Research in Practice:
A Methods Handbook for Research in the Psychological Sciences

Paul D. Kieffaber, Ph.D.

College of William & Mary

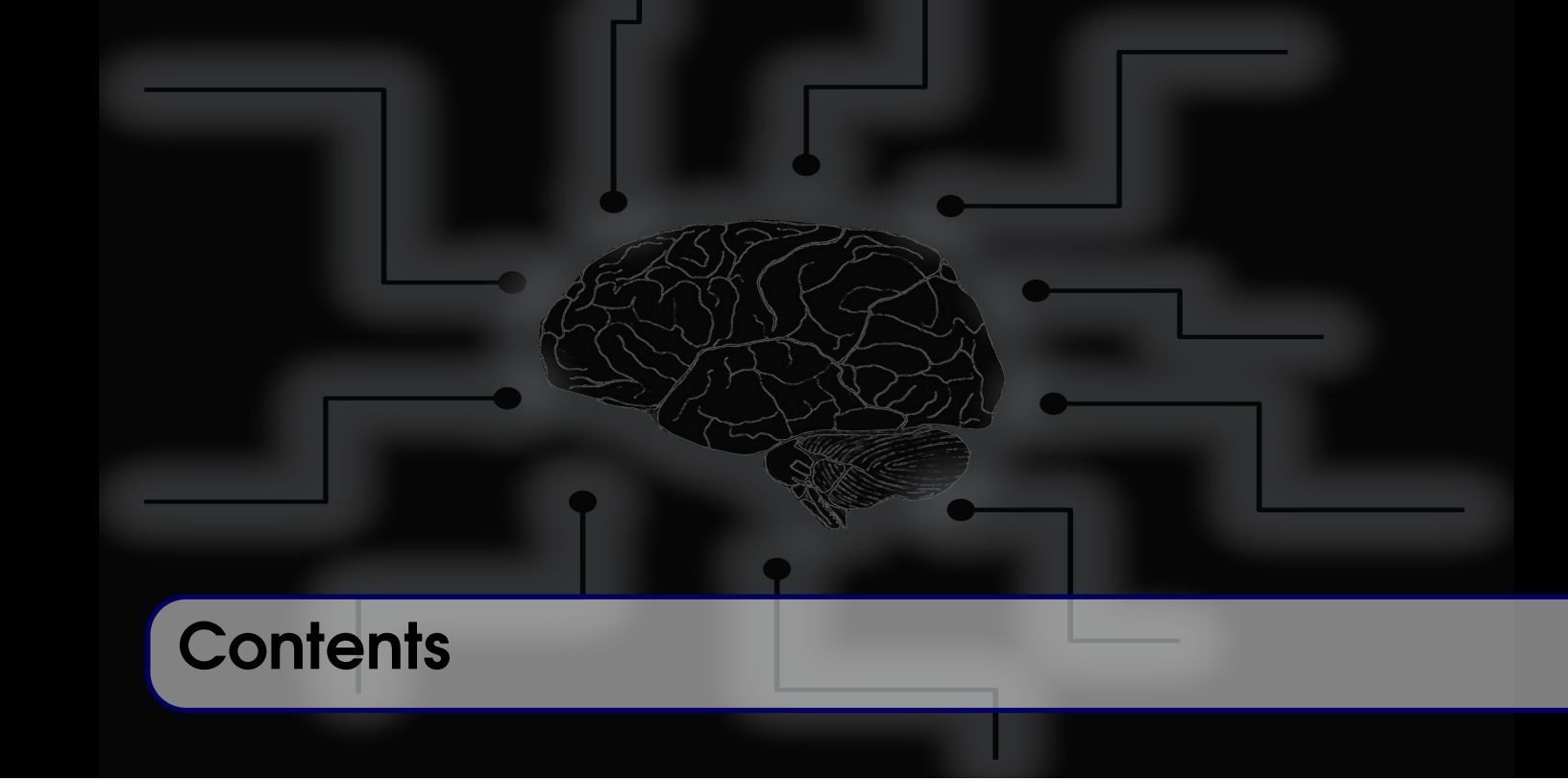
PUBLISHED BY PUBLISHER

HTTPS://WMPEOPLE.WM.EDU/SITE/PAGE/PDKIEFFABER/HOME

First printing, March 2019

Author Credits

- Sensation & Perception - Parts of the chapter were adapted by material originally developed with support from the William & Flora Hewlett Foundation, Bill & Melinda Gates Foundation, Michelson 20MM Foundation, Maxfield Foundation, Open Society Foundations, and Rice University. Powered by OpenStax CNX.
- Hello



Contents

1. Methods for Data Collection

I do not fear computers. I fear lack of them.

— Isaac Asimov

Contents

1.1 Software for Generating Experiments

The advent of desktop-computers forever changed the landscape of research in nearly all scientific fields. For psychological science, desktop computers have made possible new ways of probing the cognitive, emotional, and social functions of the brain. In addition to software that facilitates data management and analysis, computers can be used for flexible and precise control over many aspects (e.g.., stimuli and responses) of experimental procedures. It is their central role in the scientific process, from experimental design, to data collection, to data analysis, to typesetting a formal report of the results that makes a broad range of computing skills so critical to success in the psychological sciences. Of course, each aspect of research requires its own set of computing competencies. Here, emphasis will be placed on the design of experiments and collection of data.

1.1.1 Choosing the Right Software

Commercial Vs. Open Source

As with any task you wish to complete on a computer, there are several commercial and **open source software** options available. In the case of generating psychological experiments, all of the available options provide methods for displaying text, images, sound, and/or video as well as methods for recording responses using the keyboard, mouse, or other peripheral device like a specialized button box. The authors/distributors of these software packages also claim to provide accurate control of experiment timing, usually to within a millisecond.

Both the open source and the commercial options have their strong advocates. One argument is that the open-source method of developing software is far superior to commercial methods. Users of open source software have access to the source code, meaning they can enhance the program's performance, add features, and fix errors. In fast-paced scientific fields, this can mean that open source software will likely be the first to implement the latest and greatest methods. In the event of the establishment of novel methods, those who develop commercial software (a.k.a. "closed source software") must employ their programming team to implement the method within their software and run exhaustive testing in order to be able to provide assurance of accuracy/reliability to their customers before releasing a software update.

The most staunch proponents of commercial software often argue that this assurance of accuracy and reliability is precisely what makes it preferable to open source software. Another advantage of commercial software is the product support that is offered along with purchase of the software. If you download software like PsychoPy and receive an error when you attempt to install it, or are unable get it working properly after installation, your only option is to post your question to one of the online discussion boards dedicated to PsychoPy, and hope for a response. On the other hand, if you purchase a license to use E-prime©and have trouble installing the software and/or running an experiment, you can call the company and speak to an employee who's job it is to help you solve your specific problem. Ultimately, there are costs and benefits of both commercial and open source software and selection of a specific program should, most importantly, be based on how well that software will facilitate your research agenda.

Flexibility Vs. Ease of Use

Experiment control software also varies considerably on dimensions of flexibility and ease of use. Unfortunately, these two important dimensions tend to be inversely related to one another (see Figure 1.1 in almost any programming context. The reason for this inverse relationship is computers only do *exactly* what you tell them to do. When it comes to designing an experiment, instructing the computer about the many related details such as what image should appear on the monitor, where it should appear, how long it should remain on the monitor, whether or not to record a response from participants in your experiment all can be quite complicated.

Software that is easy to use tends be more "point & click", meaning that the software's author(s) have, in order to simplify the process of creating an experiment, reduced the number of instructions you are required to give to the computer by implementing a number of assumptions about how the experiment should run. Although these assumptions effectively relieve everyday users of the burden of having to understand much of the underlying programming language, this ease of use often comes at the cost of flexibility. Finding a good balance of flexibility and ease of use is important so that novel and/or dynamic research needs can quickly be addressed but implementation of those needs does not require complete mastery of a programming language. While a complete review of the available software is beyond the scope of this chapter, software reviews do occasionally appear in the literature (e.g., **?**).

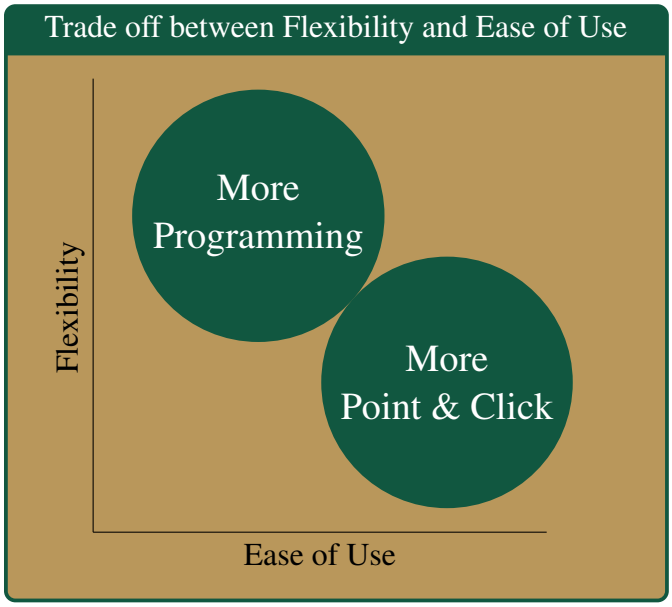


Figure 1.1: Illustration of the trade-off between flexibility and ease of use in experiment control software

Most commercial software is available through stand-alone packages, meaning that once the software has been purchased, there are no other dependencies or packages required other than those that are already installed on your computer. Some examples of commercial software are E-prime©, SuperLab, Presentation, and DirectRT, just to name a few. It can be quite difficult to draw direct comparisons between these different options because each one has a unique set of strengths and weaknesses and are typically designed to meet the needs of users in specific research contexts. For example, two historically popular choices are E-Prime and SuperLab, each of which offers an easy to use point & click style interface with which simple experiments can be generated in just a few minutes. Another popular commercial option is Presentation. Although Presentation does not have the same kind of graphic user interface (GUI) as E-prime©or Presentation, the language includes a powerful set of commands for designating experimental components (e.g., images, sounds, responses etc.) and their timing (e.g., onset, offset, duration, etc.).

A popular open-source option for the generation of experiments is Psychtoolbox. With years of development and a broad base of users, Psychtoolbox boasts a wide variety of powerful, well-developed routines for experimental control. However, Psychtoolbox is written in the Matlab programming language, meaning that using it requires that you have a licensed installation of Matlab. Thus, while Psychtoolbox is freely available, there is an indirect, prerequisite cost of obtaining Matlab[1]. Like Presentation, Psychtoolbox lacks a GUI so even simple experiments require at least some experience with the Matlab programming language, making it difficult for new users without much programming experience.

Another open source option is PsychoPy. Written in the Python programming language, itself a free alternative to programs like Matlab, PsychoPy offers an attractive balance of flexibility and ease-of-use. PsychoPy has the full flexibility of any general purpose programming language, but also includes a user-friendly GUI with point & click features that allow simple experiments to be generated with little to no knowledge of the Python programming language.

[1]Note that the latest version of the Psychtoolobx is also compatible with GNU Octave, a free alternative to Matlab

Online Data Collection

The ability to collect data online using experiment control software is a feature that has recently become especially popular. In fact, most commercial *and* open source software offers some kind of online option for data collection, but it is important to understand at least two of the central issues related to collecting data online, including (1) the type of data being collected and (2) how will the data be stored.

The type of data being collected in an online experiment is an important consideration because some measures of behavior and/or parameters of a stimulus display require a degree of temporal precision that may be difficult to achieve in practice. For example, an accurate measure of reaction time requires (1) precise measurement of the time at which a stimulus is presented to the participant (i.e., the stimulus to which participant is reacting) and (2) an accurate measure of the time at which a key or button was pressed. These may seem trivial, but while computers only do exactly what they are told, they don't always do it exactly when they are told.

Remember that modern computers are serial processors, managing many different tasks such as checking your email inbox, running virus protection software, monitoring internet traffic, etc., all while also running your experiment. Each of these tasks uses some of a limited set of resources on the computer. When the program you are using instructs the computer to display an image or play a sound, the expectation is that it will happen immediately, but this is almost never the case. Instead, the time between the instruction to present a stimulus and the appearance of that stimulus to the participant depends on a variety of factors like the amount of available resources at that moment, the priority assigned to your experimental task by the computer's CPU relative to other ongoing computer functions, characteristics of the computer's video and/or audio cards, and other computer hardware and software settings. There is also uncertainty in the measurement of key/button press responses. Computers only "check" the status (i.e., up/down) of keys and buttons at designated intervals; typically 125 times per second or every 8 milliseconds. This **polling rate** can vary from computer to computer, from device to device, and can also be influenced by the resources currently available to the computer's CPU. In short, most of this uncertainty about the precision of stimulus and response timing contributes to random **measurement error** in the assessment of reaction times, however, systematic measurement errors can also occur when hardware or software introduces consistent differences between the measured and true reaction times.

Error in online RT Measurement

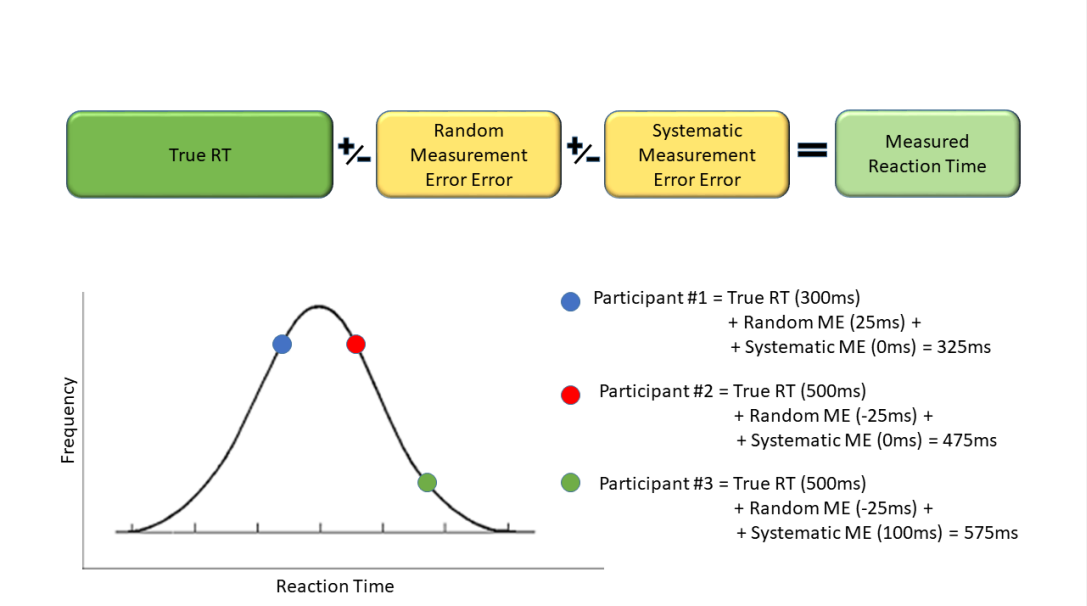


Figure 1.2: Illustration of Measurement Error in RT that can occur with online data collection.

All of the concerns just mentioned are compounded with the recording of reaction times over the internet. This is because both random and systematic measurement error can be expected to be relatively consistent from person to person when all of the participants complete your experimental task on a computer in your laboratory. When the participants each use their own computer, the degree to which random and/or systematic measurement errors influence the recorded reaction times will, in addition to being unknown, fluctuate from person to person. This potential increase in error variability between subjects can mean that more participants will be required for online research involving reaction times and/or precise control over stimulus timing.

Food For Thought

How might online data collection involving the collection of reaction times potentially introduce a confounding variable to your research?

When each participant uses their own computers, each with different components and specifications that contribute to error in the accuracy of RT measures, there is an increase in both random and systematic measurement error between subjects. This can be particularly problematic for studies involving between-groups designs because there is the possibility that systematic differences in reaction time may exist because of systematic differences between the quality of the computers used by the two groups.

It is also important to consider how the data will be stored in an online experiment. Here data storage refers to the how and where data will be stored as the participant is responding to the items on your questionnaire or stimuli in your experiment rather than how/where the compiled data will be stored after the experiment has ended and data analysis begins. In order to understand this distinction, it is important to have a basic understanding about what happens when you send a participant the link to your questionnaire or experiment.

There are many facets of hosting a web page that collects data from participants that we don't really need to get into here, but the important thing is that when your participants visit a website, a series of things are happening:

1. You set up your web page using your favorite program and send the link to your participants.
2. Participants use their web browser to enter the link, which sends a request to the website's server for a page, a document, or a file for running an application. The **URL** or address you put into the bar at the top of the browser window is the main portion of that request.
3. The web server receives the request and pulls together whatever it needs to deliver back to the participant what they requested. This might just be an existing file, or it might be a piece of a web application, or it might be an assembled document from a content management system.
4. The participant's browser application shows them the content and communicates back to the server about the participant's responses.

So, in order to host a website for data collection, you need a computer running server software capable of managing requests from many simultaneous participants and that is *always* connected to the internet so it can respond to requests and store data at any time. Of course, this isn't the kind of thing that your standard desktop computer is capable of doing, so you will need a website server, which can be expensive and difficult to maintain. Thus, the most common approach to online data collection is to subscribe to a **web hosting service**, which provides all of the services above for a fee.

Although online surveys for data collection have been readily available and used for decades (e.g., Qualtrics©), options for developing online experiments that involve presenting participants with controlled stimuli and recording responses like reaction time and accuracy have only recently become available. Some commercial software applications (e.g., E-prime©) are beginning to offer web hosting services as part of their software license. Developers of the open source software PsychoPy have also developed an online platform called Pavlovia that is capable of hosting online experiments created using PsychoPy.

Using Pavlovia, only requires the purchase of “credits" that are redeemed as participants connect to and complete your experiment(s). Yet another, completely free alternative is PsyToolkit (**??**). PsyToolkit is a free-to-use toolkit for demonstrating, programming, and running cognitive-psychological experiments and surveys, including personality tests. PsyToolkit is frequently used for academic studies, for student projects, and for teaching cognitive and personality psychology. An advantage of PsyToolkit is that it works in most web browsers and so is completely **platform independent**. Another advantage is that surveys and experiments can be created *and* hosted on their website for free. However, remember the trade off between flexibility and ease of use described earlier. A flexible platform for data collection, PsyToolkit requires learning its scripting language, which can be challenging for those with little or no prior programming experience.

Whichever software or platform you choose for your specific research needs, time and effort will be required to learn how to use it when designing your data collection protocol. Tutorials for several open source software packages are provided in the following sections, however, before tackling the task of learning to program an experiment, let’s consider the research context for these tutorials.

1.2 Research in Practice

Today, even most children in the 1st grade would likely answer correctly when asked whether the brain is the source of human thought (the answer is yes!). However, it is important to remember that it was only as recent as the latter part of the nineteenth century that this became consensus in the scientific community. Medical observations like that of Phineas Gage (1823-1860), an American railroad construction worker who survived an explosion that drove a large iron rod completely through his head, fueled that understanding and helped to shape modern thinking about brain-behavior relationships. Although the accident destroyed much of his left frontal lobe, the most profound changes following Mr. Gage’s accident were with regard to his personality and behavior. In other words, the damage Mr. Gage suffered to his brain seemed to have a direct effect on complex thought. This was important because there was considerable debate at the time as to whether this kind of “thought” was something material or non-material (e.g., “mental", or “spiritual") and observations of Mr. Gage’s experience with brain damage supported the notion that complex thought, like other aspects of behavior, had a material basis in the central nervous system.

Interestingly, at about this same time in the middle part of the 1860’s, Franciscus Cornelis Donders, a Dutch physiologist, conceived of a way to measure “thinking” in the laboratory. His novel idea was that if thought is material, then it may be measurable in terms of the time taken to generate thoughts. But how does one measure the time taken to think (especially in the 1860’s!)? To show that thinking takes time, Donders used an elegantly simple subtractive logic. He began by measuring the time required to execute an action (e.g., press a button) following perception of a stimulus (e.g., a light; sometimes called the “simple" reaction time), reasoning that this response time reflected the sum of the times taken to detect the stimulus and execute the response. Donders then used this simple reaction time as a baseline for comparison with tasks that involved more “thinking". Provided that the stimuli and responses are the same as those used to measure simple reaction time, any additional time taken to complete a task involving more “thinking" must be attributable to the time it takes to produce that “thinking".

Fundamental to Donder’s logic was the idea that mental processes can be described as stage-like “thought" units. Until Donder’s work, many assumed that mental operations likely occurred in parallel and instantaneously. One of Donder’s primary goals was to elucidate the timing of mental operations using his simple subtraction technique. In his now famous 1868 research, Donders used three different tasks: (1) The first task was a simple reaction time task wherein participants responded as quickly as possible upon detecting a stimulus. (2) The second task was a so-called Go/No-Go discrimination reaction time task in

which participants were instructed to press a button only when a target (i.e., Go) shape appeared, but to withhold their response if a different, non-target (i.e., No-Go) shape appeared. (3) Finally, the third task was a choice reaction time task in which there was more than one target and participants were instructed to make different responses to each target (see Figure 1.3).

