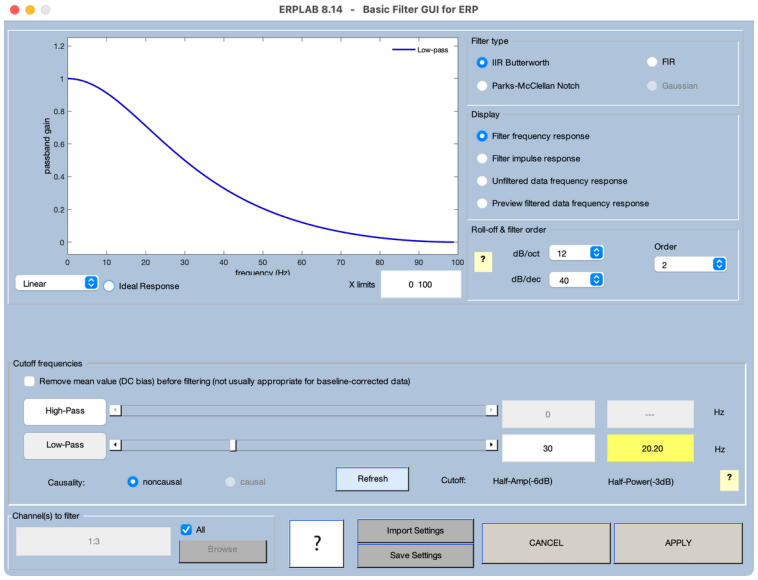
4.4: Exercise- Filtering the Artificial Waveforms

Now let's apply a simple filter to the artificial data from the previous exercise. Load **waveforms.erp** if it isn't already loaded, and select **ERPLAB > Filter & Frequency Tools > Filters for ERP data**. Set the parameters as shown in Screenshot 4.3. Most importantly, **High-Pass** should be unselected and **Low-Pass** should be selected with a cutoff at **30** Hz. For the big white plotting window, set the X limits to **0 100**. Make sure that the function shown in this window matches what is shown in the screenshot.

The function shown in the plotting window is called the *frequency response function* of the filter. It tells you the *gain* that will be applied to each frequency. The gain is just a multiplicative value: A value of 1 means that the amplitude of that frequency will be multiplied by 1 (i.e., unchanged). A value of 0.75 means that the frequency will be multiplied by 0.75, which means that it will be attenuated by 25%. The half-amplitude cutoff of 30 Hz that we specified means that the gain of the filter is 0.5 at 30 Hz; this is the point at which the gain is reduced by half (which is why it's called the *half-amplitude* frequency).

The filter we've specified has a fairly gentle roll-off of 12 dB/octave. As a result, even though it's nominally a 30 Hz filter, the gain is still well above zero at 60 Hz, and there is some significant attenuation as low as 10 Hz. Change the roll-off to **48** dB/octave, and you'll see a frequency response function that *appears* to be better. The gain is now near 1 for everything below about 20 Hz, and it's near zero for everything above about 45 Hz. However, as we'll see later in this chapter, this sharper frequency response function means that the filter produces more distortion in the time domain.

Screenshot 4.3



Set the roll-off back to **12** dB/octave, click **APPLY**, and then name the new ERPset **waveforms_30Hz_12dB**. Now plot the filtered waveforms. You'll see that the 60 Hz oscillation is now mostly (but not completely) eliminated from Channel 2. The fact that it's not completely eliminated makes sense given that the gain at 60 Hz for this filter is around .1 (as shown in the frequency response function), so about 10% of the 60 Hz noise remains after filtering. Channel 3 is now a little smoother, but it still has quite a bit of noise. This makes sense given the amplitude spectrum that we looked at in the previous exercise (Screenshot 4.2.C), in which there was substantial noise at frequencies below 30 Hz.

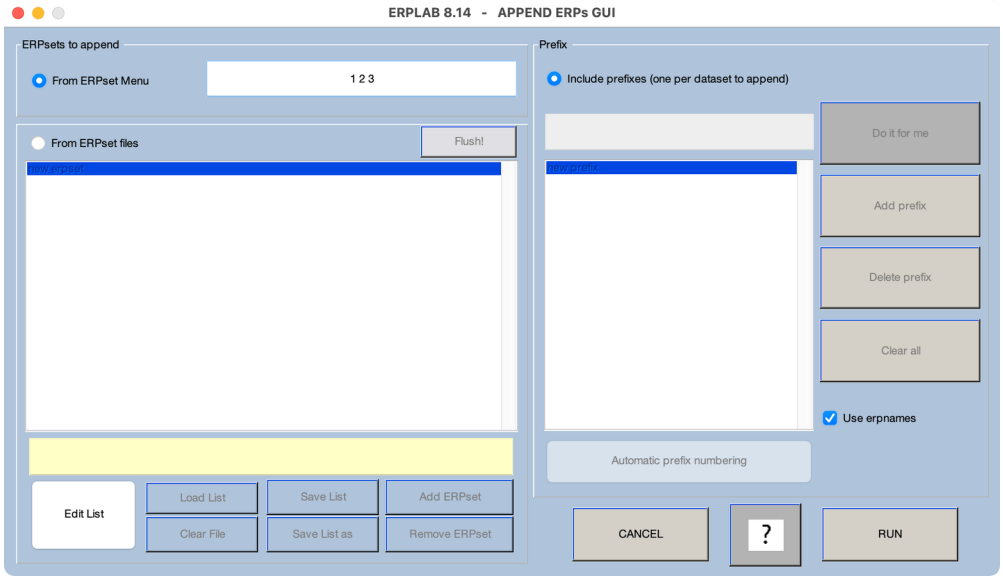
Now let's try a more severe filter. Make the original unfiltered ERPset active (by selecting it in the **ERPsets** menu) and filter it just as before, except set the low-pass cutoff to **10** Hz and set the roll-off to **48** dB/octave. Click **APPLY** and then name the new ERPset **waveforms_10Hz_48dB**. If you plot the filtered waveforms, you'll see that the 60 Hz line noise is almost completely gone from Channel 2 and that the noise in Channel 3 has been significantly reduced (but with some lower-frequency fluctuations in amplitude still visible).

To precisely compare two waveforms, it really helps to overlay them on the same plot. Unfortunately, ERPLAB's **Plot ERP waveforms** tool is designed to overlay different bins from a single ERPset and can't overlay waveforms from different ERPsets. However, there is a trick for solving this problem: We can append multiple ERPsets together into a single ERPset, with different

bins for the data from each original ERPset. Let's append the ERPsets from the original data, the data filtered at 30 Hz, and the data filtered at 10 Hz.

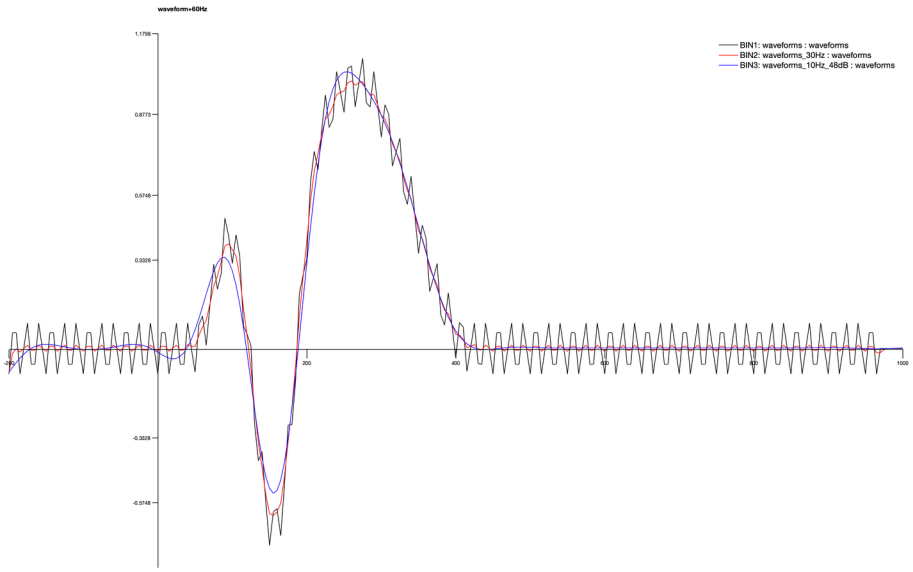
Select **EEGLAB > ERPLAB > ERP Operations > Append ERPsets**. In the window that pops up, select **From ERPsets Menu** and put **1 2 3** into the corresponding text box to indicate that you want to append the first three ERPsets. (This assumes that you just have the three relevant ERPsets loaded in the ERPsets menu; change the numbers as needed if you have other ERPsets loaded). Check the **Use erpnames** box so that it names each bin in the appended file with the names of the ERPsets that are being appended together. See Screenshot 4.4 (you'll need to select the **Include prefixes** button before your window will look like the screenshot).

Screenshot 4.4



Click **RUN** and then name the new ERPset **appended_waveforms**. Now plot the new ERPset, making sure that all three bins are being plotted. You'll see that the original waveforms are plotted as Bin 1, the waveforms filtered at 30 Hz are plotted as Bin 2, and the waveforms filtered at 10 Hz are plotted as Bin 3. These waveforms are highly overlapping, so you'll need to zoom in to see how they differ. To do this, single-click on any of the waveforms or the channel label for Channel 2 (**waveform+60Hz**) and a new window will pop up that shows just this channel. Matlab is a little fussy about this feature, so you may need to click a few times to get the click to register (but don't double-click—you'll end up with two identical windows). You should see something like Screenshot 4.5.

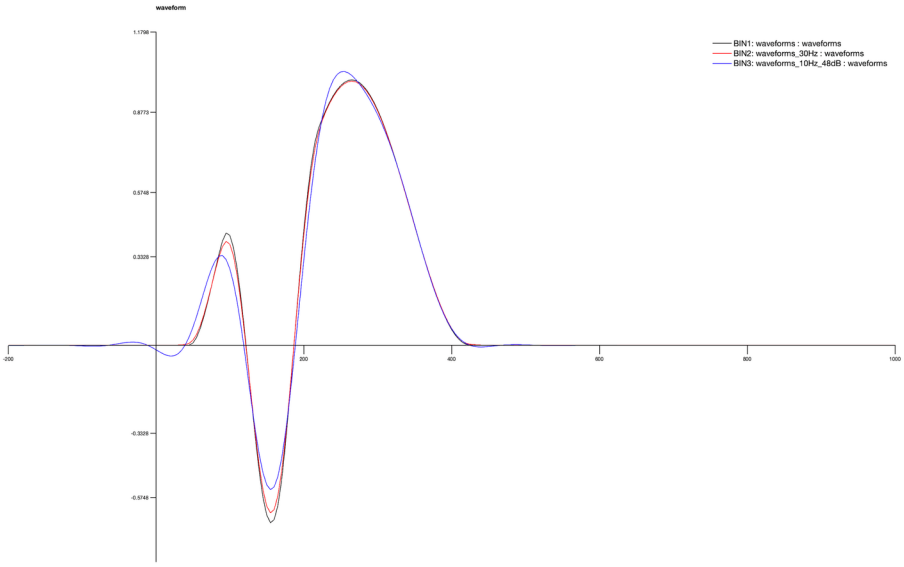
Screenshot 4.5



You'll see that the three waveforms are pretty similar except that the 60-Hz oscillations are clearly present in the original waveform, mostly but not completely attenuated by the 30-Hz filter, and completely eliminated by the 10-Hz filter. If you look closely, you'll also see that the first two peaks in the waveform (analogous to the P1 and N1 waves) are reduced in amplitude by the 10-Hz filter. This makes sense, because these peaks contain substantial power near 10 Hz, which is now being attenuated.

It's easier to see how the filter is impacting the ERP waveform by looking at the original waveform, without any noise. Go back to the plot with all three channels and click on Channel 1 (**waveform**) to zoom in. You should see something like Screenshot 4.6. The first two peaks are slightly attenuated by the 30-Hz filter and more clearly attenuated by the 10-Hz filter. You can also see that the 10-Hz filter produces a little overshoot in the third peak (which is like a P2 wave) and makes the first peak onset earlier.

Screenshot 4.6



The 10-Hz filter also produces a small artificial negative peak just after time zero and before the first positive peak. This is not because the filter is at 10 Hz; it's because we used a very steep roll-off (48 dB/octave). You can verify this for yourself by going back to the original unfiltered data and filtering it with a 10 Hz cutoff but a slope of 12 dB/octave. You'll see that the artificial negative peak is no longer present.

Now you can see why a steep frequency response function—which seems ideal when you're focused on frequency information—is not usually a good idea for ERP research. A steep roll-off can really distort the waveform, producing artificial peaks. And this can lead to wildly incorrect conclusions. For example, Darren Tanner, Kara Morgan-Short, and I wrote a paper several years ago (Tanner et al., 2015) showing that inappropriate filtering can make a P600 effect (which is usually a result of a syntactic violation in a language experiment) look like an N400 (which is usually a result of a semantic anomaly). If you learn only one thing from this chapter, I hope it's that you should use only very mild filters unless you really know what you're doing (e.g., you fully understand all the equations in Chapter 12 of Luck, 2014). I provide some specific recommendations near the end of the chapter that will allow you to avoid drawing bogus conclusions as a result of inappropriate filtering.

This page titled 4.4: Exercise- Filtering the Artificial Waveforms is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.5: Exercise- The Impulse Response Function

Most discussions of filters focus on their frequency response functions, which indicate the effects of the filter in the frequency domain. But do you actually care about the frequency content of your ERP waveforms? Probably not. If you're interested in conventional ERP waveforms (as opposed to time-frequency analyses), then you probably want to know how filters change your data in the time domain, not in the frequency domain.

Filters can be implemented either in the frequency domain (using the Fourier transform) or in the time domain (using convolutions). These two approaches yield exactly the same results, but I find that the time domain implementation makes it easier to understand exactly how a filter changes an ERP waveform. So, we'll mainly focus on the time domain for the remainder of this chapter. We'll start with an exercise designed to help you understand time-domain filtering visually, without any math. There are many types of filters, and I'm going to focus on a common class called *finite impulse response* filters, even though this ends up being a slight oversimplification for the filters implemented in ERPLAB.

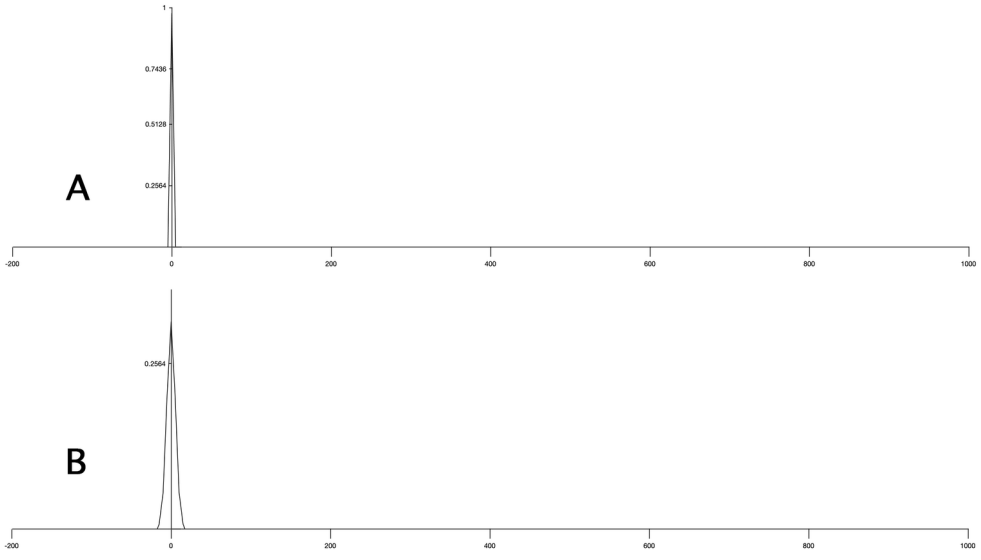
The key to understanding filtering in the time domain is to understand something called the *impulse response function*. A filter's impulse response function is simply the output of the filter when the input is an impulse of amplitude 1 at time zero. (An impulse is a waveform that is zero everywhere except for a nonzero value at a single time point). To see what I mean, quit and restart EEGLAB, load the ERPset file named **impulse0.erp**, and plot the ERP waveform. When you plot this example (and the remaining examples in this chapter), make sure you set **Baseline Correction** to **None** in the GUI for plotting ERP waveforms (see the box below if you want to know why this is necessary).

Baseline Correction

The waveform in **impulse0.erp** has a value of 1 at time zero and a value of 0 everywhere else. The value of 1 at time zero messes up the baseline when you try to plot the waveform. This is because the baseline is defined as the average of the period up to **and including** time zero. This average is slightly greater than zero, and baseline correction involves subtracting the average from every point in the waveform. Thus, the whole waveform ends up being shifted slightly downward

Once you've turned off the baseline correction, you should see something like Screenshot 4.7.A when you plot the ERPset.

Screenshot 4.7



This ERPset contains a single channel in a single bin, and you can see the impulse (a voltage of 1 μV) at time zero. It looks like a narrow triangle rather than a pure impulse because the waveform is sampled at 200 Hz (one sample every 5 ms), so there is a line going from 0 μV at -5 ms to 1 μV at 0 ms and back down to 0 μV at 5 ms.

Now filter this waveform using a half-amplitude cutoff at 30 Hz and a slope of 12 dB/octave (following the same steps you used in the previous exercise) and plot the result (which should look like Screenshot 4.7.B). You can see that the filtered waveform is now a little wider and peaks at a lower amplitude (approximately 0.32 μV). This filtered waveform is the impulse response function of the filter (i.e., the waveform produced by filtering an impulse of amplitude 1 at time zero).

You don't actually need to filter an impulse to see the impulse response function in ERPLAB. You can also see it by going to the window for the filtering routine and changing **Display** from **Filter frequency response** to **Filter impulse response**. As you can see from Screenshot 4.8, the impulse response function is now plotted. Only the right half of the function is shown, but the left half is just the mirror image. Note that it peaks at approximately 0.32 μV, just like the waveform you created by filtering an impulse (Screenshot 4.7.B). The time scale is expanded, so it's easier to see the details of the waveform.

Screenshot 4.8



At this point, you're probably wondering, "Why should I care what the output of a filter looks like when the input is an impulse? That impulse doesn't look much like an ERP waveform." You should care because the key to understanding filtering is that an ERP waveform is a sequence of voltages, one at each time point, and you can think of this as a sequence of impulses of different amplitudes. By knowing what the filter's output looks like for an impulse at one time point (i.e., the impulse response function), you can know what the filter's output will look like for the whole waveform. This is demonstrated in the next exercise.

A Slight Oversimplification

In this chapter, I discuss how finite impulse response (FIR) filters work, because they are quite easy to understand. However, ERPLAB implements filtering using a specific type of infinite impulse response (IIR) filter called a Butterworth filter. As long as you use a shallow roll-off (e.g., 12 dB/octave), ERPLAB's filters provide a close approximation of a FIR filter. So, everything I say in this chapter is approximately correct for ERPLAB's filters as long as you use a shallow roll-off.

The key difference between FIR and IIR filters is that the output of an IIR filter feeds back into the filter's input. This means that the filter is nonlinear, with a response that could theoretically extend infinitely in time. The main advantage is that IIR filters require fewer coefficients than FIR filters, making them run faster and potentially reducing edge effects (which will be described later).

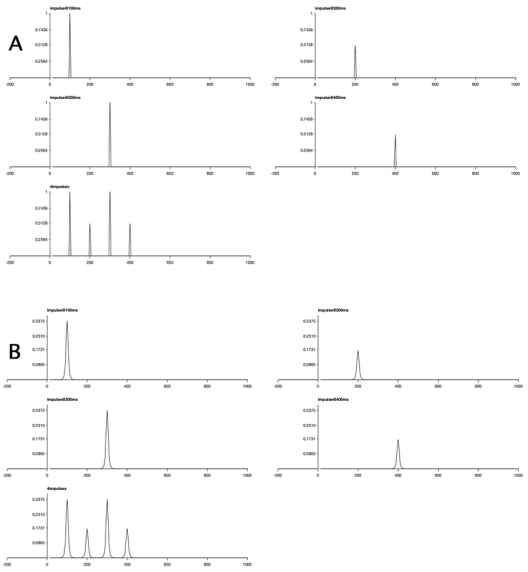
This page titled 4.5: Exercise- The Impulse Response Function is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.6: Exercise- Applying the Impulse Response Function to a Series of Impulses

For finite impulse response function filters, the output of the filter for a complex waveform is simply the sum of the filter's impulse response function for the voltages at each time point, scaled by the input amplitude at each time point. That's a pretty complicated sentence, so in this exercise we'll look at a couple simple examples.

You can start by loading and plotting the ERPset file names **impulse1.erp**. This ERPset contains one bin, and each channel has a different impulse in it. Channel 1 has an impulse of 1 μV at 100 ms. Channels 2, 3, and 4 have impulses at 200, 300, and 400 ms with amplitudes of 0.5, 1, and 0.5 μV, respectively. Channel 5 has all four of the impulses in it (see Screenshot 4.9.A).

Screenshot 4.9



Filter this ERPset using a half-amplitude cutoff at 30 Hz and a slope of 12 dB/octave (following the same steps you used in the previous exercises) and plot the result (which should look like Screenshot 4.9.B). Each impulse has been replaced by the impulse response function of the filter, shifted so that it is centered at the latency of the impulse, and scaled (multiplied) by the size of the impulse. Importantly, filtering the waveform with four impulses (Channel 5) gives you a waveform that is equivalent to the sum of the four filtered waveforms for the individual impulses (Channels 1-4). This shows you what I meant when I said that "the output of the filter for a complex waveform is simply the sum of the filter's impulse response function for the voltages at each time point, scaled by the input amplitude at each time point." That is, the output of the filter for the waveform with four impulses (Channel 5) is equivalent to replacing each individual impulse with a copy of the impulse response function that has been shifted to be centered on a given impulse and scaled by the amplitude of that impulse.

To make this even clearer, we're going to take the four impulses and make them consecutive sample points (just as an ERP waveform typically consists of a sequence of consecutive nonzero voltage values). Load the ERPset named **impulse2.erp** and plot it. You'll see that now our four impulses are at 100, 105, 110, and 115 ms, which are consecutive because we have a sampling rate of 200 Hz and therefore a voltage value every 5 ms. Filter this ERPset using a half-amplitude cutoff at 30 Hz and a slope of 12 dB/octave (just as before) and plot the result. Just as in the previous example, filtering the set of four consecutive impulses is equivalent to filtering each impulse separately and then summing them together. In other words, the filtered waveform is equivalent to replacing each impulse in the unfiltered waveform with a copy of the impulse response function, centered on each impulse and scaled by the height of each impulse.

In this exercise, we used impulses to create 4 time points in an ERP waveform. Figure 4.1 extends this idea to a more realistic ERP waveform. Panel A is the same artificial waveform shown at the beginning of the chapter in Screenshot 4.1, but blown up. We have a voltage value every 5 ms, and these voltage values are connected by lines to create a waveform. Panel B is the same set of voltage values, but with the voltage at each time point shown as an impulse. This is conceptually identical to the set of four

impulses shown in Screenshot 4.9.A, except now we have an impulse at each time point. To filter this waveform, we just replace each of these impulses with a copy of the impulse response function, centered at each impulse and scaled (multiplied) by the amplitude of the impulse. We then sum together these scaled copies of the impulse response function to obtain our filtered ERP waveform.

The process of replacing each point in one waveform with a scaled copy of another waveform is called *convolving* the two waveforms. So, filtering an ERP waveform is achieved by convolving the waveform with the impulse response function. It turns out that convolution is not as complicated (or convoluted) as it sounds!

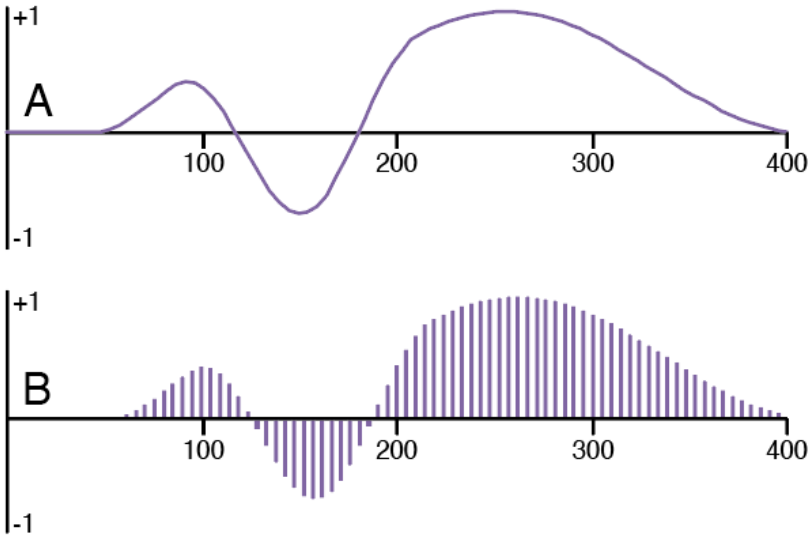
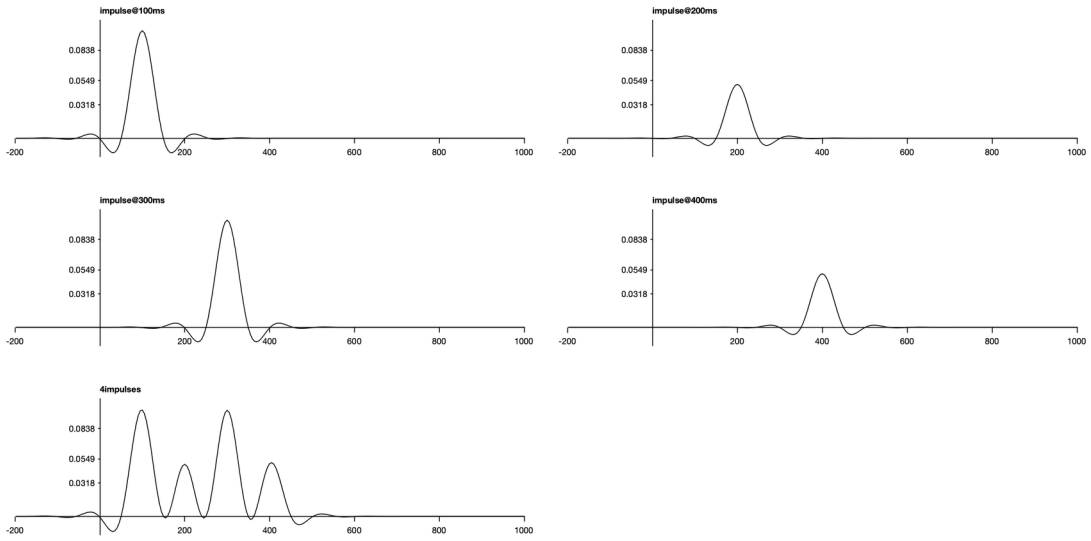


Figure 4.1. Two ways of drawing the same ERP waveform. The waveform consists of a sequence of discrete voltage values, one at each sample point (every 5 ms in this example). We typically connect these points with lines to make it look like a continuous waveform (A). However, we can also think of each sample point as an impulse going from zero to the voltage value at that point (B). We can then think of filtering as replacing each of these impulses with a copy of the impulse response function, scaled (multiplied) by the amplitude of the impulse.

I hope you can now see that filtering in the time domain is conceptually very simple as long as you know the impulse response function of the filter. That's why we designed the filtering tool in ERPLAB to show you this function. Many EEG/ERP analysis systems don't show you the impulse response function, but you can always figure it out by making a waveform that consists of a single impulse (like the one shown in Screenshot 4.7.A) and passing it through the filter.

By knowing the impulse response function, you can make a pretty good guess about how the filter might distort your data. For example, do you remember the artificial negative peak produced by the filter with the 10 Hz cutoff and 48 dB/octave roll-off (Screenshot 4.6)? That artificial peak makes perfect sense once you see the impulse response function of the filter. To see the impulse response function for this filter, load the **impulse1.erp** ERPset (or make it the active ERPset if it's already loaded) and filter it using a 10 Hz half-amplitude cutoff and a roll-off of 48 dB/octave. If you plot the filtered data, you'll see something like Screenshot 4.10.

Screenshot 4.10



Channel 1 shows you what the impulse response function looks like (but centered at 100 ms rather than 0 ms because the impulse was at 100 ms). It has a negative dip on each side of the peak. And when we filter the set of four impulses in Channel 5, we can see this dip just before the first positive peak. Now imagine what happens when you apply this filter to the more realistic waveform shown in Figure 4.1.A. This would involve replacing each point in the waveform with a scaled copy of the impulse response function. When we replace the positive impulses that start around 50 ms with this function, the negative part of the impulse response function generates the negative dip prior to 50 ms.

This sort of distortion is easy to understand when you think about filtering in the time domain using the impulse response function. However, the distortion is not at all obvious when you think about filtering using the frequency response function. That's why I prefer to think about filtering in the time domain. However, it's still useful to know the frequency response function, especially if you know something about the frequency content of the noise in your data. This is why we designed the ERPLAB filtering tool to provide you with both functions. Also, the frequency response function and impulse response function are very closely related: The frequency response function is simply the Fourier transform of the impulse response function, and the impulse response function is simply the inverse Fourier transform of the frequency response function.

This page titled 4.6: Exercise- Applying the Impulse Response Function to a Series of Impulses is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.7: Background- Filtering with a Running Average

I hope it's now clear that filtering works by replacing each point in the waveform with a scaled copy of the impulse response function. However, it's probably not obvious *why* this ends up filtering out the high frequencies. There's a slightly different—but mathematically equivalent—way of thinking about filtering that makes it more obvious.

Let's start by forgetting everything you know about EEG and filtering, and instead think about stock market prices. Figure 4.2.A shows the daily values of the Standard & Poor 500 Stock Index over a 3-month period. There are lots of day-to-day variations that are largely random and don't mean much for the overall economy. What would be an easy way to minimize these day-to-day fluctuations so that you could better visualize the overall trend?

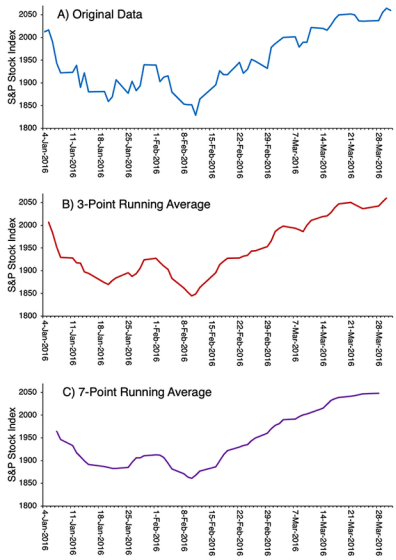


Figure 4.2. A) Standard & Poor 500 Stock Index daily values between January 4 and March 31 of 2016. B) Values after applying a 3-point running average. C) Values after applying a 7-point running average.

A common approach is to take a running average. Figure 4.2.B shows a 3-point running average of the values in Figure 4.2.A. Each value for a given day in the running average is just the average of the values from the day before, that day, and the day after. For example, the running average value on February 2 is the average of the values on February 1, 2, and 3. You can see that the running average is smoother than the original data.

We can make the data even smoother with a 7-day running average (Figure 4.2.C). Now, the running average for a given day is the average of the value on that day, the three days before, and the three days after. The more points we include in our running average, the more we attenuate rapid day-to-day changes and see the slower trends in the data. In other words, increasing the number of points in the running average increases the filtering of high frequencies in the data. So, taking a running average is a simple form of low-pass filtering, and we can control the cutoff frequency by changing the number of points being averaged together. We can apply this same algorithm to filter out high frequencies in the EEG or in ERPs (see Chapter 7 in Luck, 2014 for additional details).

Edge Artifacts

The running average approach to filtering exposes a problem that we always face in filtering, no matter what algorithm we use. The S&P 500 index values shown in Figure 4.2.A start on January 4 and end on March 31. To compute the 3-point running average value for January 4, we would need values for January 3, 4, and 5, but we don't have the value for January 3. Similarly, we can't calculate the running average value for March 31 because we don't have the value for April 1. Things are

even worse for the 7-point running average because now we need 3 days before and 3 days after a given day. As a result, we can't calculate the running average for January 4, 5, or 6 or for March 29, 30, or 31 with the data that are available to us. You can see that these points are missing from the running averages in Figure 4.2.

This problem is less obvious when we filter using impulse response functions or the Fourier transform, but the same problem is present for all filtering algorithms. We solve this problem in ERPLAB using an extrapolation algorithm to estimate the values for the points that are needed but unavailable. It works quite well in most cases, but it can lead to problems when too many points must be extrapolated. The most common situation where that arises is when we use a high-pass filter to filter out low frequencies from the continuous EEG, which requires a very large number of points before and after the current point. In this situation, we sometimes see "edge artifacts" at the beginning and end of the EEG waveforms. To avoid these edge artifacts, I recommend recording an extra ~10 seconds of data prior to the first stimulus at the beginning of each trial bock and another ~10 seconds after the last stimulus at the end of each trial block. That way, the edges of the waveforms are far from the period of time you care about, and the edge artifacts occur during a time period that is outside the epochs that you will use for averaging.

In addition, ERPLAB's filtering tool has an option that can help reduce edge artifacts. This option is labeled "Remove mean value (DC offset) before filtering". It should ordinarily be used when you are filtering continuous EEG data. However, it should not be used for baseline-corrected data (e.g., epoched EEG or averaged ERPs) because the baseline correction already eliminates the DC offset (and typically works better than removing the mean value).

This page titled 4.7: Background- Filtering with a Running Average is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.8: Background- Filtering with a Weighted Running Average

The advantage of the running average approach to filtering is that it's really easy to see why it reduces high-frequency fluctuations: Any little "uppies and downies" within the set of points being averaged together will cancel each other out. Imagine that we applied a running average filter to raw EEG, using a running average width of 50 ms (e.g., 5 points on either side of the current point if we have one point every 5 ms). If the EEG has a 20 Hz sine wave in it, there will be exactly one cycle of the sine wave in 50 ms, and the positive and negative sides of the sine wave will cancel each other out.

The disadvantage of the running average filter is that it's a bit crude. Imagine that we had a 101-point running average filter (the number is always odd because we have the current point plus an equal number of points on either side). The filtered value at time point *t* would be just as influenced by time point *t-50* as by point *t-1*. Instead of giving all 101 points equal weight, it would make more sense to give nearby points greater weight than more distant points. That's actually quite easy to do.

This *weighted running average* approach to filtering is illustrated in Figure 4.3. We start by defining a *weighting function*, which indicates how much weight each of the surrounding time points will have. For example, rather than having a 7-point running average in which each of the 7 points has equal weight, the 7-point weighting function shown in Figure 4.3 gives the greatest weight to the current point, and then the weights fall off for more distant points. The filtered value at time t is computed by taking each of the 7 points (t-3 through t+3), multiplying the unfiltered value at each point by the corresponding value from the weighting function, and the summing these weighted values.

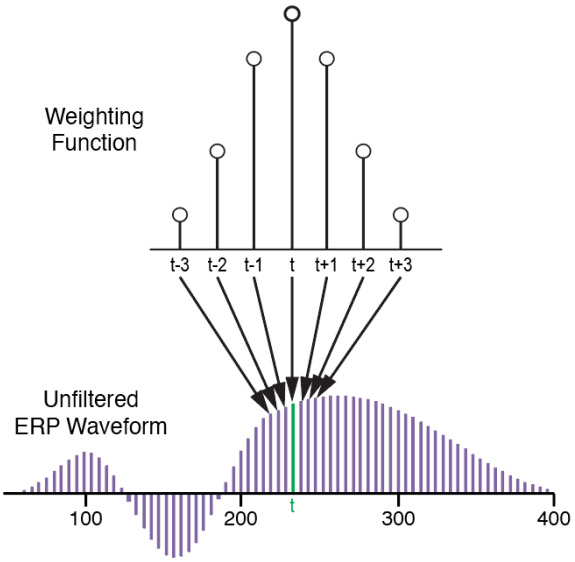


Figure 4.3. Filtering with a weighting function. To calculate the filtered value at time t, you take each of the surround points (t-3 through t+3), multiply the unfiltered value at each point by the corresponding value from the weighting function, and sum them together.

A standard 7-point running average could be computed in the same way. Our weighting function would have a value of 1/7 at each of the 7 points. The filtered value at time *t* would then be computed by taking 1/7 times the voltage at each of the 7 points and then summing these values together. That's identical to taking the average of the 7 points.

I hope that you can still see how this more sophisticated version of the running average filter will tend to attenuate high frequencies: Uppies and downies during the set of points being averaged together will tend to cancel each other out.

You may be wondering how this weighted running average approach is related to filtering with an impulse response function. The answer is beautifully simple: The weighting function is simply the mirror image of the impulse response function. And if the impulse response function is symmetrical (which is usually the case), the weighting function is exactly the same as the impulse response function. Also, because the weighting function is just the mirror image of the impulse response function, you can compute the frequency response function by applying the Fourier transform to the mirror image of the weighting function.

The only real difference between filtering with an impulse response function and filtering with a weighting function is conceptual:

the impulse response approach tells you how the unfiltered value at time *t* will influence the filtered values at the surrounding time points (because the unfiltered value at time *t* is replaced by a scaled copy of the impulse response function), whereas the weighted running average approach tells you how the filtered value at time *t* is influenced by the unfiltered values at the surround points (because the filtered value at time *t* is the weighted sum of the surrounding time points).

Thinking about filtering in terms of the weighting function should help you understand an important consequence of filtering: Filtering inevitably reduces temporal resolution (no matter how the filtering is implemented). The filtered value at a given time point is influenced by the surrounding time points, so the voltage you see at 100 ms is now influenced by the voltages at 95 ms, 105 ms, etc. And the weighting function shows you exactly how much impact the preceding and subsequent time points will have on the filtered value at a given time point. The wider the weighting function, the more you are filtering your data, and the more temporal precision you have lost. This is a very clear example of the principle that increasing the precision in the frequency domain (by limiting the set of frequencies that are in the filter output) decreases the precision in the time domain (by causing the filtered value at a given time point to be influenced by a wider range of time points from the unfiltered data).

This page titled 4.8: Background- Filtering with a Weighted Running Average is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.9: Exercise- Distortion of Onset and Offset Times by Low-Pass Filters

Now let's take a closer at how low-pass filters reduce temporal resolution. The general idea is that each point in the unfiltered waveform gets replaced by a scaled copy of the impulse response function, so the filtered data get "spread out" by the width of the impulse response function. Let's look at an example.

Load the ERPset named **peak1.erp** and plot it. You'll see that there are three identical channels, each with a single peak at 100 ms. We're going to leave Channel 1 unchanged and we're going to filter Channels 2 and 3 with different filter cutoffs. To do this, launch the filtering tool (**EEGLAB > ERPLAB > Filter & Frequency Tools > Filters for ERP data**) and set it to filter the data with a low-pass cutoff at 30 Hz and a roll-off of 12 dB/octave. Then, in the lower left corner of the window, change **Channel(s) to filter** to be **2** instead of having the **All** box checked. This will apply the filter only to Channel 2. Click **APPLY** and then specify whatever ERPset name you'd like. Now launch the filtering tool again, and change the cutoff to 10 Hz (leaving the roll-off at 12 dB/octave). Change **Channel(s) to filter** to be **3** instead of **2**, click **APPLY**, and use whatever ERPset name you'd like.

Plot the ERPs to see the effects of the filtering. Channel 1 has not been changed. Channel 2 has been filtered at 30 Hz, but it will look only slightly different from Channel 1 because this is a pretty minimal filter. Channel 3 has been filtered at 10 Hz, and if you look closely, you'll see that the waveform in Channel 3 onsets significantly earlier and offsets significantly later than the original waveform in Channel 1.

To make it easier to compare the waveforms, I overlaid them in Figure 4.4. You can see that the 30-Hz filter had almost no effect, but the 10-Hz filter caused the waveform to "spread out," making the onset earlier and the offset later. It also decreased the peak amplitude (because the original waveform had significant power in the frequencies around 10 Hz that has now been eliminated).

Now let's see how these effects can be explained by the impulse response function of the filter. Go back to the filtering tool, and check the **All** box for **Channel(s) to filter**. Leave the cutoff frequency at 10 Hz and change the **Display** setting near the top from **Filter frequency response** to **Filter impulse response**. Remember, only the right half of the impulse response function is shown; the left half is the mirror image of the right half. You can see that the impulse response function extends for about 30 ms. This means that any activity in the unfiltered waveform will be spread approximately 30 ms both leftward and rightward.

A Frustrating Moment

I forgot to set **Channel(s) to filter** back to **All** the first time I ran through this exercise. The next time I filtered an ERP, it seemed like the filtering wasn't working because only one channel was being filtered. I tried restarting ERPLAB, and that didn't help. I tried restarting Matlab, and that didn't help either. I was getting frustrated and was about to reset ERPLAB's working memory (which would have helped, because it would have reset all the filtering options to their defaults), but then I noticed that the **All** box wasn't checked. I checked it, and then everything started working as expected. This is just one of many times that I ran into a problem while creating the exercises in this book and used the troubleshooting steps described in Appendix 1. The moral of the story is that even the guy who oversaw the design of ERPLAB and has decades of experience with analyzing ERPs runs into problems from time to time!

Now change the half-amplitude cutoff from 10 Hz to 30 Hz and look at the impulse response function. It now declines to near zero within 10 ms. This means that the spreading produced by this filter will be less than ~10 ms. Note that the visual appearance of the spreading will depend on the shape of the waveform. For example, the spreading in Figure 4.4 looks more than 3 times greater for the 10 Hz filter than for the 30 Hz filter. However, you could see that this what is happening if you filtered an impulse. You could also see the 10-ms spreading for the 30 Hz filter if you zoomed in sufficiently closely.

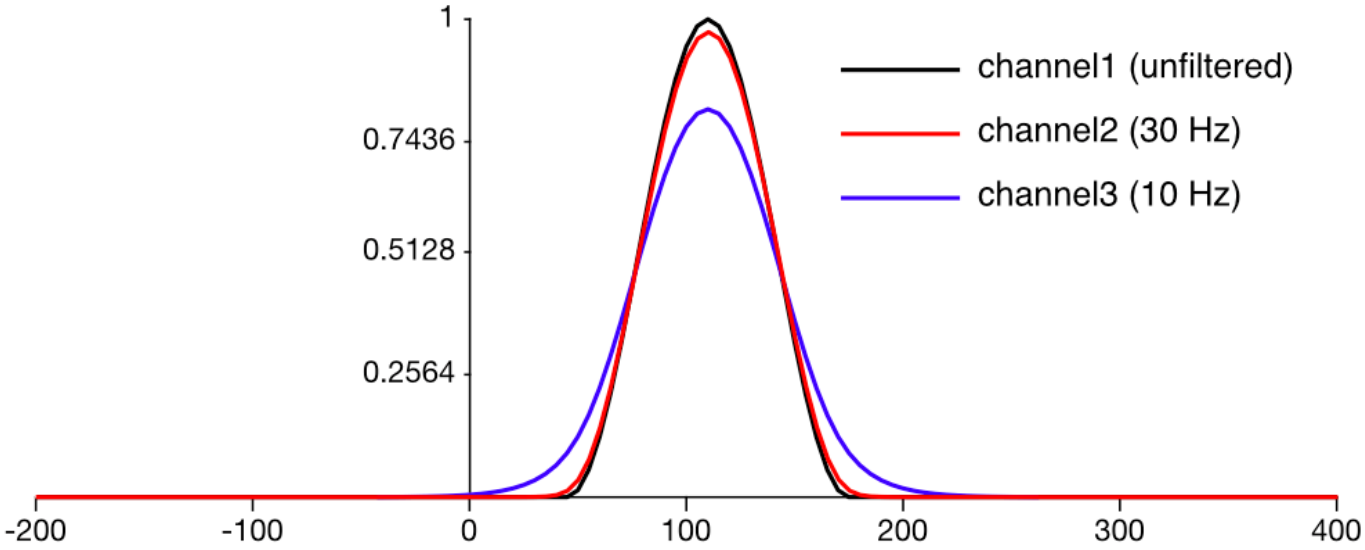


Figure 4.4. Artificial waveform without filtering, with a 30-Hz half-amplitude cutoff, and with a 10-Hz half-amplitude cutoff.

This page titled 4.9: Exercise- Distortion of Onset and Offset Times by Low-Pass Filters is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

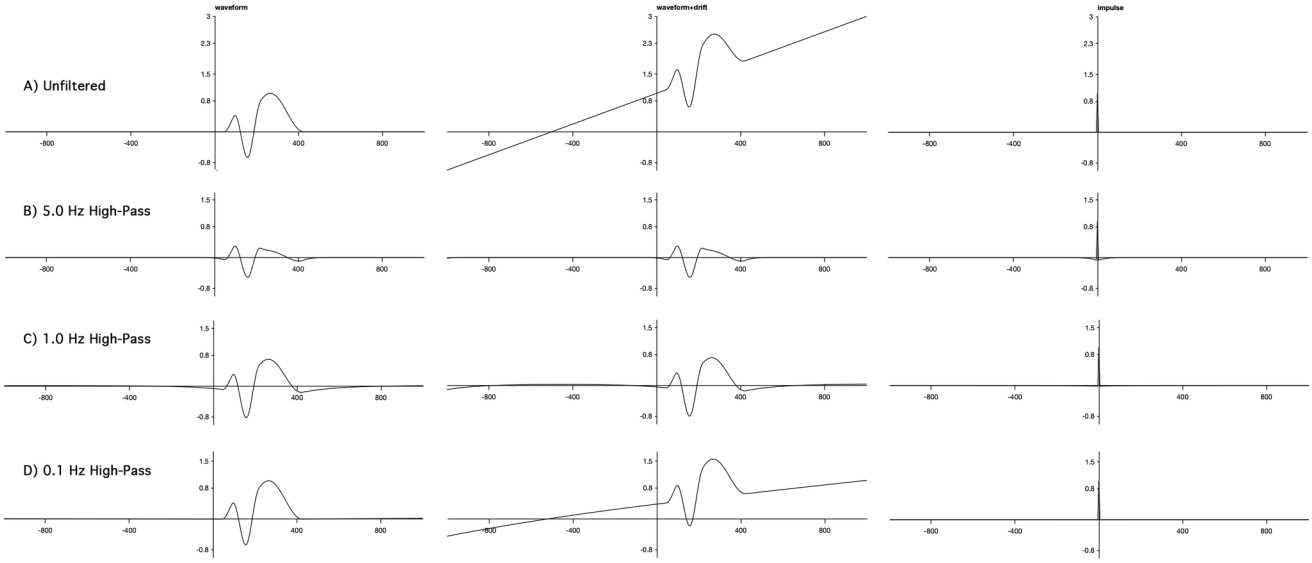
4.10: Exercise- High-Pass Filtering

Up to this point, we've focused on low-pass filters, but in this exercise we'll see how high-pass filtering works. The only fundamental difference is that the impulse response functions are different.

With typical parameters (e.g., a 0.1 Hz half-amplitude cutoff frequency), high-pass filtering requires a very long impulse response function, so it doesn't work very well with epoched EEG or ERP data (unless the epochs are many seconds long). Ordinarily, you'll apply high-pass filtering to continuous EEG (as in some of the exercises in earlier chapters). However, it's a little easier to create simulated waveforms and visualize them with ERPs, so we'll apply high-pass filters to simulated ERP data in this exercise. As a compromise, I created simulated waveforms with a longer-than-usual epoch (from -1000 to +1000 ms). But with real data, you'd want even longer epochs (or, better yet, apply high-pass filters to the continuous EEG data).

To get started, quit and restart EEGLAB and then load the ERPset in the file named **waveform+drift.erp**. If you plot the ERPs, you'll see something like Screenshot 4.11A. Channel 1 is the same artificial waveform we've used before, but with a longer epoch. Channel 2 is the same waveform, but with a linear drift superimposed on it. Channel 3 is an impulse that we can use to visualize the impulse response functions of the filters we'll be using.

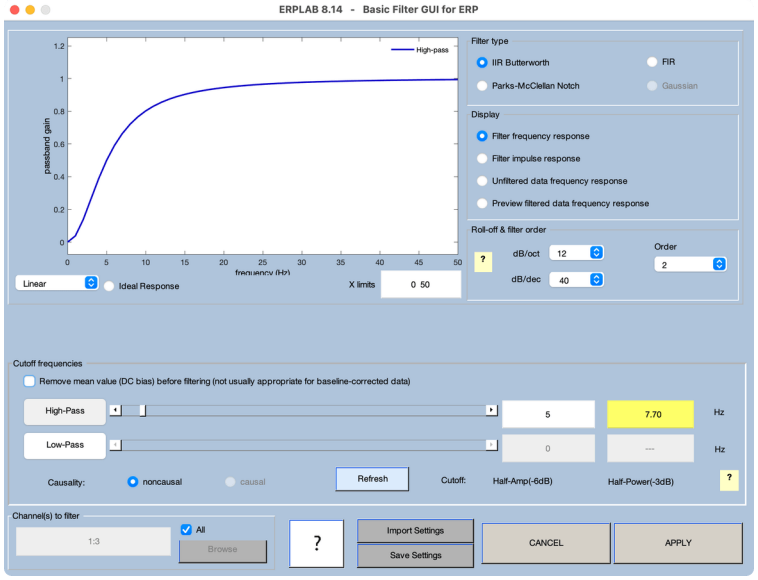
Screenshot 4.11



We're going to start by using a high-pass filter cutoff of 5 Hz, which means that we're filtering out frequencies below 5 Hz. I'd never recommend using this cutoff with real data (except for a few special cases, such as research on the auditory brainstem response). However, this will make it easier to see what the impulse response function looks like and how it impacts the filtered waveform.

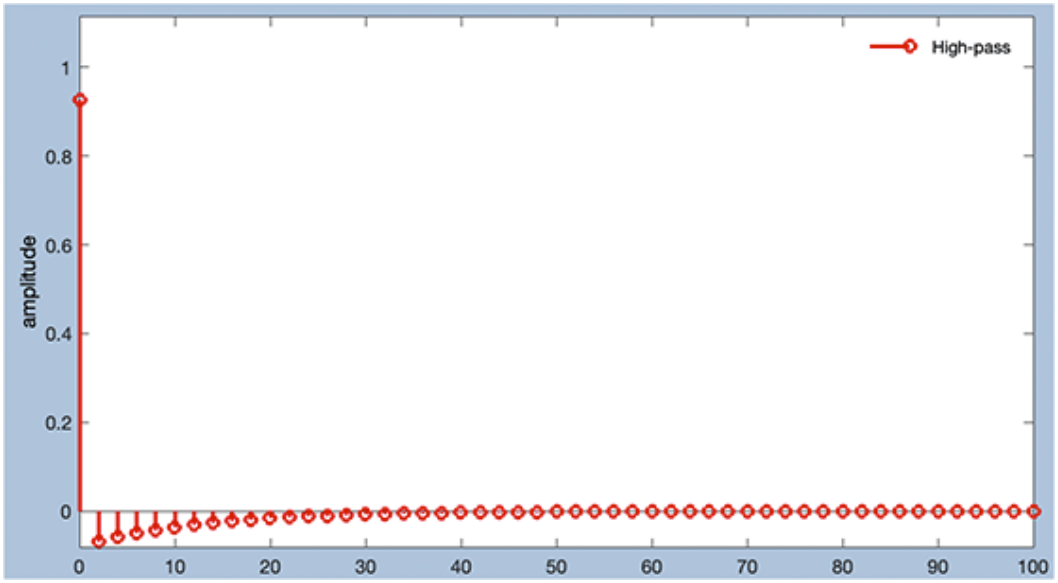
Launch the filtering tool, and set the parameters as shown in Screenshot 4.12. In particular, turn off the low-pass filter, turn on the high-pass filter, set the high-pass cutoff to 5 Hz, and set the roll-off to 12 dB/octave. And make sure that **Channel(s) to filter** is set to **All**. If you look at the frequency response function shown in the upper left, you'll see that the gain is near zero for the lowest frequencies, reaches 0.5 at 5 Hz (because that's the half-amplitude cutoff frequency) and then starts nearing the asymptote of 1.0 by approximately 10 Hz.

Screenshot 4.12



Now look at the impulse response function by changing the **Display** setting from **Filter frequency response** to **Filter impulse response**. It should look like Screenshot 4.13. It's very different from the impulse response function of a low-pass filter. Low-pass and high-pass filters have opposite effects (removing high versus low frequencies), so they have largely opposite impulse response functions. Whereas the low-pass impulse response functions we've looked at had positive values near time zero, this high-pass impulse response function is negative near time zero (but is near 1.0 right at time zero). The reasons for this are discussed in Chapters 7 and 12 of Luck (2014). Here, you'll just have to take my word for it.

Screenshot 4.13



Go ahead and click **APPLY** to run the filter, and then plot the filtered waveforms. You should see something like Screenshot 4.11B. First look at the impulse (Channel 3), which now shows the impulse response function of the filter. You can see the negative values surrounding time zero, but they're pretty small. This is because the impulse response function for a high-pass filter must sum to zero. If the impulse response extends for a long time period, the individual values must be very small.

Now look at the **waveform+drift** channel (Channel 2). The good news is that the filter has virtually eliminated the drift. The bad news is that the filter has produced artifactual negative peaks at the beginning and end of the waveform. You can also see these artifactual peaks in the channel without the drift (Channel 1). The filter has also reduced the amplitude and change the shape of the ERP waveform, but that's to be expected because much of the power in the waveform falls into the frequencies that are attenuated by the filter (which you can confirm by making the unfiltered ERPset active and using **EEGLAB > ERPLAB > Filter &**

Frequency Tools > Plot amplitude spectrum for ERP data). These distortions are why I would never recommend using a filter like this with real data (except for the auditory brainstem responses, which are largely confined to higher frequencies).

If you think about the impulse response function, you can understand why the filter produces artificial negative peaks at the beginning and end of the waveform. The unfiltered waveform starts and ends with positive values. When we replace these values with the impulse response function, the negative values to the left and right of the current point in the impulse response function produce negative values before and after the positive peaks. Note that if the waveform contained negative peaks, the artifactual peaks would be positive (because a negative voltage from the unfiltered waveform multiplied by a negative value in the impulse response function creates a positive value).

Next we're going to try a filter that's not quite as extreme, but still has a higher cutoff frequency than I'd recommend, namely 1 Hz. Make the original **waveform+drift** ERPset active again in the ERPsets menu, launch the filtering tool, and change the cutoff from 5 Hz to 1 Hz. Look at the impulse response function in the filtering tool. You can see a large positive value at time zero, but the nearby values are just barely negative. The function extends for a longer period of time than the function for the 5 Hz filter, and each individual point must be nearer to zero so that the points sum to zero.

Go ahead and click **APPLY** and then plot the filtered data. It should look like Screenshot 4.11C. The drift in Channel 2 is still largely gone, but we haven't attenuated the ERP waveform as much, so that's an improvement. However, the artifactual negative peaks at the beginning and end of the waveform are still present. That's why I wouldn't recommend using a 1 Hz cutoff.

Now let's try the high-pass filter cutoff I ordinarily recommend for most perceptual, cognitive, and affective ERP studies, namely 0.1 Hz. Make the unfiltered ERPset active again, launch the filtering tool, and change the cutoff to 0.1 Hz. If you look at the impulse response function in the filtering tool, you can't really see much beyond the positive value at time zero. That means that the filter will be very mild. Apply the filter and look at the waveforms. As shown in Screenshot 4.11D, the filter has only partially reduced the drift in Channel 2. However, it has produced no noticeable distortion of the ERP waveform. That is, the filter hasn't attenuated the waveform, and it hasn't produced any artifactual peaks.

This set of examples illustrates an important principle: You can fully attenuate the slow drifts in your data but distort your ERP waveforms, or you can fail to fully attenuate the low-frequency noise and avoid distorting your ERP waveforms. You can't both fully attenuate the drifts and avoid distorting the waveform. This is because of the inverse relationship between the time and frequency domains: The more you restrict the frequencies with extensive filtering, the more you distort the time course of the ERP waveform.

Keep in mind that the slow drifts are noise deflections that mainly arise from the skin, and they're positive-going on some trials and negative-going on others. They add random noise to your data, decreasing your power to find statistically significant effects. Obviously that's not a good thing. However, it's much worse to use a filter that induces artifactual effects that are statistically significant but completely bogus, causing you to draw incorrect conclusions. This is why I usually recommend a high-pass cutoff of 0.1 Hz—it reduces the low-frequency noise reasonably well, but it doesn't usually produce substantial artifacts.

In the first edition of my ERP textbook (Luck, 2005), I recommended using 0.01 Hz as the half-amplitude cutoff. Over the following years, however, my collaborators and I systematically compared a variety of different cutoffs, and we typically found that 0.1 Hz produced the best noise reduction without any substantial distortion of the waveforms (Kappenman & Luck, 2010; Tanner et al., 2015). If you're looking at very slow ERPs (like the contralateral delay activity), 0.01 or 0.05 might be better than 0.1, but in most cases I find that 0.1 Hz works best.

This page titled 4.10: Exercise- High-Pass Filtering is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.11: Practical Advice

This chapter has focused on helping you understand how filters actually work and how they can distort your data. I'd like to end with some practical advice about when and how to filter your data. This advice is appropriate for perhaps 95% of perceptual, cognitive, and affective ERP research. If you think your research falls into the other 5%, then you need to be very sure you fully understand how filtering works so that you don't end up either causing significant distortion of your data by overfiltering or failing to remove enough noise by underfiltering.

Let's start with the question of when to filter. As mentioned earlier, you should apply your high-pass filter to the continuous EEG to avoid edge artifacts. To further minimize these artifacts, you should use the option labeled "Remove mean value (DC offset) before filtering" when you are filtering continuous EEG data (but not when you are filtering epoched EEG or averaged ERPs).

You can apply your low-pass filter to the continuous EEG, the epoched EEG, or the averaged ERPs. For most researchers, it will be simplest just to apply both the low- and high-pass filters to the continuous EEG. In my lab, we typically apply the low-pass filter only to the averaged ERPs, but that's mainly for philosophical reasons rather than practical reasons (as described in Luck, 2014).

Now let's discuss filter parameters. My lab typically uses cutoffs of 0.1 and 30 Hz, with a roll-off of 12 dB/octave. That's what I'd recommend if you are recording very clean data, especially if you have highly cooperative participants (e.g., neurotypical young adults). If you have a fair amount of high-frequency noise (e.g., 60-Hz line noise or spiky muscle activity), you can increase the roll-off to 48 dB/octave for the low-pass filter and/or drop the cutoff from 30 to 20 Hz. You'll get a little more distortion, but not enough to matter for most studies. However, I don't recommend a roll-off of 48 dB/octave for the high-pass filter. If you have a lot of low-frequency noise, which is especially common when the participants move around a lot (e.g., infants or young children), you can raise the high-pass cutoff to 0.2 or even 0.5 Hz and/or increase the roll-off to 24 dB/octave. However, these parameters can cause noticeable distortion of the waveforms. If you mainly have line noise and don't want to use a low-pass filter at 20-30 Hz, you can use EEGLAB's *cleanline* plugin (see Bigdely-Shamlo et al., 2015 for important details). If cleanline doesn't work well for you, you can try the newer Zapline method (de Cheveigné, 2020; Klug & Kloosterman, 2022).

If you want to use a high-pass cutoff of greater than 0.1 Hz or a low-pass cutoff of less than 20 Hz, I strongly recommend that you create artificial ERP waveforms that resemble your data and pass them through the filter (even if you're filtering the continuous or epoched EEG with your real data). You can then see exactly how the filter distorts your data. If the distortion you see with the artificial data is small compared to the effects you're seeing in your study, then you don't need to worry. The next section describes how to create and import artificial waveforms.

When my lab measures the onset latency of an ERP component, we will often apply a 10 Hz low-pass filter (12 dB/octave). We find that onset latency measures are highly sensitive to noise, and we get much better statistical power by filtering at 10 Hz. This might seem problematic given that Figure 4.4 shows that a 10 Hz cutoff produces a substantial distortion of the onset latency. However, when we quantify the onset latency of an ERP component, we don't measure the time that the waveform first deviates from zero. Instead, we measure the time at which the voltage reaches 50% of the peak voltage (see Chapter 9 in Luck, 2014, for a justification of this approach). If you look closely at Figure 4.4, you'll see that this 50% point is virtually identical for the filtered and unfiltered waveforms. Also, we're usually comparing the ERPs from two conditions that have both been filtered at 10 Hz, so the effect of the filtering should be equivalent for both conditions. However, we sometimes want to examine the point at which a difference wave first deviates from zero. In these cases, we do minimal filtering or we quantify the amount of latency shift produced by the filter (see, e.g., Bae & Luck, 2018).

This page titled 4.11: Practical Advice is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.12: Exercise- Creating and Importing Artificial Waveforms

Over the years, I've found that applying filters to artificial waveforms has really helped me understand how filters work and whether they might be significantly distorting my data. In fact, I first got interested in how filters work when I used an inappropriate filter and discovered my error by filtering an artificial waveform (see text box below). As a result, I always encourage other people to trying filtering artificial waveforms, especially if they're not going to follow my standard advice about filtering from 0.1 to 20 or 30 Hz. I even created some example artificial waveforms to go along with the filtering chapter in Luck (2014) and made them available online on the publisher's web site. Here, I'm going to explain how you can make artificial waveforms in Excel and import them into ERPLAB.

How I Avoided Embarrassment by Filtering an Artificial Waveform

When I was in grad school, I had the good fortune to spend quite a bit of time with Bob Galambos, who was the mentor of my own mentor, Steve Hillyard. Bob had retired many years before, but he still came to lab meetings from time to time. He was an amazing scientist—among other things, he and his buddy Donald Griffin were the people who first demonstrated that bats navigate using echolocation (Griffin & Galambos, 1941). I learned a lot from having him around.

One day, Bob and I cooked up an idea for an experiment that involved recording both ERPs and the electroretinogram (ERG; see the next text box). Bob volunteered to be the subject. Unfortunately, although he was a great scientist, he was not a very good subject, and the data were very noisy.
That night, I processed the data, and the recordings were a mess. In an attempt to clean up the data, I applied a very strong filter (something like 2-8 Hz, 48 dB/octave). Not only did it remove the noise, it revealed that the stimuli had triggered oscillations in both the ERPs and the ERG. Oscillations were just becoming a hot topic, and I thought I had discovered something new and important. I would surely become famous!

But then I noticed something: The recordings included square-wave calibration pulses, and the filtered calibration pulses contained the same oscillations I was seeing in the ERPs and ERG. That made me realize that the oscillations were artifactually induced by the filter and were not signals that were present in the data. I asked one of the senior grad students, Marty Woldorff, about the oscillations, and he explained how very sharp filters can produce artifactual oscillations. He explained that a filter is like a bell: You put a brief input into a bell (by striking the clapper), and the output of the bell is an oscillation.
That experience of seeing how a filter impacted an artificial signal (the calibration pulse) got me interested in learning more about filters. And it saved me from the embarrassment I would have surely felt if I had tried to write a paper claiming that I had discovered oscillations in the ERPs and ERG.

I created all of the artificial waveforms for this chapter in Excel. You can find a copy of the spreadsheet file, which is named **artificial_data.xlsx**, in the Chapter_4 folder. If you don't have Excel, you can import it into Google Sheets. The first tab has the waveforms, with a separate column for each waveform. I created the ERP-like waveform shown in Screenshot 4.1 by summing together three simulated ERP components, each of which is just one cycle of a cosine function. You can see the formulas in the spreadsheet. You'll also see columns for creating 60-Hz noise and white noise. There are also columns for impulses at different times.

The other sheets are designed to pull out the waveforms for the separate ERPset files that I created for the exercises. These sheets just copy the relevant values from the first sheet (sometimes with modifications). To create an ERPset from a given sheet, you save the sheet as a text file and then import it into ERPLAB.

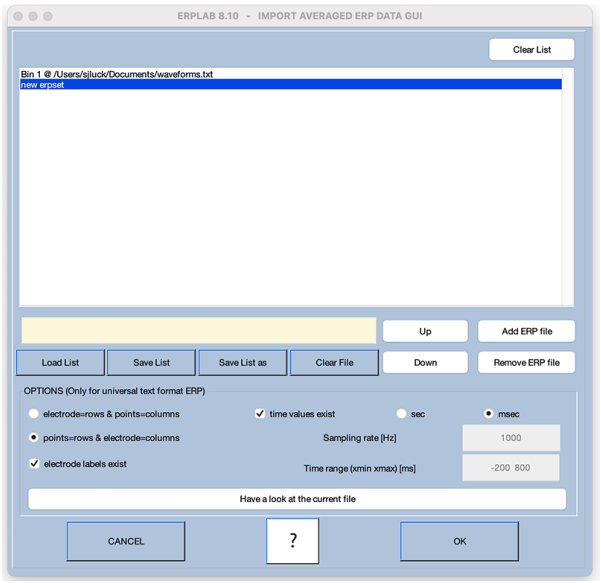
As an example, let's create a text file for the **waveforms** sheet, which contains the waveforms shown in Screenshot 4.1. Go to this sheet in Excel and select **Excel > File > Save as**. Then select **Tab delimited Text (.txt)** as the file format and save the file using **waveforms.txt** as the filename. Note that with this file format, Excel saves only the current sheet as the text file.

The resulting text file is organized with the leftmost column containing the latency of each sample point and the other column(s) containing the voltage values for the individual electrode sites. With this approach, you can only have one bin per text file.

Now let's import the text file into ERPLAB as an ERPset. Quit and restart EEGLAB, and then select **EEGLAB > ERPLAB > Export and Import ERP > Import ERP from text (universal)**. This will bring up the window shown in Screenshot 4.14. The import tool

allows you to specify multiple text files, each which will be stored as a separate bin. However, we're going to just import one text file and create one bin. To do this, click the **Add ERP file** button and select the **waveforms.txt** file that you created in the previous step. You'll then see it in the list of text files to be imported, designated as Bin 1 (see Screenshot 4.14).

Screenshot 4.14



Now we need to provide the import tool with some information about the format of the text file. For the overall structure of the file, click the **points=rows & electrodes=columns** option (which indicates that each line is a time point and each column is an electrode). Check the **electrode labels exist** box, because the first line of the text file contains the labels for the electrodes. Check the **time values exist** box, because the leftmost column contains the time values. If we didn't have the time values in the text file, we could instead indicate the sampling rate and time range, and the import tool would figure out the latency for each time point.

Once everything is set as shown in the screenshot, click **OK**. You'll then see the standard dialog box for naming and saving the new ERPset. You can name it **waveforms**. You don't need to save it, because you already have the file with these waveforms in the Chapter_4 folder (**waveforms.erp**). Finally, you should plot the waveforms and verify that they look like those in Screenshot 4.1.

That's it! Now you can use Excel to create any kind of artificial waveform you like and then import it into ERPLAB. You can then see how the waveform is changed by different filter settings. As I said before, you'll definitely want to do this if you filter more aggressively than my standard recommendation of 0.1 to 30 Hz.

The Electroretinogram (ERG)

The ERG is an EEG-like signal generated by the retina. I have a soft spot in my heart for the ERG, because it's how I got started in electrophysiology. I took a year off in the middle of college and got a job working as a research assistant for Martha Neuringer at the Oregon National Primate Research Center. I worked on a study of the effects of omega-3 fatty acid deprivation on visual system development in infant rhesus monkeys.

Martha recorded the ERG as one of our outcome measures. She anaesthetized the monkeys and then put a special-purpose electrode on the cornea to record the ERG signal. This signal was then amplified and recorded on a special tape recorder. We would then drive to a different lab 10 miles away, where we used a gigantic computer to digitize the signals, average across trials, and a quantify the amplitude of the ERG signal. This was very time-consuming, so I was tasked with writing software for an Apple II—the first mainstream personal computer—that would allow us to directly record the ERG (onto floppy disks!) and do the averaging and analysis. This started me down the road to ERP research.

Bob Galambos taught me a trick for more easily recording the ERG from humans using small EEG electrodes placed on the lower eyelids rather than an electrode placed directly on the eye. The trick is to place one electrode under each eye and put an opaque patch over one eye. When a visual stimulus is presented, you get ERG activity plus EEG activity from the electrode

under the unpatched eye, and you get nearly identical EEG activity from the electrode unpatched eye, but without the ERG activity. If you use the electrode under the unpatched eye as the active site and the electrode under the patched eye as the reference site, the EEG (which is nearly identical at both electrodes) is subtracted away, leaving just the ERG. This was one of many things I learned from Bob.

This page titled 4.12: Exercise- Creating and Importing Artificial Waveforms is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.13: Matlab Script for this Chapter

I haven't provided a script for this chapter, because scripts in previous chapters have shown how to do filtering from a script. Instead, I provided the Excel file for creating artificial waveforms (in the folder with the data for this chapter).

This page titled 4.13: Matlab Script for this Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

4.14: Key Takeaways and References

Key Takeaways

- You can filter EEG and ERPs in the frequency domain using the Fourier transform, or you can filter in the time domain using an impulse response function or a weighting function. These three approaches are mathematically identical and produce exactly the same result. However, we mainly care about the time domain in ERP research, so it's helpful to think about filtering as a time-domain operation.
- The impulse response function of a filter is just the output of the filter when the input is an impulse of amplitude 1 at time zero.
- You can think of an ERP waveform as a sequence of impulses, one at each time point. The output of a filter for a given input waveform can be computed by replacing each impulse in the input waveform with a copy of the impulse response function that has been scaled by the amplitude of the impulse and then summing them together.
- You can also think of filtering as being implemented by a weighted running average. The weighting function is the mirror image of the impulse response function. This conceptualization allows you to see how the filtered value at a given time point is related to the values at the surrounding time points.
- Precision in the time domain is inversely related to precision in the frequency domain. The more heavily you filter, the more temporal distortion you will produce. The amount of temporal "smearing" produced by a filter is easily understood by the width of the impulse response function or weighting function. Heavy filtering can introduce artifactual peaks in your waveform, especially with high-pass filters or steep roll-offs, potentially causing you to draw completely bogus conclusions.
- For most perceptual, cognitive, and affective ERP experiments, filtering from 0.1 to 30 Hz works very well. If you want to filter more heavily, you should first apply the filter to artificial waveforms so that you can see what kind of distortion is produced by the filter.

References

Bae, G. Y., & Luck, S. J. (2018). Dissociable decoding of working memory and spatial attention from EEG oscillations and sustained potentials. *The Journal of Neuroscience*, *38*, 409–422. https://doi.org/10.1523/JNEUROSCI.2860-17.2017

Bigdely-Shamlo, N., Mullen, T., Kothe, C., Su, K.-M., & Robbins, K. A. (2015). The PREP pipeline: Standardized preprocessing for large-scale EEG analysis. *Frontiers in Neuroinformatics*, *9*. https://doi.org/10.3389/fninf.2015.00016

de Cheveigné, A. (2020). ZapLine: A simple and effective method to remove power line artifacts. *NeuroImage, 207,* 116356. https://doi.org/10.1016/j.neuroimage.2019.116356

Klug, M., & Kloosterman, N. A. (2022). Zapline-plus: A Zapline extension for automatic and adaptive removal of frequency-specific noise artifacts in M/EEG. *Human Brain Mapping, 43(9)*, 2743–2758. https://doi.org/10.1002/hbm.25832

Griffin, D. R., & Galambos, R. (1941). The sensory basis of obstacle avoidance by flying bats. *Journal of Experimental Zoology*, *86*, 481–506. https://doi.org/10.1002/jez.1400860310

Kappenman, E. S., & Luck, S. J. (2010). The effects of electrode impedance on data quality and statistical significance in ERP recordings. *Psychophysiology*, *47*, 888–904. https://doi.org/10.1111/j.1469-8986.2010.01009.x

Luck, S. J. (2005). *An Introduction to the Event-Related Potential Technique*. MIT Press.

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Mitra, P. P., & Pesaran, B. (1999). Analysis of Dynamic Brain Imaging Data. *Biophysical Journal*, *76*(2), 691–708. https://doi.org/10.1016/S0006-3495(99)77236-X

Tanner, D., Morgan-Short, K., & Luck, S. J. (2015). How inappropriate high-pass filters can produce artifactual effects and incorrect conclusions in ERP studies of language and cognition. *Psychophysiology*, *52*, 997–1009. https://doi.org/10.1111/psyp.12437

This page titled 4.14: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5: REFERENCING AND OTHER CHANNEL OPERATIONS

Learning Objectives

In this chapter, you will learn to:

- Avoid the incorrect assumption that an ERP waveform primarily reflects signals from the active electrode site and instead recognize that the waveform equally reflects signals from the active and reference sites
- Compute the absolute voltage at the scalp electrodes given the internal generator voltage and the propagation weights between the generator and the scalp sites
- Compute the voltage between each electrode site and the ground electrode given the absolute voltages at each site
- Compute the referenced voltage at each site given the voltage between each site and the ground electrode
- Re-reference the voltage to another site
- Use simple algebra to determine the correct equations for re-referencing your data
- Use two alternatives to referencing your data, namely current density and global field power
- Implement several different reference schemes and alternatives to referencing in ERPLAB using ERP Channel Operations and EEG Channel Operations
- Implement common referencing scenarios using a script

This chapter dives deeper into channel operations, especially with respect to how the reference electrode works and how you can re-reference your data. Once you know what you're doing, re-referencing the data will take you just a few seconds, and many researchers don't give it much thought. However, **the reference has an enormous impact on your ERP waveforms**, so you really need to understand what you're doing when you reference or re-reference your data.

The exercises in this chapter are designed to give you greater insight into what the reference site does, why we need to reference the data, and how re-referencing the data can clarify or obfuscate the results depending on what new reference you choose. We'll also discuss the mechanics of re-referencing in ERPLAB so that you can have confidence that you're doing it correctly.

We'll start with simulated data so that you can see how the original and referenced data on the scalp are related to the underlying neural generator. The simulations use Excel rather than ERPLAB, which makes it easier to see exactly what's going on. You can use Google Sheets instead of Excel, if you prefer. I'm assuming that you know the basics of spreadsheets, including how an equation in one cell can compute a value on the basis of other cells.

5.1: Data for This Chapter

5.2: Background- Understanding Active, Reference, and Ground Electrodes

5.3: Exercise- Working with the Artificial Data

5.4: Exercise- Average Mastoids as the Reference

5.5: Exercise- Re-Referencing the N400 ERP CORE Data

5.6: Exercise- The Average Reference

5.7: What is the Best Reference Site?

5.8: Exercise- Current Density

5.9: Exercise- Global Field Power

5.10: Exercise- Referencing the EEG Data from the ERP CORE N400 Experiment

5.11: Exercise- Other Common Re-Referencing Scenarios

5.12: Matlab Script For This Chapter

5.13: Key Takeaways and References

This page titled 5: Referencing and Other Channel Operations is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.1: Data for This Chapter

The data we'll be using for the exercises in this chapter can be found in the Chapter_5 folder in the master folder: https://doi.org/10.18115/D50056.

This page titled 5.1: Data for This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.2: Background- Understanding Active, Reference, and Ground Electrodes

In this section, we'll review how active, reference, and ground electrodes work in EEG recordings. You can find a detailed discussion in Chapter 5 of Luck (2014). Here, I'll explain the concepts using the artificial ERP waveforms shown in Figure 5.1. It helps to use artificial data in this context because we know what the true signals are. Later in the chapter, you'll apply what you've learned to real data.

In the artificial example shown in Figure 5.1A, the generator dipole is represented by the arrow, with the positive side pointing toward the Pz electrode. The broken line represents the transition between the positive and negative sides of the dipole, and the voltage is zero along this line. Unfortunately, we don't know the location of this zero line when we're looking at real data, and we don't have a single zero line when more than one dipole is active (which is almost always the case).

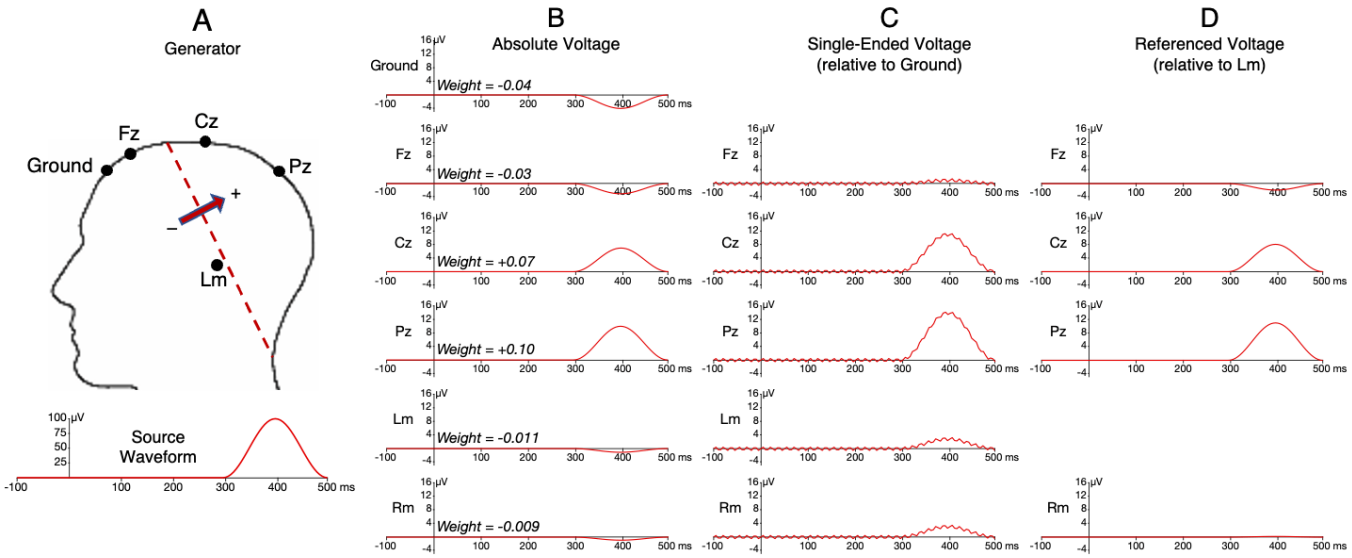


Figure 5.1. Example of active, ground, and reference electrodes. (A) Generation of the ERP. The arrow represents the generator dipole, and the source waveform shows the change in voltage over time at this dipole. The broken line represents the band of zero voltage at the transition between the positive and negative sides of the dipole. (B) Absolute voltage at each electrode site (which is known in this simulation but cannot be directly recorded). The absolute voltage at a given site is the source waveform multiplied by the weight for that site. (C) Single-ended voltage at each electrode site (i.e., the potential between a given site and the ground electrode). In most systems, this signal is present only inside the amplifier and is not available in the system's output. Note that a small amount of 60 Hz noise from the amplifier's ground circuit contaminates these signals. (D) Voltage at each site referenced to the left mastoid (Lm) electrode. This signal is obtained by subtracting (either in hardware or in software) the single-ended Lm signal from the signal at each of the other electrodes.

Voltage is the potential (pressure) for charges to move from one place to another place, so there is no such thing as the voltage at a single electrode site. However, it is convenient to use the term *absolute voltage* to refer to the potential between one electrode and the average of the entire surface of the head. We use the average of the surface of the head in our definition of absolute voltage because the average voltage across the entire surface of the head is assumed to be zero. This assumption is true only for perfectly spherical heads (Yao, 2017), but it is a reasonable approximation for our present purposes.

Figure 5.1B shows the absolute voltage that we would expect at each of our recording electrodes. The absolute voltage at a given electrode site is equal to the source waveform multiplied by a weighting factor that represents the degree to which voltage is conducted from the specific generator to a given electrode site. For example, we're assuming that 10% of the voltage from the generator dipole is conducted to the Pz electrode site, so the weight for that site is 0.10. The source waveform has a peak amplitude of 100 µV, so the absolute voltage waveform at Pz peaks at 10 µV. The weights are negative for the electrodes on the negative side of the dipole, so the waveforms are negative-going at those sites. (The weights shown in Figure 5.1A are not the true weights, but are just examples that produce nice round numbers.) If this set of concepts about ERP generation is unfamiliar to you, you can learn more by reading Chapter 2 of Luck (2014) or by taking my online Introduction to ERPs course.

There is no way to measure the absolute voltage at a given electrode site. The absolute voltage is just a convenient hypothetical entity for explaining how reference electrodes work. The EEG amplifier would actually measure the voltage between each electrode site and the ground electrode. The voltage between two electrodes is simply the difference between the absolute voltages at those two sites. For example, the absolute voltage at Pz peaks at 10 μV and the absolute voltage at the ground electrode peaks at -4 μV, so the voltage between Pz and ground is 14 μV (10 minus -4). We call the voltage between a given site and the ground electrode, which is what an EEG amplifier actually measures, the *single-ended signal* (shown in Figure 5.1C).

EEG amplifiers contain noise in the *ground circuit* (which is the part of the amplifier that the ground electrode is connected to). Because all electrodes are initially measured with respect to the ground electrode, the noise in the ground circuit is present with approximately equal amplitude in the single-ended signals from all of the electrode sites. In Figure 5.1C, I added some 60 Hz noise to every signal to simulate this noise. However, this noise is often much *much* **much** larger, obscuring the actual EEG signals.

EEG recording systems therefore contain *differential amplifiers*, which use a trick to subtract away the noise from the ground circuit. The trick is to use another electrode as the *reference electrode*. The single-ended signal at the reference electrode is also recorded relative to the ground electrode, so it also contains the noise from the ground circuit. Consequently, if we subtract the reference electrode signal from the signals at our other electrodes (our *active electrodes*), the noise is approximately the same in the active and reference electrodes, so the noise is subtracted away. This is shown in Figure 5.1D, in which the single-ended signal from the left mastoid (Lm) electrode is subtracted from the signal at each of the other electrodes to create a *referenced* or *differential* signal. You can see that the referenced waveforms no longer have the 60 Hz noise that is visible in the single-ended signals. If you didn't follow this brief overview of referencing, you can watch this brief video from the online Introduction to ERPs course or read the more detailed description in Chapter 5 of Luck (2014).

In most EEG systems, the referencing subtraction is performed in the amplifier's hardware, so you have no way of accessing the single-ended signals. You'll only ever see the referenced signals. There are, however, some exceptions. The BioSemi ActiveTwo system (which we used for the ERP CORE experiments) does not subtract the reference in hardware and instead outputs the single-ended signals. The researcher then subtracts the reference from the single-ended signals in software, after the recording session is over. During the recording, this system will show the referenced signals on the screen (to minimize noise from the ground circuit), but only the single-ended signals are saved to the file. This confuses many researchers, who do not realize that the saved data has not been referenced. If you use BioSemi, don't forget to subtract the reference! The Brain Products ActiCHamp system also obtains the single-ended signals, but the data collection software performs the referencing subtraction before the data are saved to a file. This is less confusing.

Yes, I'm a control freak

For recording the EEG from the ActiCHamp system, Brain Products provides an open source program called *Pycorder* in addition to their closed source *Recorder* software. My lab has modified the Pycorder software so that we can save the single-ended signals instead of the referenced signals. We then do the referencing offline in software. This produces the same end result that we would get by saving the referenced data, but I like having the raw single-ended data and doing the referencing myself. I guess I'm a bit of a control freak...

In the example shown in Figure 5.1, the Lm electrode is near the zero line for the generator dipole. As a result, the referenced voltages at each site are close in amplitude to the absolute voltages. However, that will not typically be the case, so you shouldn't assume that the referenced voltages are a good approximation of the absolute voltages. Instead, you should always think of the voltage at a given electrode site as being the difference between the signal at the so-called active electrode and the signal at the so-called reference electrode. I use the phrase "so-called" here because we are simply making a difference between two sites, and both contribute equally to the referenced voltage. If there is a large deflection in the absolute voltage in the reference electrode, than an inverted version of this deflection will be present at every so-called active site. (It's inverted because we subtract the reference.) So, when you see a waveform labeled "Cz", you are not looking at the voltage at the Cz electrode site. You are looking at the potential between the Cz site and the reference site, which is equivalent to the absolute voltage at Cz minus the absolute voltage at the reference site. The so-called active and so-called reference sites are equal contributors to this voltage. In fact, in some areas of research, Cz is used as the reference (which is equivalent to inverting the waveform). It's therefore more accurate to say that a waveform is from the "Cz channel" rather than from the "Cz electrode".

If there is one thing I hope you learn from this chapter, it's that **you need to think of a given ERP waveform as equally reflecting signals from the so-called active and so-called reference sites, not as being primarily from the active site**. The reference site you choose for your analyses can have a huge impact on how the waveforms look and which channels show

the experimental effects. Unfortunately, there is no perfect reference site. In most cases I recommend simply using whatever is common in your subfield. That way, your data can be compared with the data from other studies. If you use a different reference site, your data may end up looking quite odd, and you may think that you've discovered new effects.

This page titled 5.2: Background- Understanding Active, Reference, and Ground Electrodes is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.3: Exercise- Working with the Artificial Data

In the Chapter_5 data folder, you will find an Excel spreadsheet named **simulated_data.xlsx**, which contains everything I used to create the simulated data in Figure 5.1. Open the file in Excel (or import it into Google Sheets). You will see that each line is a time point. The first column shows the latency of the time point (in ms), and the second column is the source waveform. There are also columns for the absolute voltage at each electrode site, with the weighting factor for a given site in the second row.

If you look at the equations in the cells that compute the absolute voltage values, you'll see that the absolute voltage at a given electrode site was computed by multiplying the weighting factor for that site by the source waveform. You'll also see that the single-ended voltage was created by subtracting the absolute voltage at the ground electrode from the absolute voltage at each other site, and then adding the 60 Hz noise. And you'll see that the referenced voltage was computed by subtracting the single-ended signal at the Lm electrode from the single-ended signal at a given active site. At the right side of the spreadsheet, you'll see plots of the absolute voltages, the single-ended voltages, and the referenced voltages. Take a careful look at the equations in the spreadsheet and make sure that you understand how the absolute voltage is related to the source waveform and the weights, how the single-ended voltage is related to the absolute voltage, and how the referenced voltage is related to the single-ended voltage.

The spreadsheet also contains sheets with copies of the data formatted for exporting as a text file (using the **Tab delimited Text** format), which can easily be imported into ERPLAB (using **EEGLAB > ERPLAB > Export & Import ERP > Import ERP from text (universal)**). This allows you to create simulated data and see how the various ERPLAB processes work.

In the spreadsheet, change the weight above the **Absolute_Ground** label from **-.04** to **-.10**. You'll see that this greatly increases the magnitude of the absolute voltage in the ground channel. And because the ground is subtracted from the other signals to create the single-ended voltage, this also changes all the single-ended voltage waveforms. But did it change the referenced waveforms?

No, it did not! The nature of the referencing procedure means that any signals or noise at the ground electrode are subtracted away from the referenced voltages. This means that you can place the ground electrode anywhere on the head, and the location does not matter. You might then wonder why we use a ground electrode at all. The answer is simple: The amplifier will freak out if there isn't a ground electrode. You need to have a ground, and it needs to be appropriately attached to the head (or anywhere on the body). But the location does not matter.

Now try changing the weight for the Lm channel from **-.011** to **-.20**. This simulates changing the location of the reference electrode, moving it farther away from the zero line. What did this do to the referenced waveforms? Not only are some of them now larger, the polarity of the Fz waveform has now changed from negative to positive. This demonstrates how the signal at the reference electrode can have a large impact on the referenced waveforms from the so-called active electrodes.

Play around with the spreadsheet some more. For example, try changing the weights for the other electrodes or the magnitude of the 60 Hz noise.

This page titled 5.3: Exercise- Working with the Artificial Data is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.4: Exercise- Average Mastoids as the Reference

For historical reasons, many ERP studies use the mastoid process (the thick bone behind the ear) as the reference electrode. This is why the data shown in Figure 5.1D use the left mastoid as the reference. However, it seems a little odd to use a reference that is lateralized to one side of the head. Might this lead to some kind of hemispheric bias in the data? In reality, this isn't usually a significant problem. However, just to be safe, many researchers use the average of the left and right mastoids (Lm and Rm) as the reference. In this exercise, we're going to look at two ways of transforming the data to reference the simulation data to the average of Lm and Rm. This exercise exemplifies an important principle, namely that data recorded using one electrode site as the reference can easily be re-referenced offline, in software, to one of the other electrode sites (or some combination of electrode sites).

Open up the spreadsheet from the previous exercise and make sure everything is back to the way it was originally (or download the spreadsheet again). Make 3 new columns just to the right of the columns for the referenced data, and label them **Fz-Avg**, **Cz-Avg**, and **Pz-Avg**. If you look at how the referencing is done for the Fz channel at the first time point (cell O4 of the spreadsheet), you'll see that the referenced value is the single-ended value for Fz (cell **J4**) at this time point minus the single-ended value for Lm (cell **M4**, but with a "$" symbol before the letter so that it remains column **M** even if we paste it somewhere else). This is the subtraction that I used to reference the data in Figure 5.1.

We want to change this so that we subtract the average of Lm and Rm from Fz. That average is simply **(Lm+Rm)/2**, which would be **($M4+$N4)/2** given that Lm is in column **M** and Rm is in column **N** of our spreadsheet. So, to create a value for Fz using the average of Lm and Rm as the reference, we need the equation for the first time point to be **=J4-($M4+$N4)/2**. Go ahead and put this equation into cell **S4**, which should be in the new column that you labeled **Fz-Avg**. If you then copy and paste this equation into cells **T4** and **U4**, the **J4** should update to **K4** and **L4**, respectively. If you then copy and paste these three cells to all the remaining time points, the row numbers should update, and you'll have the appropriate values for all the cells.

Now compare the new values you created to the original referenced values (with Lm as the reference). They should be pretty similar. This is because the single-ended signal at Rm is nearly identical to the single-ended signal at Lm, so the average of Lm and Rm is nearly identical to the Lm signal. In turn, this is because the weights for Lm and Rm are pretty similar. However, there may be situations in which the Lm and Rm signals are not so similar (e.g., if the generator dipole happened to be near Lm without a generator near Rm). This is why it's a good idea to use the average of Lm and Rm rather than just using one side.

In most systems, you don't have access to the single-ended signals, so you wouldn't be able to use this approach for referencing the data to the average of the two mastoids. For example, the original data might all be referenced to Lm. However, if you have a recording of Rm (also referenced to Lm), there is a trick you can use to re-reference the data to the average of Lm and Rm. The trick was described by Paul Nunez in his classic book on the biophysics of EEG (Nunez, 1981), and the algebra is spelled out in Chapter 5 of Luck (2014). Specifically, if you subtract 50% of the Rm signal from each channel that was already referenced to Lm, this is equivalent to taking the single-ended data from the active electrode and subtracting the average of the single-ended Lm and Rm signals. For example, to re-reference Fz to the average of Lm and Rm, you would take the already-referenced Fz channel (which is really Fz – Lm) and subtract 0.5 of the Rm channel (which is really Rm – Lm).