Donders Reaction Time Tasks

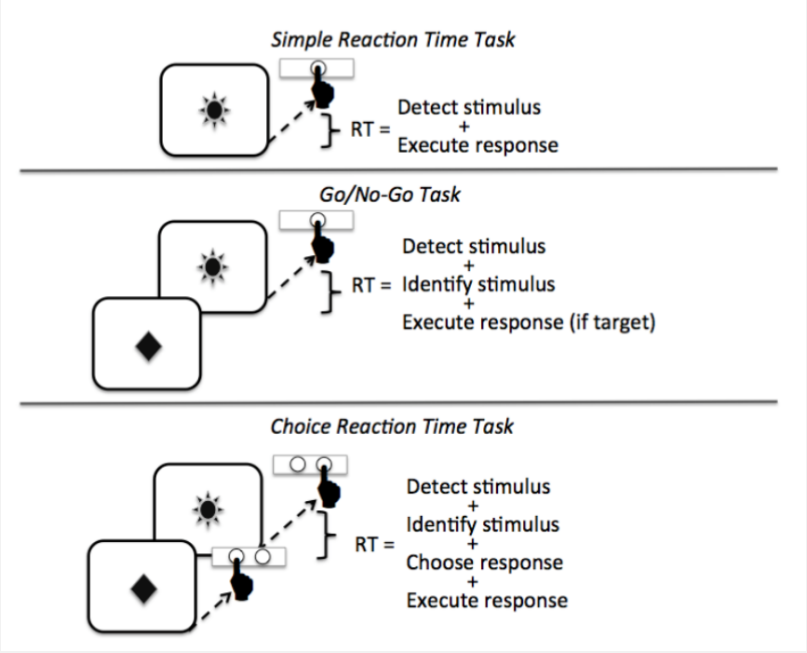


Figure 1.3: Illustration of the three Donders reaction time tasks and the subtractive logic used to infer the timing of mental events.

Donder's hypothesized that four kinds of cognitive processes were involved in these tasks: (1) The simple reaction time task would require just two of these processes, perception (detecting the light) and action (pressing the button). (2) The Go/No-Go discrimination task would require the same perception and action processes as in the simple reaction time task, plus a discrimination process in which the participant must determine whether the stimulus was a target (i.e., Go) or non-target (i.e., No-Go). (3) Finally, the choice reaction time task was reasoned to require all of the same processes required for the simple and discrimination reaction time tasks, plus a response selection process in which the participant must decide which response to execute.

As you might expect, the simple reaction time task is typically associated with the fastest reaction times, followed by the discrimination (e.g., Go/NoGo) task, followed by the choice reaction time task. Donders then proposed that the time required for each mental operation (i.e., thought) could be calculated by simple subtraction. For example, the time it takes to identify a stimulus could be calculated by subtracting the simple reaction time (i.e., perception + action) from the RT in the Go/No-Go task (i.e., perception + identify stimulus + action). While this technique has not been used without criticism, this basic procedure for isolating component cognitive processes remains the basis of the logic behind much of the research in the Psychological Sciences today.

Although Donders did not have access to computers (In fact, he had to develop new technology to make precise measurements of reaction times), this kind of experimental procedure can be easily replicated today using software like PsyToolkit, PsychoPy, and/or PsychToolbox.

1.3 PsyToolkit

Developed in the mid 2000s, PsyToolkit is a free platform for programming and running psychological experiments that runs within web browsers using Javascript and HTML5 technology to achieve performance that is comparable to commercial software like ePrime (**?**). Although used primarily as an online platform for data collection, it can also be used offline as an in-laboratory software.

Unlike other popular programs, PsyToolkit is entirely script-based programming and there is no point-and-click graphical user interface for setting up an experiment. As was mentioned earlier in this chapter, this has advantages and disadvantages. The disadvantage is that it can take more time to learn. The advantage is that it can be much more flexible and, once mastered, faster to program simple experiments than in other similar software. This tutorial will introduce you to the basics of creating surveys and experiments using PsyToolkit. Further details and answers to FAQs can be found on their website at https://www.psytoolkit.org/.

1.3.1 Getting Started with PsyToolkit

1. Go to: https://www.psytoolkit.org/
2. Click **Web based / login** and then click the link to register for a free web-based account.
3. A password will be sent to you via email
4. Use your email and password to sign in to begin creating your first experiment! (see Figure 1.4

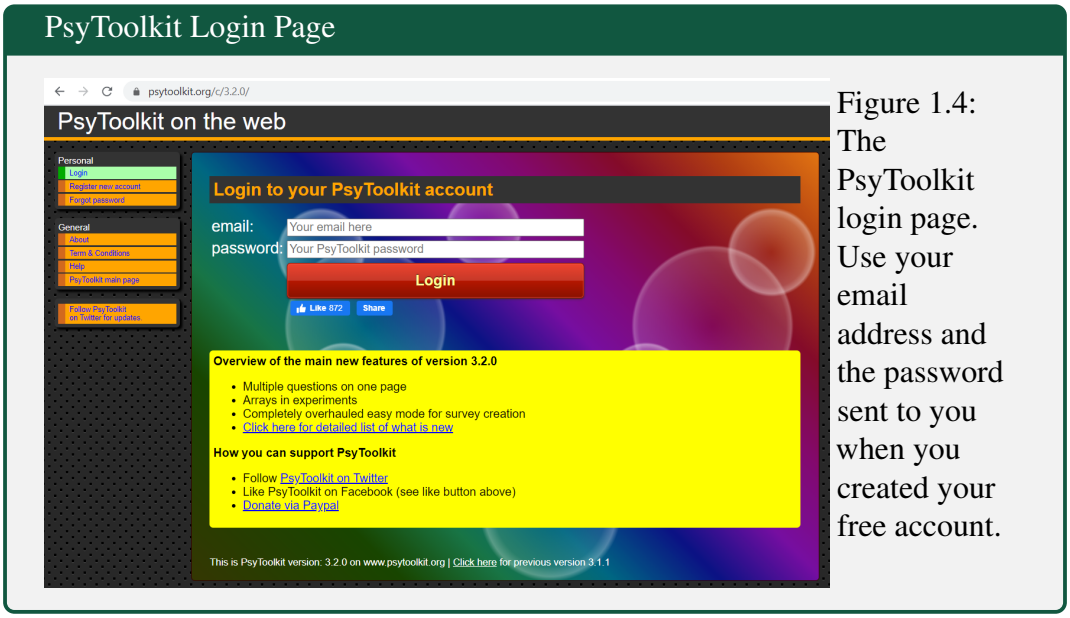


Figure 1.4: The PsyToolkit login page. Use your email address and the password sent to you when you created your free account.

1.3.2 Create Your First Survey

Once you have logged into your PsyToolkit account, you will be taken to your home page where you can see announcements from the developers, tips about using PsyToolkit, and links (in the frame on the left) to general settings and tools for creating and editing experiments and surveys (see Figure 1.5). In PsyToolkit, experiments are deployed from within surveys, so we will start this tutorial by creating a survey and later show how to add an experiment.

PsyToolkit Home Page

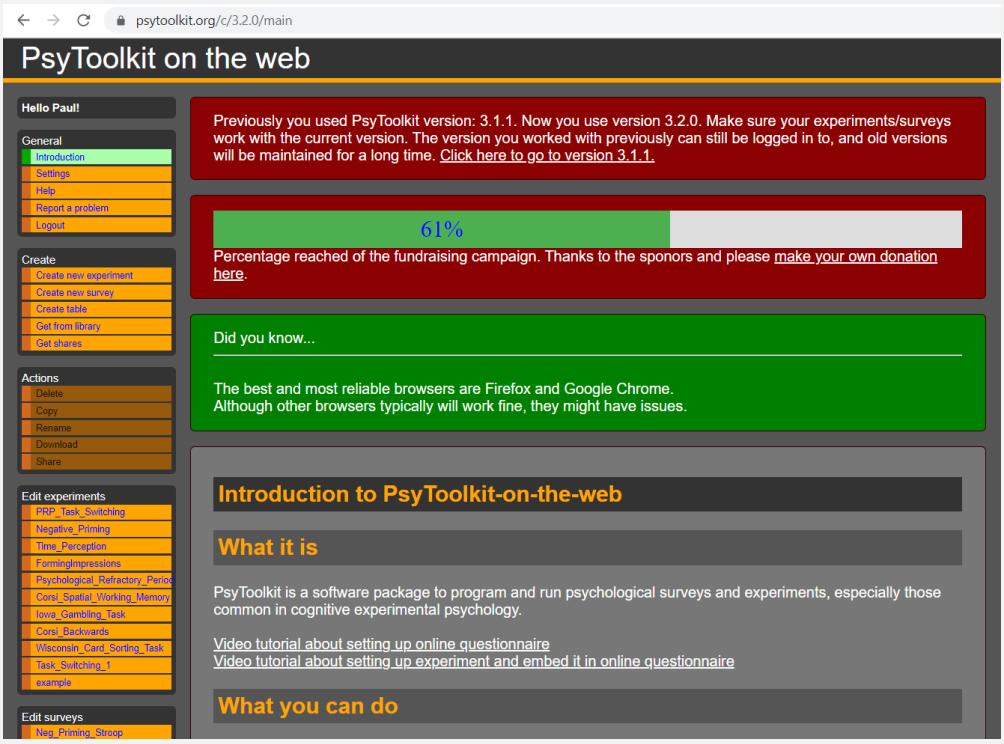


Figure 1.5: The PsyToolkit user's home page with access to personal and public libraries of surveys and experiments.

You can start creating your first survey right away by following these two simple steps:

1. Click "Create new survey"
2. Enter a name for your survey and click "Create completely new online survey"

Following these two steps will open the Survey Editor where you can create your survey. By default, new surveys include several sample items (see Figure 1.6).

PsyToolkit Home Page

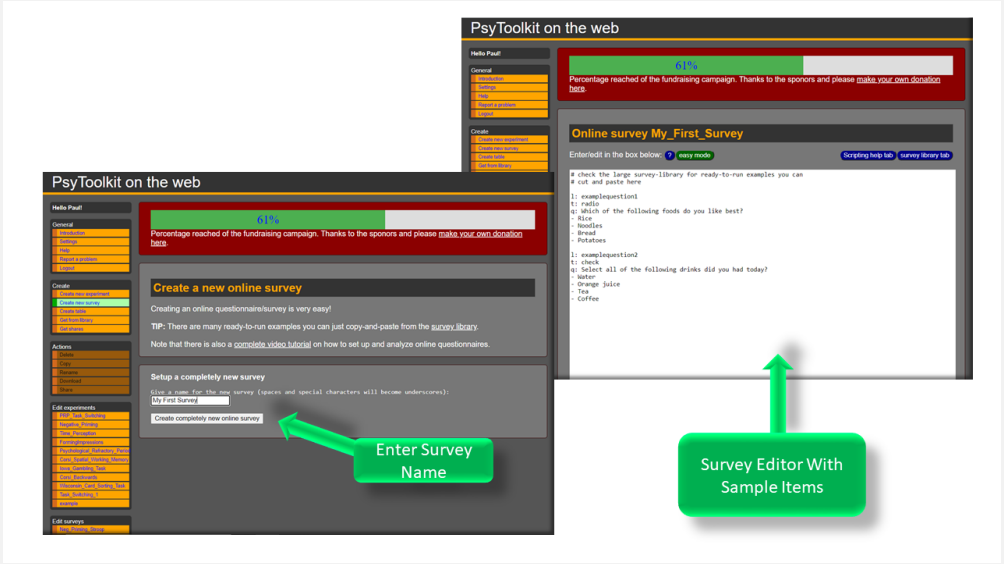


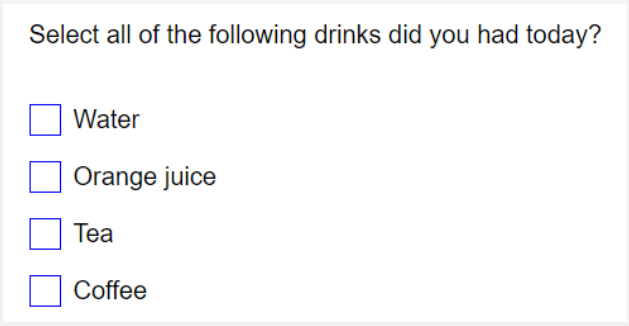
Figure 1.6: Creating an Online Survey.

Each item of a survey can be created with just a few lines of text, however, there are strict rules

governing the syntax of that text. Each item is defined by a set of “label", “type", and “question" parameters and, in some cases, a list of options defining the response(s) available to the participant. Each of the parameters that define the survey item is designated by an `l` (label), a `t` (type), or `q` (question), followed bye a colon and then the parameter definition (see 1). Some item types also include an “options" parameter. Several examples of survey items are illustrated below. For complete details about the structure of surveys and items, refer to the Online Surveys tutorial page.

PsyToolkit - Checkbox Example

```
l: examplequestion2
t: check
q: Select all of the following drinks did you had today?
- Water
- Orange juice
- Tea
- Coffee
```



Code block 1.1 PsyToolkit code (top) and preview (bottom) for creating a simple question with checkbox options.

Of course, when you are just starting out with PsyToolkit, the process of creating an entire survey with many different question types can seem a bit daunting. Fortunately, PsyToolkit also offers an “Easy" mode that can be especially helpful for beginners. To enter easy mode, simply click the button labeled “easy mode" just above the survey editor (see Figure 1.7).

When in easy mode, the items in your survey are listed in a convenient table that includes links to enable direct editing of each item. New items can be added by clicking the + button. When you add a new item, you will first be presented with a list of available item types. The decisions you make about the types of items you include will be based on the kind of data you wish to collect and the parameters you want to impose on the participant’s responses. For example, most surveys will include an item requesting the participant to indicate their gender. One possibility would be to use a text line where the participant can enter whatever response they like using the keyboard. In this case, it will be up to you as the researcher to comb through the responses and group similar responses like “male", “Male", “M", “boy", or maybe even “mail". This could be a tedious and error-prone process, so a more appropriate choice when requesting the participant’s gender might be to use the radio button type item with options for “Male" and "Female", which would simplify data entry for the participant *and* data analysis for the researcher. In short, you should carefully consider how you select items for your survey. You can add as many items to your survey as you like.

Of course, items are the most important part of your survey as they determine the kinds of information/data that are collected. However, whenever collecting data from human participants, it is important to provide them with enough information about the study that they can make an informed decision about whether or not they would like to participate *before* you start collecting data. This kind of information that is required for **informed consent** is also designated in the survey design page using the text fields available in the "Survey intro screen and special options" section of the PsyToolkit survey generation page.

PsyToolkit Easy Mode

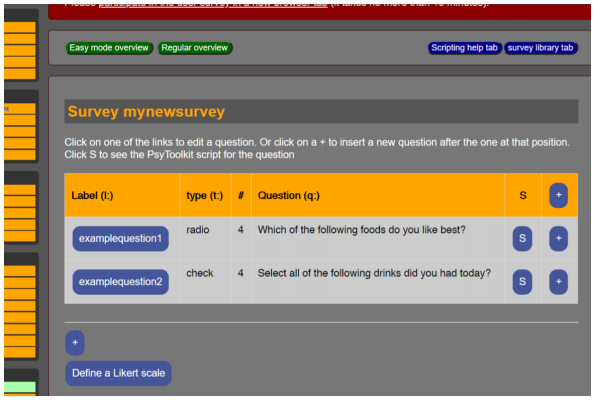


Figure 1.7. Adding survey items is as simple as a few clicks using "Easy Mode".

If you are still in easy mode, you will first need to return to the regular mode by clicking the regular overview button (see Figure 1.7). Next, scroll down to the "Survey intro screen and special options" section. There you will find options governing the look and feel of your survey, an "About this survey" section where you can enter information to be provided to the participant *before* they begin the survey, an expandable "Read more" section for additional information, and a "Contact information" section where you can provide the participant with contact information for the **principal investigator** (PI). Once you have entered all of this information (*don't forget to click the Click to save button at the bottom!*), you are ready to compile and test your survey.

Test Your Survey

Before you deploy your survey you will, of course, want to test it out. This requires that you first **compile** your survey for execution. If you are still in easy mode, you will first need to return to the regular mode by clicking the "regular overview" button (see Figure 1.7) and take the following steps:

1. Scroll down to the "Compile/test survey" section and click the compile button (see Figure 1.8).

Compile/test your survey

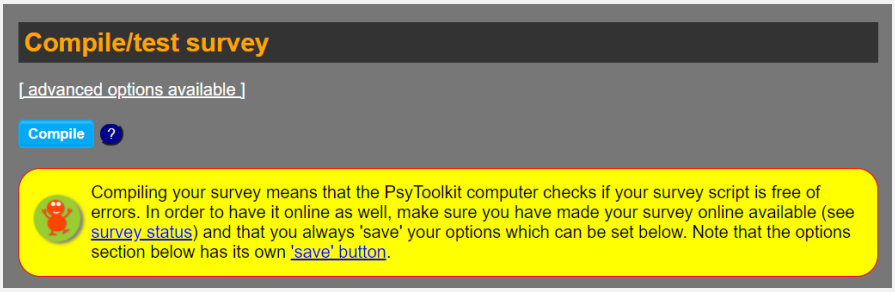


Figure 1.8: Use the "Compile" button to compile and test your survey. You will get a message letting you know if there were errors or if it compiled correctly.

Note that if your survey does not compile correctly PsyToolkit is designed to give detailed feedback of how to fix errors. PsyToolkit will tell you which line of your script file the problem occurs in. If you are having trouble getting your survey (or experiment) to compile, you may want to check the Problem solving & Tips page at `https://www.psytoolkit.org/software/problemsolving.html`.

2. Once your survey has compiled successfully, scroll down to the “Online/Offline survey status" section and select the option, “Survey is online. For when you are still designing it." and then click the Change survey status button.

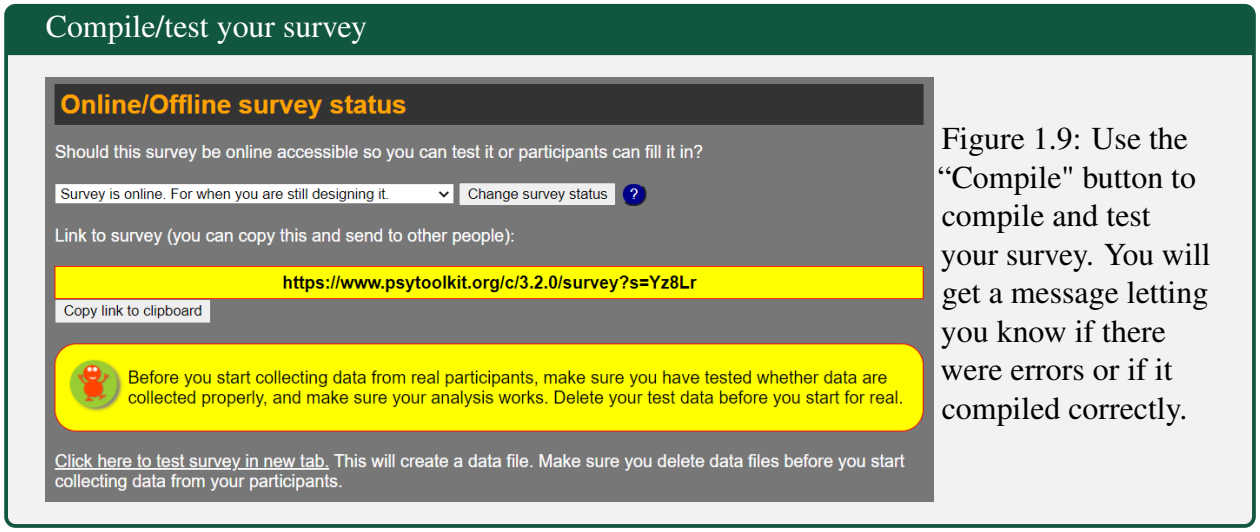


Figure 1.9: Use the “Compile" button to compile and test your survey. You will get a message letting you know if there were errors or if it compiled correctly.

3. When the status has been changed, you will be provided with a **URL** that can be copied into the address bar of your browser which will take you to the online survey (see Figure 1.9).

When you follow the link to your survey, you will first see all of the information provided in the “About this survey" section. Once the button is clicked to start the survey, each of the items is presented one-by-one until all of the items have been answered (see Figure

Check the Survey Data

All responses are automatically saved any time the survey is completed. In order to view and/or download the data, simply scroll to the bottom of the page used to edit your survey in PsyToolkit and find the “Prepare and download participant data" section. Click the Prepare datafiles for download button. Soon thereafter (after a couple of refreshes), a download data in zip file button will appear. Click that button to download your data (see Figure 1.10).

The data files will be compressed into a single .zip file. When you find the downloaded file, uncompress (i.e., unzip) it to your favorite location on your computer. The unzipped folder will be called “data" by default. Inside, you will find a number of files, including a copy of the original survey script file in “survey.txt", individual text files containing raw data from each of the participants separately, the participant’s **aggregate data** in “data.csv" and “data.xlsx" (two formats are provided for convenience), and aggregate data about the times/dates when the surveys were completed in “data_times.csv" and “data_times.xlsx".

It may take some time to understand how your data are organized. Although each row of the tabular output is dedicated to one of the participants, each of the item types has a slightly different data storage format. For example, the “radio" items only allow for one response to be selected, so the result is stored in a single column (see Figure 1.11). The “check" item types allow users to select multiple items, so each of the available responses is listed as a column and a `0` or a `1` is used to indicate the absence/presence of a check-mark for that item (see Figure 1.11).

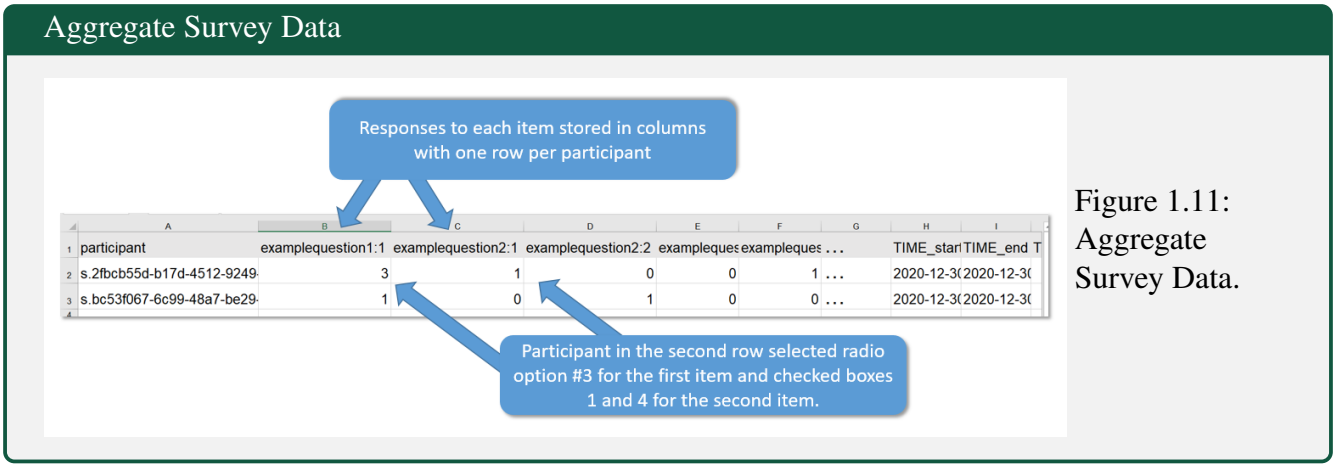


Figure 1.11: Aggregate Survey Data.

Deploy Your Survey

It is important that you inspect and understand the aggregated output from PsyToolkit *before* you deploy your survey to participants. Imagine that, only after putting all the time and effort into designing a survey, coding it in PsyToolkit, and collecting data from participants, you find out that a critical piece of information is missing from your survey or that the data you collected aren't appropriate for the analysis you had in mind. Although it can take some time, thorough testing can mean the difference between a successful and a failed research project. That said, once you have thoroughly tested your survey, it is a simple matter to deploy it for real data collection using the following steps.

1. First, it is usually a good idea to clear the data recorded while testing. Scroll to the bottom of your survey's page in PsyToolkit and click the delete participant data files button.
2. Next, scroll up to the "Online/Offline survey status" section and select the "Survey is online. For when real data collection is going on" option and then click the change survey status button.

   You will again be provided with a link to your active survey. This link can be shared with anyone you would like to complete your survey (or even shared publicly). The only difference now is that your survey cannot be modified until you again change its status in PsyToolkit.

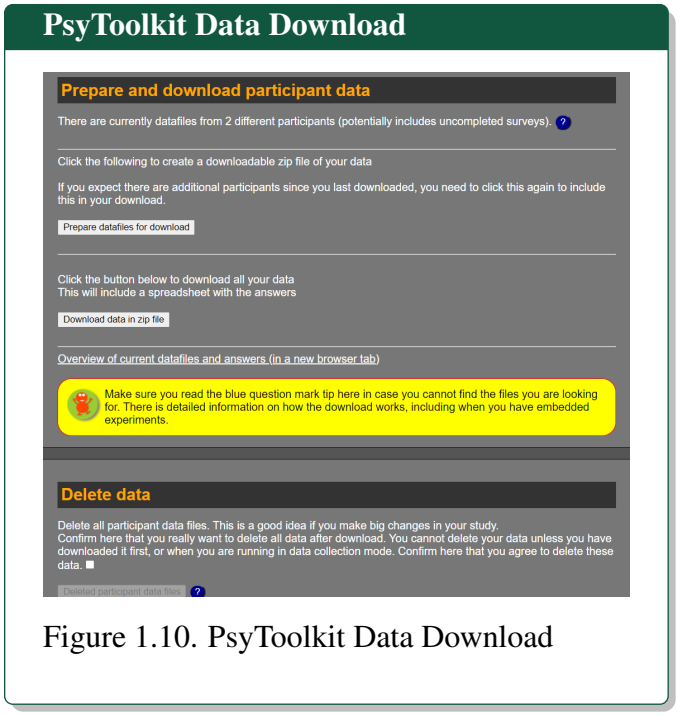


Figure 1.10. PsyToolkit Data Download

1.3.3 Create Your First Experiment

In the previous section, we covered how to create and deploy a simple survey using PsyToolkit. In the following sections we will see how to create

and deploy experiments. While both survey and experiment methods are important in data collection and both can be used to test hypotheses, experimental research involves the manipulation of an independent variable and measuring its effect on a dependent variable. In the context of research in the psychological sciences, the dependent variables are commonly measures like reaction time, accuracy, and number of items remembered while the independent variables are typically physical properties of the experimental stimuli like color, shape, emotional valence, and duration of exposure. In the following tutorial we will cover the steps necessary to create a simple reaction time task using PsyToolkit. More advanced examples and detailed information can be found in the online documentation at https://www.psytoolkit.org/doc3.2.0/.

Before you get started programming your task, it is always a good idea to lay out a specific plan. In this example, the aim is to recreate the simple reaction time task from the Donders experiments described earlier. Thus, the experiment will minimally consist of the following:

- Presentation of a fixation point (used to direct the attention of the participant)
- Presentation of a target stimulus
- Detecting and recording response times

Identifying all of the major components of a task can be made easier by developing a **task schematic** (see Figure 1.12). The task schematic will often illustrate the experimental stimuli, highlight the major parameters of an experimental task such as the time between stimuli, called the **inter-stimulus interval** (ISI), and show how responses were collected from participants.

Working from the task schematic in Figure 1.12, we can see that programming this experiment will require at least two visual stimuli that will be presented to the participant, the fixation cross and the target stimulus (a diamond shape in this case) and the collection of a response from the keyboard. We will also need to define an ISI and and an **inter-trial interval** (ITI; not shown in the schematic) which will determine how much time should elapse between the response to one stimulus and the appearance of the fixation cross indicating the impending appearance of the next stimulus.

Food For Thought

Even creating a simple survey requires creativity and critical thinking!

When a researcher creates an instrument for their research like a survey or experiment, there are many different decisions that must be made along the way. For example, if you decide to ask your participants to indicate their gender, do you use male/female radio buttons, a text box that will permit a free response, or a drop-down menu. These decisions will impact the experience of participants as well as the format of the recorded data, which may later impact the researcher's approach to data analysis. In short, survey and experiment design are important skills for any psychological scientist.

Example of a Task Schematic

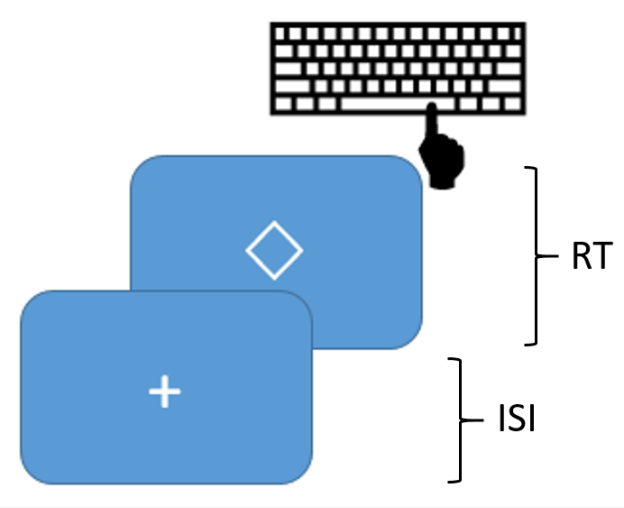


Figure 1.12: The task schematic makes explicit the parameters of an experiment and can serve as a planning tool during programming *and* as a visual aid for readers at the time of publication.

Creating a task schematic before you begin programming an experiment can be a useful in that is provides a convenient visual reference when determining how to program your experiment and it can also serve as a visual aid for readers when the experiment is completed and you are presenting the results of your experiment. For now, the task schematic can be used to determine the stimuli that need to be created.

Creating the Stimuli

After having planned out your experiment but before beginning the work of programming that experiment in PsyToolkit, it is prudent to create the experimental stimuli (i.e., images and sounds) that will be presented to participants. For present purposes, we will focus on the two images that will be presented to participants in the simple reaction time task, namely the fixation cross and the diamond shape.

PsyToolkit works best with a particular type if image file called a bitmap that includes the file extension "`.bmp/.gif/.png/.jpeg, etc.`". Simple bitmap files can be created using many different applications, including some of those installed by default on most Windows and OSx operating systems. For more complex images and/or for better control over the parameters of those images, you will need to use a commercial image editor like Adobe Photoshop©or one of its free alternatives like the Gnu Image Manipulation Program (GIMP) or Inkscape. PsyToolkit's authors recommend the free image editing software called Inkscape. You can download your own copy of Inkscape for Windows or OSx at `https://inkscape.org/`.

Figure 1.13 illustrates how the two images were created in Inkscape for this tutorial. Note that while a blue background was used in the example task schematic, a black background was selected for the actual stimuli so that they could more easily match the default background screen color in a PsyToolkit experiment. Also note that although the two images are illustrated in a single window, they were saved as two separate files, "fixation.png" and "target.png". Once you have created the bitmaps for an experiment, these files must be uploaded to your experiment folder in PsyToolkit. Follow these steps to upload files to your experiment's page in PsyToolkit:

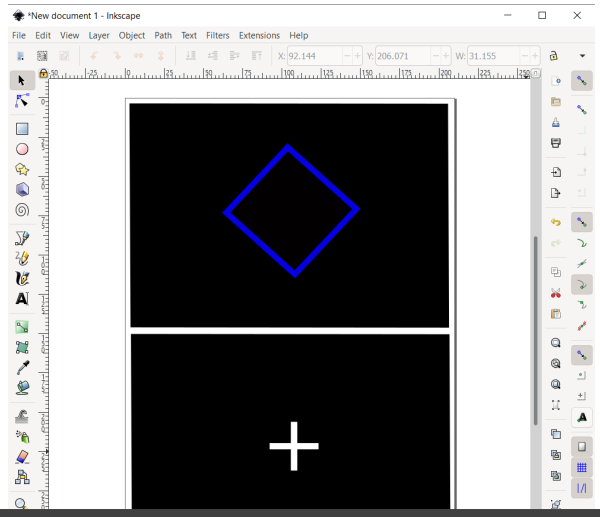
- Click the Choose files button
- Find and select the file(s) on your computer and click Open .
- Click the blue Save button in PsyToolkit to complete the upload.

Files uploaded to an experiment can be viewed, deleted, and renamed by clicking the view files button.

The Experiment Script

Now that you have both a clear plan in place for developing your experimental protocol and you have created the stimuli you will be using, you are ready to begin writing the program that will be compiled and run using PsyToolkit. This program must be written using an experiment script, which is a set of instructions that will determine all aspects of your program including which stimulus files to use, when to present them, what data to collect, etc. A minimal working example of a PsyToolkit experiment script is illustrated in the code block below. *Note that any text following the "#" is an explanatory comment and will NOT be executed when the program is compiled!*

Inkscape



The Simple RT Experiment Script

```
options
fullscreen                #run in full-screen mode

bitmaps
  fixation                #load fixation.png
  target                  #load target.png

task SimpleRTTask
  keys space              #listen to the "space" key
  delay 1500              #don't do anything for 1,500ms (ITI)
  show bitmap fixation    #show fixation.png
  delay 1000              #don't do anything for 1000ms (ISI)
  clear -1                #clear the last image shown
  show bitmap target      #show bitmap.png
  readkey 1 2000          #wait for spacebar press for no more than 2000ms
  set $rt RT              #set variable "rt" to value of RT
  save $rt                #save rt for this trial
  clear -1                #clear the last image shown

block SimpleRTBlock
  tasklist
  SimpleRTTask 50         #run the SimpleRTTask 50 times
  end
```

Code block 1.2 Minimal working example of a simple RT experiment script in PsyToolkit.

Notice that the experiment script is divided into four sections, including (1) `options`, (2) `bitmaps`,

(3) `task`, and (4) `block`. Not all of these sections are strictly required for the experiment to run and there are other named sections that are not necessary for example. Importantly, each of these sections helps to define a specific set of parameters that will govern the behavior of the experiment. For example, the "`fullscreen`" command in the `options` section tells PsyToolkit to run the experiment in full-screen mode on the participant's computer and the `bitmaps` section contains a list of names of the bitmap files that will be used.

The `task` section can be thought of as the list of events that will occur on each trial of the experiment. Recall that in the simple reaction time task, each trial consists of the presentation of a fixation cross, then the presentation of a diamond to which the participant is expected to respond by pressing a button. Of course, when writing a program to accomplish this, every last detail must be provided to PsyToolkit as a specific instruction. As with any programming language, and as you can see from code block 1.2, there is a specific syntax required to give these instructions.

Generally speaking, most instructions to PsyToolkit start with a keyword and are followed by one or more parameters. For example, the keyword "`keys`" instructs PsyToolkit to prepare to "listen" for input from the keyboard and the subsequent parameter "space" indicates that PsyToolkit need *only* listen to input from the spacebar. Similarly, the keyword "`delay`" instructs PsyToolkit to pause the experiment and the parameter that follows provides PsyToolkit with a time-limit for the pause in milliseconds (ms). In the present example, both the ITI and ISI are accomplished using the `delay` command. We will cover some of the other commands later in this tutorial later.

The final section of the experiment script is the `block` section. Most experiments are complex and require the designation of more than one `task` section in the experiment script. The block section is where you would instruct PsyToolkit about how and in which order it should present participants with each task. In the present example, we are using a very simple experiment with just a single `task` and so the `block` section simply sets up a task list with just one task (i.e., "SimpleRTTask") and

instructs PsyToolkit to repeat the simple reaction time task 50 times.

Testing and Deploying Your Experiment

Writing your own experiment script will almost always require several rounds of testing/troubleshooting before it is ready to be deployed. Before you can test your experiment, you will need to both save and compile your experiment script in PsyToolkit.

To start testing your experiment, first press the Save button just below the experiment script. Then, click the compile button in the next section, titled “Compile and run". If your experiment successfully compiles, you will see a message saying that it was successful and you will also see a Run experiment button that you can click to conduct a test-run of your experiment. If the script does not compile successfully, you will see an error message informing you about the nature of the error. Once that error has been addressed in the experiment script editor, you will need to re-save and re-compile the experiment script.

Compile and Run Section

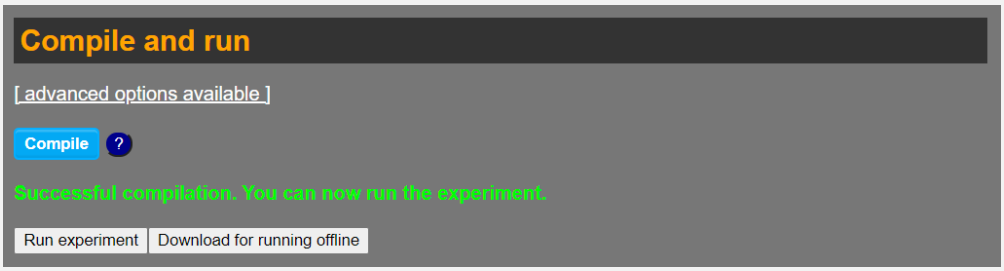


Figure 1.14: Illustration of the Compile and run section of PsyToolkit following a successful compiling of an experiment script.

During testing, the results of your experiment are immediately available to you using the View data button that appears when the experiment has completed. You also will have options to save the data for later analysis. Once the experiment has been deployed for data collection, participants will NOT have the option to view their own data.

Deploying an experiment requires that it be embedded into a PsyToolkit survey. This is incredibly simple to do. Recall that earlier in this tutorial, we created our first survey, titled “My_First_Survey"

with just two example questions (see Figure 1.6). When a survey is open for editing, you can add just a few lines of code to embed your experiment. The code below illustrates how the “SimpleRT_Experiment" script can be embedded into “My_First_Survey" just like any other survey element:

```
l: SimpleRTExperiment   #just a label for the survey item
t: experiment           #the type of items is "experiment"
- SimpleRT_Experiment   #gives the name of the experiment to include
```

Once this item has been added to a PsyToolkit survey, all that is needed is to (1) save the survey, (2) re-compile the survey, and (3) make sure the survey status is set to one of the two “online" options. After following these steps, you are ready to send out the link to your active survey and begin collecting data for your experiment.

1.3.4 Intermediate Experiment Generation

The Go_No-Go Task

In the previous section, we covered the basics of creating and deploying surveys and experiments with PsyToolkit. The experiment used was a simple RT experiment just one target stimulus. Of course the vast majority of experimental procedures in the psychological sciences involve much more complex procedures for manipulating and measuring an individual’s behavioral performance. For example, the second Donders procedure described above, called the “GoNo-Go Task", involves the presentation of two different experimental stimuli, with the participant asked to first identify each stimulus and respond to only one of the two. Unfortunately, generating such a task is more complex than simply adding a second stimulus (e.g., a square) image file and adding a new “`show bitmap`" line to the experiment script.

To understand this, think for a minute about what you learned in the prior section. The `task` section of experiment script is simply a list of instructions. If you were to simply add a second stimulus presentation after the original diamond stimulus, then the experiment script would simply generate a perfectly predictable series of alternating stimuli (e.g., diamond $\rightarrow$ square $\rightarrow$ diamond, etc.).

The predictability of the sequence would preclude the need to do any identification since one could just learn to press the space bar on every other stimulus. Thus, what is needed is a way to randomly select on each trial which of the two image files will be presented to participants.

There are several ways one could solve this problem. One idea might be to have PsyToolkit choose a random number between 1 and 2 at the beginning of each trial and present the diamond image if the number is equal to 1 and the square image if the number is equal to 2. Another possibility is to create separate Go and NoGo "`task`" sections in the experiment script and then list both of the tasks in the "`block`" section of the experiment script. As it turns out the latter option will be the simplest to implement in PsyToolkit.

Random Selection of Tasks

Before you start to incorporate random task selection into a Go/No-Go experiment, you should:

1. Create the No-Go stimulus using your favorite image editor (I used the example of a square earlier).
2. Click the orange copy button in the set of "Actions" options on the left side of the page.
3. Upload your No-Go stimulus image file.
4. Modify the experiment script to include a new "NoGo" task and add that new task to the "tasklist" in the "block" section of the experiment script (see Code block 3).

The Go/No-Go Experiment Script

<—SimpleRT experiment script snipped here—>

```
[1]     task NoGo_Task
[2]       set $TrialType "NoGo"
[3]       keys space
[4]       delay 1500
[5]       show bitmap fixation
[6]       delay 1000
[7]       clear -1
[8]       show bitmap square
[9]       readkey 1 2000
[10]      set $rt RT
[11]      save $rt $TrialType
[12]      clear -1
[13]
[14]    block GoNoGo_Block
[15]      tasklist
[16]        SimpleRTTask 50
[17]        NoGo_Task 50
[18]      end
```

Code block 1.3 Illustration of the NoGo task and changes needed to "block" section of the Simple RT experiment script to incorporate the NoGo task into the Go/No-Go experiment script in PsyToolkit.

The first thing you should notice about the NoGo task in code block 1.3 is that it is almost exactly like the original “SimpleRTTask" except that (1) the bitmap shown is “square" instead of “target" and (2) a variable called “TrialType" has been set at the beginning of the NoGo task and is also saved along with the “RT" variable. The first difference is, of course, needed to change the shape of the stimulus. The second difference will be clear later when we look at the stored data. Note that there were several design options here. For example, since participants are being asked to withhold their response when the square appears, it really isn’t necessary to record the reaction time or even set up the “keypress" listener at all. However, recording erroneous responses to the presentation of the square could potentially be informative. For example, if a participant presses the space bar whenever any of the two stimuli appear, it could be that they

either did not understand or were not following the instructions.

The second change to notice in this experiment script is the simple addition of the “NoGo_Task" to the list of tasks in the “`block`" section of the script, with 50 repeats (i.e., trials) designated for each task. Although it may not be intuitive how this produces a random series of trials, it turns out that the default behavior of PsyToolkit is to pool all of the 100 (50x2) trials inside the `tasklist` and then randomly draw from those 100 trials without replacement. Once you have completed the Go/No-Go experiment script, be sure to save and compile it as well as add a corresponding “experiment" type item to your survey just like you did for the Simple RT experiment script.

1.3.5 Retrieving Experiment Data from PsyToolkit

Although we briefly covered the way data is stored by PsyToolkit in the context of survey responses. It is important to also understand how data collected via an experiment are stored for analysis. Just as you did earlier, click the Prepare datafiles for download button, then click the Download data in zip file button to get a copy of the data on your computer.

PsyToolkit Data Files

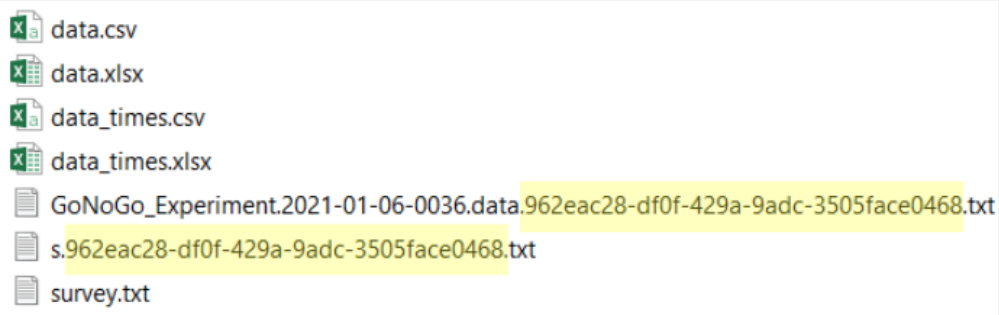


Figure 1.15: Illustration of the way files are stored and named by PsyToolkit during data collection.

The downloaded files will look very similar to those reviewed earlier. There will be survey data files in `.csv`, `.xlsx`, and `.txt` formats. You should also see, however, one or more separate text (`.txt`) files containing the data saved for the Go/No-Go experiment for each participant. These files will always start with the name of the experiment and the date, and end with a string of seemingly random characters. This random string of

characters will be constant across participants. In other words you will see matching random strings in the survey and experiment data files as well as the aggregate data.