Create three new columns labeled **Fz-Avg2**, **Cz-Avg2**, and **Pz-Avg2**. Put equations into these columns that use this different approach to re-referencing. That is, take the values that were already referenced (Columns O-R) and subtract 0.5 times the Rm channel from the Fz, Cz, and Pz channels. Once you've created these three new channels, compare them with the previous set you created. You should see that these two ways of referencing to the average of Lm create exactly identical results. However, the first method can't be used in most systems, because it requires access to the single-ended signals, so you may need to use the second method. When you're working with real data instead of simulated data, you'll do the referencing in ERPLAB using Channel Operations rather than in a spreadsheet. However, Channel Operations uses equations that are much like the spreadsheet equations. I hope that the experience you've now gotten with the spreadsheet will give you a better understanding of how the equations work in Channel Operations.

This page titled 5.4: Exercise- Average Mastoids as the Reference is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.5: Exercise- Re-Referencing the N400 ERP CORE Data

Now let's see how re-referencing is accomplished in ERPLAB with the data from the N400 ERP CORE experiment. To keep things simple for this exercise, we'll just work with the grand average ERPs.

Launch EEGLAB and set the current directory to be the **Chapter_5** folder. Load the file named **Grand_N400_diff.erp** using **EEGLAB > ERPLAB > Load existing ERPset**. Plot the data from Bins 3, 4 and 5 (**EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms**). The waveforms should look familiar from the previous chapters. As before, Bins 3 and 4 are the related and unrelated target words, respectively. Bin 5 is the unrelated-minus-related difference wave. You can see a big beautiful N400 (especially in the difference wave), with the biggest N400 in the CPz channel.

But remember, the waveform in a given channel is determined just as much by the reference electrode as by the active electrode, even though the active electrode is typically used to name the channel. Unless we know what was used as the reference, we don't really know what we're looking at in this plot. When I'm reading a journal article, I go to the Method section and find out what was used as the reference before I look at any of the waveforms. If you look at the Method section for the paper on the ERP CORE (Kappenman et al., 2021), you'll see that we used the average of the P9 and P10 electrode sites as the reference for the N400 experiment (and most of the other experiments). P9 and P10 are quite close to the left and right mastoids, so the waveforms look almost identical to what we would have gotten using the average of Lm and Rm as the reference. However, it's easier to get a good electrical connection with P9 and P10, so we're starting to use these electrodes as our standard reference sites.

Once you understand that referencing is just a matter of subtraction, you can use some very simple algebra to figure out how to re-reference data that have already been referenced. To demonstrate, we're going to re-reference the N400 data to the Cz electrode site. Almost all N400 studies use the mastoids (or something nearby) as the reference, so Cz would be an unusual choice for an N400 experiment. However, Cz is used as the default reference in some EEG recording systems (e.g., the EGI system), so it's easy to imagine that someone would look at N400 data with a Cz reference (especially someone who didn't understand the importance of the reference location). As you'll see, the data look quite different with a Cz reference.

Let's start with the algebra. I'm not much of a math person, so I promise it will be simple. For the CPz channel, the waveforms that you just looked at—with the average of P9 and P10 as the reference—can be expressed conceptually as:

$$CPz_{Referenced} = CPz_{Absolute} - [(P9_{Absolute} + P10_{Absolute})/2]$$

In other words, the referenced voltage at CPz is just the absolute voltage at CPz minus the average of the absolute voltages from P9 and P10. That's not how the referenced CPz channel was actually created, but it's conceptually equivalent. Here's the corresponding expression for the Cz channel:

$$Cz_{Referenced} = Cz_{Absolute} - [(P9_{Absolute} + P10_{Absolute})/2]$$

Our goal is to re-reference the CPz channel so that Cz is now the reference. In other words, we want to create a new channel defined as:

$$CPz_{Referenced} = CPz_{Absolute} - Cz_{Absolute}$$

It turns out that we can get this by just taking the data that have been referenced to the average of P9 and P10 and subtracting Cz from each channel. For CPz, this gives us:

$$CPz_{Referenced} - Cz_{Referenced}$$

$$= (CPz_{Absolute} - [(P9_{Absolute} + P10_{Absolute})/2]) - (Cz_{Absolute} - [(P9_{Absolute} + P10_{Absolute})/2])$$

$$= CPz_{Absolute} - Cz_{Absolute} - [(P9_{Absolute} + P10_{Absolute})/2] - [-(P9_{Absolute} + P10_{Absolute})/2]$$

$$= CPz_{Absolute} - Cz_{Absolute}$$

The original reference is in both signals, so it drops out when we do the subtraction. Pretty cool, eh? I think so. But maybe that's why I've spent much of my adult life doing ERP research.

Okay, let's try it. With **Grand_N400_diff.erp** as the active ERPset, select **EEGLAB > ERPLAB > ERP Operations > ERP Channel Operations**. Clear out any equations that remain in the text box from the last time you used this routine. Then change the **Mode** from **Modify existing ERPset** to **Create new ERPset**. We used the **Modify existing ERPset** mode previously when we added a new channel that was the average of several existing channels. But now we don't want to add new channels to our existing set of channels—we want to create a brand-new ERPset in which all of the channels have Cz as the reference. Otherwise we would have a huge number of channels in our ERPset.

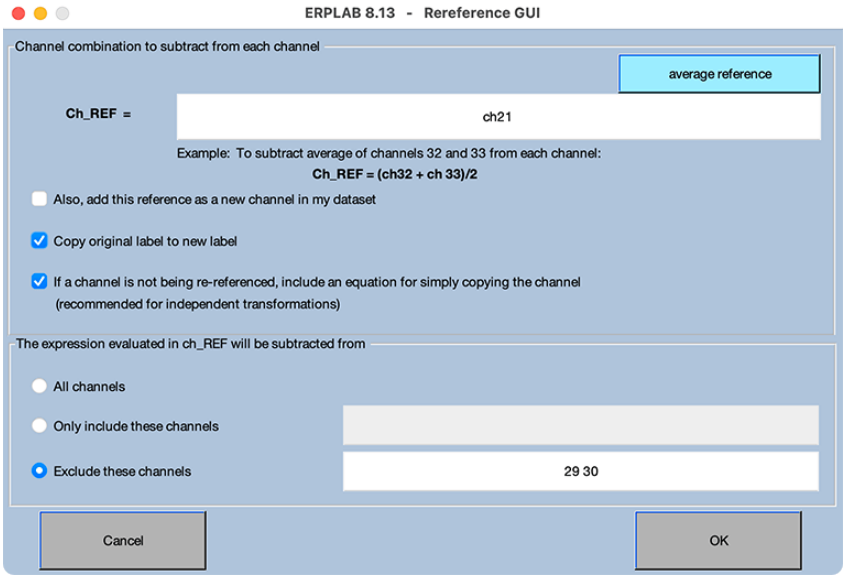
When we create a new ERPset, we use **nch** (short for "new channel") to indicate the new channels that we're creating and **ch** to indicate the original channels. For our first channel, FP1, we would subtract the original Cz channel (**ch21**) from the original FP1 channel (**ch1**) to create the new channel (**nch1**). The formula we would enter into the Channel Operations GUI would therefore be:

nch1 = ch1 - ch12 Label FP1

We would then repeat this for channels 2-28. Channels 29 and 30 are the horizontal and vertical electrooculogram (EOG) channels, which use a different reference, so we would just create new channels that are exact copies of the original channels for these sites.

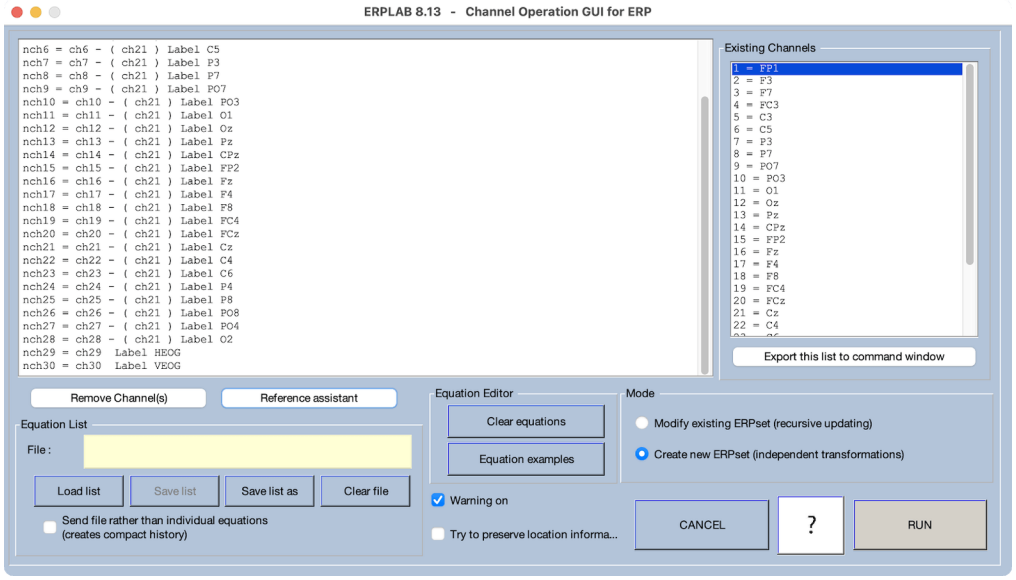
To avoid typing all these equations, click the **Reference assistant** button. You'll see a new window that looks like Screenshot 5.1. If you type **ch21** in the text box labeled **Ch_REF** near the top of the window, the Reference assistant will create appropriate equations using **ch21** as the reference for every channel. Make sure that the check boxes are set as shown in the screenshot. Also, select **Exclude these channels** and put **29 30** in the text box, which will cause it not to create re-referencing equations for the horizontal and vertical EOG channels. Then click **OK**.

Screenshot 5.1



Now the main Channel Operations window should be filled with equations, as shown in Screenshot 5.2 (but note that you may need to scroll the equations window to see all the equations). The equation for each channel simply creates a new channel that is the original channel minus channel 21 (Cz). However, channels 29 and 30 just copy the original horizontal and vertical EOG channels without changing the reference.

Screenshot 5.2



You can now click **RUN** to execute this set of equations. Because you've selected the **Create new ERPset** option, you'll see the usual window for saving a new ERPset. It will suggest a name composed of the original ERPset name with **_chop** (for "channel operations") appended to the end. However, this is just a suggestion, and it's often good to use a more informative name. Let's use **_CzRef** instead of **_chop**. You should save it as a file if you're not going to do the next step right away.

Channel Operations and the ERPLAB Design Philosophy

Creating a separate equation for each channel might seem overly complicated. After all, we're applying the same operation to almost every channel, so there's considerable redundancy in the list of equations. Other EEG/ERP analysis systems have much more concise ways of specifying how to re-reference the data. However, by specifying a literal equation for each channel, you know exactly what the operation is doing to your data. In other systems, it's not obvious exactly what the software is doing when you re-reference the data. In fact, when I've used other systems, I've resorted to passing artificial data through the re-referencing procedure so that I could figure out exactly what it was doing. In ERPLAB, you write the equations for re-referencing, so there is no uncertainty about how the re-referencing works. This reflects one of our core design philosophies when we created ERPLAB: No magic! We want researchers to know exactly what our software is doing to their precious data.

As an example of this philosophy, check out our documentation page on Timing Details. When you say you want to measure the mean amplitude between 300 and 500 ms, what happens if your sampling rate is 256 Hz and you don't have time points at exactly 300 and 500 ms? We describe the exact algorithm that we use to round up or down.

The equation approach we use in Channel Operations has another benefit: It's incredibly flexible. You can perform all kinds of interesting transformations of the data, going way beyond re-referencing. For example, you can take the absolute value of a channel to *rectify* it (which is useful if you have a channel that contains EMG data). You can create a new channel with the global field power (Skrandies, 1989), as will be described below. You can compute the difference between two channels. And, as you saw in Chapter 3, you can create a new "cluster" channel that is the average of a subset of your channels. To see the possibilities, click the **Equation examples** button in **ERP Channel Operations**.

Now plot the data from Bins 3–5. You will see that the Cz channel is now flat, because a channel minus itself is zero. You'll also see that most of the channels now have a more positive voltage for the unrelated targets than for the related targets, and the unrelated-minus-related difference is now positive instead of negative. In other words, with Cz as the reference, we have a P400 instead of an N400! Moreover, rather than being largest in the CPz channel, the unrelated-minus-related difference is now largest in the F7 channel.

Next, let's re-reference the data using Oz as the reference. That is, follow the same steps you used to re-reference the data to Cz, but use **ch12** instead of **ch21** as the reference site and name the ERPset **Grand_N400_diff_OzRef**. When you plot the data, you'll see a small N400 in the CPz channel and an opposite-polarity "P400" in the F7 channel. So, you can see that the choice of the

reference determines whether a given ERP component is positive, negative, or positive at some sites and negative at others. The reference electrode also impacts which channel has the biggest effects. It really matters!

This page titled 5.5: Exercise- Re-Referencing the N400 ERP CORE Data is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.6: Exercise- The Average Reference

Earlier in this chapter, we defined the *absolute voltage* as the potential between a given electrode and the average of the entire surface of the head. This is a strictly theoretical concept, because we can't record from the entire surface of the head. How would you get an electrode on the bottom side of someone's skull?

However, many researchers try to approximate the absolute voltage by using the average across all the electrode sites in their recording as the reference. Although this has been shown to be a good approximation for one particular generator site when an extremely broad array of electrodes is used (Dien, 1998), it often provides a very poor approximation of the absolute voltage (see Chapter 7 in Luck, 2014). Nonetheless, the *average reference* can still be useful under some conditions. For example, it is commonly used for studies of face-elicited N170 activity (Rossion & Jacques, 2012), and we used it for the N170 paradigm in the ERP CORE (Kappenman et al., 2021).

In this exercise, we'll re-reference the data in **Grand_N400_diff.erp** to the average of all the EEG electrodes. To accomplish this, load **Grand_N400_diff.erp** into ERPLAB if it's not already loaded, and make sure that the original ERPset is active (rather than the ERPsets that you created with Cz or Oz as the reference). Select **EEGLAB > ERPLAB > ERP Operations > ERP Channel Operations**, clear out any equations that remain in the text box from the last time you used this routine, and make sure that the **Mode** is set to **Create new ERPset**.

Now click the **Reference assistant** button. In the text box labeled **Ch_REF** near the top of the window, type **avgchan(1:28)**. This indicates that you want to use the average of channels 1–28 as the reference. We're excluding channels 29 and 30, which are the EOG channels. As before, select **Exclude these channels** with **29 30** in the text box. Then click **OK**. You'll see that the list of equations subtracts **avgchan(1:28)** from each individual channel, except for the EOG channels. If you go through the algebra of this subtraction, you'll see that this will create EEG channels that are equal to the active electrode minus the average of all the electrodes. Now click **RUN**, and then name the new ERPset **Grand_N400_diff_AvgRef**.

Now plot the data. Just as you saw with the Oz reference, you'll see that the difference between related and unrelated targets with the average reference is positive at some sites and negative at others. This is an inevitable consequence of using the average of all sites as the reference. That is, at every moment in time, the average-referenced voltages across the different electrode sites must sum to zero, so the voltage will be negative at some electrode sites and positive at others. This is true both for *parent waveforms* (e.g., the waveforms for the related and unrelated target words) and for difference waveforms. So, when you use the average reference, don't think you've discovered something interesting when you find that your experimental effect is positive in some channels and negative in others. It's just a necessary consequence of the algebra. Some components and experimental effects will exhibit such polarity inversions with other references, but it is inevitable with the average reference.

I'd like to point out using the average across sites as the reference in order to approximate the absolute voltage assumes that the surface of the head sums to zero, but this is only true for spheres. I have yet to meet someone with a spherical head. And no neck. Fortunately, there is a way to estimate the true zero, called the *Reference Electrode Standardization Technique* (REST), and there is an EEGLAB plugin that implements it (Dong et al., 2017). I haven't tried it myself or looked at the math, so I don't have an opinion about whether it's useful and robust. But if you really want to get an estimate of the absolute voltage, REST seems like the best current approach.

This page titled 5.6: Exercise- The Average Reference is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.7: What is the Best Reference Site?

Figure 5.2 shows the data for a few key channels with the four references we've used so far (Average of P9 and P10, Cz, Oz, and the average of all the EEG sites). As you can see, the choice of reference electrode has a massive effect on the waveforms. But which of these is the correct way to reference the data? That is, which reference site will give us the true waveforms?

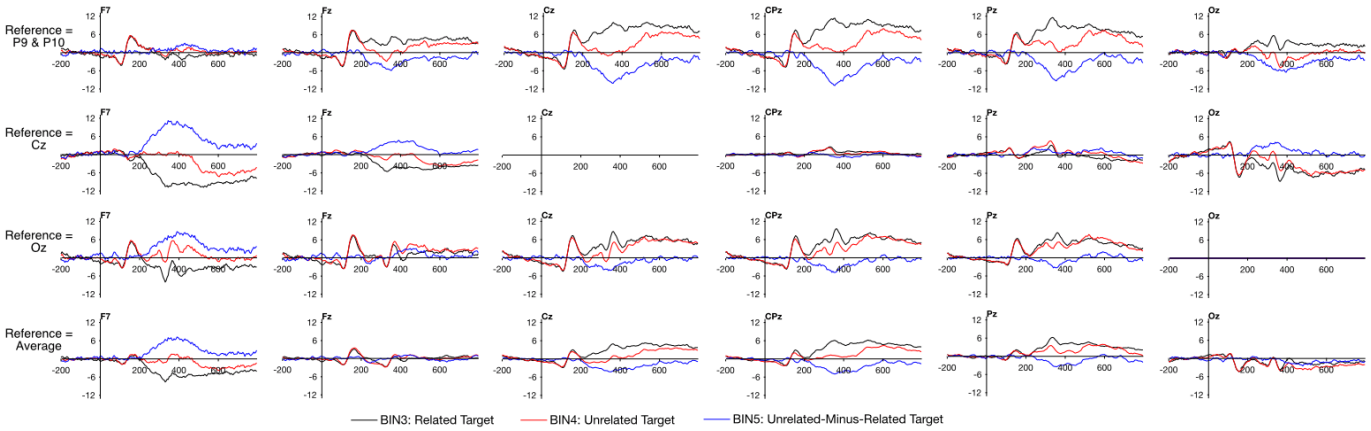


Figure 5.2. Grand average ERP waveforms for selected channels from the N400 ERP CORE experiment with four different references.

If you think about it, these are not meaningful questions. Voltage is the potential for electrical current to flow between two places. It makes no sense to talk about the voltage at a single electrode site. Even if we could measure (and not merely estimate) the absolute voltage, it would have no special truth status. For example, the cortical generator site of a given ERP component could be quite far away from the electrode with the largest voltage.

Even if there is no "correct" way to reference the data, is there a "best" way? I discuss this issue in detail in Chapter 5 of Luck (2014). My bottom line is that the best approach is usually to use whatever is most common in a particular area of research. If you don't use the same reference as most other studies, then people can't compare your data with the data from other similar studies. And you might think you've discovered a new effect. For example, most language ERP studies use the average of the mastoids as the reference, and if you use the average reference, you might think you've discovered a new "P400" component at the F7 electrode site.

If you have a good reason to use an atypical reference in a given study, you should also show what the data look like with the typical reference (e.g., by providing the waveforms with the typical reference in online supplementary materials). That way, no one will be confused about the relationship between your study and other studies.

My final piece of practical advice about the reference electrode—especially when you're first starting out in ERP research or looking at a new component—is to look at your data with multiple different references (as in Figure 5.2). That way, you won't be lulled into thinking that the waveform in a given channel primarily reflects brain activity at the so-called active electrode for that channel. That is, you'll see that the reference has a big effect on your data, and you'll realize that no matter what reference you use, you're looking at the potential between two sites.

I've made this recommendation to many people, but I don't know how many of them have followed my advice. However, I do know that the people who have followed it and then discussed their experiences with me said that it really helped them understand their data better. So give it a try!

This page titled 5.7: What is the Best Reference Site? is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.8: Exercise- Current Density

The need to use a reference electrode to measure voltage can sometimes make it difficult to answer the scientific question of interest. Fortunately, there are two common ways of transforming the data into a reference-free signal, namely converting the voltage to *current density* or computing the *global field power*. I've used both transformations, and they can be quite useful.

Let's start with current density (also called *current source density*). Unlike voltage, which always involves two places, current is the flow of charges at a single point. There is no reference for measures of current. Unfortunately, we can't directly measure the current flowing out of the scalp at a given electrode site. But fortunately, we can *estimate* the current flow from the pattern of voltage across a set of electrodes. To estimate the current flow perpendicular to the scalp (the *current density* or *current source density*) at a given time point, we apply the *Laplacian* transform to the distribution of voltage across the scalp at that time point. The details are described in Chapter 7 of Luck (2014). Here, we'll see how it's actually done using ERPLAB.

Load **Grand_N400_diff.erp** into ERPLAB if it's not already loaded, and make sure that it's the active ERPset. Plot the Bin 5 (unrelated minus related target) waveforms, and keep the plot window open so that you can compare the voltages in this ERPset with the current density values that we'll create.

The Laplacian transform requires that the 3-dimensional locations of the electrodes are specified in the ERPset. They should already be present in **Grand_N400_diff.erp**, and we provide a tool for adding them to your own data (**EEGLAB > ERPLAB > Plot ERP > Edit channel location table**). The Laplacian transform also requires that all channels have the same reference, so we'll need to eliminate the bipolar VEOG and HEOG channels from our data. To do this, select **EEGLAB > ERPLAB > ERP Operations > ERP Channel Operations**, clear out any equations that remain in the text box from the last time you used this routine, make sure that **Try to retain location information** is checked, click the **Remove Channel(s)** button, and specify **29 30** as the indices of the channels to be removed.

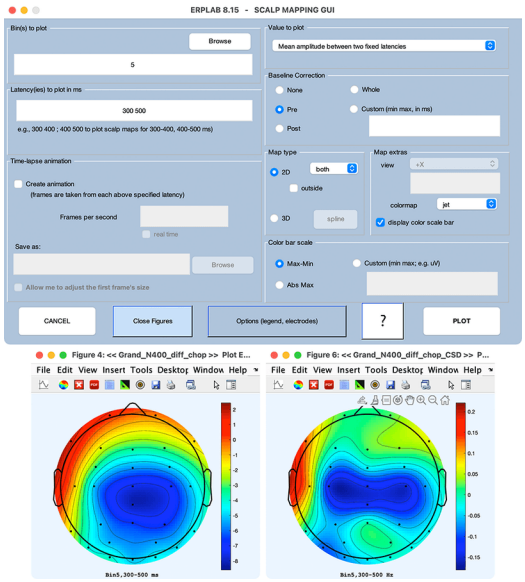
Now we're ready to convert from voltage to current density. Select **EEGLAB > ERPLAB > Datatype Transformations > Compute Current Source Density (CSD) data from averaged ERP data**. It will bring up a window that shows your electrode locations (so that you can make sure they're correct) and has some parameters. Just leave the parameters at their default values and click **Generate CSD**.

Now plot the Bin 5 (unrelated minus related target) waveforms for this new ERPset. If you look at the CPz channel, you'll see an N400 (a negativity peaking around 400 ms). However, if you look at the surrounding sites, you'll see that N400 current density has a much more focused scalp distribution than the N400 voltage that you plotted prior to performing the Laplacian transformation. For example, the N400 is quite large at Pz in the voltage waveforms but near zero in the current density waveforms. This is typical: The Laplacian transformation creates are narrower scalp distribution. This is sometimes very useful, because it allows us to separate components that have different but overlapping voltage distributions. Once we convert voltage to current density, the components may be at distinct sites, allowing us to measure them separately.

We can see this better by plotting scalp maps. Let's start with the voltage. Select the original ERPset (with the EOG channels removed) in the ERPsets menu, and select EEGLAB > ERPLAB > Plot ERP > Plot ERP scalp maps. Set the plotting parameters as shown in Screenshot 5.3. We're going to plot the difference wave (Bin 5), using the mean voltage from 300 to 500 ms. When everything is set, click **PLOT**, and you should see a scalp map like the one in the lower left of Screenshot 5.3. Note that the negative voltage is centered at the CPz electrode site and broadly distributed, with a slight bias toward the right hemisphere (which is typical for the N400).

Now select the ERPset with the current density and repeat the procedure for plotting the scalp map. The result should like the map in the lower right of Screenshot 5.3. Note that the negativity is now much sharper, and you can actually see separate foci over the left and right hemispheres. The most important thing, however, is that we are now looking at (an estimate of) the current flowing out of the scalp at each location, not a potential between each location and the reference site. The location of the original reference site no longer matters.

Screenshot 5.3

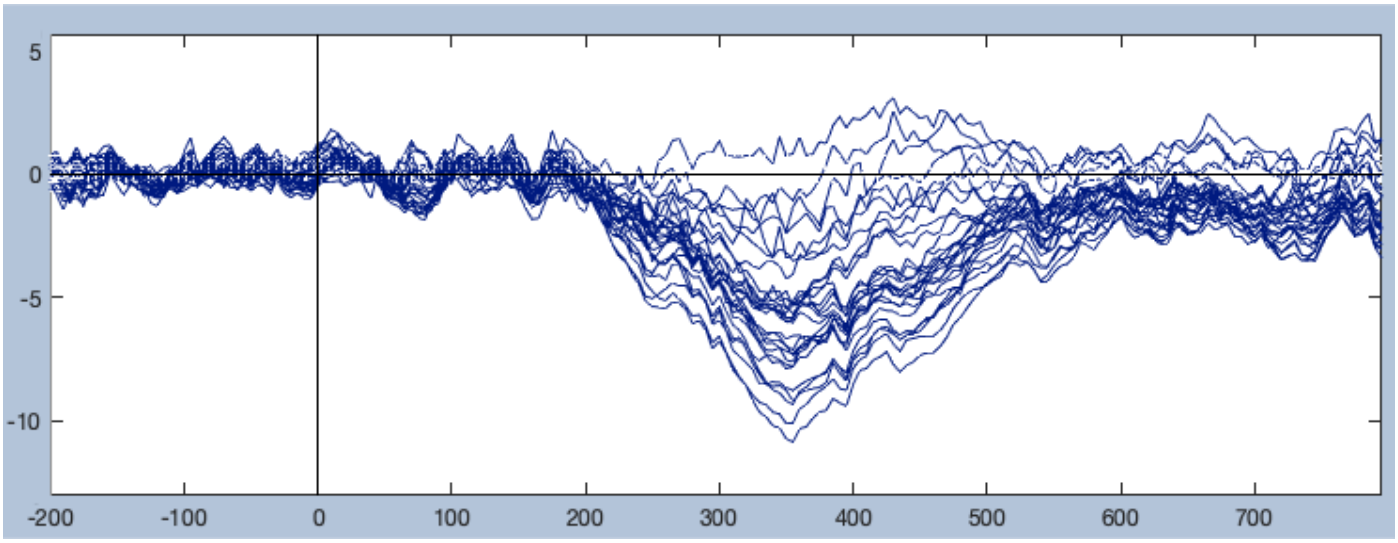


This page titled 5.8: Exercise- Current Density is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.9: Exercise- Global Field Power

Another useful reference-free transformation is *mean global field power*. If you have a reasonable number and spread of electrodes, any given ERP component will produce a systematic gradient in the amplitude across electrode sites that is proportional to the amplitude of the internal generator. For example, Screenshot 5.4 (which I generated with the Viewer in the Measurement Tool) overlays all 28 scalp electrodes for the unrelated-minus-related difference wave in the N400 experiment (referenced to the average of P9 and P10). The spread of voltage values across electrode sites is proportional to the amplitude of the N400. We can quantify this spread by taking the standard deviation across sites at any given time point. This standard deviation is called the global field power or GFP. Because the reference electrode contributes equally to each channel, it is effectively a constant and has no impact on the GFP.

Screenshot 5.4

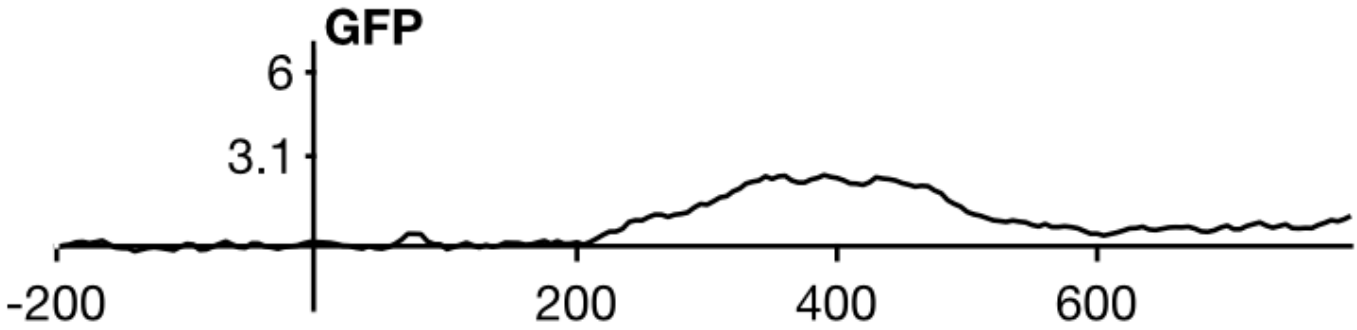


Let's compute the GFP for the data in the N400 data. **L**oad **Grand_N400_diff.erp** into ERPLAB if it's not already loaded, and make sure that it's the active ERPset. Select **EEGLAB > ERPLAB > ERP Operations > ERP Channel Operations**, clear out any equations that remain in the text box from the last time you used this routine, and change the **Mode** to **Modify existing ERPset**. We're going to create a new channel (channel 31) with the GFP for the EEG channels (channels 1–28). To do this, put the following equation in the **ERP Channel Operations** text box:

ch31 = mgfperp(1:28) label GFP

Now click **RUN**, and then plot the data from Bin 5. At the bottom of the plot, you should see a channel labeled GFP (see Screenshot 5.5). This waveform is the standard deviation across channels 1–28 at each time point, and you can see that the time course matches the time course of the difference wave at the other electrode sites. However, when we look at the GFP, we no longer have to worry about that pesky reference electrode issue.

Screenshot 5.5



GFP has some other virtues as well. For example, it is typically cleaner than the individual-channel waveforms (because noise is typically minimized by transformations that combine the data from multiple sites). In addition, rather than having to choose which electrode site or sites to use in your statistical analyses (which can be a source of bias), you can just measure the amplitude or latency from the GFP waveform (Hamburger & Van der Burgt, 1991). However, you should keep in mind that the standard deviation across channels will increase as the noise level increases, so special methods are necessary to compare GFP amplitudes across conditions that differ in the number of trials or any other factor that might impact the noise level (Files et al., 2016).

This page titled 5.9: Exercise- Global Field Power is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by LibreTexts.

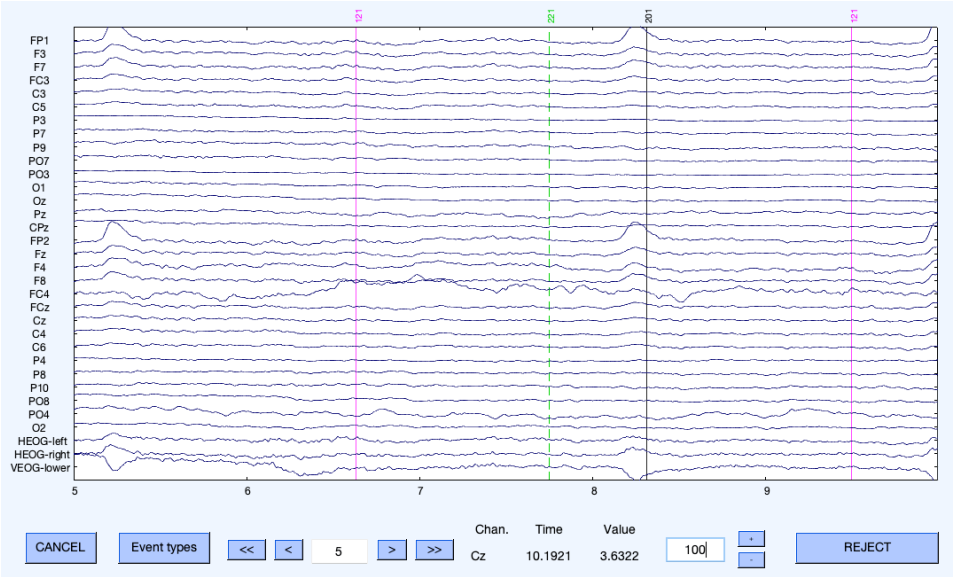
5.10: Exercise- Referencing the EEG Data from the ERP CORE N400 Experiment

For the sake of simplicity, the preceding exercises in this chapter were performed on grand average ERP waveforms. This is fine for visualization, but you will ordinarily need to re-reference the single-subject data. The kinds of re-referencing we've been discussing so far in this chapter can be applied to the continuous EEG, the epoched EEG, or the averaged ERPs. In many cases, however, you will need to re-reference the EEG prior to artifact rejection or correction. In this exercise, we'll see how to re-reference the continuous EEG.

Before you start this exercise, I recommend quitting and restarting EEGLAB so that everything is fresh. You won't need any of the data you created in the previous exercises.

In the Chapter_5 folder, you'll find a dataset named **6_N400_unreferenced.set**. This is EEG dataset for the participant we looked at in Chapter 2. In that chapter, we looked at a dataset that had already been referenced, but this version has not be referenced. However, it has been filtered (0.1–30 Hz). Launch EEGLAB and load this file (**EEGLAB > File > Load existing dataset**) and take a look at the EEG (**EEGLAB > Plot > Channel data (scroll)**). Set the vertical scale to 100 µV, and click the **>>** button once to scroll to the 5 second point. You should see something like Screenshot 5.6.

Screenshot 5.6



These data were recorded with a BioSemi ActiveTwo EEG recording system, which saves the single-ended data rather than saving the referenced data. So, the voltages that you're looking at are the raw voltages between each active electrode and the ground electrode (or, more precisely, the *common mode sense* electrode, which is BioSemi's equivalent of ground). You can see EEG from the P9 and P10 channels, which were used as the reference in data from the previous exercises. In the present exercise, we'll reference (not re-reference) the data to the average of P9 and P10.

We'll also create special *bipolar* versions of the horizontal EOG and vertical EOG (HEOG and VEOG) channels. To understand why this is useful, take a look at the channel labeled VEOG-lower at the bottom of the plot. This electrode was located just below the right eye, and the negative-going deflection that you can see in this channel shortly after the 5 second mark is an eyeblink. If you look at the FP1 and FP2 channels at the same time, you'll see a positive-going deflection. This pattern occurs because eyeblinks (and vertical eye movements) arise from a dipole located inside the eyes, and electrodes under versus over the eyes are on opposite sides of this dipole, yielding opposite polarities.

Brain activity spreads over the entire head, so the VEOG-lower electrode picks up brain activity as well as the electrooculogram

(EOG) voltage produced by eyeblinks. However, most brain activity will be quite similar at electrodes just above and just below the eye. We can therefore isolate the EOG activity and largely eliminate the brain activity by subtracting the FP2 signal (just above the right eye) from the VEOG-lower signal (just below the right eye). In addition, because the blink activity is positive at FP2 and negative at VEOG-lower, this subtraction also increases the size of the blink activity. When we try to reject trials with blinks, this subtraction makes our job much easier, because it makes the blinks bigger and makes non-blink EEG activity smaller. Thus, when we reference the data from our EEG electrodes, we'll also create a *bipolar VEOG* channel in which we subtract FP2 from VEOG-lower.

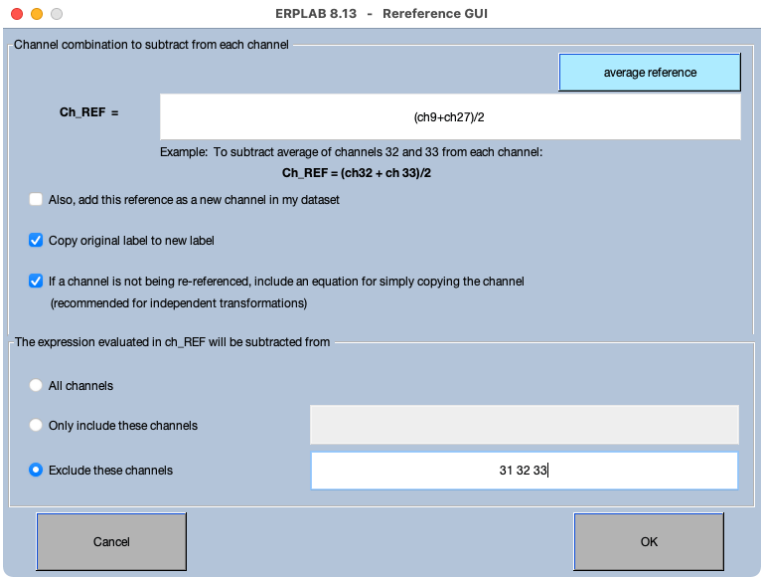
All EEG voltages are actually "bipolar" in the sense of having two poles (active and reference). However, the term *bipolar* is used in EEG recordings when a channel uses a special reference that is different from the other channels.

We also usually create a bipolar HEOG signal to isolate horizontal eye movements. During a typical EEG recording, we place one HEOG electrode next to the left eye (HEOG-left) and another next to the right eye (HEOG-right). When the eyes move leftward, this produces a negative voltage at HEOG-left and a positive voltage at HEOG-right. This reverses for rightward eye movements. By creating a bipolar channel (HEOG-right minus HEOG-left), we can effectively double the size of the eye movement voltage. In addition, brain activity is usually quite similar at these two sites, so this subtraction also eliminates most of the brain activity. Thus, the bipolar HEOG signal is very useful when we try to reject trials with horizontal eye movements.

Enough talk—let's try it! We'll reference each scalp channel to the average of P9 and P10, and we'll create bipolar VEOG and HEOG channels. With the **6_N400_unreferenced.set** dataset loaded and active, select **EEGLAB > ERPLAB > EEG Channel Operations**. This is nearly identical to ERP Channel Operations, but it operates on the EEG (whether continuous or epoched). Clear out any existing equations and set the mode to **Create new dataset**.

Click the **Reference assistant** button and type **(ch9+ch27)/2** into the **Ch_REF** text box. P9 is in Channel 9, and P10 is in Channel 27, so this expression gives us the average of P9 and P10. Check the boxes shown in Screenshot 5.7. Even the EOG channels will benefit from having a reference electrode, so indicate that **All channels** should be included. The click **OK** to create the equations. In the main **EEG Channel Operations** GUI, you should now see that each channel being created will be computed as the original channel minus the average of P9 and P10.

Screenshot 5.7



Now we need to add equations for creating the bipolar EOG signals. For reasons that will become clear in the chapter on artifact correction, it's often a good idea to have both the bipolar signals and the signals referenced to the average of P9 and P10. To make this happen, add the following two equations to the list of equations in the **EEG Channel Operations** GUI:

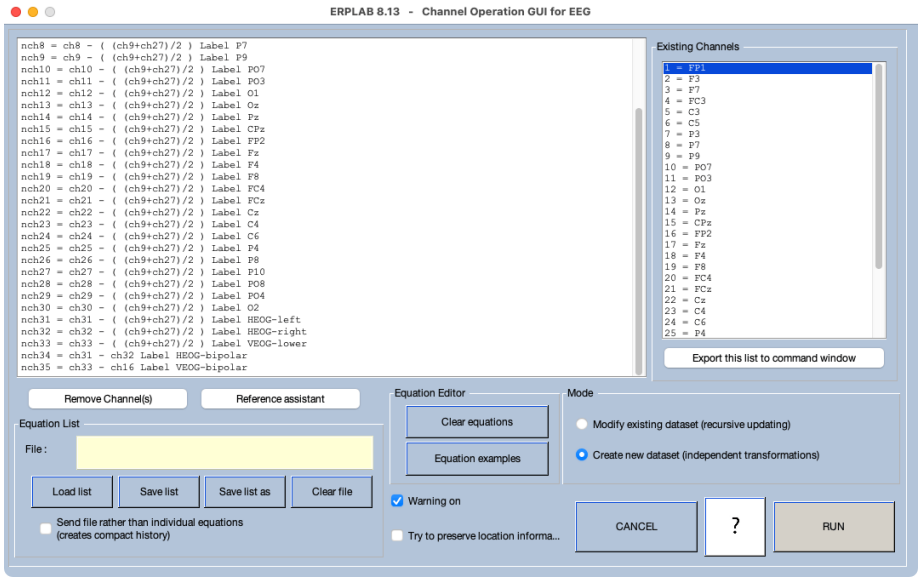
nch34 = ch31 - ch32 Label HEOG-bipolar

nch35 = ch33 - ch16 Label VEOG-bipolar

The new Channel 34 will be HEOG-right minus HEOG-left, and the new Channel 35 will be VEOG-lower minus FP2 (which is just above the right eye). The list of equations should look like that shown in Screenshot 5.8. Click **RUN**, and name the new dataset **6_N400_ref** to indicate that it has now been referenced. You'll want to refer to this dataset in the next exercise, so save it as a file if you're not going to do the next exercise right away.

Now plot the data with **EEGLAB > Plot > Channel data (scroll)**. The most obvious change is that the bipolar EOG channels are now present. You can see that blinks are larger in the VEOG-bipolar channel than in the VEOG-lower or FP2 channels. The N400 task used stimuli presented in the center of the monitor, so there aren't any obvious horizontal eye movements. We'll see what those look like in the chapter on artifact rejection.

Screenshot 5.8



This page titled 5.10: Exercise- Referencing the EEG Data from the ERP CORE N400 Experiment is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.11: Exercise- Other Common Re-Referencing Scenarios

The previous exercise began with the single-ended data produced by the BioSemi system we used for the ERP CORE experiments, so we *referenced* the data rather than *re-referencing* the data. Most EEG systems give you differential (referenced) data rather than single-ended data, but it is often necessary to re-reference the data from these systems offline . In this exercise, we'll see how to implement two common re-referencing scenarios you might encounter with differential data. In the first scenario, the data were recorded with a left mastoid reference, but you want to use the average of the left and right mastoids as the reference. In the second, the data were recorded with Cz as the reference (which is the default in the EGI system), and you again want to use the average of the mastoids as the reference. Another common scenario would be to take referenced data and re-reference to the average of all sites, but we already saw how to do that in an earlier exercise. (That exercise used averaged ERPs rather than EEG, but re-referencing works the same with ERP data and EEG data.)

I've created two versions of the EEG data from the preceding N400 example for these two scenarios, and you can find them in the Chapter_5 folder. The file named **6_N400_LmRef.set** has EEG and EOG data referenced to the left mastoid (including an Rm channel that has EEG data from the Rm site that were referenced to Lm). The file named **6_N400_CzRef.set** has EEG and EOG data referenced to Cz (including **Lm** and **Rm** channels that were referenced to Cz). Go ahead and load these two files into EEGLAB.

Where did these datasets come from?

I don't actually have files for the N400 experiment that were recorded using Lm or Cz as the reference, so I created approximations by applying Channel Operations to the single-ended data. I simply relabeled the P9 and P10 channels as **Lm** and **Rm**. That is, we're just going to pretend that P9 was actually Lm and P10 was actually Rm. In **6_N400_LmRef.set**, we'll pretend that the data were referenced to Lm (even though I actually referenced the data to P9). This file contains a channel labeled **Rm**, which we're pretending is the voltage between Rm and Lm (but is actually the voltage between P10 and P9). In **6_N400_CzRef.set**, I simply referenced the data to to Cz (including channels labeled **Lm** and **Rm** that actually have the data from P9 and P10, referenced to Cz).

Let's start by re-referencing the data in **6_N400_LmRef.set** to the average of the left and right mastoids. As we saw in the exercises using the spreadsheet, we can do this by simply subtracting 50% of the voltage between Rm and Lm from the voltage in each channel. Make sure the **6_N400_LmRef.set** dataset is active and then select **EEGLAB > ERPLAB > EEG Channel Operations**. Clear out any existing equations and make sure the mode is set to **Create new dataset**. You can then use the **Reference assistant** to create the appropriate equation for each channel. But this time, I'm not going to tell you how to do it; you should figure it out for yourself. You can look at the list of **Existing Channels** in the Channel Operations GUI to figure out the channel number for the Rm signal. (If you get stuck, the equations are in a file named **Re-Reference_Lm.txt**. But you'll learn a lot more if you figure it out for yourself.) Once you're done, you can plot the data. The waveforms should look nearly identical to those you created in the previous exercise, with the average of P9 and P10 as the reference.

If you did it the same way I did, you'll still have a channel labeled **Rm**, but now it's the voltage between Rm and the average of Lm and Rm. It's not a very useful channel, but it doesn't hurt to keep it. Alternatively, you could keep the original signal, with Rm referenced to Lm. Or you could just eliminate that channel.

Now let's re-reference the data in **6_N400_CzRef.set** to the average of the left and right mastoids. This is pretty simple: We just need to subtract the average of the two mastoids from each channel. To make it a little more challenging, you should also include an equation to recover the Cz signal, referenced to the average of the two mastoids. Again, I'm not going to tell you how to do it, but here's a hint for recreating Cz: the voltage at Cz with Lm as the reference is the same as -1 times the voltage at Lm with Cz as the reference. Don't forget to look at the list of **Existing Channels** in the Channel Operations GUI to figure out the channel numbers for the Lm and Rm signals. (If you get stuck, the equations are in a file named **Re-Reference_Cz.txt**.) Again, when you plot the data, the waveforms should look nearly identical to those you created in the previous exercise. Pay particular attention to the Cz channel to make sure the one you just created looks like the original.

This page titled 5.11: Exercise- Other Common Re-Referencing Scenarios is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.12: Matlab Script For This Chapter

I've provided a script called **referencing_examples.m** in the Chapter_5 folder. This script implements the EEG referencing/re-referencing procedures from the last few exercises. It uses equations stored in text files, which can be quite convenient.

This page titled 5.12: Matlab Script For This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

5.13: Key Takeaways and References

Key Takeaways

- Voltage is the potential for charges to flow from one location (the so-called active site) to another location (the so-called reference site). There is no such thing as the voltage at a single electrode site.
- When you're looking at data from a channel that is labeled with the so-called active electrode site for that channel (e.g., Pz), the waveforms are impacted equally by activity the so-called active and so-called reference sites. If you don't know what the reference was, you really don't know what you're looking at. You should always find out what reference was used before you spend time looking at ERP waveforms
- There is no "correct" reference location. No matter what you use, the data are impacted equally by the so-called reference and so-called active electrodes.
- The "best" reference is usually whatever is common in your subarea (because using the same reference facilitates comparisons across studies).
- To avoid falling into the trap of thinking that the waveform from a given channel is primarily a result of the so-called active electrode for that channel, it can be helpful to look at your data with multiple different reference sites. For this purpose, it is usually sufficient to re-reference the grand average ERPs.
- It is simple to re-reference data offline. In many cases, you can make one of your channels (or the average of a set of channels) the reference by simply subtracting it from the other channels (assuming that they all started with the same reference).
- You can avoid the reference issue altogether by converting your voltage waveforms into current density or global field power.

References

Dien, J. (1998). Issues in the application of the average reference: Review, critiques, and recommendations. *Behavior Research Methods, Instruments & Computers*, *30*, 34–43.

Dong, L., Li, F., Liu, Q., Wen, X., Lai, Y., Xu, P., & Yao, D. (2017). MATLAB Toolboxes for Reference Electrode Standardization Technique (REST) of Scalp EEG. *Frontiers in Neuroscience*, *11*. https://doi.org/10.3389/fnins.2017.00601

Files, B. T., Lawhern, V. J., Ries, A. J., & Marathe, A. R. (2016). A Permutation Test for Unbalanced Paired Comparisons of Global Field Power. *Brain Topography*, *29*, 345–357. https://doi.org/10.1007/s10548-016-0477-3

Hamburger, H. L., & Van der Burgt, M. A. G. (1991). Global Field Power measurement versus classical method in the determination of the latency of evoked potential components. *Brain Topography*, *3*(3), 391–396. https://doi.org/10.1007/BF01129642

Kappenman, E. S., Farrens, J. L., Zhang, W., Stewart, A. X., & Luck, S. J. (2021). ERP CORE: An Open Resource for Human Event-Related Potential Research. *NeuroImage*, *225*, 117465. https://doi.org/10.1016/j.neuroimage.2020.117465

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Nunez, P. L. (1981). *Electric Fields of the Brain*. Oxford University Press.

Rossion, B., & Jacques, C. (2012). The N170: Understanding the time course of face perception in the human brain. In S. J. Luck & E. S. Kappenman (Eds.), *The Oxford Handbook of Event-Related Potential Components* (pp. 115–141). Oxford University Press.

Skrandies, W. (1989). Data reduction of multichannel fields: Global field power and Principal Component Analysis. *Brain Topography*, *2*(1), 73–80. https://doi.org/10.1007/BF01128845

Yao, D. (2017). Is the Surface Potential Integral of a Dipole in a Volume Conductor Always Zero? A Cloud Over the Average Reference of EEG and ERP. *Brain Topography*, *30*(2), 161–171. https://doi.org/10.1007/s10548-016-0543-x

This page titled 5.13: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6: ASSIGNING EVENTS TO BINS, AVERAGING, BASELINE CORRECTION, AND ASSESSING DATA QUALITY

Learning Objectives

In this chapter, you will learn to:

- Design experiments that avoid subtle sensory confounds by following the Hillyard Principle
- Create bin descriptor files for assigning events to bins
- Predict how the signal-to-noise ratio will change as you change the number of trials being averaged together
- Assess how the number of trials and the baseline period impact the data quality as quantified with the standardized measurement error (SME)
- Create response-locked as well as stimulus-locked averages
- Compare the ERPs for correct trials and error trials
- Conduct sequential analyses
- Evaluate the impact of overlapping activity from the previous trial
- Implement a sequential analysis using a script

This chapter takes a close look at how events are assigned to bins. This is not something that gets discussed a lot in the ERP literature, even in methodology papers, but it's absolutely fundamental. After all, "event" is the first word in "event-related potential." Also, ERPLAB's tool for assigning events to bins (BINLISTER) is fairly powerful but also not very user-friendly, so we'll want to make sure you have a good grasp on how it works.

ERPLAB's online documentation contains a detailed manual page on BINLISTER, and you should read it if you want to learn all the details of how this important routine works. I won't repeat all those details here. In fact, I even wondered if it was worthwhile writing a chapter on bin assignment for this book. But I decided that the best way to learn about something like this is to actually use it. And once I started creating the exercises, I realized that they brought up some concepts about ERP data analysis that are important even if you end up using a different software package for analyzing your data.

This chapter will focus on the visual oddball P3b experiment from the ERP CORE, including an analysis of sequential effects and an analysis of the error-related negativity produced on error trials. The oddball paradigm has probably been used more than any other ERP paradigm over the years, so it's good to have a thorough understanding of it. The particular version of this paradigm that we implemented for the ERP CORE contains some subtleties that are useful for learning about the design of ERP experiments. And the oddball paradigm is particularly well suited for exploring some of the issues that come up in assigning events to bins. We'll start by taking a close look at the details of the experimental paradigm. Then we'll perform several different analyses of one participant's data so you can see some of the different ways that events can be assigned to bins and the issues that arise in this fundamental step.

Along the way, we'll see some important issues that arise in averaging, including overlap from previous trials. We'll also take a closer look at the baseline correction procedure, which seems simple but can end up creating problems in some situations. We'll also look carefully at how data quality varies according to the number of trials being averaged together and according to the specific time period being used for baseline correction.

6.1: Data for This Chapter

6.2: Design of the ERP CORE Visual Oddball P3b Experiment

6.3: The Event Code Scheme

6.4: Overview of Bin Descriptor Files

6.5: Exercise - A Basic Assignment of Events to Bins

6.6: Exercise - Looking at the Averaged ERPs

6.7: Exercise - The Signal-to-Noise Ratio
6.8: Exercise - Response-Locked Averaging
6.9: Exercise - Comparing Correct and Error Trials
6.10: Exercise - Sequential Analysis of the P3b
6.11: Exercise - Combining Bins
6.12: Exercise - Overlap
6.13: Matlab Script For This Chapter
6.14: Key Takeaways and References

This page titled 6: Assigning Events to Bins, Averaging, Baseline Correction, and Assessing Data Quality is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.1: Data for This Chapter

The data we'll be using for the exercises in this chapter can be found in the Chapter_6 folder in the master folder: https://doi.org/10.18115/D50056.

We'll focus on the data from one participant (Subject 12), but the data from all participants who were included in the final analysis are also provided in a folder named **Data**. To keep things simple, some preprocessing steps have already been applied to these data (as will be described later).

Also, whereas the data in the previous chapters were downsampled from the original sampling rate of 1024 Hz to 200 Hz, the data for this chapter were downsampled to 256 Hz, as in the published data. This creates some slight complications, but those complications will be informative.

This page titled 6.1: Data for This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.2: Design of the ERP CORE Visual Oddball P3b Experiment

In a typical oddball experiment, there are two categories of stimuli, one of which is rare (called the *oddballs* or *targets*) and one of which is frequent (called the *standards* or *nontargets*). For example, you might present a sequence of Xs and Os in the middle of a video monitor, with 10% Xs and 90% Os. The Rare category will elicit a much larger P3b component than the Frequent category, but only if participants are actively discriminating between the two categories. For example, if participants are asked to judge the colors of the Xs and Os, you won't see a larger P3 to the Rare Xs than to the Frequent Os. However, if participants are instructed to press one button for Xs and another button for Os, this requires making an active categorization of each stimulus as an X or an O, and you'll see a much larger P3b for the Xs than for the Os. However, it's not the motor response per se that leads to the P3b. The Xs will also produce a larger P3b than the Os if the task is to press a button for the Os and make no response for the Xs.

Researchers have known since the 1960s that the amplitude of the P3b is inversely related to the probability of the eliciting stimulus. That is, the P3b will be larger if the oddballs are 20% probable than if they are 30% probable, and the P3b will be even larger if the oddballs are 10% probable (Duncan-Johnson & Donchin, 1977). A fundamentally important but not widely appreciated fact is that it is not the probability of the physical stimulus that determines P3b amplitude, but instead the probability of what I call the *task-defined category*. For example, my lab once ran an oddball experiment in which 15% of the stimuli were the letter E and the other 85% were randomly selected from all the non-E letters (Vogel et al., 1998). The task was to press one button for the letter E and another button for non-E letters. The letter E was more common than any other individual letter, but the task required participants to categorize each stimulus as E or non-E, and the E category was less frequent than the non-E category. As a result, the E stimuli elicited a much larger P3b than the non-E stimuli. You can learn more about the P3b component in Chapter 3 of Luck (2014) or in John Polich's chapter on the P3 family of components in the Oxford Handbook of ERP Components (Polich, 2012).

Many oddball experiments contain an obvious confound: If 10% of the stimuli are Xs and 90% are Os, then the Xs and Os differ both in the shape of the letter and the probability of occurrence. This probably doesn't have much impact on the P3b component, but confounds like this are easy to avoid, so I'm always surprised that so many experiments have this confound. An easy way to solve this is to counterbalance the probabilities: Use 10% Xs and 90% Os for half the trial blocks and 90% Xs and 10% Os for the other half. This makes it possible to compare the ERP elicited by an X when it is Rare to the ERP elicited by an X when it is Frequent. And we can do the same for the Os. In other words, we can hold the stimuli constant and vary only the probability. To make things simpler, we can average the X-Rare and O-Rare ERPs together and compare the result to the average of the X-Frequent and O-Frequent ERPs. Many experiments use this approach (but not as many as I would like!).

However, as illustrated in Figure 6.1, a subtle *adaptation* confound still remains when this counterbalancing is used. Imagine that X is Rare and O is Frequent in the first trial block, and participants press one button for the Xs and another for the Os. Neurons in visual cortex that are sensitive to the O shape will tend to become adapted by the frequent occurrence of the O stimuli, but neurons that are sensitive to the X shape will not become adapted given the infrequent occurrence of this shape. As a result, the sensory response will be smaller for the O stimuli than for the X stimuli. In the second trial block, O is Rare and X is Frequent. Now the X-sensitive neurons become more adapted than the O-sensitive neurons, and the sensory response will be smaller for the Xs than for the Os. So, even though we have counterbalanced which stimulus is Rare and which is Frequent, we still get a larger sensory response for the Rare category than for the Frequent category.

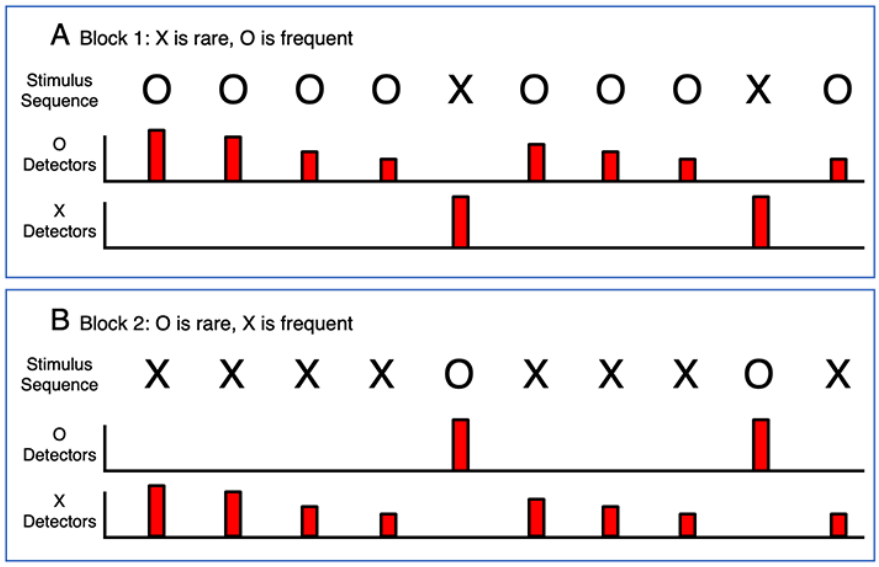


Figure 6.1. Example of the sensory adaptation confound that is present in nearly all oddball experiments. The height of each bar represents the magnitude of the sensory response to a given stimulus.

This exemplifies what I call the *Hillyard Principle* of experimental design: Keep the stimuli constant and vary only the psychological conditions. To follow this principle, you must keep the entire sequence constant across conditions, which we are not doing in the counterbalanced design shown in Figure 6.1. That is, we are using different sequences of stimuli to create our Rare and Frequent categories, and any differences in the ERPs for these categories could be a result of physical stimulus differences rather than the psychological categories of Rare and Frequent.

The Hillyard Principle

The Hillyard Principle is named after my PhD advisor, Steve Hillyard. When I was a grad student, we were constantly reminded to keep the stimuli constant and vary the psychological conditions (usually by varying the instructions). Steve was a master of experimental design, and he had a huge impact on the field by developing extremely rigorous designs (and instilling this ethos into his many graduate students and postdocs).

The sensory ERP components are very sensitive to small differences in stimuli, and the Hillyard Principle is especially important when you see differences between conditions at relatively short latencies (e.g., <200 ms after stimulus onset). If an experiment does not follow the Hillyard Principle, it's usually impossible to interpret any early effects (unless, of course, the goal of the experiment was to examine the effects of stimulus manipulations on sensory activity). However, it's a good idea to follow the Hillyard Principle even when you're looking at later effects, because it's difficult to be 100% certain that a late effect isn't a consequence of an early sensory confound.

Some experimental questions are difficult to answer while following the Hillyard principle. For example, imagine that you wanted to compare the ERPs elicited by nouns and verbs (presented as auditory speech signals). You don't have any control over what nouns and verbs sound like, and it would be difficult to create instructions that make a participant treat the word "chair" as a noun in some trial blocks and a verb in other blocks. But if you were really motivated, you could actually achieve this kind of control by using two groups of participants who spoke different languages. For example, if you compared monolingual English speakers and monolingual Mandarin speakers, you could ask whether nouns and verbs that are known by an individual produce a difference that is not present for nouns and verbs that are unknown by that individual.

In the ERP CORE, we designed the P3b experiment to follow the Hillyard principle. As illustrated in Figure 6.2, the stimuli were the letters A, B, C, D, and E, presented in random order in the center of the video display. Each of these letters was 20% probable. Each participant received 5 trial blocks, each containing 40 trials, and a different letter was designated the target in each block. They were instructed to press one button for the target letter and another button for any of the four nontarget letters. For example, when D was designated the target, they would press one button for D and a different button for A, B, C, or E. As a result, D was in the Rare category when it was designated the target and was in the Frequent category when one of the other four letters was designated the target. Thus, we followed the Hillyard Principle: we kept the sequence of stimuli constant and varied only the task

instruction.

What it Means to Keep the Sequence Constant

If we had used the same exact sequence of letters in each trial block, it is possible that participants would have learned the sequence. We therefore created a new randomized sequence for each block, but these sequences were created using the same rules and differed only randomly. That rules out any **systematic** differences in the physical stimuli between trial blocks.

Figure 6.2 also shows the grand average ERP waveforms for the Rare and Frequent stimulus categories (from the Pz channel, with the average of P9 and P10 as the reference). The Rare waveform contains equal numbers of trials for which A, B, C, D, and E were designated the target. The Frequent waveform also contains equal numbers of trials for which A, B, C, D, and E were designated the target. Thus, the larger P3b observed for the Rare category than for the Frequent category must reflect the Rareness of the task-defined category, not the rareness of the physical stimuli. It took a lot of thought and effort to design the experiment this way, but I really enjoy the process of designing experiments, especially when I need to come up with some kind of creative "trick" to rule out all possible confounds.

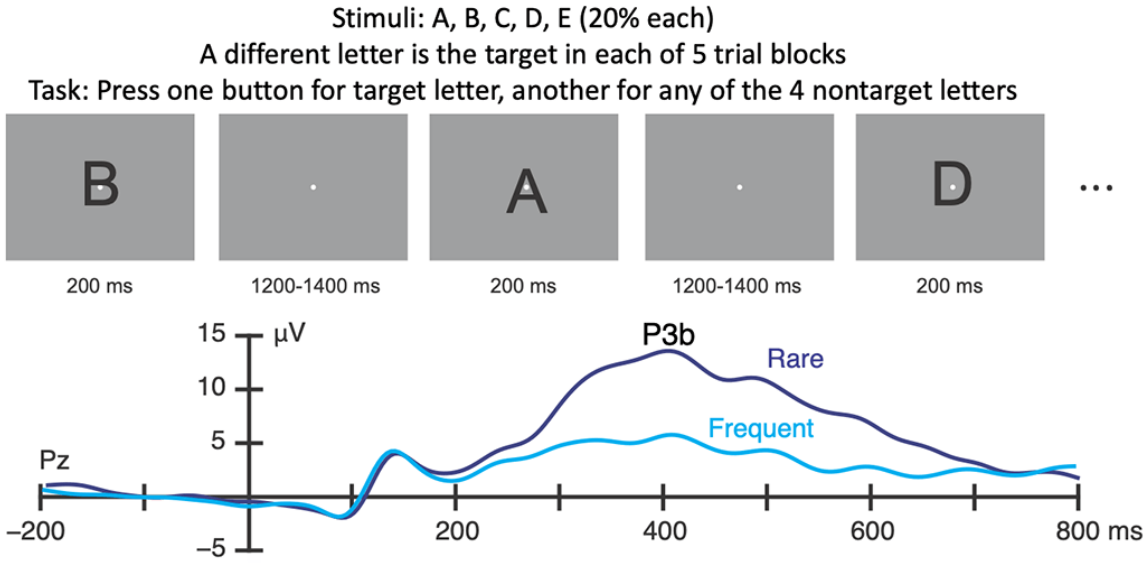


Figure 6.2. Experimental paradigm and grand average ERP waveforms from the ERP CORE visual oddball P3b experiment.

This page titled 6.2: Design of the ERP CORE Visual Oddball P3b Experiment is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.3: The Event Code Scheme

In a study like the ERP CORE P3b experiment, there are many different combinations of stimuli (5 letters) and task instructions (which of the 5 letters was the target). Some care is therefore needed to come up with a scheme for representing each possible stimulus/task combination as a separate event code using integers between 1 and 255. Table 6.1 shows our scheme. We used the tens place in the event code to indicate which letter was the target and the ones place to indicate what the current letter was. For example, event code 23 indicates that B was the target letter in the current block (because 23 has a 2 in the tens place) and that the letter C was presented on the current trial (because 23 has a 3 in the ones place). In this scheme, the Rare category is coded as event codes 11, 22, 33, 44, and 55, and the Frequent category is coded as the other combinations. We also used 201 and 202 as the event codes that were generated when the participant pressed a response button, with 201 for a correct response and 202 as an incorrect response.

Table 6.1. Event code scheme for the ERP CORE visual oddball P3b experiment.

Event Code	Target Letter for Current Block	Tens place	Current Stimulus	Ones place
11	A	1	A	1
21	B	2	A	1
31	C	3	A	1
41	D	4	A	1
51	E	5	A	1
12	A	1	B	2
22	B	2	B	2
32	C	3	B	2
42	D	4	B	2
52	E	5	B	2
13	A	1	C	3

Event Code	Target Letter for Current Block	Tens place	Current Stimulus	Ones place
23	B	2	C	3
33	C	3	C	3
43	D	4	C	3
53	E	5	C	3
14	A	1	D	4
24	B	2	D	4
34	C	3	D	4
44	D	4	D	4
54	E	5	D	4
15	A	1	E	5
25	B	2	E	5
35	C	3	E	5
45	D	4	E	5
55	E	5	E	5

This page titled 6.3: The Event Code Scheme is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.4: Overview of Bin Descriptor Files

Now let's see how event codes are assigned to bins with BINLISTER, using the very simple analysis shown in Figure 6.2 in which we have one bin for the Rare category and another for the Frequent category. We'll exclude trials in which the buttonpress response was incorrect.

As you'll recall from Chapter 2, a *bin descriptor file* is used to tell BINLISTER how event codes should be assigned to bins. In the Chapter_6 folder, you'll find a bin descriptor file named **BDF_P3.txt**, which we'll use for this exercise. Make sure that the Chapter_6 folder is the current folder in Matlab, and double-click the **BDF_P3.txt** file from the Current Folder pane in Matlab to open it. Here's what you should see:

bin 1
Rare, Correct
.{11;22;33;44;55}{t<200-1000>201}

bin 2
Frequent, Correct
.{12;13;14;15;21;23;24;25;31;32;34;35;41;42;43;45;51;52;53;54}{t<200-1000>201}

Each bin is described by a set of three lines. The first is the bin number (which must be in consecutive order, beginning with 1). The second line is the label for the bin (which can be anything you like). The third line is the actual *bin descriptor*. A bin descriptor indicates the sequence of event codes that define the bin. Each set of curly brackets ("{}") defines an *event list* that contains one or more event codes. For each bin descriptor, one event list must be preceded by a period symbol. This event list defines the time-locking event for the epoch (i.e., time zero). In the example shown above, event codes 11, 22, 33, 44, and 55 will serve as the time-locking event for Bin 1. This bin therefore includes trials with an A stimulus when A is the target, B when B is the target, C when C is the target, D when D is the target, and E when E is the target. We could have instead created a separate bin for each of these five target letters and then combined the five bins after averaging using **ERP Bin Operations**. However, it was simpler to combine them at the BINLISTER stage. The event list for Bin 2 contains all the event codes for the nontarget letters. You can verify this by comparing the event descriptors with the list of event codes in Table 6.1.

The time-locking event list may be preceded or followed by other event lists, indicating that those events must be present for an epoch of EEG to be assigned to a given bin. For example, imagine that Bin 1 was defined as:

{202}.{11;22;33;44;55}{201}

201 is the event code for a correct response and 202 is the event code for an incorrect response, so this bin descriptor would find targets (event codes 11, 22, 33, 44, and 55) that are immediately preceded by an incorrect response and immediately followed by a correct response.

In the actual bin descriptor for Bin 1, we don't require any particular event code prior to the time-locking event, but we do require that the time-locking event (the stimulus) is followed by the event code for a correct response. However, we want to make sure that the response time (RT) wasn't an outlier, indicating either a fast guess (an RT of <200 ms) or poor attention (an RT of >1000 ms). To do this, we use a *time-conditioned event list* in which the list of event codes is preceded by **t** (e.g., **t<200–1000>201** to indicate that event code 201 must be 200-1000 ms after the time-locking event).

Note that if we didn't use a time-conditioned event list and instead used **.{11;22;33;44;55}{201}** as the event descriptor, the response event code (201) would need to directly follow the stimulus event code, with no other event codes between. However, by using a time-conditioned event list to specify that the 201 must be between 200 and 1000 ms after the stimulus event code, other event codes may occur between the stimulus and the response.

Also, if a time-conditioned event list appears prior to the time-locking event, time flows backward from the time-locking event. For example, if you specify **{t<200-800>15}.{100}**, BINLISTER will search for an event code of 100 preceded by an event code of 15 that occurred 200-800 ms prior to the 100. Additional details can be found in the BINLISTER documentation.

Should you exclude trials with incorrect behavioral responses?

In this experiment, we excluded trials with incorrect behavioral responses. In other studies, however, we don't exclude the error trials. Here's the general principle: If the errors in a given task are likely the result of lapses of attention, then exclude the error trials; if the errors in a given task mainly occur because the task is very difficult, then don't exclude the error trials.

For example, errors in most oddball paradigms are typically a result of lapses of attention, so we exclude the error trials. Indeed, the ERPs are quite different on error and correct trials in the oddball paradigm (Falkenstein et al., 1990). By contrast, in tasks using the change detection paradigm to study visual working memory, most errors occur because of limits in storage capacity (Luck & Vogel, 2013). As a result, brain activity is similar on correct and error trials (Luria et al., 2016), so we don't exclude the errors (which would reduce the number of trials per bin quite a lot, decreasing the signal-to-noise ratio).

This page titled 6.4: Overview of Bin Descriptor Files is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.5: Exercise - A Basic Assignment of Events to Bins

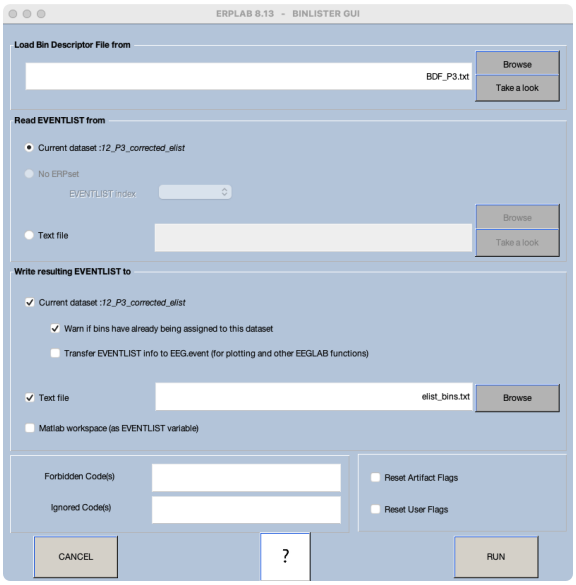
Enough theory—let's make some bins! If EEGLAB is already running, I recommend quitting it and restarting it to make sure everything is fresh. Set **Chapter_6** to be Matlab's current folder. Load the dataset named **12_P3_corrected_elist.set** (using **EEGLAB > File > Load existing dataset**). Scroll through the EEG (using **EEGLAB > Plot > Channel data (scroll)**) and familiarize yourself with it. For example, take a look at the sequence of event codes and match them with the codes in Table 6.1. When I load a dataset or ERPset, I don't do anything until I've looked at the waveforms. Many times, this visual inspection has made me realize that I have the wrong data or that there is something about the data that is incompatible with what I was planning to do next.

Do you see any eyeblinks in the VEOG-lower or FP1 channels? Did you need to remove the DC offset to see all the channels? The answer is "no" for both questions. This gives you some clues about the preprocessing steps that have already been applied to this dataset. What operations do you think have already been applied?

To keep the exercises in this chapter simple, the waveforms have had an *artifact correction* procedure applied. Instead of excluding epochs that contain blinks and eye movements, the voltages for the blinks and eye movements have been estimated and subtracted from the waveforms. That way, we won't need to throw out any trials in the exercises in this chapter. This is why the filename for the dataset has **_corrected** in it. A high-pass filter (half amplitude cutoff at 0.1 Hz) has also been applied to remove slow drifts, and the EventList was added.

We're finally read to assign the events to bins. Select **EEGLAB > ERPLAB > Assign bins (BINLISTER)**, and set it up like Screenshot 6.1. Specifically, use the **Browse** button near the top to select the bin descriptor file (**BDF_P3.txt**). We want to read the EventList from the current dataset, and we want to write the updated version with the bin information to the current dataset. We also want to write it to a text file, so check the **Text file** box and put **elist_bins.txt** in the corresponding text box. Click **RUN**, and then name the new dataset **12_P3_corrected_elist_bins**.

Screenshot 6.1



You should see the text file with the EventList in it (**elist_bins.txt**) in Matlab's Current Folder pane. Double-click on it to open it the text editor. Near the top, just under the head information, you should see this:

bin 1, # 30, Rare, Correct
bin 2, # 153, Frequent, Correct

This tells you that 30 events were found that match the bin descriptor for Bin 1, and 153 were found that match the bin descriptor

for Bin 2. How many should we have had? This is an extremely important question to answer, because errors in event codes and in assigning events to bins are quite common, and many of these errors will lead to the wrong number of trials per bin. In the task description in the beginning of the chapter, you learned that there were 5 equiprobable letters, 5 blocks of trials (one with each letter as the target), and 40 trials per block. This gives us 200 total trials. Given that 1 of the 5 letters was the target in each block, target probability was 20%. This means that we should have had 40 targets and 160 nontargets over the course of a session. Why, then, do we only have 30 instances of Bin 1 and 153 instances of Bin 2?

The answer is that Bins 1 and 2 are limited to trials with correct responses and an RT of 200-1000 ms. So, we can't use these numbers to verify that we have the correct number of event codes. As described in Chapter 2, another way that we can verify the number of event codes is to use **EEGLAB > ERPLAB > EventList > Summarize current EEG event codes**. Give that a try.

The resulting list should print in the Matlab Command Window, but it still isn't very informative. First, we have a ton of different event codes. Second, when we created the randomized sequences of events in our stimulus presentation script, we specified a certain *probability* of each letter but not a certain *number*. That is, we guaranteed that there were 8 instances of the letter A when A was the target, but the other 32 stimuli in this block were sampled completely at random from the other 4 letters. So, you can verify that we had eight targets in each block, but it's not immediately obvious that we had the right number of nontargets.

To verify that we had the right number of Rare and Frequent stimuli, make a copy of **BDF_P3.txt**, and edit this copy so that the event descriptors don't require a correct response. Then run BINLISTER again (on **12_P3_corrected_elist**) but choose a new filename for the text file that will contain the new EventList. When you look at this new EventList, you should see 40 trials in Bin 1 and 160 trials in Bin 2.

Now go back to the text file with the first EventList (**elist_bins.txt**). Every event code is listed, with the bin assignment in the **bin** column at the far right. Note that the **bin** field is empty for the response event codes, because these event codes are not used as time-locking events in this analysis. For the stimulus event codes, you should find a 1 or 2 in this field, indicating whether the stimulus was the target letter or one of the nontarget letters. However, the **bin** field is empty for some of the stimulus event codes. These are error trials. We'll take a closer look at the errors in a later exercise.

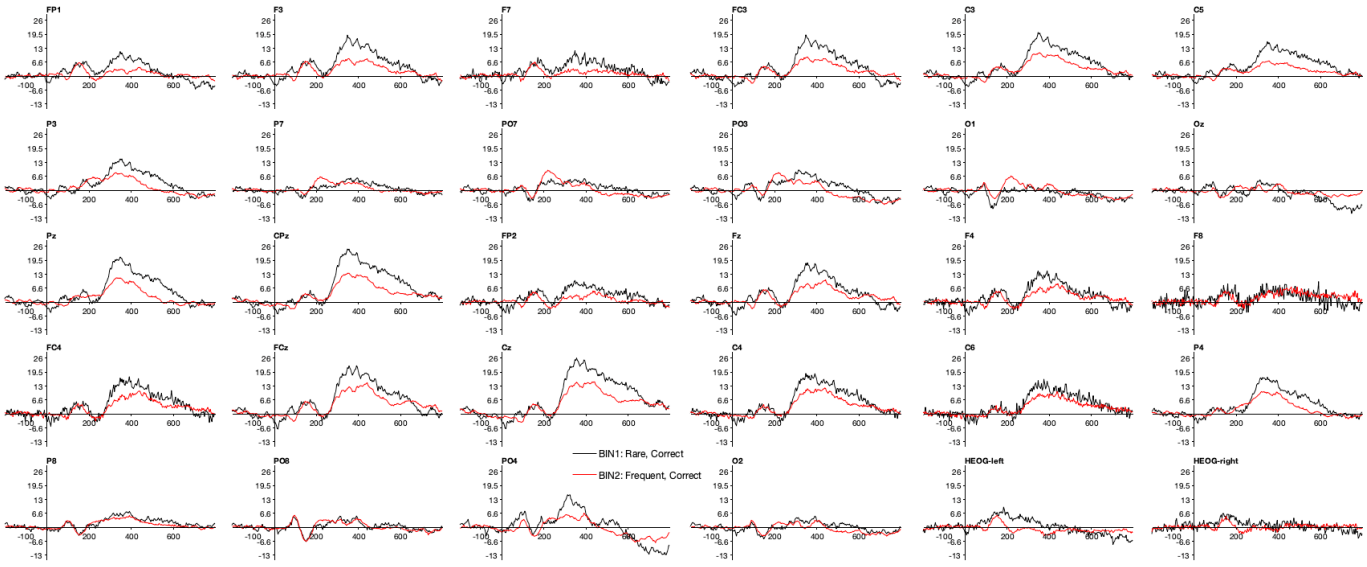
This page titled 6.5: Exercise - A Basic Assignment of Events to Bins is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.6: Exercise - Looking at the Averaged ERPs

Let's make averaged ERPs for the bins we created. We still have continuous EEG data, so we first need to divide the data into epochs, with the time-locking event code as time zero. Make sure that the correct dataset is active (**12_P3_corrected_elist_bins**) and select **EEGLAB > ERPLAB > Extract bin-based epochs**. Use the default epoch of -200 to 800 ms and the prestimulus interval as the baseline correction period. Click **RUN** and keep the default name of **12_P3_corrected_elist_bins_be** for the new dataset. Take a look at the epoched data (using **EEGLAB > Plot > Channel data (scroll)**) to make sure everything looks okay. As you may recall from Chapter 2, EEGLAB plots 5 epochs on a screen by default, making the epoched data look a lot like continuous data—you need to look closely to see the boundaries between the epochs.

Once you've looked at the EEG epochs, select **EEGLAB > ERPLAB > Compute averaged ERPs**. You should be able to use the default settings and just click **RUN**. Name the resulting ERPset **12_P3**, and save the ERPset as a file named **12_P3.erp**. Now plot the ERPs (using **EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms)**. It's a good idea to click the **RESET** button in the plotting GUI to get rid of any custom settings from the last time you plotted ERPs. We only have two bins, so you can keep it set to plot **all bins**. You should see something like Screenshot 6.2.

Screenshot 6.2



Compare the waveforms at Pz with the grand average waveforms shown in Figure 6.2. Pretty similar! But that's only because I intentionally chose a participant whose ERPs looked like the grand average for the exercises in this chapter. There are tremendous individual differences in ERP waveforms (largely due to nonfunctional differences in biophysical factors like skull thickness), and the ERPs for most of the participants in this study don't look this much like the grand average. Averaging (whether across trials or across participants) is often necessary, but the result is something of a fiction. As I like to say, "averaging hides a multitude of sins."

Although published papers often focus on just a few electrode sites, you should always look at the whole set of channels to verify that the scalp distribution is sensible. With a reference at or near the mastoids, the P3b effect (defined as the difference between Rare and Frequent) should be biggest along the midline near Cz and Pz. This participant's P3b is a little more frontal than is typical for a visual paradigm, and I would ordinarily expect a smaller effect at Fz and a larger effect at Oz. But given the large range of individual differences in ERPs, this participant's data look pretty normal.

You can also see quite a bit of high-frequency noise in the lateral frontal and central channels over the right hemisphere (especially F8). The noise in F8 and surrounding channels is probably EMG from the temporalis muscle (the muscle near the temples that is used to contract the jaws). If you scroll through the original EEG data, you can also see the high-frequency noise in the single trials. My guess is that this participant was clenching their jaw just a little bit, mainly on the right side.

This page titled 6.6: Exercise - Looking at the Averaged ERPs is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.7: Exercise - The Signal-to-Noise Ratio

In this exercise, we're going to quantify the impact of the high-frequency noise in F8 on our data quality and also see how the data quality depends on the number of trials being averaged together.

Take a look at the analytic standardized measurement error (aSME) values for the ERPset you just created (**EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table**). There's a ton of information in the table, and I find that it helps to select the **Color heatmap** option. Interestingly, the **aSME** values from the F8 electrode are not particularly high. The worst values are at PO4.

This disconnect between the **aSME** values and the high-frequency noise you can see in the waveforms occurs because these **aSME** values tell you about the precision of the measurements of the mean voltage over 100-ms periods. High-frequency noise has very little impact on the mean voltage over a 100-ms period, because the upward and downward noise deflections cancel out. However, low-frequency noise has a big impact on the mean voltage over a given time period. If you look closely at the EEG for this participant, you'll see that the voltage tends to drift around a bit more at PO4 than at the other sites. That's why PO4 has the worst (largest) aSME value. If we were to quantify P3b amplitude as the mean voltage between 300 and 500 ms (as recommended by Kappenman et al., 2021), the noise in PO4 would mean that our P3b amplitude score could be quite far from the participant's true score (i.e., the score we would obtain with an infinite number of trials). By contrast, the high-frequency noise in F8 wouldn't have much impact.

If we instead quantified P3b amplitude as the *peak* voltage between 300 and 500 ms, the high-frequency noise would be a bigger problem. Computing the standardized measurement error for peak amplitude is more complicated, so we're not going to look at it now, but if we did I'm sure that the high-frequency noise in F8 would produce a large SME value. The take-home message is that the effect of noise on your ability to precisely measure the amplitude or latency of a given ERP component depends on both the nature of the noise (e.g., high-frequency versus low-frequency) and the nature of the method used to quantify the amplitude or latency (e.g., mean amplitude versus peak amplitude).

Now let's look at how the data quality differs between the Rare and Frequent averages. A standard idea in the ERP literature is that the signal-to-noise ratio of an averaged ERP increases according to the square root of the number of trials (all else being equal). I have to admit that I didn't understand exactly what was meant by "noise" in the signal-to-noise ratio until a few years ago, when we started developing the SME metric of data quality. The "signal" part of the signal-to-noise ratio is the "true" amplitude of the averaged ERP waveform at a given moment in time (i.e., the amplitude we would obtain with an infinite number of trials). But how do we define the noise?

It turns out that the noise is quantified as the *standard error* of the voltage at this time point in the averaged ERP waveform. The voltage at a given time point in an averaged ERP waveform is simply the mean across the epochs being averaged together, and the standard error of this mean can be estimated using the standard analytic formula for the standard error of the mean: SD ÷ sqrt(N). That is, we take the standard deviation (SD) of the single-trial voltages at this time point and divide by the square root of the number of trials. Because the denominator is sqrt(N), this standard error gets smaller according to sqrt(N). So, the denominator of the signal-to-noise ratio decreases according to the square root of the number of trials, so the overall signal-to-noise ratio must increase according to the square root of the number of trials.

In our P3b oddball experiment, 20% of the trials were oddballs, so there were 4 times as many Frequent trials as Rare trials. This means that sqrt(N) was twice as great for the Frequent condition as for the Rare condition (because sqrt(4) = 2). And this implies that the standard error should be half as large for the Frequent condition as for the Rare condition. The SME value is a generalized metric of the standard error; it gives you the standard error for any amplitude or latency measure that is obtained from an averaged ERP waveform (see the box below for more details).

Take a look at the aSME values for the Rare and Frequent conditions. You should see that the values are approximately half as large for the Frequent condition (Bin 2) as for the Rare condition (Bin 1). That is, the noise (quantified as the SME) is about half as big in the condition with four times as many trials. This is exactly what we would expect from the idea that the signal-to-noise ratio varies according to the square root of the number of trials.

The SME is just an estimate, so we wouldn't expect it to be perfectly predicted by the number of trials in a finite data set. To get a more robust estimate, I made an average across all the time ranges and channels, and I found a mean of 0.617 for the Frequent condition and 1.372 for the Rare condition. This isn't quite a 1:2 ratio. But there's a good explanation for this: although

there were 4 times as many Frequent trials as Rare trials in the experiment, we ended up excluding more Rare trials from the averages because of incorrect responses. Take a look at the actual number of trials in Bins 1 and 2—does the difference between the average aSME values for the Rare and Frequent trials make sense given the actual sqrt(N) for the Rare and Frequent trials?

Some details about the SME

With ERPLAB's default settings, the SME values indicate the standard error of the amplitude scores that you'd get by quantifying the amplitude as the mean value in a set of consecutive 100-ms time periods. You can easily change the parameters (in the averaging step) to select other time intervals. Ordinarily, we would measure the P3b as the mean amplitude between 300 and 500 ms. If you're interested, you can have ERPLAB estimate the SME values for this time range by re-averaging the data and selecting **custom parameters** in the **Data Quality Quantification** section of the averaging routine.

ERPLAB's default SME values are estimated using the *analytic* formula for the standard error of the mean [SD ÷ sqrt(N)], so we call these *analytic SME* (aSME) values. This formula can't be used to estimate the SME for other amplitude or latency scores (e.g., peak amplitude or peak latency), and something called *bootstrapping* is used instead. This is more complicated and currently requires scripting. This is described in our original paper on the SME (Luck et al., 2021), and we have provided example scripts for computing bootstrapped SME values (https://doi.org/10.18115/D58G91).

This page titled 6.7: Exercise - The Signal-to-Noise Ratio is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.8: Exercise - Response-Locked Averaging

In this exercise, we're going to create response-locked averages rather than stimulus-locked averages. That is, when we create our EEG epochs, the event code for the response will be time zero.

An important feature of BINLISTER is that it does not make a distinction between stimulus events and response events. An event code is an event code, no matter whether it represents a stimulus, a response, an eye movement, a heartbeat, or a sudden change in the FTSE stock index. Any event code can be the time-locking event (time zero). And a bin can be defined by any sequence of event codes.

Open the bin descriptor file named **BDF_P3_Response_Locked.txt** in the Matlab text editor (by double-clicking it in the Current Folder pane). Here's what you should see:

bin 1
Rare, Correct
{t<200-1000>11;22;33;44;55}.{201}

bin 2
Frequent, Correct
{t<200-1000>12;13;14;15;21;23;24;25;31;32;34;35;41;42;43;45;51;52;53;54}.{201}

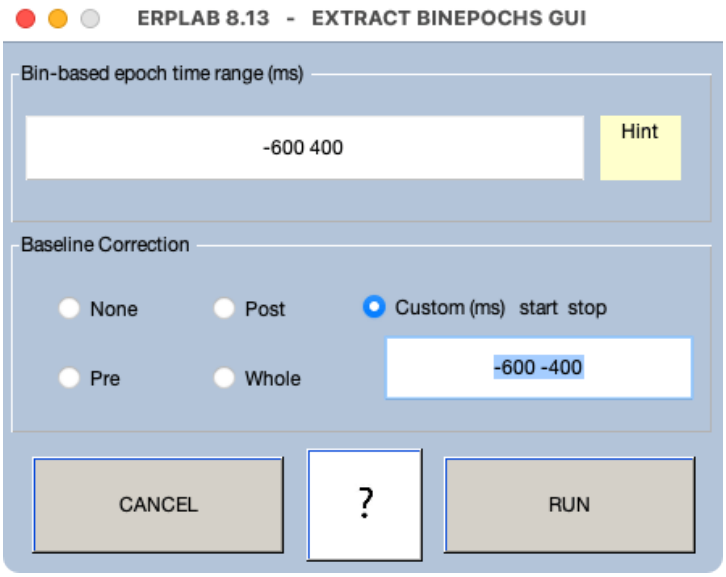
It's a lot like the original bin descriptor file (**BDF_P3.txt**). The most important change is that the period symbol in each bin descriptor is to the left of the response event list rather than being to the left of the stimulus event list. The period goes to the left of the time-locking event, so this is how we indicate that the response event code (201) will be the time-locking event. The **t<200-1000>** part has also been moved from the response event list to the stimulus event list. The timing is always given relative to the time-locking event, which is now the response, so we need to specify that the stimuli must be 200-1000 ms prior to the response rather that the response being 200-1000 ms after the stimulus.

If EEGLAB is running, quit it and restart it so that everything is fresh, and then load the original dataset again (**12_P3_corrected_elist.set**). Then run BINLISTER, using **BDF_P3_Response_Locked.txt** instead of **BDF_P3.txt** as the bin descriptor file. You can use whatever dataset names are convenient, but make sure to save the EventList as a new text file when you run BINLISTER so that you can compare it to the original version.

Take a look at the new EventList text file. If you look at the **bin** column on the far right, you'll now see bin numbers on the lines for the response event codes (the 201 codes), with a 1 if the response is preceded by a target event code (11, 22, 33, 44, or 55) and a 2 if the response is preceded by a nontarget event code. This shows us that the responses will be used as time zero when we epoch and then average the data.

To see this, go ahead and epoch the data (**EEGLAB > ERPLAB > Extract bin-based epochs**) with the settings shown in Screenshot 6.3. Rather than using the default epoch of -200 to 800 ms, we're now using an epoch of -600 to 400 ms. This is because much of the activity occurs before rather than after time zero in a response-locked average. Also, rather than using **Pre** as the baseline correction interval (which would use the period from -600 to 0 ms), select **Custom** and put **-600 -400** into the text box as the **start** and **stop** times. This period should be prior to stimulus onset on most trials because most of the RTs are <400 ms. (There is a way to "trick" ERPLAB into using the actual prestimulus interval as the baseline in a response-locked average, but it's too complicated for this exercise.)