For present purposes, the most important data files are the text (`.txt`) connected to the experiment. These files can easily be opened in most spreadsheet software like Microsoft Excel and Google Sheets. When you open files for PsyToolkit experiments, each row of the data represents a single trial of the experiment and any variables stored on each trial are represented by columns. An example of the data file saved during the Go/No-Go task described above is presented in Figure 1.16. Notice that the first entry in each row is a measure of the RT on that trial and that the No-Go trials are each denoted by the text “NoGo" as the second entry in each row.

Of course, this was a minimal working example and you, as the experimenter (and programmer), have complete control over how the experiment is designed and even how the output is stored in some cases. For example, recall from the code presented in Code block 1.3 that we added the $TrialType variable to the No-Go task so as to be able to discriminate those trials in the output file. It would be trivial to do the same for the task governing the Go trials so as to include a “Go" label in the second column on those trials. You may also notice that when the participant correctly withheld their response on No-Go trials, a value of 2000 was recorded for the RT on that trial. Hopefully you can reason that this value was entered because that was the time limit established in the experiment script for making a response. Although extremely unlikely,

PsyToolkit Experiment Data Files

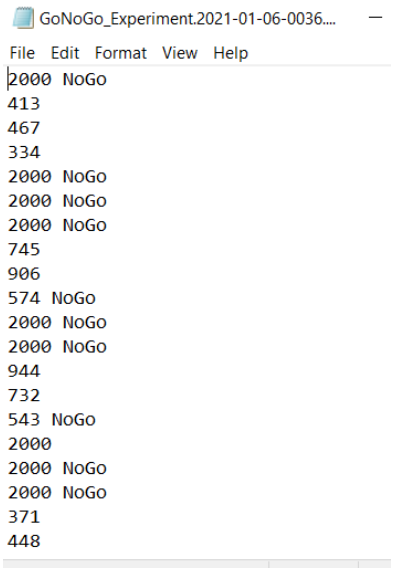


Figure 1.16. Inkscape is a free image editor that can be used to generate the bitmap files for use with PsyToolkit.

it is possible for a participant to press the space bar at exactly 2000 milliseconds and such trials would be impossible to distinguish in the data as it is presently formatted. This and many other aspects of the program are under the control of the experimenter in PsyToolkit, but do require more advanced knowledge of the scripting language.

1.3.6 Advanced Experiment Generation

One could easily devote an entire course to learning the scripting language used by PsyToolkit. Thus far, we have just scratched the surface of its capabilities. Even the most intuitive GUIs for developing experiments require some knowledge of coding in their native languages in order to implement more complex experimental designs and the same is true for PsyToolkit. Some common examples of the ways in which experiments become a little more difficult to implement are when the researcher would like to introduce randomness, when there are many experimental conditions (e.g., factorial designs), and when the researcher would like to implement conditional execution (e.g., display “correct" *(*ONLY) if the participant makes a correct response).

Introducing Randomness

When the hallmark of a good experiment is strict *control* of the variables, it may seem odd to be discussing how to add randomness to an experiment. However, there are a number of conditions under which randomness is actually used as a form of control over potential **Extraneous Variable**. In fact we already touched on this issue in the previous section. Recall that in the sample Go/No-Go it was suggested that it would be a bad idea to include presentation of the two stimuli in the same “task" section. The reason for this was that if the sequence of stimuli was perfectly predictable, participants would not need to identify the stimuli at all since they could simply anticipate pressing the button every other time the computer screen illuminated. In this case, the participants ability to anticipate target stimuli is an extraneous variable that could have a significant impact on the dependent variable of RT. In order to control this extraneous variable, our solution was to create two separate “tasks" in the experiment script (one for each stimulus) and randomize the order of presentation in the “block" section.

Another example of when randomness can help to control extraneous variables like participant anticipation is in the definition of experiment parameters like the ISI and ITI. For example, you may have noticed when testing the simple reaction time experiment above, that with the target stimulus appearing after the fixation cross at a consistent, 1 second, ISI it quickly becomes easy to anticipate the appearance of the target stimulus. In fact, if you were to plot your reaction times over trials, you would see these anticipatory benefits in the form of decreased RTs. Figure 1.17 illustrates this effect using RTs recorded using the simple RT task described above. When participants complete the task with a constant, 1000ms ISI between the fixation cross and the target stimulus, there is a clear benefit of about 100ms that is reached by around the 20th trial. Figure 1.17 also illustrates how the introduction of randomness can control for the effects of anticipation. When the simple RT task is completed with a random ISI between 800ms and 1200ms, the decrease in RT seen at the beginning of the experiment is entirely absent from the data.

Adding randomness to an experiment is quite simple in PsyToolkit using the keyword “`random`". As with most other keywords, `random` must be followed by parameters that govern its behavior. Several examples of the use of the `random` keyword are provided below:

Return a random value between 1 and 10	=	`random 1 10`
Return a random value between 2 and 100 in steps	=	`random 2 100 2`
Return a random value from specific set of values	=	`random from 12 98 105 34`
Return a random word from a list	=	`random from "A" "An" "Are"`

Each of the expressions above returns a value that is typically assigned to a variable using the "`set`" keyword. A **variable** in programming is like a container for information that can later be referenced or manipulated. In PsyToolkit variable names are preceded by the dollar sign ($) or the amperstand (&). For example, a variable named "ISI" with a stored value of 1000 could be created in PsyToolkit with either of the following lines:

Constant vs. Random ISI

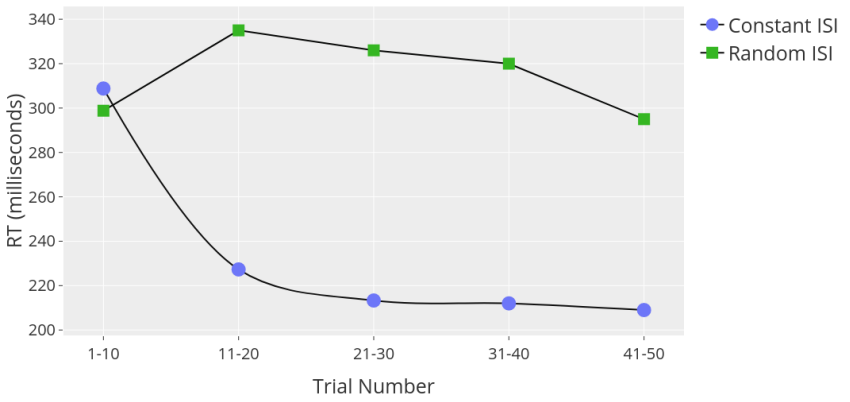


Figure 1.17. Inkscape is a free image editor that can be used to generate the bitmap files for use with PsyToolkit.

```
set $ISI 1000
      or
set &ISI 1000
```

In either case, $ISI or &ISI is a container for the value 1000. The difference between the two instances is that &ISI is a **global variable** and $ISI is a **local variable**. Global variables are "set" in the `options` section of an experiment script and can be used anywhere within the experiment script. Local variables are "set" within a `task` section of the experiment script and can *only* be used within the task in which they are defined. For now, we will focus on the local variable $ISI. The advantage of creating a constant variable like this might not be immediately obvious, but imagine that you've programmed a complex task using hundreds of lines of commands in which you have explicitly set the value of a delay interval to 1000ms in 250 separate places in the program. Now imagine that you decide that you need to use a value of 1200 instead of 1000. Without using a variable, this change would require you to manually find and change all 250 instances of the delay value. Alternatively, by setting an $ISI variable equal to 1000 at the beginning of the `task` script and then referring to that variable at each of the 250 instances, changing all of the delay times to 1200 would require *only* a single change to the definition of the $ISI variable at the top of the script.

Creating a variable that is assigned a random value is as simple as combining the `set` and `random` keywords. For example, the following line would create an $ISI variable with a randomly selected value between 800ms and 1200ms:

```
set $ISI random 800 1200
```

Finally, implementation of the random delay interval for the ISI in a PsyToolkit experiment script is as simple as referencing the $ISI variable when specifying the duration parameter of the `delay` keyword. (see lines 3 and 6 of code block 1.4).

The Go/No-Go Experiment Script

<—SimpleRT experiment script snipped here—>

```
[1]    task NoGo_Task
[2]      set $TrialType "NoGo"
[3]      set $ISI random 800 1200
[4]      keys space
[5]      show bitmap fixation
[6]      delay $ISI
[7]      clear -1
[8]      show bitmap square
[9]      readkey 1 2000
[10]     set $rt RT
[11]     save $rt $TrialType
[12]     clear -1
[13]
[14]    block GoNoGo_Block
[15]      tasklist
[16]       SimpleRTTask 50
[17]       NoGo_Task 50
[18]      end
```

Code block 1.4 Example of using a random number to define the duration of the ISI in the NoGo task

PsyToolkit Tables

In the Go/No-Go experiment script covered earlier in this chapter, we used two separate `task` sections to create the two conditions of the experiment; one presenting participants with a diamond and expecting a press of the space bar, and the other presenting a square to which participants were expected to withhold their response. This was a functional solution meant to simplify this introduction to PsyToolkit, but it is *not* the best solution and would not likely work for most experimental designs.

The recommended method for defining the separate conditions of a task in PsyToolkit is by creating a "`table`". The `table` section of an experiment script defines a table with rows and columns. Each row of the table contains the parameters of the task that will define your trial types. Each time a table is being used in a task, one of its rows is chosen at random. Columns can be referenced using the @ sign. For example, "@2" in an experiment script refers to the contents of the second column in whichever row was chosen for a given task trial. Users can specify how table lines are selected in the "`block`" section of the experiment script (default=random).

In order to demonstrate the use of tables, let's consider how they might be used to re-create the third of Donders' reaction time tasks, the Choice Reaction Time task (see Figure 1.3). Recall that the Choice RT Task involves presenting participants with two different stimuli (one at a time) and asking them to make one of two potential responses that depends on the identity of that stimulus. By comparison with the Go/No-Go task created earlier, the only major change is that there will need to be two different response options. Thus, the parameters of the Choice RT task that will change on each trial include the image file presented, and the response expected from the participant, defining the two experimental conditions.

These parameters can easily be defined in a `table` like the one below.

```
table My_First_Table
   diamond  1  "Diamond"
   square   2  "Square"
```

In the example above, the first line simply denotes that what follows will define a "`table`" section and gives this `table` the name "`My_First_Table`". The 2 subsequent lines define the parameters of each of the two conditions in the experiment, starting with the name of the bitmap file that will be presented (`diamond` or `square`), followed by the correct response for that trial (1 or 2), followed by a text label that will be used to identify the trial type in the output file ("Diamond" or "Square").

Once the table has been defined, it can be used inside the `task` section of the experiment script. This is illustrated in the full working example illustrated in code block 1.5, which illustrates all of the concepts covered thus far, including the use of tables to define experimental conditions and setting up a random ISI. One aspect of using tables to indicate correct responses that can be confusing for some new to PsyToolkit is that when more than one key is listed after the "`keys`" command (see line 16 of code block 1.5), the parameter passed to "`readkey`" (see line 21 of code block 1.5) is the number that corresponds with the serial position of the correct key in the "`keys`" list. For example, looking at the script in code block 1.5, if the second row of "`My_First_Table`" is selected on any given trial, then line 21, "`readkey @2 2000`", will cause PsyToolkit to read from keys `f` and `j` for 2000ms and consider the `j` key to be the correct response. This is because `@2` will refer to the second column of the table and the value in the second column of the second row is 2. Line 16, "`keys f j`", designates the set of appropriate keys and the key in position #2 is `j`.

The Choice RT Experiment Script

```
[1]    options
[2]      fullscreen                    #run in full-screen mode
[3]
[4]    bitmaps
[5]      fixation                      #load bitmap files
[6]      diamond
[7]      square
[8]
[9]    table My_First_Table            #create a table
[10]     diamond  1  "Diamond"         #define condition 1
[11]     square   2  "Square"          #define condition 2
[12]
[13]   task ChoiceRT_Task              #create the choice RT task
[14]     table My_First_Table          #use My_First_Table in this task
[15]     set $ISI random 800 1200      #set a random ISI
[16]     keys f j                      #designate response keys
[17]     show bitmap fixation          #show bitmap named "fixation"
[18]     delay $ISI                    #delay for ISI duration
[19]     clear -1                      #clear the last shown bitmap
[20]     show bitmap @1                #show bitmap named in column 1 of table
[21]     readkey @2 2000               #correct key is in column 2 of table
[22]     set $rt RT                    #record RT in variable $rt
[23]     set $status STATUS            #record STATUS in variable $status
[24]     save @3 $rt $status           #save column 3 label, $rt and $status
[25]     clear -1                      #clear the last shown image
[26]     delay 1500                    #delay 1500ms at end of trial
[27]
[28]    block ChoiceRT_Block           #set up a block of trials
[29]      tasklist                     #set up list of tasks
[30]       ChoiceRT_Task               #draw trials from ChoiceRT_Task
[31]      end
```

Code block 1.5 Annotated example of a Choice RT task using randomness and a table to define conditions.

One other notable addition to this latest example of an experiment script is that in addition to recording the RT from `readkey` in the variable $rt, line #23 records the STATUS from `readkey`. When the `readkey` function records that one of the listed keys has been pressed, the STATUS will be set to CORRECT (numerical value 1) if the key designated in the second parameter of the `readkey` function was pressed. Otherwise, if the participant presses the wrong key, it will be set to WRONG (numerical value 2), and if there is no response at all within the designated time limit, the STATUS code will be TIMEOUT (numerical value 3). Adding the $status variable will facilitate the identification of these different categories of responses in the analysis of the data saved by this experiment script, which will look similar to the example in Table 1.1.

Square	744	1
Square	1422	2
Diamond	748	2
Square	2000	3
Diamond	843	1

Table 1.1: Sample output data from the Choice RT experiment. The columnar data include: (1) name of the image file presented, (2) RT, and (3) the `readkey` STATUS (i.e., $status).

Conditional Execution

At this point, you should have all of the knowledge needed to develop a wide range of experiments using PsyToolkit. However, you may have noticed that all of the experiment scripts developed so far are completely determinate (aside from the order in which stimuli are presented). That is, once started, the experiment just runs through from beginning to end, even if the participant were to fall asleep at the computer! While this may be sufficient in some limited circumstances, it is very often the case the behavior of the experiment will need to change depending on the circumstances of each trial including, but not limited to, the behavior of the participant.

In programming, this kind of dynamic behavior of a program is called **conditional execution**, when some instructions in the experiment script are executed *only* if a test condition is determined to be true. Such behavior is most commonly accomplished in PsyToolkit using what is called an "*if*" statement, which takes the general form below where the condition is a logical statement that is either true or false.

```
if CONDITION
  CODE EXECUTED IF CONDITION IS TRUE
fi
```

At the heart of the if statement is the logical condition that will be evaluated when the experiment script is run. These logical statements are created using logical operators as seen in the examples in Table 1.2.

Operator	Definition	Example	Result
==	is equal to	100 == 102	FALSE
!=	is not equal to	100 != 102	TRUE
>	greater than	100 > 102	FALSE
<	less than	100 < 102	TRUE

Table 1.2: Several examples of logical statements, their interpretation, and their result.

Although each of the examples in Table 1.2 uses explicit numerical values in the logical expression, logical expressions can also involve variable names in your experiment script. Combining what we've learned about the format of an if statement in PsyToolkit, the construction of logical statements using logical operators, and the use of variables in PsyToolkit, let's look at some examples (below) of several useful if statements that could be used in an actual experiment script.

One of the most common uses of conditional execution is to control the feedback that is given to participants after they respond to a stimulus. For example, the researcher may wish to show the participant a bitmap with the word “correct" only if the response just executed was the correct response. Using what you have learned so far, that behavior could be implemented in the Choice RT experiment script (see Code block 5) by inserting the code below into the experiment script after the response has been made and the target stimulus cleared from the screen (line #25).

```
if $status == 1
  show bitmap correct
fi
```

One could use a similar if statement to display a bitmap with the word, “incorrect" when the participant makes an error by simply changing the logical statement to `if $status == 2` and changing the name of the bitmap file shown. It also turns out that there is an alternative approach that is even easier since all that is required here is to display some text to the participant. Rather than creating separate bitmaps for “correct" and/or “incorrect", the `show text` command, which simply displays text to the participant, may come in useful here. For example, the command:

```
show text "Hello World"
```

would display the text, “Hello World" to the center of the monitor.

Conditional expressions can also be combined using the logical operators `&&` (“and") and `||` (“or"). For example, what if you wanted your experiment script to show a special message like “Slow Down", but only if the participant made an incorrect response *AND* the response time was impossibly fast (e.g., less than 100ms). The simplest way to accomplish this goal would be to use a single if statement with a compound conditional expression and the `show text` command as illustrated below:

```
if $status == 2 && $rt < 100
  show text "Slow Down!"
fi
```

Tying Up the Loose Ends

In this chapter, you have learned how to create surveys and experiments using PsyToolkit. By now you should be able to create and deploy an experiment that involves the collection of RT and accuracy data, random selection of stimuli, randomization of parameters like ISI and ITI, and even conditional execution to display custom feedback messages. However, even if you were to follow all of the instructions in this chapter to the letter, would the example scripts be ready for deployment? Probably not! You, as the programmer, did not need to be instructed about how to respond to the stimuli presented, but a naive participant would have no idea what to do when that first stimulus appeared on the screen.

Incorporating task instructions into an experiment is important both because it provides participants with instructions and because it ensures standardization of instructions across all participants. Instructions screens can be created and presented just like any other bitmap you would use in your program. In other words, you already know everything you need to know in order to set up one or more instructions screens at the beginning of an experiment. Fortunately, however, there are some dedicated commands in PsyToolkit that make the process of presenting “messages" to participants exceedingly simple.

The “`message`" and “`pager`" commands can be used for instructions and/or end screens. These commands can be conveniently inserted into the the “`block`" section of the experiment script (see example below).

```
block Myblock
  message MyInstructionBitmap
  tasklist
    ChoiceRT_Task 100
  end
  message ThankYouBitmap
```

By default, the `message` command will present the named bitmap image and leave it on-screen until the participant presses the space bar key. This kind of self-pacing of the instructions is often desired to ensure that all participants have adequate time to read and understand the instructions.

In the event that several instructions screens are required, one could simply use more than one `message` command, however a drawback is that participants can't go backward after they have moved from one screen to the next. This limitation is solved with the `pager` command, which accepts a list of bitmaps and allows participants to scroll through that list using the arrow keys (see example below).

```
block Myblock
  pager MyInstruction1 MyInstruction2 MyInstruction3
  tasklist
    ChoiceRT_Task 100
  end
  message ThankYouBitmap
```

Note that, by default, the `pager` function requires the participant to press the "q" to proceed beyond the final bitmap to begin the experiment. Thus, the text of your final bitmap should instruct participants as such. That said, having finalized your experiment with a clear set of instructions, it is now ready to be added to a survey and deployed for data collection.

PsyToolkit Assignment

In a single file, re-create Donders' Simple RT, Go/No-Go, and Choice RT tasks using PsyToolkit. Your experiment must meet the following criteria:

1. First slide shows your full name
2. Includes one or more instructions pages at the start of each of the three RT tasks
3. Includes 50 trials per RT task
4. Includes Correct/Incorrect feedback after response on each trial (No response = incorrect)
5. Thank you screen at end of the experiment
6. Output data file contains four columns in the following order: [1] RT task name ("Sim", "GNG", or "CH"), [2] trial type ("Target" or "NonTarget"), [3] Accuracy ("0" or "1"), and [4] Reaction time.

Illustration of correct output format (*note that yours will not include column names*):

Task Name	*Trial Type*	*Accuracy*	*RT*
Sim	Target	1	458
Sim	Target	0	173
Sim	Target	1	634
.	.	.	.
.	.	.	.
.	.	.	.
GNG	Target	1	682
GNG	NonTarget	1	591
GNG	NonTarget	0	614
.	.	.	.
.	.	.	.
.	.	.	.
CH	Target	1	782
CH	Target	1	791
CH	Target	0	714

PSYC672

2. Reproducible Research with GIT & Git-HUB

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software environment and the complete set of instructions which generated the figures.

— David Donoho

Contents

2.1 What is Reproducible Research?

The quote by David Donoho (Stanford University, 1988) at the top of this chapter is widely cited in discussions of scientific transparency, often paraphrased to emphasize that published articles are merely "advertisements" for the actual scholarship, which lies in the data, the code used to process those data, and the computational methods used to generate the results. In the context of reproducible research, this means that sharing raw data, analysis scripts, and documentation—ideally along with information about software versions and computing environments—is essential for others to be able to verify findings, replicate analyses, or extend the work. Without access to these materials, even well-intentioned scientists cannot fully evaluate the integrity or robustness of the original conclusions. True reproducibility goes beyond publishing results; it requires open and structured sharing of the entire analytic workflow.

Historical Significance

Reproducibility has long been considered a cornerstone of the scientific method, dating back to the Enlightenment era, when thinkers like Francis Bacon and later Karl Popper emphasized the importance of empirical verification. In its most basic form, reproducibility refers to the ability of independent researchers to arrive at the same results using the same data and methodology. This principle ensures that scientific claims are not the result of chance, bias, or unrecognized error. Over time, particularly with the rise of computational science, the concept of reproducibility has evolved to include not only replication of experiments but also transparency in analytical workflows—sharing raw data, statistical code, and computational environments.

Reproducibility is often confused with generalizability, but the concepts are distinct. Generalizibility concerns whether a finding holds true in new settings, populations, or under different conditions. In contrast, reproducibility is narrower: it tests whether the same result can be obtained when the original procedures are followed exactly. A study can be reproducible but not generalizable (e.g., if it was conducted in a very specific sample), or it can be broadly generalizable but not reproducible if the analytic process was not transparent or cannot be followed precisely. Both qualities are important, but without reproducibility, we can't be confident in the validity of the original finding to begin with.