Screenshot 6.3



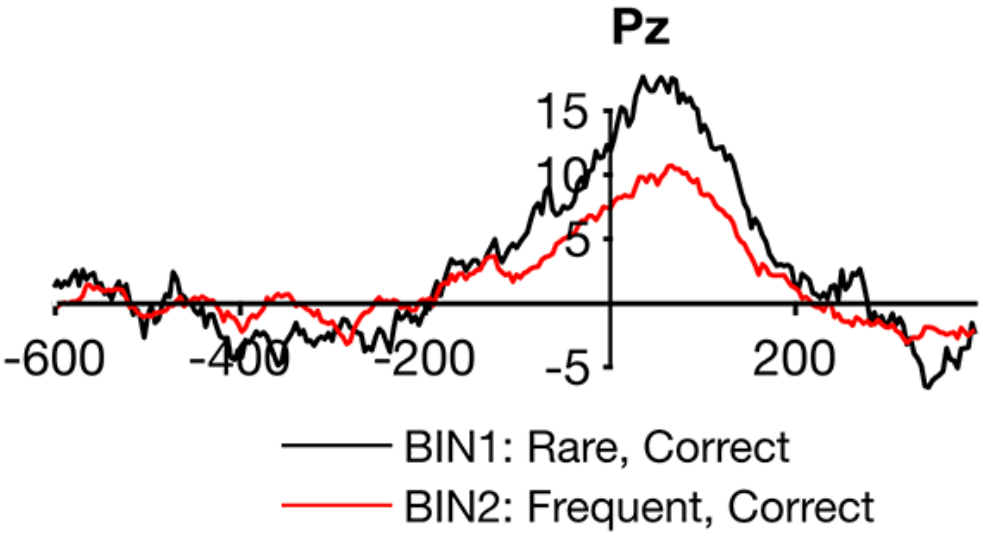
Take a look at the epoched data (using **EEGLAB > Plot > Channel data (scroll)**). If you look at the first epoch, you'll see an event code labeled **B2(201)** at time zero. This means that the event code was 201, and it's now the time-locking event for Bin 2. You'll also see that the epoch begins 600 ms before this event and ends 400 ms after the event. You can't see the stimulus event code prior to this response, because it was more than 600 ms prior to the response and therefore falls outside the epoch. (You can verify this by looking at the **diff** column in the EventList text file.) However, you can see the stimulus event code prior to the response in the second epoch. The stimulus event code isn't at time zero and it doesn't begin with **B2** because it wasn't the time-locking event. Scroll through several pages of epochs to make sure you understand what's in this file.

Now average the data, just as you did for the stimulus-locked data, but use **12_P3_Response_Locked** when you are prompted for the name of the ERPset. Next, plot the ERPs (using **EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms)**. In the region of the plotting GUI labeled **Baseline Correction**, select **None**. Otherwise the plotting routine will re-baseline the data using the interval from -600 to 0 as the baseline. Click **PLOT** to see the data.

Screenshot 6.4 shows a closeup of the waveforms from the Pz channel. You can see that the brain activity now begins approximately 200 ms before the time-locking event (the response), because the stimulus is now before time zero. In addition, the P3 peaks shortly after time zero. Compare this with the stimulus-locked waveforms from the Pz site in Screenshot 6.2, where the P3 peaked around 350 ms. Given the combination of the stimulus-locked and response-locked waveforms, can you guess the approximate mean RT for this participant?

The P3b component is usually tightly time-locked to the participant's decision about whether the current stimulus belongs to the Rare category or the Frequent category, so P3b latency usually (but not always) varies from trial to trial according to the response time As a result, it can be informative to look at the P3b waveform in both stimulus- and response-locked averages. For an example from the basic science literature, see Luck & Hillyard (1990) or the condensed description of this experiment in Luck (2014; especially Figure 8.8 and the surrounding text). For an example with a clinical population, see Luck et al. (2009).

Screenshot 6.4



This page titled 6.8: Exercise - Response-Locked Averaging is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.9: Exercise - Comparing Correct and Error Trials

In this exercise, we're going to create bins for the error trials as well as for the correct trials. This will allow us to look at the *error-related negativity* (ERN, also called *Ne*), a negative-going ERP response over frontocentral electrode sites that is produced when the participant makes an obvious error (see the excellent review by Gehring et al., 2012). Errors are pretty common on the Rare trials in the oddball paradigm because the response for the Frequent category becomes highly primed. When you're doing the task, you find yourself pressing the Frequent button even though you realize (a moment too late) that you should be pressing the Rare button.

Although the first published report of the ERN used an oddball task (Falkenstein et al., 1990), this task is less than ideal for looking at errors because error trials are a small subset of an already-rare stimulus category, leading to very few error trials. In the ERP CORE, we instead used a *flankers* task to look at the ERN, and we didn't even analyze the error trials in the oddball task. In this exercise, we're going to analyze the error trials in the oddball task and look for the ERN. Do you think we'll see an ERN? I wasn't sure we'd see it until I analyzed the error trials for the first time yesterday!

My first step was to create a new bin descriptor file that includes bins for the error trials. The file is named **BDF_P3_Accuracy.txt**—go ahead and open it in the Matlab text editor (by double-clicking it in the Current Folder pane). You'll see that I just added bins for the Rare and Frequent stimuli followed by an error response (event code 202) instead of being followed by a correct response (event code 201).

If EEGLAB is running, quit it and restart it so that everything is fresh, and then load the original dataset again (**12_P3_corrected_elist.set**). Then run BINLISTER, using **BDF_P3_ Accuracy.txt** as the bin descriptor file. You can use whatever dataset names are convenient, but make sure to save the EventList as a new text file.

Take a look at the new EventList text file. Near the top, under the header information, you'll see the number of trials in each bin. As before, there were 30 Rare trials followed by a correct response and 153 Frequent trials followed by a correct response in Bins 1 and 2. However, there were only 9 Rare trials followed by an incorrect response and a measly 3 Frequent trials followed by an incorrect response. And this is actually more error trials than was typical (probably because this participant's response times were faster than those of most participants in this experiment). In the ERP CORE flankers experiment, which was designed to look at the ERN, we controlled the number of errors by telling participants to speed up if they made errors on fewer than 10% of trials and telling them to slow down if they made errors on more than 20% of trials. But the oddball experiment was not designed to look at the ERN, so we didn't try to control the number of errors. Some participants made a lot, and some made hardly any.

As usual, the next step is to epoch the data (**EEGLAB > ERPLAB > Extract bin-based epochs**). The epoching routine may still have the settings from the response-locked averaging we did in the previous exercise, but now we're going to make stimulus-locked averages, so make sure that the epoch time range is set to **-200 800** and the baseline correction period is set to **Pre**. Click **RUN** and name the resulting dataset whatever you want. Now average the data. You should name the resulting ERPset **12_P3_Accuracy** and save it as a file named **12_P3_Accuracy.erp**. You'll need it for a later exercise.

The averaging routine prints the best, worst, and median aSME values to the command window. You should always look at these values to make sure there isn't a problem with the data quality. You'll see that the maximum value is much larger than the median, which tells you there might be a problematic bin or channel. Of course, we might expect low data quality for the error trials given how few trials were present. Consistent with this, the maximum value was for the bin with Frequent stimuli followed by an incorrect response, which had only 3 trials.

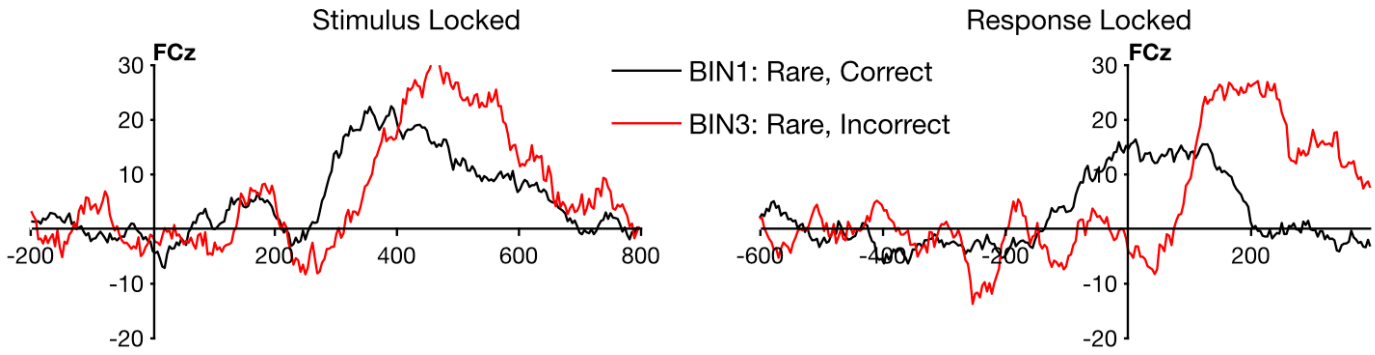
Take a look at all the aSME values using **EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table**. For the correct trials (Bins 1 and 2), the values look pretty reasonable. The SME quantifies the standard error of measurement, and a standard error of between 1 and 3 μV is reasonably small relative to the >15 μV amplitude of the P3b. Now look at the values for Rare stimuli followed by an incorrect response (Bin 3). Most of the values are still less than 5 μV, which seems reasonable given the small number of trials. If you look at the values for Bin 4, however, you'll see a lot of values that quite a bit larger. This makes sense given that we had only 3 trials in this bin.

How many trials do you need? Some researchers have tried to provide a simple answer to this question, but there is no simple answer because it depends on the number of participants (Baker et al., 2020) and the magnitude of the effect being studied (see Boudewyn et al., 2018).

Now let's plot the ERPs (using **EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms**; you may need to click the **RESET** button to clear out the plotting parameters we used in the previous exercise). We'll start by plotting the correct and incorrect Rare trials, so specify Bins 1 and 3. Now we can finally answer the question of whether errors produce an ERN in this experiment (at least for this one participant). The left side of Screenshot 6.5 shows what you should see in the FCz channel, where the ERN is usually largest. You can see that the voltage is more negative from ~200-400 ms on the error trials than on the correct trials. This is the ERN! Replication is a cornerstone of science, and it sure is nice to see that this effect can be replicated.

You can see that voltage is more positive for error trials than for correct trials from ~400-600 ms. This effect has also been seen in many prior studies, and it's called the *error positivity* or *Pe*.

Screenshot 6.5



As always, you'll want to look at all the channels to see the scalp distribution of the effect. You can see that the difference between error trials and correct trials (both in the ERN and Pe time windows) is biggest at frontal and central midline electrode sites. This is what is typically observed when the mastoids or nearby sites are used as the reference. If you're interested, you can try re-referencing to the average of all sites and see how this changes the scalp distribution.

You should also try plotting the correct and incorrect Frequent waveforms (Bins 2 and 4). However, it's hard to see much because there were only 3 incorrect Frequent trials, so the waveforms are extremely noisy.

The ERN usually occurs right around the time of the response. When RTs vary greatly from trial to trial, the ERN in the stimulus-locked averages occurs at different times on different trials, and this makes the ERN look "smeared out" in stimulus-locked averages. The waveforms also look as if the P3b is just shifted to the right on the error trials. In fact, when Bill Gehring first saw the ERN, he was looking at stimulus-locked averages, and he wasn't sure it was a real component. However, when he made response-locked averages, the ERN was a big, beautiful deflection at time zero. You can read the story of how he discovered the ERN in Chapter 3 of Luck (2014).

The participant we've been looking at in this chapter didn't have a lot of RT variability, so the ERN is easily visible in the stimulus-locked averages. But let's make response-locked averages to see if this makes the ERN even clearer. To do this, follow the instructions in the previous exercise for making a response-locked average (especially using an epoch of -600 to 400 ms), but use **BDF_P3_Accuracy_Response_Locked.txt** as the bin descriptor file.

For the FCz channel, the response-locked waveforms should look like those on the right side of Screenshot 6.5. Now the ERN appears as a relatively sharp negative deflection, peaking shortly after time zero (the time of the buttonpress). You can see that it's a distinct deflection rather than simply being a rightward shift in the P3b.

This page titled 6.9: Exercise - Comparing Correct and Error Trials is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.10: Exercise - Sequential Analysis of the P3b

A classic finding in the ERP literature is that the P3b component elicited by an oddball is smaller if the previous trial was also an oddball than if the previous trial was a standard (Squires et al., 1976). In this exercise, we will perform a sequential analysis to see this effect using the ERP CORE P3b paradigm. This will bring up some important general issues about ERPs as well as showing you more about the process of assigning events to bins.

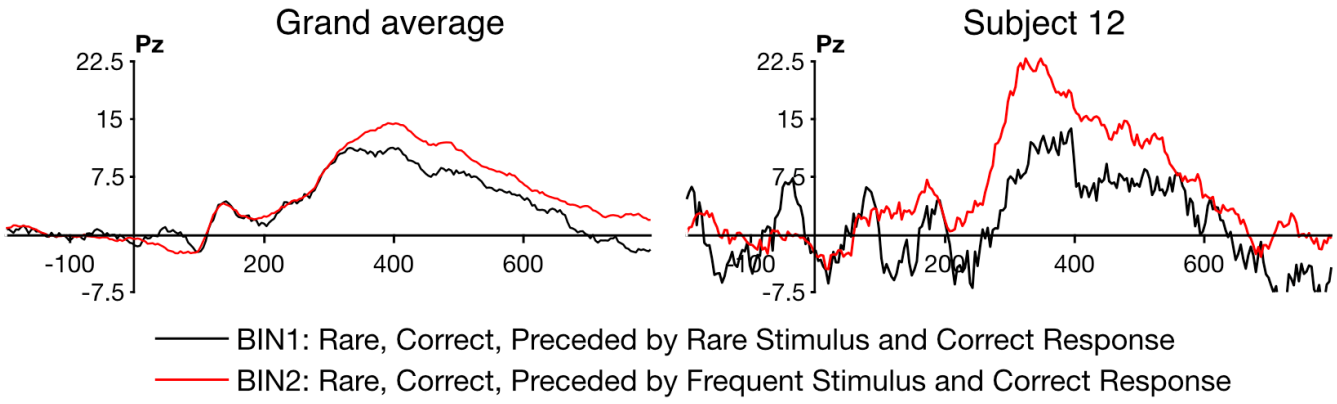
I've provided a bin descriptor file for this analysis named **BDF_P3_Sequential.txt**—go ahead and open it in the Matlab text editor (by double-clicking it in the Current Folder pane). You'll see that we have 4 bins: Bin 1, Rare preceded by Rare; Bin 2, Rare preceded by Frequent; Bin 3, Frequent preceded by Rare; Bin 4, Frequent preceded by Frequent. The bin descriptors were modified to require that the time-locking stimulus was preceded by either the Rare or Frequent stimulus and a correct response. For example, Bin 1 is defined as:

{11;22;33;44;55}{201}.{11;22;33;44;55}{t<200-1000>201}

Okay, let's make some ERPs with these bins. Quit and restart EEGLAB so that everything is fresh, and then load the original dataset again (**12_P3_corrected.set**). Now run BINLISTER, using **BDF_P3_ Sequential.txt** as the bin descriptor file. We'll need the resulting dataset for a later exercise, so name it **12_P3_corrected_elist_bins_seq**, and save it as a file named **12_P3_corrected_elist_bins_seq.set**.

Take a look at the new EventList text file to see how many trials we have in each bin. There were only 6 Rare trials that were preceded by Rare trials (with correct responses on both trials). That's not a lot! This was a pretty short experiment (about 10 minutes), and we would ordinarily use a longer session with a lot more trials to do a sequential analysis. In fact, when I was developing this exercise, the first subject I tried didn't have a clear sequential effect; there was a hint of an effect, but the data were so noisy that it wasn't very clear. I then wrote a script to do the analysis for all the participants (which is provided in the Chapter_6 folder). Fortunately, when I looked at the grand average, I saw the nice effect shown on the left of Screenshot 6.6, in which the P3b for the Rare stimulus was clearly larger when the preceding trial was the Frequent stimulus than when it was the Rare stimulus. I then looked for a participant who exhibited this effect clearly, and I used this participant (Subject 12) for all the exercises in this chapter.

Screenshot 6.6



The next step is to epoch the data (**EEGLAB > ERPLAB > Extract bin-based epochs**). Make sure that the epoch time range is set to **-200 800** and the baseline correction period is set to **Pre**. Click **RUN** and name the resulting dataset whatever you want. Now average the data. You should name the resulting ERPset **12_P3_Sequential** and save it as a file named **12_P3_Sequential.erp**. Now plot the data for the Rare stimuli (Bins 1 and 2). If you look at the Pz channel, you should see something like the waveforms shown on the right of Screenshot 6.6. Once again, we've successfully replicated a finding from prior research!

However, we need to worry about the data quality given the small number of trials. Take a look at the aSME values for Bins 1 and 2 (**EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table**). You'll see that most of the values are

worse (higher) for Bin 1 than for Bin 2, which is not surprising given that we had 6 trials in Bin 1 and 22 trials in Bin 2. However, if you look at the Pz channel from 300 to 800 ms, you'll see that the aSME values are only slightly higher for Bin 1 than for Bin 2. I think this was just good luck: by chance, this channel didn't show a lot of trial-to-trial variability in P3b amplitude in Bin 1, so the standard error was pretty good despite the small number of trials. This makes the data from the Rare-preceded-by-Rare condition somewhat believable. However, the nice big effect in the grand average is what really makes it believable.

Sequence or Time?

Although many P3b effects have traditionally been interpreted in terms of the sequence of stimuli and sequential probability, many of these effects appear to be primarily a result of the amount of time between stimuli of the same category (Polich, 2012).

This page titled 6.10: Exercise - Sequential Analysis of the P3b is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.11: Exercise - Combining Bins

In the previous exercise, we took our original Rare and Frequent epochs and subdivided them according to whether the previous trial was Rare or Frequent, yielding four bins instead of two. We should be able to reconstruct our original two bins from the four new bins. However, there is a little trick that you need to know about.

Let's start by thinking about how to reconstruct the original Rare bin, in which the preceding trial could be either Rare or Frequent. We could simply take Bin 1 (Rare preceded by Rare) and average it with Bin 2 (Rare preceded by Frequent). In **ERP Bin Operations**, we could implement this with the following equation:

bin5 = (bin1 + bin2) / 2 Label Rare preceded by Rare or Frequent

However, this would not give us the same waveforms that we obtained in our original analysis that disregarded whether the previous trial was Rare or Frequent. Can you figure out why?

The reason is that this equation gives Bins 1 and 2 equal weight in the new bin, but there were 6 trials with Bin 1 and 22 trials with Bin 2 (which you can see by typing **ERP.ntrials** in the Matlab Command Window). The equation above assumes that there were equal numbers of trials in each bin. To give each trial equal weight (which is what happened when the identity of the previous trial was disregarded by BINLISTER), we would need the following equation in **ERP Bin Operations**:

bin5 = (6*bin1 + 22*bin2) / 28 Label Rare preceded by Rare or Frequent

However, the exact equation would depend on how many trials were in Bins 1 and 2, which might vary across participants. To make this easier, **ERP Bin Operations** contains a function called *wavgbin* for performing *weighted averaging*, in which each bin is automatically weighted by the number of trials.

Let's see how these different methods of averaging work. Make sure that the dataset from the previous exercise is loaded (**12_P3_Sequential**). Select **EEGLAB > ERPLAB > ERP Operations, ERP Bin Operations**. In the equations panel, clear out any previous equations and enter the three following equations:

bin5 = (bin1 + bin2) / 2 Label Rare preceded by Rare or Frequent, equal weight

bin6 = (6*bin1 + 22*bin2) / 28 Label Rare preceded by Rare or Frequent, manually weighted

bin7 = wavgbin(bin1, bin2) Label Rare preceded by Rare or Frequent, automatically weighted

Make sure that the **Mode** is set to **Modify existing ERPset**, because we're going to add these three bins to the four existing bins. Then click **RUN**.

Now plot Bins 5, 6, and 7. You can't actually see the waveforms for Bin 6, because it's identical to Bin 7, and the Bin 7 waveform exactly covers up the Bin 6 waveform. This shows that the **wavgbin** function is working properly. However, the P3b is slightly larger for these bins than for Bin 5. Why is that? You should be able to figure it out given what you know about the amplitude of the P3b in Bins 1 and 2 and the nature of the equations used for Bins 5, 6, and 7.

An obvious question is whether you should use unweighted averaging (as in Bin 5) or weighted averaging (as in Bins 6 and 7) when combining bins together. There isn't a single answer to this question. If you're trying to create something equivalent to what you would have gotten if you hadn't subdivided the trials to begin with, then you'll want to use weighted averaging. But beyond that, the answer will depend on the logic of your experimental design and your scientific questions.

This page titled 6.11: Exercise - Combining Bins is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.12: Exercise - Overlap

A concern often arises in a sequential analysis like the one we performed in this chapter, namely that the results are distorted by overlapping activity from the previous trial. A simple example of overlap is shown in Figure 6.3, which simulates an experiment in which a stimulus is presented every 700 ms. The falling edge of the P3b from one trial is present during the prestimulus baseline period of the next trial, which impacts the baseline correction procedure. That is, baseline correction involves taking the mean voltage during the baseline period and subtracting it from each point in the waveform. The positive voltage from the overlap in the baseline causes us to overestimate the true baseline voltage, and the entire waveform is shifted downward as a result. You might think that we should just skip the baseline correction step, but it's essential in 99.9% of experiments because there are huge sources of low-frequency noise that would contaminate the waveform if we did not subtract the baseline.

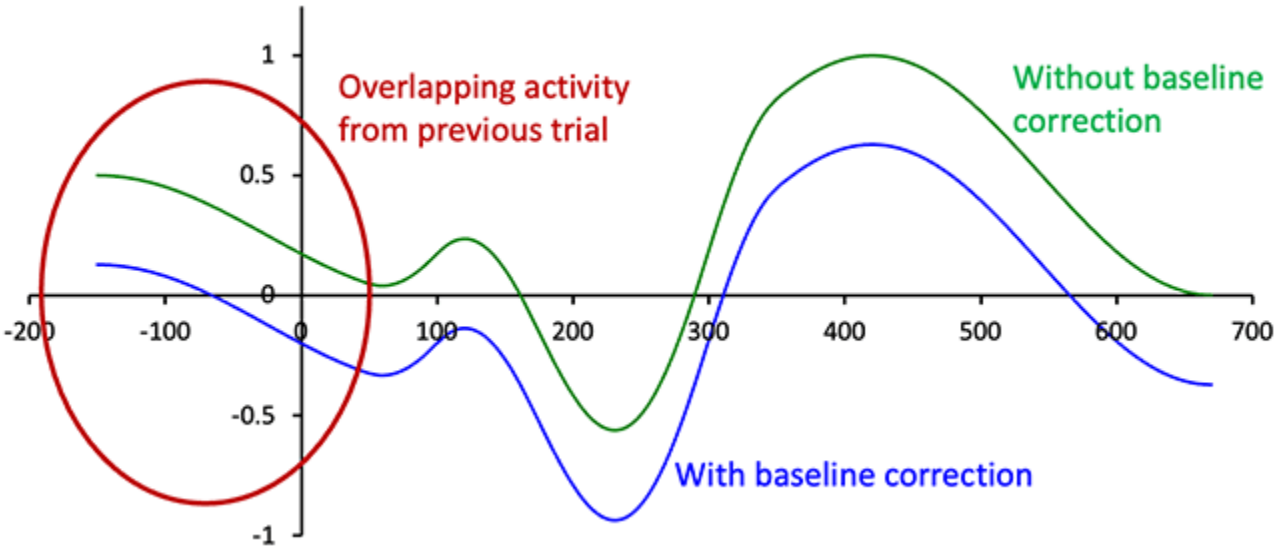


Figure 6.3. Example of the impact of overlap on an ERP waveform before and after baseline correction. In this example, we are assuming that a stimulus is presented every 700 ms, so the falling edge of the P3b from one trial is present during the prestimulus period of the next trial. When this positive voltage is subtracted out by the baseline correction process, it shifts the poststimulus waveform downward.

Some amount of overlap is present in most experiments, but it's not usually a problem unless it differs across conditions. For example, if the P3b on the previous trial was larger in Condition A than in Condition B, this would produce a larger positive voltage in the baseline of Condition A. The baseline correction procedure would then push the rest of the waveform farther downward in Condition A than in Condition B. As a result, the differences in overlap between Conditions A and B during the prestimulus period end up creating a difference between the waveforms for these conditions in the poststimulus period. This is a fundamentally important issue, and I recommend reading Marty Woldorff's foundational paper on overlapping ERP activity (Woldorff, 1993). In fact, it's on my list of papers that every new ERP researcher should read.

I hope you can now see why I'm a little worried about comparing the Rare-preceded-by-Rare waveform with the Rare-preceded-by-Frequent waveform. The overlapping voltage in the baseline might be larger when the previous trial was Rare (yielding a large P3b) than when it was Frequent. The baseline correction procedure would then push the waveform farther downward on Rare-preceded-by-Rare trials than on Rare-preceded-by-Frequent trials, artificially creating the appearance of a smaller (less positive) P3b in the Rare-preceded-by-Rare waveform than in the Rare-preceded-by-Frequent waveform.

However, if the time between trials is long enough, the P3b from the previous trial will have faded prior to the baseline period of the current trial. In the ERP CORE visual oddball P3b experiment, the time from one stimulus onset to the next (the *stimulus onset asynchrony* or *SOA*) was 1400-1600 ms. The 200-ms prestimulus interval that we've used for the current trial therefore began at least 1200 ms after the onset of the stimulus from the previous trial. That seems like it ought to be enough time for the P3b from the previous trial to end, but it's difficult to be certain without additional analyses.

In this exercise, we're going to repeat the sequential analysis from the previous exercise, but taking a closer look at the overlapping activity from the previous trial. In particular, we're going to dramatically increase the length of the prestimulus portion

of the epoch so that we can see the previous-trial ERP as well as the current-trial ERP. And we're going to see what happens when we use different parts of the prestimulus period for baseline correction.

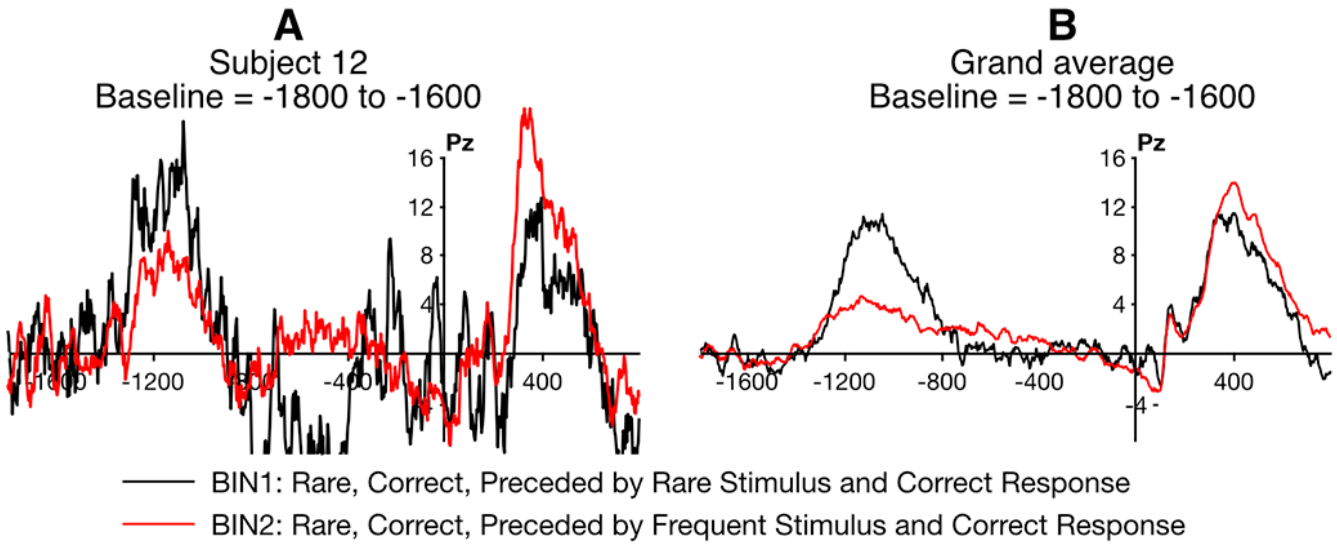
We're simply going to change the epoch length, so we can start with the EEG dataset you created using BINLISTER in the previous exercise, named **12_P3_corrected_elist_bins_seq**. Make it the active dataset (which may require loading it from the file you saved in the previous exercise). Now epoch the data (**EEGLAB > ERPLAB > Extract bin-based epochs**) using an epoch time range of **-1800 800** and a baseline correction period of **-1800 -1600**. If you look at Figure 6.2, you'll see that the maximum SOA is 1600 ms, so the period from -1800 to -1600 ms is always prior to the previous trial. By using this as our baseline, we can be certain that the baseline correction won't be influenced by whether the previous trial was the Rare or Frequent category.

Now average the data, naming the ERPset **12_Sequential_LongBaseline** and saving it as a file named **12_Sequential_LongBaseline.erp**. Plot the ERPs from Bins 1 and 2, but specify **None** in the Baseline Correction area of the plotting GUI so that it doesn't re-baseline the data. (Alternatively, you could specify **Custom** with a time range of **-1800 -1600**).

Screenshot 6.7A shows what it should look like in the Pz channel. You can still see that the current-trial P3b is larger for Rare-preceded-by-Frequent than for Rare-preceded-by-Rare. That suggests that the sequential effect was saw before was not a result of the combination of overlap and baseline correction. However, the data are pretty noisy, so I ran this analysis on all the participants. The grand average in Screenshot 6.7B confirms that the sequential effect remains when the data are baselined relative to the prestimulus period from the previous trial.

The grand average waveforms also make it clear that differential overlap is a real concern. That is, the persisting voltages from the previous trial were somewhat different according to whether that trial was Rare or Frequent, potentially contaminating the baseline period from the current trial. However, the overlapping activity seems to go in the opposite direction of the sequential effect, being more negative (near time zero) when the previous trial was Frequent than when it was Rare, whereas the current-trial P3b was more positive.

Screenshot 6.7



Although the grand averages looked fine when we used the prestimulus period from the prior trial as the baseline interval, this approach can decrease the precision and reliability of the ERP waveforms. As illustrated in Figure 6.4, this decline in precision occurs because the single-trial EEG voltage tends to drift gradually over time in a random direction, and the trial-to-trial variability increases as you get farther away from the baseline period. This increased variability increases the error in measuring the "true" amplitude, increasing variability across participants and thereby decreasing statistical power. Indeed, the waveforms in some of the channels look pretty crazy with this distant baseline.

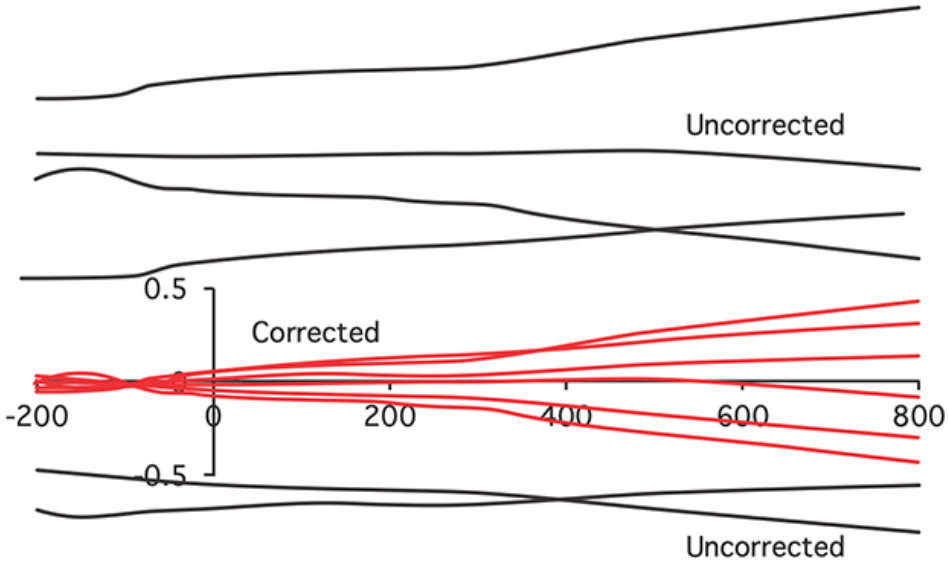


Figure 6.4. Illustration of the combined effects of low-frequency drift and baseline correction. The EEG is superimposed on low-frequency signals (mainly arising from the skin) that drift slowly over time in a random direction. When the data are baseline-corrected, this forces the voltages to be similar during the baseline period, but the voltage tends to drift farther away from the baseline value, causing a gradual increase in trial-to-trial variability over time.

You can use the Standardized Measurement Error to quantify this increase in measurement error. Take a look at the aSME values for the Pz channel in Bin 1 (**EEGLAB > ERPLAB > Data Quality options > Show Data Quality measures in a table**). We'll want to compare these values to the corresponding values from the original sequential analysis in which the baseline correction period was -200 to 0 ms. (To save the current aSME values, you can either copy them from the table and paste them in a word processor or save the values in a spreadsheet by clicking the button for **Export these values, writing to: Excel**.) Load the ERPset you saved previously as **12_P3_Sequential.erp** and look at the aSME values in that dataset, which was baselined from -200 to 0.

How do the aSME values differ between the data with the two different baseline periods? You should see that they are substantially worse (larger) when the -1800 to -1600 period was used as the baseline. That's exactly what we'd expect from Figure 6.4. So, although using a distant time period for baseline correction can sometimes be valuable for assessing overlap, it comes at the cost of increased measurement error. As a result, I find that analyses like these are mainly useful descriptively and not for statistical analysis. In other words, it's usually enough to look at the grand averages and see that the effects are largely the same with the two different baseline periods without performing a statistical analysis on the data with the distant baseline period.

This page titled 6.12: Exercise - Overlap is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.13: Matlab Script For This Chapter

I've provided a script called **sequential_analysis.m** in the Chapter_6 folder that implements the sequential analysis.

This page titled 6.13: Matlab Script For This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

6.14: Key Takeaways and References

Key Takeaways

- To ensure that ERP effects reflect psychological processes instead of low-level physical stimulus differences, it is important to follow the Hillyard Principle: Keep the stimuli constant and vary only the psychological conditions.
- The impact of noise on data quality depends on both the nature of the noise (e.g., low- versus high-frequency) and the method of quantifying an ERP component (e.g., mean versus peak amplitude).
- The standardized measurement error (SME) quantifies the standard error of an ERP amplitude or latency measurement. In simple cases, the standard error decreases as the square root of the number of trials increases. As a result, the signal-to-noise ratio increases as the square root of the number of trials increases (all else being equal).
- Data quality typically declines when the activity being measured is distant in time from the baseline correction period.
- It can be useful to look at the same data using both stimulus-locked and response-locked averages.
- When averaging bins together, you need to think carefully about whether to weight the two bins equally or to weight by the number of trials per bin.
- Overlapping activity from previous trials can be a significant confound, especially when combined with baseline correction.

References

Baker, D. H., Vilidaite, G., Lygo, F. A., Smith, A. K., Flack, T. R., Gouws, A. D., & Andrews, T. J. (2020). Power contours: Optimising sample size and precision in experimental psychology and human neuroscience. *Psychological Methods*. http://dx.doi.org/10.1037/met0000337

Boudewyn, M. A., Luck, S. J., Farrens, J. L., & Kappenman, E. S. (2018). How Many Trials Does It Take to Get a Significant ERP Effect? It Depends. *Psychophysiology*, *55*, e13049. https://doi.org/10.1111/psyp.13049

Duncan-Johnson, C. C., & Donchin, E. (1977). On quantifying surprise: The variation of event-related potentials with subjective probability. *Psychophysiology*, *14*, 456–467.

Falkenstein, M., Hohnsbein, J., Joormann, J., & Blanke, L. (1990). Effects of errors in choice reaction tasks on the ERP under focused and divided attention. In C. H. M. Brunia, A. W. K. Gaillard, & A. Kok (Eds.), *Psychophysiological Brain Research* (pp. 192–195). Elsevier.

Gehring, W. J., Liu, Y., Orr, J. M., & Carp, J. (2012). The error-related negativity (ERN/Ne). In S. J. Luck & E. S. Kappenman (Eds.), *The Oxford Handbook of Event-Related Potential Components* (pp. 231–292). Oxford University Press.

Kappenman, E. S., Farrens, J. L., Zhang, W., Stewart, A. X., & Luck, S. J. (2021). ERP CORE: An Open Resource for Human Event-Related Potential Research. *NeuroImage*, *225*, 117465. https://doi.org/10.1016/j.neuroimage.2020.117465

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Luck, S. J., & Hillyard, S. A. (1990). Electrophysiological evidence for parallel and serial processing during visual search. *Perception & Psychophysics*, *48*, 603–617.

Luck, S. J., Kappenman, E. S., Fuller, R. L., Robinson, B., Summerfelt, A., & Gold, J. M. (2009). Impaired response selection in schizophrenia: Evidence from the P3 wave and the lateralized readiness potential. *Psychophysiology*, *46*, 776–786.

Luck, S. J., Stewart, A. X., Simmons, A. M., & Rhemtulla, M. (2021). Standardized Measurement Error: A Universal Measure of Data Quality for Averaged Event-Related Potentials. *Psychophysiology*. https://doi.org/10.1111/psyp.13793

Luck, S. J., & Vogel, E. K. (2013). Visual Working Memory Capacity: From Psychophysics and Neurobiology to Individual Differences. *Trends in Cognitive Sciences*, *17*, 391–400.

Luria, R., Balaban, H., Awh, E., & Vogel, E. K. (2016). The contralateral delay activity as a neural measure of visual working memory. *Neuroscience & Biobehavioral Reviews*, *62*, 100–108. https://doi.org/10.1016/j.neubiorev.2016.01.003

Polich, J. (2012). Neuropsychology of P300. In S. J. Luck & E. S. Kappenman (Eds.), *The Oxford Handbook of Event-Related Potential Components* (pp. 159–188). Oxford University Press.

Squires, K., Wickens, C., Squires, N. K., & E. Donchin. (1976). The effect of stimulus sequence on the waveform of the cortical event-related potential. *Science*, *193*, 1142–1146. https://doi.org/10.1126/science.959831

Vogel, E. K., Luck, S. J., & Shapiro, K. L. (1998). Electrophysiological evidence for a postperceptual locus of suppression during the attentional blink. *Journal of Experimental Psychology: Human Perception and Performance*, *24*, 1656–1674.

Woldorff, M. (1993). Distortion of ERP averages due to overlap from temporally adjacent ERPs: Analysis and correction. *Psychophysiology*, *30*, 98–119.

This page titled 6.14: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

7: INSPECTING THE EEG AND INTERPOLATING BAD CHANNELS

Learning Objectives

In this chapter, you will learn to:

- Inspect the raw data to determine what problems exist that might require intervention
- Determine whether an EEG channel is “bad” and replace the bad channel with interpolated values
- Identify “by eye” the most common types of artifacts, including blinks, saccadic eye movements, and muscle noise

I often say that it’s a miracle meaningful brain activity can be recorded from electrodes on the skin overlying the skull. Although the EEG is miraculous, it's also an imperfect measure of brain activity. There are many types of artifacts that can contaminate the data, and some of the electrodes may have poor electrical connections to the scalp, creating *bad channels*.

Several preprocessing steps are necessary prior to averaging so that artifacts and bad channels don’t lead you to draw incorrect conclusions from the ERPs. In particular, we can replace the data from bad channels by interpolation from the good channels, and we can apply artifact rejection and correction to minimize the impact of artifacts.

Prior to performing these steps, it is **extremely important** that you first perform a visual inspection of a given participant’s EEG so that you understand what kinds of problems are present in that dataset. You also need to spend a lot of time looking at raw EEG data when you are learning to perform preprocessing so that you learn how to identify the most common types of problems.

The goal of this chapter is to teach you how visually inspect EEG data and how to identify bad channels and common artifacts. You’ll also learn how to replace bad channels with interpolated data. The subsequent chapters will cover artifact rejection and artifact correction. We’ll look at data from one participant in the mismatch negativity (MMN) experiment from the ERP CORE (Kappenman et al., 2021).

7.1: Data for This Chapter

7.2: Design of the Mismatch Negativity (MMN) Experiment

7.3: Video Demonstration- Performing an Initial Inspection of a Participant’s EEG

7.4: The Fundamental Goal of EEG Preprocessing

7.5: Background- Interpolating Bad Channels

7.6: Exercise - Interpolating Bad Channels

7.7: Matlab Script For This Chapter

7.8: Key Takeaways and References

This page titled 7: Inspecting the EEG and Interpolating Bad Channels is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

7.1: Data for This Chapter

The data we'll be using for the exercises in this chapter can be found in the Chapter_7 folder in the master folder: https://doi.org/10.18115/D50056.

This page titled 7.1: Data for This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

7.2: Design of the Mismatch Negativity (MMN) Experiment

The exercises in this chapter and the chapters on artifact rejection and correction will use data from the ERP CORE MMN experiment, and this section will provide a brief overview of the experimental design and main results.

As illustrated in Figure 7.1.A, the experiment involved a sequence of frequently occurring *standard* tones (1000 Hz, 80 dB, p = .8) and rare *deviant* tones that were slightly softer (1000 Hz, 70 dB, p = .2). Each tone was presented for 100 ms, and successive tones were separated by a silent interstimulus interval of 450-550 ms. The MMN does not require active attention, so the tones were task-irrelevant. Instead, the participants were instructed to simply watch a small silent movie of a sand drawing in the middle of the computer monitor (Figure 7.1.B). This kept the participants awake and alert, minimizing alpha-band EEG oscillations. The movie was quite small to avoid large eye movements. Additional methodological details can be found in Kappenman et al. (2021).

The grand average ERPs for the standards and the deviants are shown in Figure 7.1.C. The waveforms are shown for the FCz channel, where the MMN is typically largest. The MMN can be seen as a more negative voltage for the deviants than for the standards from approximately 125–225 ms. The MMN is typically isolated from the other overlapping ERP components by means of a deviant-minus-standard difference wave, which is shown in Figure 7.1.D.

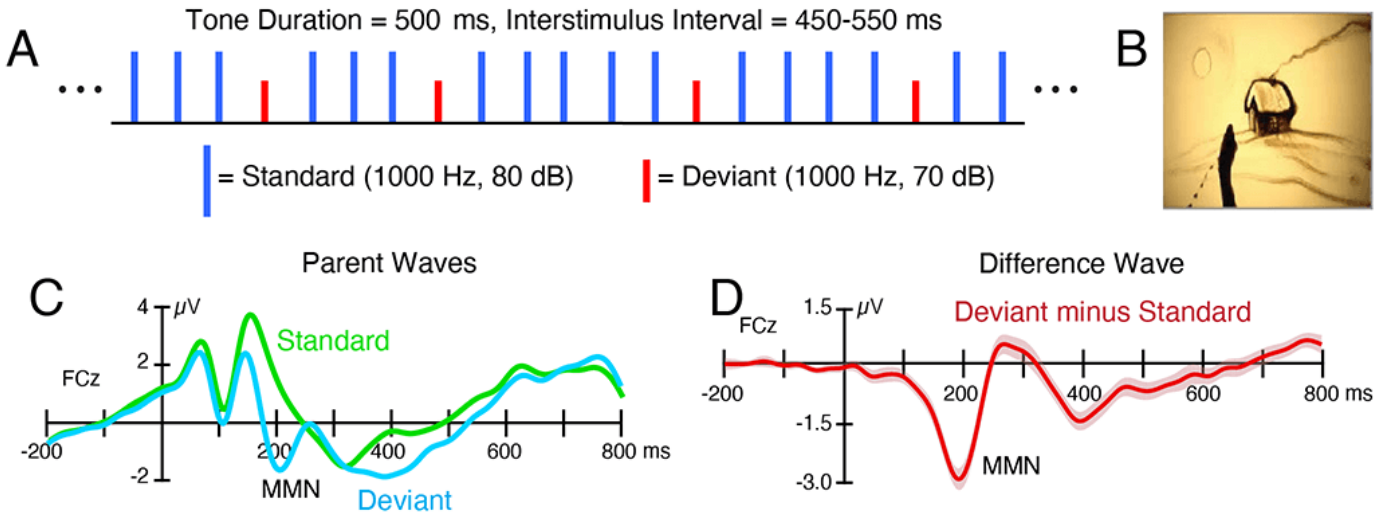


Figure 7.1. Experimental paradigm and results from the ERP CORE Mismatch Negativity experiment. (A) Example sequence of standard tones (p=.8) and deviant tones (p=.2). The tones were task-irrelevant. (B) One frame of the silent movie that participants watched while the tones were playing. (C) Grand average ERP waveforms elicited by the standard and deviant tones, averaged over all 40 participants. (D) Grand average difference wave, which was created by subtracting the averaged ERP waveform for the deviant trials from the averaged ERP waveform for the standard trials.

This page titled 7.2: Design of the Mismatch Negativity (MMN) Experiment is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

7.3: Video Demonstration- Performing an Initial Inspection of a Participant's EEG

As I mentioned at the beginning of the chapter, it's important to scan through each participant's EEG to see what kinds of artifacts are in the data. The best way to learn to do this is to go through the data from several participants with an expert who can show you what to look for. I've tried to simulate that experience by creating a video where I go through the data from Subject #1 in the ERP CORE MMN experiment.

Every participant is different, but Subject #1 is a good example because the recording contains many of the problems you'll encounter. In other words, Subject #1 is worse than usual for a basic science experiment in my lab, which is a good thing for seeing examples of problems in EEG data.

When I first started going through Subject #1's data, the EEG seemed super clean (and therefore not a good example). But as you'll see in the video, things go downhill pretty rapidly after the first couple minutes of the recording.

You should follow along on your own computer while you watch the video. To do this, quit and relaunch EEGLAB, set **Chapter_7** to be Matlab's current folder, and the load the dataset named **1_MMN_preprocessed.set**. Then select **EEGLAB > Plot > Channel data (scroll)**.

Go ahead and watch the video now. You can access it at https://doi.org/10.18115/D5V638. It's about 10 minutes long, and you'll definitely need to watch it before you read the rest of this chapter and the chapters on artifact rejection and correction.

This page titled 7.3: Video Demonstration- Performing an Initial Inspection of a Participant's EEG is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

7.4: The Fundamental Goal of EEG Preprocessing

This chapter and the following two chapters describe EEG preprocessing steps that are used to deal with problem in the data, such as bad channels and artifacts. First, we need to think about what our goal is for these steps. I find that most people haven't thought clearly about what they are trying to achieve in their preprocessing pipeline, and this leads to pipelines that are less than ideal.

So, what is the goal of EEG preprocessing? It's simple: **our fundamental goal is to accurately answer the scientific question that the experiment was designed to address**. If we include bad channels or large artifacts, this will likely move us away from this goal. However, if throw out a lot of trials or participants, this will also tend to move us away from our goal by reducing our statistical power. Most preprocessing steps—such as interpolating bad channels or rejecting trials with artifacts—have both costs and benefits, and we need to ask whether the benefits of a given processing step outweigh the costs in terms of answering the scientific question that the experiment was designed to address. We will discuss how to assess these costs and benefits as we go through this chapter and the next two chapters.

This page titled 7.4: The Fundamental Goal of EEG Preprocessing is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

7.5: Background- Interpolating Bad Channels

In the video demonstration, the C5 channel seemed quite problematic, and I mentioned that we might want interpolate that channel. Also, the F8 channel had a lot of high-frequency noise, so we should consider interpolating that channel as well. In this exercise, we'll discuss how to decide whether a channel should be interpolated and then look at the actual interpolation process.

Interpolation is performed separately at each time point, using the voltage values from the other channels at that time point. There are several algorithms that can be used to estimate a reasonable value for one channel on the basis of the other channels, and they all work reasonably well. I like to use the *spherical* algorithm provided by EEGLAB. It treats the electrodes as if they're located on a spherical head, and it fits a polynomial function to the distribution of voltages, leaving out the to-be-interpolated channel. Then, the value of the polynomial function at the to-be-interpolated location is used as the estimated voltage at that location. This process is repeated independently at each time sample. You get a pretty reasonable waveform at the interpolated location, but keep in mind that it's just an imperfect estimate of the true waveform.

The decision about whether to interpolate a given channel ultimately comes down to our fundamental goal of accurately answering the scientific question that the experiment was designed to address. I'll provide some general guidance, but ultimately you need to think about whether interpolation serves that goal. Does it get you closer to the truth or farther from the truth? Imagine that you've submitted a manuscript to a journal, and one of the reviewers visits your lab to see how you actually processed your data (which would never happen in reality, of course). Would you be happy to explain to this reviewer how you decided whether to interpolate? Or would you feel a little embarrassed?

Let's start with an extreme case. Imagine that an electrode was broken and the signal from that electrode was pure noise. And imagine that the broken electrode wasn't being used in any of the key analyses. For example, we measured MMN amplitude at FCz for the main analyses in the ERP CORE paper (Kappenman et al., 2021), so the C5 and F8 electrodes didn't play a major role our analyses (although they did make a minor contribution when we plotted scalp maps). One way to deal with the broken electrode would be to completely discard this participant's data. This would reduce the sample size in our analyses, which tends to decrease our ability to draw accurate conclusions about the population, so that's not a great option. The other main option would be to interpolate the data from the broken electrode. Given that the broken electrode doesn't contribute to the main analyses, being able to include the participant by interpolating and thereby increasing your sample size seems like it serves the truth much more than excluding the participant from all analyses.

Now let's consider an extreme case in the opposite direction. Imagine that you're analyzing data collected in another lab with really poor recording methods (or a really challenging participant), and the data from FCz and the 8 closest electrodes look terrible. Not only is the key channel for the analysis bad, but so are the surrounding channels, which will make it difficult to interpolate accurately. In this case, including the participant in the final analyses seems like it will not add real information and will do more harm than good.

I posted a message on Twitter asking how researchers decide whether any channels should be interpolated. Some people indicated that they do it informally by visual inspection. Bad channels are relatively rare in my lab, so we also use visual inspection. Other researchers indicated that they used an automated method, such as the Clean Rawdata plugin in EEGLAB or the PREP pipeline (Bigdely-Shamlo et al., 2015). Many researchers who said that they use an automated method indicated that the algorithm fails often enough that they visually confirm the results. I've never tried the automated methods, but the algorithms seem reasonable (especially the PREP pipeline), and I would expect them to work well when verified by visual inspection. However, they mainly rely on statistical criteria, such as how well a given channel correlates with other channels (with a low correlation suggesting a problem given that true brain signals spread broadly across the scalp and therefore produce high correlations between nearby electrode sites).

This will certainly work for detecting things like loose electrodes. However, if an electrode is properly connected but the signal contains biological noise (e.g., muscle activity), the question is whether the noise decreases your ability to precisely quantify the amplitude or latency value that will be the dependent variable in your statistical analyses (because this is how we reach our fundamental goal or accurately answering the scientific question that the experiment is designed to address). The Standardized Measurement Error (SME) is ideally suited for this purpose because it quantifies the extent to which the noise in the data produces error in the specific amplitude or latency value you will be obtaining from the averaged ERPs. With this in mind, let's consider the actual cases of the C5 and F8 electrodes from Subject #1 in the ERP CORE MMN experiment.

This page titled 7.5: Background- Interpolating Bad Channels is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

7.6: Exercise - Interpolating Bad Channels

If you don't already have the **1_MMN_preprocessed.set** dataset loaded, you should load it now. This dataset has already been preprocessed, including referencing to the average of P9 and P10 and high-pass filtering with a half-amplitude cutoff at 0.1 Hz. An EventList has been added, and BINLISTER has been run to assign events to bins. We have only two bins: Deviant Preceded by Standard (Bin 1, 200 epochs)) and Standard Preceded by Standard (Bin 2, 585 epochs).

We've already looked carefully at the EEG (as shown in the video), and now we need to make a decision about whether C5 and/or F8 should be interpolated. To help make this decision, let's calculate the SME. First, we need to epoch the continuous EEG, so select **EEGLAB > ERPLAB > Extract bin-based epochs**. Specify an epoch of -200 to 800 ms and **Pre** as the baseline.

Now we're ready to compute the SME. The SME is automatically computed when we create averaged ERPs, but we don't need to look at averages right now, so you should instead get the SME values by selecting **EEGLAB > ERPLAB > Compute data quality metrics (without averaging)**.

We're quantifying MMN amplitude as the mean voltage from 125 to 225 ms, so we want to get the SME values for this time range. To do this, select **Custom parameters** in the averaging GUI and click the **Set DQ options...** button. A new window will appear. Click **Add a row** to add a new time window. You'll need to scroll down to see the new row. Change to the name of this row to **aSME at 125 to 225** and change the time window start and end fields to **125** and **225**. Then click **Save** to get back to the main averaging GUI, and then click **RUN**. A table with the data quality values will then appear. If you look at the aSME values for Bin 1, the rightmost column will show the values for the 125-225 ms time range.

You'll see that the aSME values for Fp1, Fp2, and VEOG-bipolar are all quite high relative to the other channels. That's because of blinking, so you can ignore those channels. (You could perform artifact correction prior to examining the aSME to avoid this issue, especially if you were concerned that one of these channels should be interpolated.) Of the remaining channels, C5 really stands out, with a value that is much higher than the other channels.

The data quality table GUI includes two tools that can help you find problematic values. First, the **Color heatmap** option colors each cell according to the value relative to the other cells. Try that, and you'll see that the C5 channel pops out. The EOG, Fp1, and Fp2 sites also pop out, but that's because we haven't gotten rid of the blinks.

The second way of identifying problematic channels is to click the **Outliers** button. Turn off the **Color heatmap** option and then turn on the **Outliers** option. It will highlight any cells that are more than N standard deviations (SDs) away from the mean aSME value for that time period. The default setting of N is 2, which means that it will highlight cases that are more than 2 SDs above the mean. But when computing the SD, it would make sense to leave out the EOG, Fp1, and Fp2 channels, because they are a poor comparison for the other channels as a result of the blinking. The **Outliers** feature therefore allows you to specify a subset of the channels for computing the SD. Enter **3:28** in this text box. You'll now see that the C5 channel is highlighted in most or all time periods. When combined with the fact that C5 was clearly problematic in our visual inspection of the data, these aSME results provide good reason to interpolate this channel.

What about F8, which also looked pretty bad in our visual inspection? The aSME for F8 is well within the range of the other frontal electrode sites. This indicates that the high-frequency noise in F8 isn't really impacting our ability to quantify the amplitude of the MMN using the mean voltage between 125 and 225 ms. That noise could be problematic for other measures (e.g., the peak amplitude), but it's not problematic for the planned analyses of the present study (especially given that the main analysis will be limited to FCz). So, there would be little value in interpolating this channel given the way the data will be analyzed.

In the analyses provided in the ERP CORE paper (Kappenman et al., 2021), F8 was interpolated because we did not yet have the SME metric and it "looked" noisy. Now I would make a different decision. I guess you can teach an old dog new tricks!

Now that we've decide to interpolate C5 but not F8, it's time to implement the interpolation. Close the Data Quality viewer window, and select the original continuous dataset (**1_MMN_preprocessed**) in the **Datasets** menu. Then select **EEGLAB > ERPLAB > Preprocess EEG > Selective Electrode Interpolation**. Put **11** in the **Interpolate Electrodes** box (because C5 is Channel 11). We don't want our bipolar EOG electrodes to be used for the interpolation, because they don't have the same reference as C5 and would mess up the interpolation, so put **32 33** in the **Ignore Electrodes** box. Select **Spherical** as the **Interpolation Method** and click the **Interpolate** button. Name the resulting dataset **1_MMN_preprocessed_interp** (and you may want to save it to your disk because we'll use it in the following exercises).

Now look through the EEG data using **EEGLAB > Plot > Channel data (scroll)**. The C5 channel should now look beautiful, even later in the session when the original C5 channel looked terrible. Success! However, keep in mind that the C5 channel now contains *estimated* voltages, not *measured* voltages. But that's good enough for our present purposes, especially given that the main analyses will be performed on a different channel.

One last note about interpolation: If you are using the average across sites as the reference, the data from any bad channels will contaminate all the channels. You might therefore want to use a single electrode (or a pair of electrodes such as P9 and P10) as the reference prior to interpolation. You can then re-reference to the average of all sites after interpolation. A more complex but more robust approach is implemented by the PREP pipeline (Bigdely-Shamlo et al., 2015).

This page titled 7.6: Exercise - Interpolating Bad Channels is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck.

7.7: Matlab Script For This Chapter

There is no script for this chapter. The script for the next chapter shows how to implement interpolation along with artifact rejection.

This page titled 7.7: Matlab Script For This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

7.8: Key Takeaways and References

Key Takeaways

- The overarching goal for your EEG preprocessing is to maximize the likelihood that you will obtain an accurate answer to the scientific question your study is designed to answer. You can ignore any of my specific suggestions for your preprocessing pipeline if you have a better way of reaching that goal.
- You can more easily obtain an accurate answer to your scientific question if you look carefully at each participant's data prior to doing the preprocessing. By examining the data, you'll be able to adjust the preprocessing to reflect the unique problems of each individual participant's data.
- The standardized measurement error (SME) provides a useful metric for knowing whether a "bad channel" is really problematic with respect to the analyses you will be performing with your data.
- Interpolation is a low-risk procedure when the channel being interpolated will not be used for your main analyses. But if the channel will be used in your main analyses, you need to think carefully about whether to interpolate the channel or exclude the participants from analysis.

References

Bigdely-Shamlo, N., Mullen, T., Kothe, C., Su, K.-M., & Robbins, K. A. (2015). The PREP pipeline: Standardized preprocessing for large-scale EEG analysis. *Frontiers in Neuroinformatics*, *9*. https://doi.org/10.3389/fninf.2015.00016

Kappenman, E. S., Farrens, J. L., Zhang, W., Stewart, A. X., & Luck, S. J. (2021). ERP CORE: An open resource for human event-related potential research. *NeuroImage*, *225*, 117465. https://doi.org/10.1016/j.neuroimage.2020.117465

This page titled 7.8: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8: ARTIFACT DETECTION AND REJECTION

Learning Objectives

In this chapter, you will learn to:

- Conceptualize artifact rejection in terms of the overarching goal of accurately answering the scientific question that your experiment was designed to address
- Implement algorithms that work particularly well for detecting blinks, saccadic eye movements, and a broad class of artifacts termed *commonly rejected artifactual potentials*
- Select optimal parameters for the artifact detection parameters
- Evaluate the effectiveness of your artifact rejection procedures on the averaged ERP waveforms, including both data quality and confounds
- Implement a two-stage artifact rejection procedure for ensuring that experiments with lateralized stimuli or lateralized responses are not contaminated by small but consistent eye movements

EEG recordings are often filled with large artifacts. In most areas of research, blinks are the most problematic. They're large (often 200 μV), occur frequently (on over 50% of trials in many participants), and may differ systematically across groups or conditions, creating a significant confound if they aren't properly addressed. In research with infants, small children, or people who are required to move around during the task, movement-related artifacts are also a major issue. In my own area of research, eye movements toward lateralized targets are the most significant artifact.

However, in most of the ERP papers I read that use artifact rejection, it doesn't seem that much thought went into the strategy for dealing with artifacts. These papers typically use a very primitive algorithm for detecting trials with artifacts, and they use the same rejection threshold for all participants (even though artifacts differ quite a bit across individuals). This chapter is designed to help you conceptualize and implement artifact rejection in a more sophisticated manner, allowing you to minimize artifact-related confounds and maximize your data quality.

8.1: Data for This Chapter

8.2: Overview

8.3: Background- Why Do We Reject Artifacts?

8.4: Background- The General Approach

8.5: Exercise- Simple Blink Detection

8.6: Exercise- Adjusting the Threshold

8.7: An Iterative Approach to Setting Parameters

8.8: Exercise- Data Quality and Confounds

8.9: Exercise- Better Blink Detection

8.10: Exercise- Detecting Eye Movements

8.11: Exercise- Deciding on a Threshold for Eye Movements

8.12: Exercise- Commonly Recorded Artifactual Potentials (C.R.A.P.)

8.13: Using Artifact Detection to Avoid Changes to Visual Inputs

8.14: The ERP CORE N2pc Experiment

8.15: Exercise- Visualizing the Eye Movements

8.16: Exercise- Using the Averaged HEOG to Visualize Consistent Eye Movements

8.17: Exercise- A Two-Stage Strategy for Eliminating Small But Consistent Eye Movements

8.18: Matlab Script For This Chapter
8.19: Key Takeaways and References

This page titled 8: Artifact Detection and Rejection is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.1: Data for This Chapter

The data we'll be using for the exercises in this chapter can be found in the Chapter_8 folder in the master folder: https://doi.org/10.18115/D50056.

This page titled 8.1: Data for This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.2: Overview

When Javier Lopez-Calderon and I designed the artifact rejection process in ERPLAB, Javier suggested that we should refer to the process of flagging epochs as *artifact detection*, because those epochs aren't actually deleted from the EEG dataset. The rejection (or exclusion) of the flagged epochs actually occurs during the averaging process. I thought that was a great idea. So, I will use the phrase *artifact detection* to refer to the process of determining which epochs should be flagged and the phrase *artifact rejection* to refer to the process excluding flagged epochs from the averaged ERPs.

EEGLAB and ERPLAB also contain a separate set of routines that actually *delete* problematic segments of data from the continuous EEG. These routines are primarily used as a preprocessing step in the artifact correction process, as will be described in the next chapter. In the present chapter, the term *artifact rejection* will be used in the context of epoched EEG data.

Chapter 6 in Luck (2014) provides important theoretical background about a broad range of artifacts and about the nature of the artifact detection/rejection process. It will be helpful (but not absolutely necessary) for you to read that chapter before proceeding. The goal of the present chapter is to make this theoretical background more concrete and demonstrate the practical issues that arise in real data.

We'll focus on data from a few example participants who I selected not because they had "good" data but because they were quite challenging. The data will come from two of the ERP CORE experiments (Kappenman et al., 2021), one looking at the mismatch negativity (MMN) and one looking at the N2pc component. The MMN paradigm was described in the previous chapter, and the N2pc paradigm will be described later in the present chapter. We'll mainly consider blinks and eye movements, because they're the most common large artifacts, but the exercises will also teach you general principles that you can use for other kinds of artifacts and other types of experiments.

Artifact correction has many advantages over artifact rejection, and it will be covered in the next chapter. In almost all cases, however, I recommend combining rejection and correction. Also, the problems created by artifacts are the same whether you're using rejection or correction, so you'll need to read at least the first part of this chapter even if you're mainly planning to use correction instead of rejection.

Organization of the Chapter

Artifact detection is conceptually simple, but it requires a lot of decisions, and you need to know how to make the best decisions to achieve the best possible data. As a result, this chapter is pretty long. Here's the overall structure:

- The first part of the chapter describes three main problems that are typically addressed by artifact rejection and provides an overview of the detection+rejection process.
- The second part of the chapter takes you through a series of exercises in which you'll see how to detect and reject blinks, eye movements, and other miscellaneous artifacts in the context of the MMN experiment.
- The last part of the chapter takes you through exercises that teach you how to detect and reject small but consistent eye movements, which are especially problematic in experiments with lateralized target stimuli (especially N2pc and CDA experiments) or lateralized responses (mainly LRP experiments). If you don't conduct experiments of this sort, you can skip this part of the chapter.

The exercises focus on data from only two participants in each experiment. I strongly recommend looking at the data from additional participants and repeating the artifact detection procedures with those participants. You can find the data from additional participants in the MMN_Data N2pc_Data folders inside the Chapter_8 folder.

Keep in mind that we did nothing to try to minimize blinking when designing and running these experiments because we knew we would use artifact correction rather than artifact rejection to deal with blinks. As a result, many participants blinked on a large proportion of trials. We would have needed to exclude many of these participants if we had rejected rather than corrected blinks.

This page titled 8.2: Overview is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.3: Background- Why Do We Reject Artifacts?

Let's start by asking why we reject epochs containing artifacts. You might think this is a dumb question. Obviously, we don't want any artifacts in the epochs that we will be using to make our averaged ERPs! However, every single time point in every scalp EEG recording in human history contains artifactual activity. That is, the scalp EEG signal is always a mixture of brain activity, non-neural biological signals (e.g., skin potentials, EMG), and non-biological signals (e.g., line noise from nearby electrical devices). If we rejected every epoch containing an artifact, we wouldn't have any data left.

We therefore reject epochs that *problematic* artifacts, defined as artifacts that interfere with the fundamental goal described in the previous chapter: **accurately answering the scientific question that the experiment was designed to address**. There are three common ways in which artifacts can be problematic from this perspective:

1. **Reduced Statistical Power**. Artifacts add noise to the data, reducing the signal-to-noise ratio (SNR) of our averaged ERPs. This makes our amplitude and latency measurements less precise, which in turn decreases our statistical power. However, when we reject epochs containing artifacts, we have fewer epochs in our averages, and that also makes the averages noisier and decreases our power. As a result, we need to balance the need to eliminate epochs with large artifacts with the need to include as many epochs as possible.
2. **Systematic Confounds**. Artifacts can produce systematic confounds in our studies. For example, if participants blink more in response to deviant stimuli than in response to the standards, we will see a difference between deviants and standards in the averaged ERPs that is due to EOG activity rather than to brain activity. As we will see in one of the exercises in this chapter, this is not just a theoretical possibility.
3. **Sensory Input Problems**. In visual experiments, EOG artifacts can indicate a problem with the sensory input. For example, if a blink occurs just before or during the stimulus presentation, this means that the stimulus wasn't actually seen by the participant. Similarly, a deflection in the horizontal EOG can mean that the eyes weren't pointed at the center of the display. The first exercises in this chapter will use data from an auditory experiment so that we won't need to deal with this issue initially. However, we'll switch to a visual experiment in the last part of the chapter to examine how ocular artifacts might alter the sensory input.

Artifact correction can be much better than artifact rejection for addressing the problem of reduced statistical power, because we get to keep all of our epochs. Correction can also help with systematic confounds, but only to the extent that the correction fully removes the artifacts and doesn't produce any new artifacts. For example, if correction reduces the blinks by 99%, the remaining blink activity would still be 1-2 μV in the frontal channels (because uncorrected blinks are typically 100-200 μV in these channels). That might be enough to produce a significant confound. Artifact correction doesn't help at all with sensory input problems. For example, if participants are looking leftward in one condition and rightward in another condition, correcting for the EOG voltage produced by the eye movements doesn't eliminate the confound of a different sensory input in the two conditions.

For these reasons, I recommend combining artifact correction and artifact rejection for most experiments. You can use correction to minimize the noise produced by blinks (and certain other artifacts, as discussed in Chapter 6 in Luck, 2014). And then you can use rejection to eliminate epochs with blinks or eye movements near the time of the stimulus (for visual experiments) and to eliminate epochs that contain large artifacts that are not easily corrected (e.g., occasional EMG bursts).

When we're using rejection to deal with reduced statistical power, we would ideally have an algorithm that determines which epochs should be removed to best balance the benefits of eliminating noisy epochs with the cost of reducing the number of epochs that are included in our averages. There are several methods that take this approach (e.g., Jas et al., 2017; Nolan et al., 2010; Talsma, 2008). However, they try to optimize the signal-to-noise ratio in a generic sense, which may not actually maximize statistical power for the specific amplitude or latency measurement that you will be using to answer your scientific question.

The standardized measurement error (SME) was specifically designed to quantify the data quality for your specific amplitude or latency measure and is directly related to statistical power (Luck et al., 2021). The SME can therefore be used to determine the artifact detection parameters that will lead to the best power. At this moment, ERPLAB doesn't include an automated approach for determining which trials should be rejected to minimize the SME, but you can manually check the SME to compare different artifact detection parameters. We'll use this approach in several of the exercises later in this chapter.

Keep in mind, however, that low noise isn't the only consideration. For example, imagine that a participant blinked on every trial. This would be very consistent, which would lead to a low SME (because the SME reflects the amount of trial-to-trial variability in the data). However, the resulting averaged waveforms would mainly contain blink activity instead of ERPs, which could lead to

completely incorrect conclusions. So, you need to consider potential confounds as well as the data quality when selecting artifact detection parameters.

You should also keep in mind that the SME quantifies the data quality of the averaged ERPs (which is, of course, influenced by the noise in the EEG). As a result, the SME depends on the number of trials being averaged. That's a good thing, because the number of trials can have a big impact on your statistical power (Baker et al., 2020).

This page titled 8.3: Background- Why Do We Reject Artifacts? is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.4: Background- The General Approach

The general artifact detection/rejection procedure is pretty straightforward. For each participant, you apply an artifact detection algorithm to the epoched EEG data. That algorithm determines which epochs contain artifacts, and those epochs are "marked" or "flagged". When you compute averaged ERPs, those epochs are simply excluded from the averages.

There are two main classes of approaches to determining which epochs should be flagged. The approach I prefer involves knowing as much as possible about the nature of the artifacts (e.g., the typical waveshape and scalp distribution of a blink) and designing algorithms that are tailored to those artifacts. The other approach involves asking which epochs are extreme or unusual in a statistical sense. I don't like this statistical approach as much because it's not clear that "weird" epochs are necessarily problematic. How many movies have you seen about high school students in which the "popular" students rejected kids who seemed "weird" but were actually quite delightful? I just don't like the idea of rejecting trials that seem "weird" but might actually be delightful.

I suspect that the two approaches actually end up mostly flagging the same epochs for rejection, so it may not matter which approach you use in the end. And the most important factor when deciding what approach to take is to have a clear understanding of the ultimate goal of artifact rejection. As described in the previous chapter, **the ultimate goal is to accurately answer the scientific question that the experiment was designed to address**. So, go ahead and use statistical approaches to flagging epochs for rejection if that leads you to this goal. Also, every area of research is different, so you should feel free to ignore any of my specific pieces of advice if you have a better way of accurately answering your scientific questions.

As described in detail in Chapter 6 of Luck (2014), I advocate setting the artifact detection parameters individually for each participant. In the present chapter, I will show you how to select appropriate parameters manually. There are also completely automated approaches to selecting the parameters (e.g., Jas et al., 2017; Nolan et al., 2010). I haven't used those approaches myself, but they seem fairly reasonable. However, many people who use these approaches on a regular basis recommend verifying that the parameters are working well and not just accepting them blindly. So, these approaches end up not being fully automatic. An ERP Boot Camp participant, Charisse Pickron, suggested another excellent use for the automated algorithms: When you're first learning to set artifact detection parameters, you can check your parameters against the automated parameters so that you have more confidence in the parameters that you've set.

Some participants have so many artifacts that an insufficient number of trials remains to create clean averaged ERP waveforms. The standard procedure is to exclude those participants from the final analyses. However, you must have an objective, a priori criterion for exclusion. Otherwise, you will likely bias your results (as explained in the text box below). In my lab's basic science research, we always exclude participants if more than 25% of trials are rejected because of artifacts (aggregated across conditions). In our research on schizophrenia, where the data are noisier and the participants are much more difficult and expensive to recruit, we exclude participants if more than 50% of trials are rejected. We apply these criteria rigidly in every study, without fail. A different criterion might make sense in your research. Just make sure that the criterion is objective and determined before you see the data.

Although this chapter focuses on detecting and rejecting artifacts, I would like to encourage you to start thinking about artifacts before you record the EEG. This advice follows from something I call *Hansen's Axiom*: "There is no substitute for clean data" (see Luck, 2014). It's much better to minimize artifacts during the recording instead of trying to reject or correct them afterward. Strategies for minimizing artifacts are described in Chapter 6 of Luck (2014).

Excluding Participants is Dangerous!

Imagine that you run an experiment, and your key statistical analysis yields a p value of .06 (the most hated number in science!). You spent two years running the study, and the effect is going in the predicted direction, but you know you can't publish it if the effect isn't statistically significant. Given the millions of steps involved in an ERP experiment, you might go back through your data to make sure there wasn't an error in the analysis. And imagine you find that 80% of the trials were rejected for one of the participants, leading to incredibly noisy data. You would (very reasonably) conclude that this participant should not have been included in the final analyses. So, you repeat the analyses without this participant, and now the p value is .03. Hallelujah! You can now publish this important study.

> Now imagine that the *p* value was originally .03, so you have no reason to go back through all the data. And imagine that your final sample included a participant with 80% of trials rejected and very noisy data. And further imagine that excluding this participant would lead to a *p* value of .06. But because the effect was significant in the initial analysis, you had no reason to go back through the data, so you wouldn't notice that this participant should have been excluded. And even if you did, would you really have the intestinal fortitude to exclude the participant, even though this means that your *p* value is now .06?
>
> This example shows why you need an a priori criterion for excluding participants. If you decide whom to exclude after you've seen the results of the experiment, you're more likely to notice and exclude participants when it makes your *p* value better (because it was >.05 before you excluded participants) than when it makes your *p* value worse (because you don't notice participants who should be excluded when $p < .05$). As a result, this creates a bias to find $p < .05$ even when there is no true effect. So, you should develop an a priori criterion for excluding participants before you see the results.

This page titled 8.4: Background- The General Approach is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.5: Exercise- Simple Blink Detection

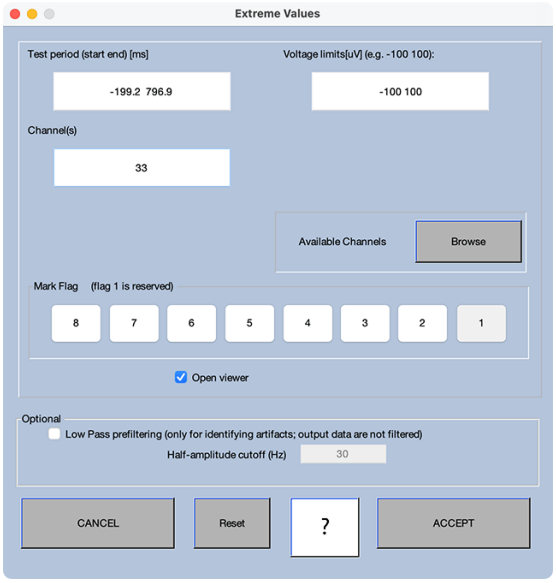
It's now time to see how artifact detection works in actual data. We'll start by detecting blinks, which are big and easy to detect. We'll begin with the most common blink detection algorithm, which simply asks whether the voltage falls outside a specified range at any point in the epoch for a given channel. This is a pretty primitive algorithm, and the next exercise will show you a much better approach.

Start by loading the data from Subject #1 in the MMN experiment, which is in the file named **1_MMN_preprocessed_interp**. This is the same dataset that we examined in the previous chapter, after the C5 channel was interpolated. Artifact detection operates on epoched data, so select **EEGLAB > ERPLAB > Extract bin-based epochs**, using an epoch of -200 to 800 ms and **Pre** as the baseline. Name the resulting dataset **1_MMN_preprocessed_interp_be**. Now select **EEGLAB > ERPLAB > Artifact detection in epoched data > Simple voltage thresholds** and set the parameters as shown in Screenshot 8.1. In particular, specify **33** as the Channel (this is the VEOG-bipolar channel) and voltage limits of **-100 100**. These voltage limits indicate that an epoch should be flagged for rejection if the voltage is more negative than -100 µV or more positive than +100 µV at any time in this channel. Blinks will be largest in the VEOG-bipolar channel, so there's no point in looking for blinks in other channels. We'll worry about other types of artifacts later.

You might wonder why the default **Test period** is set to **-199.2 796.9** rather than **-200 800** (which is what you specified for the epoch). The answer is that the data were originally sampled at 1024 Hz and were then downsampled to 256 Hz for the analyses provided in the ERP CORE paper. As a result, we have a sample every 3.90625 ms, and we don't have samples at exactly -200 and +800 ms. In the previous chapters, I downsampled to 200 Hz instead, yielding a sample every 5 ms. But I thought it was time for you to see what happens when the sampling period isn't a nice round number.

Don't worry about the flags; I'll discuss them later. Click **ACCEPT** to apply the algorithm to the selected dataset.

Screenshot 8.1



Once the algorithm has finished, you will see two windows. One is the standard window for saving the updated dataset. The other is the standard window for plotting EEG waveforms. The idea is that you'll use the plotting window to make verify that the artifact detection worked as desired. If so, you'll use the other window to save the dataset. If you're not satisfied with which epochs were flagged, you'll click **Cancel** and try again with new artifact detection parameters.

However, before you start scrolling through the plotting window, it's important to see how many artifacts were detected. This information is shown in the Matlab command window. You can see the number and percentage of epochs in which artifacts were detected in each bin and the total across bins. In most cases, I mainly worry about the total (because the percentage is much

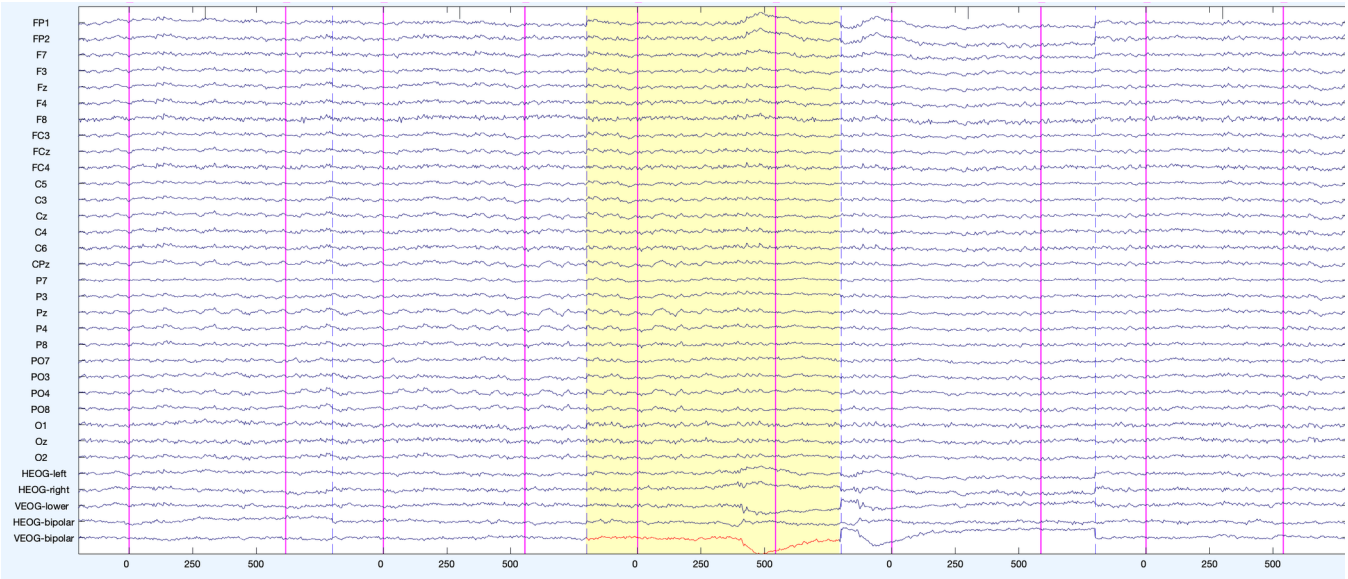
more meaningful when based on a large number of trials). You can see that 17.8% of epochs were rejected. That's reasonable. As noted above, my lab always throws out any participants for whom more than 25% of epochs were rejected, so this participant would be retained.

Now let's scroll through the EEG data and see how well the algorithm performed at flagging epochs with blinks and not flagging epochs without blinks. I recommend setting the vertical scale to 100. Keep in mind that we're now looking at epochs rather than continuous data, and the plotting window shows 5 epochs per screen by default. (If you have a large screen, I recommend going to **Settings > Time range to display** in the plotting window and telling it to show 10 or even 15 epochs per screen.) Epochs that have been flagged for artifacts are highlighted in yellow. Recall that Subject #1 had beautiful EEG in the beginning of the session, so you won't see any flagged epochs at first. But you still need to make sure that there aren't any blinks that weren't detected, so scroll through the epochs and look at the VEOG and Fp1/Fp2 channels to make sure that everything looks okay.

Epoch 103 should be the first marked epoch (see Screenshot 8.2). The waveform for the offending channel is drawn in red. You can see a classic blink shape and scalp distribution. Success!

But this is immediately followed by a failure in Epoch 104. Because we have a stimulus every 500 ms, but each epoch lasts for 1000 ms, the initial part of one epoch is the same as the latter part of the previous epoch. So, the blink that peaked just before 500 ms in Epoch 103 appears just before time zero in Epoch 104. Because the blink is during the baseline in Epoch 104, the baseline correction procedure reduced the maximum voltage during the epoch, and the blink is not detected.

Screenshot 8.2



Keep scrolling. You'll notice a large muscle burst in Epoch 107 that isn't flagged. But that's OK—we're looking for blinks right now, and we'll test for other artifacts later. You'll also see a blink that appears in the poststimulus period of Epoch 121 and in the prestimulus period of Epoch 122. This blink was larger than the one in Epochs 103 and 104, and the blink was successfully detected in both Epochs 121 and 122. The blink appearing in Epochs 162 and 163 was also successfully detected. However, the blink that appears in Epochs 169 and 170 was missed in Epoch 170.

If you keep scrolling, you'll also see that high-frequency noise (almost certainly EMG) caused Epoch 463 to be flagged. You can tell that there was no blink in this epoch because there was no positive-going voltage in Fp1 and Fp2, just a small negative-going voltage in VEOG-lower combined with some high-frequency noise. There is no reason to reject this epoch: The voltages in VEOG-lower are very localized and unlikely to impact our scalp EEG recordings. Epochs 525 and 526 are also unnecessarily flagged for rejection. In these epochs, a combination of a slow, non-blink-like voltage deflection and high-frequency noise in Fp2 (but without an opposite-polarity deflection in VEOG-lower) produced a large enough voltage in VEOG-bipolar for the voltage to exceed our ±100 µV threshold.

You can now close the plotting window and save the dataset that was created, naming it **1_MMN_preprocessed_interp_be_ar100**. We'll need it for a later exercise.

This page titled 8.5: Exercise- Simple Blink Detection is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.6: Exercise- Adjusting the Threshold

In this exercise, we'll see how adjusting the threshold changes which epochs are flagged for rejection. Let's start by seeing if we can detect some of the blinks that were missed with our ±100 μV threshold. Make **1_MMN_preprocessed_interp_be** the active dataset, and then select **EEGLAB > ERPLAB > Artifact detection in epoched data > Simple voltage thresholds.** Change the voltage limits to **-50 50** to indicate that an epoch should be flagged for rejection if the voltage is more negative than -50 μV or more positive than +50 μV at any time in the VEOG-bipolar channel. Click **ACCEPT** to run the artifact detection routine.

The first thing you should look at is the proportion of rejected trials, which is shown in the Matlab command window. Whereas 17.8% of epochs were flagged when our threshold was ±100 μV, now 42.9% have been flagged. If we were to use this ±50 μV threshold, this participant would need to be excluded from the final analyses (because my lab excludes participants if more than 25% of trials were rejected). Obviously, you don't want to exclude participants if you don't have to, so let's see if we really want to use this threshold.

If you scroll through the epochs in the plotting window that appeared, you'll see that the blinks in Epochs 104 and 170 have been detected with this threshold. That's the good news. But if you keep scrolling, you'll see the bad news: Many epochs without a clear blink are now flagged for rejection (e.g., Epochs 408, 424, 432, and 435-437). In general, decreasing the threshold for rejection increases our *hit rate* (the proportion of blinks that were detected) but also increases our *false alarm rate* (the proportion of non-blink epochs that are flagged for rejection).

Now let's try increasing our threshold to avoid flagging epochs 463, 525, and 526, which were unnecessarily flagged for rejection with our original threshold of ±100 μV. Close the plotting window and the window for saving the dataset, make sure that **1_MMN_preprocessed_interp** is still the active dataset, and run the artifact detection routine using voltage limits to **-150 150**.

The percentage of flagged trials has now dropped to 11.3%. That's good insofar as increasing the number of accepted trials will increase our signal-to-noise ratio. But it might be bad if a lot of blinks have escaped detection.

If you scroll through the data, you'll see that Epochs 463, 525, and 526 are no longer flagged for rejection, which is good. However, several clear blinks have been missed (e.g., Epochs 103, 191, 201). In general, increasing the threshold for rejection decreases the hit rate but also decreases the false alarm rate.

The take-home message of this exercise is that adjusting the threshold impacts both the hit rate and the false alarm rate, making one better and the other worse. You'll need to choose a threshold that balances the hit rate and false alarm rate in a way that best helps you achieve the fundamental goal, which is to accurately answer the scientific question that the experiment is designed to address. Is that goal best met by ensuring that all epochs with blinks are rejected, even if this means rejecting some perfectly fine epochs? Or is the goal best met by optimizing the number of included epochs, even if a few blinks escape rejection?

The answer will depend on the nature of your scientific question, the details of your experimental design, and the nature of the artifacts in your data. In particular, if blinks differ systematically across bins (especially in the time range of the ERP components of interest), then you will usually need to make sure that the vast majority are rejected to avoid confounds. And if you have a reasonably large number of trials, throwing out a few trials without blinks won't really change your signal-to-noise ratio very much (see the text box below). So, in most cases, I recommend erring on the side of throwing out too many trials rather than allowing some large artifacts to remain in the data.

Also, as you'll see in some of the later exercises, you can both increase your hit rate and decrease your false alarm rate by choosing a better algorithm for determining which epochs contain artifacts. The simple voltage threshold we've used in this example is a poor way of detecting blinks, and I'm always amazed that many software packages don't provide better algorithms.

Don't Stress About Rejecting a Few Trials

It's easy to get stressed out about excluding 20% or 50% of trials because of artifacts. Is this going to cause a 20% or 50% reduction in your data quality? It turns out that excluding trials has a smaller impact on data quality than you might expect.

This is because the signal-to-noise ratio (SNR) increases as a function of the square root of the number of trials. This square root rule is really annoying when you're designing your experiment, because doubling the number of trials only increases your SNR by 41% (because sqrt(2) = 1.41). But the same rule means that you don't lose very much SNR when you have to exclude some trials.

As an example, imagine that your single-trial SNR is 1:2 or 0.5 (i.e., your signal is half as big as your noise in the raw EEG epochs). If you average together 100 trials, the resulting SNR is 0.5 x sqrt(100) = 5. Now imagine that you have to exclude 20 trials because of artifacts. Now your SNR is 0.5 x sqrt(80) = 4.47. That is, you've decreased the number of trials by 20%, but your SNR has dropped by only about 10%.

Now imagine that you have to exclude 50 trials. The resulting SNR is 0.5 x sqrt(50) = 3.54. Even though you've decreased the number of trials by 50%, your SNR has dropped by only about 30%.

As mentioned earlier, you should have an a priori threshold for excluding participants on the basis of the percentage of rejected trials, and the square root rule will help you decide on what percentage to use as your threshold. How much is your statistical power reduced by excluding a participant versus including participants with a reduced SNR? Usually, your power is reduced more by excluding the participant unless so many trials were rejected that the SNR is truly awful.

However, this assumes that the artifacts are random, and the only difference between participants with lots of artifacts and participants with few artifacts is the number of trials available for averaging. In my experience, this assumption is false. Participants with a large number of artifacts tend to be less compliant with instructions, may be more sleep-deprived, and often have poorer EEG signals even on the trials without artifacts. Our threshold for excluding participants (25% in basic science studies, 50% in schizophrenia studies) is lower than would be necessary if we solely considered the square root rule.

In the future, we may switch to a rule that is based on the SME—a direct measure of data quality—rather than the percentage of rejected trials. This might make it possible to avoid excluding participants whose averaged ERPs are quite clean even though they had a lot of rejected trials and to exclude participants who didn't have a lot of rejected trials but had noisy averages nonetheless. This approach could be particularly valuable in research participants for whom it is difficult to obtain a large number of trials (e.g., infants and small children).

This page titled 8.6: Exercise- Adjusting the Threshold is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.7: An Iterative Approach to Setting Parameters

Many researchers use a preset artifact detection threshold for all participants, but this is not optimal. A threshold that works well for one participant might fail to flag all the blinks for another participants and might lead many non-blink epochs to be flagged in a yet another participant. I therefore recommend the interactive and iterative approach shown in Figure 8.1. It involves starting with an initial set of best-guess parameters, seeing whether these parameters adequately flag the epochs that should be rejected, and then adjusting the parameters as necessary until you are satisfied. As will be described in the next section, it also involves using the standardized measurement error (SME) to help you determine which parameters lead to the best balance of eliminating noisy trials while still having enough trials to obtain a good averaged ERP waveform.

Make sure you keep a record of the parameters you choose for each participant. I recommend using a spreadsheet for this. The example script at the end of the chapter shows you how a script can read the artifact detection parameters from a spreadsheet and then perform the artifact detection with these parameters. That will keep you from having a meltdown when you need to reprocess your data the seventh time. In fact, it's a good idea to set the parameters manually and then immediately reprocess the data using a script with those parameters. It's easy to make a mistake when you're processing data by pointing and clicking in a GUI, and this approach of manually selecting the parameters and then implementing them in a script gives you the customized parameters that you want while avoiding point-and-click errors.

I find this iterative approach to be reasonably quick (5-10 minutes for most participants once you've become well practiced). And it does an excellent job of addressing the three types of problems that were described at the beginning of the chapter. However, other approaches may be better in certain cases.

If you have a small number of trials per participant (as in many infant studies) or a small number of highly valuable participants (as in some studies of lesion patients), you may want to manually mark epochs for rejection during the visual inspection process. That is, you can mark an epoch for rejection by simply clicking on it (select **EEGLAB > Tools > Reject data epochs > Reject by inspection**; see the ERPLAB documentation for information about how to integrate these marks with ERPLAB). However, this approach is slow and awkward when you have more than ~20 trials per participant or more than ~20 participants.

Another alternative is to use one of the algorithms that automatically set the parameters for each participant (e.g., Jas et al., 2017; Nolan et al., 2010; Talsma, 2008). This approach is best suited for very large datasets (e.g., >100 participants), and it should be followed by manual verification for each participant. Note that most current algorithms assess the overall noise level of the data rather than assessing the data quality for the specific amplitude or latency measure that you will be using as the main dependent variable to test your scientific hypotheses (which is what the SME does). As a result, these algorithms may not actually select optimal parameters in terms of statistical power.

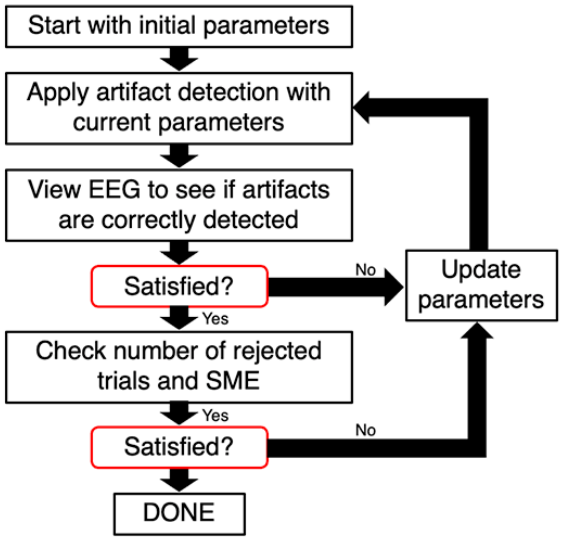


Figure 8.1. Iterative procedure for setting artifact detection parameters. If no satisfactory parameters can be found without exceeding the maximal allowable percentage of trials with artifacts, then the participant must be excluded from the final analyses.

This page titled 8.7: An Iterative Approach to Setting Parameters is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.8: Exercise- Data Quality and Confounds

When deciding on artifact rejection parameters, a key question is whether the data quality will be increased or decreased by making the rejection threshold more liberal (rejecting fewer trials) or more conservative (rejecting more trials). We can answer that question quantitatively by looking at the SME values that result from different rejection parameters. However, we also need to determine whether the artifacts are creating confounds, which involves inspecting the averaged ERP waveforms in several ways. In this exercise, we'll go through the steps needed to check both the data quality and the waveforms.

Let's start by looking at the SME values and waveforms that we get without any rejection. Select the dataset with the ±100 μV threshold (**1_MMN_preprocessed_interp_be_ar100**) and then select **EEGLAB > ERPLAB > Compute averaged ERPs**. Near the top of the averaging GUI, select **Include ALL epochs (ignore artifact detections)**. This will allow us to see what we would get without any artifact rejection. As described in the preceding chapter, select **On – custom parameters** in the **Data Quality Quantification** section and add a time window of 125 to 225 ms (the time window we will ultimately use to measure MMN amplitude). Click **RUN**, and name the resulting ERPset **1_ar_off**.

If you look at the data quality table, you'll see that the aSME at FCz during the custom time period of 125-225 ms is 0.5524 for the deviant stimuli (Bin 1) and 0.3641 for the standards (Bin 2). (Leave the data quality window open for comparison with later steps.)

Now plot the ERP waveforms. You can see some relatively large, slow deviations in the Fp2, VEOG-lower, and VEOG-bipolar channels. The key channels are shown in Figure 8.2.A, but I've applied a 20 Hz low-pass filter to more easily see the blink-related activity. If you look closely at the FCz channel, which will be the primary channel for our MMN analyses, you can see some of this same blink-related activity (i.e., the "tilt" in the prestimulus period).

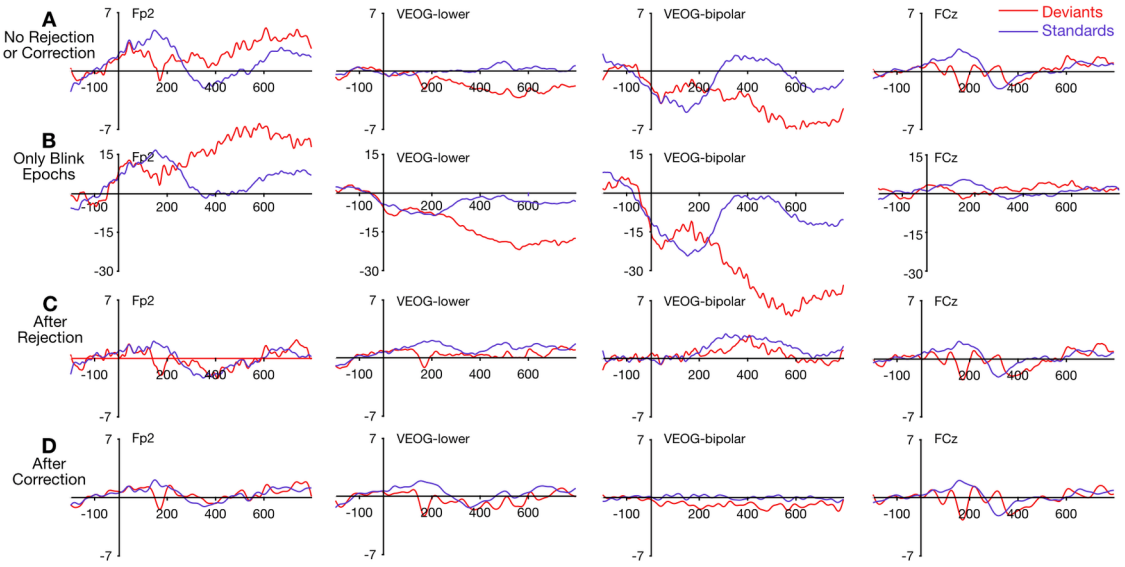


Figure 8.2. Averaged ERPs from Subject #1 in the MMN experiment, with no rejection or correction (A), including only epochs flagged for blinks (B), after rejection of epochs with blinks using an absolute voltage threshold of ±100 μV (C), and after ICA-based correction of blinks (D). To improve visualization of the data, the averaged waveforms were low-pass filtered with a half-amplitude cutoff at 20 Hz and a slope of 12 dB/octave. Note the different scale in (B).

Are the large, slow voltage deviations in Figure 8.2.A a result of blinks that are confounding the ERPs, or are they brain activity? One way to answer this question is to look for a polarity reversal under versus over the eyes. The activity prior to ~200 ms is more negative for deviants than for standards both under the eyes (VEOG-lower) and above the eyes (Fp2), so this experimental effect is probably not blink-related. However, the later voltage is more negative for deviants than for standards under the eyes but more positive for deviants than for standards above the eyes. This polarity inversion is suggestive of a blink confound (although it is possible for brain-generated activity to invert in polarity above versus below the eyes).

Another way to address this question is to reverse the usual artifact rejection procedure and include only the flagged trials in

our averages, leaving out the unflagged trials. Any blink-related confounds should be much bigger in these averages, whereas brain activity should not. To do this, run the averaging tool again, but this time select **Include ONLY epochs marked with artifact rejection**. If you look at the resulting waveforms, you'll see that the differences between deviants and standards prior to ~200 ms are about the same as before, but the differences after 200 ms are now much larger (see Figure 8.2.B). This provides additional evidence that the participant was more likely to blink following deviant stimuli than following rare stimuli (even though the auditory stimuli were task-irrelevant). Thus, blinking is not just a source of noise in this experiment; it's a confound that could create artifactual differences between conditions during the latter part of the epoch.

Now let's look at the data quality and waveforms when we reject trials that were flagged for blinks. You can just repeat the averaging process, but select **Exclude epochs marked during artifact detection** so that the flagged epochs are excluded. If you keep the previous table of data quality values open, and open a new table for the current ERPset, you'll see that the aSME at FCz from 125-225 ms has dropped from 0.5524 to 0.5436 for the deviant stimuli and from 0.3641 to 0.3377 for the standards. We have fewer trials as a result of artifact rejection, but the data quality has improved. The improvement isn't very large, because we now have fewer trials and because the blinks that we've removed are not huge at the FCz site. But it's still good to see that we get better data quality even though we have fewer trials in the averages. (There is a much larger improvement in data quality at Fp1 and Fp2, where the blinks were a large source of trial-to-trial variation.)

Even though rejecting epochs with blinks hasn't improved our data quality much, at least it hasn't hurt our data quality. And rejecting blinks helps us avoid blink-related confounds: if you plot the waveforms, you'll see that we've reduced the slow voltage deviations in the VEOG, Fp2, and FCz channels. You can see this quite clearly in Figure 8.2.C, where the voltage deviations are now reduced relative to the no-rejection data shown in Figure 8.2.A. You can see substantial differences between the standards and the deviants in the VEOG-bipolar channel in the absence of artifact rejection, which suggests that the blinks were not random and differed systematically between trial types. These difference are largely eliminated by artifact rejection.

As described at the beginning of the chapter, artifact rejection is designed to deal with three specific problems: reduced statistical power, systematic confounds, and sensory input problems. The artifact rejection procedure that you've performed has achieved the first two of these goals: you've slightly reduced the noise (as evidenced by the lower aSME values) and thereby increased the statistical power, and you've minimized a confound (differential blink activity between standards and deviants).

Figure 8.2.D shows the results with ICA-based correction of blink artifacts (which will be covered in the next chapter). You can see that this approach eliminated blink-related activity better if you look at the VEOG-bipolar channel, which largely isolates blink-related activity. This channel is nearly flat in the corrected data but not in the rejected data. However, the corrected and rejected waveforms look nearly identical at the FCz site—which is what we really care about—except that the rejected waveforms are noisier because we've lost some trials in the rejection process. This shows that rejection is working reasonably well: We're eliminating confounding activity from the blinks without a huge reduction in data quality. However, the waveforms appear to be cleaner for the corrected data (because we've retained all the epochs), and I confirmed this by computing SME values (which were 0.4992 for the deviants and 0.2815 for the standards). Because of this better data quality, I usually prefer ICA-based artifact correction instead of rejection for blinks. However, we still reject trials with blinks that occur near the time of the stimulus in visual experiments, because we want to exclude trials on which the participant could not see the stimulus.

An important take-home message of this exercise is that artifact rejection is designed to address three specific problems, and you want to choose the parameters that best solve these problems. You can assess data quality and statistical power by examining the SME values. You can assess confounds by looking for polarity inversions above versus below the eyes in the averaged ERP waveforms. It also helps to view the waveforms for averages of all epochs, averages of just the epochs with artifacts, and averages that exclude epochs with artifacts. Blinks and eye movements don't create obvious problems with the sensory input in most auditory paradigms, but we will see how they impact a visual paradigm near the end of the chapter.

This page titled 8.8: Exercise- Data Quality and Confounds is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.9: Exercise- Better Blink Detection

In a previous exercise, we saw that changing the threshold for rejection affects both the hit rate (the proportion of artifacts detected) and the false alarm rate (the proportion of non-artifact trials that are flagged for rejection). If you change the threshold to make one better, this inevitably makes the other one worse. However, there is something you can change that can improve both the hit rate and the false alarm rate. Specifically, you can use an artifact detection algorithm that is better designed to isolate blinks from other kinds of voltage deflections.

The simple voltage threshold algorithm that we have used so far in this chapter is an overly simplistic way of detecting blinks. It treats any large voltage as a blink, not taking into account the shape of the blink waveforms. As a result, it ends up flagging trials that don't contain blinks and misses some of the true blinks. We can improve blink detection by taking into account the fact that blinks are relatively short-term changes in voltage that typically last ~200 ms. In this exercise, we'll look at two artifact detection algorithms that take this into account and work much better for blink detection.

The first is called the *moving window peak-to-peak* algorithm, and it's illustrated in Figure 8.3. With its default parameters and our epochs of -200 to 800 ms, this algorithm will start by finding the difference in amplitude between the most positive and most negative points (the *peak-to-peak* voltage) between -200 and 0 ms (a 200-ms window). Then, the window will move 100 ms to the right, and the algorithm will find the peak-to-peak voltage between -100 and 100 ms. The window will keep moving by 100 ms, finding the peak-to-peak amplitude from 0 to 200 ms, 100 to 300 ms, etc. It then finds the largest of these peak-to-peak amplitudes for a given epoch and compares that value to the rejection threshold.

Figure 8.3.A illustrates the application of this algorithm to a trial with a blink. You get a large peak-to-peak amplitude during the period of the blink because of the relatively sudden change in voltage. However, the algorithm isn't "fooled" by the slow drift shown in Figure 8.3.B.

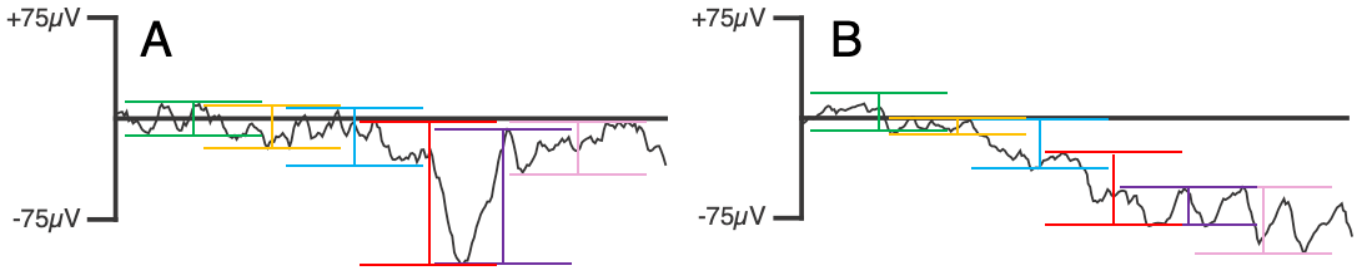


Figure 8.3. Moving window peak-to-peak algorithm. The peak-to-peak amplitude is determined in each window, and then the maximum of these values for a given epoch is compared with the rejection threshold.

Let's try it. Go back to **1_MMN_preprocessed_interp_be** as the active dataset and select **EEGLAB > ERPLAB > Artifact detection in epoched data > Moving window peak-to-peak threshold.** Set the window width to 200 ms and the window step to 100 ms (so that we get a 200-ms window every 100 ms). Set the threshold to **100** and the channel to **33** (VEOG-bipolar). Click **ACCEPT** to run the routine.

The first thing to note is that 28.2% of trials have been flagged for rejection. That's a lot more than we had with the absolute voltage threshold; we'll discuss the reasons for that in a moment.

If you scroll through the data, you'll see that every clear blink has now been flagged (including Epochs 104 and 170, which were missed by the absolute threshold algorithm). However, more trials with muscle noise have now been flagged for rejection. Here's why: Imagine that the muscle noise causes the voltage to vary from -55 to +55 μV between 200 and 400 ms. This doesn't exceed the absolute threshold of ±100 μV, but it creates a peak-to-peak amplitude of 110 μV, exceeding our threshold for the peak-to-peak amplitude. One way to solve this would be to increase the threshold to something like 120 μV. However, this would cause us to start missing real blinks.

Another approach would be to apply a low-pass filter prior to artifact detection. Let's give that a try. Go back to **1_MMN_preprocessed_interp_be** as the active dataset and select **EEGLAB > ERPLAB > Artifact detection in epoched data > Moving window peak-to-peak threshold.** Keep the parameters the same, but check the box labeled **Low-pass prefiltering...**

and set the **Half-amplitude cutoff** to **30**. This option creates a hidden copy of the dataset, applies the filter to it, and applies the artifact detection algorithm to this hidden copy. The artifact detection flags are then copied to the original dataset. That way, you get the benefits of prefiltering the data in terms of flagging appropriate trials, but you end up with your original unfiltered data.

Click **Accept** to run the artifact detection routine. You'll see that only 13.0% of trials are flagged for rejection (compared to 28.2% without prefiltering). If you scroll through the data, you'll see that all the clear blinks are flagged, but the trials with EMG noise are not. If you check the data quality measures using **EEGLAB > ERPLAB > Compute data quality metrics (without averaging)**, you'll see that the aSME for FCz has improved slightly relative to the rejection based on the absolute voltage threshold. And if you plot the ERP waveforms, you'll see that they're quite similar to what we found with the absolute voltage threshold.

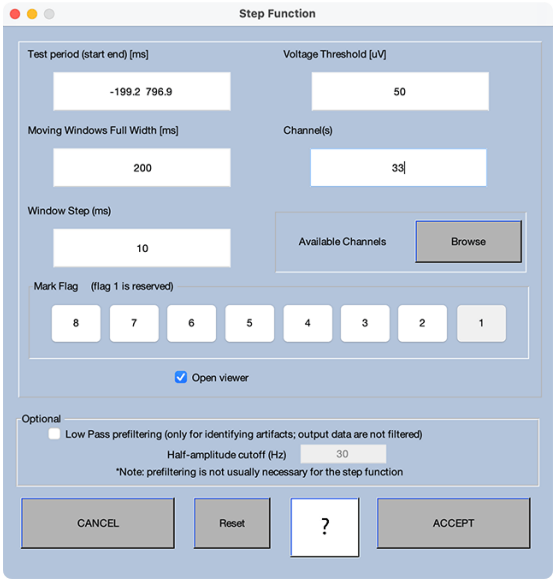
So, the moving window peak-to-peak algorithm is definitely superior to the absolute voltage threshold algorithm. It doesn't make a huge difference with this participant, but it makes a big difference for some participants and some experimental paradigms. However, in many cases, you'll want to use the low-pass prefilter option.

Now let's look at another algorithm that works quite well but doesn't require any low-pass filtering. I call this algorithm the *step function*, because I developed it to detect the step-shaped voltage deflections produced by saccadic eye movements in N2pc paradigms. I eventually discovered that it also works great for detecting blinks.

The step function also involves a moving window, with 200 ms as a reasonable default value for most studies. Within a 200-ms window, this algorithm calculates the difference between the mean voltage in the first half of the window and the mean voltage in the second half of the window. It then finds the largest of these differences for all the windows in a given epoch, and it compares the absolute value of this difference to the rejection threshold. For example, the window indicated by the red lines in Figure 8.3.A has an amplitude of approximately 20 µV during the first half and approximately 70 µV during the second half, so this would be a difference of approximately 50 µV. That wouldn't exceed a threshold of 100 µV, but you can use a lower threshold with the step function than with the other algorithms. Also, you will get the largest voltage from a blink if the center of the window is just slightly before the start of the blink, and a smaller step size (e.g., 10 ms) tends to be better.

Let's try it. Go back to **1_MMN_preprocessed_interp_be** as the active dataset and select **EEGLAB > ERPLAB > Artifact detection in epoched data > Step-like artifacts.** Set the parameters as shown in Screenshot 8.3. Specifically, set the window width to 200 ms and the window step to 10 ms (so that we get a 200-ms window every 10 ms). Set the threshold to 50 µV and the channel to 33. Click **ACCEPT** to run the routine.

Screenshot 8.3



You should first note that 13.1% of trials have been flagged for rejection, which is nearly identical to what we obtained when we used the moving window peak-to-peak algorithm with the prefiltering option. But note that no filtering is required with the step function: when the step function algorithm averages across each half of the 200-ms window, high-frequency activity is virtually eliminated.

If you scroll through the EEG, you'll see that the algorithm has successfully flagged all of the clear blinks without flagging many non-blink trials. In my experience, the step function works slightly better than the moving average peak-to-peak algorithm (especially when there is a lot of EMG noise) and significantly better than absolute voltage thresholds. It's what I recommend for detecting blinks in most cases.

This page titled 8.9: Exercise- Better Blink Detection is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.10: Exercise- Detecting Eye Movements

By this point, I hope you have a good idea of how to detect and reject blinks. In this exercise, we'll look at another common artifact, saccadic eye movements. Depending on the nature of the experiment, saccadic eye movements can be an enormous problem or largely irrelevant. How can you tell? The answer, as always, is to think about the three problems that artifact rejection is designed to solve. Because the MMN experiment uses auditory stimuli, changes in eye position won't directly impact the sensory input. And because participants were watching a silent movie at fixation during the experiment, there is no reason to suspect that the eyes will move in different directions for deviant versus standard trials (although this is something that we should verify rather than assume). If the eye movements have a random direction, then they will cancel out (because the polarity reverses for opposite directions) and are unlikely to be a confound. So, the main question is whether the eye movements add significant noise and decrease our statistical power. We can use the SME values to determine whether rejecting trials with eye movements helps us (because it reduces a source of noise) or hurts us (because it reduces the number of trials).

Subject #1 in the MMN experiment made no obvious eye movements while doing the task, so we're going to look at Subject #10 for this exercise. At this point, I'd recommend quitting and restarting EEGLAB. You can then load the dataset named **10_MMN_preprocessed.set**. Take a quick look to familiarize yourself with this participant's EEG. Ordinarily, you'd look carefully and think about whether to interpolate any channels, but here we'll just focus on the eye movements.

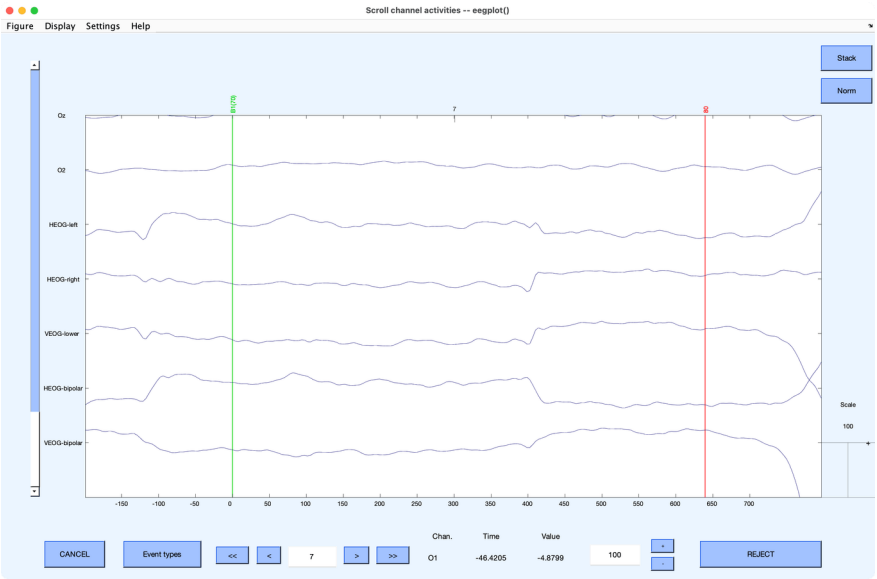
To make the eye movements easier to visualize, we're going to start by applying a low-pass filter to minimize high-frequency noise. Selected **EEGLAB > ERPLAB > Filter & Frequency Tools > Filters for EEG data** and apply a low-pass filter with a 30 Hz half-amplitude cutoff and a slope of 12 dB/octave. Save the resulting dataset as **10_MMN_preprocessed_filt**. Then epoch the data with **EEGLAB > ERPLAB > Extract bin-based epochs**, using a time range of **-200 800** ms and **Pre** as the baseline. Save the resulting dataset as **10_MMN_preprocessed_filt_be**. Now we're ready to look at the eye movements.

Why We're Filtering Now

Earlier, I said that filtering out high-frequency noise isn't necessary for the step function. So, why am I asking you to filter the data here? The answer is simple: At this point, I want you to learn what eye movements look like, and the filtering will help with that. The filtering will have virtually no impact on the operation of the step function algorithm.

Unless the participant is tracking moving objects or is walking around, most of the eye movements you'll see will be *saccades* (sudden shifts in gaze position). Saccades produce a sudden change from one voltage level to another in the EOG electrodes. Plot the EEG data and go to Epoch 7. It should look something like Screenshot 8.4. To see the eye movements more clearly, I've selected **Settings > Number of channels to display** in the plotting window and entered **6** as the number of channels, and I've told it to display only one epoch at a time with **Settings > Time range to display**.

Screenshot 8.4



You can see a classic saccadic eye movement pattern at 400 ms in Epoch 7. The voltage is fairly flat for a few hundred milliseconds, and then there is a sudden shift in the EOG channels, followed by a relatively flat signal at a different voltage level until a blink occurs near the end of the epoch. This is because in a saccadic eye movement, gaze is fixed at one location for a period of time, then moves rapidly, and then remains fixed at a new location for a period of time. At ~400 ms, the voltage goes more positive at HEOG-right and more negative at HEOG-left, indicating a rightward eye movement. But the voltage simultaneously shifts in the positive direction at VEOG-lower, meaning that the eye movement is actually angled downward as well.

This participant has a fairly large number of blinks, and we should deal with those before we assess the effects of the eye movements. To flag the blinks, select **EEGLAB > ERPLAB > Artifact detection in epoched data > Step-like artifacts** and set the window width to 200 ms, the window step to 10 ms, the threshold to 50 µV, and the channel to 33. Also, click the **2** button in the **Mark Flag** section (this will be explained a little later). Click **ACCEPT** to run the routine. If you scroll through the data, you'll see that the algorithm did a good job of flagging epochs with clear blinks. Save the dataset, naming it **10_MMN_preprocessed_filt_be_noblinks**.

Now we're going to flag trials with horizontal eye movements. In some tasks, the stimuli are presented to the left or right of fixation, so most of the eye movements are horizontal. In the MMN task, participants watched a silent movie, so the eye movements might be in any direction. Consequently, we would ordinarily want to detect both horizontal and vertical eye movements in this task. However, it will be easier to understand what's going on in this exercise if we just look for horizontal eye movements.

To flag the horizontal eye movements, start with the dataset you just created (**10_MMN_preprocessed_filt_be_noblinks**) and select **EEGLAB > ERPLAB > Artifact detection in epoched data > Step-like artifacts**. Keep the window width at 200 ms and the window step at 10 ms, but change the channel to 32 (HEOG-bipolar). You should also lower the threshold to 32 µV. I like to use multiples of 16 µV for horizontal eye movements because, for the average participant, each degree of eye rotation increases the HEOG voltage by 16 µV (Lins et al., 1993). A threshold of 32 should therefore detect eye movements of approximately 2° or larger.

You should also click the **3** button in the **Mark Flag** section. These flags are used to keep track of different types of artifacts. Flag 1 is always set for any artifact, but you can add other flags. When you detected blinks, you told it to set Flag 2. Here we're going to set Flag 3 for horizontal eye movements. This will allow us to get a separate count of the number of blinks and the number of horizontal eye movements.

Click **ACCEPT** to run the routine. Before you scroll through the EEG, take a look at the summary of artifacts in the Matlab command window, which should look something like this:

Bin	#(%) accepted	#(%) rejected	# F2	# F3	# F4	# F5	# F6	# F7	# F8
1	118(59.0)	82(41.0)	64	66	0	0	0	0	0
2	341(58.4)	243(41.6)	185	211	0	0	0	0	0
Total	459(58.5)	325(41.5)	249	277	0	0	0	0	0

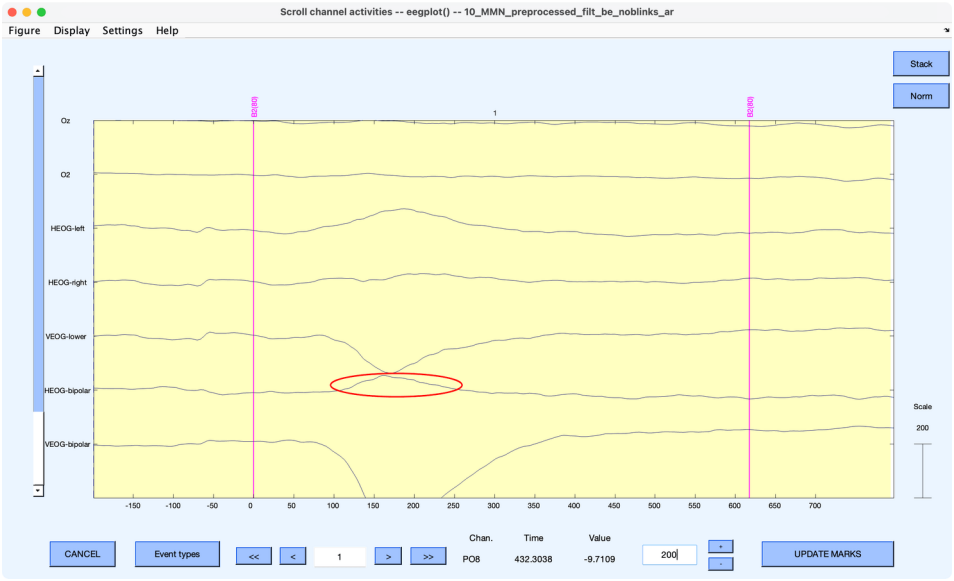
When we previously detected trials with blinks, 31.8% of epochs were flagged for rejection (collapsed across bins). That has now increased to 41.5% (270 epochs). The column labeled **#F2** shows the number of epochs with Flag 2 set (blinks), and the column labeled **#F3** shows the number of epochs with Flag 3 set (horizontal eye movements). In Bin 1, for example, 82 epochs were flagged for rejection overall, with 64 epochs with Flag 2 and 66 epochs with Flag 3. Of course, 64 + 66 is not equal to 82, because some trials were flagged for both blinks and eye movements. There were 18 trials flagged for eye movements that weren't flagged for blinks (because 82 total rejected epochs – 64 epochs flagged for blinks = 18 epochs flagged only for eye movements).

Too Many Blinks?

As described earlier, my lab excludes any participants with >25% rejected trials in our basic science experiments, so you might expect that this participant would have been excluded. However, we used artifact correction to deal with blinks in the analyses reported in the ERP CORE paper (Kappenman et al., 2021), so we did not need to exclude this participant.

Now take a look at the EEG and EOG data. (I recommend telling the plotting tool to display only 6 channels so that you can focus on the EOG data.) You'll see that the very first epoch was flagged for both blinks and eye movements (because both the VEOG-bipolar and HEOG-bipolar waveforms are drawn in red in Epoch 1). The HEOG signal is clearly not an eye movement—it doesn't show the sudden step from one voltage level to another voltage level that is produced by a saccadic eye movement. Instead, the blink that you can see in the VOEG-bipolar channel has "leaked" into the HEOG-bipolar channel (see the region marked with the red oval in Screenshot 8.5). In an ideal world, a blink would produce equal activity to the sides of the two eyes, and the HEOG-left minus HEOG-right subtraction would therefore eliminate the blink in the HEOG-bipolar channel. However, if one HEOG electrode was placed a little higher or lower than the other, the blink activity won't be identical at the left and right sites, and the subtraction won't completely eliminate the blink activity. This is one reason why a large number of epochs were flagged for both blinks and eye movements.

Screenshot 8.5



In Epoch 6, you'll see a clear eye movement (a step-like voltage change) in the HEOG-bipolar channel, which was correctly flagged. The eye movement must have been diagonal, because it also created a step-like deflection at the same time in the VEOG-bipolar channel. Only the HEOG-bipolar channel is drawn in red, however, because we applied the threshold of 32 µV only to the HEOG-bipolar channel, and the deflection in the VEOG-bipolar channel was not big enough to be detected when we looked

for blinks with a threshold of 50 μV.

If you scroll through the whole dataset, you'll see that many horizontal eye movements were successfully flagged, but others were missed (e.g., Epochs 3, 67, 190, 389). Let's reduce the threshold and see whether we can detect those artifacts. First, save the current dataset, naming it **10_MMN_preprocessed_filt_be_noblinks_HEOG32**. You'll need it for the next exercise.

Now repeat the artifact detection process, keeping **10_MMN_preprocessed_filt_be_noblinks_HEOG32** as the active dataset so that we can add to the previous detections. When you launch **Artifact detection in epoched data > Step-like artifacts**, reduce the threshold to **16** and select the **4** button instead of the **3** button in the **Mark Flag** section. Click **ACCEPT** to run the routine, and then take a look at the summary of artifacts in the Matlab command window, which should look something like this:

Bin	#(%) accepted	#(%) rejected	# F2	# F3	# F4	# F5	# F6	# F7	# F8
1	71(35.5)	129(64.5)	64	66	128	0	0	0	0
2	183(31.3)	401(68.7)	185	211	394	0	0	0	0
Total	254(32.4)	530(67.6)	249	277	522	0	0	0	0

You can see that we've now rejected a lot more trials than before (67.6% of the total). The **#F3** column shows how many trials were flagged with the previous threshold of 32 μV, and the **#F4** column shows how many trials were flagged with the new threshold of 16 μV. If you scroll through the data, you'll see that almost all trials with a clear horizontal eye movement are now flagged for rejection.

But do we really want to use this threshold, even though it means that we'd be rejecting over 2/3 of the trials? We'll consider how to answer that question in the next exercise. But first, save this dataset as **10_MMN_preprocessed_filt_be_noblinks_HEOG16**.

This page titled 8.10: Exercise- Detecting Eye Movements is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

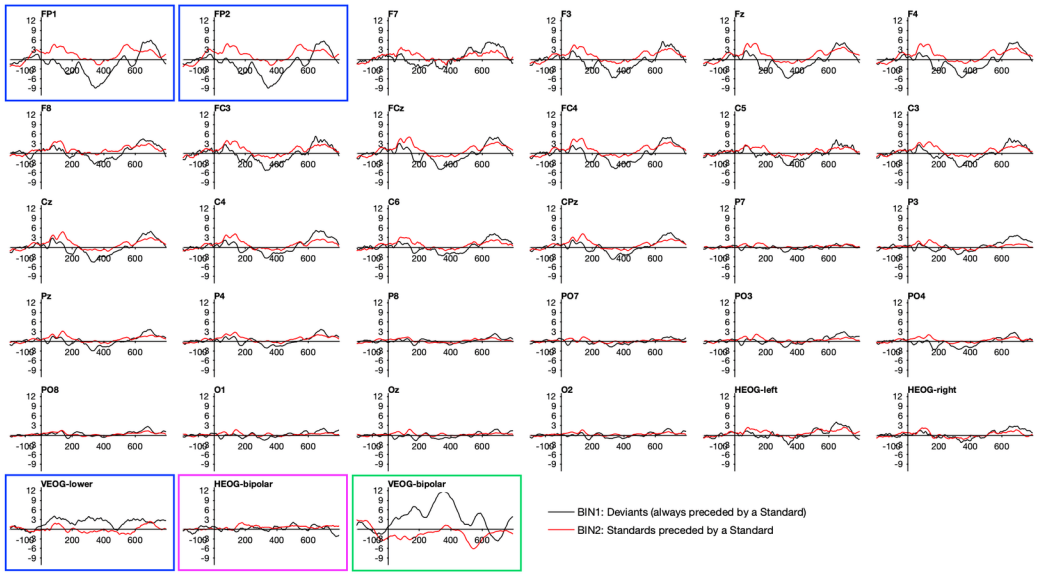
8.11: Exercise- Deciding on a Threshold for Eye Movements

To decide on the threshold for rejecting trials with eye movements, we need to return to our goals for artifact rejection. What threshold maximizes our data quality while avoiding confounds in our data?

Let's first consider whether horizontal eye movements are a confound in this experiment. Specifically, might horizontal eye movements cause different voltages at our FCz electrode site on deviant trials relative to standard trials? This is unlikely for two reasons. First, because FCz is on the midline, it should be near the line of zero voltage between the positive and negative sides of the voltage field produced by horizontal eye movements. Second, there is no reason to suspect that the frequency of leftward versus rightward eye movements would differ between deviants and standards. However, this is just an assumption, and we should check to make sure.

We can assess this assumption by looking at the ERP waveforms without any artifact rejection. To do this, select the dataset for Subject 10 that was created prior to any artifact detection (**10_MMN_preprocessed_filt_be**) and then select **EEGLAB > ERPLAB > Compute averaged ERPs**. If you plot the resulting ERP waveforms, you'll see a large voltage deflection for the deviants in the VEOG-bipolar channel (indicated by the green box in Screenshot 8.6). You can also see that this voltage is opposite in polarity below the eyes (VEOG-lower) versus above the eyes (Fp1 and Fp2; see the blue boxes in Screenshot 8.6). This indicates that this participant blinked more following the deviants than following the standards, just as we saw for Subject 1 (see Figure 8.2.A). However, unlike Subject 1, Subject 10 showed this pattern even during the MMN time window, so blinks could confound the MMN effects for this participant.

Screenshot 8.6



Now look at the HEOG-bipolar channel (indicated by the magenta box in Screenshot 8.6). The differences between deviants and standards in that channel are not any larger than the noise deflections in the prestimulus baseline period. This tells us that we don't have to worry about differences between deviants and standards in the frequency of leftward versus rightward eye movements, confirming our assumption. This means that we mainly need to be concerned about whether the eye movements are a source of noise, not a confound.

To assess noise, we can ask how the artifact rejection impacted the SME values. Specifically, we'll look at the SME values after rejecting only trials containing blinks, rejecting trials with blinks and eye movements using a 32 µV eye movement threshold, and rejecting trials with blinks and eye movements using a 16 µV eye movement threshold.

To start, let's get the SME values after rejecting trials with blinks but before rejecting trials with eye movements. Make the dataset named **10_MMN_preprocessed_filt_be_noblinks** active, and **EEGLAB > ERPLAB > Compute data quality metrics (without**

averaging). In the **Data Quality** section of the GUI, select **Custom parameters**, click the **Set DQ options...** button, and create a custom time range of 125-225 ms. Make sure that the main GUI is set to exclude epochs marked during artifact detection, and then click **RUN**. In the data quality table that appears, look at the aSME values for Bin 1 and Bin 2 from the FCz channel in the 125-225 ms time range. These values will be our reference points for asking whether rejecting trials with eye movements makes the data quality better (because of less random variation in voltage) or worse (because of a reduction in the number of epochs being averaged together). Keep the data quality window open so that you can refer to it later.

Now repeat this process with the datasets in which eye movements were flagged for rejection using a threshold of 32 μV (**10_MMN_preprocessed_filt_be_noblinks_HEOG32**) and 16 μV (**10_MMN_preprocessed_filt_be_noblinks_HEOG16**). Compare the resulting data quality tables with the data quality table you obtained without rejection, focusing on the aSME values for Bin 1 and Bin 2 from the FCz channel in the 125-225 ms time range.

These values are summarized in Table 8.1. You can see that the data quality was slightly reduced (i.e., the aSME was increased) when large eye movements were rejected by means of the 32 μV threshold and substantially reduced when virtually all eye movements were rejected by means of the 16 μV threshold. Given that the previous analyses indicated that horizontal eye movements were not a confound, the 16 μV threshold appears to be taking us farther from the truth rather than closer to the truth (because it impairs our ability to precisely measure MMN amplitude). The 32 μV threshold decreases the data quality only slightly (probably because there is some benefit of reduced noise but some cost of a smaller number of trials). I would be inclined to go with this 32 μV threshold (instead of not excluding trials with horizontal eye movements), even though it slightly reduces the data quality, just in case there is some small confounding effect of large eye movements that wasn't obvious.

Table 8.1. Effects of eye movement rejection on data quality and percentage of rejected trials.

Rejection	aSME for Deviants	aSME for Standards	% Rejected
Blinks Only	0. 8264	0.5353	31.8%
Blinks + Eye Movements (32 μV)	0. 8335	0.5874	41.5%
Blinks + Eye Movements (16 μV)	0.9901	0.7437	67.6%

Viewing a Summary of Artifacts

Table 1 shows the percentage of rejected trials. This information was printed to the Matlab Command Window when the data quality metrics were computed (and are also printed when you average). If you want to see this information for a given dataset at a later time, select the relevant dataset and then select **EEGLAB > ERPLAB > Summarize artifact detection > Summarize EEG artifacts in a table**. You'll then be asked where you want to save the summary. I usually choose Show at Command Window.

As you have seen, there is some subjectivity involved with artifact rejection. In my experience, a well-trained researcher can meaningfully increase the final data quality and avoid confounds by carefully setting the artifact detection parameters individually for each participant in this manner. It takes some time, but you will get much faster as you gain experience. The two participants we've examined so far in this chapter are particularly challenging cases that require some careful thought and analysis, but most of the participants in this study were much more straightforward. I find that we can use a standard set of detection parameters in about 80% of participants in my lab's basic science experiments, and it takes only 5–10 minutes to verify that everything is working fine in these participants.

Beyond the time investment, it's also important to consider whether customizing the artifact rejection for each participant might lead to some kind of bias in the results. Most basic science ERP studies involve within-subjects manipulations, in which the same

artifact detection parameters are used for all conditions for a given participant. Because the parameters are identical across conditions, there is little opportunity for bias. In theory, the experimenter could try many different artifact detection parameters for a given participant and then choose the parameters that produce the desired effect. But this is obviously cheating. If someone wants to cheat, there are much easier ways to do it, so I don't worry much about this possibility. To avoid unconscious bias, you should avoid looking at the experimental effects in the averaged ERP waveforms when you're setting the parameters (although you may need to look at the averaged EOG waveforms to assess the presence of systematic differences in artifacts between conditions).

My advice is different for research that focuses on comparing different groups of participants (e.g., a patient group and a control group). In these studies, the main comparisons are between participants, and now we may have different artifact detection parameters for our different groups. This could lead to unintentional biases in the results. To minimize any biases, I recommend that the person setting the artifact detection parameters for the individual participants should be blind to group membership. For example, in my lab's research on schizophrenia, the person setting the artifact detection parameters is blind to whether a given participant is in the schizophrenia group or the control groups. That's a bit of a pain, but it's worth it to avoid biasing the results. Note that some subjectivity also arises in artifact correction (e.g., choosing which ICA components to eliminate), so the person doing the correction should also be blind to group membership.

When to set artifact detection parameters (and how to avoid a catastrophe)

Imagine that you spend 9 months collecting data for an ERP study, and at the end you realized that there was a major problem with the data that prevented you from answering the question that the study was designed to answer. Your heart would start racing. Your face would become flushed. You would feel like vomiting. And you would want to crawl into a hole and never come out.

Would you like to avoid that situation? If so, then here's an important piece of advice: Do the initial processing of each participant's data within 48 hours of the recording session. This includes every step through averaging the data and examining the averaged ERPs. Of course, this includes setting the artifact detection parameters. And it includes quantifying the behavioral effects (which is the step that people most frequently forget).

If you don't do this, there is a very good chance that there will be a problem with your event codes, or with artifacts, or with something unique to your experiment that I can't anticipate, and that this problem will make it impossible for you to analyze your data at the end of the study. I have seen this happen many, many, many times. Many times!

You can catch a lot of these problems by doing a thorough analysis of the first participant's data before you run any additional participants. And in my lab, we have a firm rule that experimenters aren't even allowed to schedule the second participant until we've done a full analysis of the first participant's data. I estimate that we catch a problem about 80% of the time when we analyze the first participant's data. So you absolutely must do this.

However, some problems don't become apparent until the 5th or the 15th participant. And sometimes a new problem arises midway through the study. For this reason, you really must analyze the data from each participant within a couple days.

There is another side benefit to this: You won't be in the position of needing to set the parameters for 30 participants in a single two-day marathon preprocessing session. Not only would these be two of the most dreariest days of your life, it would be difficult for you to pay close attention and do a good job of setting the parameters. The task of setting the parameters is best distributed over time.

This page titled 8.11: Exercise- Deciding on a Threshold for Eye Movements is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.12: Exercise- Commonly Recorded Artifactual Potentials (C.R.A.P.)

By this point, I hope you have a good idea of how to detect and reject blinks and eye movements. Now we'll shift to a class of miscellaneous artifacts that I like to call Commonly Recorded Artifactual Potentials (C.R.A.P.). Whereas virtually all participants in all experiments exhibit blinks and eye movements, which have a predictable waveshape and scalp distribution, C.R.A.P. artifacts tend to be idiosyncratic in their waveshape and scalp distribution. This category includes EMG bursts, movement artifacts, skin potentials, and sudden voltage changes of unknown origin.

In the ERP CORE experiments, our approach to C.R.A.P. was to apply both a simple voltage threshold and the moving window peak-to-peak algorithm to all EEG channels. The simple voltage threshold is particularly good for detecting large drifts in voltage, and a threshold of ±200 μV works well for most participants. The moving window peak-to-peak algorithm is good for detecting muscle bursts and idiosyncratic artifacts that involve sudden voltage changes. The threshold for this routine needs to be adjusted for each participant to ensure that large artifacts are detected without throwing out too many epochs. In most participants, we used a threshold between 100 and 150 μV.

Let's try the absolute voltage threshold of ±200 μV with Subject 10 from the MMN experiment. Make the dataset named **10_MMN_preprocessed_filt_be_noblinks** active, and select **EEGLAB > ERPLAB > Artifact detection in epoched data > Simple voltage thresholds**. Set the channels to **3:28**, the voltage limits to **-200 200**, and the Flag **5** button. We're leaving out Fp1, Fp2, and the EOG channels so that we don't end up flagging blinks. (I would ordinarily include Fp1 and Fp2, but it will be easier to see how the artifact detection is working in this exercise without them.) Click **ACCEPT** to run the routine.

Now look at the artifact table in the Matlab command window. You'll see that only 17 epochs were flagged by this new artifact test (see the **#F5** column in the table). We often catch a relatively small number of artifacts with this test, but that's okay. Our goal here is to get rid of rare but large artifacts.

Now scroll through the EEG. All of the epochs that were previously flagged for blinks are still flagged, but you'll also see occasional epochs that are flagged from our new absolute threshold test. The blinks are indicated by a red waveform for the VEOG-bipolar channel, and the new artifacts are indicated by a red waveform for any of the channels between 3 and 28. For example, Epoch 44 was flagged because of large voltage changes in the F7 and F8 channels. Epochs 43, 63, and 64 were also flagged because of the F7 channel. If you keep scrolling, you'll see a lot of large, sudden voltage changes in F7 (and to a lesser extent F8). But not all of these sudden voltage changes were large enough to be flagged. There are also some fairly large artifacts of this nature in PO4 that were too small to be flagged (but are still quite large). Our threshold of ±200 μV is designed to flag only very large artifacts.

I don't know what caused these artifacts in F7, F8, and PO4. They're not biological in origin: With the exception of saccadic eye movements, I don't know of any biological signals that look like this, changing suddenly from one voltage level to another. However, they're about 100 times larger than the MMN, so they seem like a significant source of noise that will degrade our data quality (i.e., increase the SME).

To verify this, go ahead and check the aSME values, using a custom window of 125-225 ms as before. As shown in Table 8.2, the aSME values for the F7 channel were improved by removing the C.R.A.P. artifacts. Only 17 trials were rejected, but the artifacts were quite large on those trials, so the rejection produced a noticeable improvement in the aSME values. Our main concern, however, is the FCz channel, which will be used to quantify the MMN in our main analyses. Because the artifacts detected by the absolute threshold test were not present in FCz, rejecting those 17 trials reduced the data quality in this channel, but only slightly.

Now let's add a moving window peak-to-peak amplitude test to get rid of more C.R.A.P. epochs. Save the dataset created with the absolute threshold test as **10_MMN_preprocessed_filt_be_noblinks_CRAP200**. We're going to add onto the flags in this dataset, so make sure it's the active dataset. Select **EEGLAB > ERPLAB > Artifact detection in epoched data > Moving window peak-to-peak threshold**, set the threshold to **125**, set the channels to **3:28**, set the window length to **200**, set the window step to **100**, and select the Flag **6** button. Note that the choice of 125 μV as the threshold was somewhat arbitrary. It's just a reasonable starting point that works well for the basic science studies in my lab.

If you look at the artifact table in the Matlab command window, you'll see that 77 epochs were flagged by this test. However, many of those epochs had previously been flagged for blinks and/or C.R.A.P., and the total number of flagged epochs has increased from 259 to 291. So, we've flagged an additional 32 epochs.

If you look at the EEG, you'll see that we've now flagged many trials with sudden voltage shifts that were missed before, including several in the PO4 channel. All of the epochs that were flagged certainly seemed to have pretty large artifacts in them. We're not flagging trials with "ordinary" EEG noise. If you check the data quality, you'll see that the aSME for the F7 channel has improved considerably, and we've produced only a small reduction of data quality at FCz (see Table 8.2).

I also looked at the effects of reducing the threshold from 125 μV to 100 μV. When I inspected the EEG, more trials with clear artifacts were flagged. The data quality for F7 also improved a bit, with only a small reduction in data quality at FCz (see Table 8.2). I tried decreasing the threshold to 75 and even 50 μV, and this resulted in slight improvements at F7 but further worsening at FCz.

Table 8.2. Effects of C.R.A.P. rejection on data quality.

Rejection	aSME at F7 for Deviants	aSME at F7 for Standards	aSME at FCz for Deviants	aSME at FCz for Standards
Blinks Only	1.9480	0.9115	0. 8264	0.5353
Blinks + Voltage Threshold (±200)	1.6381	0.9038	0.8345	0.5410
Blinks + Voltage Threshold (±200) + Moving Window (125)	1.1711	0.7890	0.8443	0.5493
Blinks + Voltage Threshold (±200) + Moving Window (100)	1.1158	0.7368	0.8504	0.5449

So, what threshold is best? Compared to rejection only of blinks, rejecting C.R.A.P. with an absolute threshold of ±200 caused a fairly substantial drop in aSME for F7, and adding the moving window peak-to-peak test with a threshold of 125 caused another large drop in aSME at F7 (and other channels, as well). This rejection of C.R.A.P. caused the aSME at FCz to rise only slightly. Dropping the threshold further caused only minor decreases at F7, along with small increases at FCz. Given that FCz will be the main channel used for measuring the MMN, with the other channels only being used to quantify the scalp topography, the threshold of 125 μV seems like a good compromise for this particular experiment.

The optimal threshold, and even the optimal set of test to apply, will vary across participants and across experimental paradigms. The key is to ask what set of parameters will be most likely to bring you to a true conclusion about your primary scientific hypothesis. In almost all experiments, this means rejecting artifacts to the extent that they degrade your data quality at the key electrode sites (which you can assess with aSME values) and ensuring that any remaining artifacts are not producing differences between conditions.

A third issue—which does not typically arise in auditory experiments like our MMN experiment—is whether blinks and eye movements produced changes in the sensory input. We will turn to that issue in a later set of exercises.

This page titled 8.12: Exercise- Commonly Recorded Artifactual Potentials (C.R.A.P.) is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.13: Using Artifact Detection to Avoid Changes to Visual Inputs

Up to this point, we've focused on the first two of the three problems associated with artifacts, namely noise and systematic differences in voltage across conditions. The remainder of the chapter will focus on the third problem, namely that blinks and eye movements can change the sensory input when visual stimuli are used.

For most researchers, this is not a big issue. Once you've dealt with the noise produced by blinks and eye movements (either by rejection or correction), problematic changes in sensory input are relatively rare. If you're using artifact correction for blinks, you should still reject any trials with a blink that occurs at the time of the stimulus, because these trials are obviously not valid (see the chapter on artifact correction for details). And if the stimuli are presented in the middle of the display, participants won't make a lot of eye movements, and if they do, they probably won't differ systematically across conditions. This is what we saw in the MMN experiment. Some participants made eye movements as they watched the silent movie in the middle of the display, but these eye movements didn't vary across conditions, and they couldn't impact the sensory processing of the main stimuli (the auditory tones). They were a source of noise, but not a confound.

However, eye movements can be a significant systematic confound in some types of studies, mainly those using peripheral visual stimuli. For example, consider the spatial cuing paradigm shown in Figure 8.4, in which an arrow is used to indicate the likely location of a subsequent target. Many studies have used this paradigm to determine whether covert shifts of visual attention to the cued location cause the P1 and N1 waves to be larger when the stimulus is presented at the cued location compared to the uncued location (e.g., Eimer, 1994; Luck et al., 1994; Mangun & Hillyard, 1991). However, participants are likely to shift their gaze toward the cued location in these studies. If that happens, the target will appear in the center of gaze when it is presented at the cued location, whereas it will appear in the periphery when it is presented at an uncued location. We know that foveal stimuli produce larger sensory responses than peripheral stimuli, so this difference in the retinal location of the stimuli is a major confound that must be avoided in these studies. To avoid this confound, we can reject trials with eye movements. However, this is more difficult than it seems, even if you're using a high-quality eye tracker. The next exercise will explain how to do this effectively.

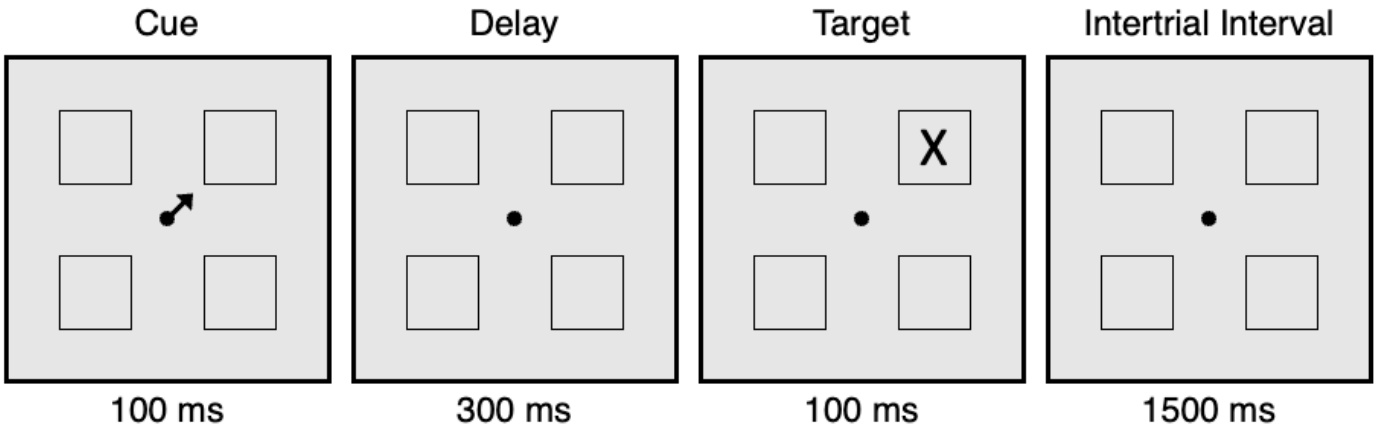


Figure 8.4. Prototypical spatial cuing paradigm. The cue indicates the likely location of the target. After a short delay, the target appears at the cued location (80% of trials) or one of the uncued locations (20% of trials). Participants press one of two buttons, as quickly as possible, to indicate whether the target is an X or an O. Participants are also instructed to maintain fixation on the central point and focus their "covert" attention onto the cued location. The goal is to determine whether sensory processing is enhanced at the cued location relative to the uncued location.

Small eye movements are also a problem in studies that look at lateralized visual ERP components, such as the N2pc component and contralateral delay activity (CDA). Both of these components are negative voltages contralateral to the location of a to-be-perceived or to-be-remembered object or set of objects. There are two specific problems that arise in these experiments. First, if the eyes move to the relevant location, then this location is now foveal, and that may impact the lateralization that would otherwise be observed. This problem is especially acute if the stimulus is presented for more than 200 ms, which is the approximate amount of time required to make a controlled eye movement in these paradigms. With long stimulus durations, you may have one period of time in which the relevant stimuli are lateralized (prior to the eye movement) and then another period in which they are foveal (after the eye movement). Even with brief stimulus durations, however, changes in eye position could potentially change the

lateralization of processing after the stimulus has disappeared (because the brain may remap the prior location of the internal neural representation onto its new retinal location).

The second problem is not due to the change in the sensory per se but is instead a confound in the EOG voltage. If participants tend to look leftward when the relevant stimuli are on the left side and rightward when the relevant stimuli are on the right side, then the EOG will be negative on the right side of the head when the relevant stimuli are on the left side and negative on the left side of the head when the relevant stimuli are on the right side. In other words, the EOG will appear as a negative voltage over the hemisphere contralateral to the relevant information, just like the N2pc and CDA. Moreover, the EOG is so large that even a small eye movement in the direction of the relevant information can produce a contralateral negativity that is as large or larger than the N2pc and CDA. The next exercise describes how to address both of these problems.

In theory, eye movements can also be a confound in studies of lateralized motor responses, such as the lateralized readiness potential (LRP; a negative voltage over the hemisphere contralateral to the response hand). This is because participants may make a small, unconscious eye movement toward the hand that responds. Such eye movements would produce a negative voltage over the right hemisphere for a left-hand response and a negative voltage over the left hemisphere for a right-hand response. That's the same pattern as the LRP. The strategy described in the following sections for eliminating these small eye movements for the N2pc and CDA can also be used for the LRP.

If you don't use lateralized visual stimuli or look at the LRP, then you can probably skip the rest of the chapter. However, you might want to read it and do the exercises anyway, because they provide good examples of the general principles of artifact rejection.

This page titled 8.13: Using Artifact Detection to Avoid Changes to Visual Inputs is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.14: The ERP CORE N2pc Experiment

The following exercise will focus on data from the ERP CORE N2pc experiment. This section will describe the experimental design and event code scheme (see Kappenman et al., 2021 for a detailed description). Example stimuli are shown in Figure 8.5.A. N2pc experiments are typically designed to study how people focus their covert attention onto peripheral stimuli without looking at the stimuli, so participants in this experiment were instructed to maintain gaze on the central fixation point at all times.

The experiment was divided into 8 blocks of 40 trials, and participants were instructed to attend to pink in half the blocks and blue in the other half. The task was to find the square of the attended color (using peripheral vision) and press one of two buttons to indicate whether this square had a gap on the top or on the bottom. Each stimulus array had 1 pink square and 11 black squares on one side, along with 1 blue square and 11 black squares on the other side. Except for these constraints, the stimulus locations varied randomly from trial to trial. Most importantly, pink and blue were always on opposite sides, but we randomized which side contained pink and which side contained blue on each trial. Consequently, participants could not know where to shift attention until a given display appeared.

The N2pc component reflects the focusing of attention onto a visual object and is largest when this object is surrounded by nearby distractors (see review by Luck, 2012). It's a negative voltage in the N2 latency range (typically from 200-300 ms) at posterior electrode sites, and its distinguishing characteristic is that the negative voltage is larger over the hemisphere contralateral to the attended object. For example, the voltage for left-hemisphere electrodes will typically be more negative when the target is in the right visual field than when it is in the left visual field, whereas the voltage for right-hemisphere electrodes will typically be more negative when the target is in the left visual field than when it is in the right visual field.

To make the N2pc easier to visualize, we typically collapse the data into a contralateral waveform (left hemisphere when the target is on the right averaged with right hemisphere when the target is on the left) and an ipsilateral waveform (left hemisphere when the target is on the left averaged with right hemisphere when the target is on the right). These collapsed waveforms are shown in Figure 8.5.B. You can see that the voltage is more negative (less positive) from ~200-300 ms in the contralateral waveform than in the ipsilateral waveform. This voltage is summed with the other ERP components that are active at the same time, which are mainly positive. Thus, the overall voltage is typically positive in both the contralateral and ipsilateral waveforms, but more negative for contralateral than for ipsilateral. To isolate the N2pc from these other components, we make a contralateral-minus-ipsilateral difference wave, as shown in Figure 8.5.C. This difference wave subtracts away all the nonlateralized components, making it easier to see and quantify the time course of the N2pc component.

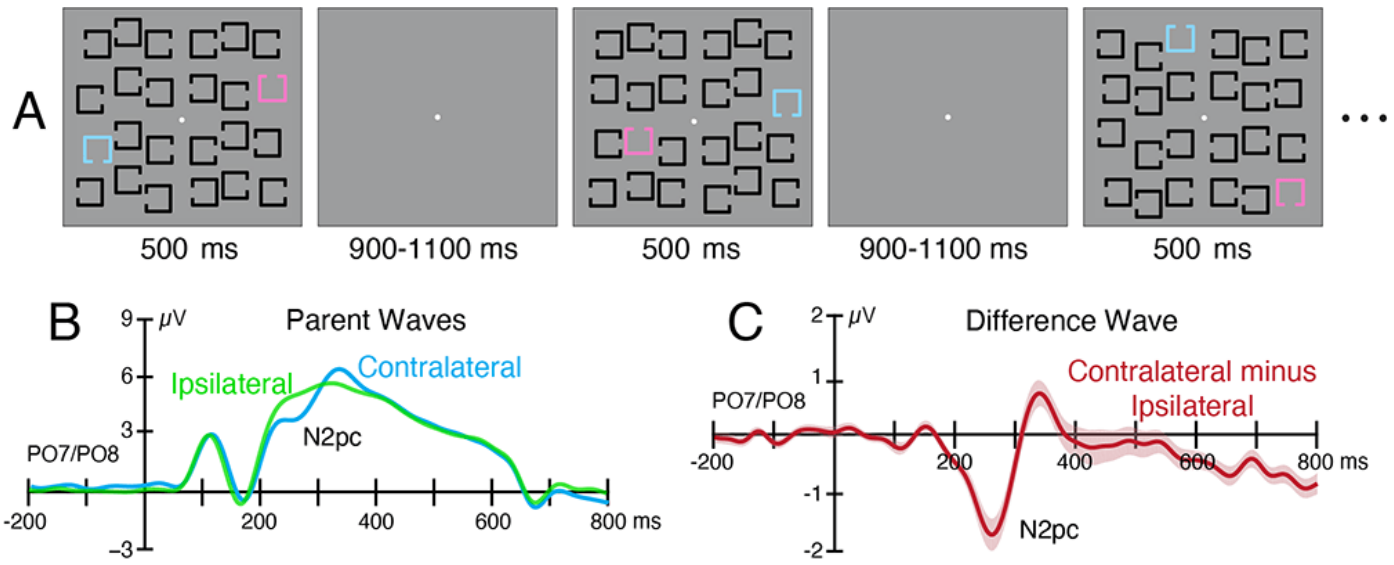


Figure 8.5. ERP CORE N2pc task (A), grand average ERP waveforms (B), and grand average contralateral-minus-ipsilateral difference wave.

The event code scheme is summarized in Table 8.3. You can see that the event code for each stimulus indicates which color was relevant for the current trial block (in the 100s place), which side contained the target square on the current trial (10s place), and whether the gap on the target square was on the top or bottom of the square (1s place). There were also event codes for correct and incorrect responses.

Table 8.3. Event codes for the ERP CORE N2pc experiment.

	Relevant Color for Current Block	Target Side	Target Gap Location	Event Code
Stimuli	Blue	Left	Top	111
	Blue	Left	Bottom	112
	Blue	Right	Top	121
	Blue	Right	Bottom	122
	Pink	Left	Top	211
	Pink	Left	Bottom	212
	Pink	Right	Top	221
	Pink	Right	Bottom	222
	Accuracy			**Event Code**
Responses	correct			201
	incorrect			202

The bin descriptor file is located in the Chapter_8 folder (named **BDF_N2pc.txt**), and the bin descriptors are shown below. You can see that there are separate bins for left and right targets, irrespective of which color was relevant in the current block and whether the target gap was on the top or the bottom. However, we require a correct response between 200 and 1000 ms after the stimuli.

bin 1
Left Target
.{111;112;211;212}{t<200-1000>201}

bin 2
Right Target
.{121;122;221;222}{t<200-1000>201}

This page titled 8.14: The ERP CORE N2pc Experiment is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.15: Exercise- Visualizing the Eye Movements

Now that you know about the ERP CORE N2pc paradigm, let's take a look at the eye movements. As you can imagine, it's difficult to maintain gaze on the central fixation point and not look toward the target. But as described above, it's important to make sure that the N2pc results aren't impacted by eye movements, which will change the location of the target relative to the center of gaze and also create a negative voltage over the contralateral hemisphere.

In this exercise, we'll focus on the data from Subject 15. If you look in the Chapter_8 folder, you'll see two datasets from this participant, one containing the continuous data (**15_N2pc_ICA_preprocessed.set**) and one containing the epoched data (**15_N2pc_ICA_preprocessed_epoched.set**). To make this exercise simpler, ICA-based artifact correction was already applied to these datasets to eliminate blinks. However, the datasets include a VEOG-uncorrected channel that contains the original uncorrected VEOG-bipolar signal. We like to keep this uncorrected channel so that we can reject any trials on which the participant blinked during the time period of the stimulus and therefore could not see the stimulus. This particular participant never blinked during that time period, however, so we don't need to worry about rejecting those epochs in this exercise. I also applied several other preprocessing operations, including filtering (bandpass 0.1–30 Hz, 12 dB/octave). The EEG channels and the HEOG-left and HEOG-right channels have all been referenced to the average of P9 and P10, and there is also an HEOG-bipolar channel (HEOG-left minus HEOG-right).

Before we get started on the exercise, quit and restart EEGLAB, load the continuous dataset (**15_N2pc_ICA_preprocessed.set**), and scroll through the data. You should always scroll through the continuous data as the first step so that you know what's in the file (as we did in the video demonstration in the previous chapter). Subject 15 has some weird stuff in the F7 channel near the end of the session, but we won't worry about that for this exercise.

Now load the epoched dataset (**15_N2pc_ICA_preprocessed_epoched.set**) and scroll through the data. On my widescreen desktop monitor, I like to show 15 epochs per screen. And to focus on the eye movements, I like to display only 6 channels, with a vertical scale of 100 or 150. This participant did a good job of following the fixation instructions initially, but you'll start to see a fair number of eye movements beginning at Epoch 70.

If you look at the event codes and the polarity of the HEOG deflections, you'll see that most of the clear eye movements are toward the side containing the target. Remember that the HEOG-bipolar channel was calculated as HEOG-left minus HEOG-right, and the dipole is positive at the front of the eyes, so a leftward eye movement produces a positive deflection and a rightward eye movement produces a negative deflection. For example, Epoch 70 has a negative deflection, indicating a rightward eye movement. You can see that the time-locking event code for this epoch is labeled "B2(121)", indicating that the event code was 121 (attend-blue, target on the right, gap on the top) and was assigned to Bin 2 (right-side targets). This is important because it indicates that we had a rightward eye movement on a trial with a right-side target.

When participants are trying to maintain central fixation, the eyes will typically move away from the fixation point rapidly, stay in a new location for a few hundred milliseconds, and then "snap back" to the fixation point. This leads to a "boxcar" shape in the HEOG signal (a flat signal at one voltage level corresponding to the location of the fixation point, a sudden change to a different voltage level for 100–500 milliseconds, and then a return back to the original voltage level corresponding to the fixation point). You can see this pattern in Epoch 70. The voltage level changes suddenly at approximately 285 ms after the time-locking event and then back to the original level approximately 180 ms later. (You can see the latencies by hovering the mouse pointer over the relevant part of the waveform and looking at the **Time** value near the bottom of the plotting window.)

If you look at epoch 73, you'll see a voltage deviation in the HEOG-bipolar channel, but it isn't a saccadic eye movement. It doesn't have the classic boxcar shape. Instead, it's a little bit of blink voltage that has leaked through to the HEOG electrodes.

There's a small leftward (positive) eye movement at approximately 570 ms after the time-locking event in Epoch 78, which had a left-side target. Epoch 81, with a right-side target, has a somewhat complicated pattern that looks like a brief leftward movement followed by a clearer rightward movement. Epoch 82, also with a right-side target, has a clear rightward eye movement.

There's also a brief spike in voltage at the onset of the eye movement in Epoch 82. That's probably an EMG burst coming from the muscles that produce the eye movement. Ordinarily, the muscle contraction that produces an eye movement occurs briefly at the start of the eye movement (to overcome the inertia in eye position) but then becomes too small to see as the eyes maintain their new location (which requires very little muscle activity). The present data have been low-pass filtered, so this *spike potential* isn't very clear. Without the filtering, the spike potential is quite large in some participants. It's fairly localized to frontal electrode

sites, and it's easy to filter out, so I don't usually worry about it as an artifact. However, if you perform time-frequency analyses, it's easy to mistake this EMG burst for gamma-band EEG activity (Yuval-Greenberg et al., 2008), so be cautious when someone says they're seeing gamma-band EEG oscillations at frontal electrode sites.

If you keep scrolling through the data, you'll see quite a few eye movements (mainly in the direction of the target, and mainly starting around 200 ms after stimulus onset). We clearly have a lot of work to do to make sure that these eye movements don't confound our N2pc data!

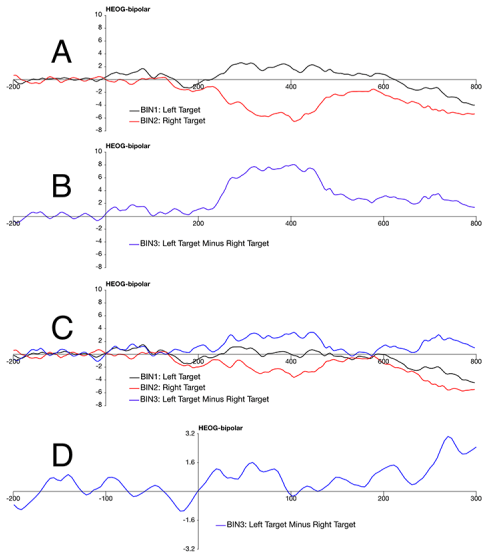
This page titled 8.15: Exercise- Visualizing the Eye Movements is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.16: Exercise- Using the Averaged HEOG to Visualize Consistent Eye Movements

In an N2pc (or CDA or LRP) experiment, we're mainly concerned about systematic eye movements in the direction of the target, not random eye movements. A good way to assess these systematic eye movements is to make one averaged HEOG waveform for all the left-target trials and another averaged waveform for all the right-target trials. That's what we'll do in this exercise, before we work on rejecting trials with artifacts.

To begin the exercise, make sure that **15_N2pc_ICA_preprocessed** is still loaded and active. Select **EEGLAB > ERPLAB > Compute averaged ERPs**, and run it with the default settings. You can name the resulting ERPset **15_N2pc_no_rejection**. Select **EEGLAB > ERPLAB > Plot ERPs > Plot ERP waveforms** to plot the data, but telling the routine to plot only Channel 32 (HEOG-bipolar). It should look something like Screenshot 8.7.A.

Screenshot 8.7



Starting just after 200 ms, you can see a positive voltage deviation (leftward eye movement) for the left-target trials and a negative voltage deviation (rightward eye movement) for the right-target trials. (There are also small differences between left- and right-target trials at earlier latencies, but they must be random noise because it takes at least 150 ms for the visual system to find a color-defined target in a visual search array and execute a saccade to it.)

The negative voltage in the right-target average is larger than the positive voltage for the left-target average. This is a fairly common pattern, and it may be related to the widely observed right hemifield bias for spatial attention. The main issue, however, is the overall difference in HEOG activity between left-target trials and right-target trials. To visualize this, we can make a left-target-minus-right-target difference wave. Select EEGLAB > ERPLAB > ERP Operations > ERP Bin Operations, and create an equation that reads **bin3 = bin1 - bin2 label Left Target Minus Right Target**. Make sure that the **Mode** is set to **Modify existing ERPset**, and then click **RUN**.

If you plot the resulting difference wave, you'll see that the difference between the left- and right-target averages is approximately 8 µV between approximately 250 and 450 ms (see Screenshot 8.7B). However, keep in mind that this reflects a mixture of some trials with a large eye movement and some trials without.

Is this a large enough difference to be problematic? To answer this question, we need to think about why horizontal eye movements are problematic for interpreting an N2pc experiment (or a CDA experiment, LRP experiment, etc.). First, we need to ask whether our N2pc effect might actually be HEOG voltage that has propagated from the eyes to the posterior electrodes where

the N2pc is ordinarily observed. Fortunately, a great paper by Lins et al. (1993) provides propagation factors for blinks, vertical eye movements, and horizontal eye movements. They don't provide values for the PO7 and PO8 electrodes, but they do provide values for the O1 and O2 electrodes and the P5 and P6 electrodes. The PO7 and PO8 electrodes that we used to measure N2pc amplitude in the ERP CORE paper are halfway between O1/P5 and O2/P6. Lins et al. reported a propagation of 1% from HEOG-bipolar to O1 and O2 and a propagation of 3% from HEOG-bipolar to P5 and P6. If we split the difference and assume a 2% propagation to PO7 and PO8, the 8 μV voltage deflection we observed at the HEOG channel in this participant would be expected to lead to a voltage of 0.16 μV at P5 and P6. That's quite a bit smaller than the N2pc, which is typically 1-2 μV, but still as big as some between-condition differences in N2pc amplitude. And in many participants, the HEOG-bipolar voltage would be quite a bit larger.

We also need to consider whether the eye movements caused the target to shift toward the center of the retina, reducing the lateralization that is necessary for seeing an N2pc (because it is defined by the difference between contralateral and ipsilateral channels). Again, the paper by Lins et al. (1993) provides useful information. Specifically, they found that the HEOG-bipolar signal increases by 16 μV for every degree of lateral eye rotation (which my lab has confirmed several times). Because the difference between left-target and right-target trials in the average HEOG-bipolar signal was approximately 8 μV, we can conclude that there was a 0.5° difference in eye rotation between left-target and right-target trials, on average. However, this is just an average. It's quite plausible that the target was fully foveated by ~250 ms on a substantial proportion of trials, with little or no eye movement on other trials. On the subset of trials with large eye movements, we can't be certain that a target on the left side of the video display was actually in the left visual field and that a target on the right side of the video display was actually in the right visual field.

If we want to be careful, we need to both decrease the artifactual EOG voltage produced by the eye movements and eliminate trials with large deviations in eye position. Artifact correction can be used to reduce the artifactual EOG voltage (as described in the next chapter), but we need to reject trials with large eye movements to deal with large deviations in eye position.

This page titled 8.16: Exercise- Using the Averaged HEOG to Visualize Consistent Eye Movements is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.17: Exercise- A Two-Stage Strategy for Eliminating Small But Consistent Eye Movements

If eye movements were either large or absent, rejection of trials with eye movements would be easy. However, people may make small but consistent eye movements toward the target. Small eye movements produce small HEOG deflections, and in practice it is difficult to detect eye movements smaller than approximately 1° (16 μV) on single trials in most participants. In this exercise, we'll see how to use a *two-stage procedure* (originally described by Woodman & Luck, 2003) to ensure that the data are not contaminated by small but consistent eye movements toward the target. In Stage 1, we throw out single trials with eye movements of greater than ~1°. In Stage 2, we look at the averaged HEOG-bipolar waveforms for left- and right-target trials to assess the effects of any small eye movements that remain after Stage 1.

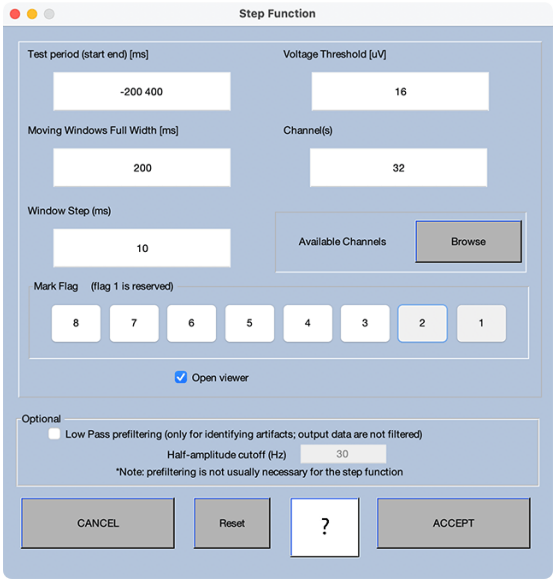
Let's start with Stage 1. If we're not thoughtful about the artifact rejection parameters used at this stage, we'll end up rejecting so many trials that we'll need to exclude the participant. One way to minimize the number of rejected trials is to look for eye movements only until the end of the N2pc measurement window. The N2pc measurement window in the ERP CORE N2pc experiment was 200-275 ms, but to keep things simple we'll assume a measurement window of 200-300 ms here. If the eyes move toward the target after 300 ms, this won't impact our N2pc amplitude measurements, so we won't reject trials with those late eye movements. This will give us more trials in our averages.

To get started on Stage 1, make sure that **15_N2pc_ICA_preprocessed_epoched** is loaded and active. Select **EEGLAB > ERPLAB > Artifact detection in epoched data > Step-like artifacts**, and set the parameters as shown in Screenshot 8.8. We're specifying a moving window of **200** ms, and the step function gives us the absolute value of the difference in mean voltage between the first and second halves of this window (the first 100 ms and the last 100 ms). We're specifying a Window Step of **10**, which means that we're shifting this window in 10-ms increments. We're specifying a test period of **-200 400** rather than the entire epoch so that we don't have to throw out trials with eye movements after the N2pc measurement window (200–300 ms).

Timing Details

The last moving window being tested will be from 200–400 ms, so an eye movement that starts at 300 ms will be detected. If we ended the test period at 300 ms, the last moving window would be from 100–300 ms, which corresponds to the difference in mean voltage between the 100–200 and 200–300 ms periods. An eye movement that began at, for example, 280 ms would only influence the voltage during the very last part of this period and would probably be missed. But this eye movement would be caught by a moving window from 180–380 ms. In general, if you are using a 200 ms moving window (which works well for eye movements), the window should end 100 ms after the time period of interest.

Screenshot 8.8



We've specified a threshold of **16** µV, which means that we will detect eye movements that are ~1° or larger. I like this threshold, because 1° is a nice round number, and this threshold works reasonably well with most adult participants (a higher threshold is needed for participants with noisy HEOG-bipolar signals to avoid rejecting too many trials that don't actually have eye movements).

Go ahead and click **ACCEPT** to run the routine. As usual, the first thing to look at is the percentage of rejected trials. A total of 12% were rejected, which is very reasonable in terms of not reducing the signal-to-noise ratio very much. Now you should scroll through the data. (As before, I recommend displaying only the bottom 6 channels, with a vertical scale of 100 or 150.) You'll see that the first clear eye movement, in Epoch 70, has been flagged for rejection. The blink that leaked through to the HEOG-bipolar channel in Epoch 73 was outside our window of -200 to 400 ms and was therefore not flagged, which is good. The eye movement in Epoch 78 was also outside our window and was not flagged. That's also good, because the eye movement is too late to impact our N2pc measurement, so we want to keep this trial.

If you keep scrolling, you'll see that the eye movements in Epochs 81, 86, and 88 were flagged. There's a leftward (positive) eye movement in Epoch 84 that wasn't flagged, but it was after our rejection window, so that's good. If you go through the whole session, you'll see that the step function did an excellent job of flagging clear eye movements that occurred during or prior to the 200–300 ms time period that we plan to use to measure N2pc amplitude.

Now let's see what rejecting these epochs will do to the data quality. Get the table of data quality values for the data prior to flagging the artifacts and after flagging the artifacts. If you look at the aSME values for the PO7 channel in Bin 1 for the 200-300 ms time period, you'll see that the aSME increased only slightly from the original data (0.4504) to the data excluding the marked epochs (0.4693). So, we've eliminated eye movements that exceeded ~1° of eye rotation without much decline in data quality. That's good!

Now we need to implement Stage 2 of our two-stage process. Stage 2 is designed to deal with the fact that we probably failed to reject a substantial number of smaller eye movements that were directed toward the target side. These small eye movements would create a negative voltage over the contralateral hemisphere that might impact our N2pc measurements. They might also change the lateralization of the target for trials on which the target was very close to the fixation point. To assess the possibility of small but consistent eye movements remaining in the data, we need to look at the averaged HEOG-bipolar waveforms for the left-target and right-target trials.

To do this, run the averaging routine, making sure that it's set to exclude epochs marked for rejection. Then make a left-target-minus-right-target difference wave using **ERP Bin Operations**. Now plot the ERPs at Channel 32 (HEOG-bipolar) for all three bins. It should look something like Screenshot 8.7C. Note that the EOG deflection is much smaller now than it was before we rejected trials with eye movements (Screenshots 8.5.A and 8.5.B), especially during the N2pc measurement window (200-300 ms).

But there is still some consistent eye movement activity in the direction of the target during this time window (i.e., the voltage is more negative on right-target trials than on left-target trials). We could try to eliminate this residual eye movement activity by decreasing our rejection threshold. However, in my experience, you can never complete remove this activity in most participants without rejecting a huge proportion of trials. For example, I tried reducing the threshold to 8 μV with the present participant, but the residual HEOG signal was quite large even though 55.4% of trials were rejected. Not surprisingly, this also increased the aSME value quite a bit. So, unless you are using a high-resolution eye tracker, you'll always have some residual HEOG activity after artifact rejection in most participants in experiments with lateralized targets.

The question then becomes, how much residual HEOG activity can we tolerate? If we think of this question in terms of the goals described at the beginning of the chapter, we can break it into two sub-questions: 1) Is enough of the residual HEOG activity being propagated to the N2pc measurement electrodes to create a significant confound? 2) Is the amount of eye rotation implied by the residual HEOG activity large enough to create a significant change in the sensory input?

For my lab's basic science experiments, we can afford to be extremely conservative in our answers to these questions. Our threshold for "good enough" in these experiments is a difference between left-target and right-target trials of <3.2 μV during the N2pc measurement window. In terms of eye rotation, this is an average difference in eye rotation of <0.2° between the left-target and right-target trials (which I like to think of as approximately a difference of ±0.1°). That's a pretty tiny deviation (although we need to keep in mind that this is an average, and the deviation on single trials might be up to 1° with our 16 μV threshold). So, this seems "good enough" in terms of the change to the sensory input.

We ordinarily measure the N2pc at all of the parietal and occipital electrode sites, and the propagation factor is 3% or less from the HEOG-bipolar sites to each of these sites (according to Lins et al., 1993). Thus, a voltage difference of <3.2 μV at HEOG-bipolar corresponds to a voltage difference of <0.1 μV at the sites where we are measuring the N2pc component. That seems "good enough" in terms of any confounding voltage in our N2pc measurements.

Have we succeeded in meeting this 3.2 μV criterion in the present participant? It's difficult to be sure in the current plot. A convenient way to see if we've met the criterion is to plot the difference wave using a Y range of -3.2 to +3.2 μV and a time range of -200 to +300 ms. Go ahead and do this. The result should look something like Screenshot 8.7D. If the voltage ever exceeded 3.2 μV, the waveform would be "clipped off" in the plot. Although the waveform did get near the top of this voltage range near the end of the 200-300 ms N2pc measurement window, it never exceeded this threshold. In other words, the small amount of residual eye movement activity for this participant meets our criterion for "good enough."

What should you do if the residual eye movement activity for a given participant is >3.2 μV? The first step is to try changing the rejection parameters. Most obviously, you can try reducing the rejection threshold. Sometimes changing the rejection time window can also help. Your goal is to see if you can reduce the residual HEOG activity to <3.2 μV in the N2pc measurement window (or the measurement window for whatever component you're studying) without rejecting too many trials.

As mentioned earlier in this chapter, my lab automatically excludes participants in our basic science experiments if more than 25% of trials are rejected (which includes trials rejected for other reasons, such as blinks). If we can't get the residual HEOG under 3.2 μV without rejecting more than 25% of trials, we exclude the participant from the final analyses. We've used this approach for about 30 years, and it has worked very well. We end up excluding approximately 20% of participants, which is tolerable. In our schizophrenia studies, we find that both the patient and control groups make more eye movements than the college-age participants in our basic science studies, so we double our thresholds. That is, we require that the residual HEOG activity is <6.4 μV, and we exclude participants for whom more than 50% of trials were rejected.

What thresholds should you use in your own analyses? I can't answer that question, because it depends on the nature of your research. As always, your choice should be made on the basis of the fundamental goal of increasing your ability to accurately answer the scientific question that your studies are designed to address. And you might use aSME values rather than the percentage of trials rejected to decide whether too many trials have been rejected for a given participant. Whatever criteria you choose, however, it is extremely important that those criteria are set prior to analyzing the data.

I hope that these N2pc exercises have provided you with a clear procedure for minimizing eye movement artifacts in experiments with lateralized targets or responses. But even more, I hope these exercises serve as a good example of how to conceptualize the goals of artifact rejection and how to achieve those goals.

This page titled 8.17: Exercise- A Two-Stage Strategy for Eliminating Small But Consistent Eye Movements is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.18: Matlab Script For This Chapter

I've created a script named **MMN_artifact_rejection_example.m** that shows how to implement the interpolation and artifact detection processes described in this chapter. You can find it in the Chapter_8 folder. It runs on the data from Subjects 1-10 in the ERP CORE MMN experiment. The datasets for these participants are in a subfolder named **MMN_Data**.

The script demonstrates how you can put subject-specific information in an Excel spreadsheet, such as which channels to interpolate and what artifact detection parameters to use, and then have the script read this information and use it to control the interpolation and artifact detection processes. This is a super useful trick!

I didn't spend much time customizing the parameters. You can probably do a better job given what you've learned in this chapter.

Most of these participants have a lot of blinks and would need to be excluded from the final analyses because they exceed our criterion of 25% rejected trials. As I noted before, this is because we planned to use artifact correction rather than rejection for blinks, and we did nothing to minimize blinking. I should also note that Subject 7 has a ton of low-frequency drift (probably coming from the reference electrodes, because it's present in all the EEG channels) and was excluded from the final analyses in the ERP CORE paper.

This page titled 8.18: Matlab Script For This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

8.19: Key Takeaways and References

Key Takeaways

- The overarching goal in designing an artifact rejection strategy is to maximize the likelihood that you will obtain an accurate answer to the scientific question your study is designed to answer. You can ignore any of my specific suggestions for implementing artifact rejection if you have a better way of reaching that goal.
- Artifacts are typically problematic for one of three reasons: 1) they are a large source of noise and therefore reduce your statistical power; 2) they differ systematically across groups or conditions, creating a confound; 3) they indicate a problem with the sensory input (e.g., closed eyes during the presentation of a visual stimulus). In most cases, you are rejecting trials with artifacts to address one or more of these issues.
- Decreasing the threshold for rejection typically reduces the confounding effects of artifacts and the problems with the sensory input, and it may also reduce the noise caused by the artifacts. However, when the threshold gets too low, the number of trials remaining in the averaged ERP waveforms gets small enough that the data quality suffers. You can use the aSME values to help find the optimal threshold for rejection.
- You will typically want to implement several different artifact detection procedures for each participant so that you can intelligently detect the different types of artifacts. This is often achieved with one procedure for detecting blinks, another for detecting eye movements, and a third for detecting C.R.A.P.
- Dealing with small but consistent eye movements is tricky, because small eye rotations are difficult to detect but can be a significant confound in experiments with lateralized stimuli or lateralized responses. The two-stage procedure deals with this by using the greater precision of averaged HEOG waveforms to determine whether the small eye movements that escape rejection are large enough to have a substantial impact.

References

Baker, D. H., Vilidaite, G., Lygo, F. A., Smith, A. K., Flack, T. R., Gouws, A. D., & Andrews, T. J. (2020). Power contours: Optimising sample size and precision in experimental psychology and human neuroscience. *Psychological Methods*. http://dx.doi.org/10.1037/met0000337

Eimer, M. (1994). “Sensory gating” as a mechanism for visuospatial orienting: Electrophysiological evidence from trial-by-trial cuing experiments. *Perception & Psychophysics*, *55*, 667–675.

Jas, M., Engemann, D. A., Bekhti, Y., Raimondo, F., & Gramfort, A. (2017). Autoreject: Automated artifact rejection for MEG and EEG data. *NeuroImage*, *159*, 417–429. https://doi.org/10.1016/j.neuroimage.2017.06.030

Kappenman, E. S., Farrens, J. L., Zhang, W., Stewart, A. X., & Luck, S. J. (2021). ERP CORE: An open resource for human event-related potential research. *NeuroImage*, *225*, 117465. https://doi.org/10.1016/j.neuroimage.2020.117465

Lins, O. G., Picton, T. W., Berg, P., & Scherg, M. (1993). Ocular artifacts in EEG and event-related potentials I: Scalp topography. *Brain Topography*, *6*, 51–63.

Luck, S. J. (2012). Electrophysiological correlates of the focusing of attention within complex visual scenes: N2pc and related ERP components. In S. J. Luck & E. S. Kappenman (Eds.), *The Oxford Handbook of ERP Components* (pp. 329–360). Oxford University Press.

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Luck, S. J., Hillyard, S. A., Mouloua, M., Woldorff, M. G., Clark, V. P., & Hawkins, H. L. (1994). Effects of spatial cuing on luminance detectability: Psychophysical and electrophysiological evidence for early selection. *Journal of Experimental Psychology: Human Perception and Performance*, *20*, 887–904.

Luck, S. J., Stewart, A. X., Simmons, A. M., & Rhemtulla, M. (2021). Standardized measurement error: A universal metric of data quality for averaged event-related potentials. *Psychophysiology*, *58*, e13793. https://doi.org/10.1111/psyp.13793

Mangun, G. R., & Hillyard, S. A. (1991). Modulations of sensory-evoked brain potentials indicate changes in perceptual processing

during visual-spatial priming. *Journal of Experimental Psychology: Human Perception and Performance*, *17*, 1057–1074.

Nolan, H., Whelan, R., & Reilly, R. B. (2010). FASTER: Fully Automated Statistical Thresholding for EEG artifact Rejection. *Journal of Neuroscience Methods*, *192*(1), 152–162. https://doi.org/10.1016/j.jneumeth.2010.07.015

Talsma, D. (2008). Auto-adaptive averaging: Detecting artifacts in event-related potential data using a fully automated procedure. *Psychophysiology*, *45*(2), 216–228. https://doi.org/10.1111/j.1469-8986.2007.00612.x

Woodman, G. F., & Luck, S. J. (2003). Serial deployment of attention during visual search. *Journal of Experimental Psychology: Human Perception and Performance*, *29*, 121–138.

Yuval-Greenberg, S., Tomer, O., Keren, A. S., Nelken, I., & Deouell, L. Y. (2008). Transient induced gamma-band response in EEG as a manifestation of miniature saccades. *Neuron*, *58*(3), 429–441. https://doi.org/10.1016/j.neuron.2008.03.027

This page titled 8.19: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9: ARTIFACT CORRECTION WITH INDEPENDENT COMPONENT ANALYSIS

Learning Objectives

In this chapter, you will learn to:

- Conceptualize artifact correction in terms of the overarching goal of your research and the specific problems that artifacts pose when you try to reach that goal
- Decompose the EEG into a set of independent components (ICs), identify ICs that represent artifacts, and reconstruct the EEG without the artifactual ICs
- Apply special preprocessing steps to obtain optimal ICs
- Evaluate the effectiveness of the artifact correction procedure in terms of both data quality and confounds
- Intelligently choose which ICs should be removed from your data

This chapter explains how to use independent component analysis (ICA) to correct certain kinds of artifacts (especially blinks and eye movements). ICA-based artifact correction is a real godsend for experiments in which artifact rejection would throw out too many trials. And it can improve the data quality for other experiments by allowing you to include most or all of the trials in your averaged ERPs.

However, ICA-based artifact correction massively changes your data. Every single data point is impacted. And if done improperly, ICA can make your data worse and lead to incorrect conclusions. It's a bit like using *backburn* to deal with a wildfire (i.e., starting a controlled fire to eliminate the fuel for the wildfire). If you're not careful, it can get out of control and damage what you were trying to save. You really need to know what you're doing with ICA to get the best results and avoid getting burned.

Before I wrote this chapter, I did a lot of reading to make sure I was up to date and that the strategies described in this chapter would reflect the current state of the art. I also spent a lot of time applying ICA to the ERP CORE data and carefully assessing the results. The Makeig group at UCSD are still the world's experts at ICA-based artifact correction, so much of what I write in this chapter is based on their recommendations. EEGLAB's ICA documentation is an excellent resource, especially the videos created by Arnaud Delorme. The page of informal advice from Makoto Miyakoshi (called Makoto's Preprocessing Pipeline) is also extremely useful. I recommend that you read these sources after reading the present chapter. I also recommend reading the general overview of artifact correction near the end of Chapter 6 of Luck (2014), along with the online supplement to that chapter.

As you read this chapter, keep in mind that the ultimate goal of artifact correction is the same as the ultimate goal of artifact detection, which is to accurately answer the scientific question that the experiment was designed to address. In addition, you should keep in mind the three main problems that we need to address in artifact rejection and correction: reduced statistical power as a result of increased noise, systematic confounds, and sensory input problems.

9.1: Data for this Chapter

9.2: Exercise- A First Pass at ICA-Based Blink Correction

9.3: Exercise- Evaluating the Impact of Artifact Correction

9.4: Background- A Quick Conceptual Overview of ICA

9.5: Exercise- Making ICA Work Better

9.6: Exercise- Transferring the Weights and Assessing the ICs

9.7: Exercise- Deciding Which ICs to Exclude

9.8: Exercise- Deleting C.R.A.P. Prior to ICA

9.9: General Recommendations

9.10: Matlab Scripts For This Chapter

9.11: Key Takeaways and References

This page titled 9: Artifact Correction with Independent Component Analysis is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.1: Data for this Chapter

The data we'll be using for the exercises in this chapter can be found in the Chapter_9 folder in the master folder: https://doi.org/10.18115/D50056.

This page titled 9.1: Data for this Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.2: Exercise- A First Pass at ICA-Based Blink Correction

I find that it's easiest to understand ICA by starting with an example and then explaining how it works after you've seen it in action. We're going to start with the data Subject 10 from the ERP CORE MMN experiment, which we looked at in the previous chapter in the section on eye movement detection.