A notable case that highlights the importance of reproducibility and transparency is the 2010 study by Reinhart and Rogoff, which claimed a strong negative relationship between national debt and economic growth. The study was widely cited in economic policy debates around austerity measures. However, when a group of graduate students attempted to replicate the findings, they discovered major coding errors and selective data exclusion in the original Excel spreadsheet **?**. Once corrected, the negative relationship between debt and growth was much weaker than originally reported. This episode not only damaged the authors' reputations but also revealed how the absence of transparent, reproducible methods can lead to widespread misinformation and misguided policy decisions.

Such examples underscore the broader consequences of irreproducible research—not just for scientific integrity, but for public trust, policy, and the allocation of resources. As modern research becomes increasingly complex and data-driven, reproducibility is no longer optional; it is a professional obligation.

Methods to Improve Reproducibility

fill in later

2.2 Introduction to Git

Git is a popular and powerful open-source, version control software. Version control software manages changes in files as they are edited over time and by different users. With Git, users can track changes

made to files and back up old versions of files. When multiple variants of a product need to be managed or different people make changes, Git tracks the differences across branches.

Installing Git

Installing Git on Windows

1. Download Git for Windows
   - Visit the official Git website: https://git-scm.com/
   - Click Download for Windows. This will download an .exe installer.
2. Run the Installer
   - Double-click the downloaded .exe file.
   - When prompted by Windows, allow the installer to make changes.
   - Proceed through the setup wizard. The default options are fine for most users, but pay attention to these key choices:
     (a) Choose "Git from the command line and also from 3rd-party software."
     (b) Select "Use Git Bash only" unless you prefer integrating with Windows Command Prompt.
     (c) Use the default: "Use the OpenSSL library."
     (d) Choose "Checkout Windows-style, commit Unix-style line endings" (default).
   - Click Install, then Finish to complete the setup.
3. Open Git Bash
   - Git Bash is a terminal emulator that gives you a Unix-style shell on Windows. Find it in the Start Menu under "Git" → "Git Bash." Open it to start using Git

Git on macOS Option 1: Using Xcode Command Line Tools (Recommended)

1. Open the Terminal application (in Applications > Utilities).
2. Type the following command:

```
$ git -version
```

   If Git is not yet installed, macOS will prompt you to install Command Line Developer Tools. Click Install. This will install Git along with compilers and other tools needed for software development. Once the install is complete, try the "git –version" command again. You should now see a version number (e.g., git version 2.39.1).

Git on macOS Option 2: Install via Homebrew

1. Open Terminal.
   Install Git by typing:

```
$ brew install git
```

   After installation, verify with:

```
$ git -version
git version 2.39.1
```

Configuring Git (All Platforms)

After installing Git, configure your identity so your commits (changes) are properly attributed.

In your terminal (Git Bash or macOS Terminal), run:

```
$ git config -global user.name "Your Name"|
$ git config -global user.email "your.email@example.com"
```

You can confirm your settings with:

```
$ git config -list
Your Name
your.email@example.com
```

These values will be stored in a global configuration file (usually located at /.gitconfig).

Repositories

A Git repository (or repo) is a structured folder that tracks changes to files using Git. Think of it as a time machine for your project: it stores snapshots of your code, documents, or data at various points in time. You can revisit, compare, or restore earlier versions whenever needed. A Git repository can be "local", meaning that is only stored on your computer, or it can be stored on a shared platform like GitHub, GitLab, or Bitbucket where it can be shared with others.
A Git repository contains:

- Your project files (code, notebooks, scripts, etc.)
- A hidden .git folder that stores all the version history
- Metadata like commit messages, author info, and timestamps
- Branches for parallel development

Git repositories are powerful tools used to manage and track the evolution of a project's files over time. At their core, they enable users to keep a detailed record of changes, making it easy to see what was modified, when, and by whom. This is especially valuable in collaborative settings, where multiple people are working on the same files—Git helps prevent accidental overwrites by keeping each contributor's work organized and coordinated. Repositories also support the use of branches, which allow individuals to experiment with new features or ideas without affecting the main project. If a mistake is made or a change is no longer needed, Git makes it easy to revert to an earlier version.

In data science and research, Git repositories are commonly used to organize R or Python scripts, data cleaning routines, analysis code, manuscripts, reports, figures, and supplementary materials like README files. By storing all of these elements in a version-controlled environment, researchers can ensure transparency, reproducibility, and efficient collaboration throughout the lifecycle of a project.

Navigation Using the Terminal or Command Prompt

Even when using a shared platform like GitHub, you will want to designate a location for a repository on your computer. To do this, simply navigate to the desired location on your computer using the terminal.

1. On macOS or Linux
   - Open the Terminal
     Press Cmd + Space to open Spotlight Search.
     Type Terminal and press Enter.

   - See where you currently are
     When the terminal opens, you're usually in your home directory. You can check using the "pwd" (print working directory) command:

     ```
     $ pwd
     ```

This prints the current working directory (e.g., /Users/yourname).
You can list files and folders in the current directory using the "ls" (list):

```
$ ls
```

Navigate to another directory using the "cd" (change directory) command:

```
$ cd Documents/Projects
```

This moves you to the Projects folder inside Documents.
To move up one level, type:

```
$ cd ..
```

To go directly to your Desktop, type:

```
$ cd  /Desktop
```

Again, use pwd to verify where you are.

2. On Windows (Using Command Prompt)
   - Open Git Bash or Command Prompt
     Press Windows + S, type Git Bash or cmd, and press Enter.
   - See where you currently are
     When the terminal opens, you're usually in your home directory. You can check using the "cd" (current directory) command:

```
$ cd
```

You can list files and folders in the current directory using the "ls" (list):

```
$ dir
```

Navigate to another directory using the "cd" (change directory) command:

```
$ cd C:
Users
YourUsername
Documents
Projects
```

(Replace YourUsername with your Windows username.)
This moves you to the Projects folder inside Documents.
To move up one level, type:

```
$ cd ..
```

To go to a specific location, like the Desktop:

```
$ cd  /Desktop
```

Creating a Repository

1. Create a Project Folder Once you have found a suitable location for your repository, you can create a new folder to hold its contents using the "mkdir" (make directory) command, then navigate to the newly created folder/repository:

```
$ mkdir my-first-repo
$ cd my-first-repo
```

   Once you are inside the newly created folder, you can initialize it as a Git repository using:

```
$ git init
Initialized empty Git repository in /Users/paul/Documents/PSYC672/.git/
```

   This creates a .git folder and marks the directory as a Git repository.
2. Add Files Create or copy some files into the folder. Then stage the files for your first commit. To "stage files for commit" in Git means to mark specific changes (files or parts of files) that you want to include in your next commit. Staging acts like a buffer zone between editing files and permanently recording those changes in the project's history. Imagine you're packing a box to mail. You don't just throw everything in the box and seal it right away. First, you select the items you want to include — that's the staging area. Once you're sure, you seal the box and ship it that's the commit.

```
$ git add .
```

   This stages all files for commit.
   You can stage only some changes from multiple files—useful for separating unrelated changes into clean, logical commits. It helps ensure that only reviewed and intentional edits are committed. Git lets you review what's staged (git diff –staged) before committing.
3. Make Your First Commit

```
$ git commit -m "Initial commit"
```

   In Git, to "commit" files means to permanently record a snapshot of your staged changes into the repository's history. Think of it as saving a version of your project that you can always go back to. The "-m" flag allows you to add a comment describing this commit, in this case "initial commit."
4. View History As you commit changes to a project as it evolves, you can always view a record of those changes by viewing the repository's "log."

```
$ git log
```

Connect to a Remote Repository

Assuming you're using GitHub:

1. Create a free account at https://github.com
   As of 2019, the free account will allow you to create and share unlimited repositories (both public and private), however, private repositories will have some limited functionality when using a free GitHub account.

2. Create a new GitHub repository

Create a new repository in GitHub

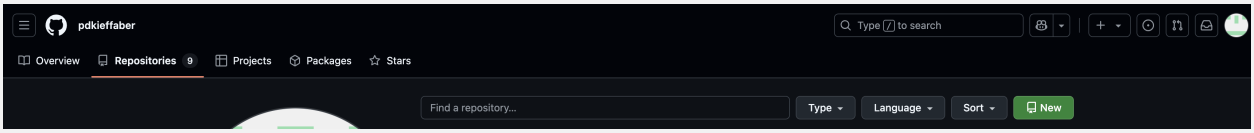


Figure 2.1: To create a new repository, first click the "Repositories" tab on the top left of the page, then click the "New" button.

Click the "New" button on the "Repositories" tab (See Figure **??**) to create a new repository. You'll be prompted to name your repository, optionally add a description, choose whether it will be public or private, and select initialization options (See Figure **??**. These include adding a README.md file (useful for describing the project), a .gitignore file (to exclude certain files or folders from version control), and a license (to define how others can use your code). If you initialize with a README, your repository will start with that file already committed to the main branch. This setup provides a clean starting point for managing and sharing your project.

Initial Repo Parameters

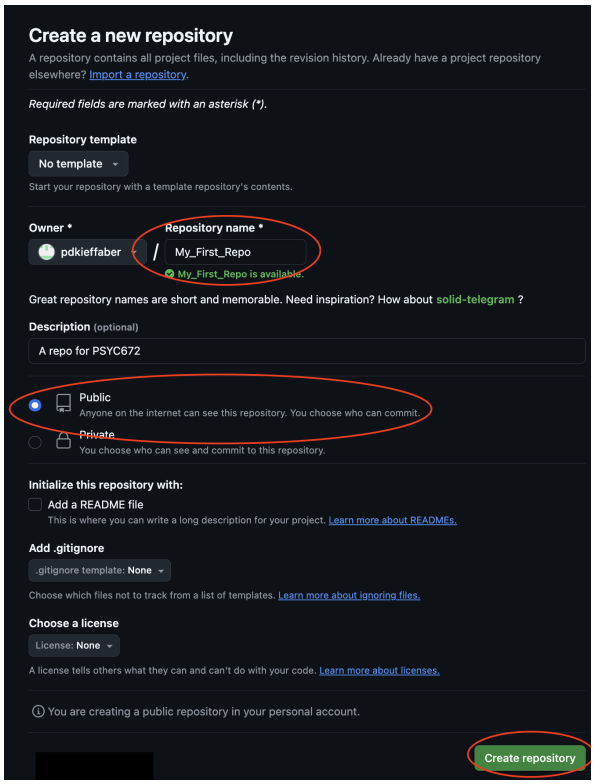


Figure 2.2. Give your new repository a name and designate it as public or private.

Note that at this point in the process, you have simply created one repository on your computer and another in GitHub. The next step is to connect and synchronize your local repository with the one you created on GitHub. This can sometimes be a bit tricky because the two repositories are, at this point, completely different. The first thing you will need is to generate a personal access token. A personal access token is a secure, user-specific alternative to a password that allows you to authenticate with Git hosting services like GitHub when using the command line or external tools. Because GitHub removed password authentication for Git operations over HTTPS in 2021, a personal access token is now required to push, pull, or clone repositories via HTTPS. This token functions like a temporary password and can be granted specific scopes or permissions, such as access to public repositories, private repositories, or workflow automation. You generate it from your GitHub account settings, and once created, you can use it in place of a password when prompted during Git operations. For security, tokens should be stored securely and rotated periodically.

3. Generate a personal access token.
   (a) Verify your email address if it hasn't been verified yet.
   (b) Click your profile photo in the upper-right corner of any GitHub page, then click Settings.
   (c) In the left sidebar, click Developer settings.
   (d) Under Personal access tokens, click Tokens (classic).
   (e) Click Generate new token, then Generate new token (classic).
   (f) Enter a name for the token in the "Note" field.
   (g) Select an expiration date under Expiration.
   (h) Choose the scopes to grant the token. For example, select repo to access repositories from the command line.
   (i) Click Generate token.

   Be sure to save the token somewhere discrete. Remember, it is like your private Git password.
4. Sync Local and Remote Repositories
   You might think of your local copy of the repository as your first "branch." A branch in Git is essentially a parallel version of your project. It is like a timeline of your project, where each branch represents a different line of development. The default branch is usually named "main" (or "master" in older repositories). Branches are created in order to do things like try out new features, fix bugs, or work on different versions of a project without changing the main version. Importantly, each branch tracks its own history of commits. Because the main branch in the newly created repository on GitHub is empty, you are going to move/copy your current branch (what is in your local repository) to the "main" branch on GitHub, overwriting the existing main branch if necessary.
   - Copy the GitHub repo URL (e.g., https://github.com/username/my-first-repo.git)
   - Set your local branch to be the main branch.

     ```
     $ git branch -M main
     ```

   - "Push" your Main Branch to the Remote Repository

     ```
     $ git remote add origin https://github.com/username/my-first-repo.git
     $ git push -u origin main
     ```

   The command above will push your local commits to GitHub. The -u flag sets the upstream (tracking) relationship between your local branch and the remote branch. After running this, your local main branch "knows" it corresponds to origin/main remotely.
   When you git clone, git fetch, git pull, or git push to a private remote repository using HTTPS URLs on the command line, Git will ask for your GitHub username and password. When Git prompts you for your password, enter your personal access token. Password-based authentication for Git has been removed in favor of more secure authentication methods.

Cloning a Repository

The previous example illustrated the process of moving an existing local repository to a GitHub repository that can be shared with others. To begin working on a project that is already hosted on GitHub (or another Git server), you can just clone the repository to your local machine using the git clone command. This creates a complete copy of the remote repository, including its history and all tracked files. For example:

```
$ git clone https://github.com/yourrepo.git
```

This command sets up a folder on your computer containing the project files and a .git directory for

tracking changes. It also links your local copy to the original remote repository, making it easy to fetch updates or push your own changes. In fact, if you are setting up a new repository from scratch, it would be more straightforward to create a new repository on GitHub, then clone that repository to your local machine.

The Basic Git Workflow

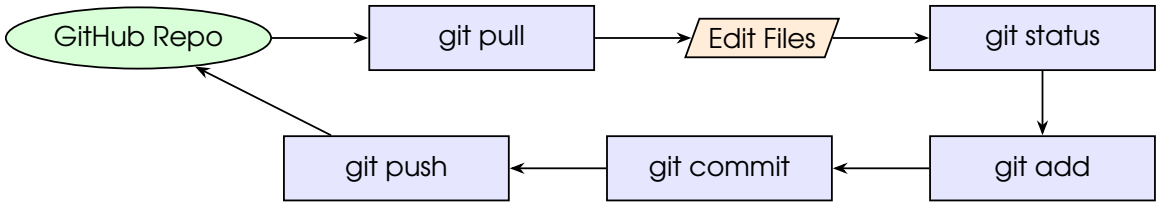


Figure 2.3: Recommended Git Workflow: Pull → Edit → Status → Add → Commit → Push

1. Staying Up to Date
   When working with collaborators (i.e., multiple people accessing the same repository), or even if it is just you working on multiple computers, it is important that you are working with the most recent version of the repository before you try to make any changes. By running "git pull" before making any changes to a repository you update your local branch with any new commits from GitHub and avoid accidentally overwriting someone else's work. Git will try to automatically merge the remote changes with your local changes and alert you if any conflicts exist, and let you know if you are already up to date!
   You can pull the most updated version of a branch using:

   ```
   $ git pull origin main
   From https://github.com/pdkieffaber/PSYC672
    * branch main -> FETCH_HEAD
    Already up to date.
   ```

2. Making Changes
   Once the repository is synchronized/cloned, you can begin making changes. This typically involves editing existing files or creating new ones using your preferred text editor or integrated development environment (IDE). Changes might include writing code, updating documentation, or adding figures and data files. Git will keep track of these modifications automatically, but nothing is saved to version control until you explicitly stage and commit the changes.
3. Checking Status
   At any point, you can check which files have been modified, added, or deleted using the git status command. This provides a quick overview of the current state of your working directory. It highlights which changes are untracked (new files), which have been modified but not yet staged, and which are ready to be committed. Regularly running git status is a good habit, as it helps avoid confusion and ensures you're aware of everything that's changed.

   ```
   $ git status
   ```

4. Staging Files
   Before committing changes, you need to stage them with the git add command. Staging tells Git which changes you want to include in your next commit. You can stage individual files (e.g., git

add report.md) or all modified files at once using "git add .". This step allows you to group related changes together for clarity and cleaner commit history. *Only staged changes will be included in the next commit.*

```
$ git add .
```

5. Committing Changes
   Once your changes are staged, use git commit to save them to your local repository. A commit is like a snapshot of your project at a specific point in time. You'll include a descriptive message with each commit using the -m flag, such as:

   ```
   $ git commit -m "Fix typo in introduction and add conclusion section."
   ```

   This message helps you (and your collaborators) understand the purpose of the changes later. Each commit is stored in the repository's history, making it easy to review or revert if needed.
6. Pushing to GitHub
   After committing your changes locally, you can upload them to the remote repository (e.g., GitHub) using git push. This command syncs your local branch with the corresponding branch on the server. Most commonly, you'll use:

   ```
   $ git push origin main
   ```

   Here, origin refers to the remote repository, and main is the branch you're pushing to. This step makes your work available to collaborators and serves as a remote backup of your progress.

Branching

Branching is one of Git's most powerful features, allowing developers to diverge from the main line of development and continue work without affecting the main branch. A branch is essentially a pointer to a specific commit, allowing you to isolate changes for a particular feature, bug fix, or experiment. By default, a new Git repository starts with a single branch, often called main, which represents the official version of the project.

When you create a new branch (e.g., git branch feature-login), you're creating a parallel version of your repository where you can make changes independently of the main branch. You can then switch to this branch using "git switch." This makes branching especially useful for collaborative work: each contributor can develop features in their own branch without risk of introducing conflicts into the main project.

After finishing work on a branch, changes can be merged back into the main branch. This is typically done using git merge, which combines the histories of both branches. In cases where changes conflict, Git will prompt the user to resolve these conflicts manually. This ensures that only well-tested and intentional changes are added to the main branch.

Importantly, pushing a new branch to GitHub (e.g., git push) does not overwrite the main branch. Each branch exists independently on the remote as well as locally. This setup supports pull requests and code reviews, which are best practices in team-based development. By maintaining a clean and organized branching strategy, such as Git Flow or GitHub Flow, teams can work more efficiently and with fewer integration headaches.

For example, let's assume you are working with a research team on a project containing some data and an analysis script using R. The current code is working fine for now, but you have been tasked with coding a bootstrapping procedure in R that will enhance the analysis. With Git branching, you can begin

developing your code without interfering with any the core analysis pipeline that will still be used by your collaborators while you are developing your part of the project. Also, by cloning your branch, other team members can implement, test, and revise the bootstrapping logic independently. Once complete and validated, the branch can be merged back into the main branch, integrating the improvements into the official project codebase while maintaining a clean development history. This workflow helps ensure transparency, reproducibility, and minimal disruption to ongoing analyses.

Example Workflow

Assuming you have already cloned the repository to your local machine, the first step in an example workflow would be to execute a pull to make sure you are up to date. Make sure you're on the main branch and up to date:

```
$ git checkout main
Already on 'main'
Your branch is up to date with 'origin/main'.
$ git pull origin main
From https://github.com/pdkieffaber/PSYC672
 * branch main -> FETCH_HEAD
 Already up to date.
```

Now you are ready to start a new branch called "bootstrap-procedure". Let's say you're currently on the main branch and want to start developing bootstrapping the code:

```
$ git checkout -b bootstrap-procedure
Switched to a new branch 'bootstrap-procedure'
```

You're now on your new branch, ready to start making changes—Git will keep these changes isolated from main until you're ready to merge.

Now edit your R scripts (or create new ones) in your local repository to include the new bootstrapping procedure. When you want to make them available to your collaborators on GitHub you'll just need to stage the necessary file(s):

```
$ git add bootstrap_code.R
```

Note that you can use git add . to stage all changed files if appropriate.

Commit the changes you've made in the bootstrap-procedure branch.

```
$ git commit -m "Add initial version of bootstrapping procedure"
[bootstrap-procedure 03f6df2] Add initial version of bootstrapping procedure
1 file changed, 1 insertion(+)
create mode 100644 bootstrap_code.R
```

You can commit multiple times as you iteratively make changes to and test your new code.

Now push your branch to GitHub if you would like others to be able to work on it simultaneously:

```
$ git push -u origin bootstrap-procedure
Enumerating objects:  4, done.
Counting objects:  100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects:  100% (2/2), done.
Writing objects:  100% (3/3), 315 bytes | 315.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:  Create a pull request for 'bootstrap-procedure' on GitHub by
visiting:
remote:  https://github.com/pdkieffaber/PSYC672/pull/new/bootstrap-procedure
remote:  To https://github.com/pdkieffaber/PSYC672
 * [new branch] bootstrap-procedure -> bootstrap-procedure
branch 'bootstrap-procedure' set up to track 'origin/bootstrap-procedure'.
```

This workflow ensures that the main branch always reflects stable code, while branches provide safe spaces for iterative development.

Once you have finished developing the bootstrapping_code.R file and you have completed all testing, your changes can be seamlessly integrated back into the main branch. After merging, the feature branch can be deleted or kept for record-keeping.

Start the merge process by first switching back to the main branch and pull the latest changes:

```
$ git checkout main $ git pull origin main
```

Next, use git merge to merge the changes from your branch:

```
$ git merge bootstrap-procedure
```

If there are no conflicts, Git will automatically merge the changes.

Finally, push the updated main branch to GitHub:

```
$ git push origin main
```

Once merged and verified, you can (optionally) delete the branch locally:

```
$ git branch -d bootstrap-procedure
```

...and remotely:

```
$ git push origin -delete bootstrap-procedure
```

2.3 Collaboration with Git

Forking Vs. Branching

Branching is the standard approach for managing changes within a single Git repository, especially when working on new features, bug fixes, or experiments. A branch is like a parallel version of your project where you can make changes without affecting the main codebase. When you're ready, you can merge your changes back into the main branch (often called main or master). Branches are lightweight and

designed for collaborative development within a shared repository — they're perfect for teams working closely together with shared write access.

Forking, on the other hand, creates a personal copy of an entire repository under your own GitHub account. It's commonly used in open-source or large-scale collaborative projects where contributors do not have write access to the original repository. When you fork a repo, you can make changes freely in your copy and later submit a pull request to suggest your changes to the original project. Forking is ideal for contributing to public projects, exploring changes independently, or creating a personal adaptation of a tool or dataset.

In short, use branches when you're collaborating within a team or organization where everyone has access to the same repository. Use forks when you're contributing to someone else's project or want to maintain a separate version. Both support experimentation and version control, but forks provide an additional layer of ownership and permission separation.

You cannot create a fork directly from the Git command line — forking is a feature specific to GitHub (and similar platforms like GitLab or Bitbucket), not Git itself.
To fork a repository (via GitHub):

1. Go to the repository page on GitHub.
2. Click the "Fork" button in the upper right corner.
3. GitHub creates a copy of the repository under your own account.

After you've forked it on GitHub, you can clone your forked repository to your computer:

```
$ git clone https://github.com/your-username/forked-repo-name.git
$ cd forked-repo-name
```

Pull Requests

A pull request (PR) is a way to propose changes you've made in a branch (or forked repository) to be merged into another branch, typically the main branch of the original project. Pull requests are central to collaboration on GitHub, allowing teams to review, discuss, and integrate code changes in a structured and trackable way.

Before you can create a pull request, your changes must be pushed to a branch on GitHub. For example, if you've created a new branch locally called feature/bootstrap-analysis:

```
$ git checkout -b feature/bootstrap-analysis
# MAKE YOUR CONTRIBUTIONS...
$ git add .
$ git commit -m "Add bootstrap analysis to R script"
$ git push origin feature/bootstrap-analysis
```

Once your branch has been pushed to GitHub, go to your repository's GitHub page in your browser. GitHub will often show a banner suggesting you create a pull request for the newly pushed branch. Click "Compare & pull request". If no banner appears, go to the "Pull requests" tab and click "New pull request". Choose the branch you want to merge into (typically main) and the branch you want to merge from (e.g., feature/bootstrap-analysis).

When prompted, give your pull request a descriptive title and detailed description of the changes you made. This helps reviewers understand the intent of your changes. For example:

```
   Title:  Add bootstrapping functionality to analysis pipeline
Description:This PR adds a bootstrapping procedure to the analysis.R file for
```

```
resampling reaction times.  It uses 10,000 resamples and returns 95% confidence
intervals.  Also includes updated unit tests in test_analysis.R.
```

Finally, click "Create pull request". You can now request specific reviewers, assign labels, and link related issues.

2.4 Version Control for Experiments and Data Analysis

Version control is not just for software developers, it can help to manage the evolution of analysis scripts, experimental materials, and documentation. Researchers can leverage Git and GitHub to make every stage of the scientific process more organized, transparent, and reproducible. By versioning analysis scripts, experimental code, stimuli, and lab notebooks, researchers create an audit trail that enhances collaboration and scientific rigor. This section explains how Git can be used to manage different aspects of experimental research.

Tracking Analysis Scripts with Git

In many areas of research, analysis scripts often evolve over time as analytic strategies change, new datasets are collected, or new subjects are added to an existing dataset. Without version control, tracking the provenance of an analysis pipeline or understanding the impact of a small code change can become almost impossible. Git allows researchers to:

1. Track line-by-line changes in R, Python, or MATLAB scripts.
2. Experiment with new methods in branches without disrupting the main pipeline.
3. Revert to previous versions when errors or regressions are discovered.
4. Tag important milestones, such as versions used in a submitted manuscript.

Example Workflow: Add and commit your new script:

```
$ git add analysis_v1.R
$ git commit -m "Initial version of analysis script"
```

Make updates and track changes:

```
$ git add analysis_v2.R
$ git commit -m "Refactored script to include new preprocessing step"
```

Tag manuscript version:

```
$ git tag -a v1.0 -m "Version used in preprint submission"
```

Managing Experimental Designs and Stimuli

In behavioral and cognitive experiments, stimuli files (images, audio clips, text), presentation scripts (e.g., in PsychoPy, E-Prime, jsPsych), and configuration settings change frequently during the development and piloting phases. Using Git to manage these files helps you avoid the confusion of having multiple folders labeled "Final", "Final2", or "Final_final", makes collaboration with lab members or co-authors more transparent, and allows you to roll-back to working versions if bugs are introduced.

When using Git to manage experimental designs and stimuli, it is generally recommended that you follow a standard repository structure with separate folders for stimuli, data, scripts, and results.
Suggested Repository Structure for Experiments:

```
experiment-project/
|
|-- stimuli/
|    |-- images/
|    |-- audio/
|    |-- text/
|
|-- data/
|    |-- (add to .gitignore if large)
|
|-- scripts/
|    |-- present_experiment.py
|    |-- experiment_config.json
|
|-- results/
|    |-- (add to .gitignore if large or auto-generated)
|
|-- README.md
|-- LICENSE
```

2.5 Using Markdown with Git/GitHub

What is Markdown?

Markdown is a lightweight markup language created by John Gruber in 2004 to make writing formatted text simple and readable in plain text. Markdown files use the .md file extension and are widely adopted for technical documentation, README files, project wikis, and more. The appeal of Markdown lies in its easy-to-learn syntax and its ability to be rendered as rich HTML or PDF documents by a variety of tools and platforms.

Why Use Markdown with Git?

Markdown and Git are a natural pairing for scientific and technical documentation:

- Version Control: Git tracks every change to your Markdown files, letting you revert, compare, or collaborate on documentation just as you would with code.
- Transparency: All edits are timestamped and attributed, supporting reproducibility and accountability.
- Collaboration: Multiple users can work on the same files, resolve conflicts, and review changes via pull/merge requests.
- Web Integration: Many platforms (e.g., GitHub, GitLab) automatically render .md files as formatted web pages, making your documentation instantly readable.

Features Included: Headings
Bold and Italic text
Lists (ordered and unordered)
Links
Images
Inline code
Code blocks

2.6 Tracking & Reviewing Changes

Once you start version-controlling your work with Git, it's important to understand how to inspect changes, compare revisions, and understand the history of your project. These capabilities are essential for debugging, documenting, and collaborating in scientific research. Git offers a suite of powerful tools for reviewing file changes, commit history, and differences between branches.

Checking What Has Changed: `git status`

The first step in tracking changes is to ask Git what's new, modified, or staged for commit:

```
$ git status
```

This command will show:

- Untracked files: new files not yet added to Git
- Modified files: tracked files that have changed but not yet staged
- Staged changes: files that are ready to be committed

SHOW EXAMPLE OUTPUT AND EXPLAIN

Seeing the Details of Your Changes: `git diff`

Sometimes you may want to see exactly what lines have changed in your files:

```
$ git diff
```

This will display the changes in the working directory that are not staged. Once you stage them (git add), the changes disappear from git diff. To see staged changes:

```
$ git diff -cached
```

You can also compare specific files:

```
$ git diff analysis_script.R
```

Or compare different commits or branches:

```
$ git diff main experiment-branch
```

SHOW EXAMPLE OUTPUT AND EXPLAIN

Reviewing Commit History: `git log`

To see a record of commits, `git log` shows commit hashes, authors, dates, and messages:

```
$ git log
commit 3a81b2f3018d2cbbd540e8d1e0db457ca170c9c4 (HEAD -> main)
Author:  Paul Kieffaber <paul@example.edu>
Date:  Tue Jun 18 14:03:00 2025 -0400
Refactored bootstrapping code into separate function
commit c71ae6c8481732ccf7c993dcfcdf5e1a24c5fd9c
Date:  Mon Jun 17 16:20:34 2025 -0400
Added initial analysis script for Cognitive Control Study
```

For a simpler view:

```
$ git log -oneline
```

Or with a graphical view of branches:

```
$ git log -oneline -graph -all
```

Viewing Specific File History

If you want to see the history of a single file:

```
$ git log - analysis_script.R
```

To view changes made to that file:

```
$ git diff HEAD^HEAD - analysis_script.R
```

To see who last edited each line of a file:

```
$ git blame analysis_script.R
```

This can be helpful when debugging or reviewing collaborative contributions.

2.7 Handy Tips and Best Practices for Using Git and GitHub

While Git's core functionality—tracking changes and managing versions—is already powerful, several features and best practices can make your workflow cleaner, more efficient, and easier to manage, especially in collaborative or research-focused projects.

Use `.gitignore` to Exclude Unwanted Files

In most projects, there are files that should not be tracked by Git—such as temporary outputs, compiled files, or large datasets. A `.gitignore` file can be created using any text editor and it tells Git to ignore the files and folders designated in that file. Create a .gitignore file in the root of your repository, and Git will apply these rules when adding and committing files. This helps keep your version history clean and prevents accidentally uploading sensitive or unnecessary data.
Below is an example of a `.gitignore` file for a research project:
NOTE: the hashtag symbol indicates a descriptive comment

```
# Ignore everything in the results folder and all .log files
/results/
/*.log

# Ignore all .csv and .tsv files in the data folder
/data/*.csv
/data/*.tsv
```

Commit Often and Write Clear Messages

Frequent commits make it easier to track changes and revert specific modifications if needed. A good commit message should be concise but descriptive, stating what was changed and why.
Poor message:

```
$ git commit -m "05/23/26 - updated file"
```

Better message:

```
$ git commit -m "05/23/26 - Added log transformation to reaction time analysis for
normality"
```

Use `git tag` to Mark Significant Milestones

Git tags are a powerful way to mark specific points in your repository's history—typically used to indicate releases, milestones, or versions (e.g., v1.0, paper-submission, final-analysis).
What Does Git Tag Actually Do?

- Git tags a specific commit, not a branch or a file directly.
- It does not tag individual files (like marking one .R or .py file)—instead, it marks the snapshot of the entire repository at a particular commit.
- Because tags point to commits, and branches also point to commits, a tag can be thought of as a "bookmark" for a particular state of your repo, regardless of which branch it was on.

There are two types of tags, lightweight tags and annotated tags.
Lightweight Tag →Just a name that points to a commit

```
$ git tag manuscript-submission
```

Annotated Tag→Contains a message, tagger name, and date—useful for documentation and releases.

```
$ git tag -a manuscript-submission -m "Version for journal submission"
```

If you are working on a local repository, you will need to push the tag to GitHub:

```
$ git push origin manuscript-submission
```

To go back to a tagged version in Git, you can use the git checkout command—but how you do it depends on what you want to achieve.
Case 1: Temporarily inspect a tagged version (detached HEAD)
If you just want to look at or run the code at the tagged state (without making changes):

```
$ git checkout manuscript-submission
```

NOTE: This puts you in a "detached HEAD" state. In Git, HEAD is a pointer that tells you where you currently are in your repository—specifically, it points to the latest commit on the current branch. In other words, "HEAD" is your current location in the project history. For example, If you're on the main branch, HEAD points to the latest commit on main. A "Detached HEAD" state is when HEAD is pointing directly to a commit, not to a branch. When you checkout a tagged commit from the past, you're not on a branch, so you can't commit changes unless you create a new branch. This state is useful only for inspecting exactly what the repo looked like at the time of the tag.
Case 2: Create a new branch from the tag (to make changes)
If you want to modify code based on a tagged version:

```
$ git checkout -b manuscript-revision manuscript-submission
```

This creates a new branch called manuscript-revision starting from the commit the tag manuscript-submission points to. You can now make changes (i.e., commits, merge changes, etc.) to this new branch.

2.8 Assignment

1. Initial Setup by Instructor
   Create a GitHub repo (public or private) with:
   - A LaTeX template
   - A bibliography file (refs.bib)
   - A repo with the folder structure:
     research-report/
     |—- intro.tex
     |—- methods.tex
     |—- results.tex
     |—- discussion.tex
     |—- report.tex
     |—- refs.bib
     |—- README.md
2. Student Instructions
   (a) Fork the repo https://github.com/pdkieffaber/PSYC672.
   (b) Create a new branch of your fork using:

   ```
   $ git checkout -b yourname-intro
   ```

   Replace yourname with your name.
   (c) Add .tex section (e.g., intro.tex) and cite at least one source using BibTeX.
   (d) Add a comment to the log file PSYC470.md
   (e) Commit changes and push:

   ```
   $ git add intro.tex
   $ git commit -m "Added my introduction section"
   $ git push origin yourname-intro
   ```

cool quote about LMM.

— some philosopher

Contents

3.1 Background: Introduction to Statistical Modeling

In many fields of research, from psychology and education to medicine and social sciences, the primary goal is to understand and explain relationships between variables. Often, researchers collect data that involves complex relationships, where outcomes (dependent variables) are influenced by several factors (independent variables). Statistical modeling provides the tools needed to quantify these relationships, identify patterns, and make predictions based on data. Without a structured approach to data analysis, researchers risk making inaccurate conclusions, or even worse, overlooking important relationships that could drive insights or interventions.

At its core, statistical modeling is a way of representing data through mathematical equations, allowing researchers to describe, explain, and predict real-world phenomena. These models help simplify complex systems by providing a framework for analyzing the effects of multiple variables simultaneously, while also accounting for random variation and uncertainty. For example, in psychology, statistical models are used to examine how different cognitive, emotional, or environmental factors influence behavior. In education, models might be used to study how teaching methods or school environments affect student performance. In medicine, researchers might use statistical models to identify factors that influence the progression of a disease or the effectiveness of a treatment. And in social sciences, models can help understand complex social behaviors and the impact of policy changes or societal trends.

In each of these fields, the challenge is often that data is not independent or identically distributed. Real-world data frequently comes with some form of hierarchy or clustering — for instance, multiple measurements from the same individual over time, or students nested within different schools. This means that the observations in a dataset are not independent of each other, violating the assumptions of traditional statistical methods, such as ordinary least squares regression. As a result, more sophisticated modeling techniques are needed to properly account for these complex data structures and avoid biased or misleading conclusions. This is where models like linear mixed-effects models (LMM) become invaluable, as they offer a way to include both fixed effects (representing population-level relationships) and random effects (accounting for individual or group-level variability).

Statistical models thus serve as a powerful tool in research by transforming raw data into insights. These models can reveal patterns and associations that might not be immediately obvious, test hypotheses, control for confounding factors, and enable researchers to make inferences about larger populations based on sample data. As datasets continue to grow in size and complexity, the need for robust, flexible, and accurate statistical modeling has never been greater. Linear mixed-effects models represent one such advancement, allowing for a more nuanced understanding of how various factors affect outcomes in hierarchical or grouped data structures.

Limitations of Simple Linear Models

Traditional linear models, such as ordinary least squares (OLS) regression, have long been the foundation for statistical analysis due to their simplicity and interpretability. These models assume that the relationship between the independent variables (predictors) and the dependent variable (outcome) is linear, and that all observations in the dataset are independent of each other. Under these assumptions the dependent variable can be expressed as a weighted sum of the independent variables, plus an error term:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$$

Where:

- y_i is the dependent variable (response) for observation i
- x_i is the independent variable (predictor)
- β_0 is the intercept (value of y when $x = 0$)
- β_1 is the slope (change in y for a one-unit change in x)
- ε_i is the error term (random deviation)

Multiple independent variables are possible too in this context. In multiple linear regression, we simply have n more predictors:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ...\beta_n x_{in} + \varepsilon_i$$

The goal of OLS is to find the values of $\beta_0, \beta_1, \beta_2, ...\beta_n$ that minimize the sum of squared residuals (SSR). A residual is the difference between the observed value y_i and the predicted value $\hat{y}_i$.

So, OLS minimizes:

$$SSR = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

This is why it's called "least squares": it finds the line (or hyperplane in multiple regression) that minimizes the squared distance from each data point to the regression line.

OLS regression works well when the data points are independent and identically distributed, and when there is no clustering or grouping of observations. However, this idealized scenario is rare in real-world data, especially in fields like psychology, education, medicine, and social sciences, where nested or hierarchical data structures are common.

In many research designs, data are inherently grouped. For example, in a longitudinal study of patients, multiple measurements are taken from the same individuals over time, creating repeated observations within each subject. Similarly, in educational research, students are nested within schools, and observations from students in the same school are likely to be more similar to one another than to students from different schools. This hierarchical structure introduces two key sources of variability that traditional linear models fail to account for: within-group variability (differences among individuals within the same group) and between-group variability (differences between groups themselves).

The standard OLS model assumes that all data points are independent, which is violated in these hierarchical structures. When this assumption is not met, the model's estimates of regression coefficients can become biased or inefficient, leading to inaccurate conclusions. Specifically, by ignoring the clustering of data, the model treats all observations as if they are equally independent, which underestimates the true variability in the data. This often results in incorrect standard errors, leading to misleading significance tests and inflated Type I error rates (false positives). In other words, the model may indicate that certain relationships are statistically significant when, in fact, they may be due to the unaccounted-for correlation between observations within the same group. Furthermore, OLS regression cannot properly model the variability between groups. For instance, when multiple groups (e.g., schools, hospitals, or geographic regions) are involved, the assumption that all subjects respond the same way to predictors becomes unrealistic. Some schools may have inherently better academic outcomes due to local factors, and this variation needs to be explicitly modeled. Without accounting for this group-level variability, the OLS model does not allow for differences in intercepts or slopes between groups, thus simplifying the model to an unrealistic assumption that one set of coefficients applies universally across all levels of the data.

The result of using a simple linear model on nested data is not only potential bias in estimates but also a loss of the ability to capture the complexities inherent in the data structure. This is where linear mixed-effects models (LMMs) provide an important advantage. By incorporating random effects to account for variability within and between groups, mixed-effects models offer a more flexible and accurate way to model hierarchical data, providing a better understanding of how predictors influence outcomes at both the individual and group levels.

3.2 The Need for Mixed-Effects Models

Fixed vs. Random Effects

In the context of linear mixed-effects models (LMMs), one of the key features is the distinction between fixed effects and random effects. These two types of effects allow researchers to model both the population-level relationship between predictors and the response variable, as well as the variability among groups or individuals that cannot be captured by fixed effects alone.

Fixed Effects: Population-Level Relationships

Fixed effects represent the systematic, population-level relationship between the predictors (independent variables) and the response variable (dependent variable). These effects are constant across all levels of the grouping factor and are typically used to estimate the average effect of a treatment or intervention, or the overall relationship between a predictor and the outcome. For example, in a study where the effect of a new teaching method on student performance is assessed, the fixed effect for the teaching method would represent the average difference in performance between the treatment group and the control group, across all students. This effect is the same for all individuals or groups in the dataset, and it reflects the population-level trend that one expects to see on average.

In a simple linear regression model, all coefficients are considered fixed effects. In mixed-effects models, the fixed effects still represent the population-level effects, but they are combined with random effects to account for variability at different levels of the data hierarchy. For example, in a multilevel model examining students' test scores, fixed effects could include predictors like the student's age, gender, or teaching method, which are expected to have the same effect for every student across all schools.

Mathematically, the fixed effect coefficients ($\beta_0, \beta_1, \dots$) are typically estimated from the data using methods like maximum likelihood estimation (MLE) or restricted maximum likelihood estimation (REML). These fixed effects are what we are most interested in when trying to understand the general trends or relationships in the data that apply across the entire population.

Random Effects: Group-Level and Individual-Level Variability

In contrast to fixed effects, random effects account for variability at the group or individual level. These effects capture the fact that different groups or individuals may respond differently to the same treatment or exposure. For instance, even if a new teaching method has an average positive effect on students' performance (captured by the fixed effect), some students might

show a much stronger response, while others might show a weaker or even negative response. This variation among individuals, or among groups (such as schools), is modeled as random effects.

In a typical mixed-effects model, random effects are included to model the variation in the intercept (random intercepts) or the variation in the slopes (random slopes) across groups or individuals. For example, in a longitudinal study of patient outcomes, the response variable might be influenced by both a fixed effect (e.g., treatment) and a random effect (e.g., individual patient differences). Random intercepts would allow each patient to have their own baseline level of the outcome variable, accounting for the fact that some patients start with higher or lower values. Random slopes could be included if the treatment effect varies across individuals, meaning some patients might benefit more from the treatment than others.

Mathematically, random effects are often represented as deviations from the overall fixed effect, where the random effect term (u_j) is assumed to follow a normal distribution with a mean of zero and a variance of σ_u^2. These random effects are estimated along with the fixed effects and provide a way of partitioning the variance in the response variable into components that reflect both systematic effects (fixed) and individual or group-level variability (random).

Fixed vs. Random Effects: Complementary Roles

The distinction between fixed and random effects is essential for understanding the full range of variation in the data. Fixed effects tell us about the overall trend in the data, such as the effect of a treatment, whereas random effects tell us about the individual differences in how subjects or groups respond to that treatment. Together, they offer a more complete and nuanced understanding of the data, allowing for more accurate predictions and insights.