Launch EEGLAB (or quit and restart it if was already running). Set **Chapter_9** to be Matlab's current folder, and load the dataset named **10_MMN_preprocessed.set**. *Make sure you're getting the data from the Chapter_9 folder rather than the Chapter_8 folder, because the data are preprocessed a little differently but the filenames are the same.*

The dataset you loaded has been high-pass filtered at 0.1 Hz and referenced to the average of P9 and P10 (except for the VEOG-bipolar and HEOG-bipolar channels). Take a look at the data to remind yourself what it looks like. You'll see lots of blinks and eye movements prior to the start of the stimuli at ~25 seconds.

When we looked at this participant's data in the previous chapter, the EEG had been low-pass filtered. Without the low-pass filtering, you can now more easily see the *spike potential* at the beginning of each eye movement, which is a result of the muscle activity that sets the eyes in motion. This is illustrated in Figure 9.1, which shows the first 5 seconds of data from Subject 10. You can see a brief spike in several channels at the moment when the HEOG voltage starts to change from one level to another (which indicates a change in gaze location from one location to another). You may want to adjust the vertical scale and number of channels being displayed so that you can see these voltage changes more clearly.

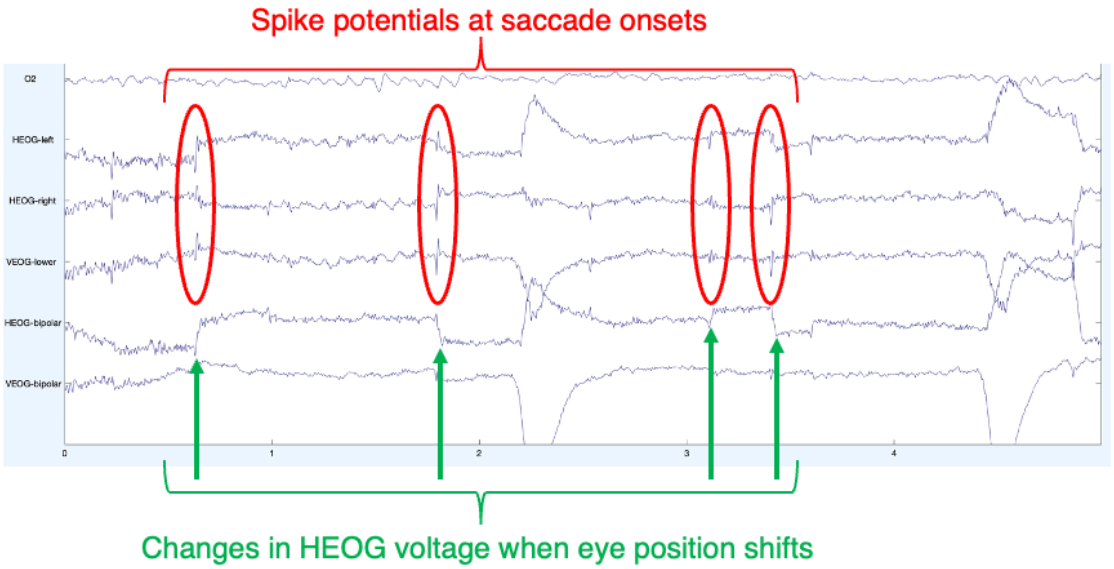
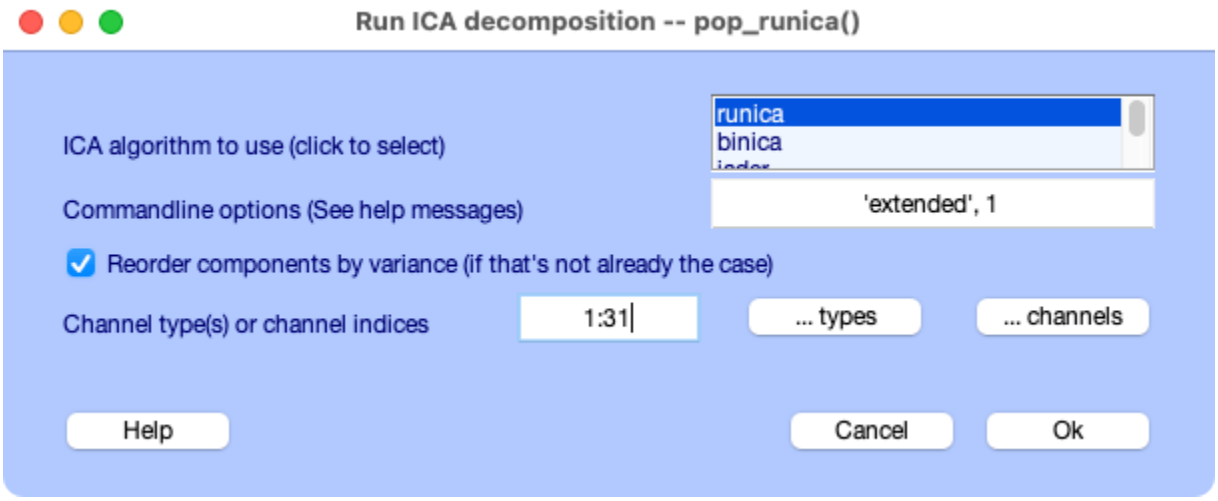


Figure 9.1 Examples of spike potentials and HEOG voltage associated with saccadic eye movements.

Once you're done scanning through the EEG to see what the whole session looks like, select **EEGLAB > Tools > Decompose data by ICA** and set the parameters as shown in Screenshot 9.1 Everything is probably already set correctly, except for the channels, which you should set to **1:31**. With this channel list, we're including all the channels that have the same reference, and we're excluding VEOG-bipolar and HEOG-bipolar. ICA assumes that the channels are linearly independent of each other, and the VEOG-bipolar and HEOG-bipolar were created by recombining other channels, so they would violate this assumption. However, they are useful channels to have so that we can tell when blinks and eye movements occurred even after we've corrected the EEG data.

Screenshot 9.1



Click **OK** to start the process of computing the ICA weights. You'll see some text appear in the Matlab command window, followed by a long series of lines that look like this:

step 1 - lrate 0.001000, wchange 16.88431280, angledelta 0.0 deg

step 2 - lrate 0.001000, wchange 0.90426626, angledelta 0.0 deg

ICA works by training a machine learning algorithm (much like a neural network), and it can take the algorithm a very long time to converge on a solution. The amount of time depends on the number of channels, the noise level of the data, and the speed of your computer. It took over 5 minutes to reach a solution for this dataset on my laptop. If you don't want to wait for it to finish, you can kill the process by clicking the **Interrupt** button that should be visible while ICA is running. If that doesn't work, you can usually kill a Matlab process by typing **Ctrl-C**.

If you waited for it to finish, you should change the name of your dataset to indicate that ICA weights have been added to it. You can do this with **EEGLAB > Edit > Dataset info**. You can just add **_rawICAweights** to the name of the dataset and click **OK**. If you didn't wait for the process to finish, you can just load the dataset named **10_MMN_preprocessed_rawICAweights.set**, which has the ICA weights that I got when I ran the decomposition.

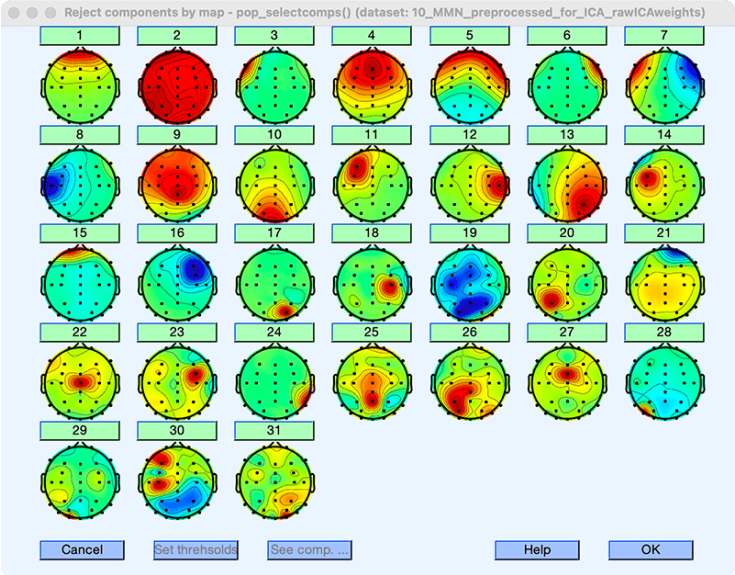
An easy way to see that the ICA weights have been added is to type **EEG** in the Matlab command window. It will show you the contents of the current EEG structure, and you can see that the fields beginning with **ica** are now filled:

```
    icaact: [31×157184 single]
   icawinv: [31×31 double]
 icasphere: [31×31 double]
icaweights: [31×31 double]
icachansind: [1×31 double]
```

ICA decomposes the EEG data into a set of *independent components* or *IC*s. These are statistically defined components, not necessarily ERP components, and they may not be biologically meaningful. I will refer to them as *IC*s rather than as *components* to maintain this important distinction.

Each IC is a *spatial filter*. When we apply an IC to the distribution of voltages over the scalp at a given moment in time, the output is the magnitude of that IC at that moment in time. Conveniently, ICA creates scalp maps of the ICs showing how each channel is weighted by the IC. Take a look at the scalp maps now by selecting **EEGLAB > Tools > Inspect/label components by map**. (You can also view the maps with **EEGLAB > Plot > Component Maps**). You should see something like Screenshot 9.2.

Screenshot 9.2



If you didn't let the component decomposition process finish but instead loaded the file that already had the weights in it, the scalp maps you see should look exactly like those in the screenshot. However, if you ran the component decomposition process yourself, the maps might look a little different. This is because the ICA machine learning algorithm contains some randomization and therefore doesn't yield exactly the same results every time. If things are working well, then the results should be similar if you repeat the decomposition process multiple times. If you get very different results, then something is wrong (usually either very noisy data, not enough data, or a linear dependency among your channels).

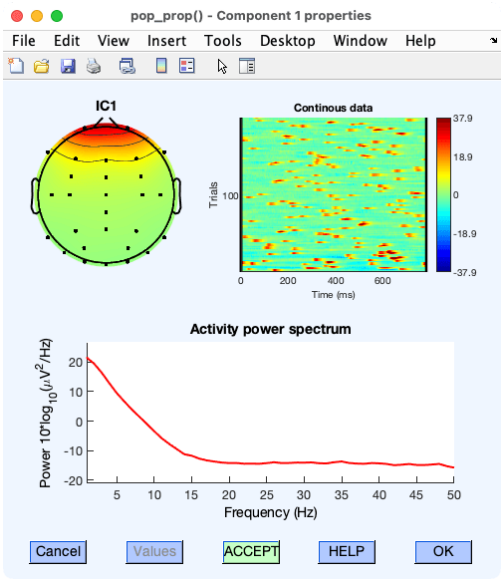
There are some things that may differ across repetitions that don't indicate a problem. First, the ordering of the ICs might change a little. For example, the scalp maps for ICs 13 and 14 might be swapped in your data. Second, there might be small changes in the weights for a given scalp map. Third, the polarity of the scalp maps might be inverted (which we will discuss in more detail later). These are normal and no cause for concern. However, if you see radically different maps, then something is wrong.

Notice that there are 31 ICs. This is because we gave the algorithm data from 31 channels. Ordinarily, the number of ICs is equal to the number of channels. This is necessary to make the math work. But it indicates an important way in which ICA is an imperfect method for identifying the true components underlying your data. The number of channels that you record from determines the number of ICs, but putting more or fewer electrodes on the scalp doesn't change the number of sources of brain activity!

When the decomposition algorithm finishes, it reorders the components in terms of the amount of variance in the data they account for, with IC 1 accounting for the greatest variance. If the participant blinked frequently, the IC corresponding to the blinks is usually in the top five ICs (and often IC 1). In our current example, IC 1 has the kind of far frontal scalp distribution that you'd expect for a blink. But is this really a blink IC, or is it some other kind of artifact, or maybe even frontally distributed brain activity?

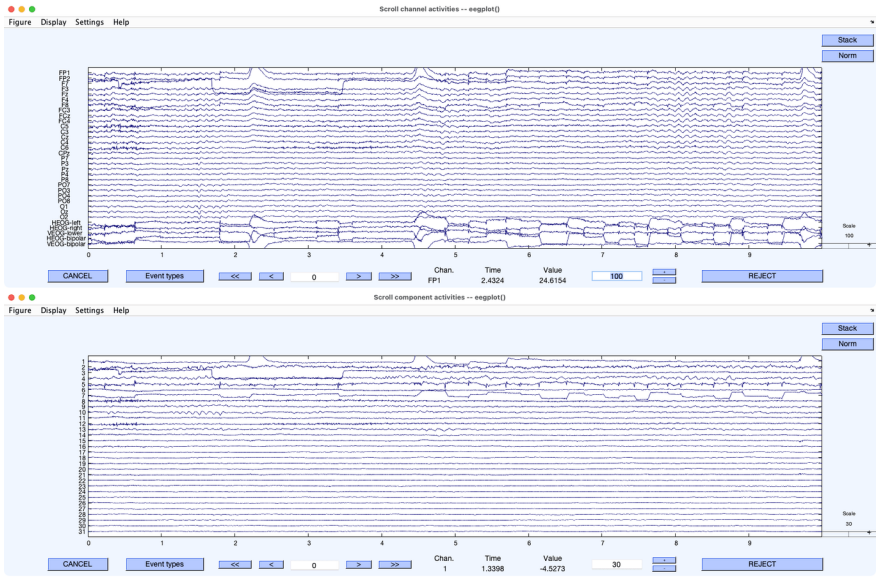
There are a few things we can check to determine whether IC 1 reflects blinks. If you click the number above IC 1, a new window will pop up showing you more details about this IC. In the upper right of the window, you'll see a heat map showing how the magnitude of this IC varies over time. The X axis is time within a "trial" (which are arbitrary time periods when continuous data are used). The Y axis is time over the course of the recording. It's labeled **Trials** but actually corresponds to time in continuous data. You can see many small red blobs. Each of those is a brief time period in which the magnitude of IC 1 was large. This is exactly what we'd expect for blinks: many brief blips of voltage. The window also shows the frequency spectrum. The spectrum for IC 1 shows a gradual decline as the frequency increases, which is also characteristic of blinks (but is also seen for many types of non-blink activity).

Screenshot 9.3



The next thing you can do (which I find the most informative) is to directly compare the time course of IC 1 with the time course of the EEG and EOG signals. Start by selecting **EEGLAB > Plot > Channel data (scroll)** to show the time course of the EEG and EOG signals. While that window is still open, select **EEGLAB > Plot > Component activations (scroll)**. Then arrange the windows above and below each other as shown in Screenshot 9.4 (which is set to show 10 seconds per screen). Now you can look at how IC 1 activation varies over time and compare it to the EOG and EEG signals at the corresponding time points. Try scrolling through both windows to see whether IC 1 corresponds to the VEOG-bipolar signal. You may want to display only ~6 channels in each plotting window so that you can see these signals better.

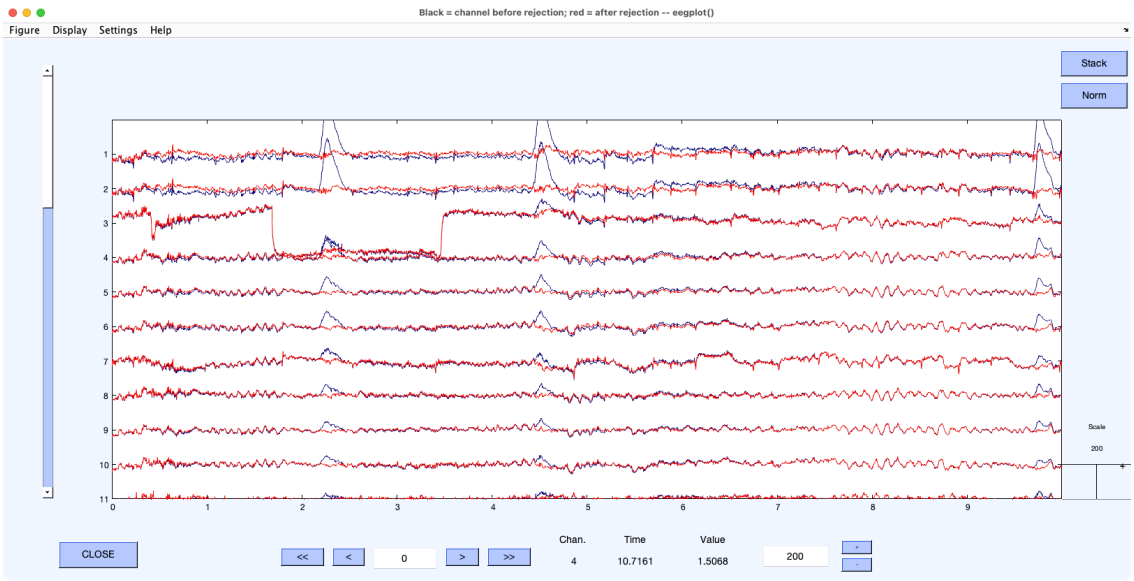
Screenshot 9.4



The first thing to notice is that each blink that you can see in the VEOG-bipolar channel is accompanied by a deflection with the same shape in IC 1 (e.g., at 2.2 and 4.4 seconds). You should also notice that there are occasional step-like deflections in both VEOG-bipolar and IC 1. These are vertical eye movements. Often, but not always, blinks and vertical eye movements will be captured by the same IC. Given the scalp distribution of IC 1 and its close correspondence in time with the blinks and eye movements in the VEOG-bipolar channel, we can be quite sure that this IC reflects vertical EOG activity.

Now let's remove IC 1 from the EEG/EOG data to eliminate the blinks and vertical eye movements. Ordinarily, we would also remove other artifact-related ICs at this point (e.g., horizontal eye movements), but we'll just focus on blinks and vertical eye movements for now. To do this, select **EEGLAB > Tools > Remove components from data**, put **1** in the window labeled **List of component(s) to remove from data** so that we remove activity corresponding to IC 1, and click **OK**. You'll then see a confirmation window. Click the button in this window for plotting single trials. You should see something like Screenshot 9.5 (but I've told it to display the top 10 channels and show a 10-second time period). The blue waveform shows the original data and the red waveform shows what the data will look like after removing IC 1.

Screenshot 9.5



You can see that the algorithm has done a good job of eliminating the blinks and vertical eye movements. For example, the blinks at 2.2 and 4.4 seconds are now gone, and there is no longer a sudden step in voltage at 5.7 seconds. There are also some slower differences between the corrected and uncorrected data, which probably represent sustained changes in the vertical EOG corresponding to changes in eye or eyelid position.

Now click the **ACCEPT** button in the confirmation window, and use **10_MMN_preprocessed_rawICAweight_pruned** as the dataset name. Next, plot the new dataset with **EEGLAB > Plot > Channel data (scroll)**. You'll see that the blinks are now gone from all channels, except for the VEOG-bipolar and HEOG-bipolar channels. We excluded those channels from ICA, and now we can use them to see when the blinks were in the original data. You can still see blinks at 2.2 and 4.4 seconds in the VEOG-bipolar channel, but the blink activity has been removed from the other channels.

If you think about it, this is something of a miracle. We've mathematically eliminated the contribution of blinks and vertical eye movements to the EEG at every single time point. But did this help us address the three types of problems caused by artifacts that were discussed in Chapter 8? The next exercise shows how to answer this question.

This page titled 9.2: Exercise- A First Pass at ICA-Based Blink Correction is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.3: Exercise- Evaluating the Impact of Artifact Correction

One of our goals for artifact rejection and correction is to reduce noise that would otherwise decrease the data quality in our averaged ERPs. Another goal is to minimize the confounds that happen when the artifacts differ across groups or conditions. To evaluate these issues, we need to calculate the SME values and the averaged ERP waveforms for the corrected data, compare them with the original data, and compare them with the data after applying artifact rejection. I've done all this and provided the ERPsets in the Chapter_9 folder. Go ahead and load these three ERPsets (**10_MMN_uncorrected**, **10_MMN_rawblinkcorrection**, and **10_MMN_blinkrejection**).

Creating the ERPsets

If you want to create these ERPsets yourself, you'll need to add an EventList to the dataset, run BINLISTER (using **BDF_MMN.txt**), and epoch the data. You'll need to do this once for the original data (prior to correction) and once for the new dataset in which IC 1 was removed. And for the original uncorrected data, you'll want to have two versions, one with artifact detection (step-like artifacts, threshold = 50) and one without artifact detection. This will allow you to make three ERPsets, one without correction or rejection, once with correction only, and one with rejection only. Make sure to customize the data quality parameters to add a window of 125-225 ms.

Now plot the ERP waveforms for each ERPset. To make them comparable, uncheck the **auto y-scale** box and put **-12 12** in the **Y range** text box before plotting. The results for several key channels are summarized in Figure 9.2.

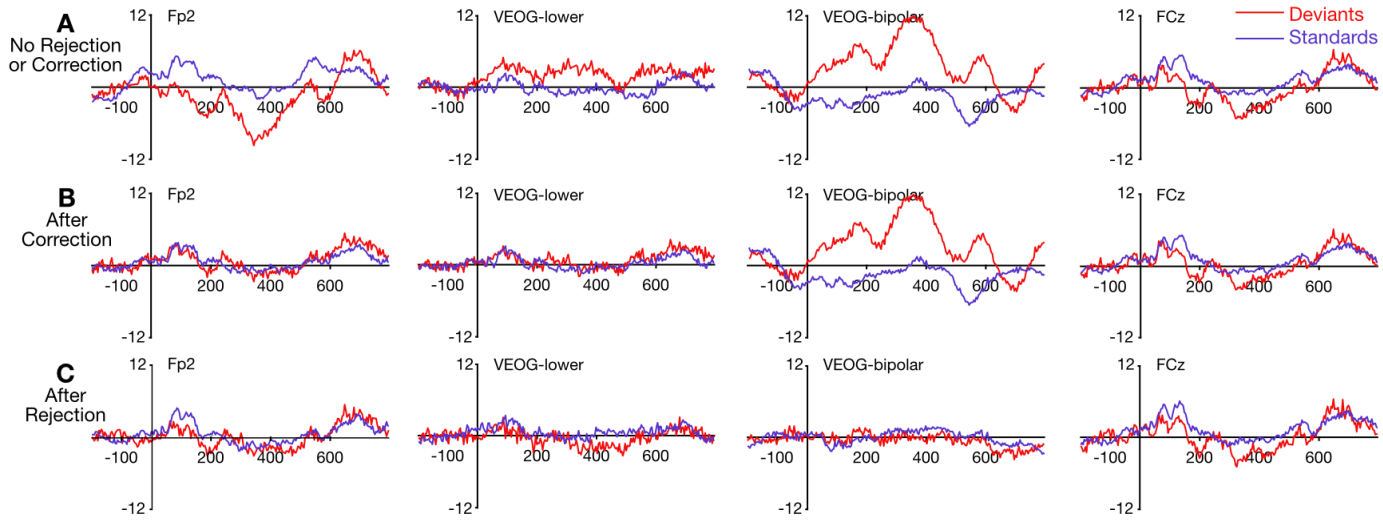


Figure 9.2. Averaged ERP waveforms from Subject 10 in the ERP CORE MMN experiment without any correction or rejection (A), after correcting for blinks by removing IC 1 (B), and after rejecting trials containing blinks (C). Note that the VEOG-bipolar channel was excluded from artifact correction in (B) and was therefore identical to (A).

Large differences between standards and deviants can be seen in the Fp2 and VEOG channels when nothing was done to deal with the blinks (Figure 9.2.A). You can see a polarity reversal of the deviance effect between the Fp2 and VEOG-lower channels, consistent with blink activity. That was also true for Subject 1 in the previous chapter (see Figure 7.2.A), although the specific pattern was different. The finding of different blink-related activity for standards and deviants indicates that blinks are a confound that we need to eliminate.

The blink-related voltage deflections were eliminated by both artifact correction rejection. Note, however, that the VEOG-bipolar channel was excluded from correction, so the large deflections remain in this channel. Following both correction and rejection, the voltage is more negative for deviants than for standards in FCz in the 125-225 ms time range, which is our primary measure of the MMN. This effect was larger when no rejection or correction was performed, so part of this effect may have been coming from the blink activity that we can see during this time range in the Fp2 and VEOG electrodes. This provides even more evidence that we need to deal with the blinks to avoid confounding our measure of brain activity with blink activity (or vertical eye movements).

When artifact correction was performed, the difference between standards and deviants at Fp2 and VEOG-lower was largely eliminated. However, some difference remained in these channels when artifact rejection was performed. Which of these reflects the true pattern of brain activity? It's difficult to be certain on the basis of these waveforms. However, given that the MMN appears to be generated primarily in the supratemporal plane (Näätänen & Kreegipuu, 2012), it should be substantially larger at FCz than at Fp2, and it should not invert in polarity between Fp2 and VEOG-lower (when referenced to P9/P10). I therefore suspect that some ocular activity escaped rejection (most likely vertical eye movements, which may fail to reach the 50 μV threshold for rejection that I used for detecting blinks). As a result, it seems likely that correction brought us "closer to the truth" than rejection in this particular case. And that's our ultimate goal!

It's also important to assess the impact of correction and rejection on data quality. For each of the three ERPsets, you should display the data quality measures in a table. Let's focus on the aSME values for FCz from 125-225 ms. For both the deviants (Bin 1) and the standards (Bin 2), the aSME was worst (highest) for the data without correction or rejection (Bin 1 = 0.9774, Bin 2 = 0.5688), and was improved (reduced) by rejection (0.8338, 0.5360), and was improved even more by correction (0.7802, 0.4523). This pattern makes sense because both rejection and correction minimize the uncontrolled variation produced by the blinks, but rejection reduces the number of trials whereas correction does not.

From these results, correction seems to be the better method for this particular participant in terms of both minimizing ocular confounds and maximizing data quality. That fits with my experience: When implemented correctly, ICA-based artifact correction tends to be better than rejection for dealing with blinks. And sometimes the difference is quite large, especially when a large number of trials would need to be rejected.

However, we didn't really implement correction very well in this example. When ICA is working properly, the scalp maps of most of the ICs should look like nice gradual gradients with a unipolar pattern (a single positive or negative focus, like IC 1 in Screenshot 9.2) or a dipolar pattern (opposing positive and negative focus, like IC 7 in Screenshot 9.2). Scalp distributions that cover the entire head (like IC 2) or are complex and irregular (like ICs 19, 26, and 30) are a problem. They don't resemble the topography we'd expect for brain activity or common artifacts, and they're a sign that multiple sources are being mixed together rather than being separated. A few such maps are okay, especially in the latter ICs that don't account for much variance. But you really don't want to see a map like IC 2 in the top half of the ICs. In a later exercise, we'll see how to improve the ICA decomposition (and make it faster as well). First, however, we need to discuss how ICA works.

This page titled 9.3: Exercise- Evaluating the Impact of Artifact Correction is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.4: Background- A Quick Conceptual Overview of ICA

The online supplement to Chapter 6 in Luck (2014) provides a conceptual overview of how ICA works in general and how it is applied to artifact correction. Here I'll provide a quick summary. Several different algorithms are available for performing the ICA decomposition. For the vast majority of cases in which ICA is used for artifact correction, there aren't big differences among the algorithms, so here I'll focus on EEGLAB's default ICA algorithm (*Infomax*, implemented with the **runica** routine).

The first thing you should know is that ICA is purely a statistical technique, and it was not developed for neural data per se. It knows nothing about brains or electricity. It doesn't know that the data are coming from electrodes or where the electrodes are located. Most ICA algorithms don't even know or care about the order of time points. They just see each time point as a set of N abstract variables, one for each of the N channels. Infomax uses a machine learning algorithm (much like a neural network) that learns a set of N ICs that are maximally independent when applied to the data.

By *maximally independent*, I mean that the activation level of one IC provides no information about the activation levels of the other ICs at that time point. For example, if the blink-related voltage level at each time point does not predict other sources of activity at the same time point, blinks will likely be extracted as a separate IC. However, it's not a problem if blink activity at one time point predicts activity from other sources at earlier or later time points.

ICA learns an *unmixing matrix*, which converts the EEG data at a given time point to the activation level of each IC. The inverse of the unmixing matrix is the *mixing matrix*, which is just the scalp distribution of each IC. You can also think of the scalp distribution of an IC as a set of weights. The voltage produced by an IC in a given channel at a given time point is the activation level of the IC at that time point multiplied by the weight for that channel. Some randomness is applied to the learning algorithm, so you won't end up with exactly the same set of ICs if you repeat the decomposition multiple times.

An important practical consideration is that the machine learning routine needs a lot of data to adequately learn the ICs. The EEGLAB team has provide an informal rule for this, which is that the number of time points in the dataset must be at least 20 x (# channels)^2. It's probably the number of minutes of data that matters rather than the number of time points, but the key thing to note is that the number of channels is squared. This means that doubling the number of channels requires four times as much data. For example, you would need four times as many minutes of data for a 64-channel recording as for a 32-channel recording (and sixteen times as much data for a 128-channel recording as for a 32-channel recording).

ICA is somewhat like principal component analysis (PCA). However, whereas PCA tries to lump as much variance as possible into the smallest number of components, ICA tries to make the components maximally independent. ICA is also like PCA insofar as it just takes the dataset and represents it along a different set of axes. You can go from the original data to the ICA decomposition with the unmixing matrix, and then you can apply the mixing matrix to the ICA decomposition and perfectly recover the original data.

This decomposition-and-recovery sequence is how ICA corrects for artifacts. After running the ICA decomposition to get the ICs, you simply set one or more of the ICs to have an activation of zero at each time point and then use the mixing matrix to recover the original data (but without the artifactual ICs). This means that ICA influences your data at every single time point. When you remove a blink IC, ICA doesn't just find time periods with blinks and correct the data during those time periods. It reconstructs your EEG data at every time point, but with the artifactual ICs set to zero. There will be some nonzero activity in the blink IC at each time point, so zeroing this IC at each time point means that the data will be changed at least slightly at every time point. This is actually good, because there may be quite a lot of EOG activity between blinks as a result of small changes in eye rotation or eyelid position, and ICA will remove this non-neural activity when you remove the IC corresponding to blinks.

ICA makes several important assumptions (see Luck, 2014), but two are particularly important to know about. The first is that the scalp distribution of a given source of activity must remain constant over the entire session. For example, we can assume that the locations of the eyes relative to the electrode sites will not change over the course of a session (unless there is some kind of catastrophe), so blinks and eye movements meet this criterion. Similarly, the location of the heart relative to the electrodes doesn't change over time, so the EKG artifact also meets this criterion. However, the scalp distribution produced by skin potentials will depend on which sweat pores are activated, which may change over time, so skin potentials do not meet this assumption. By the way, this assumption means that you must perform ICA separately for each participant (because the scalp distributions will differ at least slightly across participants).

There is dispute in the literature about whether ICA works well with EMG. The argument against using ICA with EMG is that

different muscle fibers may contract at different time points, changing the scalp distribution. The argument for using ICA is that the scalp distribution does not actually change very much over time. To be on the safe side, my lab doesn't use ICA for EMG. We minimize EMG by having participants relax during the EEG recording, and we can filter out the remaining EMG so that it has minimal impact on our results. However, if you cannot avoid having a lot of EMG in your data, and you can't filter it out without creating other problems (e.g., because you're looking at high-frequency ERP activity), you can read the literature and decide for yourself whether the benefits of using ICA for EMG outweigh the costs.

A second key assumption of ICA is that the number of true sources of activity is equal to the number of channels. This is related to the fact that the number of ICs must be equal to the number of channels in order for the math to work.

Exceptions Make the Rule

There are occasional exceptions to the rule that the number of ICs is equal to the number of channels, particularly when you are using the average of all sites as the reference. See Makoto's Preprocessing Pipeline or EEGLAB's ICA documentation for details.

As I mentioned earlier, the fact that the number of ICs must equal the number of channels means that ICA is an imperfect method. You don't change the number of sources of activity when you add or subtract electrodes! Also, there will always be more sources of activity in the EEG signal than there are channels (because each synapse in the brain is a potential source of activity). As a result, ICA will lump multiple true components into the same IC. In addition, a single true source may also be split among multiple ICs. So, you will definitely have lumping of true components, and you will likely have some splitting as well.

Given the failure of EEG data to meet this second assumption, you may wonder whether it is valid to use ICA for artifact correction. As famously noted by the statistician George Box, all statistical models are wrong, and the question is not whether they are correct but whether they are useful (Box, 1976). In practice, ICA is useful for correcting some kinds of artifacts despite the invalid assumptions. The saving grace of ICA is that the lumping and splitting problems are minimal for components that account for a lot of variance (e.g., components that are both large and frequently occurring). Most participants blink a lot, and blinks are very large, so ICA typically works very well for blinks. Depending on the experiment and the participant, eye movements can be large or small and they can be frequent or rare. In my experience, ICA works only modestly well for eye movements, and it can't correct for the change in sensory input produced by the change in gaze position, so we only use ICA to correct for eye movements when necessary. However, I recently came across a nice paper by Dimigen (2020) showing that ICA can work quite well for large and frequent eye movements when the right preprocessing steps are applied prior to the ICA decomposition (as I'll discuss in more detail later). Drisdelle et al. (2017) also provide evidence that ICA can work well for eye movements in certain types of paradigms.

ICA can be applied either to continuous or epoched EEG. When my lab first started using ICA many years ago, I emailed Scott Makeig and Arnaud Delorme to get their advice, and they recommended applying it to the continuous EEG. They still give this advice today in the EEGLAB documentation. You can apply ICA to epoched data if necessary, but the epochs must be at least 3 seconds long (e.g., -1000 to +2000 ms). Adjacent epochs cannot contain the same data points, so this means that you must have relatively long trials for this approach to work. If you get your pipeline set up properly (see Chapter 11 and Appendix 3), there isn't any reason why you'd need to apply ICA to epoched data, so my view is that the safest thing to do is to apply it to the continuous data. As described in the text box below, there may also be a practical advantage.

A Practical Advantage

Over the years, we've found a significant practical advantage to doing ICA at the earliest possible stage of EEG preprocessing (which means applying to continuous EEG, because epoching is a relatively late stage). Specifically, ICA is a time-consuming process that you don't want to repeat if you can possibly avoid it. If you need to change some of your processing steps after you've already analyzed your data once, putting ICA at the earliest possible stage minimizes the likelihood that this change will require repeating the ICA.

The ICA decomposition process typically takes somewhere between 2 minutes and 2 hours depending on the nature of your data and your computer. If you need to process data from 30 participants, this is now between 60 minutes and 60 hours. That can be done overnight while you're asleep, but another 2-20 minutes of human effort are required for each participant to make sure that the decomposition has worked properly and to determine which ICs should be removed. That's 60 to 600 minutes of your precious time.

> What's the likelihood that you will need to re-process your data? In my experience, the likelihood is close to 100%! Reviewers always seem to want some change (or some secondary analysis). And when you're new to ERP analysis, you're likely to do something that is less than optimal and will require a re-analysis. But if you've done the artifact correction at the earliest possible point in your processing pipeline, chances are good that you won't need to repeat this time-consuming part of your pipeline.

A key step in ICA-based artifact correction is to determine which ICs correspond to artifacts and should be removed. There are automated algorithms for this, but I recommend doing it manually for the vast majority of studies. As you will see, you need to carefully determine whether a given IC should be removed, which requires taking into account the three underlying goals of artifact rejection and correction, and this often goes beyond what an algorithm can do.

ICA-based artifact correction massively changes your data, and we know we are violating at least one of its assumptions, so I recommend being conservative in using it. We almost always use it for blinks, and we sometimes use it for eye movements, but we don't ordinarily use it for other kinds of artifacts. If we frequently encountered large EKG artifacts, we'd probably use ICA for those as well. Some labs use ICA for anything that looks "weird", but I personally don't like that approach. There are other ways of dealing with these other types of artifacts, and I just don't trust an algorithm to solve every problem in my data.

Finally, don't forget Hansen's Axiom: There's no substitute for good data. Do everything you can to minimize artifacts during the recording, and then you won't end up getting an ulcer from worrying about how to deal with a ton of artifacts during the analysis.

This page titled 9.4: Background- A Quick Conceptual Overview of ICA is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.5: Exercise- Making ICA Work Better

Our first attempt at ICA-based blink correction was quite simplistic, and as a result the decomposition was not very good (as evidenced by the irregularly shaped scalp maps for some of the ICs in Screenshot 9.2). The main reason for this is that our dataset (like most EEG datasets) contains some really large activity that is difficult for ICA to handle because it is either infrequent or lacks a consistent scalp distribution (e.g., skin potentials). In this exercise, we'll look at an approach that will minimize the effects of those large noise sources and also make the decomposition run faster. It's a win-win!

There are several steps for making the ICA decomposition work better. Some of them are based on a trick, which is to create a new dataset in which we've removed signals that will be problematic for ICA, do the decomposition on this new dataset, and then transfer the ICA weights back to the original dataset. This approach was controversial when I wrote my other book, so I didn't recommend it in Chapter 6 of Luck (2014). However, the field has largely converged on the idea that this approach is both theoretically justified and practically useful.

The first thing we're going to do is heavily filter the data, using filter settings that I would never ordinarily use for ERP experiments. However, the filtered data will be used only to do the ICA decomposition, and then we'll transfer those ICA weights to the original data.

Why Heavy Filtering is OK for the ICA Decomposition

Filters change the time course of an underlying IC, but not the scalp distribution, Heavy filtering is therefore fine for the data used for the ICA decomposition, because we only care about scalp distributions at this stage, not the timing. Once we have the IC weights, we'll transfer them back to the original data without the temporal distortion caused by the filtering.

To get started, quit and restart EEGLAB and then load the original dataset for Subject 10 (**10_MMN_preprocessed**). Select **EEGLAB > ERPLAB > Filter & Frequency Tools > Filters for EEG data**, specifying a high-pass cutoff of 1 Hz and a low-pass cutoff of 30 Hz, with a slope of 48 dB/octave. Run the routine and save the resulting dataset as **10_MMN_preprocessed_filt**. We've now eliminated most of the skin potentials and EMG, both of which may violate the assumptions of ICA and interfere with the decomposition.

Our next step is to downsample the data to 100 Hz. This just makes the ICA decomposition run faster (as does the reduction of noise produced by the filtering). To do this, select **EEGLAB > Tools > Change sampling rate**, enter **100** as the new sampling rate, and then name the resulting dataset **10_MMN_preprocessed_filt_100Hz**.

The next thing we're going to do to improve ICA is remove the segments of data during breaks, when the participants may be moving, scratching their heads, chewing, etc. The voltages produced by these actions are typically large but don't have a consistent scalp distribution. EEGLAB allows you to manually select and delete these time periods from a continuous dataset. ERPLAB adds an automated procedure for this. It just finds periods of time without any event codes and deletes them. Let's give it a try.

Select **EEGLAB > ERPLAB > Preprocess EEG > Delete Time Segments (continuous EEG)**. In the window that appears, specify **1500** as the **Time Threshold**. This tells the routine to delete segments in which there are no event codes for at least 1500 ms. The event codes in the MMN experiment occur every ~500 ms, so periods of 1500 ms without an event code must be breaks. Of course, you'd need to use a longer value for experiments with a slower rate of stimuli. Also specify **500** as the amount of time prior to the first event code in a block of trials. This will make sure that we have at least 500 ms for the prestimulus baseline period for the first event in a trial block. Similarly, specify **1500** ms as the time period after the last event code in a block so that we have a good period of data following this event code.

Enter **1** in the field for **Eventcode exceptions**. This tells the routine to ignore this event code. Our stimulus presentation program sent some event codes with a value of 1 during the breaks, and we want those to be ignored when looking for break periods. Similarly, check the box for ignoring boundary events. Boundary events occur whenever there is a temporal discontinuity in the data. Most commonly, this happens when you have a separate data file for each block and then concatenate them together into a single file. When you do this, a boundary event is inserted at the time point of the transition between one block and the next. Boundary events often occur during breaks, so we want them to be ignored when we're looking for long periods between event codes. Make sure the **Display EEG** button is checked and click **RUN**.

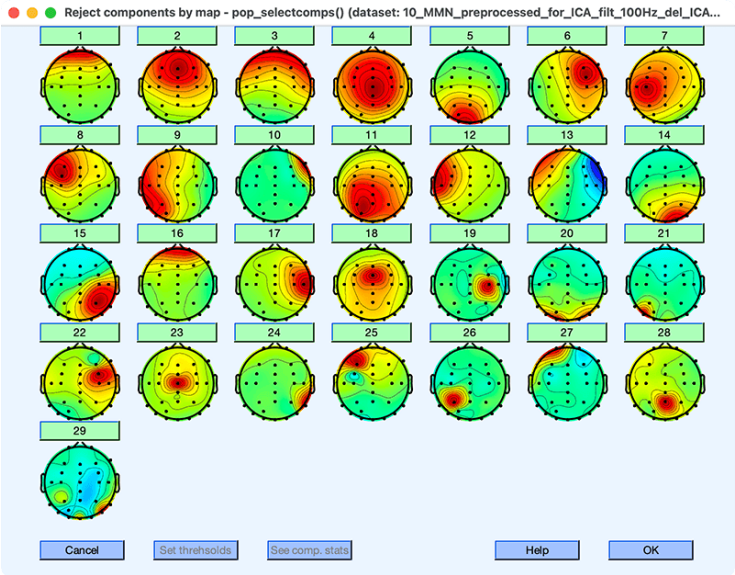
You'll then see two new windows, one for displaying the EEG dataset and one for saving the new dataset. If you scroll through the data, you'll see a red background for the first ~25 seconds and the last ~3 seconds, which are the break periods at the beginning and end of the trial block (we have only one trial block in this experiment). These are the time periods that will be deleted. You should always verify that it worked correctly, and then you can save the new dataset as **10_MMN_preprocessed_filt_100Hz_del**.

Finally, we're going to deal with bad channels. F7 and PO4 both show some crazy behavior in this dataset. If we leave them in the data, we'll end up with an IC that just represents F7 and another that just represents PO4. In fact, you can see this in our original ICA decomposition in Screenshot 9.2, where IC 3 just represents the F7 channel and IC 17 just represents the PO4 channel. We're going to interpolate these channels eventually, so why include them in the ICA decomposition? We could interpolate them prior to the ICA decomposition, but that might create a linear dependency among the channels that would mess up the decomposition. Instead, we'll just exclude them from the ICA decomposition process, and then we can interpolate them after artifact correction is complete.

Now let's apply the ICA decomposition to this dataset by selecting **EEGLAB > Tools > Decompose data by ICA**. We're going to exclude Channels 3 and 24 (the channels for F7 and PO4, respectively), and we're also going to exclude Channels 32 and 33 (the bipolar EOG channels). To do this, type 1 2 4:23 25:31 into the **Channel type(s) or indices** field. As before, make sure to use **runica** as the ICA algorithm and that **'extended', 1** is specified in the field for options (see the EEGLAB ICA documentation for an explanation of this option). Then run the routine. If you don't want to wait for it to finish, you can just load the dataset I created after running the decomposition, named **10_MMN_preprocessed_filt_100Hz_del_ICAweights**.

Once you have the weights, you can view the results with **EEGLAB > Tools > Inspect/label components by map**. The result I obtained in shown in Screenshot 9.6 (your results may differ slightly, especially the ordering and polarity of the ICs). These maps are much nicer than our original maps (Screenshot 9.2). Almost all have either a single focus with a largely monotonic decline (e.g., IC 1 and IC 2) or a bipolar configuration (e.g., IC 13 and IC 15). IC 20 has two foci that are approximately mirror-symmetric across the left and right hemispheres, which is also a perfectly normal pattern (and arises when the two hemispheres operate synchronously). The only irregular map is for IC 29. Because the maps are ordered according to the amount of variance they explain, you shouldn't worry if the last couple of maps aren't perfect.

Screenshot 9.6



This page titled 9.5: Exercise- Making ICA Work Better is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.6: Exercise- Transferring the Weights and Assessing the ICs

Our goal in heavily filtering the data was to improve the ICA decomposition, but as was described in the chapter on filtering, this kind of filtering is usually a bad idea for ERP analyses. So, we're going to take our optimized ICA decomposition and transfer the weights to the original dataset, which has not been distorted by filtering. To do this, first look at the **Datasets** menu in EEGLAB and note the dataset number for the dataset containing the ICA weights (**10_MMN_preprocessed_filt_100Hz_del_ICAweights**). It's probably #4, but it might be something else if you've done some other processing. Now select the original dataset (**10_MMN_preprocessed**) in the **Datasets** menu and then select **EEGLAB > Edit > Dataset info**. In the new window that appears, click the **From other dataset** button for the **ICA weights array**, and enter the number for the dataset with the ICA weights (probably 4). Also change the **Dataset name** to **10_MMN_preprocessed_transferredICAweights**, and then click **OK**. That's it! You've now transferred the weights. You might want to save this dataset to your disk with **EEGLAB > Files > Save current dataset as**.

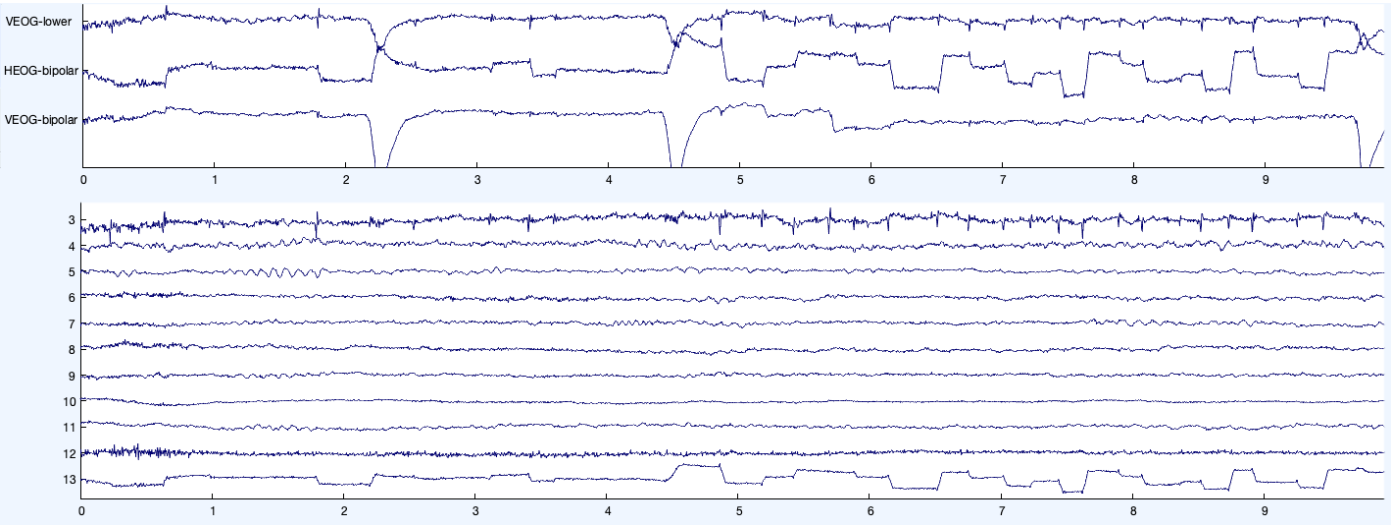
You can verify that the weights have been transferred by selecting **EEGLAB > Tools > Inspect/label components by map** (but first close the window you created with this routine in the previous exercise, if it's still open). You should see the same maps as in the previous exercise (Screenshot 9.6), because you transferred those weights to the current dataset.

Now let's compare the time course of the ICs (**EEGLAB > Plot > Component activations (scroll)**) with the time course of the EEG and EOG data (**EEGLAB > Plot > Channel data (scroll)**). You can see that the time course of IC 1 closely matches the time course of the blinks and vertical eye movements in VEOG-bipolar. To the naked eye, there isn't any obvious difference between IC 1 from this decomposition and IC 1 from our original decomposition (Screenshots 9.2-9.4). However, given that the decomposition as a whole improved, IC 1 from our new decomposition is less likely to be contaminated by activity from other sources (less lumping).

Now let's see if we can figure out what's going on with the other ICs and see if any of them should be removed along with IC 1. Let's start by looking for an IC related to horizontal eye movements. Go back to the plot of the scalp maps (Screenshot 9.6) and see if any of the maps have the distribution you'd expect for horizontal eye movements (i.e., opposite-polarity foci just to the sides of the two eyes). IC 13 looks promising, so click on the **13** above the scalp map for IC 13 (or the IC that has the right scalp distribution if your maps don't exactly match Screenshot 9.6). A new window will pop up to show the details of this IC. The time course heatmap has a lot of activity at the beginning and end of the dataset, which is what we would expect if large eye movements were most common during the break periods at the beginning and end of the session. (Note that we deleted the break periods in the dataset used for the ICA decomposition, but we've now transferred the weights to the original dataset, which still has data during the break periods. The same set of weights is used for every time point, so it's not a problem to transfer weights to time points that weren't used in the ICA decomposition.)

Now go back to the scrolling time course plots for the ICs and EEG data and see if the time course of IC 13 matches the time course of the horizontal eye movements in HEOG-bipolar. You might want to display only 6 channels and ICs at a time so that you can see the eye movements better. Screenshot 9.7 shows the EOG data (top) and a few of the ICs (bottom) for the first 10 seconds of the dataset. You can see a beautiful correspondence between IC 14 and HEOG-bipolar. And if you look closely at the other ICs, none of them show the same pattern. ICA has clearly done a good job of isolating the horizontal eye movements with IC 13. We should definitely consider removing IC 13 from the data.

Screenshot 9.7



Although none of the other ICs show the same pattern as the HEOG-bipolar channel, IC 3 has a sharp voltage spike at the onset time of each shift in eye position. This is the spike potential shown in Figure 9.1, which is the muscle contraction that causes the eyes to move. Given that the spike potential and the change in HEOG voltage are closely linked, you might wonder why they end up in different ICs. The answer is that ICA tries to find ICs that are maximally independent at a given time point. The HEOG signal is sustained, and the spike potential occurs only at the very beginning of the eye movement, so you often have a large HEOG voltage with no spike potential. And the spike potential slightly precedes the HEOG change, so you can have a large spike potential with HEOG. Consequently, they end up as different ICs. Note that the magnitude of the spike potential in the EEG/EOG recordings varies considerably across participants, so you don't always see a separate IC for it.

IC 4 has a midline centro-parietal focus and a broad scalp distribution, much like the P3b wave. However, the participants weren't doing a task in the MMN experiment, so we wouldn't expect to see the P3b. If you scroll through the component activations and EEG data, it's hard to see any obvious correspondence between IC 4 and the EEG signals. Moreover, the power spectrum doesn't reveal a clear peak, so it doesn't seem to be an oscillation. I really don't know what IC 4 is.

IC 5, on the other hand, shows a clear peak at 10 Hz (which you can see by clicking on the **5** above the scalp map for IC 5). Also, the scalp map has a strong focus at the occipital pole. Both of those are consistent with the classic alpha-band oscillation first described by Hans Berger in his original EEG study (Berger, 1929). You can see some beautiful alpha bursts in both IC 5 and the posterior EEG electrodes between 57 and 60 seconds.

IC 2 also has a clear peak in its power spectrum, but at approximately 7 Hz instead of 10 Hz. Its scalp map has a focus around Fz. Given this power spectrum and scalp topography, this is probably the commonly-observed *midfrontal theta* oscillation. You can see bursts of oscillatory activity in this IC at a variety of time points (e.g., 127-128 s, 219 s, 297-302 s, 401-403 s), and you can see corresponding bursts of oscillations in the far frontal channels. However, the oscillation bursts sometimes occur simultaneously in other ICs (e.g., at 127 s), and IC 2 also shows some transient (non-oscillating) activity, so I wouldn't try to use this IC to isolate midfrontal theta and then use it as a dependent variable.

Check out the rest of the ICs and see if you can figure out what they represent. You'll see that some contain alpha-band activity, like IC 5, but with somewhat different scalp topographies. This isn't surprising given that alpha can be observed over different cortical areas at different times. Many ICs have no obvious interpretation. IC 16 has a scalp distribution that is much like IC 1. Blinks often appear in two or even three ICs, so you should take a close look at any IC that has a blink-like scalp map.

If you scroll through the component activations and EEG/EOG data, you'll see that IC 16 exhibits a high-frequency burst at the beginning of each blink, superimposed on a negative dip. This could be the EMG activity from the muscles that produce the blink. However, the slower negative dip doesn't seem like EMG, so IC 16 may be mixing EMG and EOG signals. It's still blink-related, so we'll consider whether to exclude it in the next exercise.

Blink-related EOG activity is primarily a result of the eyelids sliding across the eyeballs, but there may also be a slight vertical rotation of the eyeballs that produces a voltage. These two effects have similar but slightly different scalp distributions, so they

may appear as separate ICs (especially when you have 60 or more channels and therefore 60 or more ICs). In theory, the two eyelids could start and stop moving at slightly different times, which might also produce separate ICs.

This page titled 9.6: Exercise- Transferring the Weights and Assessing the ICs is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.7: Exercise- Deciding Which ICs to Exclude

IC 1 is clearly capturing most of the blink activity, and we established in the previous chapter that blinks are both a major source of noise and a confound in the MMN experiment. It's therefore clear that we should exclude this IC when we reconstruct the EEG data from the ICs. But what about the other ICs?

To answer this question, we need to go back to the three issues described at the beginning of the previous chapter. First, does the artifactual activity create substantial noise that degrades our data quality? Second, does the artifactual activity vary across groups or conditions, creating a confound? Third, does the artifactual activity indicate a problematic change in the sensory input (e.g., because the direction of gaze has changed)? The third issue is mainly relevant for visual experiments, so we won't worry about it for the MMN.

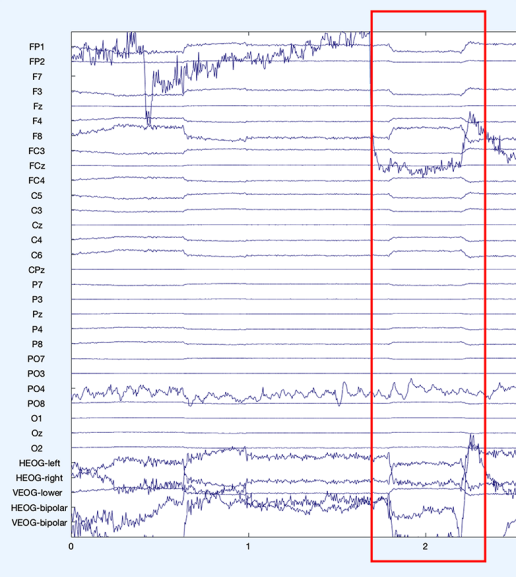
In the chapter on artifact rejection, we addressed the issue of noise by looking at whether the SME was improved by rejection. We addressed the issue of noise by looking at the averaged ERPs, especially when we inverted the artifact rejection process and included only the trials that were marked for rejection. We can use the same two approaches with artifact correction.

We already assessed the effects of blink rejection and correction on the SME for our first pass at the ICA decomposition earlier in this chapter, and we found that correction improved the SME. Now let's assess the effects of correcting for horizontal eye movements by excluding IC 13. Specifically, we'll compare excluding only blinks with excluding both blinks and horizontal eye movements. First, make two copies of the dataset with our improved ICA decomposition (**10_MMN_preprocessed_transferredICAweights**). Then apply **EEGLAB > Tools > Remove components from data** as in the earlier exercise to remove the blink IC from one dataset and to remove both the blink and horizontal eye movement ICs from the other dataset. You should then create averaged ERPs for these two datasets (which will require adding an EventList, running BINLISTER, and epoching the data). Make sure to specify a custom data quality window of 125-225 ms. Before averaging, use **EEGLAB > ERPLAB > Preprocess EEG > Selective Electrode Interpolation** to interpolate F7 and PO4 (Channels 3 and 24), excluding the bipolar EOG channels (32 and 33).

If you look at the SME values for FCz in the 125-225 ms time range, you'll see that eliminating the horizontal eye movement IC had virtually no impact on the data quality. This is consistent with what we saw with artifact rejection in the previous chapter. Horizontal eye movements just don't have much impact on midline electrode sites, and most of the eye movements occurred during the break periods.

We also need to consider whether the horizontal eye movements were a confound. With artifact rejection, we did this by making averages from only the trials marked with artifacts. Here, we'll conduct an analogous procedure, in which we reconstruct the data only from the eye movement component (IC 13) and then make averaged ERPs. To do this, select the dataset with the transferred weights (**10_MMN_preprocessed_transferredICAweights**), select **EEGLAB > Tools > Remove components from data**, leave the **List of component(s) to remove from data** field blank, and enter **13** into the field labeled **Or list of components to retain**. Name the resulting dataset **10_MMN_preprocessed_IC13only**. If you scroll through this dataset, you'll see how the eye movement potentials spread to the other scalp sites (see, e.g., the highlighted time period in Screenshot 9.8.). These potentials are largest at the lateral frontal electrodes but can also be seen at posterior scalp sites (e.g., P7 and P8). However, the positive and negative sides of the dipole largely cancel out at the midline sites (e.g., Fz, FCz, Cz).

Screenshot 9.8



Now go through the steps needed to create averaged ERPs from this dataset, and then plot the resulting ERPs. You should see that the EEG channels are essentially flat lines. (There is some activity in the bipolar channels, but these are the original waveforms, not reconstructed from IC 13.) From these flat ERP waveforms, which should solely reflect the horizontal eye movements, we can conclude that the horizontal eye movements were not a meaningful confound. In other words, the fact that the averaged ERPs were largely flat in the EEG channels indicates that any horizontal eye movements were approximately equally likely to be leftward and rightward, resulting in opposite-polarity EOG signals that canceled out in the averages. And the fact that there was no difference between standards and deviants, especially at the key FCz site, indicates that there were no meaningful differences in horizontal eye movements between standards and deviants that could confound our MMN results.

So, should we correct for the horizontal eye movements by excluding IC 13 when we reconstruct the EEG data? Probably not. There is little to be gained, and given that ICA is imperfect and IC 13 may contain brain activity mixed with the horizontal eye movements, we have more to lose than to gain. However, IC 13 didn't produce much activity in the ERPs when we looked only at this IC, so removing it will also have little impact. Before making a final decision about this, I would want to see how horizontal eye movements impact the data in the other participants. If they're generally not a problem, I would not correct for them. But if they seem problematic (in terms of data quality and/or confounding activity) in several of the participants, I would probably correct for them in all participants (for the sake of consistency).

We also need to consider whether to exclude IC 16, which has a blink-like scalp distribution and shows small deflections for each blink. Repeat the series of steps you conducted for IC 13 but use IC 16 instead. That is, assess the SME values after excluding both IC 16 and IC 1 with the SME values after excluding only IC 1, and then look at the averaged ERPs after reconstructing the data from only IC 16. When I did this, I found that excluding both IC 16 and IC 1 led to a tiny improvement in SME compared to excluding only IC 1, and I found a small but concerning difference between deviants and standards in FCz when the data were reconstructed from IC 16. Given the small confound in the averaged waveforms, and the fact that we've already established that blinks differ between deviants and standards, I would exclude IC 16 along with IC 1 in the final analyses.

You can also try this set of procedures with the component that represents the spike potential (IC 3).

Does this sound like a lot of work? It is! But if your goal is to publish a paper in a scientific journal, making a permanent contribution to the scientific literature that others can build on, it's worthwhile to take the time to deal with artifacts in a careful and thoughtful manner. Also, the scripts provided at the end of the chapter show you how to make the process more efficient by automating some parts of it.

This page titled 9.7: Exercise- Deciding Which ICs to Exclude is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.8: Exercise- Deleting C.R.A.P. Prior to ICA

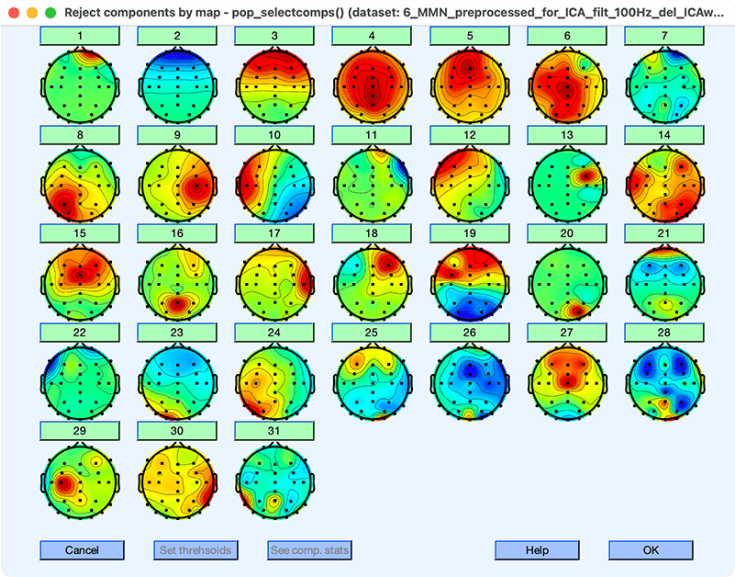
In this exercise, we're going to look at another type of large, idiosyncratic voltage deflection that should be eliminated prior to the ICA decomposition, namely huge C.R.A.P. In particular, we want to get rid of voltage deflections that are sufficiently large to impact the quality of the ICA decomposition but are sufficiently infrequent or irregular in their scalp distribution that they can't be well captured by a single IC.

The participant we've been working with so far, Subject 10, doesn't really have any huge C.R.A.P., so we're going to look at Subject 6 for this exercise. Quit and restart EEGLAB, and the open the **6_MMN_preprocessed_filt_100Hz_del** dataset. This dataset has already been preprocessed to improve the ICA decomposition, including filtering from 1-30 Hz, resampling at 100 Hz, and deletion of the break periods.

As usual, you should start by scrolling through the data to see what kinds of artifacts are present. In addition to the usual blinks and eye movements, you'll see some odd-looking voltage deflections at several time points, including 70, 72, 96, 254, 403, and 413 seconds. These deflections are somewhat like blinks, but much larger and with a different and variable scalp distribution. For example, the deflection at ~70 seconds is present in Fp2 but not Fp1, and the deflection at ~72 seconds is huge in VEOG-lower and smaller at Fp1 and Fp2, with no polarity inversion. I have no idea what caused these deflections, but they're large, rare, and have an inconsistent scalp distribution, so they're likely to mess up the ICA decomposition.

To see this, let's take a look at the ICA decomposition, which I did just as in the previous exercises (excluding the bipolar EOG signals in Channels 32 and 33). Load the dataset I created with the decomposition (**6_MMN_preprocessed_filt_100Hz_del_ICAweights**) and then look at the ICs using **EEGLAB > Tools > Inspect/label components by map**. The IC maps are shown in Screenshot 9.9. (You can do the decomposition yourself if you'd like, but your maps might look a little different.)

Screenshot 9.9



The first thing you should notice is that many of the maps are irregular, even in the top half of the ICs (e.g., ICs 6, 7, and 14). That's a strong hint that the decomposition did not work well. The next thing you should note is that IC 1 (which must account for a lot of variance, because it's the first IC) doesn't have a scalp map that corresponds to a typical artifact or brain signal, with a narrow focus at Fp2. Click on the **1** above the map for IC 1 to see its properties. The time course shows that it is strongly active during a few brief time periods and not at other times. Again, that's unusual.

Now look at IC 2 (and click on the **2** above the map to see its properties). It has the kind of scalp map we'd expect for blinks, and the time course contains the kind of distributed bursts that we'd expect for blinks. But note that the weights in the scalp map are

negative, whereas we saw positive weights at the frontal sites for blink-related ICs in our previous exercises. This is because the polarity of the IC maps is arbitrary, and a given IC may have either positive weights or negative weights. Just for fun, I repeated the decomposition a second time, and I found that IC 2 had positive weights in this repetition of the decomposition. The polarity of the weights in the maps is completely arbitrary, and the polarity of the activation values will also be reversed to come up with the right voltage polarity when we reconstruct the data. So, reversals of map polarity are not a problem.

To see this, scroll through the IC activations using **EEGLAB > Plot > Component activations (scroll)**. You'll see that IC 2 has negative deflections at the times of blinks, whereas we saw positive deflections for the blink IC in our previous examples (e.g., Screenshot 9.4). If we multiply the negative activation values by the negative weights for the frontal sites, we'll get a positive voltage. So, it doesn't matter if we have positive weights along with positive activations or negative weights along with negative activations. In either case, blinks will be reconstructed as a positive voltage at the frontal sites (but a negative voltage at VEOG-lower).

Another thing you should notice in the IC maps is that there isn't an IC with the scalp topography we'd expect for horizontal eye movements. There were many clear horizontal eye movements in the HEOG-bipolar channel when we scrolled through the data, but we haven't captured these eye movements as a distinct IC in this decomposition. That also indicates that it didn't work well. Some participants don't have many horizontal eye movements, and the lack of an IC for horizontal eye movements would not be a problem for those participants.)

Now let's try to improve the decomposition by deleting the time periods with the huge C.R.A.P. artifacts. Make **6_MMN_preprocessed_filt_100Hz_del** the active dataset and select **EEGLAB > ERPLAB > Preprocess EEG > Artifact rejection (continuous EEG)**. This routine is like the moving window peak-to-peak amplitude routine for artifact detection that we used in the previous chapter, but with two differences. First, it operates on continuous EEG rather than epoched EEG. Second, it eventually deletes sections of the data with artifacts rather than simply marking them.

In the window that appears for this routine, enter **500** as the threshold. We're not trying to reject blinks or other ordinary artifacts, so we need a much higher threshold for this routine than we would use for normal artifact detection. 500 is a good starting threshold, but you may need to adjust it for some participants. Enter **1000** for the moving window width and **500** as the step size. This will cause it to look for 1000-ms time periods in which the peak-to-peak voltage exceeds the threshold, shifting the window in 500-ms increments. Finally, enter **1:31** for the channels. We're going to exclude Channels 32 and 33 (the bipolar EOG channels) when we do the ICA decomposition, so we don't care about huge C.R.A.P. in these channels. (You might also want to exclude Fp1 and Fp2 if you have such large blinks in these channels that blinks end up exceeding the threshold for rejection.) Leave all the check boxes unchecked, and click **ACCEPT** to run the routine.

When it finishes, you should see in the Matlab command window that the routine has found 15 segments of data to reject. At this point, these segments have just been marked, but they will be deleted when we save the dataset. You should also see a window for scrolling through the marked dataset. If you scroll through the data, you'll see segments of data that are marked in yellow at ~70 and ~72 seconds (as well as at 96, 254, 403, and 413 seconds). These are the time periods where we saw the huge C.R.A.P., so the artifact rejection routine is working well.

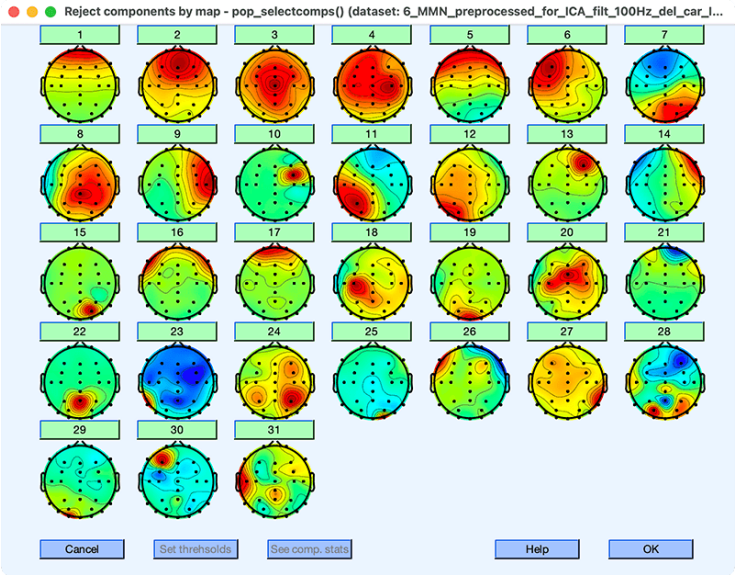
There's one thing that's a little odd, though, namely that we'll delete the data from ~70-71 seconds and from ~71.5-73 seconds, leaving just a 500 ms of unrejected data in the middle. The rejection routine has a feature for avoiding this kind of oddity. Let's give it a try. First, close the scrolling plot window and cancel the window for saving the dataset. Now select **EEGLAB > ERPLAB > Preprocess EEG > Artifact rejection (continuous EEG)** again. Keep the parameters the same, but check the box labeled **Join artifactual segments separated by less than** and put **1000** in the corresponding text box. This will cause any segments of <1000 ms between deleted segments to be deleted. Click **ACCEPT** to run the routine. Now you should see that a single continuous segment from ~70-73 ms now has been marked for deletion instead of two separate but nearby segments.

To actually delete the marked segments, go to the window that appeared for saving the new dataset and save it with the default name. Now go to **EEGLAB > Plot > Channel data (scroll)** and look at the result of the artifact rejection you just performed. If you look at the 5-second period starting at 68 seconds, you'll see that the artifactual activity at ~70 seconds is gone and has been replaced with a **boundary** event, which has been inserted to mark the discontinuity in the data that was produced by deleting the segment from ~70-73 seconds.

Note that you can also manually reject sections with huge C.R.A.P. using **EEGLAB > Tools > Inspect/reject data by eye**. ERPLAB's **Artifact rejection (continuous EEG)** routine is faster and more easily scriptable than the manual rejection routine, but sometimes it's useful to do it manually instead (or in addition). Just remember that your goal is to delete segments that will cause problems for ICA, not to delete segments with ordinary artifacts. That is, you want to delete segments with voltage deflections that are large and infrequent, especially if they have an inconsistent scalp distribution.

Now let's see how deleting the huge C.R.A.P. has impacted the ICA decomposition. You can see the resulting decomposition by loading the dataset named **6_MMN_preprocessed_filt_100Hz_del_car_ICAweights** and then selecting using **EEGLAB > Tools > Inspect/label components by map**. The scalp maps are shown in Screenshot 9.10.

Screenshot 9.10



The first thing to notice is that the scalp maps are more regular than those we obtained without deleting the huge C.R.A.P. (see Screenshot 9.9). There are still some irregular maps, but mainly in the bottom half, which don't account for much variance (e.g., ICs 20 and 23). The second thing to notice is that IC 1 is now a blink component. And the third thing to notice is that we now have an IC with the usual scalp map for horizontal eye movements (IC 14). So, by deleting segments with huge C.R.A.P., we obtained a much better ICA decomposition, even though we only deleted ~10 seconds worth of data.

The next step is to transfer the ICA weights to the original dataset (which are in a file named **6_MMN_preprocessed**). Then you should scroll through the data and component activations (simultaneously) to begin the process of determining which ICs to exclude from the reconstructed data.

Go ahead and remove IC 1 (**EEGLAB > Tools > Remove components from data**), and then scroll through the resulting dataset. Note that this dataset has the break periods at the beginning and end, so the huge C.R.A.P. artifacts are now about 10 seconds later than before.

You should see that the blink correction has done a good job of eliminating the voltages produced by ordinary blinks. However, the periods with huge C.R.A.P. artifacts are still present and still have large artifacts. You would therefore want to perform artifact detection on the epoched data to mark these epochs and then reject them in the averaging process.

This page titled 9.8: Exercise- Deleting C.R.A.P. Prior to ICA is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.9: General Recommendations

In this section, I summarize my recommendations for ICA-based artifact correction. These recommendations reflect my assessment of the current state of the art. It is difficult to determine the optimal approach to ICA analytically or with simulations, so these recommendations are mainly based on a combination of informal experience (my own and others') and empirical studies (which are often unsatisfying because ground truth is not usually known). As a result, these recommendations may change as more information accumulates, and other researchers may reasonably disagree with some of them.

Also, these recommendations are based largely on studies of compliant adult participants, and some changes may be necessary for other populations. So, use them as a starting point for designing your own artifact correction approach, not as a recipe to be followed blindly. And make sure that your approach is designed to achieve the fundamental goal of obtaining an accurate answer to the scientific question your study is designed to answer (which typically involves reducing noise and avoiding confounds from artifactual voltages and from changes in the sensory input).

Now that I'm done with the caveats, here are my recommendations:

- The correction should ordinarily be carried out on continuous data. If you have very long intertrial intervals, you can use epoched data, but the epochs must be at least 3 seconds long (e.g., -1000 to +2000 ms). No baseline correction should be applied (Groppe et al., 2009). There should be no overlap between consecutive epochs. Also, the epoching should be performed after the high-pass filtering step described later.
- The data should be referenced, and all of the channels included in the ICA decomposition should share the same reference. Any reasonable reference is fine, but extra steps are required if you use the average of all sites as the reference (see Makoto's Preprocessing Pipeline or EEGLAB's ICA documentation). Create additional bipolar channels for blinks and eye movements (but these will be left out of the decomposition).
- If you have large amounts of line noise, use the *cleanline* plugin to eliminate it (see Bigdely-Shamlo et al., 2015 for important details about implementing this tool). This is better than applying a low-pass filter (but a low-pass filter is sufficient for modest amounts of line noise).
- Scroll through the entire dataset to make sure you know what kinds of artifacts and other problems are present. Determine which channels should be interpolated.
- Once you have applied the above steps, make a separate copy of the dataset for use in the ICA decomposition. Apply the following steps to that copy, but not to the original data.
  - High-pass filter at 1 Hz (I recommend 48 dB/octave). If you will not be analyzing high-frequency activity (e.g., gamma-band oscillations), you should also apply a low-pass filter at 30 Hz (48 dB/octave).
  - Downsample the data to 100 Hz.
  - Delete time periods during breaks.
  - Delete segments of huge C.R.A.P.
  - Run the ICA decomposition with the **runica** algorithm, using the **'extended', 1** option. Leave out the bipolar channels and any channels that you plan to interpolate.
  - Verify that the ICs are reasonable (e.g., not too many irregular scalp maps, especially for the top half of the ICs). If they're not, take another look at the data and see if there are problems you missed.
  - When you are first starting out (or switching to a different kind of experimental paradigm or participant population), it's a good idea to repeat the decomposition and make sure that you get similar results each time. If you don't, then the decomposition is not working well. The RELICA plugin can be used to provide a quantitative assessment of the reliability of the decomposition.
  - If you can't get a good decomposition, check to make sure that you have enough data. The informal rule is that the number of time points in the dataset must be at least 20 x (# channels)^2 (assuming that your original data were sampled at ~250 Hz). If you don't have enough data, one option is to apply PCA first to reduce the dimensionality of your data. However, this can create significant problems (Artoni et al., 2018), and you shouldn't use it unless you have more than 128 channels.
- Transfer the ICA weights to the original version of the dataset (the version right before you made the copy).
- Evaluate the ICs with the following steps:
  - Examine the ICs carefully to make sure everything looks OK and to identify the key ICs. This includes looking at the scalp map, the frequency spectrum, and the time course heat map for each key IC, and then scrolling through the IC activations and voltages simultaneously to see what voltage changes co-occur with the key ICs.
  - Compute SME values corresponding to your planned data analysis (e.g., mean amplitude from 125-225 for the MMN) before versus after correcting for each IC to see if correction actually improves your data quality.

    - For each IC that you may want to remove, reconstruct your data using only that IC. Then average the data and see if that IC varies systematically across conditions (or groups in a between-subjects design, but that requires making grand averages across participants).
- Remove the ICs that correspond to clear, well-understood artifacts, have been well isolated by ICA, and are actually problematic (e.g., reduce your data quality or differ across conditions in the averaged ERPs).
- Interpolate the bad channels that you previously identified.
- After performing artifact correction, you should perform artifact detection on the epoched data. At a minimum, you want to eliminate epochs with C.R.A.P., because these are not handled well by artifact correction. In visual experiments, you should also mark and reject epochs with blinks and eye movements that might have interfered with the perception of the stimuli (e.g., between -200 and 400 ms). You can use the bipolar EOG channels for this because they were not corrected.
- As a final check, you can apply artifact detection/rejection to your data instead of artifact correction and then compare the grand averages from these two approaches. Ideally, the grand averages should be similar, but noisier for the rejection version than for the correction version (because fewer trials are available). If you see large differences between the rejected and corrected versions, this may indicate that the correction has reduced an important source of neural activity (because your artifact ICs contained a mixture of brain activity and artifacts) or that it has failed to fully correct for the artifacts.

I'd like to say a few words about how interpolation interacts with artifact correction. You don't want the "bad channels" to mess up the ICA decomposition, so these channels need to be excluded from the decomposition stage. You'll then perform the interpolation after the decomposition has been performed and the data have been corrected. That way, the interpolated channels will reflect the corrected data. My lab spent about 20 minutes one day talking through the different possible orders of steps for combining interpolation and correction, and this approach was clearly the best.

This page titled 9.9: General Recommendations is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.10: Matlab Scripts For This Chapter

I've created a set of three scripts that shows how to implement the artifact correction approach described in this chapter. There are three scripts, which you can find in the Chapter_9 folder. The script names begin with **MMN_artifact_correction_** and then end in **phase1**, **phase2**, or **phase3**. They divide the process into—you guessed it!—three phases.

- The first phase takes the original dataset and creates a copy that is optimized for the ICA decomposition by filtering from 1-30 Hz, downsampling to 100 Hz, deleting the break periods, and rejecting segments that contain huge C.R.A.P. artifacts.
- The second phase performs the ICA decomposition on this optimized dataset. I put that in a separate script because it takes quite a while to run.
- The third phase transfers the ICA weights to the original dataset, removes the artifactual ICs, and then interpolates any bad channels.

The script processes the data from Subjects 1–10. I spent an entire afternoon going through the data carefully, deciding which channels should be interpolated, determining which ICs to remove, etc. But even after devoting that much time, I did only a superficial job. You can probably do better if you spend more time.

While I was going through the data to determine the various parameters for a given participant, I put the parameters in a set of spreadsheets. The scripts then read from these spreadsheets. For the first two phases, I didn't actually run the scripts on all 10 participants at once. As noted in the comments inside the scripts, you can modify the scripts slightly to work on one participant at a time. I went back and forth between the scripts and the GUI to set the various parameters. You won't need to do that to run the scripts, because all the parameters are in the spreadsheets (which are provided in the Chapter_9 folder). However, you'll probably want to adopt this one-participant-at-a-time approach when you're analyzing your own data.

Note that the Excel spreadsheets contain a column for comments. These comments aren't used by the scripts, but they're very useful in helping you to remember why you made various decisions. Your future self will thank you.

This page titled 9.10: Matlab Scripts For This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

9.11: Key Takeaways and References

Key Takeaways

- Just as with artifact rejection, the overarching goal in designing an artifact correction strategy is to maximize the likelihood that you will obtain an accurate answer to the scientific question your study is designed to answer. This typically involves reducing noise to maximize statistical power, avoiding artifactual voltages, and avoiding unwanted changes in the sensory input.
- You can assess noise reduction by examining the SME before and after correction.
- You can assess confounds by reconstructing the data only with the artifactual ICs and looking for differences between conditions.
- Artifact correction cannot be used to avoid artifactual changes in the sensory input, and you will typically want to employ artifact detection and rejection for that purpose (in addition to artifact correction) in studies with visual stimuli.
- Some of the assumptions of ICA are known to be incorrect, and it is therefore imperfect. In practice, ICA works best for large and frequent artifacts, such as blinks. I therefore recommend a conservative correction strategy in which you remove only the small set of ICs that correspond with well-understood artifacts and only after you have established that they are actually problematic (i.e., reduce the data quality substantially and/or create confounds).

References

Artoni, F., Delorme, A., & Makeig, S. (2018). Applying dimension reduction to EEG data by Principal Component Analysis reduces the quality of its subsequent Independent Component decomposition. *NeuroImage*, *175*, 176–187. https://doi.org/10.1016/j.neuroimage.2018.03.016

Berger, H. (1929). Ueber das Elektrenkephalogramm des Menschen. *Archives Fur Psychiatrie Nervenkrankheiten*, *87*, 527–570.

Bigdely-Shamlo, N., Mullen, T., Kothe, C., Su, K.-M., & Robbins, K. A. (2015). The PREP pipeline: Standardized preprocessing for large-scale EEG analysis. *Frontiers in Neuroinformatics*, *9*. https://doi.org/10.3389/fninf.2015.00016

Box, G. E. P. (1976). Science and Statistics. *Journal of the American Statistical Association*, *71*(356), 791–799. https://doi.org/10.1080/01621459.1976.10480949

Dimigen, O. (2020). Optimizing the ICA-based removal of ocular EEG artifacts from free viewing experiments. *NeuroImage*, *207*, 116117. https://doi.org/10.1016/j.neuroimage.2019.116117

Drisdelle, B. L., Aubin, S., & Jolicoeur, P. (2017). Dealing with ocular artifacts on lateralized ERPs in studies of visual-spatial attention and memory: ICA correction versus epoch rejection. *Psychophysiology*, *54*(1), 83–99. https://doi.org/10.1111/psyp.12675

Groppe, D. M., Makeig, S., & Kutas, M. (2009). Identifying reliable independent components via split-half comparisons. *Neuroimage*, *45*, 1199–1211. https://doi.org/10.1016/j.neuroimage.2008.12.038

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Näätänen, R., & Kreegipuu, K. (2012). The mismatch negativity (MMN). In S. J. Luck & E. S. Kappenman (Eds.), *The Oxford Handbook of Event-Related Potential Components* (pp. 143–157). Oxford University Press.

This page titled 9.11: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10: SCORING AND STATISTICAL ANALYSIS OF ERP AMPLITUDES AND LATENCIES

Learning Objectives

In this chapter, you will learn to:

- Score the magnitude of an ERP component using the peak amplitude and the mean amplitude.
- Score the timing of an ERP component using the peak latency, fractional area latency, and fractional peak latency.
- Evaluate these different scoring methods in terms of measurement error, effect sizes, and bias.
- Conduct statistical analyses that minimize the likelihood of bogus-bug-significant results.
- Correlate ERP measures with behavioral measures.

This chapter focuses on the final steps of data analysis, in which you quantify the amplitudes and/or latencies of your ERPs and conduct a statistical analysis. It doesn't cover every possible way of scoring amplitudes and latencies, and it barely scratches the surface of the statistical analysis of ERP data. However, it covers the scoring procedures that are used most often (or that should be used most often), along with some very simple statistical analyses. Additional details about scoring can be found in Chapter 9 of Luck (2014), and a much more in-depth treatment of statistical analysis can be found in Chapter 10 of that book. In particular, I encourage you to read about the *jackknife* and *mass univariate* statistical approaches (which are too advanced for the present book).

One reason that I don't go too deeply into statistical analyses in this chapter is that ERPLAB doesn't include statistical functions, and I don't want to have to explain how to use some other statistical package. I'm assuming that you already know how to conduct basic statistical analyses (*t* tests and within-subjects ANOVAs) and have a statistical package that you can use to perform these analyses. If you don't, I recommend JASP (Love et al., 2019), which is free and easy to use. It's what I used for the analyses in this chapter.

The exercises in this chapter will examine the *lateralized readiness potential* (LRP), which reflects motor preparation. The data are from the ERP CORE flankers experiment. However, the lessons you will learn can be applied to almost any ERP component in almost any paradigm. And the LRP provides excellent opportunities to ask interesting questions about both amplitudes and latencies.

Quantifying amplitudes and latencies is often called the *measurement* process, and in ERPLAB it's done with the *Measurement Tool*. Recently, however, I've started using the term *scoring* instead of *measurement*. When we put electrodes on the scalp and record the EEG, that feels like we're actually measuring something (the voltages on the scalp). But applying an algorithm to an ERP waveform and hoping that it accurately captures the magnitude or timing of some underlying brain signal doesn't seem like taking a *measurement*. I now prefer the term *scoring* as more neutral term that is used in many other research areas. For example, you might score the amplitude or latency of a given ERP component.

10.1: Data for This Chapter

10.2: Design of the Flankers Experiment

10.3: Exercise- Examining the Grand Averages

10.4: Exercise- A First Pass at Scoring and Statistical Analysis

10.5: Exercise- Simplifying the Statistical Analysis

10.6: Exercise- Peak Amplitude

10.7: Exercise- Peak Latency

10.8: Exercise- Fractional Area Latency

10.9: Exercise- Quantifying Onset Latency

10.10: Exercise- Collapsing Across Channels and Correlating Latencies with Response Times
10.11: Matlab Scripts For This Chapter
10.12: Key Takeaways and References

This page titled 10: Scoring and Statistical Analysis of ERP Amplitudes and Latencies is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.1: Data for This Chapter

The data we'll be using for the exercises in this chapter can be found in the Chapter_10 folder in the master folder: https://doi.org/10.18115/D50056.

This page titled 10.1: Data for This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.2: Design of the Flankers Experiment

This section will provide a brief overview of the experimental design and main results from the ERP CORE flankers experiment. This experiment was designed to isolate two different ERP components, the lateralized readiness potential (LRP) and the error-related negativity (ERN).

The LRP is a negative voltage over the hemisphere contralateral to a hand that is being prepared for a response. For example if I were to tell you "raise your left pinkie," a negative voltage would be present over the motor cortex of the right hemisphere, preceding the actual pinkie response by 100-200 ms. The LRP is related to the preparation rather than the execution of the response. For example, if I were to tell you "raise your right pinkie when I say GO," the negative voltage over the right hemisphere would appear right away, even before I said "GO." Like the N2pc component, the LRP is isolated with a contralateral-minus-ipsilateral difference wave (relative to the side of the response hand). For an excellent review of the LRP, see Smulders and Miller (2012).

The ERN is a negative voltage with a maximum amplitude near FCz that occurs on error trials (usually in tasks where the error is obvious to the participant). It usually begins shortly before the actual buttonpress response and peaks 50-100 ms after the response. To get a good ERN, you need participants to make enough errors that you have a decent number of trials for averaging, but not so many errors that they're not really putting any effort into doing the task correctly. For an excellent review of the ERN, see Gehring et al. (2012).

In the ERP CORE, we used a version of the Eriksen flankers task (Eriksen, 1995) to elicit these components. In the present chapter, we'll focus on the LRP. As illustrated in Figure 10.1.A, each stimulus array contained a central target arrow that pointed leftward or rightward with equal probability, surrounded by two flanking arrows on each side. Participants were instructed to make a rapid buttonpress response for each array, pressing with the left index finger if the central arrow pointed leftward and with the right index finger if the central arrow pointed rightward. They were instructed to ignore the flankers, which pointed in the same direction as the central target arrow on 50% of trials (called *Compatible* trials) and pointed in the opposite direction on the other 50% (called *Incompatible* trials). Each array was presented for 200 ms, and successive arrays were separated by an interstimulus interval of 1200–1400 ms. To make sure we had a good number of error trials, participants were told to speed up if they were making errors on fewer than 10% of trials and to slow down if they were making errors on more than 20% of trials.

In the ERP CORE paper (Kappenman et al., 2021), we focused our LRP and ERN analyses on ERPs that were time-locked to the response (i.e., the response rather than the stimulus was used as time zero when the data were epoched and averaged). The LRP was isolated by examining trials with left-hand versus right-hand responses. Figure 10.1.B shows the grand average ERPs, with one waveform for the ipsilateral hemisphere (C3 for left-hand response trials averaged with C4 for right-hand response trials) and another waveform for the contralateral hemisphere (C4 for left-hand response trials averaged with C3 for right-hand response trials). The LRP is a negative voltage over the contralateral hemisphere that is superimposed on the other brain activity. To isolate the LRP, we construct a contralateral-minus-ipsilateral difference wave (Figure 10.1.C). You can see that the LRP in this difference wave starts to head in a negative direction approximately 120 ms prior to the response, and it peaks shortly before the response. (The slight positive voltage in the difference wave prior to -150 ms is likely an artifact of the 0.1 Hz high-pass filter that was applied to the continuous EEG at an early step in the analysis.)

The ERN is examined by comparing trials with correct trials and trials with incorrect trials (Figure 10.1.D). A sharp negative wave is superimposed on the positive voltage that would otherwise occur, peaking shortly after the response. This is then followed by a more positive voltage on the error trials than on the correct trials (called the error positivity or P_E). The ERN and P_E are often isolated from the other brain activity by making an error-minus-correct difference wave (Figure 10.1.E).

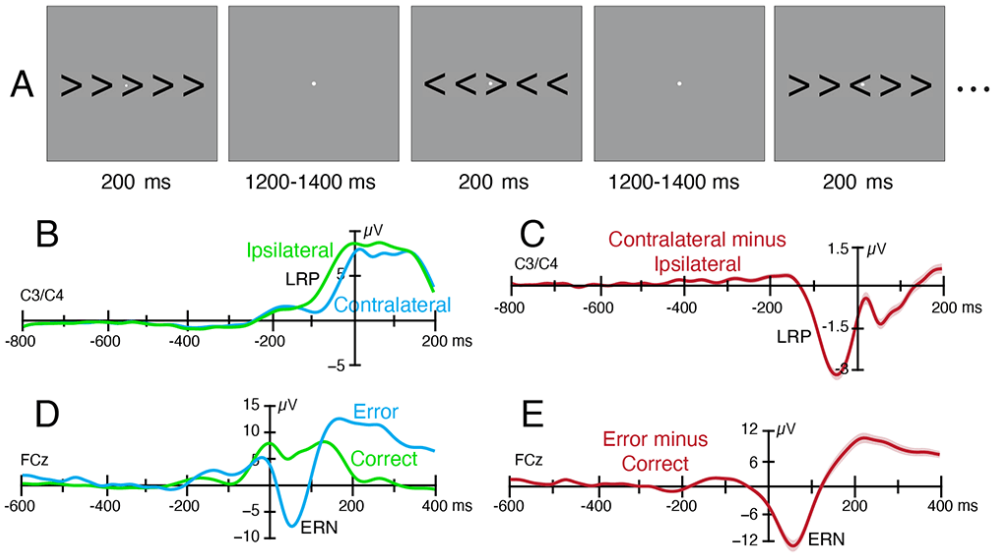


Figure 10.1. Experimental paradigm and results from the ERP CORE flankers experiment. (A) Example sequence of stimuli. Participants were instructed to press a left- or right-hand button depending on whether the central arrow pointed leftward or rightward, respectively. They were instructed to ignore the flanking arrows, which could be either compatible or incompatible with the direction of the central arrow. (B) Grand average ERP waveforms contralateral or ipsilateral to the response hand, time-locked to the response on correct trials, and averaged over compatible and incompatible trials and all 40 participants. (C) Grand average LRP difference wave, which was created by subtracting the ipsilateral ERP waveform from the contralateral ERP waveform. (D) Grand average ERP waveforms on correct trials and error trials, time-locked to the response, and averaged over compatible and incompatible trials and all 40 participants. (E) Grand average ERN difference wave, which was created by subtracting the ERP waveform for the correct trials from the ERP waveform for the error trials.

In the main LRP analyses in the ERP CORE paper (Kappenman et al., 2021), we collapsed across compatible and incompatible trials for the sake of simplicity. In the present chapter, we're going to look at the LRP separately for these trial types, focusing on stimulus-locked instead of response-locked averages. We'll consider only the correct trials. Many prior experiments have found that response times (RTs) are slowed on incompatible trials relative to compatible trials, and this is mainly because information about the flankers "leaks through" to response selection mechanisms. On compatible trials, this helps to activate the correct response. On incompatible trials, however, the incorrect response may be activated, which slows down the response (and often leads to errors). This can sometimes be observed in the LRP waveform, which may be contralateral to the incorrect hand briefly before becoming contralateral to the correct hand on incompatible trials (Gratton et al., 1988). In addition, the onset latency of the LRP is delayed on incompatible trials relative to compatible trials.