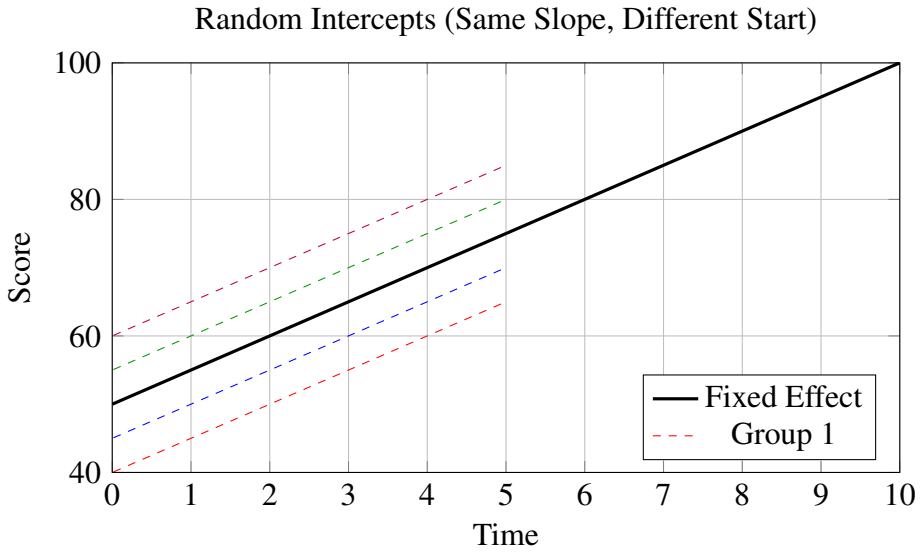


Figure 3.1: Each group shares the same slope (fixed effect), but starts at a different baseline.

In many real-world datasets, observations are naturally organized into groups or clusters, creating a hierarchical structure. This means that the data are not independent of each other; rather, individual observations within a group are more similar to one another than to observations in other groups. Hierarchical data structures commonly arise in fields like education, medicine, and psychology, where subjects or units are nested within higher-level units. For instance, in an educational study, students are nested within classrooms, and classrooms are nested within schools. Similarly, in medical research, patients are nested within hospitals, and hospitals are nested within regions.

Hierarchical structures like these present unique challenges for statistical modeling, as traditional linear models assume that all observations are independent. Ignoring this clustering can lead to biased estimates and incorrect inferences. Linear mixed-effects models (LMMs) are designed to handle this kind of hierarchical data by explicitly modeling both fixed effects (population-level relationships) and random effects (group-level variability). In this framework, the data are divided into multiple levels, and the model accounts for the correlation within groups, while still allowing for variability between groups.

Consider a study examining student performance on a standardized test, where students are nested within classrooms, and classrooms are nested within schools. In this case, the data have three levels: individual students (Level 1), classrooms (Level 2), and schools (Level 3). A hierarchical model can accommodate this structure by including random effects for classrooms and schools, acknowledging that students within the same classroom or school may be more similar to each other than to students in other classrooms or schools.

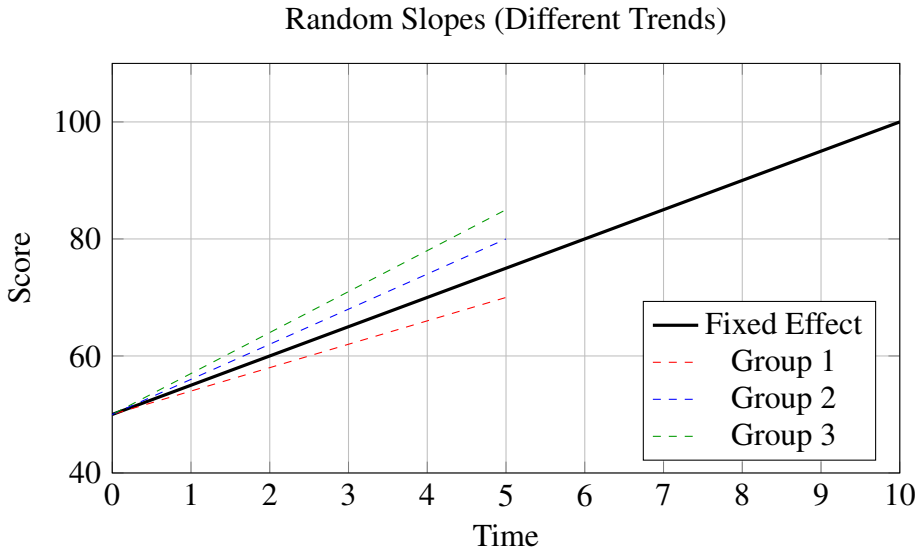


Figure 3.2: All groups start at the same baseline, but their rates of change differ.

For example, the fixed effects might include predictors such as study hours and parental involvement, representing the overall population-level effect on test scores. The random effects would capture the variability at the classroom and school levels. This means that while study hours might have a similar effect on students across all schools (captured by the fixed effect), there could be differences in how the effect of study hours is experienced at the classroom level, or how schools differ in their baseline student performance (captured by the random intercepts). A linear mixed-effects model would allow us to properly account for both the within-classroom and between-school variability, ensuring that we make accurate estimates of the treatment effects and group-level differences. By modeling the hierarchical structure in this way, mixed-effects models not only provide more accurate parameter estimates but also give us insights into how variability is distributed across different levels of the hierarchy, which is crucial for making valid conclusions in clustered data.

3.3 Definition: Linear Mixed-Effects Models

A linear mixed-effects model (LMM) is an extension of traditional linear regression that incorporates both fixed effects and random effects, allowing for more flexible modeling of data with hierarchical or grouped structures. While a standard linear regression model examines the relationship between one or more predictor variables and a response variable by estimating population-level coefficients, it assumes that the observations are independent and identically distributed. This assumption often fails when the data are nested or grouped, as is common in many real-world situations. To account for this, mixed-effects models introduce random effects, which model the correlation between observations within groups or clusters and allow for variability at multiple levels (e.g., individual, group, or time).

At its core, a linear mixed-effects model is similar to a linear regression model, but with the addition of random effects to capture the variability within and between groups. A typical linear regression model is specified as:

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i \tag{3.1}$$

Where Y_i is the response variable for observation i, β_0 is the intercept, β_1 is the coefficient for the predictor X_i, and ε_i is the error term representing residual variation. Note that this model assumes that all observations are independent and have the same error distribution.

In contrast, a linear mixed-effects model introduces random effects to account for clustering or grouping in the data. For example, in a study where multiple measurements are taken from individuals over time, a mixed-effects model would allow for each individual to have their own baseline (random intercept) and potentially a unique slope (random slope). The model can be written as:

$$Y_{ij} = \beta_0 + \beta_1 X_{ij} + \mu_j + \varepsilon_{ij} \tag{3.2}$$

Where:

- Y_{ij} is the response for observation i within group j,
- β_0 is the fixed intercept, representing the population-level average outcome when all predictors are zero,

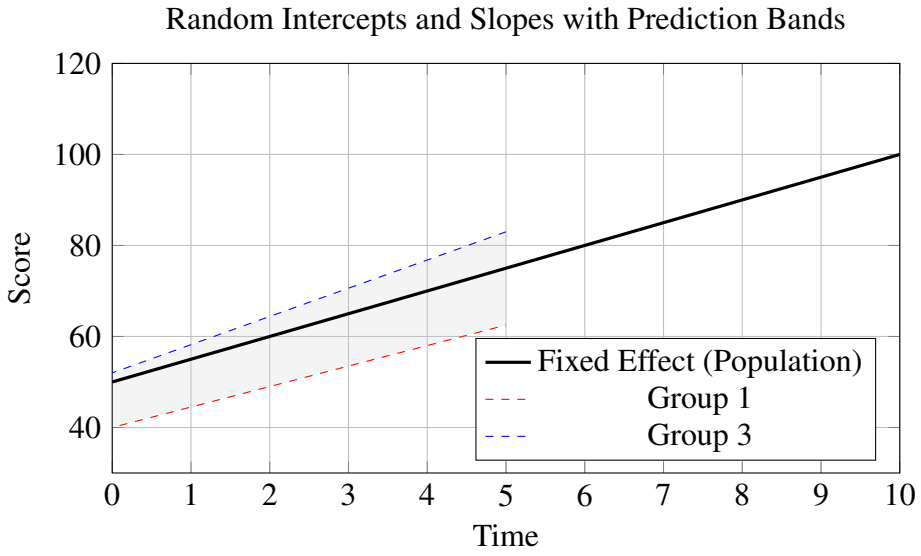


Figure 3.3: Combining both random slopes and intercepts, showing how predictions vary by group.

- β_1 is the fixed effect for the predictor X_{ij},
- μ_j is the random effect for group j, capturing the variation in the intercept across groups,
- ε_{ij} is the residual error term for the individual observation, assumed to be independently distributed with zero mean and constant variance.

Fixed Effects vs. Random Effects

The main distinction between fixed and random effects is their interpretation and role in the model:

Fixed effects represent the population-level relationship between the predictors and the outcome. These effects are assumed to be the same across all groups or individuals in the study. For example, in the model above, β_0 and β_1 are fixed effects, representing the overall intercept and the average effect of the predictor across all groups.

Random effects represent individual or group-level deviations from the overall trend captured by the fixed effects. These effects are unique to each group or individual and account for variability that cannot be explained by the fixed effects alone. In the model above, u_j is a random intercept that allows each group to have its own baseline outcome, independent of the fixed intercept, β_0.

Why Use Mixed-Effects Models?

Linear mixed-effects models are particularly useful when the data exhibit a hierarchical or clustered structure, such as repeated measures from the same individuals or measurements nested within groups. For instance, in a longitudinal study, measurements from the same subject over time are likely to be correlated, and using a simple linear regression would ignore this dependency, leading to biased estimates. By incorporating random effects, LMMs can model the correlation between repeated measurements from the same individual and provide more accurate and meaningful results.

Mixed-effects models also offer flexibility in modeling variability at multiple levels. For example, in a study of student performance across schools, random effects could model both the individual variation in performance (random intercepts) and the variation in how teaching methods affect different students (random slopes). This level of detail allows researchers to better understand not just the average effects of predictors, but also how those effects might vary across different contexts or groups.

3.4 Assumptions About Random Effects in Linear Mixed Effects Models

Linear mixed effects models are a powerful tool for analyzing data with hierarchical or grouped structures, incorporating both fixed effects (parameters of interest) and random effects (parameters accounting for variability within groups). A key assumption in these models is that the random effects u_j are normally distributed with a mean of zero and a variance of σ_u^2. This assumption has several important implications for the interpretation and performance of the model.

The assumption of a mean of zero reflects the idea that the random effects u_j represent deviations from the overall fixed effects. These deviations are assumed to be symmetrically distributed around zero, ensuring that there is no systematic bias in the random effects across groups. For instance, in a study analyzing student performance across schools, the school-level

random effects account for variability between schools but are not expected to favor one direction (e.g., consistently higher or lower performance) without evidence in the fixed effects.

The assumption of normality allows for the use of maximum likelihood or restricted maximum likelihood estimation methods, which rely on the distributional properties of the random effects to estimate parameters efficiently. Normality simplifies computations, as it ensures the joint likelihood of the data is tractable. While deviations from normality may not severely impact model performance in large samples, strong departures could lead to biased estimates or reduced power, particularly in small datasets.

The variance σ_u^2 quantifies the variability in the random effects across groups. It provides insight into the extent of heterogeneity among groups in the dataset. For example, a small σ_u^2 suggests minimal variability between groups, whereas a large σ_u^2 indicates substantial differences. Correctly estimating σ_u^2 is crucial for understanding the structure of the data and for making appropriate inferences.

In practice, the assumption of normality should be checked using diagnostic tools such as quantile-quantile (Q-Q) plots or residual plots of the estimated random effects. While the model is robust to mild deviations from normality, significant violations may require alternative approaches, such as using non-parametric methods or transforming the random effects. Nevertheless, the assumption that u_j are normally distributed with mean zero and variance σ_u^2 is foundational in ensuring the validity and interpretability of linear mixed effects models.

3.5 Covariance Structure in Linear Mixed Effects Models

Linear mixed effects models are designed to account for the inherent correlations in grouped or hierarchical data structures. This is achieved by modeling both the random effects and residual errors using a covariance matrix. The covariance structure of the model describes how observations within the same group and across different groups are related, ensuring appropriate handling of dependencies in the data.

The within-group covariance, $Cov(Y_{ij}, Ykj) = \sigma^2$, where Y_{ij} is the response (or observed outcome) for the i-th observation within the j-th group and Y_{kj} is the response for another observation k within the same group. These represent two observations from the same group. For example, if you are analyzing test scores of students (Y), Y_{ij} would represent the test score of the i-th student in the j-th school and Y_{kj} would represent the test score of the k-th student in the j-th school. This within-group covariance, $Cov(Y_{ij}, Ykj)$, reflects the variability in responses among observations within the same group that is not explained by the fixed effects or random effects. This covariance is attributed to the residual errors, which are assumed to be independent and identically distributed within a group. In simpler terms, the residual errors introduce randomness to the observations, but this randomness is consistent in magnitude (σ^2) across all pairs of observations within the same group.

The between-group covariance, $Cov(Y_{ij}, Y_{kl}) = \sigma_u^2$, where Y_{ij} and Y_{kl} are different observations from the j-th and l-th groups, represents the shared variability introduced by the random effects across different groups. The random effects model group-level deviations from the fixed effects and create correlations between observations within the same group. For example, in a study involving students nested within schools, the random effect for each school introduces a common component that links all students in the same school, resulting in a covariance of σ_u^2 for observations from the same group.

Together, σ^2 and σ_u^2 define the covariance structure of the model, distinguishing between variability within and across groups.The total covariance structure in a linear mixed effects model thus reflects the hierarchical nature of the data. Observations within a group exhibit higher correlations due to the shared group-level random effect and the residual error, while observations between groups are correlated only through the random effects. This structure is key to capturing the dependencies in the data and ensures that the model produces accurate estimates of both fixed and random effects.

The covariance matrix also informs estimation and inference in the model. By specifying the structure of the covariance, the model accounts for both within- and between-group variability, which is crucial for obtaining valid standard errors and hypothesis tests. Different assumptions about the covariance structure can be tested and compared using information criteria, ensuring that the chosen structure is well-suited to the data. Properly specifying and understanding the covariance structure is thus integral to the success of a linear mixed effects model in capturing the complexities of grouped data.

3.6 Model Fitting and Estimation

3.6.1 Maximum Likelihood Estimation (MLE) in Linear Mixed Effects Models

Maximum Likelihood Estimation (MLE) is a fundamental method for fitting linear mixed effects models (LMMs). The core idea of MLE is to estimate the model parameters—including fixed effects, random effects, and variance components—by maximizing the likelihood of the observed data. In other words, MLE seeks the parameter values that make the observed data most probable under the assumed model.

In the context of LMMs, the observed data consist of responses Y_{ij}, which are modeled as a combination of fixed effects, random effects, and residual errors. The likelihood function represents the probability of observing the data, given specific

parameter values for the fixed effects (β), random effects (u), and variance components (σ^2 and σ_u^2). To perform MLE, the likelihood function is expressed in terms of these parameters, and the parameter values are chosen to maximize this function.

The likelihood for an LMM involves the marginal distribution of the responses Y, which accounts for the contributions of both fixed and random effects. Because the random effects are unobserved, they are treated as latent variables with a prior normal distribution $N(0, \sigma_u^2)$. To compute the likelihood, the random effects are integrated out of the joint distribution of Y and u, leaving a marginal likelihood that depends only on the fixed effects and variance components. This integration results in a complex likelihood function that involves the covariance structure of the model.

Maximizing the likelihood in LMMs often requires numerical optimization techniques because the likelihood function is not analytically tractable. Specialized algorithms, such as the Expectation-Maximization (EM) algorithm or Newton-Raphson methods, are commonly used. These methods iteratively adjust the parameter estimates to find the values that maximize the likelihood.

MLE provides estimates for the fixed effects (β) that describe the average trends in the data and the variance components (σ^2 and σ_u^2) that quantify within-group and between-group variability. While the random effects (u) themselves are not directly estimated in MLE, their distributions are used to predict group-level deviations, often through empirical Bayes methods.

Although the mathematical details are beyond the scope of this review, the intuition behind MLE is that it finds values for the model parameters (e.g., β, σ^2, and σ_u^2) that make the observed data most likely under the model. Think of the likelihood function as a "score" that evaluates how well a particular set of parameter values explains the data. MLE maximizes this score.

In LMMs, the observed data (Y) depend on fixed effects (β), random effects (u), and residual errors (ε). Since we don't have direct measures of u because it represents underlying, group-specific effects, we estimate their contributions through the patterns of variability in the observed data by integrating over all possible values of u to calculate how likely the observed Y is, given the estimated fixed effects (β) and variance components (σ^2 and σ_u^2).

For a single parameter, the geometric interpretation of Maximum Likelihood Estimation (MLE) focuses on finding the value of the parameter that maximizes the likelihood function, $L(\theta)$, which quantifies how likely the observed data are given the parameter. The likelihood function is a function of the parameter θ and reflects how well a particular value of θ explains the observed data.

Geometrically, $L(\theta)$ can be represented by a curve plotted in 2D space where the x-axis represents the different values of θ and the y-axis represents the likelihood values, $L(\theta)$ (see Figure 2.4) . The MLE is the value of θ at the peak of this curve, where $L(\theta)$ is maximized (at the peak of the curve).

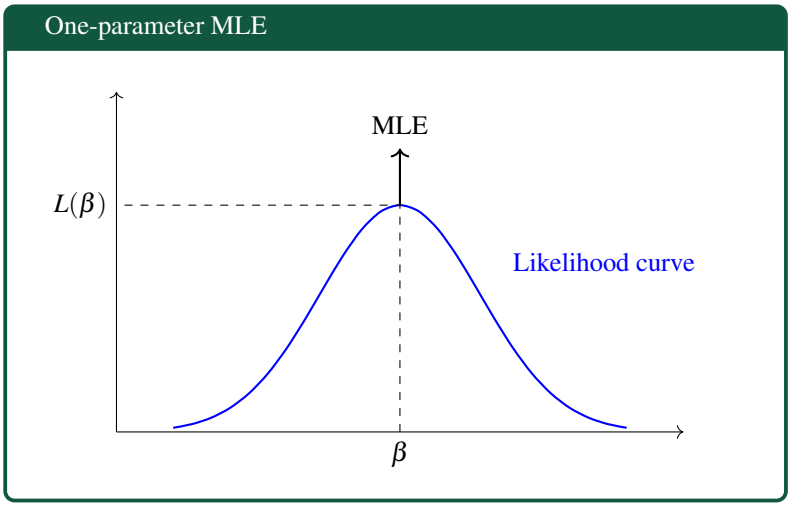


Figure 3.4: Illustration of MLE for a single-parameter, β in a LMM. The MLE is the value of β at the peak of this curve, where $L(\beta)$ is maximized

Of course, we just discussed that there are three parameters in a LMM, β, σ^2, and σ_u^2. While it can be difficult to visualize MLE in three-dimensional space, we can easily extend the geometric interpretation of MLE with a single-parameter to a two-parameter MLE. Imagine a mountain "landscape" where the height of the terrain represents the likelihood value for a given set of parameter values. MLE is like climbing to the highest point on this terrain, which represents the parameter estimates that best explain the data. For example, MLE for a LMM will find the parameter estimates for σ^2 and σ_u^2 that best explain the observed data (see Figure 2.5).

In practice, MLE has several desirable properties, including consistency (estimates converge to true values with increasing sample size) and asymptotic normality (parameter estimates are approximately normally distributed in large samples). However, MLE can be sensitive to model misspecification, and its performance may be affected in small sample sizes, where alternatives like Restricted Maximum Likelihood (REML) are sometimes preferred to improve variance component estimation.

3.6.2 Restricted Maximum Likelihood

Restricted Maximum Likelihood (REML) is a technique commonly used in mixed-effects models to obtain unbiased estimates of variance components, such as the variance of the random effects. Unlike standard Maximum Likelihood Estimation (MLE),

which estimates both fixed and variance parameters simultaneously, REML focuses on estimating the variance components in a way that accounts for the loss of degrees of freedom due to the estimation of the fixed effects.

In mixed-effects models, the data consist of both fixed effects (such as population-level parameters) and random effects (such as group-specific deviations). The challenge with MLE is that it estimates both fixed effects and random effects variance simultaneously, leading to biased estimates of the variance components, especially when the sample size is small.

REML addresses this issue by "restricting" the likelihood to the random effects. This is achieved by conditioning on the fixed effects, thereby removing the influence of the fixed effects from the estimation of the random effects variance. Specifically, REML maximizes the likelihood of the residuals (the part of the data after accounting for the fixed effects), which provides a more accurate estimate of the variance components. By focusing on the residuals, REML ensures that the estimation of the variance components is unbiased, particularly in small sample sizes.

REML is especially useful when the goal is to obtain reliable estimates of the random effects variance σ_u^2 and σ^2, as it corrects for the bias that conventional MLE might introduce in small datasets.

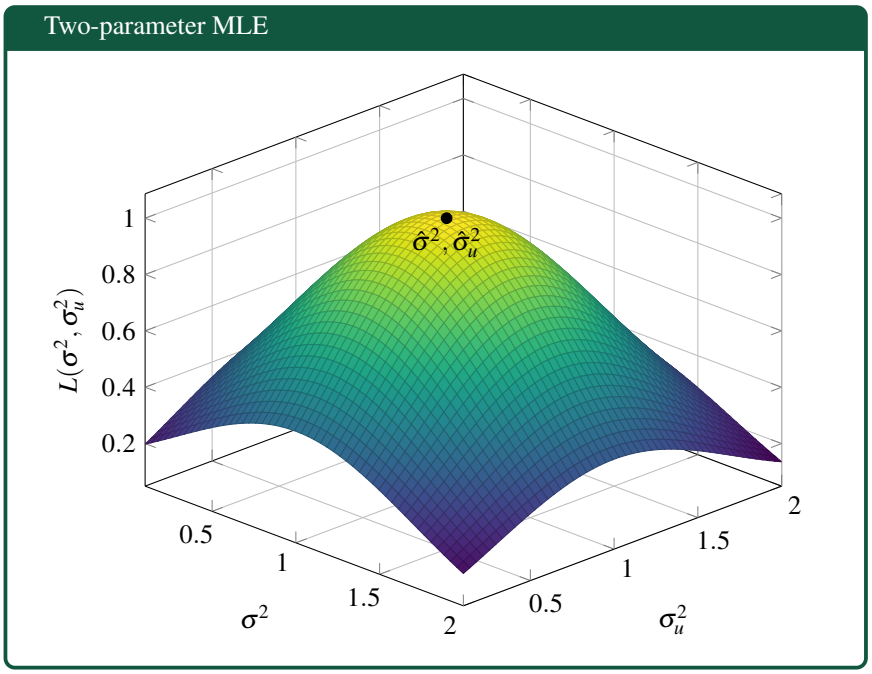


Figure 3.5: Illustration of MLE for two-parameters, σ^2 and σ_u^2 in a LMM. The MLE is the set of values for σ^2 and σ_u^2 at the peak of this surface, where the likelihood $L(\sigma^2, \sigma_u^2)$ is maximized

The main distinction between REML and MLE lies in how they handle the estimation process. MLE estimates both fixed and random effects parameters simultaneously. REML, on the other hand, estimates the variance components based only on the residuals after the fixed effects are accounted for, which generally results in less bias, especially for the random effects variance. Overall, REML is the preferred method when the primary focus is on the random effects and variance components, as it provides unbiased estimates that are crucial for valid inference in mixed-effects models.