We didn't do any of these analyses for the ERP CORE paper, but I thought they would be interesting to do in the present chapter. For one thing, they'll give us an opportunity to look at several different measures of amplitude and latency. For another, we won't know the results until we do the analyses, so there will be some drama!

This page titled 10.2: Design of the Flankers Experiment is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.3: Exercise- Examining the Grand Averages

Ordinarily, you would decide exactly how to quantify and analyze the ERP amplitudes and latencies prior to seeing the data. If you first look at the grand average ERP waveforms, the analysis parameters you select will likely be influenced by noise in the data, and you'll have a high likelihood of finding significant effects that reflect noise rather than true effects and are completely bogus. This is a super important point! I'm not going to dwell on it here, because I've written about it extensively elsewhere (see especially Luck & Gaspelin, 2017). However, keep this point in mind throughout the chapter (especially in this first exercise, where we are going to look at the data before we develop our analysis plan—exactly what you shouldn't do!).

I've already created the averaged ERP waveforms for looking at the LRP. If you go to the **Chapter_10** folder, you'll see a subfolder named **Data**, and inside that subfolder you'll see another subfolder named **ERPsets** that contains an ERPset for each of the 40 participants. To create these ERPsets, I referenced the data to the average of P9 and P10, high-pass filtered at 0.1 Hz (12 dB/octave), and then applied ICA-based artifact correction for blinks and horizontal eye movements (using the optimized approach described in the chapter on artifact correction). The next step was to add an EventList and then run BINLISTER to create 4 bins:

- Bin 1: Left-Pointing Target with Compatible Flankers, Followed by Left Response
- Bin 2: Right-Pointing Target with Compatible Flankers, Followed by Right Response
- Bin 3: Left-Pointing Target with Incompatible Flankers, Followed by Left Response
- Bin 4: Right-Pointing Target with Incompatible Flankers, Followed by Right Response

Note that only correct responses were included in these bins because we're going to focus on the LRP rather than the ERN.

Next, I epoched the data from -200 to 800 ms relative to stimulus onset. I then performed artifact detection to mark trials with C.R.A.P., and finally I averaged the data, excluding the marked trials.

Let's load the data and make a grand average. Quit and restart EEGLAB, and set **Chapter_10** to be Matlab's current folder. Select **EEGLAB > ERPLAB > Load existing ERPset**, navigate to the **Chapter_10 > Data > ERPsets** folder, select all 40 ERPset files at once, and click **Open**. You should then be able to see all 40 ERPsets in the **ERPsets** menu. To make a grand average, select **EEGLAB > ERPLAB > Average across ERPsets (Grand Average)**, and indicate that the routine should average across ERPsets 1:40 in the ERPsets menu. All the other options should be kept at their default values. Click **RUN** and name the resulting ERPset **grand**. Save it as a file named **grand.erp**, because you'll need it for a later exercise. Now plot the ERPs (**EEGLAB > ERPLAB > Plot ERP > Plot ERP waveforms**), making one plot for Bins 1 and 2 (compatible trials) and another plot for Bins 3 and 4 (incompatible trials). Find the C3 and C4 channels (where the LRP is typically largest) and look for the contralateral negativity from ~200-400 ms.

The key waveforms are summarized in Figure 10.2.A. For the compatible trials, the voltage at C3 from ~200-400 ms is more negative on trials with a right-hand response than on trials with a left-hand response, and the voltage at C4 during this period is more negative on trials with a left-hand response than on trials with a right-hand response. The overall voltage is positive in this time range (because of the P3b component), and the LRP sums with the positive voltages to make the voltage more negative (less positive) over the contralateral hemisphere than over the ipsilateral hemisphere.

The pattern is a little more complicated for the incompatible trials. At approximately 200 ms, you can see an opposite-polarity effect, with a more negative voltage for left-hand than for right-hand responses at C3 and a more negative voltage for right-hand than for left-hand responses at C4. This then reverses beginning at approximately 250 ms.

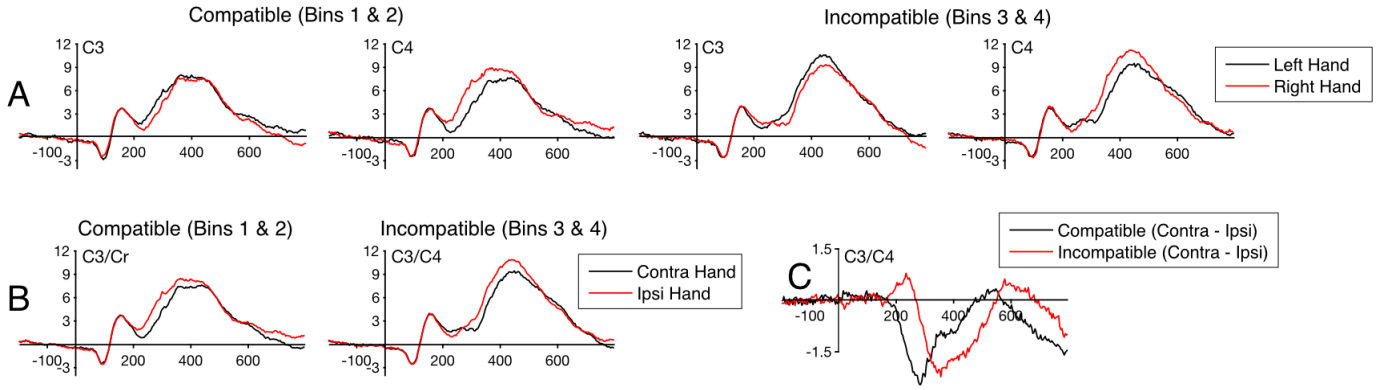


Figure 10.2. Grand averages from the new analysis of the ERP CORE flankers experiment, time-locked to stimulus onset. (A) Original parent waveforms. (B) Waveforms after collapsing into contralateral and ipsilateral hemispheres (relative to response hand). (C) Contralateral-minus-ipsilateral difference waves.

It's a little challenging to piece together everything that's happening with the waveforms shown in Figure 10.2.A. There are just a lot of waveforms to look at. We can simplify things by using ERP Bin Operations to collapse the data into contralateral waveforms (left-hemisphere waveforms on right-response trials averaged with right-hemisphere waveforms on left-response trials) and ipsilateral waveforms (left-hemisphere waveforms on left-response trials averaged with right-hemisphere waveforms on right-response trials).

Making these collapsed ERPsets is a little tricky (and is explained in the ERP Bin Operations section of the ERPLAB Manual). To save some of your time, I've already made these collapsed ERPsets for you. To open them, first clear out any existing ERPsets from ERPLAB using **EEGLAB > ERPLAB > Clear ERPset(s)**. You should have 41 ERPsets (which you can verify in the **ERPsets** menu), so enter **1:41** when asked which ERPsets to clear. Next, load the 40 ERPsets in the **Chapter_10 > Data > ERPsets_CI** folder. You can now make a grand average of these ERPsets and plot the results (with one plot for Bins 1 and 2, and a separate plot for Bins 3 and 4).

Figure 10.2.B shows the results from the C3 and C4 electrode sites (which are now combined, as indicated by the **C3/C4** label). Now we have only two pairs of waveforms rather than four pairs of waveforms, which makes it easier to see the contralateral negativity.

To make things even easier, and to isolate the LRP from other overlapping voltages, we can use ERP Bin Operations make a contralateral-minus-ipsilateral difference wave (Bin 1 minus Bin 2 for the compatible trials, and Bin 3 minus Bin 4 for the ipsilateral trials). I've already done this for you. Clear out the ERPsets and load these difference wave files from the **Chapter_10 > Data > ERPsets_CI_Diff** folder. Make a grand average, and plot the results (in a single plot with Bins 1 and 2).

Figure 10.2.C. shows the results at C3/C4. Now we have only one pair of waveforms, making it much easier to compare the LRP for the compatible and incompatible trials. On the compatible trials, you can see a nice clear negativity from ~200-500 ms. On the incompatible trials, you can see an initial contralateral positivity (which is really a negativity relative to the incorrect response), followed by a delayed contralateral negativity.

This is pretty cool! Our main goal in including both compatible and incompatible trials in this experiment was to generate a sufficient number of errors for the ERN analysis (because errors are a lot more common on incompatible trials). We didn't intend to do any comparisons of compatible and incompatible trials, so this is the first time anyone has looked at these effects. It's gratifying to see that we found the same pattern as in prior studies (e.g., Gratton et al., 1988), with a delayed LRP on incompatible trials that is preceded by an opposite-polarity deflection.

This page titled 10.3: Exercise- Examining the Grand Averages is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.4: Exercise- A First Pass at Scoring and Statistical Analysis

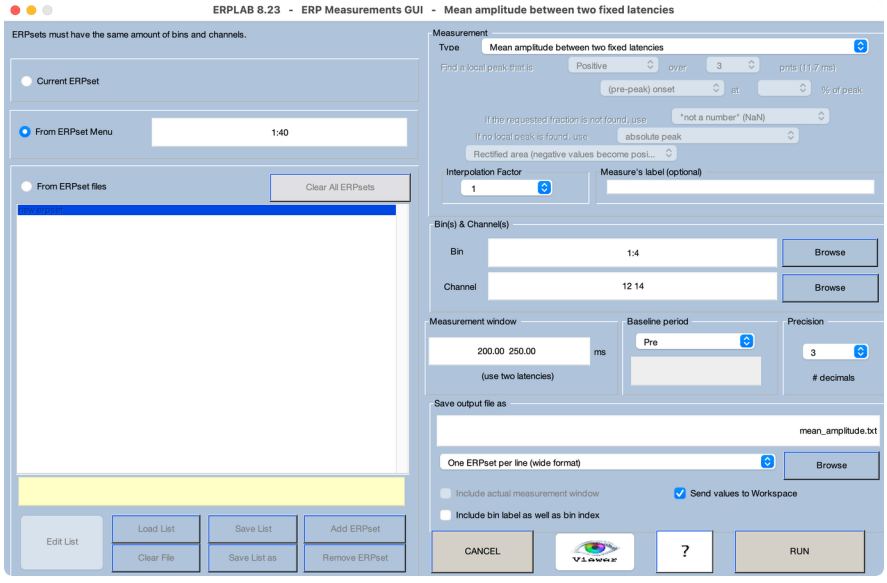
Now let's quantify these effects and then see if they're statistically significant. In this exercise, we'll take the approach that many studies (including my own) used many years ago but has become less common as people figured out an approach that is both simpler and better. We'll start with the outdated approach because that will allow you to better see (a) why the newer approach is better, and (b) how the newer approach provides equivalent information about the key questions.

Let's start by scoring the LRP amplitude from each participant's ERP waveforms. There are many different ways to score the amplitude or latency of an ERP component, as described in detail in Chapter 9 of Luck (2014). In most cases, *mean amplitude* is the best way to quantify ERP amplitudes. In the present exercise, we'll measure the mean amplitude from 200-250 ms. This just means that the scoring routine will sum together the voltage values for each time point in this latency range and then divide by the number of time points. It's that simple! The simplicity of mean amplitude means that it's very easy to understand and even make mathematical proofs about, and it has some very nice properties that we'll see as we go through the next few exercises.

One of the most important issues involved in scoring ERPs is the choice of the time window. I chose 200-250 ms for this exercise because this is the approximate time range in which the opposite-polarity effect for incompatible trials is typically seen.

Quit and restart EEGLAB, make sure that **Chapter_10** is Matlab's current folder, and then load all 40 ERPsets from the **Chapter_10 > Data > ERPsets** folder. Then select **EEGLAB > ERPLAB > ERP Measurement Tool** and enter the parameters shown in Screenshot 10.1. The left side of the GUI is used to indicate which ERPsets should be scored. We're going to measure from all 40 ERPsets that you just loaded. The right side of the GUI controls the scoring algorithm. You'll specify that the basic algorithm is **Mean amplitude between two fixed latencies**, and you'll indicate that the starting and stopping latencies are **200 250**. This is the *measurement window*. We're going to measure from the C3 and C4 channels (**12 14**) in all four bins (**1:4**). We're going to save the scores in a text file named **mean_amplitude.txt**.

Screenshot 10.1



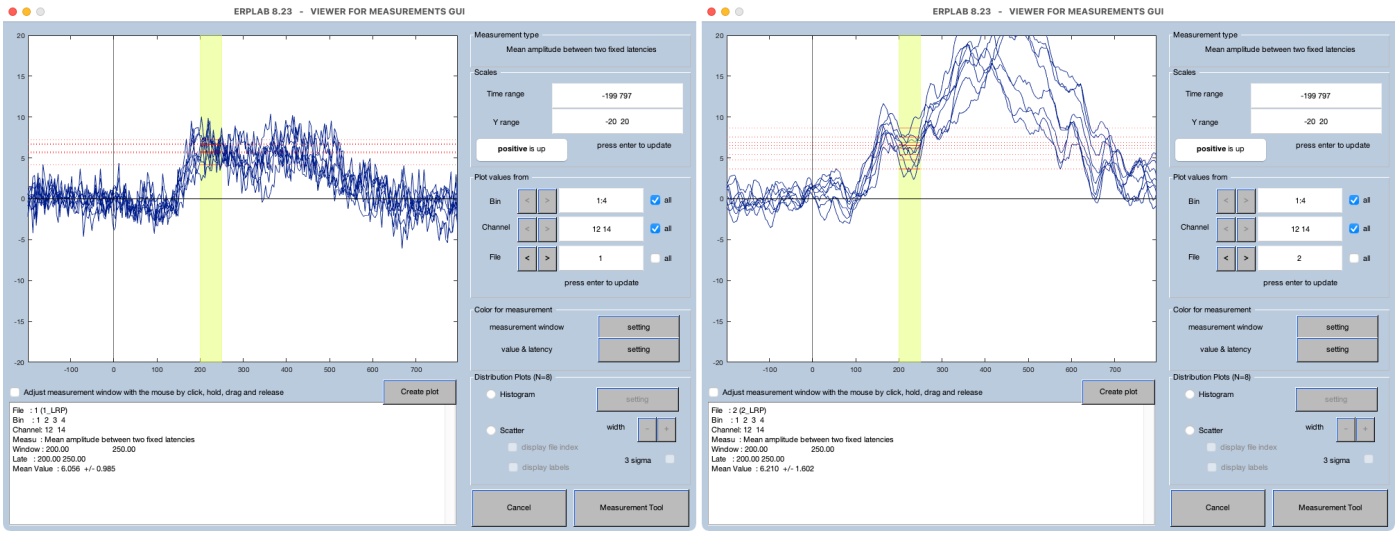
It's really tempting to hit the **RUN** button and get the scores, but you should always check the measurements against the ERP waveforms first. You can do this by clicking the **Viewer** button. The Viewer tool will open, and you'll see the ERP waveform from the first bin and first channel in the first ERPset. The measurement window is indicated by the yellow region, and the value produced by the scoring algorithm (5.628 μV) is shown in the window at the bottom.

You can then step through the different bins, channels, and ERPsets to verify that the algorithm is working sensibly. You may find

it convenient to look at multiple waveforms per screen. In the two cases shown in Screenshot 10.2, for example, I clicked the **all** box for **Bin** and **Channel** to overlay the two bins and the two channels.

Not much can go wrong with the algorithm for measuring mean amplitude, but you may find surprising and problematic scores for some participants when you use other algorithms (e.g., peak amplitude or peak latency). Even with mean amplitude, it's humbling and informative to see how much the ERP waveforms vary across participants. For example, the participant on the right in Screenshot 10.2 has waveforms that are similar to those in the grand average (Figure 10.2)—with distinct P2, N2, and P3 peaks—and the measurement window runs right through the N2 peak. By contrast, the participant on the left doesn't have very distinct peaks, and the measurement window is at the time of a positive peak.

Screenshot 10.2



This brings up an important point about ERPs (and most other methods used in the mind and brain sciences): Averages are a convenient fiction. The ERP waveforms we get by averaging together multiple single-trial epochs may not be a good representation of what happened on the single trials, and a grand average waveform across participants may not be a good representation of the individual participants. However, it is difficult to avoid averaging (or methods that are generalizations of the same underlying idea, such as regression). Chapters 2 and 8 in Luck (2014) discuss this issue in more detail.

Once you've finished scanning through all the ERP waveforms using the Viewer, click the **Measurement Tool** button to go back to the Measurement Tool, and then click **RUN** to get the scores. Assuming that **Chapter_10** is still the current folder in Matlab, a file named **mean_amplitude.txt** should now be present in the Chapter_10 folder. Double-click on this file in Matlab's **Current Folder** pane to open it in the Matlab text editor. You'll see that it consists of a set of tab-separated columns. Matlab's text editor doesn't handle the tabs very well, so the column headings may not line up properly. I recommend opening it instead in a spreadsheet program like Excel. Here's what the first few lines should look like:

bin1_C3	bin1_C4	bin2_C3	bin2_C4	bin3_C3	bin3_C4	bin4_C3	bin4_C4	ERPset
5.628	4.124	6.818	5.741	6.623	5.66	7.247	6.607	1_LRP
6.902	7.534	4.72	8.7	3.629	6.546	6.178	5.47	2_LRP
3.149	5.122	1.19	1.962	4.361	4.309	4.441	4.638	3_LRP

Each row contains the data from one participant, and each column holds the score (mean amplitude value) for a bin/channel combination for that participant. The Measurement Tool can also output the measurements in a "long" format in which each score is on a separate line. This long format is particularly good for using pivot tables to summarize the data in Excel, and it works well with some statistical packages. The "wide" format shown in the table above is ideal for statistical packages in which all the data for a given participant are expected to be in a single row (e.g., SPSS, JASP).

Now that we have the scores, let's do a statistical analysis using a traditional ANOVA. You can use any statistical package you like. As I mentioned earlier, I recommend JASP if you don't already have a package that can do basic *t* tests and within-subjects ANOVAs.

The ANOVA should have three within-subjects factors, each with two levels: Electrode Hemisphere (left or right), Response Hand (left or right), and Compatibility (Compatible or Incompatible). When you load the data into your statistical software and specify the variables, it's really easy to get the columns in the wrong order. Your first step in the statistical analysis should therefore be to examine the table or plot of the descriptive statistics provided by your statistical software so that you can make sure that the data were organized correctly. Figure 10.3 shows what I obtained in JASP.

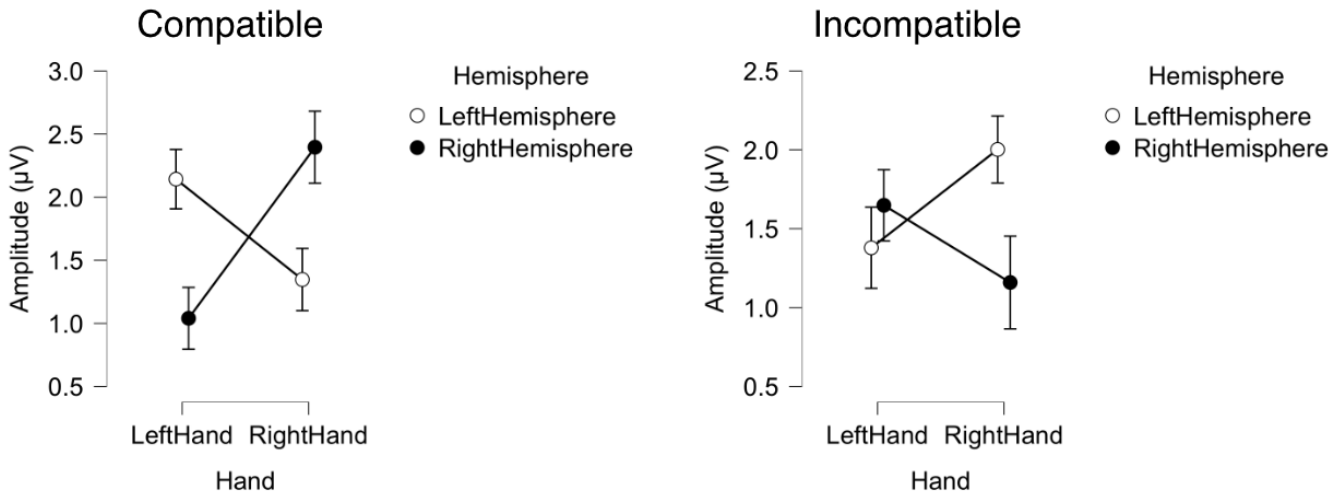


Figure 10.3. Means from each cell of the design, obtained with JASP. Error bars show the standard error of the mean in each cell.

But how do you know what the correct values should be? It turns out that with mean amplitude scores (but not most other scores), you get the same result by averaging the single-subject scores and by obtaining the scores from the grand average waveforms (see the Appendix in Luck, 2014 for details). Load the grand average you created earlier (**grand.erp**) and run the Measurement Tool again, but specifying that it should measure only from this ERPset and save the results in a file named **mean_amplitude_grand.txt**. You can then compare those numbers to the values in the table or figure of descriptive statistics. Here are the values I obtained:

bin1_C3	bin1_C4	bin2_C3	bin2_C4	bin3_C3	bin3_C4	bin4_C3	bin4_C4	ERPset
2.143	1.04	1.348	2.396	1.379	1.649	2.002	1.159	grand

These values exactly match the means shown in Figure 10.3. Success! Note that if you use some other scoring algorithm (e.g., peak amplitude) in your own studies, the values won't match exactly. However, you can at least make sure that the pattern is the same.

This verification process is very *very* **very** important! I estimate that you will find an error at least 10% of the time if you have three or more factors in your design.

Before we look at the inferential statistics, let's think about what main effects and interactions we would expect to see. First consider the Compatible condition, in which the voltage should be more negative for the contralateral hemisphere than for the

ipsilateral hemisphere. This gives us a more negative voltage for left-hand responses than for right-hand responses over the right hemisphere, and the reverse pattern over the left hemisphere. In other words, the presence of the LRP is captured in the ANOVA as an interaction between Hemisphere and Hand. During this 200-250 ms time period, we expect to see an opposite effect for the Incompatible trials (because the voltage is more negative over the hemisphere contralateral to the incorrect response, which makes it more positive contralateral to the correct response). Consequently, the difference between the Compatible and Incompatible trials should lead to a three-way interaction between Compatibility, Hemisphere, and Hand.

Table 10.1 shows the inferential statistics I obtained from JASP. You can see that the main effects of Hand and Hemisphere are not significant, consistent with the fact that Figure 10.3 shows little or no overall difference between left-hand and right-hand responses or between the left and right hemispheres. The main effect of Compatibility is also not significant, consistent with the fact that the average voltage across cells for the Compatible condition was about the same as the average voltage across cells for the Incompatible condition.

By contrast, the interaction between Hemisphere and Hand was significant. This interaction is equivalent to asking about the contralaterality of the voltage if we averaged across Compatible and Incompatible trials. These two conditions yielded opposite-direction effects that partially cancel each other out. However, the contralateral negativity for the Compatible trials was larger than the contralateral positivity for the Incompatible trials, and this gives us an overall significant interaction. But this interaction is meaningless at best and misleading at worst, because the patterns were opposite for the Compatible and Incompatible trials, as indicated by the significant three-way interaction between Hemisphere, Hand, and Compatibility. This kind of complication is one of the reasons why many researchers have stopped using this approach and have shifted to the simpler approach described in the next exercise.

Table 10.1. Inferential statistics from JASP.

Within Subjects Effects

Cases	Sum of Squares	df	Mean Square	F	p
Hemisphere	1.970	1	1.970	0.380	0.541
Residuals	202.060	39	5.181		
Hand	2.414	1	2.414	1.612	0.212
Residuals	58.404	39	1.498		
Compatibility	2.715	1	2.715	0.517	0.477
Residuals	204.961	39	5.255		
Hemisphere ✻ Hand	5.388	1	5.388	8.224	0.007
Residuals	25.552	39	0.655		
Hemisphere ✻ Compatibility	1.349	1	1.349	3.193	0.082
Residuals	16.475	39	0.422		
Hand ✻ Compatibility	0.911	1	0.911	0.289	0.594
Residuals	123.078	39	3.156		
Hemisphere ✻ Hand ✻ Compatibility	53.246	1	53.246	34.623	< .001
Residuals	59.977	39	1.538		

The next step in our statistical analysis would be to perform specific contrasts so that we can see, for example, if the Hemisphere x Hand interaction is significant when the Compatible and Incompatible trials are analyzed separately. However, we're not going to take that next step, because this way of analyzing the data is less than ideal. First, the size of the LRP is captured by the Hemisphere x Hand interaction rather than a main effect, which makes things difficult to understand. Second, this approach generates a lot of *p* values, which means that the probability that we obtain one or more bogus-but-significant effects is quite high. If you run a three-way ANOVA, you get 7 *p* values (as shown in Table 10.1), and you'll have about a 30% chance of getting at least one bogus-but-significant effect (if the null hypothesis is actually true for all 7 effects). So, it's important to minimize the number of factors in your analyses (see Luck & Gaspelin, 2017 for a detailed discussion of this issue). The next exercise will show you a better approach.

Bogus Effects

When an effect in the data is just a result of random variation and does not reflect a true effect in the population, I like to refer to that effect as *bogus*. And if the effect is statistically significant, I refer to it as a *bogus-but-significant* effect. The technical term for this is a *Type I error*. But that's a dry, abstract, and hard-to-remember way of describing an incorrect conclusion that might be permanently etched into the scientific literature.

This page titled 10.4: Exercise- A First Pass at Scoring and Statistical Analysis is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.5: Exercise- Simplifying the Statistical Analysis

Most recent studies of the LRP (and other lateralized components, such as N2pc and CDA) obtain amplitude and latency scores from contralateral-minus-ipsilateral difference waves (like those shown in Figure 10.2.C). This has two advantages. First, it isolates the component of interest from all of the overlapping ERP components. Second, it reduces the number of factors in the statistical analysis.

Let's see how it works. Clear out all the ERPsets from ERPLAB, and load the 40 ERPsets in the **Chapter_10 > Data > ERPsets_CI_Diff** folder. Then launch the **Measurement Tool**. Set it to measure from the 40 files in the ERPsets menu, using Bins 1 and 2 (Compatible and Incompatible) and Channel 5 (C3/C4). Specify **mean_amplitude_CI_Diff.txt** as the name of the output file. Click **Viewer** and make sure that the measurements look reasonable given the waveforms. Click **Measurement Tool** to go back to the Measurement Tool, and then click **RUN** to obtain the scores. You'll see that we now have only two scores per participant, one for the Compatible condition and one for the Incompatible condition.

Now let's do a statistical analysis on these scores. Start by doing a paired t test comparing the Compatible and Incompatible conditions. You should get a statistically significant *t* value of -5.885 (or +5.885, depending on which condition came first). And here's something very important: This *t* test is exactly equivalent to the three-way interaction between Hemisphere, Hand, and Compatibility in the three-way ANOVA from the previous exercise. That is, it tests exactly the same null hypothesis, and it yields exactly the same *p* value (except for possible rounding error). The *t* value for this paired *t* test corresponds exactly to the *F* value from the three-way interaction once you realize that *F* is the same as t^2. If we square 5.885, we get 34.63, which is the same (except for rounding error) as the *F* value for the three-way interaction in Table 10.1. So, a *t* test on difference scores can be a simpler and more convenient way of testing for an interaction, and it doesn't cause a proliferation of *p* values.

Fixing a Problem

When I tried to load the data into JASP, all the scores ended up in a single column. After spending a few minutes trying to figure out what was causing this, I realized that the problem was that the column labels in the first row had commas in them, which confused JASP. When I replaced the commas with underscores, everything worked fine.

Now let's ask whether we have a statistically significant negativity for the Compatible condition and a statistically significant positivity for the Incompatible condition. This just involves performing separate one-sample *t* tests for each of these conditions (comparing the means to zero). When I did that, I obtained a significant negativity for the Compatible condition ($t(39) = -6.107$, $p < .001$) and a significant positivity for the Incompatible condition ($t(39) = 3.605$, $p < .001$).

Isn't this approach a lot simpler and more direct than a three-way ANOVA followed by a bunch of contrasts? We really have three primary hypotheses, and with this approach we have one simple test for each of them.

This page titled 10.5: Exercise- Simplifying the Statistical Analysis is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

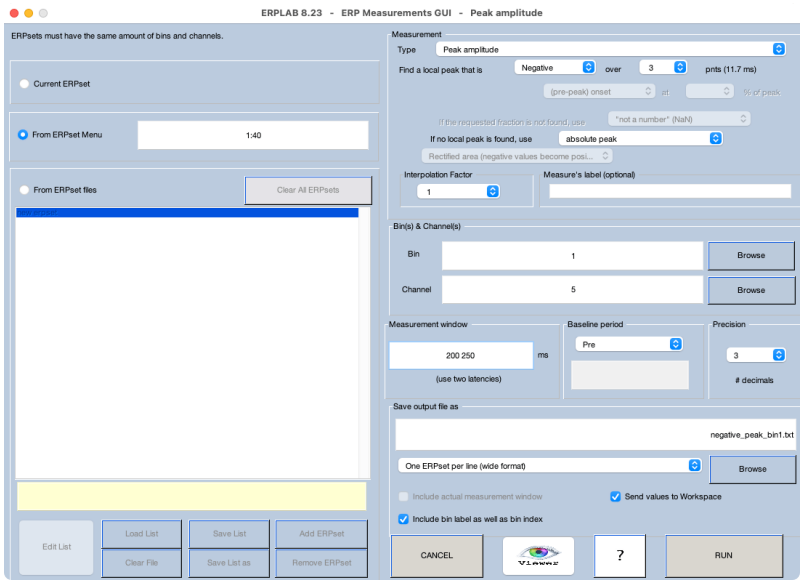
10.6: Exercise- Peak Amplitude

For the first couple decades of ERP research, the primary way of scoring ERP amplitudes was to find the peak voltage during the measurement window (either the most positive voltage for a positive peak or the most negative voltage for a negative peak). This approach was used initially because ERPs were processed using primitive computers that created a printout of the waveform, and researchers could easily determine the peak amplitude from the printout using a ruler (see Donchin & Heffley, 1978). This tradition persisted long after more sophisticated computers and software were available, but in many ways the peak voltage is a terrible way of scoring the amplitude of an ERP component. Mean amplitude is almost always superior. I provide a long list of the shortcomings of peak amplitude and the benefits of mean amplitude in Chapter 9 of Luck (2014). More generally, peaks are highly overrated in ERP research. Why should we care when the voltage reaches a maximum? Chapter 2 of Luck (2014) explains why peaks can be very misleading, even when they're measured well. Mean amplitude is now much more common than peak amplitude in most ERP research areas, but there are some areas where peak amplitude is still common.

In this exercise, we'll repeat the analyses from the previous exercise except that we'll measure peak amplitude instead of mean amplitude. And then you'll see for yourself some of the shortcomings of peak amplitude.

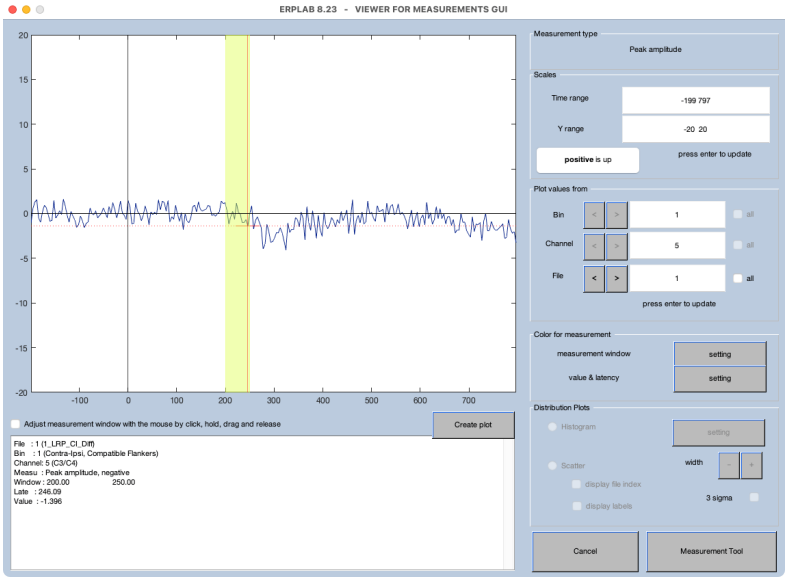
Make sure that the 40 ERPsets from the previous exercise (from the **Chapter_10 > Data > ERPsets_CI_Diff** folder) are loaded. Launch the Measurement Tool, and set it up as shown in Screenshot 10.3. As in the previous exercise, we want to see if there is a contralateral negativity for the Compatible condition and a contralateral positivity for the Incompatible condition. We therefore need to look for a negative peak for Bin 1 and a positive peak for Bin 2. This will take two steps. Screenshot 10.3 is set up for finding the negative peak in Bin 1. (Technically, we'll find the local peak, defined in this example as the most negative point that is also more negative than the 3 points on either side; for details, see in Chapter 9 in Luck, 2014).

Screenshot 10.3



Once you have all the parameters set, click the **Viewer** button to verify that everything is working as intended. You should immediately see a problem: As shown in Screenshot 10.4, the peak of the LRP falls outside the measurement window for the first participant. And this isn't an isolated incident; you'll see the same problem for the 2nd and 3rd participants (and many others as well). This makes sense if you look at the grand average waveforms in Figure 10.2.C. In general, you need a wider window to find peaks than you need for mean amplitude.

Screenshot 10.4



To fix this, click the **Measurement Tool** button in the Viewer tool, and then change the **Measurement Window** to **150 400**. Then click the **Viewer** button to see the waveforms again. You should see that the algorithm is now correctly finding the peak for every participant who has a clear peak. Go back to the Measurement Tool and click **RUN** to save the scores to a file named **negative_peak_bin1.txt**.

Now repeat the measurement for the positive peak in Bin 2. Leave the window at **150 400**, but change **Negative** to **Positive**, change the bin from **1** to **2**, and change the filename to **positive_peak_bin2.txt**. Make sure everything looks okay in the Viewer and then click **RUN** to save the scores.

Now perform the same *t* tests as you did in the previous exercise on these peak amplitude values (which may first require combining the scores into a single spreadsheet). You should see that the mean across participants is -3.23 μV for the Compatible condition and +1.46 μV for the Incompatible condition and that the difference between conditions is significant ($t(39) = -16.34$, $p < .001$). Also, the mean across participants is significantly less than zero for the Compatible condition ($t(39) = -18.61$, $p < .001$) and significantly greater than zero for the Incompatible condition ($t(39) = 8.71$, $p < .001$).

But this is a completely invalid way of analyzing these data! First, the positive peak for the Incompatible trials is much earlier than the negative peak for the Compatible trials, and it doesn't usually make sense to compare voltages at different time points. Second, peak amplitude is a biased measure that will tend to be greater than zero for positive peaks and less than zero for negative peaks even if there is only noise in the data.

To see this bias, let's repeat the analyses, but with a measurement window of **-100 0** (i.e., the last 100 ms of the prestimulus baseline period). There shouldn't be any real differences prior to the onset of the stimuli, and any differences we see must be a result of noise. To see this, find the negative peak between -100 and 0 ms for Bin 1 and the positive peak between -200 and 0 ms for Bin 2. Then repeat the *t* tests with these scores.

You should see that the mean across participants is -0.89 μV for the Compatible condition and +1.04 μV for the Incompatible condition. You should also see that the difference between conditions is significant ($t(39) = -13.27$, $p < .001$). Also, the mean across participants is significantly less than zero for the Compatible condition ($t(39) = -12.50$, $p < .001$) and significantly greater than zero for the Incompatible condition ($t(39) = 11.30$, $p < .001$). Thus, we get large and significant differences in peak amplitude between conditions during the baseline period, and each condition is significantly different from zero, even though there is only noise during this period. These are bogus-but-significant effects that a result of the fact that peak amplitude is a biased measure.

I hope it is clear why this happened. If you look at the baseline period of the single-participant waveforms with the Viewer, you'll see that the noise in the baseline is typically positive at some time points and negative at others. That's what you'd expect for random variations in voltage. If we take the most positive point in the period from -100 to 0 ms, it will almost always be greater than zero. If we take the most negative point in this period, it will almost always be less than zero. So, noise alone will tend to give

us a difference in amplitude between the positive peak and the negative peak, and it will tend to make the positive peak greater than zero and the negative peak less than zero.

It should be clear that it is not ordinarily legitimate to compare a positive peak with a negative peak (because noise alone will cause a difference). And it should also be clear that it is not ordinarily legitimate to test whether an effect is present by comparing a peak voltage to zero (because noise will cause a non-zero voltage).

A related point (which is not shown directly by this example) is that the peaks will tend to be larger when the noise level is higher. This means that it is not ordinarily legitimate to compare peak amplitudes for two conditions that differ in noise level (e.g., standards and deviants in an oddball paradigm), because the averaged ERP waveforms will be noisier for the deviants owing to a smaller number of trials. This can be solved by equating the number of trials in the averaged ERPs for each condition, but that requires throwing away a large number of trials from the more frequent condition. Also, there may be other systematic sources of noise. For example, some electrode sites are noisier than others (because they are closer to EMG sources), and some groups of participants are noisier than others (e.g., patient waveforms are often noisier than control waveforms).

The bottom line is that the peak voltage is not usually the best way to quantify the amplitude of an ERP component. Mean amplitude is much better in the vast majority of cases.

This page titled 10.6: Exercise- Peak Amplitude is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

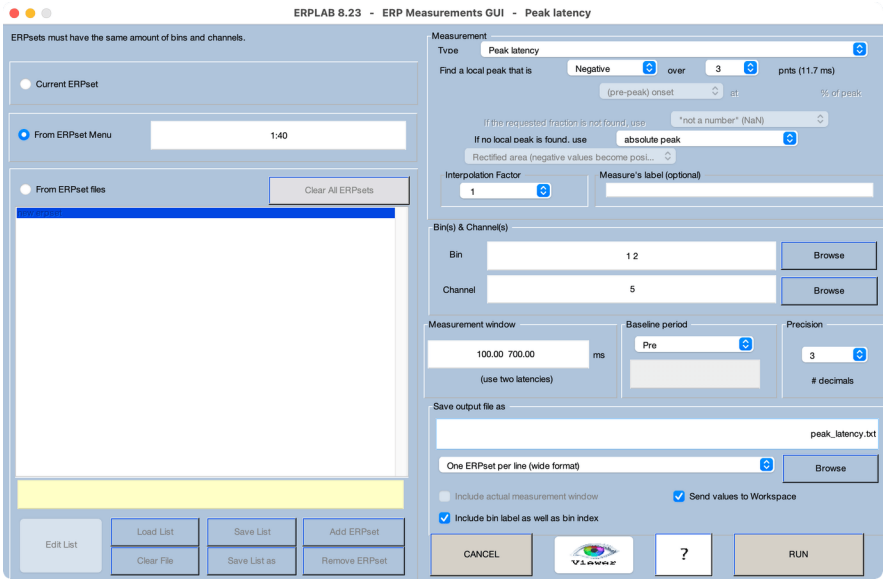
10.7: Exercise- Peak Latency

Now we're going to switch from scoring amplitudes to scoring latencies. The traditional method for scoring latencies is to find the peak voltage in the measurement window (positive or negative) and record the latency at which this peak occurred. Like peak amplitude, peak latency is not usually the best scoring algorithm (see Chapter 9 in Luck, 2014). We'll look at some better alternatives in the following exercises.

In the present exercise, we're going to ask whether the peak latency of the LRP in the contralateral-minus-ipsilateral difference wave is later on incompatible trials than on compatible trials. Make sure that the 40 ERPsets from the previous exercise (from the **Chapter_10 > Data > ERPsets_CI_Diff** folder) are loaded. Launch the Measurement Tool and set it up as shown in Screenshot 10.5. The measurement **Type** is **Peak latency**, and we're looking for a negative peak. We're measuring from Bins 1 and 2 (Compatible and Incompatible) in the C3/C4 channel, and we're saving the scores in a file named **peak_latency.txt**.

A key question in scoring ERP amplitudes and latencies is how to determine the time window. This is a complicated question, and you can read about several strategies in Chapter 9 of Luck (2014) and in Luck and Gaspelin (2017). As mentioned earlier, the most important thing is to avoid being biased by the data, which is best achieved by deciding on the measurement windows before you start the study. Of course, it's too late for that now with the ERP CORE experiments. However, if I were to choose a time window in advance for the LRP in a flankers paradigm, I'd assume that the LRP begins after 100 ms and ends by 700 ms. For this reason, we'll use a measurement window of 100 to 700 ms in this exercise.

Screenshot 10.5



As always, the next step is to click the **Viewer** button to see how well the algorithm is working. You'll see that it has mixed success. It works reasonably well for waveforms that are clean and contain a large peak (e.g., File 2), but the scores are distorted by high-frequency noise (e.g., Files 1 and 12), and the values are largely random for waveforms without a distinct peak (e.g., Files 9 and 10).

Now go back to the Measurement Tool and click **RUN** to save the scores. Load the data into your statistics package and perform a paired *t* test to compare the Compatible and Incompatible conditions. Verify that the means provided by the statistics package are reasonable. You should see a mean of 318 ms for Compatible and 375 ms for Incompatible. Unfortunately, the trick we used with mean amplitude—comparing the means from the statistical package with the values measured from the grand average—doesn't work with peak latency. If you measure the peak latency directly from the grand average ERP waveforms, you'll see a value of 285 ms for Compatible and 355 ms for Incompatible. The values from the grand average aren't the same as the mean of the single-subject values, but at least they show the same ordering (Compatible < Incompatible).

Now look at the actual *t* test results. You should see that the peak latency was significantly shorter for Compatible trials than for

Incompatible trials ($t(39) = -3.647$, $p < .001$). Given the huge differences between Compatible and Incompatible trials in the grand average waveforms (Figure 10.2.C), it's not surprising that the difference in peak latency was significant, even if peak latency isn't an ideal scoring algorithm. You should also look at the effect size, measured as Cohen's d_z, which indicates how far apart the means are relative to the pooled standard deviation. You should see an effect size of -0.577 (or +0.577, depending on the order of conditions in your analysis), which is a medium effect size.

If you're familiar with effect sizes in ERP studies, you might be surprised that this effect size isn't bigger. After all, the peaks in the grand averages are very far apart in time. It therefore seems reasonable to suppose that we had a lot of measurement error when we computed the peak latency, which increased the standard deviation of the scores and therefore reduced the effect size. Given that peak latency scores are distorted by high-frequency noise, we should be able to reduce the measurement error and increase the effect size by applying a low-pass filter to the averaged ERPs prior to obtaining the peak latency scores.

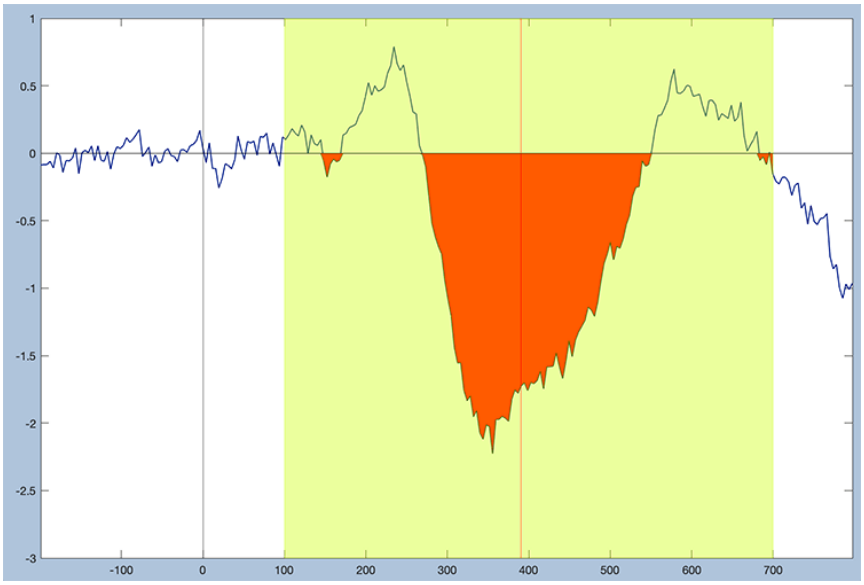
Let's try it. It would take quite a while for you to filter all 40 of the ERPsets using the GUI, so I've provided the ERPsets for you in **Chapter_10 > Data > ERPsets_CI_Diff_filt**. They've been low-pass filtered with a half-amplitude cutoff of 20 Hz and a slope of 12 dB/octave. Clear out the existing ERPsets from ERPLAB, load the filtered ERPsets, and repeat the measurement and analysis procedure (but changing the name of the measurement file to **peak_latency_filt.txt**). You'll see that the effect size is now a little larger ($d = -0.630$). So, filtering helped, but only a little. Sometimes it helps a lot, especially when there is a lot of high-frequency noise in the data (which is not true for most of the waveforms in this experiment).

This page titled 10.7: Exercise- Peak Latency is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.8: Exercise- Fractional Area Latency

In this exercise, we're going to look at a different latency scoring algorithm called *fractional area latency*, which is often superior to peak latency (especially when measured from difference waves). For a negative component like the LRP, this algorithm calculates the area of the waveform below the zero line and then finds the time point that divides the area into two areas at a particular percentage. If you want to estimate the *midpoint* of the waveform, you will look for the 50% point (the time that divides the area into two equal halves). This is then called the *50% area latency* (see Chapter 9 in Luck, 2014, for more details). Screenshot 10.6 shows what it looks like when I apply this algorithm to Bin 2 (Incompatible) from the grand average, using a measurement window of 100 to 700 ms. The area under the curve in this measurement window is shaded in red, and the point that divides this area into two equal halves is indicated by the red vertical line. This region includes some little areas near the beginning and end of the measurement window, but that's just how it goes. It's difficult to perfectly quantify ERP amplitudes and latencies, and we have to live with some error.

Screenshot 10.6

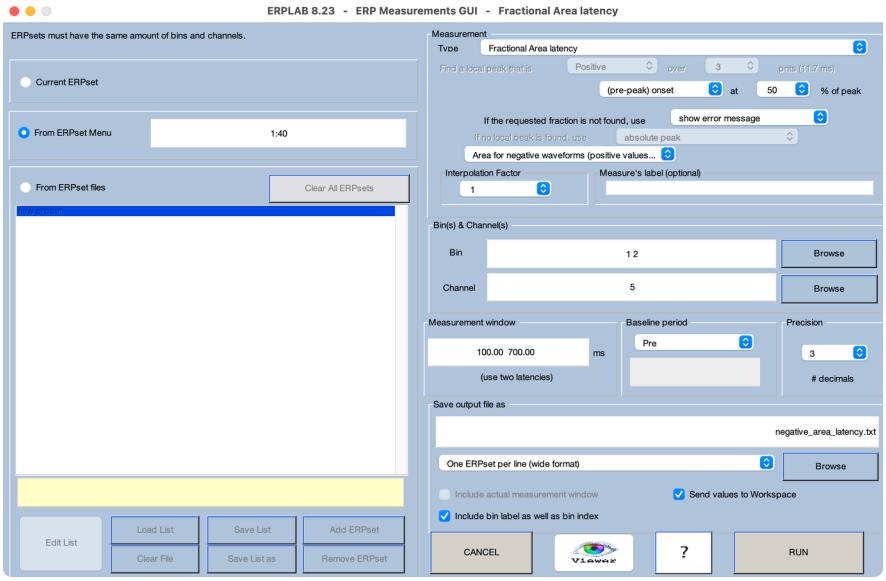


Don't Worry About High-Frequency Noise in Area-Based Measures

Area-based measures like fractional area latency are relatively insensitive to high-frequency noise, so we will apply this method to the unfiltered data. It's also usually unnecessary to filter out high-frequency noise when measuring mean amplitude.

Let's apply this scoring algorithm to the single-participant waveforms. If the filtered ERPsets from the previous exercise are still loaded in ERPLAB, clear them (or quit and restart EEGLAB). Then load the 40 unfiltered difference waves (from the **Chapter_10 > Data > ERPsets_CI_Diff** folder). Launch the Measurement Tool, and set it up as shown in Screenshot 10.7. The measurement **Type** is **Fractional area latency**, and we're looking for the **50%** point in the **Area for negative waveforms**. We're again measuring from Bins 1 and 2 (Compatible and Incompatible) in the C3/C4 channel, and we're saving the scores in a file named **negative_area_latency.txt**.

Screenshot 10.7



Using the Viewer, make sure that the scoring algorithm is working properly. Then go back to the Measurement Tool and click **RUN** to save the measurements. Load the data into your statistics package and do a paired *t* test, as in the previous exercise. You'll see that the mean latency is ~45 ms shorter for the Compatible condition than for the Incompatible condition, which is actually a somewhat smaller difference than we saw for peak latency (a 57 ms difference). However, the Cohen's *d* has increased substantially, from -0.577 for peak latency to -0.823 for the 50% area latency measure. And if you look at the descriptive statistics, you'll see that the standard deviations are now quite a bit lower. So, we now have a large effect size instead of a medium effect size, due to reduced variability (presumably owing to reduced measurement error).

The increased effect size we're seeing for 50% area latency relative to peak latency is consistent with what I've seen in many previous experiments. This is one of the reasons I recommend using 50% area latency, especially when the measurements are being obtained from difference waves.

When to Use Fractional Area Latency

The fractional area latency algorithm works well only if the waveform is dominated by a single component. This is usually true of difference waves, which are designed to isolate a single component. The 50% area latency measure also works well on parent waves when the component of interest is so large that it dominates everything else (e.g., the N400 for semantically deviant words or the P3b for rare targets).

A more direct way to compare measurement error for these two different scoring algorithms would be to look at the standardized measurement error (SME). Unfortunately, it's complicated to compute the SME for anything other than mean amplitudes. When some other scoring algorithm is used, or when the measurements are obtained from difference waves, a method called *bootstrapping* is necessary for calculating the SME. Currently, this can't be done from the ERPLAB GUI and instead requires scripting. I've provided a script for this at the end of the chapter. The script demonstrates that the SME was in fact much better (lower) for the 50% area latency measure than for the peak latency measure in this experiment.

This page titled 10.8: Exercise- Fractional Area Latency is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.9: Exercise- Quantifying Onset Latency

As described in Chapter 2 of Luck (2014), the onset time of a difference between two conditions can be extremely informative. For example, the brain can't have a more negative response over the contralateral hemisphere than over the ipsilateral hemisphere until it has determined which hand should respond, so the onset latency of the LRP can be used as a marker of the time at which the brain has decided on a response (see Chapters 2 and 3 in Luck, 2014, for a more detailed and nuanced discussion).

How can we quantify the onset time of a difference wave? Consider, for example, the Compatible waveform in Figure 10.2.C. The negativity of the LRP first falls below the zero line a little before 200 ms. However, the negativity before 200 ms is no larger than the noise level (as assessed, e.g., by the variations in voltage during the prestimulus period). Early research attempted to solve this problem by using a statistical criterion such as the first of N consecutive points that are at least 2 standard deviations greater than the noise level (where the standard deviation is measured from the variation in voltage during the prestimulus period). However, this approach suffers from low power, and single-participant scores will vary according to the noise level as well as the true onset time. A terrific study by Kiesel et al. (2008) rigorously compared this technique with peak latency and two other measures that were not very widely used at the time—fractional area latency and fractional peak latency—and found that the two less widely used scoring methods were actually the best. These two methods are now more commonly used, and we'll focus on them here.

We already looked at fractional area latency in the previous exercise, but we used it to estimate the midpoint latency (the 50% area latency) rather than the onset latency. To estimate the onset latency, we simply need to use a lower percentage. In the present exercise, we'll calculate the time at which the area reaches the 15% point. To get started, make sure that the 40 ERPsets from the **Chapter_10 > Data > ERPsets_CI_Diff** folder are loaded. Launch the Measurement Tool, and set it up as in the previous exercise (Screenshot 10.7), except change the percentage from **50** to **15**, and change the name of the output file to something like **FAL15_latency.txt**. Take a look at the scores for the individual waveforms using the Viewer, and then run the measurement routine to save the scores.

As before, load the resulting scores into your statistical package and compute the paired *t* test to compare the Compatible and Incompatible conditions. You should see that difference in means across conditions is ~50 ms and that the effect is statistically significant ($t(39) = -6.06$, $p < .001$) with a very large effect size ($d = -1.044$).

Now let's try the other scoring algorithm, fractional peak latency, which is illustrated in Figure 10.4. This method finds the peak and then moves backward in time until the voltage reaches some fraction of the peak voltage (usually the 50% point). The latency of this point is then used as the estimate of onset latency. You might wonder why we usually choose the 50% point. Isn't the 15% point, for example, closer to the true onset? There are two reasons to choose the 50% point. First, it's less influenced by noise and therefore more reliable than lower percentages. Second, it actually does a better job of capturing the average onset time given that there is almost always significant trial-to-trial variation in onset times. As discussed in Chapter 2 of Luck (2014), the first moment that an averaged waveform deviates from zero is driven by the trials with the earliest onset times. And as discussed in Chapter 9 of that book, the 50% peak latency point accurately captures the average of the single-trial onset times under some conditions.

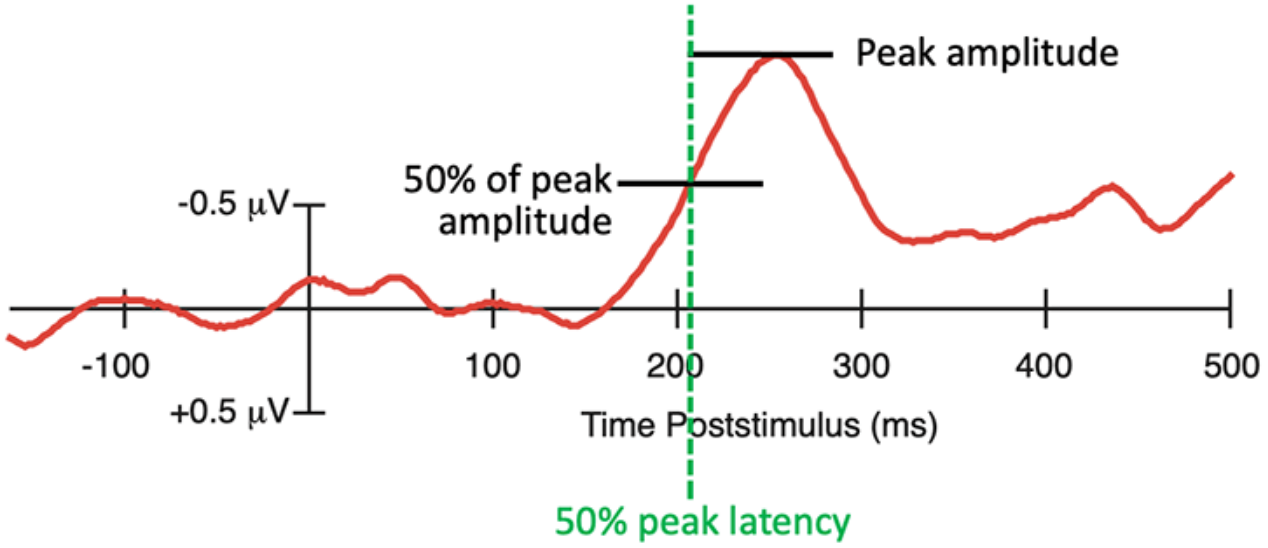
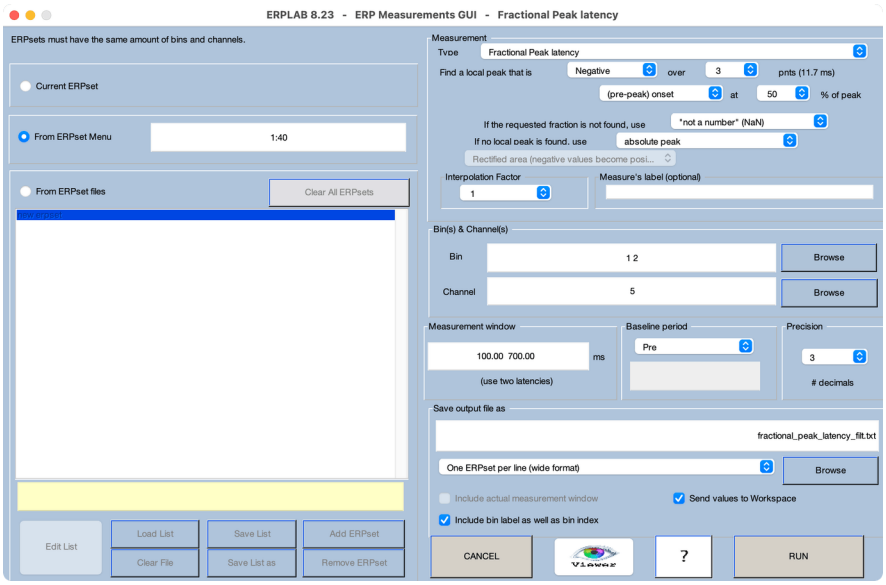


Figure 10.4. Example of the fractional peak latency method. In this example, we obtained the 50% peak latency (the latency at which the voltage reached 50% of the peak voltage).

Let's give it a try. First, clear the existing ERPsets out of ERPLAB (or quit and restart EEGLAB) and load the filtered ERPsets in the **Chapter_10 > Data > ERPsets_CI_Diff_filt** folder. This scoring method is, unfortunately, very sensitive to high-frequency noise, so we ordinarily apply fairly aggressive low-pass filtering (which, fortunately, has relatively little impact on the 50% peak latency). Launch the Measurement Tool, and set it up as shown in Screenshot 10.8. Once you have the parameters set, use the Viewer to make sure that the scores look appropriate for the single-participant waveforms. Then run the routine to save the scores to a file named **fractional_peak_latency_filt.txt**.

Screenshot 10.8



Load the resulting scores into your statistical package and compute the paired *t* test. You should see that difference in means across conditions is ~50 ms, just as for the 15% area latency measure from the previous exercise. The effect is statistically significant ($t(39) = -4.39$, $p < .001$), but the effect size is smaller than observed in the previous exercise (d =-0.695 for 50% peak latency versus d = -1.044 for 15% area latency).

So, which of these two scoring methods is best? The effect size was larger for 15% area latency than for 50% peak latency in the analysis you just did. Also, Kiesel et al (2008) found that 50% area latency yielded less variability than 50% peak latency.

Unfortunately, they didn't examine 15% area latency, and they didn't apply an aggressive low-pass filter prior to obtaining the 50% peak latency scores. We also found lower standard deviations for 50% area latency than for 50% peak latency for all of the basic difference waves in the six ERP CORE paradigms (see Table 3 in Kappenman et al., 2021). However, the 50% area latency captures the midpoint of the difference wave, not the onset, which is less sensitive to noise, so this really isn't a fair comparison. I think it's fair to say that this issue is unresolved at this point.

However, there is an important conceptual difference between these two scoring methods. Specifically, fractional area latency is impacted by voltages throughout the entire measurement window. For example, the negative voltage late in the waveform for the Incompatible trials (see Figure 10.2.C) will have an impact on the 15% fractional area latency score. By contrast, fractional peak latency is not influenced by anything that happens after the peak. This is particularly clear in the example shown in Figure 10.4, where there is a long "tail" to the difference wave that will have a large impact on the fractional area latency score but will have no impact on the fractional peak latency score. For this reason, I usually use the fractional peak latency score.

This exemplifies a broader issue: Although minimizing measurement error is important, it's also important to make sure that your scoring method is *valid* (i.e., measures what you are trying to measure with minimal influence from other factors).

This page titled 10.9: Exercise- Quantifying Onset Latency is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.10: Exercise- Collapsing Across Channels and Correlating Latencies with Response Times

So far, we've been measuring and analyzing the data only from the C3 and C4 channels, where the LRP is largest. However, it's often valuable to include the data from multiple channels. One way to do that is to measure from multiple different channels and include channel as a factor in the statistical analysis. However, that adds another factor to the analysis, which increases the number of *p* values and therefore increases the probability of getting bogus-but-significant effects. Also, interactions between an experimental manipulation and electrode site are difficult to interpret (Urbach & Kutas, 2002, 2006). In most cases, I therefore recommend averaging the waveforms across the channels where the component is large, creating a *cluster*, and then obtaining the amplitude or latency scores from the cluster waveform. Averaging across channels in this manner tends to produce a cleaner waveform, which decreases the measurement error (as long as we don't include channels where the ERP effect is substantially smaller). Also, it avoids the temptation to "cherry-pick" the channel with the largest effect.

In the ERP CORE flankers experiment, the LRP effect was only slightly smaller in the F3/F4 and C5/C6 channels than in the C3/C4 channel, so we should be able to decrease our measurement error and increase our effect sizes by creating a cluster of these three sets of channels.

I've already created this cluster in the difference waves in the **Chapter_10** folder. They're in Channel 12, which is labeled **cluster**. Let's try measuring the peak latency from this channel in the filtered data (which should already be loaded). You can just repeat the measurement and analysis procedures from the earlier exercise where we measured the peak latency from the filtered data, but changing the **Channel** from **5** to **12**. You should see that the effect size has increased a bit (-0.683 for the cluster analysis relative to -0.630 when we measured from the C3/C4 channel). Collapsing across channels doesn't always increase the effect size, but it doesn't usually hurt, and it avoids the need to include channel as a factor in the analysis or the need to determine which one channel to use. My lab now measures from a cluster in virtually all of our studies.

Our last step will be to ask whether the peak latency values are correlated with response times (RTs). That is, do participants with later LRP peaks also have slower RTs? Unfortunately, it takes some significant work to extract RTs using ERPLAB. In your own experiments, you might want to do this using the output of your stimulus presentation system instead of using ERPLAB. For the present exercise, I wrote an ERPLAB script to obtain the mean RTs for Compatible and Incompatible trials for each participant. I saved the values to an Excel spreadsheet named **RT.xlsx**, which you can find in the **Chapter_10** folder. An advantage of using an ERPLAB script to get the RTs is that you can exclude the trials that were rejected because of artifacts from the mean RTs so that the ERP data and the behavioral data are based on exactly the same set of trials. A disadvantage is that it takes a bit of work. I've provided the script I wrote for this purpose (named **get_LRP_RTs.m** in the **Chapter_10** folder), which you can use as a model for your own studies.

Let's look at the correlations. You'll need to combine the RT values from the spreadsheet with the peak latency values that you just created from the waveforms that were collapsed across the three pairs of electrode sites. And then you'll need to read them into your statistical package or into a spreadsheet program (see **peak_latency_filt_collapsed.xlsx** in the **Chapter_10** folder). You'll then want to look at the correlation between peak latency and RT separately for the Compatible condition and the Incompatible condition.

Figure 10.5 shows the results I obtained in Excel. The peak latency scores were correlated fairly well with the RTs, especially for the incompatible trials. Some outliers in the LRP peak latency scores for the Compatible trials clearly reduced the correlation in the that condition. Nonetheless, these correlations show that the timing of the brain activity is related to the timing of the behavioral response.

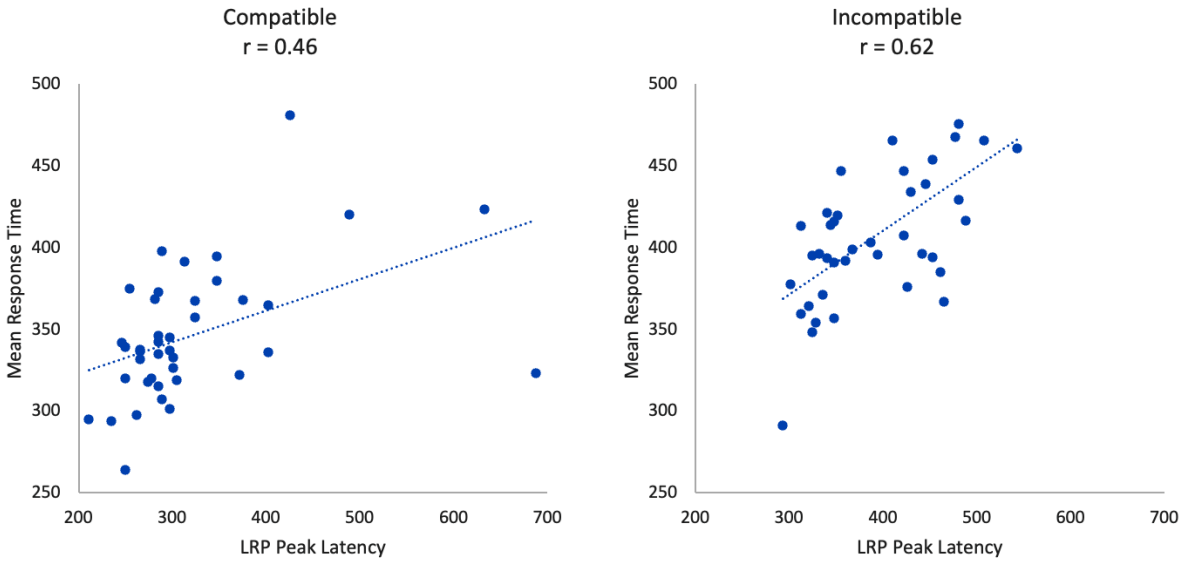


Figure 10.5. Scatterplots showing the relationship between LRP peak latency and response time for the Compatible and Incompatible conditions.

This page titled 10.10: Exercise- Collapsing Across Channels and Correlating Latencies with Response Times is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.11: Matlab Scripts For This Chapter

I've created three scripts for this chapter. The first script is named **LRP_RTs.m**, and it shows you how to get the single-trial RTs for the trials without artifacts flagged, compute the mean RT for each condition, and save the results to an Excel spreadsheet.

The second script, **LRP_scoring.m**, shows you how to obtain several of the amplitude and latency scores described in this chapter.

The third script, **LRP_bSME.m**, demonstrates how to get the SME values for several of the scores. For scores other than mean amplitude, bootstrapping is required (Luck et al., 2021), and we call the result the *bootstrapped SME* or *bSME*. The script demonstrates how to implement the bootstrapping procedure and compute the bSME values. Bootstrapping requires re-averaging the data for a given participant many times, and it can be slow. The script is set to do only 100 iterations per participant so that it runs reasonably quickly. There is a variable you can change to a larger value (e.g., 10,000) to get more robust SME estimates.

This page titled 10.11: Matlab Scripts For This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

10.12: Key Takeaways and References

Key Takeaways

- Many methods exist to quantify the timing or magnitude of an ERP component or experimental effect. The traditional methods—peak amplitude and peak latency—have many shortcomings, and better methods are available.
- In many cases, it is advantageous to obtain scores from a difference wave that isolates the component or experimental effect of interest.
- You should always visually verify that the scores are being calculated appropriately for each individual ERP waveform.
- Most scoring methods require specifying a measurement window, and this needs to be done in an unbiased manner. If you decide on the measurement window after seeing the waveforms, you may consciously or unconsciously choose a window that increases the probability of bogus-but-significant effects.
- When you have many conditions and/or channels, it's easy to accidentally put the cells of the design into the wrong order in the statistical analysis. You should always check the table of means produced by your statistical package and make sure it matches what you are seeing in the grand average ERP waveforms.
  - For mean amplitude, but not most other scoring methods, taking the score from the grand average waveforms gives you the same result as measuring from the single-participant waveforms and then averaging. This makes it easy to compare the table of means with the grand averages.
  - For other measures, you can still make sure that the table of means shows the same pattern as the grand averages, even if the individual values are not identical.
- To reduce the number of *p* values and the likelihood of bogus-but-significant effects, you should use the smallest possible number of factors in your statistical analyses. This can often be achieved by collapsing across channels and obtaining scores from difference waves.

References

Donchin, E., & Heffley, E. F. (1978). Multivariate analysis of event-related potential data: A tutorial review. In D. Otto (Ed.), *Multidisciplinary Perspectives in Event-Related Brain Potential Research* (pp. 555–572). U.S. Government Printing Office.

Eriksen, C. W. (1995). The flankers task and response competition: A useful tool for investigating a variety of cognitive problems. *Visual Cognition*, *2*, 101–118.

Gehring, W. J., Liu, Y., Orr, J. M., & Carp, J. (2012). The error-related negativity (ERN/Ne). In S. J. Luck & E. S. Kappenman (Eds.), *The Oxford Handbook of Event-Related Potential Components* (pp. 231–292). Oxford University Press.

Gratton, G., Coles, M. G. H., Sirevaag, E. J., Eriksen, C. W., & Donchin, E. (1988). Pre- and post-stimulus activation of response channels: A psychophysiological analysis. *Journal of Experimental Psychology: Human Perception and Performance*, *14*, 331–344.

Kappenman, E. S., Farrens, J. L., Zhang, W., Stewart, A. X., & Luck, S. J. (2021). ERP CORE: An Open Resource for Human Event-Related Potential Research. *NeuroImage*, *225*, 117465. https://doi.org/10.1016/j.neuroimage.2020.117465

Kiesel, A., Miller, J., Jolicoeur, P., & Brisson, B. (2008). Measurement of ERP latency differences: A comparison of single-participant and jackknife-based scoring methods. *Psychophysiology*, *45*, 250–274. https://doi.org/10.1111/j.1469-8986.2007.00618.x

Love, J., Selker, R., Marsman, M., Jamil, T., Dropmann, D., Verhagen, J., Ly, A., Gronau, Q. F., Šmíra, M., Epskamp, S., Matzke, D., Wild, A., Knight, P., Rouder, J. N., Morey, R. D., & Wagenmakers, E.-J. (2019). JASP: Graphical Statistical Software for Common Statistical Designs. *Journal of Statistical Software*, *88*(1), 1–17. https://doi.org/10.18637/jss.v088.i02

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Luck, S. J., & Gaspelin, N. (2017). How to get statistically significant effects in any ERP experiment (and why you shouldn't). *Psychophysiology*, *54*, 146–157. https://doi.org/10.1111/psyp.12639

Luck, S. J., Stewart, A. X., Simmons, A. M., & Rhemtulla, M. (2021). Standardized measurement error: A universal metric of data

quality for averaged event-related potentials. *Psychophysiology*, *58*, e13793. https://doi.org/10.1111/psyp.13793

Smulders, F. T. Y., & Miller, J. O. (2012). The Lateralized Readiness Potential. In S. J. Luck & E. S. Kappenman (Eds.), *The Oxford Handbook of Event-Related Potential Components* (pp. 209–229). Oxford University Press.

Urbach, T. P., & Kutas, M. (2002). The intractability of scaling scalp distributions to infer neuroelectric sources. *Psychophysiology*, *39*, 791–808. https://doi.org/10.1017/S0048577202010648

Urbach, T. P., & Kutas, M. (2006). Interpreting event-related brain potential (ERP) distributions: Implications of baseline potentials and variability with application to amplitude normalization by vector scaling. *Biological Psychology*, *72*(3), 333–343. https://doi.org/10.1016/j.biopsycho.2005.11.012

This page titled 10.12: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11: EEGLAB AND ERPLAB SCRIPTING

Learning Objectives

In this chapter, you will learn to:

- Use the Matlab command line to execute EEGLAB and ERPLAB routines.
- Efficiently move back and forth between scripts and the GUI to have the best of both worlds.
- Use the EEG and ERP histories to see the commands that correspond to the EEGLAB and ERPLAB procedures that you run from the GUI.
- Create simple scripts by copying commands from the EEG and ERP histories into a text file.
- Use variables to define paths in a way that avoids problems and makes it easy to move your scripts and data to new locations.
- Use loops so that you can efficiently repeat a set of processing steps on the data from multiple participants.
- Read from and write to spreadsheets and text files to increase the power and flexibility of your scripts.
- Create an entire processing pipeline that begins with the raw EEG and ends with amplitude and latency scores that are ready for statistical analysis.
- Implement good programming practices that will minimize errors and increase the readability of your code.

You can do a lot with the EEGLAB and ERPLAB GUIs. However, you will eventually grow tired of all the pointing and clicking, especially the seventh time you reanalyze the data from an experiment (and believe me, you will be lucky if it's only seven times). It's straightforward to write Matlab scripts that automate almost every processing step. Scripts can also help you avoid the errors that inevitably arise in the thousands of clicks required to conduct every processing step for every participant. Better yet, scripts can allow you to implement new or modified processing steps, making your research more innovative. Scripts also play an important role in open science: When you publish a paper, you can make your data and scripts available (e.g., using the Open Science Framework), and then the world can see exactly how you processed the data and can exactly reproduce your analysis methods. These are the reasons why this whole chapter is devoted to scripting.

11.1: Data for This Chapter

11.2: Expected Background Knowledge

11.3: Bugs as an Opportunity for Growth

11.4: Design of the N170 Experiment

11.5: Exercise- The Matlab Command Line and the EEG Variable

11.6: Exercise- The ALLEEG Variable and Redrawing the GUI

11.7: Exercise- EEG.history and eegh

11.8: Exercise- From the Command Line to a Script

11.9: Exercise- Using a Variable for the Path

11.10: Exercise- Loops

11.11: Exercise- Looping Through Data from Multiple Participants

11.12: Rapid Cycling Between Coding and Testing

11.13: Exercise- Referencing with a Script

11.14: Exercise- Improving the Referencing Script

11.15: Exercise- Preprocessing the EEG and Using a Spreadsheet to Store Subject-Specific Information

11.16: Exercise- Building an Entire EEG Processing Pipeline

11.17: Exercise- Averaging with a Custom aSME Time Window
11.18: Exercise- Scoring Amplitudes and Latencies and Performing Statistical Analyses
11.19: Key Takeaways and References

This page titled 11: EEGLAB and ERPLAB Scripting is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.1: Data for This Chapter

The data we'll be using for the exercises in this chapter can be found in the Chapter_11 folder in the master folder: https://doi.org/10.18115/D50056.

This page titled 11.1: Data for This Chapter is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.2: Expected Background Knowledge

This chapter is designed for individuals all levels of prior experience, including people who have very little programming experience, people with substantial experience in other languages but little or no Matlab programming experience, and people who already know how to program in Matlab and just want to learn how to use EEGLAB and ERPLAB routines. I'm not actually a very experienced Matlab programmer myself, but I have a lot of experience with other languages.

Learning how to write scripts may seem daunting if you've never done any serious computer programming before. Fortunately, EEGLAB and ERPLAB have a *history* feature that make it fairly easy to write simple scripts for automating your analyses. Every time you run a routine from the GUI, the equivalent script command is saved in the history. To begin writing a script, you simply go through all the steps in the GUI, and then copy the commands from the history into a script file. You'll usually need to make a few minor modifications to the commands, and then you need to add a little bit of general Matlab code so that your script can automatically loop through all your participants. But that's the essence of creating automated EEGLAB/ERPLAB scripts. In fact, that's exactly the process I used to create the example scripts for this book.

The scripts you'll write to automate your EEGLAB/ERPLAB analyses will be pretty simple, but you do need to understand some basic programming concepts, especially *variables*, *arrays*, and *loops*. I'm not going to explain these concepts, so you will need to do a little preparation before starting this chapter. When someone without a lot of programming experience joins my lab, I typically have them take the online Introduction to Programming with MATLAB course offered on Coursera. Then, I have them work through a great book called *Matlab for Behavioral Scientists* (Rosenbaum et al., 2014). If you already have a lot of experience with other programming languages, but you haven't programmed in Matlab, I recommend getting any of the books that provide a general introduction to Matlab so you can learn its specific syntax and its unusual but powerful approach to matrix operations.

This page titled 11.2: Expected Background Knowledge is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.3: Bugs as an Opportunity for Growth

No matter how much prior experience you have, you need to be prepared for some frustration in this chapter. First, mistakes and bugs are inevitable in programming. As I've said before, you need to think of mistakes and problems as opportunities for learning. In fact, I encourage you to play with the example scripts in this chapter, changing them to see how they work or to try new things. You'll get lots of error messages, but that's actually part of the reason for playing. And that's why I wrote the Troubleshooting Guide in Appendix 2.

I made literally hundreds of mistakes while creating the example scripts in this chapter, and I can't count how many times I heard Matlab's error sound. Most of these mistakes were small and easily fixed. But others were more conceptual and took some time to understand and remedy. In this chapter, I'll describe some of my mistakes, how I discovered them, and how I solved them. I want to both show you that making mistakes is part of the process and give you some insights into troubleshooting approaches. There are probably still some mistakes in the scripts for this chapter. If you see something that doesn't quite make sense, it could well be an error on my part.

As you go through the chapter, keep in mind that there are millions of small details about Matlab and EEGLAB that I don't mention. If I provided every possible detail, this chapter would have turned into an entire book, and it probably would have required several volumes. I wanted to keep the chapter reasonably brief, so I'm counting on you to draw inferences from what you see and to figure some things out for yourself. This will be a little frustrating at times, but you'll learn a lot more this way. When you have questions, do what I did hundreds of times while writing this chapter: Google it. Or ask someone with more experience. Or post a question via email to the EEGLAB or ERPLAB listservs. And don't forget about the Troubleshooting Guide in Appendix 2.

Once you've completed the exercises in this chapter, you should take a look at the example scripts in the previous chapters. They include examples of specific processing steps and options that aren't covered in the present chapter. You can also find lots of examples of EEGLAB/ERPLAB scripts online, and you may have labmates or colleagues who can provide example scripts. However, I have an important rule about this: you should never apply someone else's script to your own data unless you fully understand every line of code in that script. People often violate this rule, and they end up with garbage (and often don't realize it).

This page titled 11.3: Bugs as an Opportunity for Growth is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.4: Design of the N170 Experiment

This section will provide a brief overview of the experimental design and main results from the ERP CORE N170 experiment, which was based on a prior study by Rossion and Caharel (2011). The N170 is typically found to be larger for faces than for almost any class of non-face stimuli. In this experiment, cars were used as the non-face stimuli. We also presented phase-scrambled faces and cars, which contain the same low-level information as the faces and the cars but without the higher-level features that are essential for discriminating between the broader classes of faces and cars.

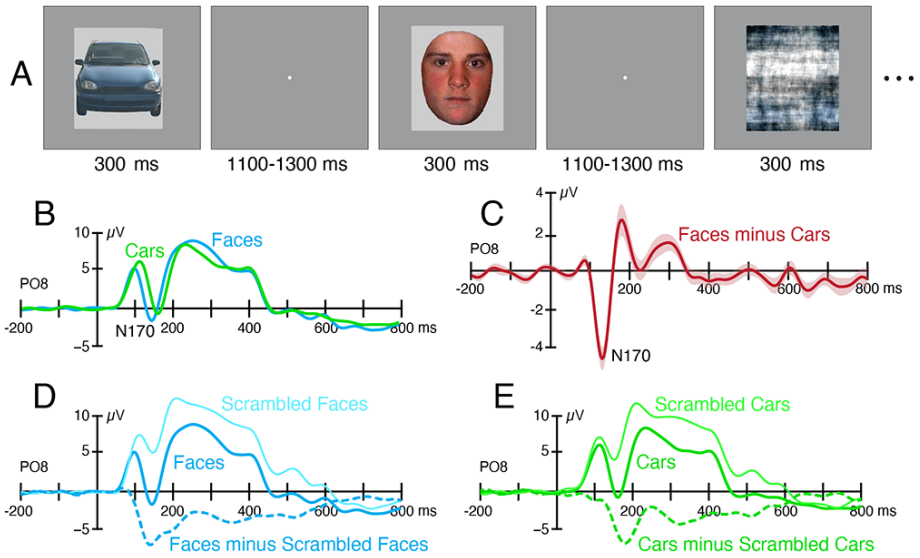


Figure 11.1. Experimental paradigm and results from the ERP CORE N170 experiment. (A) Example sequence of stimuli. Participants were instructed to press one of two buttons to indicate whether a given stimulus was intact (face or car) or scrambled (scrambled face or scrambled car). (B) Grand average ERP waveforms for car and face stimuli, averaged overall all 37 participants. (C) Grand average N170 difference wave, which was created by subtracting the car ERP waveform from the face ERP waveform. (D) Grand average ERP waveforms for faces, scrambled faces, and the difference between faces and scrambled faces. (E) Grand average ERP waveforms for cars, scrambled cars, and the difference between cars and scrambled cars. All waveforms show data from PO8 referenced to the average of all channels.

The N170 effect is largely independent of what task the participants are performing, but it can be helpful to have participants perform some kind of task to keep them alert and attentive. In this experiment, participants were instructed to discriminate whether a given stimulus was an intact image or a scrambled image. Specifically, they pressed one button for faces and cars and another button for scrambled faces and scrambled cars. These four stimulus classes were presented in random order. Stimulus duration was 300 ms, and a blank interstimulus interval of 1100-1300 ms occurred between stimuli.

We used 40 different face images, 40 different car images, and scrambled versions of each of these images. Each of the 160 images was presented once. A different event code was used for each of the 160 images, as indicated in Table 11.1.

Table 11.1. Event codes for the ERP CORE N170 experiment.

		Event Code
Stimuli	Faces	1–40
	Cars	41–80

	Scrambled Faces	101–140
	Scrambled Cars	141–180
	Accuracy	**Event Code**
Responses	correct	201
	incorrect	202

Of the 40 participants who were tested, three had to be excluded because of artifacts. The grand average waveforms from the remaining 37 participants are shown for the face and car stimuli in Figure 11.1.B. Note that, for the sake of simplicity, the exercises in the present chapter include only Subjects 1-10. Subject 5 was one of the excluded subjects, so the final analyses in this chapter are based on the data from only 9 participants.

As illustrated in Figure 11.1.B, the peak of the N170 was slightly greater (more negative) for faces than for cars. The main difference between the waveforms appeared to be a faster onset latency for the faces. However, this means that the amplitude was greater for faces than for cars between approximately 110 and 150 ms. This is illustrated in the faces-minus-cars difference wave shown in Figure 11.1.C. When I first saw these results, I thought we had done something wrong. However, this pattern is actually quite common in N170 experiments.

When you compare the ERPs elicited by faces and cars, it's possible that any differences in the waveforms could be a result of differences in low-level features (e.g., luminance, spatial frequency) that are not actually important in perceiving whether a stimulus is a face or a car. This is why the experiment included phase-scrambled face and car images, which contain the same low-level features as the faces and cars but do not contain the higher-level features that we used to determine whether an image is a face or a car.

Figure 11.1.D shows the ERPs elicited by the faces and the scrambled faces, along with the faces-minus-scrambled-faces difference wave. This difference wave is designed to subtract out any brain activity related to the low-level features shared by faces and scrambled faces. You can see a nice N170 in this difference wave. Figure 11.1.E shows the corresponding waveforms for cars, scrambled cars, and the cars-minus-scrambled-cars difference. You can again see an N170, but it's smaller and later than the N170 in the faces-minus-scrambled-faces difference wave.

This page titled 11.4: Design of the N170 Experiment is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.5: Exercise- The Matlab Command Line and the EEG Variable

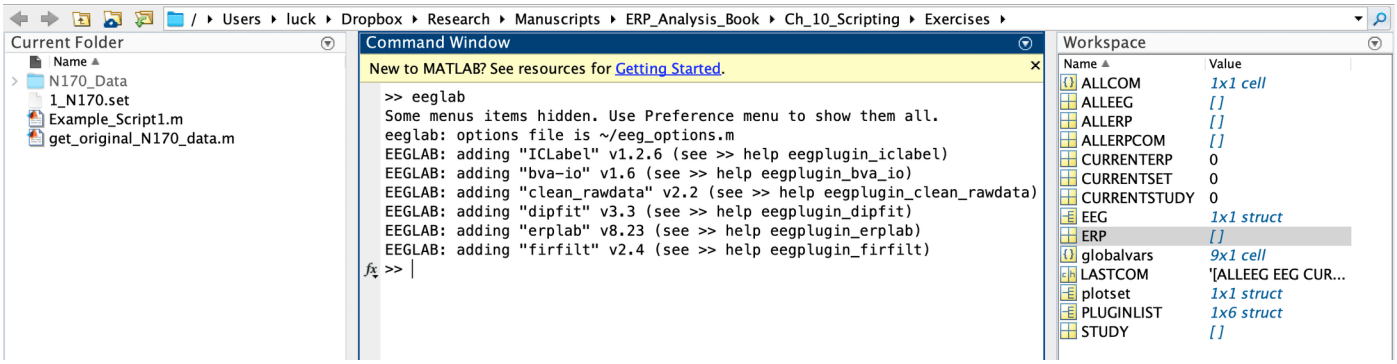
A Matlab *script* is simply a series of Matlab commands that are stored in a text file. Running a script is equivalent to typing the commands in the Matlab command line. So, we're going to start by running some commands from the command line.

To start, quit EEGLAB if it's already running. Then, type **clear all** on the Matlab command line (i.e., at the prompt in the Matlab Command Window pane). This clears everything out of Matlab's memory, which is a good thing to do when you're first getting started on a new task. When you clear the variables, anything that was in Matlab's Workspace pane should disappear. Another nice housekeeping command is **clc**, which clears the command window so that you're not distracted by what happened earlier.

By the way, Matlab doesn't do anything with a command until you hit the **Return** key (which may instead be labeled **Enter** on your keyboard). When I say that you should type something on the Matlab command line, you should follow it with **Return** or **Enter**. If you didn't already press **Return/Enter** after **clear all**, do it now.

Now launch EEGLAB by typing **eeglab** on the command line, set the **Chapter_11** folder to be the current folder, and load the dataset named **1_N170.set** into EEGLAB. You should now see a set of variables in the Workspace pane, as shown in Screenshot 11.1. This includes **EEG**, which EEGLAB uses to store the current dataset, and **ALLEEG**, which EEGLAB uses to store all of the datasets that are available in memory. ERPLAB also creates corresponding **ERP** and **ALLERP** variables to hold the current ERPset and all available ERPsets.

Screenshot 11.1



You can see the contents of a variable by typing its name on the command line. Let's try it! Type **EEG** on the command line (followed by the **Return** key, of course). Variable names in Matlab are case-sensitive, so make sure you type **EEG** and not **eeg** or **Eeg**. Once you type this, the contents of the **EEG** variable will be shown in the Command Window. Screenshot 11.2 shows the first few lines.

Screenshot 11.2

```
>> EEG

EEG =

  struct with fields:

            setname: '1_N170'
           filename: '1_N170.set'
           filepath: '/Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Ch_10_Scripting/Exercises/'
            subject: ''
              group: ''
          condition: ''
```

EEG is a complicated variable that contains many individual fields (you can learn about the details by typing **help eeg_checkset** on the command line). For example, the field named **EEG.setname** stores the name of the dataset, which is shown in **EEGLAB > Datasets**. Let's change the name of the dataset. To do this, type **EEG.setname = 'My First Custom Dataset'**. (Note that the period at the end of the sentence is not part of the command you should type. I use **boldface** to indicate the exact text you should type.) Matlab will then print out the whole EEG variable again in the Command Window, and you'll be able to see that the name has changed.

Here are a couple important things to note about the command you just entered:

- Matlab uses single quote marks to indicate literal text. If you didn't use the quote marks and had instead typed **EEG.setname = My First Custom Dataset**, Matlab would have assumed that **My First Custom Dataset** was a sequence of four variable names (**My**, **First**, **Custom**, and **Dataset**). See the text box below for a hint about single quote marks.
- Most Matlab commands return one or more variables, and the value of the returned variables is ordinarily printed in the Command Window. You can suppress this by placing a semicolon at the end of the command. To see this in action, type **x = 1** (followed by the **Return** key), and then type **x = 2;** (again followed by the **Return** key).
- When you change the set name using the command line, you won't see the new set name in the **Datasets** menu. The reason for this will be explained later in this section.

You can also see the contents of a variable by double-clicking on the name of the variable in Matlab's Workspace pane. Try double-clicking the **EEG** variable in this pane. A new Variables pane should appear in Matlab, showing you the fields of the **EEG** variable. One of those fields is named **times**, and it contains the latency in milliseconds of each time point in the dataset. Double-click it to see its contents; a new tab will open labeled **EEG.times**, and you'll see a very wide list of latency values. The EEG was sampled at 250 Hz, so the first point is 0 ms, the second point is 4 ms, the third point is 8 ms, etc.

Single Quotes

Click on the tab for the **EEG** structure and take another look at the **times** field. Next to the **times** name, you should see **1 x 170750 double**. The term **double** is used by Matlab (and many other programming languages) to refer to a number that is stored in scientific notation (e.g., X times 10^Y) using double the ordinary precision (and therefore double the amount of storage space). The **1 x 170750** part indicates that **EEG.times** is an array of these double-precision numbers with 1 row and 170750 columns. If you go back to the tab for **EEG.times**, you'll see that it has one row and 170750 columns (one column for each data point in the dataset).

In the tab showing the **EEG** variable, you'll see a variable named **data**, which is listed as **33 x 170750 double**. This variable stores the actual voltages in the dataset. It has 33 rows (one for each channel) and 170750 columns (one for each time point). That's a pretty natural way to store EEG data, isn't it?

When you start writing scripts, it's easy to get confused about the rows versus the columns of an array. I find it helpful to look at the array in the Variables pane to remind myself which dimension is the rows and which is the columns.

Click on the tab for the **EEG** structure and take another look at the **times** field. Next to the **times** name, you should see **1 x 170750 double**. The term **double** is used by Matlab (and many other programming languages) to refer to a number that is stored in scientific notation (e.g., X times 10^Y) using double the ordinary precision (and therefore double the amount of storage space). The **1 x 170750** part indicates that **EEG.times** is an array of these double-precision numbers with 1 row and 170750 columns. If you go back to the tab for **EEG.times**, you'll see that it has one row and 170750 columns (one column for each data point in the dataset).

In the tab showing the **EEG** variable, you'll see a variable named **data**, which is listed as **33 x 170750 double**. This variable stores the actual voltages in the dataset. It has 33 rows (one for each channel) and 170750 columns (one for each time point). That's a pretty natural way to store EEG data, isn't it?

When you start writing scripts, it's easy to get confused about the rows versus the columns of an array. I find it helpful to look at the array in the Variables pane to remind myself which dimension is the rows and which is the columns.

This page titled 11.5: Exercise- The Matlab Command Line and the EEG Variable is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.6: Exercise- The ALLEEG Variable and Redrawing the GUI

EEGLAB can have multiple datasets stored in memory simultaneously, which you can see in the **Datasets** menu. The current dataset is stored in the **EEG** variable. All of the datasets (including a copy of the current dataset) are stored in a variable named **ALLEEG**, which is just an array of **EEG** variables. When you go to the **Datasets** menu, you're seeing a list of the datasets stored in **ALLEEG**. This is one reason why changing the name of **EEG.setname** didn't cause a change in the **Datasets** menu. That is, we changed the name of this field in the **EEG** variable, but not in the **ALLEEG** variable that is used for the **Datasets** menu.

Let's fix that by typing **ALLEEG(1) = EEG;** on the command line (including the semicolon so that it doesn't print the new value in the Command Window). Now type **ALLEEG(1).setname**, and you'll see that the first (and only) **EEG** structure in **ALLEEG** has the new name.

However, if you look in the **Datasets** menu, you still won't see the new name for the dataset. This is because the Matlab GUI doesn't "know" that the **ALLEEG** variable has changed. That is, the GUI only updates the names listed in the menu when it thinks that something has changed. You can tell Matlab to update the EEGLAB GUI by typing **eeglab redraw** on the command line. Try this, and then look at the **Datasets** menu. Now you should see the updated name. Note that there's also an **erplab redraw** command for updating the ERPLAB and ERPsets menus.

Now let's see what happens when we have two datasets loaded in EEGLAB. To make the second dataset, filter the dataset you already have loaded (**EEGLAB > ERPLAB > Filter and Frequency Tools > Filters for EEG data**) using a high-pass cutoff at **0.1** Hz, a low-pass cutoff at **30** Hz, and a roll-off of **12** dB/octave. Name the resulting dataset **01_N170_filt**. You should now see two datasets in the **Datasets** menu.

If you look at the **EEG** tab in the Variables pane, you will see that **EEG.setname** is now **01_N170_filt**. This is because the EEG variable always holds the currently active dataset (which is the filtered dataset you just created). If you look in the Variables pane, you'll see that **ALLEEG** now has a size of 1 x 2 because we have two datasets loaded. If you type **ALLEEG(1).setname**, you'll see the name of the first dataset. If you type **ALLEEG(2).setname**, you'll see the name of the second dataset.

Leave these two datasets loaded for the next exercise.

This page titled 11.6: Exercise- The ALLEEG Variable and Redrawing the GUI is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.7: Exercise- EEG.history and eegh

EEGLAB has a super convenient *history* feature, which we shamelessly copied when we created ERPLAB. Whenever you run a routine from the EEGLAB GUI, the equivalent command text is saved in the history. The **history** field of the **EEG** variable (i.e., **EEG.history**) stores the history of the routines that were applied to that specific dataset, and you can see this history by simply typing **EEG.history** on the command line. You can also get a history of everything that was done since the last time you launched EEGLAB (which might include routines applied to many different datasets) by typing **eegh** on the command line.

Go to the **Datasets** menu and select the first dataset (which should be named **My First Custom Dataset**). The first dataset should now be stored in the **EEG** variable, and if you type **EEG.history** you should see the history of the routines that were applied to this dataset:

EEG = pop_loadset('filename','1_N170.set','filepath','/Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Ch_10_Scripting/Exercises/');

EEG = eeg_checkset(EEG);

EEG = eeg_checkset(EEG);

The first line is the command you used to load the dataset from a file. This is followed by two **eeg_checkset** commands. This command just verifies that everything is okay in the dataset. It was run once when you first loaded the dataset, and then it was run again when you switched from the second dataset back to the first dataset. Notice that the Matlab command that you used to change the name of the dataset isn't in the history. Only EEGLAB GUI operations are automatically saved in the history.

I want to pause for a minute and encourage you to appreciate what just happened. You used your mouse to load the dataset using the EEGLAB GUI, which should have been simple and natural now that you've spent a lot of time using EEGLAB. And then the history allowed you to see the actual Matlab command that was used to load the dataset. This is the same command you'll use to load datasets when you start writing scripts. So, you don't need to read some opaque documentation to figure out what command to use and how to use it. You just need to run the routine from the GUI and look at the history.

This makes scripting so much easier! And it's not just for beginners: when I created the scripts for each chapter of this book, I first ran the routines from the GUI so that I could get the history, which I then copied into the scripts that I was writing. I'm not sure which of the original EEGLAB developers came up with this scheme, but it's brilliant!

Now that I've gushed about how great this is, I want to note a small complication. If you look closely at the **pop_loadset** command that was used to load the dataset, you'll see that it specifies the filename (**'filename', '1_N170.set'**) and the location of the file on your hard drive (the *path* to the file, listed in the string following **'filepath'**). However, the path that you can see on your computer is not the same as the path shown above, which shows the path on my computer (**'/Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Ch_10_Scripting/Exercises/'**).

Keep in mind that the paths you'll see in this book won't be the same as the paths you'll see on your computer. And paths can be confusing even when you're not using this book. One of the most common problems people have when learning to write scripts is getting the paths wrong. I often get them wrong myself. Later, I'll show you some strategies for reducing the likelihood of having path problems.

Now select the second dataset in the **Datasets** menu, and type **EEG.history** to see the history for that dataset:

EEG = pop_loadset('filename','1_N170.set','filepath','/Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Ch_10_Scripting/Exercises/');

EEG = eeg_checkset(EEG);

EEG = pop_basicfilter(EEG, 1:33 , 'Boundary', 'boundary', 'Cutoff', [0.1 30], 'Design', 'butter', 'Filter', 'bandpass', 'Order', 2, 'RemoveDC', 'on'); % GUI: 10-Jun-2021 09:49:46

EEG.setname='1_N170_filt';

EEG = eeg_checkset(EEG);

EEG = eeg_checkset(EEG);

You'll see that the first couple lines are the same as the history for the first dataset. That's because those commands are part of the sequence of operations that was applied to the second dataset. Following those lines, you can see the command for filtering the EEG. When you saved the filtered dataset, that generated an additional command for setting the new **setname** value.

Now type **eegh** to see the history for the current EEGLAB session. It should look something like this:

[ALLEEG EEG CURRENTSET ALLCOM] = eeglab;

EEG = pop_loadset('filename','1_N170.set','filepath','/Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Ch_10_Scripting/Exercises/');

[ALLEEG, EEG, CURRENTSET] = eeg_store(ALLEEG, EEG, 0);

EEG = pop_basicfilter(EEG, 1:33 , 'Boundary', 'boundary', 'Cutoff', [0.1 30], 'Design', 'butter', 'Filter', 'bandpass', 'Order', 2, 'RemoveDC', 'on'); % GUI: 10-Jun-2021 10:08:08

[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, 1,'setname','1_N170_filt','gui','off');

[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, 2,'retrieve',1,'study',0);

[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, 1,'retrieve',2,'study',0);

The first line is the command for launching EEGLAB, which returns the **ALLEEG** and **EEG** variables (which are empty initially). It also returns a variable named **CURRENTSET**, which just indicates which dataset is currently active, and a variable named **ALLCOM**, which stores the history.

The second line is the command for loading the first dataset from your hard drive into the **EEG** variable. This is followed by an **eeg_store** command that adds it to the **ALLEEG** variable. It would be possible to replace this with **ALLEEG(1) = EEG**, but the **eeg_store** command takes care of a few additional details (like updating the value of **CURRENTSET**).

The next line is for the filtering routine, which operates on the **EEG** variable. This is followed by a **pop_newset** command that creates an entry for this new dataset in the **ALLEEG** variable (and sets the **setname**). We then have two more **pop_newset** commands that correspond to you setting the first dataset to be active in the **Datasets** menu and then setting the second dataset to be active in the **Datasets** menu.

You now know how to see the Matlab commands for all the routines you know how to run from the EEGLAB GUI. This will make your life much easier in the following exercises, where you'll start putting together scripts. I like to use **eegh** when I'm writing scripts that will interact with the EEGLAB GUI, where it's important to keep track of **ALLEEG** and **CURRENTSET**. But I typically use **EEG.history** for scripts that run independently of the GUI, in which case I typically have only one dataset in memory at a given time and don't need to worry about **ALLEEG**.

This page titled 11.7: Exercise- EEG.history and eegh is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.8: Exercise- From the Command Line to a Script

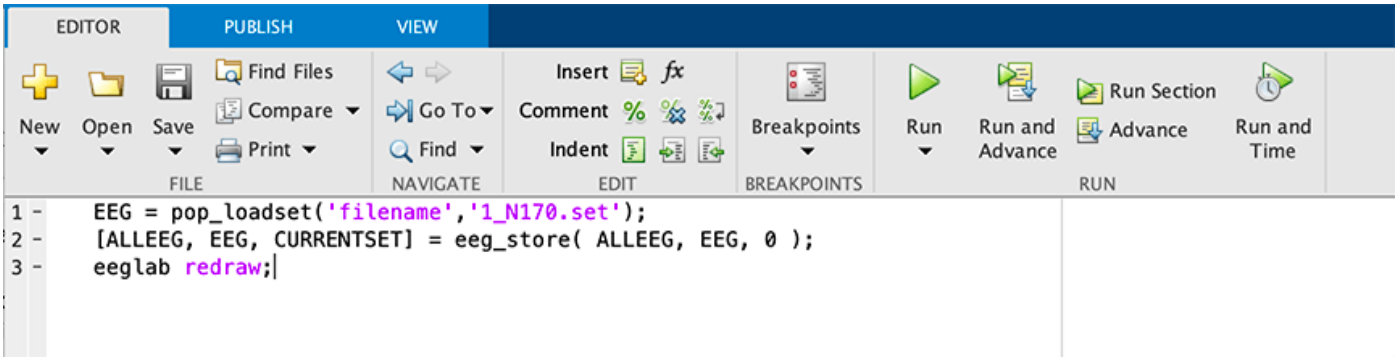
In this exercise, we'll see how commands that you type on the command line can be run from a script file. To start from a blank slate, you should quit EEGLAB, type **clear all** on the command line, and relaunch EEGLAB. Make sure that **Chapter_11** is the current folder in Matlab, and open the file named **Script1.m** by double-clicking the name from Matlab's **Current Folder** pane. This script consists of three lines, which open a dataset, add it to **ALLEEG**, and redraw the Matlab GUI.

Copy the first line of the file into your computer's clipboard, and then paste it onto the command line to execute the command. You can determine whether it has successfully loaded the dataset into the **EEG** variable by looking at this variable in the Matlab Workspace pane. Now paste the second line of the file onto the command line to add the dataset to **ALLEEG** (which you can verify by looking at **ALLEEG** in the Workspace pane). Now paste the third line to the command line to update the GUI. You can see that the dataset is now in the **Datasets** menu.

Let's take a closer look at the first line: **EEG = pop_loadset('filename','1_N170.set');** This is just like the first line of the **EEG.history** in the previous exercise, except that it doesn't specify the path. That's because the **1_N170.set** file is in the current folder (which you can see in Matlab's Current Folder pane). When you open a file but don't specify a path, Matlab will first look in the current folder. If it doesn't find the file there, it will search the entire Matlab path. When you save a file but don't specify a path, Matlab will save it in the current folder. In the next exercise, we'll look at a robust way to specify the path.

First, though, we'll see how to run an actual script. In theory, you could just put a set of commands in any text file and then copy-and-paste them onto the command line. The file would just be a way to store the commands so that you can easily reuse them. In fact, I sometimes use script files for this purpose. But there are easier ways to execute the commands in a script file. The simplest is just to click the **Run** button in the Matlab script editor. In my version of Matlab, it's a green triangle (see Screenshot 11.3). To see this in action, quit EEGLAB, type **clear all**, restart EEGLAB, and click the **Run** button in the editor window for **Script1.m**. If you look at the **EEG** variable, the **ALLEEG** variable, and the **Datasets** menu, you can confirm that each line of the script has executed. It's just as if you had copied all three lines into the clipboard and pasted them onto the command line (but faster and easier).

Screenshot 11.3



Now let's look at three ways that we can execute a portion of a script rather than the whole thing. The first is simple: Copy the set of lines that you'd like to execute into the clipboard and paste them onto the command line. You've already done this with single lines of code, but you can do this with multiple lines.

The other two ways are easier to demonstrate with a script that has several distinct parts, so load the script named **Script2.m** (e.g., by double-clicking on this name in the Current Folder pane). This script loads the data from Subjects 1 and 2. I'll explain how this script works in the next exercise. For now, we'll just see how to execute parts of it.

Start by quitting EEGLAB, typing **clear all**, and restarting EEGLAB. Now, go into the text editor for **Script2.m**, and select the first 7 lines (by dragging your mouse over those lines; note that you can see the line numbers along the left side of the window). Now right-click (or control-click) on the selected text, and select **Evaluate Selection** from the menu that pops up. (You can also see a keyboard equivalent, which will be more convenient for using this approach in the future.) You'll be able to see that the dataset from Subject 1 has been loaded (e.g., in the **ALLEEG** variable), but we haven't yet updated the EEGLAB GUI. To do that, select the last line of the script (line 12, **eeglab redraw**) and execute it (by selecting it, right-clicking, and selecting **Evaluate Selection**

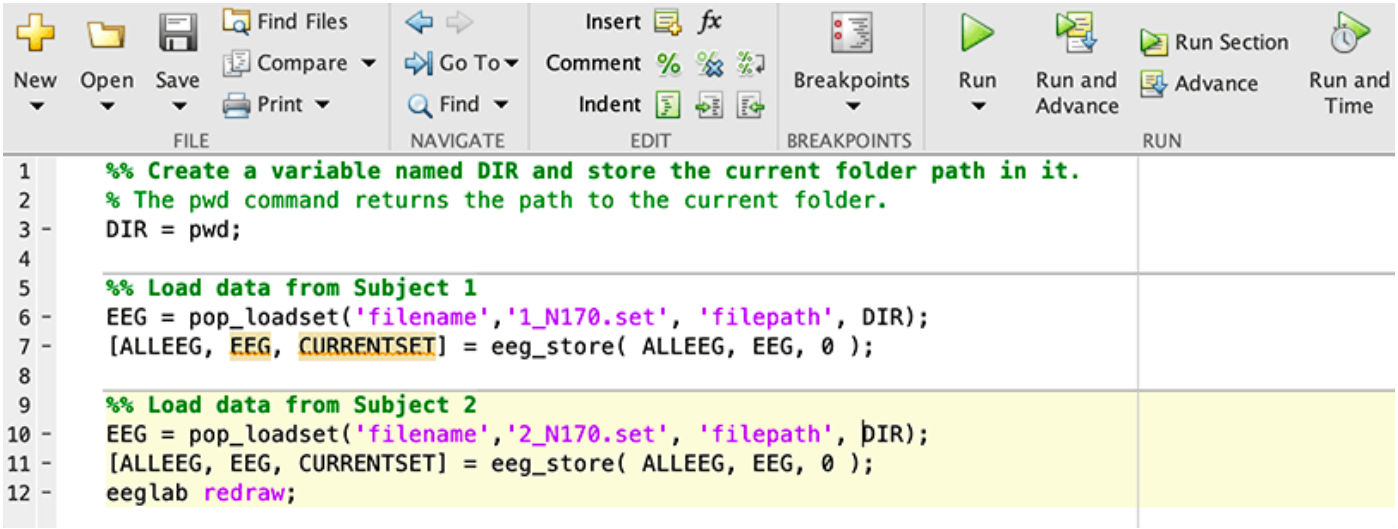
from the popup menu). Now the dataset should appear in the **Datasets** menu.

Now select and execute lines 9-12 to load the dataset for Subject 2 and update the GUI. You should now see the datasets from both Subject 1 and Subject 2 in the **Datasets** menu. Note that you didn't execute the lines of the script in order. You executed line 12 after lines 1-7 and then again after lines 9-11. This approach gives you a lot of flexibility. The lines in a script usually need to go in a particular order to give you the desired result. For example, you can't filter a file until after you've loaded it. But sometimes you're just using the script as a convenient place to store a bunch of commands that you might execute in some other order.

Now we're going to look at one more way to execute commands from a script, in which we divide the script into sections and run one section at a time. If you look at **Script2.m**, you'll see that some lines begin with a **%** symbol. This symbol indicates that the line is just a comment, not code that will be executed. You can also put a **%** symbol after a command to provide a comment about that command.

You should also note that some of the lines begin with two consecutive **%** symbols. This indicates the start of a new *section*. If you click anywhere within a section, the background color of that section changes. Screenshot 11.4 shows what it looks like when I click in the last section of the script. Once you've highlighted a section in this way, you can run that section of code by clicking the **Run Section** button in the tool bar near the top of the script editor window.

Screenshot 11.4



This page titled 11.8: Exercise- From the Command Line to a Script is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.9: Exercise- Using a Variable for the Path

In this exercise, we're going to take a closer look at **Script2.m**, which demonstrates how to use variables to specify the paths to your files, which is much more robust. Make sure that Chapter_11 is still the current folder, and type **pwd** (which stands for *print working directory*) on the command line. You should see the path for your current folder. Line 3 of **Script2.m** uses the **pwd** command to get the path to the current folder and store it in a variable named **DIR** (short for *directory*, but we could have named it almost anything). Run line 3 of the script (using any of the methods described in the previous exercise) and then look at the value of **DIR** (by typing **DIR** on the command line or by double-clicking it in the Workspace pane). You'll see that **DIR** is a string that holds that path to the current folder.

Now we can use **DIR** as the starting point for providing paths to our Matlab commands. This is much more robust than specifying something like **C:/ERP_Analysis_Book/Ch_10_Scripting** as the path. For example, if you move your data and scripts to a new location on your computer or a new computer, using **DIR** as the path will still work (because the **pwd** command will return the new location of the script), but specifying the path directly will now fail.

We didn't really need to specify the paths in **Script2.m**, because the files were located in the current folder. Close **Script2.m** and open **Script2b.m**, which shows a more realistic example. This new script assumes a more complicated but better organization for your data files, in which each participant has a separate folder inside of a general data folder.

This is the organization that I generally recommend for EEG and ERP data (whether or not you use EEGLAB and ERPLAB). You will end up with a lot of files for each participant, and this just keeps them well organized. There's some redundancy, because I also recommend including the Subject ID number in each filename. But this redundancy is useful because it minimizes the likelihood of errors. Nothing about EEGLAB or ERPLAB requires this organization, but it's used for all the scripts in this book. You can choose a different organization, but please don't just put all the files for all the participants into a single folder. It will seem simpler at first, but it will make your life much more difficult in the long run. You just don't want to have 3478 files in the same folder!

This organization means that you need a different path for each participant. In **Script2b.m**, we implement this by using **DIR** as a base folder (the folder that contains the script) and then use this to create a variable named **Subject_DIR** that holds the path for the participant we're currently processing.

Important note: This approach assumes that Matlab's current folder is set to be the folder that contains the script. You should make sure this is true when you run the scripts in this book.

Line 6 of **Script2b.m** sets the value of **Subject_DIR** like this:

```
Subject_DIR = [DIR '/N170_Data/1/']; % Folder holding data for this subject
```

Putting multiple character strings inside of square brackets causes Matlab to concatenate the strings together. For example, **['ERPs' 'Are' 'Great']** is equivalent to '**ERPsAreGreat**', and **['ERPs ' 'are ' 'great']** is equivalent to '**ERPs are great**'. If **DIR** has a value of **'C:/ERP_Analysis_Book/Ch_10_Scripting'**, then **[DIR '/N170_Data/1/']** has a value of **'C:/ERP_Analysis_Book/Ch_10_Scripting/N170_Data/1/'**.

To see this in action, run lines 3 and 6 of **Script2b.m** and then look at the value of **Subject_DIR**. This is what it looks like on my computer:

/Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Ch_10_Scripting/Exercises/N170_Data/1/

This may seem like a lot of work, but I guarantee that it will make your life easier in the long run. For example, if you develop a script for one experiment and want to use it for a second experiment, you can place a copy of the script in the folder for the second experiment, and the script will automatically look for the data files in the correct place.

If you're using a Windows machine, **Script2b.m** may not work properly because it uses slashes between folder names rather than backslashes. You can just replace the slashes in the paths with backslashes.

This page titled 11.9: Exercise- Using a Variable for the Path is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.10: Exercise- Loops

In this exercise, we'll introduce the concept of *loops*, which will make your life much easier by allowing you to automate the processing of multiple participants. There are several kinds of loops, but the most common is called a *for loop*. It allows you to repeat the same sequence of steps multiple times, but with one variable changing (e.g., a variable that indicates which participant should be processed). We're going to go through several examples in this exercise, starting simple and working our way up to a script that processes the data from multiple participants.

As usual, let's start by quitting EEGLAB, typing **clear all**, and restarting EEGLAB. Then open **Script3.m** by double-clicking on it in the Current Folder pane. Run the script (e.g., by clicking the **Run** button) and see what it does. You should see that it prints a list of numbers between 1 and 10. In a later script, these will be the ID numbers of our participants. Note that the number 5 is missing. This is meant to indicate that Subject 5 is being excluded (because that subject had too many artifacts and was excluded from the final analyses).

Now take a look at the script. The main body of the script is this:

```
for subject = [ 1 2 3 4 6 7 8 9 10 ]

   display(subject);

end
```

In Matlab, a **for** loop begins with **for** and ends with **end**. The lines of code between the **for** and **end** lines are the *body* of the loop. This body will be executed multiple times, once for each element specified in the array on the **for** line. The **for** line defines a variable (which we've named **subject** in this example) and specifies an array of values that will be stored in this variable as we go through the loop (**[1 2 3 4 6 7 8 9 10]** in this example).

In **Script3.m**, the body of the loop is a single line consisting of the **display(subject)** command, which just prints the value of the variable named **subject** that we specified on the **for** line. Note that it's conventional to indent the body of a loop using tabs. The tabs are ignored by Matlab, but they make it easier to see the structure of a script. Not required, but highly recommended!

Each time we go through the loop, the variable named **subject** is set to a new value in the array of values following the equals sign. When the loop starts, **subject** will be set to 1 (because 1 is the first value in the array). The **display(subject)** line will then execute, and it will display a value of 1 because that's the value of the **subject** variable. Then the **end** line occurs, telling Matlab to go back to the start of the loop and set **subject** to the next value in the array (2). The **display(subject)** line will then execute, but this time it will display a value of 2 because that's now the value of the **subject** variable.

Matlab will keep repeating the loop, setting **subject** to 3, then to 4, then to 6, etc. There is no 5 in our array of values, so **subject** will never take on that value. The array is just a list of values, and any sequence of values will work. For example, we could use **[3 1 5 9]** and then **subject** would take on that set of values in that order (3, then 1, then 5, then 9). We could even use non-integer numbers (e.g., **[5.23 6.1 -5.442 10021.2]**) or character strings (e.g., **['S1' 'S2' 'S3' 'S5']**). Matlab is much more flexible than most programming languages in this regard. Spend some time playing with the array in the script so that you get a good sense of how it works.

Now close **Script3.m** and open **Script3b.m**, which is a slightly more sophisticated version of the same set of ideas. Run the script to see what it does. You should see a set of lines starting with this:

```
Processing Subject 1

Processing Subject 2

Processing Subject 3
```

Now look at the script. You'll notice two main changes from the first script. First, instead of providing an explicit list of subject IDs in the **for** line, we've defined a variable named **SUB** that stores this array:

```
SUB = [ 1 2 3 4 6 7 8 9 10 ]; %Array of subject IDs
```

We then specify this variable as our array in the **for** statement:

for subject = SUB

This approach embodies my #1 principle of writing good code: All values used by a script should be defined as variables at the top of the script. I'll say more about this principle later in the chapter.

The second change to the script is that it uses the **fprintf** command instead of the **display** command to print the value of the **subject** variable. The **fprintf** command is much more powerful and flexible. It takes a little time to learn to use it, but it's well worth the time. Here's the specific version used in our script:

fprintf('Processing Subject %d\n', subject);

The first parameter in the **fprintf** command is a *formatting string*. It contains plain text that is printed by the routine, and it also includes formatting statements for variables that appear as subsequent parameters. The **%d** tells the command to print a whole number (the value of the **subject** variable), and the **\n** tells it to print a newline (a return). You can do much more than this with **fprintf**, but we're keeping it simple for now (see the **fprintf** documentation for details).

Once you understand how this script works, close it and open **Script3c.m**. If you run it, you'll see that the output has more information than provided by the previous script. For each iteration of the loop, it indicates how many times we've gone through the loop, plus the subject's ID. This allows us to see that the fifth subject has an ID of 6, not 5.

Look at the script to see how it works. Near the top, you'll see that we use a Matlab function called **length** to define a variable named **num_subjects** that stores the number of subjects in the **SUB** array:

num_subjects = length(SUB);

We've then used this new variable to define the array of values for the loop, which we've defined as **1:num_subjects**. Type **1:num_subjects** on the command line. You'll see that it is equivalent to a list of integers between **1** and **num_subjects** (rather than 1 through 10, skipping 5). As a result, we're no longer looping through the subject IDs. As a result, I've changed the name of the variable in the **for** statement to **subject_index**.

Each time through the loop, we get the subject ID by finding the element in **SUB** that corresponds to **subject_index** and store it as a text string in a variable named **ID**. For example, when **subject_index** is 5, we get the 5th element of **SUB**, which is 6 (because **SUB** skips subject 5). **SUB** is an array of numbers, but as you'll see in the next script, it's useful to store the ID as a text string. We therefore use a Matlab function called **num2str** to convert the number to a string before storing it in **ID**. Note that the format string for the **fprintf** command uses **%s** to indicate that this command should print a string variable for **ID**.

Why it Pays to Include Good Comments and Meaningful Variable Names in Your Scripts

When you're in the middle of writing a script to process the data for an experiment, you will get very focused on *getting the job done*. That is, you just want to script to work so that you can get to the next step of the project (and ultimately to the point of submitting a paper for publication). However, the fastest route to a goal is not always the straightest: If you focus too much on the immediate goal of getting the script to work, you may actually slow your progress toward the final goal of getting the paper submitted. It really pays to take your time when writing a script and write the code in a way that will be optimal in the long run.

In practice, this means following good coding practices that reduce the likelihood of errors, like defining all important values as variables at the top of the script. Errors can really slow you down if you don't realize the error until you're near the point of submitting the paper and now need to repeat all the analyses, change all the figures, and update the text of your paper. It's also important to realize that you will probably need to come back to your script many months after you've written it (e.g., when you're writing your Method section or you realize you need to reanalyze your data), and you will save yourself a lot of time if you write your code in a way that's easy to read later.

There are two straightforward ways of making your code more readable. The first is to use variable names that have an obvious meaning. For example, I could have used something like **ns** as the name of the variable that holds the number of subjects, but instead I used **num_subjects**. The second is to add lots of comments. For examples, take a look at the example scripts I created for this chapter. Of course, these scripts were designed to be read by other people, so I put more work into

that comments than I might have for a regular script that I wasn't planning to share. But I still include tons of comments in scripts that I don't plan to share. It's a gift to my future self, because I know I will probably need to come back to those scripts after months or even years, and the comments will make my life much easier then. And when I come back to a script with good comments, I always try to thank my past self for the gift.

You should also keep in mind that it's becoming more and more common to make your data and scripts available online when you publish a paper. This means that you're never really writing scripts just for yourself. Other researchers are going to be looking at your scripts, so you don't want the scripts to be embarrassing, and you want the other researchers to be able to understand your code. If you make the code easy to understand, this increases the likelihood that the other researchers will follow up on your research, which means that your research will have a larger impact. And aren't you doing research so that it has an impact?

I've found that people often spend a huge amount of time polishing their scripts (making them more logical and adding lots of comments) right before they're going to submit the paper for publication. They often find mistakes, and then they end up having to change their figures and the statistics in their Results sections. It's really inefficient. It makes much more sense to write your scripts with the intent of sharing them—including clear logic and lots of comments—right from the beginning. This takes a lot of discipline, because when you're writing the script you just want to get the job done. But this approach will save you a lot of time and agony in the long run.

This page titled 11.10: Exercise- Loops is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.11: Exercise- Looping Through Data from Multiple Participants

In this exercise, we'll get to see how loops are used to load the datasets from a set of participants. We'll add some processing steps that make the script more useful in the next exercise, but I wanted to keep things simple for now. To get started, quit EEGLAB, close any open scripts, type **clear all**, and open **Script4.m**. But don't launch EEGLAB—we'll have the script do that!

Go ahead and run **Script4.m** to see what it does. It should launch EEGLAB, load the datasets for Subjects 1-10 (except Subject 5), and refresh EEGLAB to make the datasets available in the **Datasets** menu.

Now let's look at the script and see how it works. The first line of code launches EEGLAB, which creates several variables that we will find useful (e.g., **EEG** and **ALLEEG**). The next line of code creates the **DIR** variable, as in the previous scripts, which holds the location of the script (and should be the **Chapter_11** folder). Then the script creates a new variable named **Data_DIR**, which appends **'/N170_Data/'** onto the **DIR** variable. This gives us a path to the folder containing the single-participant data folders.

The next step is to define a variable named **Dataset_filename,** which has a value of **'_N170.set'**. We'll eventually combine this variable with the subject ID to get the entire filename for a given participant (e.g., **1_N170.set**).

Then we define variables for the list of subjects and the number of subjects, just as in the previous example. Note that these steps embody the principle that all values used by a script should be defined as variables at the top of the script. It's a little extra up-front work to do this, but it dramatically reduces the likelihood of bugs later (especially when you take a previous script and modify for a new purpose).

The next step is to loop through the subjects. The first part of this is just like what we did in the previous script, including setting **ID** to be a string with the current subject's ID. Then the script creates a variable named **Subject_DIR**, which specifies the folder that holds that data for the subject currently being processed by the loop (e.g., **.../Chapter_11/N170/1/** for the first subject). We do this by concatenating the **Data_DIR** variable with the **ID** variable and then a **/** character. We also create a variable named **Subject_filename** by concatenating the **ID** variable with the **Dataset_filename** variable. This gives us a value of **1_N170.set** for the first subject.

We then load the dataset, using **Subject_filename** as the filename and **Subject_DIR** as the path. The dataset is stored in the **EEG** variable, and our last step in the body of the loop is to add this dataset to the **ALLEEG** variable using the **eeg_store** routine. The zero we specify as the last parameter for this routine tells it to add the new dataset to the end of **ALLEEG**.

After the loop finishes, **eeglab redraw** is called to update the EEGLAB GUI.

There are actually 40 participants in this experiment, each with a dataset. This script is a much faster way of loading these 40 datasets than using the GUI to separately load each one. Because all the key values are specified as variables at the top of the script, you can easily find them and modify them so that you can use the same script with another experiment, assuming that the data are organized in the same way on your computer. You'd just need to modify the list of subject IDs (the **SUB** variable), the name of the folder holding the data (**Data_DIR**), and the base dataset name (**Dataset_filename**). This will be much faster and easier if you're consistent in how you organize the data for each experiment (see the text box below).

Consistency

There is a famous line from the poet Ralph Waldo Emerson that is frequently misquoted as "Consistency is the hobgoblin of little minds." People sometimes use this incorrect version of the quote to belittle people for being consistent. However, the actual quote is "A foolish consistency is the hobgoblin of little minds" (Emerson, 1841 p. 14; my emphasis). It's not the least bit foolish to be consistent about your data organization, your filenames, your variable names, etc. You will save yourself huge amounts of time and grief by developing a good organizational strategy early in your career and then sticking to it (but with thoughtful changes when necessary).

This page titled 11.11: Exercise- Looping Through Data from Multiple Participants is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.12: Rapid Cycling Between Coding and Testing

If you think about the exercises we've done so far, you'll realize that we started with an extremely simple script (**Script1.m**) that simply loaded a file, improved it (**Script2.m** and **Script2b.m**), and then added a bit more functionality (**Script3.m** and **Script4.m**). This is a good way to learn, but it's also the way you should write your scripts. Don't try to write 30 lines of code (or an entire script) without doing any testing along the way. If you write a 30-line script, you will probably have 8 different errors in the script, and it will be really hard to figure out what's going wrong.

Instead, the best approach is to write a small amount of code, test it, debug it if necessary, and then add more code. At this point, you should be adding only 1-3 lines of code at a time. As you gain experience, you can write more lines before testing, but even an experienced programmer usually does some testing after every 20-40 new lines of code.

I almost put this piece of advice in a text box, but I decided that it's so important it deserves it's own section!

This page titled 11.12: Rapid Cycling Between Coding and Testing is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

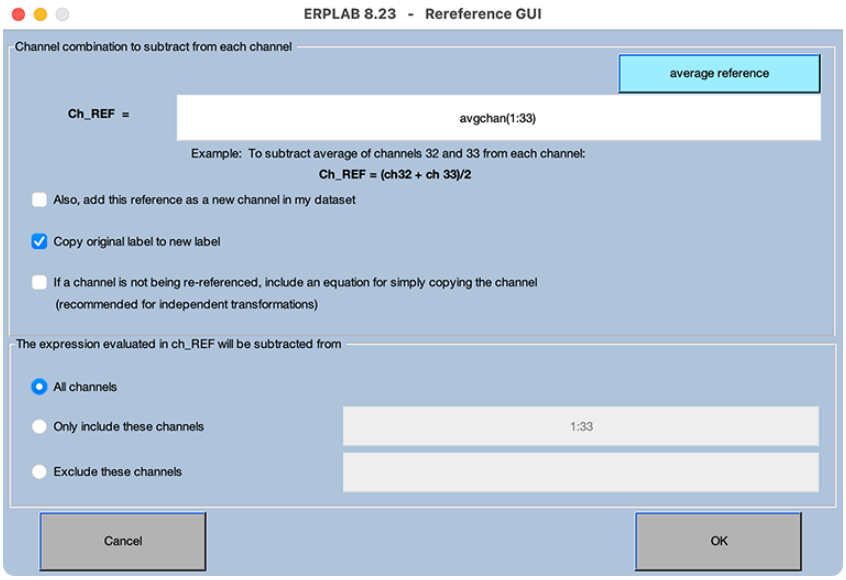
11.13: Exercise- Referencing with a Script

In this exercise, we'll add referencing to the script from the previous exercise so that the new script is more useful. You'll also learn some important strategies along the way.

As you may recall, the ERP CORE data were recorded without a reference (which is possible in only a few recording systems), so we need to use EEG Channel Operations to reference the data. We'll also add bipolar VEOG and HEOG channels. The N170 is traditionally referenced to the average of all sites (see Chapter 5), and that's what we'll do in this example.

You should already have 9 datasets loaded from the previous exercise. If you don't, run **Script4.m** again. With the **10_N170** dataset active, select **EEGLAB > ERPLAB > EEG Channel operations**. Clear out any equations that are already set, set the **Mode** to **Create new dataset**, and make sure the box labeled **Try to preserve channel locations** is checked. Click the **Reference Assistant** button, and enter the parameters shown in Screenshot 11.5.

Screenshot 11.5



The key parameter is **avgchan(1:33)**, which tells it to use the average of all 33 channels as the reference. Then click **OK**, which should fill the equations window with an equation for referencing each of the 33 channels to the average of Channels 1–33. Then add these two equations to create bipolar HEOG and VEOG channels:

nch34 = ch31 - ch32 Label HEOG-bipolar

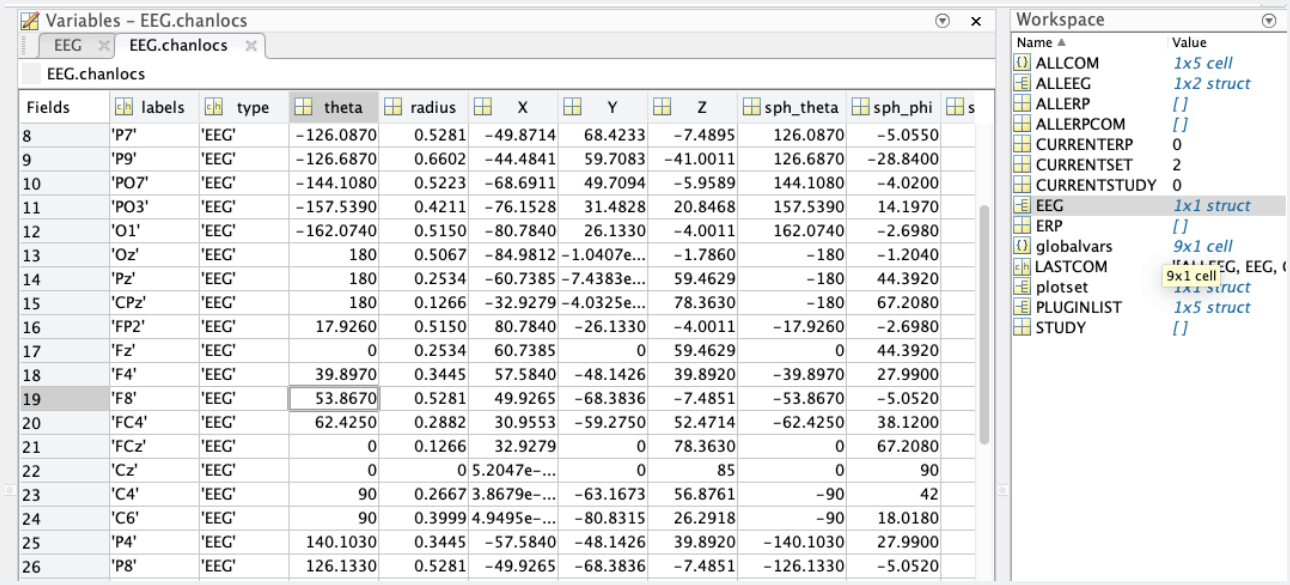
nch35 = ch33 - ch16 Label VEOG-bipolar

Then click **Run** to execute the routine. Now type **eegh** to see the script command for EEG Channel Operations, which begins with **EEG = pop_eegchanoperator**. This command is extremely long, because it contains all 35 equations in it. There's a better way to do this: Instead of having this huge command, we can put all the equations in a file and pass the filename instead of the list of equations.

Seeing a List of the Channels in a Dataset or ERPset

When you're using the EEGLAB/ERPLAB GUI, it's usually pretty easy to see what channels are in a dataset or ERPset. However, this information isn't so obvious when you're writing scripts. Here's a little trick that I use quite often: If you double-click on the name of a variable in the Workspace pain of the main Matlab window, a *Variables* pane will appear and show the contents

of that variable. For example, you can double-click on the **EEG** variable to see its contents. **EEG** is a complex structure with many fields, and you can double-click on the name of a field to see the contents of that field. The screenshot below shows what I see when I first double-click on the **EEG** variable and then double-click on the **chanlocs** (channel locations) field. This also works for ERPsets if you double-click on the **ERP** variable.



Fields	labels	type	theta	radius	X	Y	Z	sph_theta	sph_phi	s
8	'P7'	'EEG'	-126.0870	0.5281	-49.8714	68.4233	-7.4895	126.0870	-5.0550	
9	'P9'	'EEG'	-126.6870	0.6602	-44.4841	59.7083	-41.0011	126.6870	-28.8400	
10	'PO7'	'EEG'	-144.1080	0.5223	-68.6911	49.7094	-5.9589	144.1080	-4.0200	
11	'PO3'	'EEG'	-157.5390	0.4211	-76.1528	31.4828	20.8468	157.5390	14.1970	
12	'O1'	'EEG'	-162.0740	0.5150	-80.7840	26.1330	-4.0011	162.0740	-2.6980	
13	'Oz'	'EEG'	180	0.5067	-84.9812	-1.0407e...	-1.7860	-180	-1.2040	
14	'Pz'	'EEG'	180	0.2534	-60.7385	-7.4383e...	59.4629	-180	44.3920	
15	'CPz'	'EEG'	180	0.1266	-32.9279	-4.0325e...	78.3630	-180	67.2080	
16	'FP2'	'EEG'	17.9260	0.5150	80.7840	-26.1330	-4.0011	-17.9260	-2.6980	
17	'Fz'	'EEG'	0	0.2534	60.7385	0	59.4629	0	44.3920	
18	'F4'	'EEG'	39.8970	0.3445	57.5840	-48.1426	39.8920	-39.8970	27.9900	
19	'F8'	'EEG'	53.8670	0.5281	49.9265	-68.3836	-7.4851	-53.8670	-5.0520	
20	'FC4'	'EEG'	62.4250	0.2882	30.9553	-59.2750	52.4714	-62.4250	38.1200	
21	'FCz'	'EEG'	0	0.1266	32.9279	0	78.3630	0	67.2080	
22	'Cz'	'EEG'	0	0	5.2047e-...	0	85	0	90	
23	'C4'	'EEG'	90	0.2667	3.8679e-...	-63.1673	56.8761	-90	42	
24	'C6'	'EEG'	90	0.3999	4.9495e-...	-80.8315	26.2918	-90	18.0180	
25	'P4'	'EEG'	140.1030	0.3445	-57.5840	-48.1426	39.8920	-140.1030	27.9900	
26	'P8'	'EEG'	126.1330	0.5281	-49.9265	-68.3836	-7.4851	-126.1330	-5.0520	

To see how this works, select the original version of **10_N170** from the **Datasets** menu and launch **EEG Channel operations** again. You should still have the 35 equations from before. Near the bottom of the window, click the **Save list as** button to save the list of equations as a file. Name the file **equations.txt**. In addition to saving the file, this command puts the filename (including the entire path) in the **File:** text box. Now, select the check box labeled **Send file rather than individual equations**, and click **Run** to execute the routine. Now when you type **eegh** on the command line, you should see a much shorter line of code that looks something like this:

EEG = pop_eegchanoperator(EEG, '/Users/luck/Ch_10_Scripting/Exercises/equations.txt' , 'ErrorMsg', 'popup', 'KeepChLoc', 'on', 'Warning', 'on');

Let's take a closer look at this command. We send the **EEG** variable as the first parameter. The routine then sends back a new version of this variable as its output. The second parameter is the filename for the equations file (including the whole path, which will be different on your computer). Then we have a sequence of pairs of parameters, which occur in the order *parameter name*, *parameter value*. For example, the sequence of parameters **'ErrorMsg', 'popup'** tells the routine to set the error message option to a value of **popup**, which means that error messages will be delivered in popup windows. The next pair of parameters is **'KeepChLoc', 'on',** which tells the routine that the option for keeping the channel locations should be on (which you had set from the GUI by checking the box the option labeled **Try to preserve channel locations**). The use of pairs of parameters—one for the name of the parameter and one for the value—makes it easier to understand what the parameters are doing. This approach is common in Matlab scripts and functions, and it's used extensively in EEGLAB and ERPLAB routines.

How do you know what options are available for a command like **pop_eegchanoperator**? You can use Matlab's **help** feature. There are several ways to access this feature. One common way is to type **help pop_eegchanoperator** on the command line. Another way is to select the name of the routine in the Command Window (or in a script) and then right-click or control-click on it to get a contextual menu, which contains a **Help on Selection** item. Try one of these methods to see the help for **pop_eegchanoperator**. When you do this, Matlab simply shows you all of the comments (lines of text beginning with the **%** character) at the top of the **.m** file containing the command (e.g., the **pop_eegchanoperator.m** file, which contains the code for this routine and is located in a subfolder deep inside the EEGLAB folder that was created when you installed EEGLAB and ERPLAB). This works with any command, including built-in Matlab commands like **pwd** and EEGLAB and ERPLAB commands like **pop_eegchanoperator**.

Now it's time for an embarrassing admission: When I looked up the help for **pop_eegchanoperator**, I found that it didn't actually list the optional parameters. In fact, it wasn't very helpful at all. This is one of the first routines that we wrote for ERPLAB, and I'm guessing that we hadn't yet established a good workflow that included careful documentation of all the features of each routine. This has now been fixed. And here's another embarrassing admission: I've seen lots of shortcomings in the ERPLAB help

information as I've worked on this book. I don't have a good excuse for this. I can tell you that most programmers hate writing documentation—once the code is done, it's not much fun to write the documentation. But that's not a good excuse. I'm going to add "go through all the ERPLAB help" to my list of things to do after I finish this book.

But let's turn lemons into lemonade and use this as an opportunity to learn how to overcome unclear or missing documentation.

Lemonade

There's an old saying that "When life gives you lemons, make lemonade." I prefer the snarkier version, "When life gives you lemons, squeeze them into the open wounds of your enemies."

When I ran into this problem with **pop_eegchanoperator**, I simply opened the **pop_eegchanoperator.m** file and looked at the code. I didn't have to go searching through my file system for this file. I just found **pop_eegchanoperator** in the Command Window, selected it, control-clicked on it to get a contextual menu, and selected **Open Selection**. Give that a try to open the **pop_eegchanoperator.m** file.

If you're not an expert in Matlab scripting, you may wonder how you're going to understand the code in the **pop_eegchanoperator.m** file? You're interested in finding out about the options, and you know that one of the options is **ErrorMsg**, so it would make sense to do a search for that string in the file. If you do this, you'll eventually find this line:

```
p.addParamValue('ErrorMsg', 'cw', @ischar); % cw = command window
```

From this line you can infer that a possible value for this parameter is **'cw'**, and that this will send the error messages to the Command Window. **I**f you search some more, you'll find this set of lines:

```
if strcmpi(p.Results.ErrorMsg,'popup')

    errormsgtype = 1; % open popup window
```

From these two lines, you can see that the **'popup'** value opens a popup window. So, now you've learned a strategy for figuring out how a routine works and what the options are (and I've learned that I need to put more effort into making sure that the ERPLAB help information is complete).

Now that you've learned about the command for EEG Channel Operations, let's add it to the script we started in the previous exercise. Open **Script4.m** (if it's not already open), and save it as **MyScript4.m**. Copy the **pop_eegchanoperator** line from **eegh** history in the Command Window to the clipboard and then paste it into **MyScript4.m** right after the line that begins with **EEG = pop_loadset**. Make sure you've pasted the version that uses the **equations.txt** file. Then save the updated **MyScript4.m** file. (I always save a script before running it—if there's an error that crashes Matlab, I don't want to lose the new code that I just added.)

Now quit EEGLAB, type **clear all**, and run **MyScript4.m** to see it work. You should see EEGLAB launch and then lots of text appear in the Command Window (mainly from **EEG Channel Operations**), including a line that says something like **Processing Subject 9 of 9, ID = 10** as each individual subject is being processed. Once the script finishes, you should plot the EEG from one of the datasets, and then you'll see that you now have bipolar VEOG and HEOG channels.

If you run into an error, make sure that you followed the above instructions carefully, and think about the logic of what you're doing. You can also consult the Troubleshooting Tips in Appendix 2. The next several sections of this chapter will require you to make various additions to **MyScript4.m**, and you'll almost certainly make at least one error at some point. When you do, feel free to make a few unkind comments about my parentage, and then remember that errors are a big part of scripting and that one of the goals of this book is for you to learn how to find and fix problems. Appendix 2 ends with some recommendations for writing code in a way that reduces the likelihood of errors.

After you get the script to run correctly, I want you to spend a moment thinking about how easy it was to add the step for referencing the data to the prior script. Once you had the basic script for looping through the participants and opening their datasets, you just had to run EEG Channel Operations from the GUI, get the history, and copy the command from the history to the appropriate point in the script. Done!

Well, not so fast. You'll usually want to make a few changes to the command that you copied from the history. For example, the new line of code you added to the script broke my rule about values (such as filenames) being defined as variables at the top of

the script. If you moved this script and the **equations.txt** file to a new folder, the script would break. The next exercise will solve this problem and make some additional improvements.

This page titled 11.13: Exercise- Referencing with a Script is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.14: Exercise- Improving the Referencing Script

In this exercise, we're going to make some improvements to the script from the previous exercise, including defining a variable for the name of the equations file and saving the dataset to your hard drive.

Make sure that **MyScript4.m** is still open. At the end of the set of lines that define the other folders and filenames, add this line:

Equations_filename = [DIR 'equations.txt'];

That line will create a variable that holds the path and filename of the file with the equations. It's really easy to make an error when you create a variable like this. In fact, I initially made an error when I wrote that line of code. When you create a new variable for a path and filename, you should check it first before running your script. To do this, run that one line of code. (The **DIR** variable should already be set from when you ran the script earlier.) Then, type **dir(Equations_filename**) on the command line. The **dir** command lists any files that match the variable you specify inside the parentheses. When you run this command, you should get an error message saying that the file is not found.

Now type **Equations_filename** on the command line to see the value of this variable. Do you see the problem? We're missing the slash between the path and the filename (or a backslash on a Windows computer; see the text box below). Fix this by changing the code to:

Equations_filename = [DIR '/equations.txt'];

Now run that line of code and then the **dir(Equations_filename**) command. If **Equations_filename** is set correctly, the **dir** command should just print the name of the file (**equations.txt**).

Now replace the string holding the path and filename in the **pop_eegchanoperator** command with **Equations_filename** so that the command looks like this:

EEG = pop_eegchanoperator(EEG, Equations_filename , 'ErrorMsg', 'popup', 'KeepChLoc', 'on', 'Warning', 'on');

Quit EEGLAB, type **clear all**, and then save and run the updated script. It should lead to the same result as the previous version of the script that didn't use the **Equations_filename** variable, but the script is now more robust.

Forward Slashes and Backslashes

Windows uses a backslash (\) to separate the elements of a path, whereas Mac OS, UNIX, and Linux use a forward slash (/). Matlab is smart enough to convert to the appropriate separator for your computer. I don't really trust it, though, so I like to use a more explicit approach using a Matlab function named **filesep**. This function returns the appropriate file separator for the computer that the script is running on. For example, I would define the variable for the equations file as **[DIR filesep 'equations.txt']** rather than **[DIR '/equations.txt']** or **[DIR '\equations.txt']**. If you type **filesep** on the command line, you'll see that it returns **'/'** if you're running Mac OS, UNIX, or Linux and **'\'** if you're running Windows.

Let's make a couple more improvements. First, when a script is running, you don't usually want error and warning messages to appear as new windows that wait for you to respond. If you're running a script overnight because it takes a long time, you don't want the script to be waiting for a response to a warning message when you arrive the next morning. So, let's change **'ErrorMsg', 'popup'** parameters in the **pop_eegchanoperator()** function to **'ErrorMsg', 'cw'** so that error messages are printed in the Command Window and change **'Warning', 'on'** to **'Warning', 'off'** so that warning messages aren't printed. (But note that error messages will still be printed.)

Now let's make some changes that will have a more direct impact. First, we're going to add **_ref** to the end of the name of the dataset to indicate that the data have now been referenced. Type **EEG.setname** on the command line so that you can see what the current setname is. If Subject 10 is the active dataset, you should see that the setname is **'10_N170'**. To update the setname, add the following line to your script after the line with the **pop_eegchanoperator** command:

EEG.setname = [EEG.setname '_ref'];

This takes the current setname and appends **_ref** to the end, storing the result in the **EEG.setname** variable. That was pretty easy! Note that this is right before the line with the **eeg_store** command, so the new setname will be stored in **ALLEEG** and therefore appear in the **Datasets** menu once we get to the **eeglab redraw** command. This line of code violates my rule about using variables to define values, because I didn't define a variable for **'_ref'**. Rules are made to be broken, but only when there's a good reason. In this case, the **'_ref'** string is something that I always use, so putting it into the body of the script is unlikely to cause problems later.

Now let's add some code for saving the referenced dataset to your hard drive, in the same folder as the original dataset. You should first use the GUI to save the current dataset on your hard drive so that you can use the history to see the command for saving a dataset file. Select **EEGLAB > File > Save current dataset as**, and use **tmp.set** as the filename (in the **Chapter_11** folder). Because you're running this routine to see the history, you don't actually need the file, and this filename will help you remember that it's a temporary file that you can delete at some point. Now type **eegh** to see the history. You should see something like this:

EEG = pop_saveset(EEG, 'filename','tmp.set','filepath','/Users/luck/Ch_10_Scripting/Exercises/');

Copy that line from the Command Window and paste it into your script after the line with the **eeg_store** command. But we want to make some changes before we use it. First, we don't want the file to be named **tmp.set**. I like to have the filename be the same as the setname, but with **.set** on the end. To accomplish this, change the new line of code by replacing **'tmp.set'** with **[EEG.setname '.set']** (which creates a string with the setname followed by **.set**).

We also want to change the path so that it uses the **Subject_DIR** variable we previously created to hold the path for this subject's data. To do this, replace the string that currently lists the path (which is **'/Users/luck/Ch_10_Scripting/Exercises/'** on my computer but will be something else on your computer) to **Subject_DIR**. This line of code should now look like this:

EEG = pop_saveset(EEG, 'filename', [EEG.setname '.set'], 'filepath', Subject_DIR);

Before you run the code, I want to show you one more little trick for dealing with bugs and other kinds of errors. I probably made an error at least 50% of the time when I added code to one of the scripts for this book. If you have a script that loops through the data from 40 participants, it can be really annoying to have the script show an error message 40 times before it finishes. You can usually type ctrl-C to stop the code after the first error, but that doesn't always work. So, the first time I try out a new script, I just try it with the data from the first participant. I do that by following the line that defines the subject IDs with another line that lists only the first subject ID. For example, near the top of MyScript4.m, I have these two lines:

SUB = [1 2 3 5 6 7 8 9 10]; %Array of subject numbers

%SUB = [1]; %Just the first subject

The first time I run the script after making a change, I simply remove the **%** symbol on the second of the two lines. This overwrites the original contents of **SUB** and replaces it with only the ID for Subject 1, and the loop runs only for this one subject. Once I get the script to work for that one subject, I then put the **%** symbol back and run the script again.

To see this in action, add the line with **SUB = [1];** (without the **%** symbol). Now you can run the new version of the script that changes the setname and saves the dataset as a file. If you get an error, or it doesn't seem to work correctly, you can fix it and try again. Once you get the script to work without any problems, you can comment out that line (i.e., add the **%** symbol) and run the script again to process all the subjects.

Now you have a robust, well-designed script that loads the datasets, references the data, saves the referenced data to your hard drive, and makes the datasets available in the EEGLAB GUI. You can now open **Script4b.m** to see my version of this script. It has comments that explain each line of code.

Script4b.m also uses a slightly different version of the equations file, named **Reference_Equations.txt**, which reorders the channels in addition to referencing them. The order of the channels in the original data files reflects the order of electrodes in the BioSemi recording system we used to collect the data. This ordering is convenient for the process of placing the electrodes on the head, but not for viewing the data. I reordered the channels so that they are in sets that go from left to right for a given line of electrodes (e.g., F7, F3, Fz, F4, F8 for the Frontal channels).

This page titled 11.14: Exercise- Improving the Referencing Script is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.15: Exercise- Preprocessing the EEG and Using a Spreadsheet to Store Subject-Specific Information

As described in earlier chapters, I strongly recommend that you go through each step of EEG processing manually, looking at the data from each step, before running a script. Actually, I don't merely recommend it, I insist on it! This will allow you to catch problems that would otherwise contaminate your data. It will also let you determine some subject-specific parameters, such as which channels should be interpolated and what artifact detection parameters should be used. But if some parameters differ across subjects, how can you use one script that processes the data from all the subjects?

The answer is to store the subject-specific parameters in a spreadsheet, and then have your script read the parameters from that spreadsheet. This exercise is designed to show you how this works in the context of interpolating bad channels and other preprocessing steps. The interpolation process won't work properly if there are big voltage offsets in the data, so we'll apply a high-pass filter that eliminates these offsets prior to interpolation. Also, we need to figure out the 3-dimensional locations of the electrodes, because this information is needed by the interpolation routine (which uses the distance between the to-be-interpolated electrode and the other electrodes to compute sensible, distance-weighted values).

I'm not going to go through the process of running these routines in the GUI and looking at the history to see the corresponding Matlab code. I assume you now understand that process, so I'll go directly to the code. You can look up the help information for these routines if you want a better understanding of the available options.

Start by quitting EEGLAB, typing **clear all**, and loading **Script5.m**. Let's take a look at the script before running it. The first thing to notice is a variable named **Interpolation_filename** that holds the name of an Excel file, **interpolate.xlsx**. This file contains information about which channels are bad and should be interpolated. Take a look at the file in Excel (or import it into Google Sheets or some other spreadsheet program). Here's what you should see:

Table 11.2. Spreadsheet of information for interpolation (interpolate.xlsx).

ID	Bad_Channels	Ignored_Channels	Channel_Names
1	[6 13]	[31 32 33]	C5, Oz
2		[31 32 33]	
3	[25]	[31 32 33]	P8
4		[31 32 33]	
6		[31 32 33]	
7		[31 32 33]	
8		[31 32 33]	

ID	Bad_Channels	Ignored_Channels	Channel_Names
9		[31 32 33]	
10		[31 32 33]	

The first column contains the Subject ID values (without Subject 5, who we're still excluding). The second column indicates the bad channels (if any) corresponding to the Subject ID values. They're specified using square brackets so that they will be interpreted as arrays by Matlab. You'll see why that's important in a bit. The next column indicates which channels should be excluded when we compute the interpolated values (the EOG channels). Those are the same for each participant, so they could be listed in the script, but I found it more convenient to put them into the spreadsheet. The last column shows the names of the bad channels. This column isn't used by the script, but it's nice to have that information when you're looking at the spreadsheet.

If you look closely at the contents of the cells (e.g., by looking at the Formula Bar), you'll see that the value for each cell begins with a single quote (except for the ID values). This tells Excel that the contents of that cell should be treated as a text string and never interpreted as a number. I've found that this can avoid problems when we read the values into Matlab, because we want every cell in a column to be the same data type. You should also note that the column labels don't have any spaces or special characters in them (except for the underscore character). The Matlab routine we'll use to read the spreadsheet will use the column labels to create new variables, so the labels need to be legal Matlab variable names.

Now look at the script again and find the line with the **readtable** command. This is a Matlab function that reads data from a file into a special *Table* data structure. It's a very powerful function that can read from many different file types. It uses the filename to determine what kind of file is being read (e.g., the **.xlsx** filename extension is used to indicate that it's an Excel XML file). It creates a Table from the data file, and we've told it to store this Table in a variable named **interpolation_parameters**.

To see how this works, run the first part of the script, starting with the **DIR = pwd** line and going through the **interpolation_parameters = readtable(Interpolation_filename)** line. Then double-click on the **interpolation_parameters** variable in the Workspace pane so that you can see the contents of this variable. You'll see that it contains the same rows and columns that were in the spreadsheet. Now type **interpolation_parameters.Bad_Channels** on the command line. You'll see a list of the bad channels for each subject:

```
9×1 cell array

{'[6 13]'}

{0×0 char}

{'[25]'  }

{0×0 char}

{0×0 char}

{0×0 char}

{0×0 char}

{0×0 char}

{0×0 char}
```

You can see that this list is in a special Matlab-specific format called a *cell array*. Cell arrays are a little difficult to understand and tricky to use correctly. This is especially true for beginners, but I still often make mistakes when I try to use them. At some point you'll need to learn about them, because they're very useful, but for now you can rely on code that I wrote that extracts the

contents of the **interpolation_parameters** table into a set of simple numeric arrays.

This code is embedded within the loop in **Script5.m**. Let's execute the code, but without actually going through the whole loop. To do this, first type **subject = 1** on the command line so that the looping variable has the correct value for the first subject. Then execute the **ID = num2str(SUB(subject))** line in the body of the loop, because we're going to need this variable. Now execute the three lines of code beginning with **table_row =**. The first of these lines determines which row of the **interpolation_parameters** table contains the values for this subject. The second line gets the array of bad channels for this subject and stores it in a variable named **bad_channels** (which has zero elements if there are no bad channels). The third line gets the array of to-be-ignored channels and stores it in a variable named **ignored_channels**. If you're not already an experienced Matlab programmer, the code on those lines probably looks like hieroglyphics—like I said, cell arrays are a little complicated. Once you're more familiar with Matlab coding, and you've wrapped your brain around cell arrays, you should come back to this code and figure out how it works. But for now, you can treat it like a bit of magic that gets you the information you need.

Now you should inspect the contents of **bad_channels** and **ignored_channels**, either by typing the variable names on the command line or looking at them in the Workspace. You'll see that **bad_channels** is an array with the values 6 and 13 (the two bad channels for Subject 1), and **ignored_channels** is an array with the values 31 through 33 (the three channels we will be ignoring when computing interpolated values for the bad channels).

The next line of code does the interpolation using the **pop_erplabInterpolateElectrodes** function. You can see that we send the **bad_channels** and **ignored_channels** variables to this function. But don't run this line of code yet, because we haven't run all of the preceding lines in the body of the loop. Let's take a look at those lines before we run them.

The body of the loop begins by setting some variables and loading the dataset, just as in the previous script. Then it runs a routine called **pop_erplabShiftEventCodes**, which shifts all the stimulus event codes to be 26 ms later. This is necessary because there is a fairly substantial delay between when an LCD display receives an image from the computer's video card and when it actually displays that image. We measured that delay using a photosensor and found that it was 26 ms. We therefore shift the event codes so that they occur at the actual time that the stimulus appeared instead of at the time when the image was sent to the display. If you're using an LCD display, you must do this. If you don't know how, contact the manufacturer of your EEG recording system.

The next step uses the **pop_basicfilter** function to run a bandpass filter with a bandpass of 0.1 to 30 Hz (12 dB/octave, which corresponds to a *filter order* of 2 in the code). Filtering out the low frequencies is essential prior to interpolation, because otherwise the random voltage offsets in each channel will produce a bizarre scalp distribution and the interpolation algorithm (which assumes a smooth scalp distribution) will produce bizarre results. Note that it is a good idea to remove the DC offset before filtering continuous EEG data, and this is implemented by specifying **'RemoveDC', 'on'** when we call the **pop_basicfilter** function.

The next step runs the **pop_chanedit** function to add information about the 3D location of each electrode site based on the electrode names. The function uses a file named **standard-10-5-cap385.elp** that is provided by EEGLAB. It contains a list of standard electrode names (e.g., **CPz**) and their idealized locations on a spherical head. This doesn't give you the true location for each participant, which would require using a 3D digitization system, but it's a good enough approximation for the interpolation process.

The next few lines extract the information from the **interpolation_parameters** table and run the interpolation routine.

Finally, we reference the data and save the dataset to the hard drive (just as in the previous exercise).

Note that, for each of these processing operations, we send the dataset to the routine in the **EEG** variable, and then the routine returns a modified dataset that we store in the **EEG** variable. In other words, the new dataset overwrites the old dataset in the **EEG** variable. That's much more efficient than storing the result of each new operation in **ALLEEG**. But note that keeping the individual datasets makes sense when you're processing the data in the GUI, because you want the flexibility of going back to a previous dataset.

Now that we've looked at the code, let's run the script, but only for the first subject. To limit it to the first subject, make sure that the **SUB = [1]** line near the top isn't commented out. Then run the script. You can now see that the script has created a new dataset file named **1_N170_shift_filt_chanlocs_interp_ref.set** in the **Chapter_11 > N170_Data > 1** folder.

However, this dataset isn't visible in the **Datasets** menu. **Script5.m** differs from the previous example scripts in that it doesn't make the datasets available in the EEGLAB GUI. It just does the processing and saves the results for each participant in a dataset file. As you'll see later in the chapter, you'll often have a series of scripts for processing a given experiment (e.g., one for the

initial preprocessing, another for ICA, another for post-ICA EEG processing, and another for dealing with averages). Each of these scripts will read in the files created by the previous script, so there's often no need to make the datasets available in the GUI.

If you do want to look at the results of a given script, you'll want a convenient way of reading in the files that were just created. **Script5.m** accomplishes this by including code at the end for loading the new dataset files into **ALLEEG**. This makes it possible for you to access these datasets from the EEGLAB GUI. This code is preceded by a **return** command, which causes the script to terminate, so the code at the end won't ordinarily execute when you run the script. But after you run the script, you can just select the code at the end and run it manually (e.g., by clicking the **Run Section** button at the top of the script editor window). Give it a try, and then verify that you can see the new dataset in the **Datasets** menu.

Plot this new dataset with **EEGLAB > Plot > Channel data (scroll)**. Then load the original dataset (**1_N170.set**) for this participant from the **Chapter_11 > N170_Data > 1** folder and plot it as well. You'll see that the new dataset is much smoother and doesn't have large DC offsets (because it has been bandpass filtered). And you can see some huge artifacts in the C5 and Oz channels in the original data that are gone in the new data (because the script interpolated those channels). You should also notice that the channels are in a more convenient order in the new dataset than in the original dataset as a result of the EEG Channel Operations step near the end of the loop.

Now comment out the **SUB = [1]** line near the top of the script and run the script again to process the data from all 9 participants. You can then select and execute the code at the end of the script for loading the new datasets into the EEGLAB GUI.

At this point, I'd like to remind you of some advice I gave you at the beginning of the chapter: Play! The best way to understand how these scripts actually work is to play around with the code. If you're not sure you understand one of the steps, try changing it to see what happens. But don't just do this randomly. Come up with hypotheses and test them by means of experimental manipulations (i.e., by changing the code and seeing if the results confirm or disconfirm your hypotheses). You can also try modifying the scripts to process your own data. Unless you're a very experienced programmer, you probably won't actually understand the key points from this chapter unless you engage in this kind of active exploration. And now is a great time to play around with the code, because the rest of the chapter assumes that you fully understand the basics and are ready to put them together into a complete data processing pipeline.

This page titled 11.15: Exercise- Preprocessing the EEG and Using a Spreadsheet to Store Subject-Specific Information is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.16: Exercise- Building an Entire EEG Processing Pipeline

You've now learned the basics of scripting, so we're ready to build an entire EEG preprocessing pipeline. This pipeline will execute all the steps prior to averaging. It closely matches the example pipeline described in Appendix 3. Here, we've divided the pipeline into five separate scripts:

- **Step1_pre_ICA_processing.m**- Performs the initial preprocessing steps prior to ICA-based artifact correction
- **Step2_ICA_phase1.m**- Creates an optimized dataset for the ICA decomposition
- **Step3_ICA_phase2.m**- Performs the ICA decomposition
- **Step4_ICA_phase3.m**- Transfers the ICA weights from the optimized dataset to the original dataset and reconstructs the data without the artifactual ICs
- **Step5_post_ICA_processing.m**- Performs the steps following artifact correction that are necessary prior to averaging

I encourage you to divide your EEG and ERP processing into multiple scripts in this manner. A general principle of good programming is to divide a complex job into a set of small, independent modules. This makes each part of the job simpler and less error-prone. It also makes it easier for you to find and fix problems. And it makes it easier for you to reuse your code for future experiments. Ideally, none of your scripts should be more than about 200 lines long (including comments). If you find a script is getting a little long, try to figure out how to break it up into a sequence of smaller scripts.

Step1_pre_ICA_processing.m is similar to **Script5.m** from the previous exercise, but with two major changes. First, interpolation has been removed from this script and moved to a later stage, following artifact correction. Second, the new script references the data to O2 rather than to the average of all sites. This is because using the average of all sites as a reference makes ICA complicated (see the chapter on artifact correction). After the ICA step, we'll re-reference to the average of all sites. By the way, I could have used any site as the reference at this initial stage. See the following text box for more information.

Recognizing a Conceptual Error

When I first started writing these scripts, I referenced the data to the average of all sites at the beginning of the pipeline (because this is standard reference for the N170). I had gotten through the stage of performing the ICA decomposition, and I started going through the data to determine which ICs should be removed (using the process described in the chapter on artifact correction). The ICs from the first participant looked okay but not great. The time course of one of the top ICs had a lot of weird high-frequency noise that I wasn't seeing in the EEG. The ICs from the second participant were even worse, with the top two ICs showing lots of high-frequency noise, and the eye movements distributed across three ICs. I was starting to get suspicious. When I looked at the ICs for the third participant, the blinks were spread across the top four ICs, and two of them again had a ton of weird high-frequency noise.

I then asked myself what I was doing differently from before, and then I realized that I was now referencing to the average of all sites prior to the decomposition. I then changed the scripts to use O2 as the reference, re-ran the ICA decomposition, and then everything worked better.

The moral of the story is that you may get occasional participants for whom the ICs don't look great, but if you see more than one or two, you need to think through your process and figure out what's going wrong. The reference is one possible problem. Another common problem is an insufficient recording duration (especially if you have >64 channels). A third common problem is huge C.R.A.P. that hasn't been eliminated prior to the decomposition.

Step2_ICA_phase1.m implements the procedures described in Chapter 9 for creating datasets that are optimized for the ICA decomposition, including downsampling to 100 Hz, eliminating breaks and other periods of huge C.R.A.P., and implementing an aggressive high-pass filter. It's similar to one of the example scripts at the end of Chapter 9 (**MMN_artifiact_correction_phase3.m**). Open **Step2_ICA_phase1.m** and take a look at it.

One very important element of this script is that it assumes you've already gone through the EEG to determine the parameters that you will use for finding huge C.R.A.P. with the **Artifact rejection (continuous EEG)** routine and stored these parameters in a spreadsheet named **ICA_Continuous_AR.xlsx**. I've already done this for you. To determine these parameters, I first commented out the part of the script that performs the continuous artifact rejection, and then I ran the code at the end of the script for loading the datasets into the EEGLAB GUI. These datasets have been downsampled and aggressively filtered, and I wanted to

see what the artifacts looked like in these datasets because they will be used for the continuous artifact rejection. Two of the subjects had some large C.R.A.P., and I ran the **Artifact rejection (continuous EEG)** routine from the GUI for these subjects to figure out the best rejection criteria. As discussed in the chapter on artifact correction, you really need to look carefully at the data when setting these parameters if you want ICA to work well. The spreadsheet also indicates which channels to include. I've left out the EOG channels and Fp1/Fp2 so that ordinary ocular artifacts don't get deleted. I've also left out any channels that will be left out of the ICA decomposition and interpolated later. Crazy periods in these channels won't influence the decomposition, so we don't need to delete them.

If you look at the code for reading these parameters from the spreadsheet, you'll see that it's much like the code for reading the parameters for interpolating bad channels in the previous exercise, except that we have different columns labels in the spreadsheet.

You'll also see that the script calls the **pop_erplabDeleteTimeSegments** routine to delete the periods of time during the breaks, which also helps get rid of large C.R.A.P. Note that the parameters that control this routine are defined as variables at the top of the script, following good programming practice.

Once you've looked through the script to see how it works, go ahead and run it. You'll see that it creates a new dataset file for each participant with **_optimized** at the end of the filename. You can also load the new datasets into the EEGLAB GUI by running the bit of code at the end of the script. This allows you to see what the optimized datasets look like and make sure everything worked properly.

Step3_ICA_phase2.m runs the ICA decomposition process on the dataset created by **Step2_ICA_phase1.m**. It assumes that an Excel spreadsheet named **interpolate.xlsx** has already been created to indicate which channels will be interpolated after correction has been performed and should therefore be excluded from the ICA decomposition. I've already created this file for you. Note that the decomposition process is quite slow, so this script takes a long time to run.

Step4_ICA_phase3.m takes the ICA weights in the optimized dataset and transfers them back to the pre-optimization dataset. It then removes the artifactual ICs, which are listed in a file named **ICs_to_Remove.xlsx**. I had to determine which ICs were artifactual by looking at the IC scalp maps and by comparing the IC time courses with the EEG/EOG time courses (as described in the chapter on artifact correction). To make this easier, I used the bit of code at the end of **Step3_ICA_phase2.m** to load the datasets into the EEGLAB GUI.

I didn't spend a lot of time making careful decisions about which ICs to remove (and making sure that the ICA decomposition was truly optimal). For example, Subject 1 still has some blink activity remaining in F4 after the correction. It's really boring to spend many hours in a row getting the ICA perfect for a large set of participants! This is one more reason why you should do the initial preprocessing of each participant within 48 hours of data collection. It's a lot easier to spend the time required to optimize the ICA when you're only doing it for one participant at a time and don't have to spend an entire day processing the data from 20 participants.

The last script is **Step5_post_ICA_processing.m,** which performs the steps following artifact correction that must be executed prior to averaging. This includes re-referencing to the average of all sites (and putting the channels into a more useful order), performing interpolation for any bad channels, adding an EventList, assigning events to bins with BINLISTER, epoching the data, and performing artifact detection. The script also prints a summary of the overall proportion of trials marked for rejection in each participant and creates an Excel file with this information broken down by bin.

Open the script and take a look. You'll see that the first part of the script (prior to the loop) defines and opens a set of files with the artifact detection parameters. We've already corrected for blinks, so we only want to flag epochs with blinks that occurred at a time that might interfere with the perception of the stimulus. Eye movements aren't typically an issue in this paradigm because the stimuli are presented briefly in the center of the display, but we flag trials with eye movements that might interfere with the perception of the stimulus (which were quite rare). We use the uncorrected bipolar channels for the blink and eye movement detection. We also flag trials with large C.R.A.P. in any of the EEG channels (using both an absolute voltage threshold and a moving window peak-to-peak amplitude algorithm). Each of these artifact detection routines uses a different flag so that we can keep track of the number of trials flagged for each type of artifact.

The top portion of the script also pre-allocates a set of arrays that will be used to store the number of accepted and rejected trials for each participant. This pre-allocation isn't strictly necessary, but it's good programming practice—it makes it clear what the dimensions of the arrays are.

The body of the subject loop loads the dataset created by the previous script, interpolates any bad channels, re-references the

data, creates the EventList, runs BINLISTER, and epochs the data. These steps are pretty straightforward.

The next set of lines reads the parameters for blink detection from a spreadsheet and then runs the step function algorithm to find blinks that occurred between -200 and +300 ms. These lines use the same "magic code" that I used to grab parameters from spreadsheets in the previous scripts. I didn't spend a lot of time setting the artifact detection parameters—you could do a better job if you spent some time using the strategies described in the chapter on artifact rejection. Once the parameters have been extracted from the spreadsheet, the artifact detection routine is called. We then repeat this process for the eye movement and C.R.A.P. artifacts.

Each of these artifact detection steps adds to the flags set by the previous step. At the end, we save the dataset to the hard drive. It's now ready for averaging!

After the end of the loop, I added some code to grab information about the number of trials that were flagged for each participant. The code demonstrates how to save this information in Matlab's special Table format, which then makes it easy to save the information as a spreadsheet.

As usual, the end of the script has some code after the **return** statement that you can use to load the datasets into the EEGLAB GUI.

Whew! That's a lot of code. But I hope it shows you how to break a complex sequence of processing steps into a set of relatively simple modules. And I hope it also demonstrates the process of going back and forth between the GUI (to set various participant-specific parameters and make sure the data look okay) and scripts (which are much faster, especially when you need to reprocess the data multiple times).

One last thing: Once you've created a set of scripts that perform the different processing stages for your experiment, you can create a "master script" that simply calls the scripts for each stage. Then you can execute all the stages by calling this one script.

This page titled 11.16: Exercise- Building an Entire EEG Processing Pipeline is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.17: Exercise- Averaging with a Custom aSME Time Window

The series of scripts in the previous exercise produce a dataset for each participant that is ready for averaging. In the present exercise, we'll compute the averaged ERPs, including calculating aSME for a custom time window of 110-150 ms (the measurement window for the N170). We'll then create a grand average across participants and plot it. Finally, we'll print a summary of the aSME values to the Command Window.

Go ahead an open the script (**Step6_averaging_and_SME.m**) and take a look at it. As usual, the script begins by defining a set of variables that will be used later in the script. It also opens a file named **ERPset_files.txt**. This file will be used when we make the grand average. When we made grand averages in previous chapters, we loaded all the single-participant ERPsets into ALLERP and then told the grand averaging routine which of these ERPsets to include the grand average. However, it's sometimes more convenient just to work with one ERPSet at a time, save the ERPsets as files on the hard drive, and not keep all the ERPsets loaded in memory. In this case, we can send the grand averaging routine a list of the filenames of the ERPsets, which are themselves stored in a text file. This is what **ERPset_files.txt** is used for in the present script. I could have just loaded all the ERPsets into ALLERP, but I wanted to demonstrate this alternative approach and show you how to open a text file and write to it.

The main loop in the script begins by loading the dataset that was created by the previous script, which is all ready for averaging. The next few lines define the custom aSME time window. This information is stored in a data structure called a **DQ_spec** (*data quality specification*). This structure both defines the time windows prior to obtaining the data quality measures and stores the data quality measures once they've been calculated. It's a little complicated, so ERPLAB provides a routine called **make_DQ_spec** to create it.

We use this routine to create a variable named **custom_aSME_spec**. There are potentially many different types of data quality metrics that can be stored in a **DQ_spec** variable. The first metric stored in a DQ_spec structure is a measure of the baseline noise. The second is the standard error of the mean at each individual time point. The third is the aSME. It's possible for you to define additional types, the most common of which is the bSME (bootstrapped SME). Here, we're going to use aSME, but we're going to specific a custom time window for the aSME calculation so that we get an aSME value that corresponds to our N170 measurement window (110–150 ms). We do this by finding out how many time windows have already been defined by default and then adding a new one.

The next step is to call the averaging routine (**pop_averager**), sending it the **custom_aSME_spec** variable that we just created so that it will compute the aSME for our custom time window (along with the default time windows). This routine returns an ERPset that we store in the **ERP** variable. We then create a name for the ERPset and save the ERPset to a file on the hard drive. We also save the name of this file (including the path) in the **ERPset_files.txt** file so that we have it when we make the grand average later.

The **ERP** variable includes a field named **ERP.dataquality** that stores the data quality metrics that were calculated during the averaging process. The script shows how you can grab the aSME values from **ERP.dataquality** for the channel we will ultimately use to score the N170 amplitude (PO8, Channel 27). For each subject, we get an array of four aSME values for this channel, one for each of the four bins. We store this in a two-dimensional array named **aSME_custom_values,** which has one dimension for subjects and another dimension for bins. The aSME data inside **ERP.dataquality** are stored in a 3D array with dimensions of channels, time ranges, and bins, and we use a Matlab function called **squeeze** to convert this 3D array into the 1D array of values for each bin for the current subject:

```
aSME_custom_values(subject,:) = squeeze(ERP.dataquality(where_aSME).data(measurement_channel,custom_entry_index,:));
```

When you're first learning to write scripts in Matlab, you'll probably find that you frequently get confused about how arrays work and when you need to use a function like **squeeze** to obtain the desired results. I still sometimes get confused, and I often make mistakes when writing code that operates on complicated arrays. But I've learned how in interpret Matlab's error messages, and I often search the Internet for solutions. I also recommend getting a good Matlab book and spending some time learning the logic behind how Matlab operates on arrays. As I mentioned earlier, my lab uses a book called *Matlab for Behavioral Scientists* (Rosenbaum et al., 2014).

After the main loop finishes, the script makes a grand average using the filenames stored in **ERPset_files.txt**. It then plots the grand average so that you can see how to script the plotting routine (**pop_ploterps**).

Finally, the script prints out the custom aSME values that we saved in the variable named **aSME_custom_values** for each participant. It also prints the mean across participants for each bin, along with the RMS (root mean square). The RMS is like the mean, but it does a better job of capturing how the noise level from the individual participants will impact the variability in N170 amplitude scores across participants, the effect size, and the statistical power (see Luck et al., 2021). I recommend taking a look at the aSME values for each participant and then looking at their EEG and ERPs to see if you can understand why some participants have worse (larger) aSME values than others.

The aSME quantifies the data quality for the mean voltage within a given time window, so it's most directly useful when you're scoring amplitude using the mean voltage (which is how we scored N170 amplitude in the ERP CORE paper). If you're using some other scoring method (e.g., peak amplitude, fractional peak latency), you need to use a more complicated method called *bootstrapping* to obtain the SME values. You'll also need to use bootstrapping if you'll be obtaining scores from difference waves or if you apply any other kind of processing to the ERP data after averaging but before scoring (e.g., filtering or channel operations). ERPLAB currently requires scripting to compute bootstrapped SME values, and of the example scripts at the end of Chapter 10 demonstrate how to do this. Chapter 10 also contains a script that demonstrates how to obtain behavioral data.

This page titled 11.17: Exercise- Averaging with a Custom aSME Time Window is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.18: Exercise- Scoring Amplitudes and Latencies and Performing Statistical Analyses

This exercise demonstrates the final stages of processing, in which we obtain amplitude and/or latency scores from each participant and perform statistical analyses. There are many ways we could analyze the N170 data, but here we will look at three simple approaches.

In the first approach, we'll measure the mean amplitude in the N170 latency range (110-150 ms) for the faces and cars in the left-hemisphere and right-hemisphere electrode clusters. We'll then perform a 2 ´ 2 ANOVA with factors of stimulus type (face vs. car) and electrode hemisphere (left vs. right). The N170 is typically larger for faces than for cars, and this effect is usually larger in the right hemisphere than in the left hemisphere. As a result, we would expect a main effect of stimulus type and a stimulus type ´ electrode hemisphere interaction. You'll need to perform this statistical analysis using your own statistics package.

Any differences between the faces and the cars in the first analysis could reflect differences in low-level features (e.g., luminance, spatial frequency) rather than differences between the face and car categories per se. To address this possibility, the experiment also presented phase-scrambled face and car images, which contain the same low-level features as the faces and cars but are unrecognizable as faces or cars. In our second analysis, we'll make a face-minus-scrambled-face difference wave and a car-minus-scrambled-car difference wave. The ERP activity directly attributable to the low-level features should be subtracted away in these difference waves, and any differences between the two difference waves can be attributed to higher-level features.

The third analysis will be just like the second analysis, except that it will be performed on the peak latency of the N170 rather than the mean amplitude. Because peaks are easily distorted by high-frequency noise, we'll apply a stronger low-pass filter to the data prior to measuring the peak latency.

Important: For the second and third analyses, the scripts will perform a paired *t* test comparing the two difference waves (only for the right-hemisphere electrode cluster). This uses a Matlab function called **ttest**, which is a part of the Statistics and Machine Learning Toolbox. You can see if you have that toolbox by typing **ver** on the Matlab command line. If you don't have that toolbox, you can just delete (or comment out) those lines of the script.

Go ahead an open the script (**Step7_scoring.m**) and take a look at it. As usual, the script begins by defining a set of variables that will be used later in the script. It also opens a couple files that we'll use to store the names of the ERPset files. We'll send these files to the Measurement Tool so that it knows which ERPsets to use for scoring the ERPs (just like we did with the grand averaging routine in the previous exercise).

The main loop loads the ERPsets created by the script in the previous exercise. It then uses ERP Channel Operations (the **pop_erpchanoperator** routine) to create a left-hemisphere cluster channel and a right-hemisphere cluster channel. This is pretty simple, so we just send the equations directly in the script rather than saving them in a file.

The next step is to create the difference waves using ERP Bin Operations (the **pop_binoperator** routine). It sends a file named **BinOps_Diff.txt** that contains the equations for making the difference waves. The channel and bin operations create updated versions of the **ERP** variable, and we save this ERPset to the hard drive. We also save the name of the ERPset in a file named **Measurement_files.txt**.

Then we apply a low-pass filter with a half-amplitude cutoff at 15 Hz and a slope of 48 dB/octave, which help us measure the peak latency more precisely. The resulting ERPset is saved to the hard drive, and the name of the ERPset is saved in a file named **Measurement_15Hz_files.txt**.

After we loop through all the subjects, we close the two files that are used to store the ERPset filenames. Then we start the first analysis stage, in which we obtain the mean amplitude scores from the parent waveforms. This is achieved by calling the **pop_geterpvalues** routine, which is the script equivalent of the Measurement Tool. We send it the name of the file that holds the names of all the unfiltered ERPsets that we created in the loop so that it knows which ERPsets should be measured. We also send it the start and end times of the measurement window (110 and 150 ms, which are the values recommended in the ERP CORE paper). We also send two arrays, one containing a list of the bins that we want to measure (Bins 1-4, which contain the parent waveforms) and one containing a list of the channels that we want to measure (35 and 36, the left- and right-hemisphere cluster channels). There are also some parameters that you should recognize from using the GUI version of the Measurement

Tool. Finally, we tell it the name of the text file that it should use for saving the amplitude scores (**MeanAmp.txt**).

Go ahead and run the script, and you'll see that it creates the **MeanAmp.txt** file, with one line for each subject and one column for each of our 8 measurements (4 bins ´ 2 channels). Load these data into a statistical package. We're going to ignore the bins for the scrambled stimuli and perform a 2 ´ 2 ANOVA with factors of stimulus type (face vs. car) and electrode hemisphere (left vs. right). If you run the analysis, you should get something like the ANOVA table shown in Table 11.3.

Table 11.3. ANOVA table for the first N170 analysis (from JASP).

Cases	Sum of Squares	df	Mean Square	F	p
Hemisphere	3.228	1	3.228	1.552	0.248
Residuals	16.639	8	2.080		
StimType	43.727	1	43.727	62.393	< .001
Residuals	5.607	8	0.701		
Hemisphere ´ StimType	2.180	1	2.180	3.485	0.099
Residuals	5.004	8	0.625		

As predicted, the greater N170 amplitude for the faces than for the cars led to a significant main effect of stimulus type. The effect was somewhat greater in the right-hemisphere cluster than in the left-hemisphere cluster, but the stimulus type ´ electrode hemisphere interaction did not reach significance. We have only 9 participants, so this probably just reflects low power.

The next part of the script obtains the mean amplitude scores from the difference waves. This time, however, it doesn't save the scores in a file. Instead, we use the **'SendtoWorkspace', 'on'** option to save the scores in a variable named **ERP_MEASURES** in the Matlab workspace. This variable is a 3-dimensional array with dimensions of bin, channel, and subject. For example, ERP_MEASURES(2, 1, 7) is the score for the second bin, the first channel, and the seventh subject. The script grabs these values and stores them in two one-dimensional arrays, one for the faces-minus-scrambled-faces bin and one for the cars-minus-scrambled-cars bin. The one dimension is subject, so each of these arrays has 9 values.

We then send these two arrays to the **ttest** function like this:

```
[h,p,ci,stats] = ttest(faces_minus_scrambled_meanamp, cars_minus_scrambled_meanamp);
```

The function returns four variables: **h** is a 1 if the effect was significant and 0 otherwise (assuming an alpha of .05); **p** is the *p* value from the test; **ci** is the 95% confidence interval for the difference between the two means; and **stats** holds the actual *t* value along with the degrees of freedom. This routine can be used to perform a paired *t* test or a one-sample *t* test. The **ttest2** function can be used for an independent-samples *t* test, which you would use to compare two groups of subjects. After the script calls the **ttest** function, it uses **fprintf** to print the results in the Command Window. As you can see by looking at your Command Window, there was a significant difference in amplitude between the faces-minus-scrambled-faces and cars-minus-scrambled-cars difference waves.

The last part of the script measures the peak latency of the N170 instead of the mean amplitude. It uses a wider measurement window (which is often needed for latency measures), and it performs the measurements from the more aggressively filtered

ERPsets. The script then calls the **ttest** function and prints the results in the Command Window. The peak latency was significantly earlier for the faces-minus-scrambled-faces waveform than for the cars-minus-scrambled-cars waveform (consistent with Figures 11.1D and 11.1E).

This is the last script for this chapter. You've now gone through every major step of EEG and ERP processing, all the way from reading in the raw EEG to conducting a statistical analysis. Congratulations!

But remember, you should go back and forth between scripts and the GUI rather than relying solely on scripts. For example, you should use the Viewer option in the Measurement Tool to look at the scores alongside each averaged ERP waveform to verify that the measurement process is working properly. However, the script is also useful, because it makes it easier to repeat the processing if you need to make a change somewhere earlier in the pipeline. Also, if you make your scripts and data available when you publish a paper, other researchers can see exactly how you implemented each step of processing rather than relying on the relatively brief and vague description of the processing that is typically provided in a Method section.

Scripting is a skill that takes a long time to master, and you may initially wonder if it's worthwhile. It may seem like it's faster to do everything in the GUI than to spend hours debugging scripts that do the same thing. But in the long run, scripting is incredibly useful, and you will get faster with experience. You'll still make lots of mistakes—I certainly do!—but you'll be able to find and fix them much more rapidly once you have more experience.

This page titled 11.18: Exercise- Scoring Amplitudes and Latencies and Performing Statistical Analyses is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

11.19: Key Takeaways and References

Key Takeaways

- The process of creating a script typically involves running the processing steps in the EEGLAB/ERPLAB GUI, copying the commands from the history into a script, and adding a loop.
- You will save yourself a lot of time and pain in the long run by using intrinsically meaningful variable names, including lots of comments, and defining all key values as variables at the top of the script.
- It's almost always better to write a sequence of relatively simple scripts rather than one large script that does everything. This makes errors less likely, makes errors easier to find, and gives you more flexibility.
- You should make extensive use of the GUI the first time your process the data from a given participant. This allows you to detect problems in the data, set participant-specific parameters (e.g., for artifact rejection and correction), and make sure that everything is working properly. Once you've gone through the data in this manner, you should reprocess the data with your scripts (which helps avoids errors that can occur in manual processing).
- You can learn a lot by getting example scripts from other people, but don't apply those scripts to your own data unless you fully understand every line of code.

References

Emerson, R. W. (1841). *Self-Reliance*. Lulu Press.

Luck, S. J., Stewart, A. X., Simmons, A. M., & Rhemtulla, M. (2021). Standardized measurement error: A universal metric of data quality for averaged event-related potentials. *Psychophysiology*, *58*, e13793. https://doi.org/10.1111/psyp.13793

Rosenbaum, D. A., Vaughan, J., & Wyble, B. (2014). *MATLAB for Behavioral Scientists* (2nd Edition). Routledge.

Rossion, B., & Caharel, S. (2011). ERP evidence for the speed of face categorization in the human brain: Disentangling the contribution of low-level visual cues from face perception. *Vision Research*, *51*(12), 1297–1311. https://doi.org/10.1016/j.visres.2011.04.003

This page titled 11.19: Key Takeaways and References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

12: Appendix 1: A Very Brief Introduction to EEG and ERPs

This appendix provides a very brief introduction to the EEG and ERPs (and necessarily oversimplifies some complex issues). It is meant to provide enough background for you to understand the big picture of the analysis procedures described in this book, but it is just a start.

The EEG

The neurons in your brain produce two main kinds of electrical potentials (voltages), called *postsynaptic potentials* and *action potentials*. We can't ordinarily pick up action potentials from the scalp, but the postsynaptic potentials produced by individual neurons can propagate through the brain, skull, and scalp. If we put electrodes on the scalp, as shown in Figure A1.1, we can pick up the propagated postsynaptic potentials. These propagated voltages are what we call the *electroencephalogram* or *EEG*. This is a simplification; if you want to know the full story, I recommend Jackson and Bolger (2014) or Buzsáki et al. (2012).

The postsynaptic potentials produced by individual neurons sum together and spread widely before reaching the scalp, so any given electrode is picking up voltages generated by thousands or millions of neurons in a broad set of brain regions. As a result, a voltage picked up at a given electrode site may not represent activity coming from the cortex directly under the electrode.

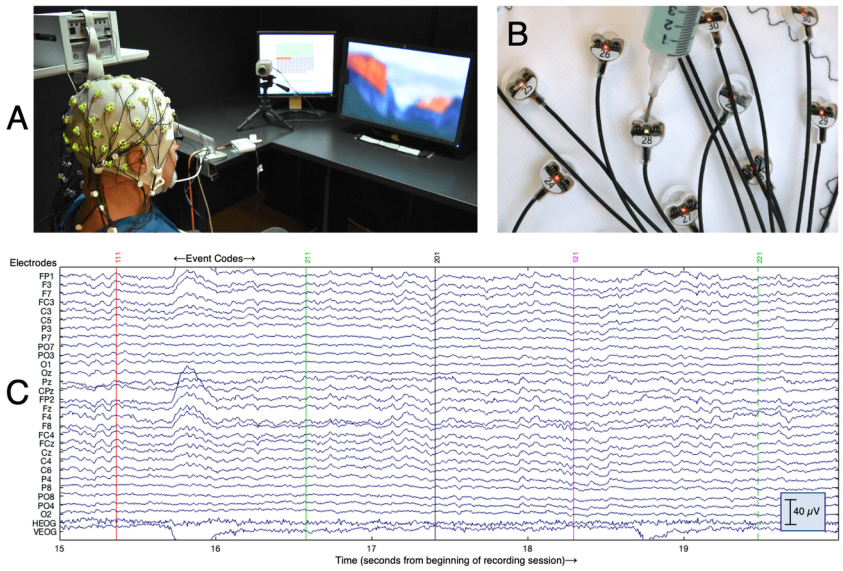


Figure A1.1. Typical EEG recording. (A) The subject is in the lab, wearing an electrode cap and facing the computer monitor where the stimuli will be presented. (B) Each electrode is a metal pellet encodes in plastic. A conductive gel is squirted into the electrode holder to make an electrical contact between the electrode pellet and the skin. (C) A 5-second period of EEG. Each waveform is the signal from one of the 30 electrodes that was used in this experiment. The X axis is time (in seconds), and the Y axis is voltage (in microvolts, μV). The colored vertical lines indicate event codes, which are numeric codes that mark the time and identity of the stimuli that were presented to the subject. The EEG signals are referenced to the right mastoid. HEOG (horizontal electrooculogram) is the potential between two electrodes placed just lateral to the eyes, and VEOG (vertical electrooculogram) is the potential between electrodes placed under and over the right eye.

The EEG may be recorded from just a few electrodes or from as many as 256 electrodes, but most studies include between 16 and 64 electrodes. As shown in Figure A1.2, there is a widely used electrode naming system (the International 10/20 System) in which each electrode is given a 1-2 letters to indicate the general region of the head and a number to indicate the left-right position (or a "z" for the midline). For example, electrode P1 is over parietal cortex, just slightly to the right of the midline, and electrode FCz is on the midline, between the frontal and central regions.

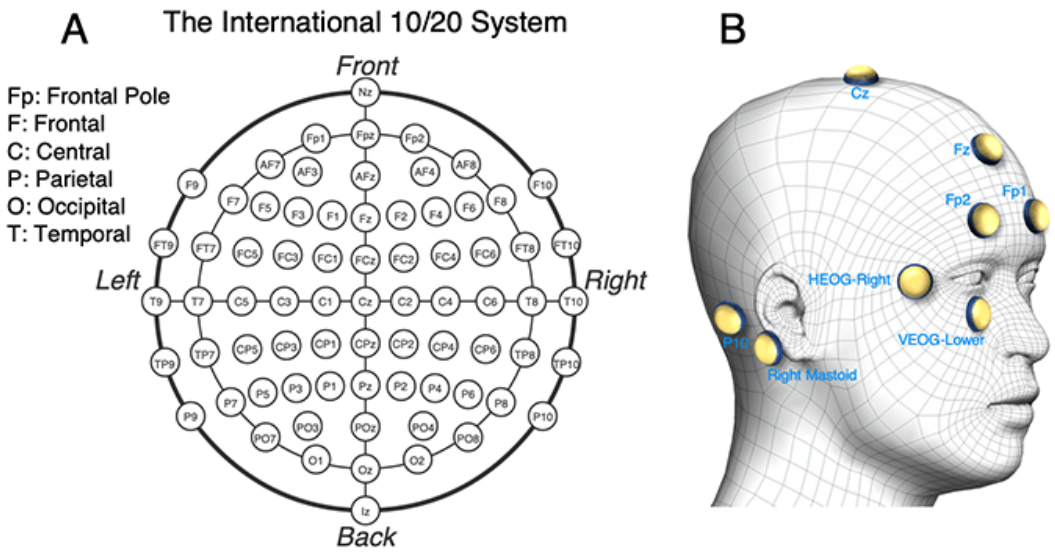


Figure A1.2. (A) The International 10-20 system. In this system, each electrode is named with 1-2 letters indicating the general region of the scalp and a number indicating the left-right location. The numbers are odd for the left hemisphere, even for the right hemisphere, and get larger as we move away from the midline (which is labeled with a "z" to indicate "zero"). For example, Fp2 is over the frontal pole, just to the right of the midline, and Cz is over the central sulcus on the midline. (B) Additional common electrode sites. Electrodes are often placed over the left and right the mastoid processes (the bony protrusions just behind the earlobes) as reference electrodes. The horizontal electrooculogram (HEOG) is often recorded from electrodes just lateral to the left and right eyes to detect horizontal eye movements. The vertical electrooculogram (VEOG) is often recorded from an electrode just under the left or right eye (referenced to the corresponding Fp1 or Fp2 electrode over the eye) to detect blinks and vertical eye movements.

The Reference Electrode

A voltage is the *potential* for electrical charges to move from one place to another (e.g., from one terminal of a car battery to the other). As a result, the EEG is always a voltage between two electrodes, which we call the *active electrode* and the *reference electrode*. Typically, most or all of the active electrodes share the same reference electrode. Consider, for example, the EEG waveforms in Figure A1.1.C. Each waveform has a label, which indicates the active electrode for that waveform, and all of these channels except the bottom two used the same reference electrode, which was located on the *mastoid process* (the bony protrusion behind the ear, shown in Figure A1.2.B). This means that the waveform labeled F7 shows the electrical potential between the F7 electrode site and the mastoid. In theory, we would like a reference that is electrically neutral and does not impact the observed waveform. In practice, however, there is no neutral site, and the waveform for a given channel is equally impacted by activity arising from the active and reference electrodes.

The horizontal and vertical electrooculogram signals at the bottom of Figure A1.1.C (labeled HEOG and VEOG) use different reference electrodes. The HEOG signal uses HEOG-right as the active electrode (see Figure A1.2.B) and the mirror-image HEOG-left electrode as the reference. The VEOG signal uses the VEOG-lower electrode as the active and the FP2 electrode as the reference. These *bipolar* configurations make it easier to detect eye movements and blinks, which are large artifacts that propagate across the scalp.

Choosing the best reference electrode can be challenging. Fortunately, you can re-reference your data offline to any electrode (or combination of electrodes) that was in the original recording.

Artifact Rejection and Correction

Some portions of the EEG are contaminated by very large artifacts arising from sources such as eye blinks, eye movements, and muscle activity. In some studies, these epochs are excluded from all analyses, which is called *artifact rejection*. In other studies, a mathematical procedure is used to estimate the contribution of the artifacts and remove them from the epochs, which is called *artifact correction*. Many studies use correction for some artifacts and rejection for others.

Signal Averaging and Event-Related Potentials

The EEG is an incredibly complex signal that includes activity generated throughout the cortex from hundreds or even thousands of processes, all of which are summed together in our scalp electrodes. As a result, you can't get a lot of information about specific neurocognitive processes from looking at the raw EEG. The raw EEG signal can tell you if someone is asleep, awake but zoned out, or highly alert and attentive, and you can sometimes see evidence of pathology (e.g., epileptic spikes). Signal processing methods must be applied to the EEG data to isolate specific processes. Perhaps the simplest of these methods is *signal averaging*, which we use to extract ERPs from the EEG.

Signal averaging is illustrated in Figure A1.3. The goal of signal averaging is to isolate the electrical *potentials* (voltages) that are *related* to specific *events* (i.e., the *event-related potentials*). Signal averaging accomplishes this by assuming that an event such as a stimulus produces a similar waveform on each trial, and random noise is added to this waveform to produce the EEG that we record. If we average together the recorded EEG following many events, then the noise will "average out" and the signal will remain in the average. Figure A1.3.A illustrates the EEG from a single electrode. Whenever an event such as a stimulus or response occurs, an *event code* is inserted into the EEG data file to mark the time of the event. In this example, the events are stimuli (e.g., shapes flashed on a computer monitor).

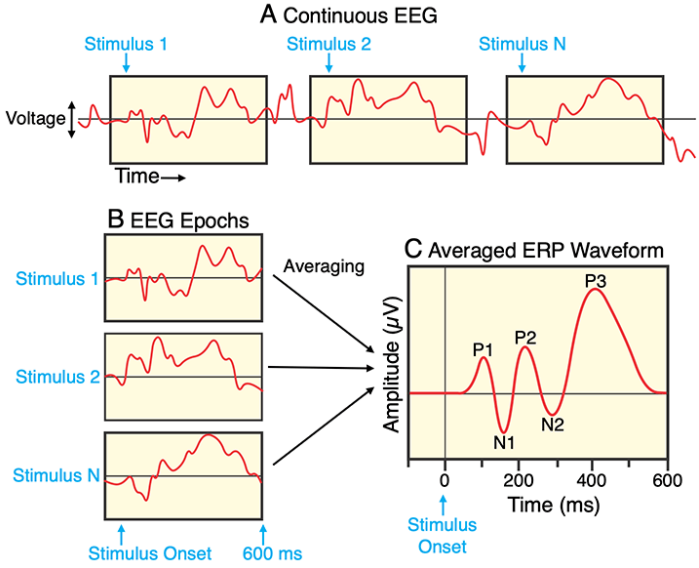


Figure A1.3. The signal averaging process. The EEG is recorded as a continuous signal (A) while stimuli are presented. An event code marks the onset of each stimulus. An epoch of EEG data is extracted for each stimulus (B), beginning slightly before stimulus onset (e.g., 100 ms) and extending for a fixed period after stimulus onset (e.g., 600 ms). The single-trial epochs are lined up and then averaged together to create an averaged ERP waveform (C). That is, the averaged value at time *t* is the average voltage at time *t* across all *N* epochs. The X axis is time in thousandths of a second (milliseconds, ms), and the Y axis is voltage in millionths of a volt (microvolts, μV). In a typical experiment, there would be several different types of stimuli, and a separate averaged ERP waveform would be created for each stimulus type. We would also have a separate waveform at each electrode site.

For each event, we extract an *epoch* of EEG data surrounding that event. In this example, the epochs begin 100 ms before stimulus onset (giving us a 100-ms *prestimulus baseline* period) and extend until 600 ms after stimulus onset. As shown in Figure A1.3.B, all of the epochs for a given type of stimulus are then lined up with respect to time zero. The voltage at a given time for a given stimulus is assumed to be the consistent ERP at that time plus random noise. If we simply average all the epochs together, the noise will largely cancel out, and the averaged ERP waveform will mainly consist of the consistent ERP response (plus some residual noise because we always have a finite number of trials). The number of trials required to produce an acceptable average varies widely across experiments and depends on factors such as how large the ERP is and how much noise is in the EEG. Note that artifact rejection and/or correction are applied prior to averaging.

In ERP research, we use the term *noise* to refer to any uncontrolled variability in the EEG or in the averaged ERP waveform. We are assuming that the EEG consists of a consistent ERP signal on each trial plus random variability. Anything that causes the EEG to deviate from the consistent ERP signal is considered noise. If we repeated an experiment multiple times for a given participant (assuming no fatigue, no learning, etc.), any differences in the averaged ERP waveforms across repetitions would be a result of this noise. Some of this noise comes from electrical devices in the recording environment that is unintentionally picked up by the recording electrodes. Some of it comes from nonneural biological sources, such as blinks and muscle contractions. Some of it

comes from brain activity that is unrelated to the stimuli. All else being equal, an averaged ERP waveform will have less residual noise if more trials are averaged together.

What Do We Mean by "Noise"?

In ERP research, we use the term *noise* to refer to any uncontrolled variability in the EEG or in the averaged ERP waveform. We are assuming that the EEG consists of a consistent ERP signal on each trial plus random variability. Anything that causes the EEG to deviate from the consistent ERP signal is considered noise. If we repeated an experiment multiple times for a given participant (assuming no fatigue, no learning, etc.), any differences in the averaged ERP waveforms across repetitions would be a result of this noise. Some of this noise comes from electrical devices in the recording environment that is unintentionally picked up by the recording electrodes. Some of it comes from nonneural biological sources, such as blinks and muscle contractions. Some of it comes from brain activity that is unrelated to the stimuli. All else being equal, an averaged ERP waveform will have less residual noise if more trials are averaged together.

ERP Peaks and Components

Figure A1.3.C shows a hypothetical averaged ERP waveform for a visual stimulus recorded at an occipital electrode site (with no noise). The X axis is time, with stimulus onset at time zero. The Y axis is amplitude. The prestimulus period is flat because there should be no consistent stimulus-related activity prior to the stimulus. The waveform begins to deviate from zero at approximately 50-60 ms, which is when visual information typically reaches visual cortex. There is no delay between the postsynaptic potentials generated in the brain and the voltage deflections that we see in an ERP waveform. As a result, the sequence of voltages over time in the ERP waveform corresponds to the sequence of sensory, cognitive, affective, and motor processes that follow the onset of a stimulus.

The ERP waveform consists of a set of positive-going and negative-going *waves* or *peaks*, which are related in a complicated way to a set of underlying *components* in the brain that reflect specific neurocognitive processes. We can't directly see these components, but we try to draw inferences about them from the observed scalp ERP waveforms (see Chapter 2 in Luck, 2014, for a detailed discussion).

Most ERP components are named with a *P* or an *N* to indicate whether they are positive-going or negative-going, followed by a number to indicate the timing. For example, *N1* is the first major negative wave and *P3* is the third major positive wave. If the number is large (>10), it instead indicates the approximately latency of the peak in milliseconds (e.g., *N170* for a negative-going wave that peaks at 170 ms). Sometimes, a component is given a name that reflects the conditions under which it is observed (e.g., the *error-related negativity* for a negative voltage that is present when the participant makes an incorrect response). Note that the relationship between an underlying component and the observed peaks and waves is complex, and the component names can sometimes be misleading. For example, the P1 elicited by a visual stimulus is completely unrelated to the P1 elicited by an auditory stimulus, but the P3 is the same for auditory and visual stimuli. A detailed discussion is provided in Chapters 2 and 3 of Luck (2014) and by Kappenman and Luck (2012).

To statistically analyze ERP data, researchers typically start by obtaining an amplitude or latency *score* from each individual participant's averaged ERP waveform in each group or condition. Then these scores are entered into a statistical analysis that is much like the analysis of a behavioral variable such as response time. For example, you might find the *peak latency* (the time point when the voltage reaches its maximum value) for the P3 wave in each participant's ERP waveform and then determine whether the latencies are significantly later in a patient group than in a control group using a *t* test.

References

Buzsáki, G., Anastassiou, C. A., & Koch, C. (2012). The origin of extracellular fields and currents—EEG, ECoG, LFP and spikes. *Nature Reviews Neuroscience*, *13*, 407–420. https://doi.org/10.1038/nrn3241

Jackson, A. F., & Bolger, D. J. (2014). The neurophysiological bases of EEG and EEG measurement: A review for the rest of us. *Psychophysiology*, *51*(11), 1061–1071. https://doi.org/10.1111/psyp.12283

Kappenman, E. S., & Luck, S. J. (2012). ERP components: The ups and downs of brainwave recordings. In S. J. Luck & E. S. Kappenman (Eds.), *The Oxford Handbook of ERP Components* (pp. 3–30). Oxford University Press.

This page titled 12: Appendix 1: A Very Brief Introduction to EEG and ERPs is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

13: APPENDIX 2: TROUBLESHOOTING GUIDE

Overview

Computers can be maddening. No matter how much we test ERPLAB, we still get occasional bug reports. There can also be bugs in EEGLAB, in Matlab, or in your operating system. Even if there isn't a bug, EEGLAB, ERPLAB, you might encounter a problem because you specify an incorrect setting or leave out a necessary step. And if you're writing your own scripts, you're likely to introduce your own bugs. I can't tell you how much time I spent troubleshooting problems when I was writing this book!

As I mentioned in Chapter 1, you should look at these problems as an opportunity for working on your general problem-solving skills. Indeed, I find that troubleshooting computer problems is like a microcosm of science. You generate hypotheses, you collect data to test those hypotheses, and you make sure that you can replicate both the problem and the solution.

In this Troubleshooting Guide, I've provided a variety of hints and strategies for solving the most common types of problems you're like to encounter. This includes both specific strategies for issues that often arise in Matlab, EEGLAB, and ERPLAB, along with general strategies for solving computer problems and debugging computer programs. I've also provided some exercises to help you learn to interpret Matlab's error messages and to monitor the operation of a script while it's running.

13.1: A2.1 The First Step

13.2: A2.2 Some Basic Solutions

13.3: A2.3 Taking a Scientific Approach

13.4: A2.4 Deciphering Matlab's Error Messages

13.5: A2.5 Debugging Scripts by Performing Experiments and Collecting Data

13.6: A2.6 Avoiding Bugs in Your Scripts with Good Programming Practices

13.7: A2.7 References

This page titled 13: Appendix 2: Troubleshooting Guide is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

13.1: A2.1 The First Step

Here's something that should be obvious but often isn't: **When you run into a problem you can't easily solve, you should read the documentation and online help**. Over half of the "bug reports" we get for ERPLAB aren't bugs at all but are simple misunderstandings. These misunderstandings typically arise when someone starts using ERPLAB without spending the time to go through our tutorial, when someone tries to use a complicated feature without first reading the documentation for that feature, or when someone encounters a common issue but hasn't consulted our Frequently Asked Questions page. In tech support, these are called **RTFM** problems. Google **RTFM** if this is an unfamiliar term and you'd like a quick smile.

Here are the main sources of EEGLAB and ERPLAB information:

- EEGLAB: Online Documentation, Frequently Asked Questions
- ERPLAB: Online documentation, Frequently Asked Questions, Tutorial

You can also often find solutions by typing something like **ERPLAB binlister** or **EEGLAB ICA** into a search engine. You can also post questions to email list for EEGLAB or the email list for ERPLAB. But please don't send me a personal email—I don't have the bandwidth to provide individualized tech support for everyone who uses ERPLAB.

This page titled 13.1: A2.1 The First Step is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

13.2: A2.2 Some Basic Solutions

Resetting and Restarting

If restarting EEGLAB doesn't work, I try restarting Matlab. Again, this is especially useful when I'm encountering a weird problem. If that doesn't work, I try resetting ERPLAB's *working memory*. This is a file that ERPLAB uses to store your current settings. The problem you're having might be a side effect of one of your settings, or the settings may have become corrupted, so clearing the settings sometimes helps. To do this, select **EEGLAB > ERPLAB > Settings > ERPLAB Memory Settings > Reset ERPLAB's working memory**.

And if restarting Matlab and resetting ERPLAB's working memory doesn't work, and the problem is really odd, I try restarting my computer. But that's very rare. And just like I restart EEGLAB on a regular basis to prevent problems, I recommend restarting your computer once a week or so. If the weird problem persists, you might try moving to a different computer to see if that solves the problem.

Finally, you can try uninstalling and reinstalling EEGLAB and ERPLAB. Although rare, it is possible that the code has been corrupted. I recently encountered a case of a user who reported that one of the ERPLAB dialog boxes was missing some buttons, and she found that reinstalling ERPLAB solved the problem.

Missing Channel Locations or Data Quality Metrics in Datasets or ERPsets

Are your channel locations or data quality metrics missing from a dataset or ERPset? The most common reason for this is that you have applied **EEG Channel Operations**, **ERP Channel Operations**, or **ERP Bin Operations** to your data.

When you apply EEG or ERP Channel Operations, ERPLAB may not be able to figure out the location information for your new channel. For example, when you create a new channel, how can ERPLAB know the 3-D location of this new channel? Indeed, if you create a channel that is the average of several electrode sites, it doesn't really have a location. There is a box you can check in the GUI labeled **Try to preserve location information**, and it can preserve the channel locations in many cases (e.g., when you re-reference your data). But it uses a fairly simple algorithm, and it may strip out your previous channel location information if the algorithm can't figure out a sensible set of locations.

A similar issue arises for the data quality metrics that are computed during averaging. If you modify the channels using **ERP Channel Operations** or modify the bins using **ERP Bin Operations** to your data, ERPLAB has no way of knowing whether the data quality metrics are still valid. For example, if you create a new bin with a difference wave or an average of two prior bins, what is the data quality for this new bin? There may be analytic solutions for some cases, but these solutions would require many assumptions. As a result, ERPLAB strips out the data quality information from the ERPset that is created when you run **ERP Channel Operations** or **ERP Bin Operations**. Our logic in stripping out the channel location information or data quality metrics in these situations is that we'd rather eliminate the information than provide you with incorrect or misleading information.

Solutions to Miscellaneous Common Problems

- Make sure the MATLAB PATH is set correctly (especially if you're having problems right after installing a new version). See the section describing the Matlab PATH near the end of Chapter 1 for more details.
- If you don't see ERPLAB in the EEGLAB GUI, make sure that ERPLAB is installed inside the **plugins** folder within the **EEGLAB** folder (e.g., eeglab2020_0 > plugins > ERPLAB8.30). You should see a file named **eegplugin_erplab.m** inside that folder (and not inside another folder).
- Make sure you're using the right version of Matlab, EEGLAB, ERPLAB.
- Some ERPLAB functions (e.g., filtering) require the Matlab Signal Processing Toolbox. You may need to install this toolbox. If you purchased Matlab yourself, you may need to buy this toolbox from The Mathworks. If you obtained Matlab through your institution, ask your IT group how to get this toolbox.
- If EEGLAB/ERPLAB/Matlab doesn't seem to respond to the mouse or keyboard, you may have a window hidden somewhere that's waiting for input. For example, if you open the GUI for filtering, and then you look at another window, the filtering GUI

may be waiting for input, and nothing else will work right until you close the filtering GUI. In general, “figures” can be left open without interfering with other operations, but dialog boxes cannot.

- If you load an EEG dataset and try to view the EEG data with EEGLAB > Plot > Channel data (scroll), but some or all of the channels appear to be missing, the DC offset of the EEG signal is probably shifting the EEG outside of the range that is visible in the plotting window. To solve this, select **Display > Remove DC offset** in the plotting window.
- If you have trouble plotting ERP waveforms, click the RESET button in the plotting GUI to get rid of custom settings.
- EEGLAB also has a routine for extracting epochs from the continuous EEG (**EEGLAB > Tools > Extract epochs**), but do not use it!!! Instead, use **ERPLAB > Extract bin-based epochs**.
- Don’t forget to read the Frequently Asked Questions (FAQ) pages for both EEGLAB and ERPLAB.
- Also, EEGLAB and ERPLAB often print information in the command window that can help you figure out why things aren’t working properly.

This page titled 13.2: A2.2 Some Basic Solutions is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

13.3: A2.3 Taking a Scientific Approach

You're a scientist, right? This means that you've been trained to develop and test hypotheses. That's exactly what you do when you're troubleshooting and debugging. That is, you need to come up with a hypothesis about the nature of the problem and then figure out how to test that hypothesis. This section will flesh out the scientific approach to finding and fixing problems with ERPLAB and EEGLAB.

The Importance of Replication

If you really want to understand a problem, you first need to make sure that you can replicate it. Does it persist after you restart Matlab? Do you get the same problem on a different computer?

Also, once you think you've found the solution, you should make sure that you can make the problem reappear and then disappear again. For example, imagine that a part of ERPLAB keeps crashing when you try to run a particular processing step, and you try 12 different things to make it stop crashing. And you finally get it to work after installing a new version of EEGLAB. Was the EEGLAB version really the problem? After all, you probably updated the PATH when you installed the new version of EEGLAB, so perhaps the PATH was the problem. If you want to be sure that it was a version problem, you should try going back to the previous version and seeing if ERPLAB crashes again. And then you should verify that updating the version again eliminates the crash. In experimental psychology, this is often called the ABAB experimental design.

In many cases, you don't actually care whether you fully understand the problem, and this ABAB approach is unnecessary. Once the program is no longer crashing, you may not care what was causing the crash. But in other cases it's really important to understand the problem. For example, imagine that you're writing a script, and you find that you can get your script to run by sending some parameter that you don't really understand when calling an ERPLAB function. Do you really trust that you've now solved the problem? In this kind of situation, the ABAB approach is usually worth the time and effort.

Start with a Literature Review

When you're designing an ERP experiment, you already know that it's important to read the relevant literature so that you know what has already been done in your research area and so that you can learn about useful methods for answering your scientific question. Without the necessary background knowledge, you probably won't have a good hypothesis and probably won't design a good experiment.

When troubleshooting or debugging, the analog is to read the documentation, do the tutorials, and scan the frequently asked questions page. If you don't take these steps first, you're likely to waste a lot of time pursuing bogus hypotheses about the source of the problem and trying solutions that are unlikely to work.

Carefully Observe the World and then Develop a Hypothesis

You were probably taught the "scientific method" when you were in elementary school. In the usual version, a scientist observes the world, develops a hypothesis, and then conducts an experiment to test the hypothesis. This is a gross mischaracterization of how science actually works (see, e.g., Feyerabend, 1993). However, this oversimplification is useful for troubleshooting and debugging, especially insofar as it proposes that you should carefully observe the world in the process of developing your hypothesis about the source of the problem.

In the context of software troubleshooting, this means that you should look carefully at the inputs and outputs. The main inputs are the data that are being processed and the parameters that are specified (either the settings in the GUI or the values that are being sent to a function in a script). The main outputs are the messages printed in Matlab's command window (including but not limited to error messages) and the output data that are created. I particularly recommend taking a careful look at the parameters and the error messages (because the data are usually so complicated that you need a hypothesis before knowing what to look for).

Matlab error messages often appear to be written in a language that you don't speak (*Programmerese*). But if you spend enough time looking at the messages, you may find that you know enough of the words to glean some valuable information. You may be

able to get some insight by typing the key part of the message into a search engine, enclosed in double quotes and preceded by **matlab**. For example, if you get the error message **character Vector Is Not Terminated Properly**, you can type this into Google: **matlab "character Vector Is Not Terminated Properly"**. A later section of this appendix provides more detailed information about how to decipher Matlab's error messages.

The error message will tell you which lines of code generated the error message, both in your script and inside a function you were calling when the problem occurred. This is a major clue. You can then open the code for the function and see what was happening in that function. But keep this in mind: The actual problem in the code may have been several lines before the problem was detected, so you should look at that whole section of code.

Once you've done your careful observation, you should be able to develop a hypothesis.

Run Experiments and Collect New Data to Refine and Test Your Hypothesis

Now it's time to test your hypothesis. Sometimes this is trivial: You hypothesize that you need to update your version of EEGLAB, and you then see if updating EEGLAB solves the problem. But in other cases, your hypothesis does not specify the solution, but is instead a more general hypothesis, such as which part of the code is the problem or which variable might not be set properly. In these cases, you need to figure out how to test your hypothesis.

The method for testing your hypothesis will, of course, depend on the nature of your hypothesis. However, there are two general strategies, which are analogous to recording versus manipulating brain activity. That is, you can examine the representations that your code produces (e.g., values of variables, output files) at various steps in processing, or you can attempt to modify the operation of the code (e.g., by changing the inputs or by turning off various parts of the code). I'll say more about these options in a later section.

Science is Social!

One of the main shortcomings of the elementary school version of the scientific method is that it does not treat science as a deeply human, social activity. Much of science is driven by vanity, competition, and cooperation. I wouldn't recommend vanity or competition when you're troubleshooting or debugging, but cooperation is extremely valuable. More specifically, if you get stuck, get advice from other people. This could be a more experienced EEGLAB/ERPLAB/Matlab person in your lab or your department. Or it could be some stranger on the Internet. (The part of the Internet inhabited by programmers and scientists is much kinder and more helpful than most other parts.)

Often, your question has already been answered. EEGLAB has an extremely active listserv, which you can join. You can then search the archive to see if your question has already come up. ERPLAB also has a listserv with a searchable archive. If you don't see an answer to your question, post it to the listserv.

If you're writing a script and having a problem with a built-in Matlab function, or you're having a problem figuring out the right syntax, you can probably find an answer on the Internet. The two best sources of answers are the resources provided by Matlab and the StackOverflow forum. I usually just type something like this into Google: **Matlab "sorting a list"**. I probably did that 300 times while writing this book!

Rubber Duck Debugging

If you can't find the solution to your problem online, it might be time to ask a colleague. However, you might not have an appropriate colleague, or you might not want to bother anyone. In these cases, you can try something called *rubber duck debugging*. This concept comes from the observation that when you go to ask someone for help, the process of explaining the problem to that person often leads you to realize the answer. So, you really just need to try explaining the problem. Instead of taking up someone else's time with this, you can just explain the problem to a rubber duck (Thomas & Hunt, 2019). Of course, it doesn't need to be a rubber duck. It can be a plant, your dog, a photo of Einstein, etc. You can even try writing an email to an imaginary friend explaining the problem. The key is to describe the problem carefully in natural language. I like this approach so much that I bought rubber ducks for everyone in my lab (Figure A2.1).



Figure A2.1. Rubber ducks for debugging. They're always happy to listen to you talk about your code, and they never judge you.

This page titled 13.3: A2.3 Taking a Scientific Approach is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

13.4: A2.4 Deciphering Matlab's Error Messages

As noted in the previous section, Matlab's error messages are often helpful when you're developing a hypothesis about the cause of your problem. In this section, we'll do some exercises that are designed to help you learn how to decipher these messages.

Before we do the exercises, we need to divide software problems into three categories:

- A **syntax error** occurs when the script contains information that is not legal Matlab code. Often, this is a result of a typo (e.g., a missing comma). When you run a script, Matlab first checks for syntax errors, and the script won't even start running if a syntax error is detected. Instead, an error message is printed in the command window.
- A **runtime error** occurs when the code, while legal in its syntax, produces some kind of problem that causes the script to terminate with an error message in the command window. For example, if the script tries to open a file with a specific name, but no file with that name exists in the PATH, the script will terminate with an error message.
- A **logical error** occurs when the program runs to completion without an error message, but the result is incorrect. This usually means that the code in the script does not correctly implement the desired processing steps, but it sometimes means that the input data violate the assumptions of the script.

Note that only syntax errors and runtime errors produce error messages. Logical errors are usually the most difficult to diagnose and solve.

Now it's time to try some exercises. These exercises require scripts and data in the Appendix_2 folder in the master folder: https://doi.org/10.18115/D50056.

Exercise A2.1: A Syntax Error

Launch EEGLAB, make Appendix_2 the current folder, and double-click on **Test_Case_1.m** in the Current Folder panel of the Matlab GUI. This should open the script in the Matlab editor. It's a very simple script, but it has a bug. Click the **Run** button in the editor window to run the script. You should see the following error message in the Command Window:

Error: File: Test_Case_1.m Line: 3 Column: 30
Character vector is not terminated properly.

This message tells you where in the script it detected a problem (Line 3, Column 30), and it tells you the nature of the problem ("Character vector is not terminated properly"). However, the nature of the problem is described in Programmerese, so you might not understand what the error message means.

However, at least it tells you what line was running when the error message was generated (Line 3). Take a look at Line 3 of the **Test_Case_1.m** script. This line is trying to open a dataset named **1_N170.set**. Notice that the text **'1_N170.set);** is underlined in red. This is Matlab's way of telling you that it think this part of the code is a syntax error. If you hover your mouse over this part of the code, some more Programmerese pops up, saying "A quoted character vector is unterminated." Even if you don't fully understand this message, you should be able to infer that there is some problem with the termination of the filename, **1_N170.set**.

This filename is specified as a text string, and Matlab text strings need to be enclosed in single quote characters (e.g., '**1_N170.set**'). Otherwise Matlab doesn't know where the string starts and stops. In this script, we are missing the single quote that terminates the string. That's what Matlab means when it tells you "Character vector is not terminated properly". This is a very common error that I made at least a dozen times when preparing the scripts in this book.

Try adding a single quote mark after **.set** in the script. You'll see that the red underline disappears from under this string. Now try running the script. Voila! No more error message.

If you can't figure out an error message in this way, try Googling it (e.g., with the search phrase **matlab "character Vector Is Not Terminated Properly"**). When I did this, I found a page on stackallflow.com that proposed a good solution.

Exercise A2.2: A More Subtle Syntax Error

Close **Test_Case_1.m** and double-click on **Test_Case_2.m** to load it in the Matlab editor. Click the **Run** button in the editor window to run the script. You should see the following error message in the Command Window:

File: Test_Case_2.m Line: 4 Column: 26
Invalid expression. When calling a function or indexing a variable, use parentheses.
Otherwise, check for mismatched delimiters.

Take a look at Line 4 of the script. The error message says that this is an "Invalid expression" — not very informative.

Do you see anything marked as problematic by Matlab in the editor window? If you look very closely, you'll see that a right square bracket near the middle of the line is underlined in red, as is a right parenthesis near the end of the line. Matlab isn't sure which one of these is the source of the problem, so you now have two hypotheses to examine.

When there are problems with square brackets or parentheses (which Matlab calls "delimiters"), this usually means that they're not paired properly. That is, every left bracket needs a corresponding right bracket. To fix a problem like this, you need to make sure that each delimiter has a pair and that both delimiters are in the correct places. If you look closely at Line 4, you'll see that there is a left parenthesis that is correctly paired with the underlined right parenthesis.

But if you look at the square brackets, you'll see that there are two right brackets and only one left bracket. This means you need to figure out if we are missing a left bracket or if we have a right bracket that shouldn't be there. In this case, the problem is an extra right bracket. Try deleting the underlined right bracket. You should see that the red underlines disappear, including the one under the right parenthesis. Now try running the script. It should now run properly.

Exercise A2.3: A Simple Runtime Error

Close **Test_Case_2.m**, double-click on **Test_Case_3.m,** and click the **Run** button to run the script. You should see the following error message in the Command Window:

Unrecognized function or variable 'EGG'.
Error in Test_Case_3 (line 4)
[ALLEEG, EEG, CURRENTSET] = eeg_store(ALLEEG, EGG, 0);

This error message is actually reasonably easy to understand: There is a variable named **EGG** on Line 4 that isn't recognized. There are usually two possible explanations for this kind of problem. The first is that the variable is defined in the wrong part of the program, so it hasn't yet been defined when Matlab tries to execute Line 4. You can just search for the variable to see if it's defined later. The second common explanation is a simple typo. In this case, it's pretty clear that variable is supposed to be named **EEG**, not **EGG**. If you change the name to EEG and run the script, you'll find that the problem has been eliminated.

Note that Matlab doesn't underline any of the code in the script when it detects this error. That's because it's a runtime error rather than a syntax error. In other words, Line 4 is perfectly legitimate Matlab code when considered in isolation. It's only a problem in the context of the rest of the code (i.e., because no variable named **EGG** has been defined on lines that execute before Line 4).

Exercise A2.4: A Slightly More Complicated Runtime Error

Close **Test_Case32.m**, double-click on **Test_Case_4.m,** and run the script. You should see the following error message in the Command Window:

Error using load
Unable to find file or directory
'/Users/luck/Dropbox/Research/Manuscripts/ERP_Analysis_Book/Appendix_2_Troubleshooting/Exercises1_N170.set'.
Error in pop_loadset (line 139)
        TMPVAR = load('-mat', filename);
Error in Test_Case_4 (line 7)
EEG = pop_loadset('filename', Dataset_filename);

This error message is a little more complicated. The script calls an EEGLAB function named **pop_loadset**, and Matlab actually detected the error while **pop_loadset** was executing. The error message therefore tells you what line of **pop_loadset** was running when the error was detected, along with the line of **Test_Case_4.m** that called **pop_loadset**.

When you get an error message like this, it probably means that you have sent some kind of invalid data to the **pop_loadset** function. It's possible that there is a bug in an EEGLAB or ERPLAB function that you are calling, but it's much more likely that the problem is originating in your script. However, if you can't figure out the problem by looking at your script, you might want to open the script for the function that your script is calling. You can do that by selecting the name of the function within your script, right-clicking on it, and selecting **Open "pop_loadset"**.

I'm not actually going to reveal the problem with **Test_Case_4.m** here. That will be revealed in the next section.

This page titled 13.4: A2.4 Deciphering Matlab's Error Messages is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

13.5: A2.5 Debugging Scripts by Performing Experiments and Collecting Data

Earlier, I said that you should act like a scientist when debugging code, which include performing experiments and collecting data. In this section, I'll describe some simple methods for doing this.

Scripting is all about storing information in variables and applying operations to those variables (e.g., addition, subtraction). When there is a runtime error or logical error, some variable probably has an incorrect value. A key aspect of debugging is therefore assessing the values of the variables at different points in the execution of the script. This is much like monitoring the representations of the brain at different time points following the onset of a stimulus while a participant performs a task.

There are a couple ways to monitor variables. The simplest is to print their values to the command window at key points during the execution of the script. To see how this works, let's use this approach to understand what is wrong with the **Test_Case_4.m** script.

Exercise A2.5: Displaying Variable Values

Open the script (if it's not already open) and take a look at it. The goal of the script is to load a dataset named **1_N170.set** that is located in the current folder (the same folder as the script). To achieve this, we start by getting the path to the current folder with the **pwd** (*print working directory*) function and store the result in a variable named **DIR**. We then concatenate this value with **'1_N170.set'** to create the full path to the file, and we store the result in a variable named **Dataset_filename**. We then send **Dataset_filename** to the **pop_loadset** function. But we're getting an error message indicating that Matlab can't find the file that we've specified in this manner.

To figure out the source of the error, let's examine the values of the **DIR** and **Dataset_filename** variables. To accomplish this, add the following two lines to the script, right after line 5 (i.e., after the lines of code that set the values of these variables):

```
display(DIR);
display(Dataset_filename);
```

These commands will display the values of the **DIR** and **Dataset_filename** variables to the Command Window. To see this, run the script. You should see the same error message in the Command Window as before, but prior to that you should see something like this:

```
DIR =
'/Users/luck/ERP_Analysis_Book/Appendix_2_Troubleshooting/Exercises'
Dataset_filename =
'/Users/luck/ERP_Analysis_Book/Appendix_2_Troubleshooting/Exercises1_N170.set'
```

The value of **DIR** will be different on your computer because your current folder is not the same as mine. However, if you look closely at it, it should be correct.

If you look at the value of **Dataset_filename**, you should see the error: there is no slash (or backslash) between **Exercises** and **1_N170.set**. In other words, the script is telling Matlab to look for a file named Exercises1_N170.set instead of a file named **1_N170.set** inside the **Exercises** folder.

To fix this problem, change Line 5 from this:

```
Dataset_filename = [DIR '1_N170.set'];
```

to this:

```
Dataset_filename = [DIR filesep '/' '1_N170.set'];
```

(but use '\' instead of '/' if you are on a Windows machine). Now run the code. You will see that the **Dataset_filename** variable now

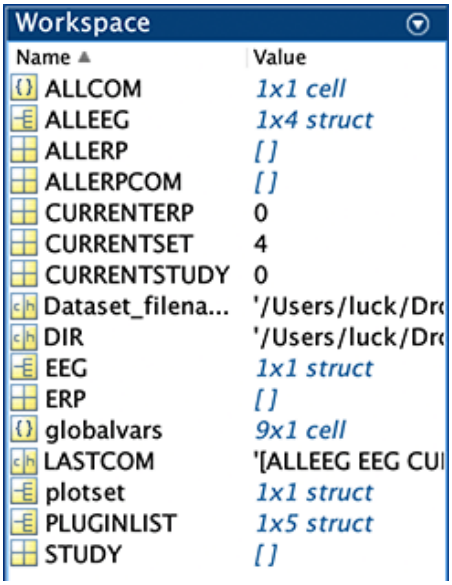
has a slash (or backslash) between the folder name and the file name, and everything now works correctly.

By the way, I recommend learning how to use the **fprintf** command to display the values of variables. It's much more powerful than the **display** command. It's explained briefly in Chapter 10, and you can Google it to find more details. Very worth knowing!

Exercise A2.6: The Workspace Pane and the Variables Pane

Once a script has stopped running, you can see the valuable of a variable by simply typing the name of the variable in the Matlab command window. Try this by typing **Dataset_filename**. You can also see the value of a variable by looking at the *Workspace* pane in the Matlab GUI (which is probably at the right side of the GUI). This pane should look something like Screenshot A2.1.

Screenshot A2.1

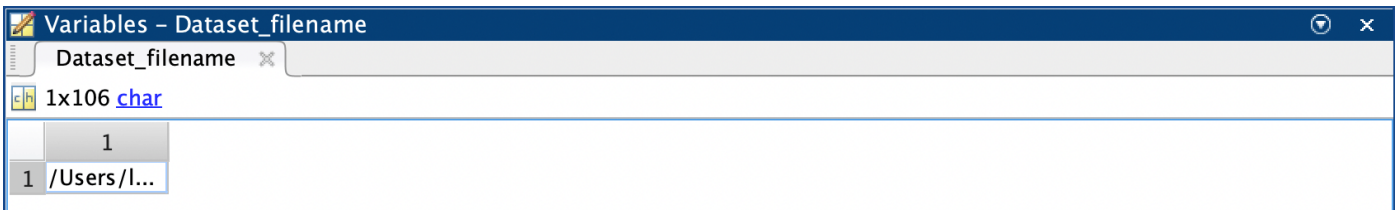


This pane shows all the variables that Matlab knows about. For example, I currently have four datasets loaded, and the fourth dataset is currently active, so **CURRENTSET** has a value of 4. For more complicated variables, the dimensions of the variable are shown instead of the value. For example, **ALLEEG** is a data structure that holds all of the datasets, and it is listed as a **1x4 struct** to indicate that it is a structure that is 1 row high by 4 columns wide (because I have 4 datasets loaded).

You can see that the Workspace pane in Screenshot A2.1 also contains the **DIR** and **Dataset_filename** variables that were created by the last script I ran, **Test_Case_4.m**. You should also have these variables listed; if you don't, run **Test_Case_4.m** again (after fixing the bug). The strings stored in these variables are too long to be seen well in the Workspace pane, but Matlab has a *Variables* pane that you can use to inspect variables more carefully.

To see this, double-click on the **Dataset_filename** variable in the Workspace pane. You should see the Variables pane open up in the Matlab GUI (probably above the command window). It should look something like Screenshot A2.2.

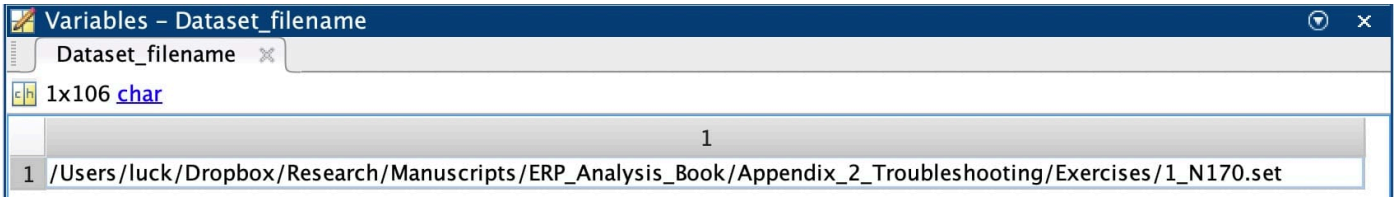
Screenshot A2.2



The value of the **Dataset_filename** variable is now shown (in the little box with "1" to the left and "1" to the right). However, the

box is too small to see much of the variable. If you place your mouse over the right edge of the box with the "1" above the variable value and drag rightward, the box will expand, and you'll be able to see the whole variable (as in Screenshot A2.3).

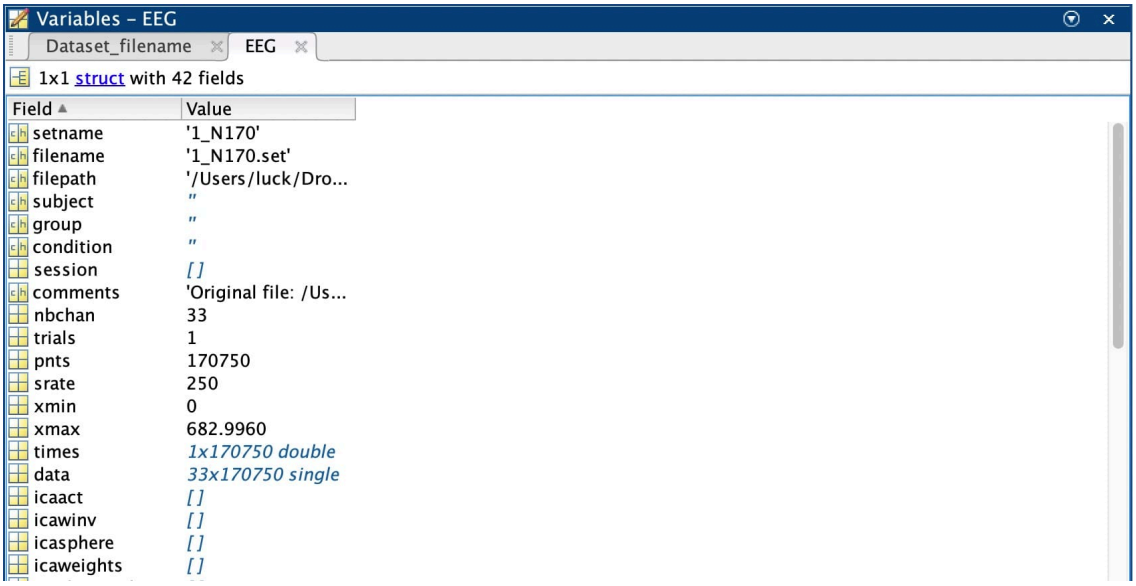
Screenshot A2.3



If you've fixed the bug in **Test_Case_4.m**, you should be able to see the slash (or backslash) between **Exercises** and **1_N170.set**. This might seem like a lot of work to see the contents of a variable, but once you're practiced at this approach, it will become very efficient. Also, it's a great way to look at more complex variables.

For example, double-click on the **EEG** variable in the Workspace. The Variables pane should now look something like Screenshot A2.4. You can easily see the various fields of this complex data structure, such as the number of channels (the **nbchan** field) and the number of time points (the **pnts** field). The actual voltage values are stored in the **data** field, and you can see that this field is 33 rows high (one row for each of the 33 channels) and 170750 columns wide (one column for each of the 170750 time points). Double-click on the **data** field, and now you can see the actual voltage values for each combination of channel and time point. Try double-clicking on other fields as well to get a sense of what information is held in the **EEG** variable.

Screenshot A2.4



Exercise A2.7: Interrupting a Script to View Variable Values

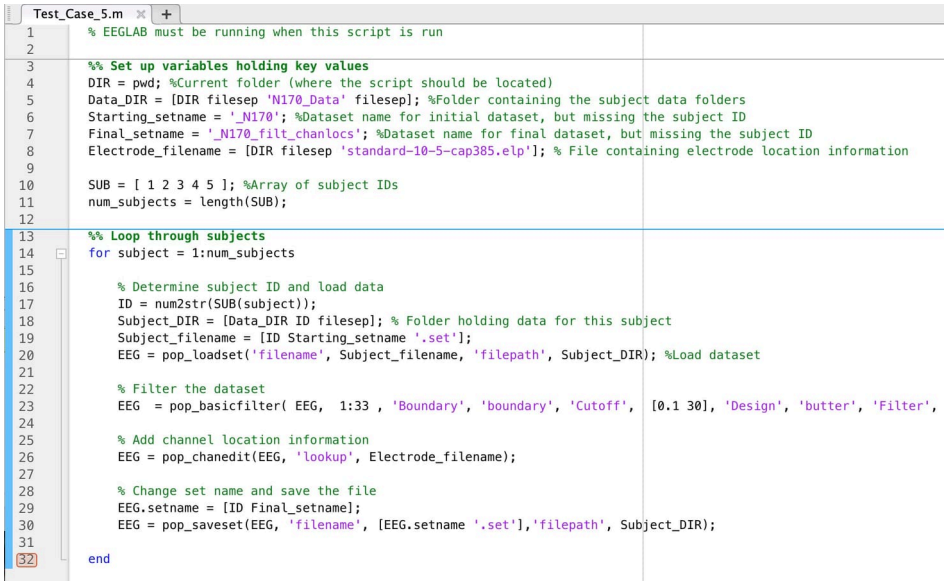
The Workspace and Variables panes are great ways to see the values of variables, but you can ordinarily use this approach only after a script is finished. But Matlab also contains a *debugger* that allows you to pause the operation of a script at various times so that you can examine the values of the variables at those times.

To see this in action, load the **Test_Case_5.m** script. This script is much more complicated than the previous test cases, but it is actually a simplified version of the **Step1_pre_ICA_processing.m** script from Chapter 10. It loops through the data from the first five subjects in the N170 experiment, loading each subject's dataset, filtering it, adding channel location information, renaming it, and then saving the new dataset to the disk.

There isn't a bug in this script, but let's imagine that there was a problem and that you suspected that the problem arose in the loop that starts on Line 14 and ends on Line 32. If you just start running the script, you won't be able to inspect the values of the variables using the Workspace and Variable panes until the script ends. However, you can set a *breakpoint* that causes the program to pause at the end of each time the program runs through the loop.

To set a breakpoint at a given line, you simply click the corresponding line number in the editor window for the script (but make sure that the script has been saved first if you've made any changes). For example, Screenshot A2.5 shows what happened when I clicked on the **32** corresponding to the line number for the **end** statement at the end of the loop (Line 32). This line number is now highlighted in a red box.

Screenshot A2.5



Go ahead and click on the line number for Line 32 in the **Test_Case_5.m** script. It should now have a red box around it, as in Screenshot A2.5. Now run the script. You should see that the script runs, printing a bunch of information in the command window. However, it has only run through the loop once and has paused. You can tell that it has paused because the command window prompt is now **K>>** instead of just **>>**. You can also see a green arrow next to the **end** statement in the editor window for the script. And if you look at the Workspace, you can see that the **subject** variable (which loops from 1 to 5 in this script) is set to **1**.

Now let's look at the current value of the **EEG** data structure. You may already be showing **EEG** in the Variables pane, in which case you are looking at the current value of this variable. If it's not already showing, you can double-click **EEG** in the Workspace pane to look at it in the Variables pane. You can see that the set name has been correctly updated to **1_N170_filt_chanlocs** (although seeing the full name of this field may require double-clicking on the **setname** field in the Variable pane.

You can resume execution of the script by clicking the Continue button in the toolbar along the top of the editor window for **Test_Case_5.m**. When you do that, the script resumes and runs through the loop one more time. If you now look at the value of the **setname** field of **EEG** in the Variables pane, you'll see that it has been updated to **2_N170_filt_chanlocs**. You can click the **Stop** button in the editor window's toolbar to quit from the script.

This feature of Matlab is very useful for debugging. You can read more about it in the Matlab documentation for breakpoints. Other related tools are described in Matlab's general debugging documentation.

This page titled 13.5: A2.5 Debugging Scripts by Performing Experiments and Collecting Data is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

13.6: A2.6 Avoiding Bugs in Your Scripts with Good Programming Practices

The first step in debugging a script is to get in a time machine, go back to the moment when you started writing the script, and tell yourself that you're now wasting a huge amount of time trying to debug the script. "Please," you should tell your earlier self, "follow good programming practices while writing this script so that I won't need to waste so much time. I'm even busier now than I was when I first wrote the script."

If you don't have a time machine, you should resolve to follow good programming practices from now on. It will take a little more time now, but you will be giving a great gift to your future self. As Ben Franklin famously said, "An ounce of prevention is worth a pound of cure."

This section contains a set of good programming practices that I find to be particularly relevant for scientists who are analyzing EEG/ERP data in Matlab and are relatively new to coding.

Rapid cycling between coding and testing

Perhaps the most common mistake that novice coders make is trying to write an entire script without doing any testing along the way. If you write a 30-line script, you will probably have 8 different errors in the script, and it will be really hard to figure out what's going wrong.

As I mentioned before, the best approach is to write a small amount of code, test it, debug it if necessary, and then add more code. When you're new to programming, this might be only 1-3 lines of code at a time. As you gain experience, you can write more lines before testing, but even an experienced programmer usually does some testing after every 20-40 new lines of code.

Define all values as variables at the top of the script

If you've already read the chapter on scripting, you'll know that this is my #1 principle of writing good code. For example, in the N170 experiment that is the focus of the chapter, we analyzed the data from subjects 1-10, but leaving out subject t. When we looped through the subjects, we needed a line of code like this:

```
for subject = [ 1 2 3 4 6 7 8 9 10 ] # Note that 5 is missing from this list
```

Imagine that we need to loop through the subjects in three different parts of the script (e.g., once for pre-ICA EEG processing, once for post-ICA EEG processing, and once for ERP processing). We could just repeat that same loop in each of these three different parts of the script. But now imagine that, a year after we've analyzed the data, we get reviews back from a journal and a reviewer wants us to reanalyze the data without excluding subject 5. Now we need to find all the parts of the script with this loop and modify them. Will we remember that we had three loops? There's a good chance that we will have forgotten and won't find all three of them. As a result, we will have a bug. And we will either end up with the wrong result or waste hours of time trying to find the problem.

To avoid this problem, you should always, **always**, **ALWAYS** use a variable at the top of the script to define a list like this. Here's an example:

```
% This line is in the top section of the script
SUB = [ 1 2 3 5 6 7 8 9 10 ]; % Array of subject IDs, excluding subject 5

% This is how we use the list later in the script
for subject = SUB
```

The same principle applies to individual numbers (e.g., the number of subjects) and strings (e.g., a filename).

Even if you understand and appreciate this advice, it's easy to ignore it by saying to yourself, "This script is just a few lines. I don't need to worry about putting the values into variables at the top." Most long scripts start as short scripts, and this is just being

shortsighted. So, at the risk of repeating myself, you should always, **always**, **ALWAYS** use a variable at the top of the script to define values.

Note that zeros and ones can be an exception to this rule when they are being used more conceptually. For example, zero and one are sometimes used to mean TRUE and FALSE. Or you might do something like this:

```
% This line is in the top section of the script
SUB = [ 1 2 3 5 6 7 8 9 10 ]; % Array of subject IDs, excluding subject 5
num_subjects = length(SUB); % Number of subjects

% This is how we use the list later in the script
for subject_num = 1:num_subjects
    subject = SUB(subject_num);
    % More code here to process the data from this subject
end
```

Make your code readable

The reality of science is that you will often start a script, come back to it a few weeks later to finish it, but then modify it 18 months later (after you get the reviews for a manuscript). And someone else may get a copy of your script and modify it for their own studies. If the code isn't easily readable, bugs are likely to be introduced at these times. Here are a few simple things you can make your code more readable:

- Include lots of internal documentation in your scripts. It's a great gift to your future self.
- Define all values as variables at the top, as noted before, but also make sure there is a comment indicating the purpose of each variable
- Divide your code into small, modular sections (or separate functions), with a comment at the beginning of each section or function that explains what that section or function does
- Use intrinsically meaningful variable names (e.g., **num_subjects** instead of **ns**) and function names (e.g., **ploterps** instead of **npbd**). I wasted a couple hours one night in my first year of graduate school because someone had used the name **npbd** for a function that plotted ERP waveforms, and I'm still bitter...

You can find more discussion of the importance of readability in the scripting chapter.

Make your code modular

If you have a single script that is more than 200 lines long, it should probably be broken into a sequence of multiple scripts. It's a lot harder to find problems in a long script than in a short script. And it's a lot easier to introduce problems into a long script (e.g., by adding code to the wrong section). For example, the EEG/ERP processing pipeline in Chapter 10 consists of a series of 7 scripts. In the ERP CORE experiments, we had about 20 different scripts for each individual experiment.

Make your code portable by using relative paths

Almost all EEG/ERP processing scripts need to access files via a path. The worst way to handle this is something like this (for loading a dataset):

```
EEG = pop_loadset('filename', '/Users/luck/ERP_Analysis_Book/Appendix_2_Troubleshooting/Exercises/1_N170.set');
```

This violates the principle of defining all values at the top of the script. A better, but still problematic, approach is this:

```
% Variables defined at the top of the script
Data_DIR = '/Users/luck/ERP_Analysis_Book/Appendix_2_Troubleshooting/Exercises/';
setname = '1_N170.set';

% Loading the data later in the script
EEG = pop_loadset('filename', setname, 'filepath', Data_DIR);
```

The problem with this approach is that it will break if you switch to a different computer, move your data to a different location, or share your script with someone else.

A better approach is to determine the path from the location of the script (assuming that the script is kept with the data):

Data_DIR = pwd; %Current folder (where the script should be located)

Chapter 10 describes this in much more detail.

Code review

It is becoming very common (and sometimes required) for researchers to post their data analysis code online along with their data when publishing a paper. That way, other researchers can verify that they get the same results with your data and can use your code in their own studies. I think this is a wonderful trend.

When you realize that other people will be looking at and running your code, this tends to increase the pressure to make sure that the code actually works correctly. In theory, you should already be highly motivated to make sure that your code works, because your findings depend on code that works correctly. But public scrutiny is often an even stronger motivator.

A really good way to make sure that your code works correctly is to use *code review*. This is just a fancy term for having someone else go through your code to make sure it's correct. Of course, code review is a lot easier and more effective if you've made your code readable and portable. The person who reviews your code will also likely have suggestions for making your code even more readable and will provide a good test of whether your code is portable (i.e., whether it works on the reviewer's computer).

This page titled 13.6: A2.6 Avoiding Bugs in Your Scripts with Good Programming Practices is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

13.7: A2.7 References

Feyerabend, P. (1993). *Against Method* (3rd ed.). Verso.

Thomas, D., & Hunt, A. (2019). *The Pragmatic Programmer: Your journey to mastery, 20th Anniversary Edition*. Addison-Wesley Professional.

This page titled 13.7: A2.7 References is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

14: Appendix 3: Example Processing Pipeline

This appendix provides an example of a typical EEG preprocessing pipeline for a single participant, including all the steps prior to averaging. It is nearly identical to the pipeline scripts provided in the second half of the chapter on Scripting (Chapter 10). Much of the logic for the ordering of the steps is spelled out in Luck (2014), especially Appendix 1.

This pipeline is optimized for the kinds of experiments typically conducted in my lab, such as the ERP CORE experiments, in which highly cooperative college student participants are tested with in relatively simple cognitive tasks. Changes will often be necessary for different types of tasks and/or participants. In other words, don't blindly follow this pipeline!

Step 1: Preprocessing Prior to ICA-Based Artifact Correction

- Import the participant's original EEG data file(s) into EEGLAB
  - If there are multiple data files (e.g., one per trial block), combine them into a single dataset.
    - If the cap was removed and replaced between data files (e.g., because you had recording sessions on multiple days), you should merge the data files after the ICA step (because the electrodes might not be in exactly the same places for the different files, which will mess up ICA but isn't a big deal for most other kinds of analyses).
    - If a given channel is bad in a subset of the blocks, you could interpolate that channel for those blocks at this stage. If you do this, you should apply the filtering step described below at this stage, which will make the interpolation work better. (And then you will skip the filtering at the later stage.)
  - If your data were recorded without a reference (usually this is only for the BioSemi ActiveTwo system), you may receive a warning that strongly encourages you to reference the data now. Do not do it. You will reference the data at a later step.
  - Recommended: Save this dataset as a file
- If appropriate, shift the stimulus event codes in time to account for any consistent mismatch between the time of the event code and the actual stimulus onset time
  - All LCD displays interpose a constant delay between the time that the video information arrives from the computer's video card and the time that the image is presented on the screen. This delay cannot be detected by the computer and cannot be determined reliably from the manufacturer's specifications. It must be measured with some kind of photosensor. This delay is often greater than 25 ms, so it must be accounted for.
  - Almost all video displays (LCD and CRT) have a delay that depends on the vertical position of the stimulus on the display. This delay is typically ~15 ms for the bottom of the display relative to the top (assuming a refresh rate of 60 Hz). If you are using stimuli at very different vertical locations in your different experimental conditions, you should measure the delay separately for the different locations and adjust accordingly. For large stimuli that are vertically centered, we measure the delay in the vertical center of the display.
  - Delays may also occur for stimuli in other modalities. The delay should be measured and, if substantial, accounted for by shifting the event codes.
  - Make sure that you shift only the stimulus event codes and not the response event codes (unless the there is also a delay for the responses). Use a positive shift value because the event code needs to be shifted later in time (because the stimulus occurred after the event code).
  - ERPLAB's function for shifting event codes creates a CSV file showing how each event code was shifted. You should look at this file to verify that the shifting was correct.
  - It's also important to verify that the delay is consistent (±1 sample period). If it is not consistent, there is probably a problem with the stimulus presentation script.
- If desired, downsample your data to a lower sampling rate
  - My lab typically records at 500 Hz and then downsamples to 250 Hz. This gives us a higher sampling rate if we ever need it, but downsampling to 250 Hz makes the subsequent data processing faster and the data files half as large.
  - Note that EEGLAB's pop_resample routine will automatically apply an appropriate antialiasing filter prior to the resampling process.
- Apply a bandpass filter to the EEG and EOG channels
  - We ordinarily use a non-causal Butterworth impulse response function, 0.1–30 Hz half-amplitude cut-off, 12 dB/oct roll-off.
    - Some researchers prefer a lower cutoff for the high-pass filter (e.g., .05 Hz or .01 Hz), especially when looking at slow components like the late positive potential. However, we find that 0.1 Hz works best in most cases. Do not use anything higher than 0.1 Hz unless you really know what you're doing!
    - You should apply the option for removing the DC offset in the data prior to filtering. This can reduce the edge

artifacts that occur at the beginning and end of each recording period.
    - If extensive 60 Hz noise is present, you can apply the cleanline plugin (Mitra & Pesaran, 1999; see Bigdely-Shamlo et al., 2015 for important details about implementing this tool). If cleanline doesn't work well for you, you can try the newer Zapline method (de Cheveigné, 2020; Klug & Kloosterman, 2022). Alternatively, you could decrease the low-pass frequency to 20 Hz and/or increase the slope of the low-pass filter to 48 dB/octave (which requires doing the low-pass and high-pass filtering as separate steps).
    - If you have also recorded other signals (e.g., a photosensor), you may not want to filter those channels (or you might want different filtering parameters for those channels).
- Add channel location information specifying the 3-D locations of the electrodes.
- Perform channel operations to create bipolar channels (and to reference the data if they were not referenced during the recording).
    - Create a bipolar HEOG channel (HEOG-left minus HEOG-right or vice versa)
        - If you do not have HEOG electrodes, you can use F7 minus F8 (or just skip this step).
        - If your data already contain a bipolar HEOG channel, you should skip this step.
    - Create a bipolar VEOG channel (VEOG-lower minus the closest electrode that is above the eyes). For example, if the VEOG-lower electrode is under the left eye, the FP1 signal would ordinarily be subtracted from the VEOG-lower signal.
        - If your data already contain a bipolar VEOG channel, you should skip this step.
    - Keep the original, monopolar EOG channels, which will be important for ICA.
    - Make sure that the channel location information is preserved by the Channel Operations routine.
    - If your data were not recorded with a reference, reference the data now.
        - In some systems (e.g., BioSemi ActiveTwo), the data saved to disk are not referenced and you should perform referencing now. However, if your data were referenced during recording, do not re-reference now. The re-referencing will be performed later, after artifact correction.
        - In most labs that use the Brain Products ActiCHamp system, the data are already referenced at this point. However, our lab has a custom version of the data acquisition system in which the data are not referenced. Our lab therefore references at this point (and we re-reference later, after artifact correction).
        - If you are referencing now, use a single electrode site as the reference. If a combination of sites is desired in the long run (e.g., the average of all sites, the average of the mastoids), you will re-reference to that combination later.
        - It doesn't really matter what site is used as the reference at this point. My lab uses P9.
    - Recommended: Save the resulting dataset as a file. We call this dataset the **pre-ICA dataset**.
- View the EEG to make sure everything looks okay and to identify channels that require interpolation. These channels will be left out of the ICA decomposition process. The actual interpolation will be done after the ICA correction has been performed.
    - To identify channels that should be interpolated:
        - Visually inspect the EEG on both short (e.g., 5-second) and long (e.g., 60-second) time scales. This video demonstrates how to perform an initial visual inspection of continuous EEG data https://doi.org/10.18115/D5V638.
        - If the voltage drifts around or there are many sudden changes in voltage, that channel should be interpolated.
        - If a channel is fine for most of the session but shows occasional periods of large deflections, this channel can be interpolated after epoching but limited to epochs in which the large deflections occur (as determined with the artifact detection procedures).
        - If a channel shows a lot of high-frequency noise, it may not actually need to be interpolated.
            - If the main dependent variable will be the mean voltage over some time range (e.g., 200-300 ms for N2pc or 300-500 ms for N400), high-frequency noise is not usually a problem. The best way to tell if it will be a problem is to look at the standardized measurement error (SME). If the SME for that channel is >2 SD beyond the mean SME of the other channels (excluding EOG, Fp1, and Fp2), then the channel should be interpolated. If the channel is the main channel for data analysis, you could interpolate if it is >1.5 SD beyond the mean.
            - If there is a lot of high-frequency noise, the signal may also be corrupted in this channel. To see this, make a note to check the averaged ERP for that channel and surrounding channels after the averaging process. If the channel looks quite different from the surrounding channels, the signal is probably corrupted and the channel should be interpolated (which will require going back to this stage and repeating all the subsequent processes.
            - If the main dependent variable will be a latency value and/or a peak-related variable, high-frequency noise is more likely to be a problem. You can confirm this by computing the bootstrapped SME when you get to the averaging step.
    - Save information about channels that should be interpolated in an Excel file. That makes it possible to repeat the processing with a script.
    - If a given channel misbehaves only occasionally, save this information the Excel file that controls artifact detection.

Step 2: ICA-Based Artifact Correction

- Make a copy of the **pre-ICA dataset**. We will use this new dataset for the ICA decomposition. This new dataset is called the **ICA decomposition dataset**.
- Apply the following operations to the **ICA decomposition dataset**.
  - Bandpass filter with half-amplitude cutoffs at 1 and 30 Hz, 48 dB/octave
    - It's OK that we've double-filtered the data. The original filtering is so much milder that it will be dwarfed by the new filter.
    - If your previous low-pass cutoff was at 20 Hz, use 20 Hz here as well
    - If your previous low-pass slope was 48 dB/octave, just do a high-pass filter at this stage (1 Hz, 48 dB/octave).
    - Note: It is important that this filtering is done before the following steps. Some of the following steps introduce discontinuities in the EEG data, and those can lead to edge artifacts if the filtering is applied later.
  - Resample the data at 100 Hz.
    - This is not strictly necessary, but it makes the ICA decomposition faster.
    - Don't resample if the recording was brief. You need enough data to train the ICA decomposition routine. See the chapter on artifact correction for details on how much data is necessary.
  - Delete break periods, because the EEG is often "crazy" during breaks, which degrades the ICA decomposition.
    - A break is defined as a period of at least X ms without an event code, where X is some reasonable value (e.g., 2000 ms, assuming that you never have 2000 ms between event codes except during breaks).
    - You can tell it to ignore certain event codes. You should always exclude boundary events. You may also want to exclude response events (in case the subject makes button presses during the breaks).
    - You need a buffer at the edges of the break periods so that you do not cut into your eventual epochs (e.g., 1500 ms at the beginning of the break and 500 ms at the end of the break).
  - Delete periods of "crazy" EEG (wild deflections, beyond what you see with ordinary artifacts).
    - This can ordinarily be done using ERPLAB's **Artifact rejection (continuous data)** routine.
    - When setting the parameters, make sure that you are mainly rejecting segments of data with "crazy" EEG and are not rejecting segments with blinks or other ordinary artifacts.
    - You should ordinarily exclude the EOG channels. If you include these channels, you may need a rejection threshold that is too high for the EEG channels. For subjects with unusually large blinks, you may also need to exclude Fp1, Fp2, or any other channels that are very close to the eyes.
    - For most subjects, a threshold of 500 μV and a window size of 1000 ms works well. However, you should visually inspect the results and adjust these parameters if necessary. If blinks exceed the threshold, you will need to increase the threshold (or perhaps exclude Fp1, Fp2, or other channels with huge blink activity). The final parameters should be stored in an Excel spreadsheet for use in scripting.
    - Check the box labeled **Join artifactual segments separated by less than** and put 1000 in the corresponding text box.
    - Note that this deletion of segments with "crazy" EEG is designed only the improve the ICA decomposition. Once the ICA weights are transferred back to the pre-ICA dataset to create the post-ICA dataset (see below), these segments will be present in the data. You will later epoch the post-ICA data and apply ordinary artifact detection to mark and eventually excluded any epochs that contain wild voltage deflections. This way, you will have an accurate count of the number of trials excluded because of artifacts.
  - Recommended: Save the final version of the **ICA decomposition dataset** as a file.
- Perform the ICA decomposition process on the **ICA decomposition dataset**.
  - Make sure that **'extended', 1** is set (it should be set by default). This allows ICA to detect sub-Gaussian components, such as line noise and slow drifts.
  - Exclude the bipolar EOG channels (because we will use these later to see the blinks and eye movements). If your dataset does not contain unipolar EOG signals, the ICA decomposition will still work reasonably well.
  - Exclude "bad channels" that will be interpolated.
  - If the ICA crashes or fails to converge, check the dataset to see if you have a lot of very short segments between boundary events. If so, you may need to alter your procedures for deleting periods of "crazy" EEG (e.g., by increasing the value for **Join artifactual segments separated by less than**).
  - Recommended: Save the result as a new file.
- Examine the components (especially the scalp maps) to make sure that the decomposition worked correctly. You should have 1-2 channels corresponding to blinks (and possibly vertical eye movements) and 1-2 channels corresponding to horizontal eye movements.
  - My lab doesn't ordinarily remove components corresponding to other artifacts, but this could be done when necessary for EKG artifacts and perhaps others.
- Determine which components correspond to artifacts that should be removed. This is done by both examining the scalp

maps and comparing the time course of the components with the time course of the EOG signals. This information should be stored in a spreadsheet so that it can be used in scripts.
- Transfer the ICA weights from the **ICA decomposition dataset** to the **pre-ICA dataset**. The result is called the **post-ICA dataset.**
- Remove the independent components corresponding to the artifacts.
  - Recommended: Save the result as a file.
- Visually inspect the corrected data to make sure that the correction worked properly.

Step 3: Post-ICA EEG Processing (starting with the post-ICA dataset)

- Re-reference the data
  - My lab's current preference is the average of P9 and P10, with the average of the left and right mastoids as a second best alternative. It is usually best to use whatever is most common in your area of ERP research. For example, we use the average of all sites as the reference when we look at the N170 elicited by faces, because that is the most common reference in that research area. This makes it easier to compare ERPs across studies (because a different reference can make the waveforms and scalp distributions look radically different).
  - Also create bipolar HEOG and VEOG from the ICA-corrected EOG channels and look at them to assess the effectiveness of the artifact correction.
  - Keep the uncorrected bipolar HEOG and VEOG signals, which we will use for detecting blinks and eye movements that occurred near that time of stimulus onset and may have interfered with perception of the stimulus.
- Perform the interpolation for the "bad channels" that were identified earlier.
  - Non-EEG channels (e.g., EOG channels, a photosensor channel, a skin conductance channel) should be ignored in the process of computing interpolated values.
  - Note that interpolation should ordinarily be performed after high-pass filtering.
  - If you are using the average of all sites as the reference, you should either perform the interpolation before referencing or exclude any to-be-interpolated sites from the reference.
- Add an EventList to the dataset
- Run BINLISTER to assign events to bins
- Extract bin-based epochs
  - Important: Use ERPLAB's tool (Extract bin-based epochs), not EEGLAB's tool (Extract epochs). The subsequent ERPLAB operations will not work correctly if you use EEGLAB's tool.
  - Our standard epoch is from -200 to +800 ms. A longer prestimulus is used if we will be doing frequency-based analyses or if we want to maximize trial-to-trial stability of the signal (e.g., for decoding, which is extraordinarily sensitive to trial-to-trial variability).
  - Baseline correction is ordinarily applied at this step.
  - Recommended: Save the epoched dataset as a file.

Step 4: Artifact Detection

- General procedure for a given type of artifact
  - Start with default detection parameters
  - Apply to data
  - Check number of trials detected
  - Scroll through data to determine whether the parameters were effective
  - If the parameters were not effective, update them and try again until effective parameters are found
  - The parameters should be stored in a spreadsheet for future scripting.
- Standard artifacts to detect
  - Blinks and eye movements that might interfere with the perception of the stimulus
    - Test for blinks in the uncorrected bipolar VEOG channel and test for eye movements in the uncorrected HEOG channel
    - Use the step function with a window size of 200 ms and a step size of 10 ms
    - The test period should start at -200 ms (because vision is functionally suppressed for ~100 ms after the blink is complete, so we want to detect blinks even if they ended shortly before stimulus onset)
    - The test period should be at least 200 ms and should terminate at or after the offset of the stimulus
    - Start with a threshold of 50 μV for blinks and 32 μV for eye movements (which corresponds to a 2° rotation of the eyes)
    - You can often look at the averaged EOG signals to see if you've successfully removed these artifacts.

    - General C.R.A.P. that increases measurement error
        - Apply to all channels except for the uncorrected HEOG and VEOG channels
        - Use both the absolute voltage threshold and moving window peak-to-peak algorithms; they occasionally catch different trials
        - The test period should typically encompass the whole epoch.
        - 150 μV is a good starting threshold.
        - The goal is to reduce measurement error, which you can quantify by looking at the SME. When you discard a trial because of C.R.A.P., you reduce trial-to-trial variability, which decreases measurement error. However, you also reduce the number of trials, which increases measurement error. The SME takes both into account and tells you whether, when both factors are considered, the measurement error is better or worse.
        - You can tolerate more C.R.A.P. in a given channel if that channel will not contribute to your main analyses (e.g., if your analyses will be limited to other channels). In many cases, most of the channels are used only when plotting scalp maps. However, if you will be re-referencing after artifact detection, any channels that are part of the reference are important.
    - Note: Use a different flag for each type of artifact. This makes it possible to track how many artifacts of each type were flagged.
- The number of trials with versus without artifacts should be recorded. We do this aggregated across conditions (i.e., collapsed across bins). If the percentage of rejected trials is greater than 25%, my lab always excludes the subject from the final analyses. (We raise this threshold to 50% for studies of psychiatric and neurological conditions.) A different threshold for exclusion may be appropriate for other kinds of research. However, it is important for the exclusion criterion to be set prior to looking at the data.
- After a participant's data have been fully processed in this way (including visualizing the data and saving all the participant-specific analysis parameters in one or more spreadsheets), the participant's data should be reprocessed with a script
    - This should yield the same result, but it can avoid errors that may occur in manual processing
    - The ICA decomposition process does not need to be repeated when the data are re-processed with a script. That is, the weights can be transferred from the **ICA decomposition dataset** created during the manual processing.

References

Bigdely-Shamlo, N., Mullen, T., Kothe, C., Su, K.-M., & Robbins, K. A. (2015). The PREP pipeline: Standardized preprocessing for large-scale EEG analysis. *Frontiers in Neuroinformatics*, *9*. https://doi.org/10.3389/fninf.2015.00016

de Cheveigné, A. (2020). ZapLine: A simple and effective method to remove power line artifacts. *NeuroImage, 207,* 116356. https://doi.org/10.1016/j.neuroimage.2019.116356

Klug, M., & Kloosterman, N. A. (2022). Zapline-plus: A Zapline extension for automatic and adaptive removal of frequency-specific noise artifacts in M/EEG. *Human Brain Mapping, 43(9)*, 2743–2758. https://doi.org/10.1002/hbm.25832

Luck, S. J. (2014). *An Introduction to the Event-Related Potential Technique, Second Edition*. MIT Press.

Mitra, P. P., & Pesaran, B. (1999). Analysis of Dynamic Brain Imaging Data. *Biophysical Journal*, *76*(2), 691–708. https://doi.org/10.1016/S0006-3495(99)77236-X

This page titled 14: Appendix 3: Example Processing Pipeline is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

INDEX

A

analytic standardized measurement error
6.7: Exercise - The Signal-to-Noise Ratio

C

current density
5.8: Exercise- Current Density

G

global field power
5.9: Exercise- Global Field Power

H

Hillyard Principle
6.2: Design of the ERP CORE Visual Oddball P3b Experiment

O

oddballs
6.2: Design of the ERP CORE Visual Oddball P3b Experiment

This page titled Index is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

GLOSSARY

S

Sample Word 1 | Sample Definition 1

This page titled Glossary is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.

DETAILED LICENSING

Overview

Title: Applied Event-Related Potential Data Analysis (Luck)

Webpages: 182

All licenses found:

- CC BY 4.0: 98.4% (179 pages)
- Undeclared: 1.6% (3 pages)

By Page

- Applied Event-Related Potential Data Analysis (Luck) - *CC BY 4.0*
  - Front Matter - *CC BY 4.0*
    - TitlePage - *CC BY 4.0*
    - InfoPage - *CC BY 4.0*
    - Table of Contents - *Undeclared*
    - Hardware and Software Requirements - *CC BY 4.0*
    - Licensing - *Undeclared*
    - Preface - *CC BY 4.0*
    - Acknowledgments - *CC BY 4.0*
    - How to Cite This Book - *CC BY 4.0*
  - 1: First Steps - *CC BY 4.0*
    - 1.1: Getting Started - *CC BY 4.0*
    - 1.2: Installing the Software and Downloading the Data - *CC BY 4.0*
    - 1.3: Exercise- Loading an EEG File - *CC BY 4.0*
    - 1.4: Exercise - Viewing Continuous EEG Waveforms - *CC BY 4.0*
    - 1.5: Exercise - Viewing EEG Spectra - *CC BY 4.0*
    - 1.6: Exercise- Loading ERPs and Plotting ERP Waveforms - *CC BY 4.0*
    - 1.7: Exercise- Plotting ERP Scalp Maps - *CC BY 4.0*
    - 1.8: Finding the Right Routine in EEGLAB and ERPLAB - *CC BY 4.0*
    - 1.9: Understanding the Matlab Path - *CC BY 4.0*
    - 1.10: Key Takeaways and References - *CC BY 4.0*
  - 2: Processing the Data from One Participant in the ERP CORE N400 Experiment - *CC BY 4.0*
    - 2.1: Data For This Chapter - *CC BY 4.0*
    - 2.2: Design of the N400 Experiment - *CC BY 4.0*
    - 2.3: Exercise - Looking at the EEG and the Event Codes - *CC BY 4.0*
    - 2.4: Exercise- Filtering Out Low-Frequency Drifts from the EEG - *CC BY 4.0*
    - 2.5: Exercise- Creating an EventList - *CC BY 4.0*
    - 2.6: Exercise- Assigning Events to Bins with BINLISTER - *CC BY 4.0*
    - 2.7: Exercise- Epoching and Baseline Correction - *CC BY 4.0*
    - 2.8: Exercise- Artifact Detection - *CC BY 4.0*
    - 2.9: Exercise- Averaged ERPs - *CC BY 4.0*
    - 2.10: Exercise- Data Quality - *CC BY 4.0*
    - 2.11: Review of Processing Steps - *CC BY 4.0*
    - 2.12: A Simple Matlab Script - *CC BY 4.0*
    - 2.13: Key Takeaways and References - *CC BY 4.0*
  - 3: Processing Multiple Participants in the ERP CORE N400 Experiment - *CC BY 4.0*
    - 3.1: Data for This Chapter - *CC BY 4.0*
    - 3.2: Exercise- Preprocessing and Averaging the Data from 10 Participants - *CC BY 4.0*
    - 3.3: Exercise- Examining the Single-Participant ERPsets - *CC BY 4.0*
    - 3.4: Exercise- "Bad" Data - *CC BY 4.0*
    - 3.5: Exercise- Making a Grand Average - *CC BY 4.0*
    - 3.6: Exercise- Low-Pass Filtering - *CC BY 4.0*
    - 3.7: Exercise - Scoring N400 Amplitude - *CC BY 4.0*
    - 3.8: Exercise- Simple Statistical Analysis of N400 Data - *CC BY 4.0*
    - 3.9: Exercise- A More Complex Analysis - *CC BY 4.0*
    - 3.10: Exercise- ERP Channel Operations - *CC BY 4.0*
    - 3.11: Exercise- ERP Bin Operations - *CC BY 4.0*
    - 3.12: Review of Processing Steps - *CC BY 4.0*
    - 3.13: Matlab Scripts For This Chapter - *CC BY 4.0*
    - 3.14: Key Takeaways and References - *CC BY 4.0*
  - 4: Filtering the EEG and ERPs - *CC BY 4.0*
    - 4.1: Data for this Chapter - *CC BY 4.0*
    - 4.2: Classes of Filters - *CC BY 4.0*
    - 4.3: Exercise- Assessing the Frequency Content of the Noise - *CC BY 4.0*
    - 4.4: Exercise- Filtering the Artificial Waveforms

- *CC BY 4.0*
    - 4.5: Exercise- The Impulse Response Function - *CC BY 4.0*
    - 4.6: Exercise- Applying the Impulse Response Function to a Series of Impulses - *CC BY 4.0*
    - 4.7: Background- Filtering with a Running Average - *CC BY 4.0*
    - 4.8: Background- Filtering with a Weighted Running Average - *CC BY 4.0*
    - 4.9: Exercise- Distortion of Onset and Offset Times by Low-Pass Filters - *CC BY 4.0*
    - 4.10: Exercise- High-Pass Filtering - *CC BY 4.0*
    - 4.11: Practical Advice - *CC BY 4.0*
    - 4.12: Exercise- Creating and Importing Artificial Waveforms - *CC BY 4.0*
    - 4.13: Matlab Script for this Chapter - *CC BY 4.0*
    - 4.14: Key Takeaways and References - *CC BY 4.0*
- 5: Referencing and Other Channel Operations - *CC BY 4.0*
    - 5.1: Data for This Chapter - *CC BY 4.0*
    - 5.2: Background- Understanding Active, Reference, and Ground Electrodes - *CC BY 4.0*
    - 5.3: Exercise- Working with the Artificial Data - *CC BY 4.0*
    - 5.4: Exercise- Average Mastoids as the Reference - *CC BY 4.0*
    - 5.5: Exercise- Re-Referencing the N400 ERP CORE Data - *CC BY 4.0*
    - 5.6: Exercise- The Average Reference - *CC BY 4.0*
    - 5.7: What is the Best Reference Site? - *CC BY 4.0*
    - 5.8: Exercise- Current Density - *CC BY 4.0*
    - 5.9: Exercise- Global Field Power - *CC BY 4.0*
    - 5.10: Exercise- Referencing the EEG Data from the ERP CORE N400 Experiment - *CC BY 4.0*
    - 5.11: Exercise- Other Common Re-Referencing Scenarios - *CC BY 4.0*
    - 5.12: Matlab Script For This Chapter - *CC BY 4.0*
    - 5.13: Key Takeaways and References - *CC BY 4.0*
- 6: Assigning Events to Bins, Averaging, Baseline Correction, and Assessing Data Quality - *CC BY 4.0*
    - 6.1: Data for This Chapter - *CC BY 4.0*
    - 6.2: Design of the ERP CORE Visual Oddball P3b Experiment - *CC BY 4.0*
    - 6.3: The Event Code Scheme - *CC BY 4.0*
    - 6.4: Overview of Bin Descriptor Files - *CC BY 4.0*
    - 6.5: Exercise - A Basic Assignment of Events to Bins - *CC BY 4.0*
    - 6.6: Exercise - Looking at the Averaged ERPs - *CC BY 4.0*
    - 6.7: Exercise - The Signal-to-Noise Ratio - *CC BY 4.0*
    - 6.8: Exercise - Response-Locked Averaging - *CC BY 4.0*
    - 6.9: Exercise - Comparing Correct and Error Trials - *CC BY 4.0*
    - 6.10: Exercise - Sequential Analysis of the P3b - *CC BY 4.0*
    - 6.11: Exercise - Combining Bins - *CC BY 4.0*
    - 6.12: Exercise - Overlap - *CC BY 4.0*
    - 6.13: Matlab Script For This Chapter - *CC BY 4.0*
    - 6.14: Key Takeaways and References - *CC BY 4.0*
- 7: Inspecting the EEG and Interpolating Bad Channels - *CC BY 4.0*
    - 7.1: Data for This Chapter - *CC BY 4.0*
    - 7.2: Design of the Mismatch Negativity (MMN) Experiment - *CC BY 4.0*
    - 7.3: Video Demonstration- Performing an Initial Inspection of a Participant's EEG - *CC BY 4.0*
    - 7.4: The Fundamental Goal of EEG Preprocessing - *CC BY 4.0*
    - 7.5: Background- Interpolating Bad Channels - *CC BY 4.0*
    - 7.6: Exercise - Interpolating Bad Channels - *CC BY 4.0*
    - 7.7: Matlab Script For This Chapter - *CC BY 4.0*
    - 7.8: Key Takeaways and References - *CC BY 4.0*
- 8: Artifact Detection and Rejection - *CC BY 4.0*
    - 8.1: Data for This Chapter - *CC BY 4.0*
    - 8.2: Overview - *CC BY 4.0*
    - 8.3: Background- Why Do We Reject Artifacts? - *CC BY 4.0*
    - 8.4: Background- The General Approach - *CC BY 4.0*
    - 8.5: Exercise- Simple Blink Detection - *CC BY 4.0*
    - 8.6: Exercise- Adjusting the Threshold - *CC BY 4.0*
    - 8.7: An Iterative Approach to Setting Parameters - *CC BY 4.0*
    - 8.8: Exercise- Data Quality and Confounds - *CC BY 4.0*
    - 8.9: Exercise- Better Blink Detection - *CC BY 4.0*
    - 8.10: Exercise- Detecting Eye Movements - *CC BY 4.0*
    - 8.11: Exercise- Deciding on a Threshold for Eye Movements - *CC BY 4.0*
    - 8.12: Exercise- Commonly Recorded Artifactual Potentials (C.R.A.P.) - *CC BY 4.0*
    - 8.13: Using Artifact Detection to Avoid Changes to Visual Inputs - *CC BY 4.0*
    - 8.14: The ERP CORE N2pc Experiment - *CC BY 4.0*
    - 8.15: Exercise- Visualizing the Eye Movements - *CC BY 4.0*
    - 8.16: Exercise- Using the Averaged HEOG to Visualize Consistent Eye Movements - *CC BY 4.0*
    - 8.17: Exercise- A Two-Stage Strategy for Eliminating Small But Consistent Eye

Movements - *CC BY 4.0*
- 8.18: Matlab Script For This Chapter - *CC BY 4.0*
- 8.19: Key Takeaways and References - *CC BY 4.0*

◦ 9: Artifact Correction with Independent Component Analysis - *CC BY 4.0*
- 9.1: Data for this Chapter - *CC BY 4.0*
- 9.2: Exercise- A First Pass at ICA-Based Blink Correction - *CC BY 4.0*
- 9.3: Exercise- Evaluating the Impact of Artifact Correction - *CC BY 4.0*
- 9.4: Background- A Quick Conceptual Overview of ICA - *CC BY 4.0*
- 9.5: Exercise- Making ICA Work Better - *CC BY 4.0*
- 9.6: Exercise- Transferring the Weights and Assessing the ICs - *CC BY 4.0*
- 9.7: Exercise- Deciding Which ICs to Exclude - *CC BY 4.0*
- 9.8: Exercise- Deleting C.R.A.P. Prior to ICA - *CC BY 4.0*
- 9.9: General Recommendations - *CC BY 4.0*
- 9.10: Matlab Scripts For This Chapter - *CC BY 4.0*
- 9.11: Key Takeaways and References - *CC BY 4.0*

◦ 10: Scoring and Statistical Analysis of ERP Amplitudes and Latencies - *CC BY 4.0*
- 10.1: Data for This Chapter - *CC BY 4.0*
- 10.2: Design of the Flankers Experiment - *CC BY 4.0*
- 10.3: Exercise- Examining the Grand Averages - *CC BY 4.0*
- 10.4: Exercise- A First Pass at Scoring and Statistical Analysis - *CC BY 4.0*
- 10.5: Exercise- Simplifying the Statistical Analysis - *CC BY 4.0*
- 10.6: Exercise- Peak Amplitude - *CC BY 4.0*
- 10.7: Exercise- Peak Latency - *CC BY 4.0*
- 10.8: Exercise- Fractional Area Latency - *CC BY 4.0*
- 10.9: Exercise- Quantifying Onset Latency - *CC BY 4.0*
- 10.10: Exercise- Collapsing Across Channels and Correlating Latencies with Response Times - *CC BY 4.0*
- 10.11: Matlab Scripts For This Chapter - *CC BY 4.0*
- 10.12: Key Takeaways and References - *CC BY 4.0*

◦ 11: EEGLAB and ERPLAB Scripting - *CC BY 4.0*
- 11.1: Data for This Chapter - *CC BY 4.0*
- 11.2: Expected Background Knowledge - *CC BY 4.0*
- 11.3: Bugs as an Opportunity for Growth - *CC BY 4.0*
- 11.4: Design of the N170 Experiment - *CC BY 4.0*
- 11.5: Exercise- The Matlab Command Line and the EEG Variable - *CC BY 4.0*
- 11.6: Exercise- The ALLEEG Variable and Redrawing the GUI - *CC BY 4.0*
- 11.7: Exercise- EEG.history and eegh - *CC BY 4.0*
- 11.8: Exercise- From the Command Line to a Script - *CC BY 4.0*
- 11.9: Exercise- Using a Variable for the Path - *CC BY 4.0*
- 11.10: Exercise- Loops - *CC BY 4.0*
- 11.11: Exercise- Looping Through Data from Multiple Participants - *CC BY 4.0*
- 11.12: Rapid Cycling Between Coding and Testing - *CC BY 4.0*
- 11.13: Exercise- Referencing with a Script - *CC BY 4.0*
- 11.14: Exercise- Improving the Referencing Script - *CC BY 4.0*
- 11.15: Exercise- Preprocessing the EEG and Using a Spreadsheet to Store Subject-Specific Information - *CC BY 4.0*
- 11.16: Exercise- Building an Entire EEG Processing Pipeline - *CC BY 4.0*
- 11.17: Exercise- Averaging with a Custom aSME Time Window - *CC BY 4.0*
- 11.18: Exercise- Scoring Amplitudes and Latencies and Performing Statistical Analyses - *CC BY 4.0*
- 11.19: Key Takeaways and References - *CC BY 4.0*

◦ 12: Appendix 1: A Very Brief Introduction to EEG and ERPs - *CC BY 4.0*

◦ 13: Appendix 2: Troubleshooting Guide - *CC BY 4.0*
- 13.1: A2.1 The First Step - *CC BY 4.0*
- 13.2: A2.2 Some Basic Solutions - *CC BY 4.0*
- 13.3: A2.3 Taking a Scientific Approach - *CC BY 4.0*
- 13.4: A2.4 Deciphering Matlab's Error Messages - *CC BY 4.0*
- 13.5: A2.5 Debugging Scripts by Performing Experiments and Collecting Data - *CC BY 4.0*
- 13.6: A2.6 Avoiding Bugs in Your Scripts with Good Programming Practices - *CC BY 4.0*
- 13.7: A2.7 References - *CC BY 4.0*

◦ 14: Appendix 3: Example Processing Pipeline - *CC BY 4.0*

◦ Back Matter - *CC BY 4.0*
- Index - *CC BY 4.0*
- Glossary - *CC BY 4.0*
- Detailed Licensing - *Undeclared*

This page titled Applied Event-Related Potential Data Analysis (Luck) is shared under a CC BY 4.0 license and was authored, remixed, and/or curated by Steven J Luck directly on the LibreTexts platform.