3.7 Model Assumptions in LMM

In Linear Mixed-Effects Models, several key assumptions underpin the validity of the model and the accuracy of the results. These assumptions include linearity, independence, normality, and homoscedasticity. Violations of these assumptions can lead to biased parameter estimates or incorrect conclusions. It's essential to assess whether these assumptions hold before interpreting the results of an LMM.

Linearity

The linearity assumption posits that the relationship between the predictors (both fixed and random effects) and the response variable is linear. This means that the response Y is modeled as a linear function of the predictors, with the fixed effects contributing to this linear relationship and the random effects accounting for variability between groups or clusters. Violations of this assumption can lead to misinterpretation of the effects of the predictors. For example, non-linear relationships might be overlooked, resulting in misleading conclusions about the strength or nature of the relationship between variables.

There are several ways to test whether the assumption of linearity is met.

- **Residual vs. Fitted Plot**: This plot shows the residuals (errors) on the y-axis and the fitted values (predictions from the model) on the x-axis. A random scatter of points around zero indicates that the linearity assumption holds (.

If there are systematic patterns, such as curves or trends, this suggests that the relationship between predictors and the response might be nonlinear. If patterns appear, consider adding polynomial terms or using other nonlinear modeling approaches (e.g., splines, generalized additive models).

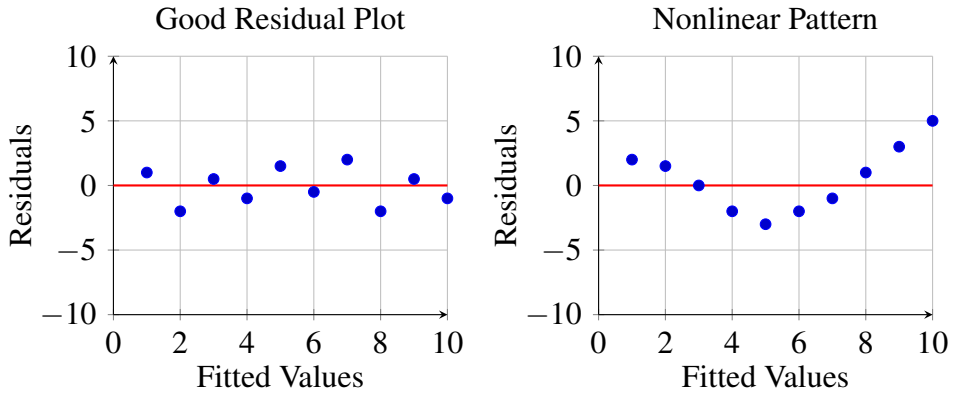


Figure 3.6: Left: Random residuals suggest linearity. Right: Curved pattern suggests nonlinearity.

- **Component Plus Residual (CERES) Plot**: A more advanced plot that can detect both linearity and other functional form issues by adding fitted values for each predictor.

Independence

The independence assumption states that the residual errors within each group are independent of each other. In other words, the errors for different observations within the same group should not be correlated. This assumption is critical because correlated errors violate the assumption of independent observations, which can bias both the estimation of fixed effects and the inference about variance components. However, the random effects are explicitly modeled as correlated within groups, which allows the model to account for this correlation at the group level. Residuals should not show patterns within groups, which could indicate a violation of the independence assumption.

There are several ways to test the independence assumption.

- **Durbin-Watson Test**: This test evaluates the autocorrelation of residuals. It is commonly used in time series models, but it can also be applied to check for residual dependence in mixed models. Values close to 2 suggest no autocorrelation, while values below 1 or above 3 indicate positive or negative autocorrelation, respectively.
- **Plot of Residuals by Group**: If there is evidence of dependence within a group, such as a pattern in the residuals within certain levels of the grouping variable, it may indicate a violation of independence.
- **Random Effects Variance**: If random effects are estimated to be large, this could suggest residual dependence, especially when they are unexplained by the model.

Normality

Both the random effects u and the residual errors ε are assumed to be normally distributed with mean zero. This assumption is crucial for statistical inference, as many of the inferential procedures (such as hypothesis testing and confidence interval estimation) rely on the normality of these components. If the residual errors or the random effects deviate significantly from normality, it may affect the precision of the estimates and the validity of statistical tests. Checking normality is important, especially for small sample sizes, as deviations can lead to larger biases or over-optimistic p-values.

There are several approaches to testing the normality assumption.

- **Q-Q Plot (Quantile-Quantile Plot)**: A Q-Q plot compares the quantiles of the residuals to the quantiles of a normal distribution. If the points fall approximately along a straight line, the residuals are likely normally distributed. Significant deviations from the straight line suggest departures from normality (e.g., heavy tails or skewness).
- **Histogram of Residuals**: A histogram allows visual inspection of the distribution of residuals. For normality, the histogram should resemble a bell-shaped curve. If the shape is skewed or shows heavy tails, this indicates that the residuals are not normally distributed.
- **Shapiro-Wilk Test / Anderson-Darling Test**: These statistical tests assess the normality of residuals. If the p-value is small (e.g., less than 0.05), the null hypothesis of normality is rejected, indicating non-normality.
- **Normality of Random Effects**: You can also check the normality of random effects using a Q-Q plot or histogram of the random effects (if available), as random effects should follow a normal distribution by assumption.

Homoscedasticity

The homoscedasticity assumption assumes that the variance of the residuals is constant across all groups. This means that the spread of residuals should be roughly the same across different levels of the grouping variable. If the residuals show unequal variance (a phenomenon known as heteroscedasticity), this can indicate that the model is mis-specified or that a transformation of the data may be necessary. Heteroscedasticity can lead to inefficiencies in the estimation process, and it can affect the reliability of the standard errors, making statistical inference problematic.

There are several ways to test the homoscedasticity assumption.

- **Residual vs. Fitted Plot**: This plot can also be used to check for homoscedasticity. If the residuals fan out or contract as fitted values increase, it suggests heteroscedasticity (non-constant variance). Homoscedasticity would be indicated by a random scatter of residuals without any clear pattern.
- **Scale-Location Plot (Spread-Location Plot)**: This plot shows the square root of the standardized residuals versus the fitted values. A horizontal line with equally spread points suggests homoscedasticity. If the plot shows a funnel shape (points spread out more as fitted values increase), this indicates heteroscedasticity.
- **Breusch-Pagan Test**: This statistical test assesses the presence of heteroscedasticity by checking if the residual variance is related to the fitted values or other predictors. A significant result suggests heteroscedasticity.
- **Levene's Test**: A test that compares the variances between groups (useful in grouped data). It tests the null hypothesis that the variances are equal across groups, with significant p-values indicating a violation of homoscedasticity.

Assumption	Test Methods
Linearity	Residual vs. Fitted Plot, Component Plus Residual (CERES) Plot, Polynomial Terms
Independence	Durbin-Watson Test, Plot of Residuals by Group, Random Effects Variance
Normality	Q-Q Plot, Histogram of Residuals, Shapiro-Wilk Test, Anderson-Darling Test
Homoscedasticity	Residual vs. Fitted Plot, Scale-Location Plot, Breusch-Pagan Test, Levene's Test

Table 3.1: Summary of Assumption Tests for Linear Mixed-Effects Models

3.8 A Complete Working Example

In order to better understand LMM, let's consider an example, in this case a clinical psychology study that investigates the effectiveness of two therapeutic interventions (CBT vs. Mindfulness) on reducing anxiety levels in patients over six months. Measurements of anxiety scores are taken at three time points: baseline (0 months), mid-point (3 months), and endpoint (6 months).

Patients are nested within therapists, meaning each therapist treats multiple patients. The study includes:

- Fixed effects: **Type of therapy** (CBT or Mindfulness), **time** (baseline, 3 months, 6 months), and their interaction.
- Random effects: **Therapist-level variability** (random intercepts for therapists).

The goal of the analysis is to assess whether the type of therapy and time have significant effects on anxiety scores while accounting for the clustering of patients within therapistss . . . but, before moving on, let's consider an important question . . . How is accounting for clustering of patients in therapists using LMM different from including therapists as an additional factor in regression or ANOVA?

They are fundamentally different approaches to handling grouped or hierarchical data. Here's a detailed comparison:

ANOVA with Therapists as a Fixed Factor

- In ANOVA, therapists are treated as a fixed factor, meaning each therapist is explicitly represented in the model.
- This approach estimates a separate effect (mean adjustment) for each therapist, similar to including dummy variables for each therapist.
- The number of parameters grows with the number of therapists, which can quickly become impractical if there are many therapists.
- Assumes that the levels of the therapist factor are exhaustive or of specific interest (e.g., a finite set of therapists for whom you want individual comparisons).
- Assumes homogeneity of variance across therapists, which may not reflect reality if therapists vary in effectiveness.
- Results are specific to the therapists included in the study and cannot be generalized to a broader population of therapists.

LMM with Therapists as a Random Effect

- LMM treats therapists as a random effect, assuming the therapists in the dataset represent a sample from a larger population of therapists.

- Instead of estimating a separate parameter for each therapist, the variability among therapists is modeled as a random variable with a mean of zero and a variance (σ_u^2).
- LMM accounts for the clustering of patients within therapists without explicitly modeling each therapist as a fixed factor, making it more scalable and generalizable.
- Captures the variability among therapists using a random-effects term, allowing for correlated observations within each therapist's group.
- Results generalize to the population of therapists, as the random-effects structure assumes therapists are sampled from a larger population.

Now that we've considered why LMM may be more appropriate to help address the primary research question in this case, let's move on with the full example ...

1. Set up the R environment

Before getting started with this example, let's make sure that all of the required R packages are installed and loaded into the R environment.

R Code

```
# Load necessary libraries
install.packages("lme4")  # For mixed-effects models
install.packages("ggplot2")  # For visualization
install.packages("lmerTest")  # For p-values in mixed models

library(lme4)
library(ggplot2)
library(lmerTest)
```

2. Simulate Data

Eventually, you will be using this example to conduct analyses with your own data, but for now, let's simulate some data to fit with our therapy study.

R Code

```
set.seed(123)
# Set parameters
n_patients <- 10          # Number of patients per therapist
n_therapists <- 5         # Number of therapists
time_points <- c(0, 3, 6) # Baseline, Mid-point, Endpoint

# Total number of observations
n_obs <- n_patients * n_therapists * length(time_points)

# Simulate therapist-level data
therapist_id <- rep(1:n_therapists, each = n_patients *
     length(time_points) / n_therapists)
therapist_effects <- rep(rnorm(n_therapists, mean = 0, sd = 2),
     each = n_patients * length(time_points) / n_therapists)

# Simulate patient-level data
therapy_type <- rep(c("CBT", "Mindfulness"), length.out = n_obs)
time <- rep(time_points, times = n_patients * n_therapists)
```

```
# Fixed effects: Therapy and time effects
cbt_effect <- ifelse(therapy_type == "CBT", -5, 0)
time_effect <- time * -0.8 # Decreasing anxiety over time
interaction_effect <- ifelse(therapy_type == "CBT", time * -0.8, 0)
     # Greater improvement for CBT

# Generate anxiety scores
residual_error <- rnorm(n_obs, mean = 0, sd = 5)
     # Residual error matches n_obs
anxiety_scores <- 50 + cbt_effect + time_effect + interaction_effect
      + therapist_effects + residual_error

# Create data frame
data <- data.frame(
  anxiety_scores = anxiety_scores,
  therapy_type = factor(therapy_type),
  time = factor(time, levels = time_points),
  therapist_id = factor(therapist_id)
)

# Display summary of the data
summary(data)
```

The above code will generate an R data frame containing simulated data from 50 patients nested in 10 therapists. The random effects of therapist are drawn from a random normal distribution with a mean of zero and a standard deviation of 5.

3. Visualize the Data Before modeling

As with ANY data analysis, it is important to generate some visualizations of your data before you start conducting your statistical analyses. The code below illustrates how you might review the raw data using a grouped longitudinal dot plot. This type of visualization is often used to depict changes in response variables over time across different groups or conditions. It could also be referred to as a time-series scatter plot by group.

R Code

```
# Plot anxiety scores over time by therapy type
ggplot(data, aes(x = time, y = anxiety_scores, color = therapy_type)) +
    geom_point(position = position_jitter(width = 0.1, height = 0)) +
    geom_smooth(method = "loess", se = FALSE) +
    labs(title = "Anxiety␣Scores␣Over␣Time␣by␣Therapy␣Type",
         x = "Time␣(Months)", y = "Anxiety␣Score") +
    theme_minimal()
```

4: Fit the Linear Mixed-Effects Model

In this analysis, we will fit a LMM to account for the hierarchical structure of the data, where patients are nested within therapists. The model includes random intercepts for therapists, which allow each therapist to have their own baseline anxiety level, reflecting unmeasured therapist-level differences that might influence patient outcomes. Fixed effects are included for therapy type (CBT vs. Mindfulness), time (baseline, mid-point, and endpoint), and their interaction. The main effects for therapy type and time capture the overall differences between therapies and the progression of anxiety scores over time,

respectively. The interaction term allows us to assess whether the rate of change in anxiety scores over time differs between the two therapy types. By including random intercepts, the LMM accounts for the clustering of patients within therapists, providing more accurate estimates of fixed effects and properly partitioning variance between therapists and patients.

R Code

```
# Fit the model
model <- lmer(anxiety_scores ~ therapy_type * time +
     (1 | therapist_id), data = data)

# Display summary
summary(model)
```

Code block 3.1 Fitting the LMM and displaying a summary of the result.

Example of a Grouped Longitudinal Dot Plot

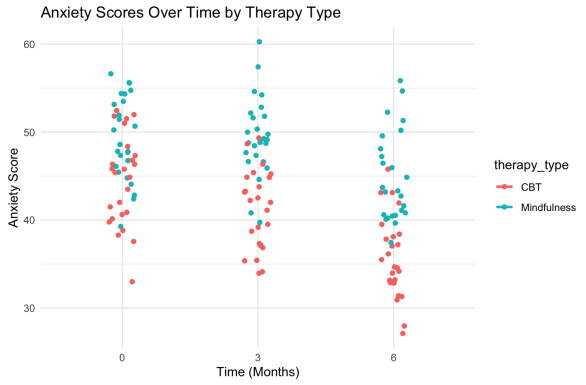


Figure 3.7. This plot depicts changes in anxiety over time across the CBT and Mindfullness conditions.

The output (i.e., summary) of this model fit will contain several sections summarizing measures of model fit and the fit statistics. Each section is illustrated and explained in the table below.

Output:
Linear mixed model fit by REML. t-tests use Satterthwaite's method ['lmerModLmerTest']
Formula: anxiety_scores ~ therapy_type * time + (1 | therapist_id)
Data: data

Explanation:
This part of the output provides key information about how the linear mixed model (LMM) was fit, how statistical tests were performed, and specifies the LMM formula.

- Note that the model was fit using REML (default). If you prefer to use maximum likelihood (ML), you can add the REML=false argument when fitting the model.
- The fixed effects include therapy type, time, and their interaction. The random effects term `(1 | therapist_id)` allows each therapist to have a random intercept, accounting for between-therapist variability.
- The fixed-effect coefficient significance is tested using Satterthwaite's approximation to calculate degrees of freedom. This method adjusts the degrees of freedom to account for the hierarchical structure of the data (e.g., clustering by therapists), providing more accurate p-values for fixed effects.
- The output also shows the class of the model object created by the lmer() function in the lmerTest package, which extends the functionality of the lme4 package to include p-value calculations for fixed effects.

Output:
REML criterion at convergence: 877.6

Explanation:
The REML criterion is a measure of model fit. Lower values indicate a better fit.

- The REML convergence criterion itself is not typically interpreted directly as "good" or "bad." Instead, it is a numerical value used to determine whether the optimization algorithm has successfully converged during the fitting process of a LMM.
- Lower values of the REML criterion indicate a better fit, but only in a relative sense. For example, when comparing nested models (e.g., using likelihood ratio tests), the model with the lower REML criterion is the better fit. The absolute value of the REML criterion depends on the scale of the outcome variable, the size of the dataset, and the complexity of the model (number of fixed and random effects).

Output:

Scaled residuals:	Min	1Q	Median	3Q	Max
	-2.31120	-0.73545	-0.05507	0.64579	2.36512

Explanation:
Scaled residuals provide a summary of the model's residuals (differences between observed and predicted values). They are standardized for interpretation, and extreme values may indicate outliers or model issues.

Output:

Random effects:	**Groups**	**Name**	**Variance**	**Std.Dev.**
	therapist_id	(Intercept)	0.5635	0.7507
		Residual	22.3566	4.7283

Number of obs: 150, groups: therapist_id, 5

Explanation:
The random intercept variance for therapists (`0.5635`) reflects differences in baseline anxiety scores across therapists. The residual variance (`22.3566`) represents unexplained within-patient variability in anxiety scores.
There are 150 total observations (patients across time points) nested within 5 therapist groups. This clustering is modeled using random effects.

Output:

Fixed effects:	Estimate	Std. Error	df	t value	Pr(>|t|)	
(Intercept)	44.47	1.003	48.067	44.322	< 2e-16	***
therapy_typeMindfulness	4.847	1.337	140.000	3.624	0.0004	***
time3	-3.374	1.337	140.000	-2.523	0.0127	*
time6	-8.805	1.337	140.000	-6.584	8.55e-10	***
therapy_typeMindfulness:time3	3.551	1.891	140.000	1.878	0.0625	.
therapy_typeMindfulness:time6	4.359	1.891	140.000	2.305	0.0226	*

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Explanation:

- The intercept (`47.476`) represents the baseline anxiety score for CBT at time 0. Therapy type and time effects capture differences in anxiety scores due to therapy type (Mindfulness vs. CBT) and progression over time. Significant effects (`time6`, $p < 0.001$) suggest lower anxiety scores at 6 months.
- In typical statistical output from models, like those produced by R, the "time0" category (often the baseline or reference level for a factor) is not explicitly listed in the fixed effects table. This happens because the reference level is implicitly represented by the intercept term. When you have categorical variables like therapy_type and time, the model automatically chooses one level of each factor as a reference. In this case, the model has probably chosen therapy_type with CBT and time0 as the reference levels.
- The interaction terms test whether the effect of time differs between therapy types. Non-significant results ($p > 0.05$) indicate no strong evidence that the change over time depends on therapy type.

Output:

Correlation of Fixed Effects:	(Intr)	thrp_M	time3	time6	th_M:3
thrp_typeMn	-0.666				
time3	-0.666	0.500			
time6	-0.666	0.500	0.500		
thrp_typeM:3	0.471	-0.707	-0.707	-0.354	
thrp_typeM:6	0.471	-0.707	-0.354	-0.707	0.500

Explanation:

- The correlation matrix shows relationships between fixed effect estimates. For example, `thrpy_typMn` (therapy type) is negatively correlated with the intercept (`-0.666`). High correlation between two fixed effects suggests that the two predictors share a lot of common information, potentially leading to multicollinearity (which can distort the interpretation of the coefficients).
- For example, thrp_typeMn and time3: The correlation here is 0.500, meaning there is a moderate positive relationship between the effect of therapy_typeMindfulness and time3. This suggests that the higher the effect of therapy_typeMindfulness, the stronger the effect of time3, and vice versa.
- High correlations between predictors (such as thrp_typeMn, time3, time6, etc.) might suggest collinearity issues. In your case, correlations like -0.666 between the intercept and the other variables, and 0.500 between some variables, suggest that some of the predictors share a fair amount of common variance. While not necessarily problematic, you should check whether these correlations lead to multicollinearity issues (which can inflate standard errors and make coefficient estimates unstable).

Table 3.2: Summary of the LMM fit

Reporting LMM Results

Once you have fit the LMM, it is important to consider how you will report the results. While there is no strict APA standard for reporting linear mixed models, it is important to provide enough detail about the model specification and results that the reader can evaluate the model specification, fit, and statistical results.

For example . . . "We conducted a linear mixed model analysis in R, with [dependent variable] as the outcome variable, [fixed effects] as fixed effects, and [random effects] as random effects. The model specification was: [DV] [fixed effects] + (1|random effects). The analysis revealed a significant effect of Therapy Type (β = 4.85, SE = 1.00, $t(140) = 3.62$, $p < .001$, 95% CI [2.23, 7.47]) and a significant Time × Therapy Type interaction at six months (β = 4.36, SE = 1.89, $t(140) = 2.31$, $p = .022$, 95% CI [0.65, 8.07]). The random effect of participant accounted for 1.2% of the total variance."

Note that the confidence intervals and variance accounted for (R^2) are not part of the default output in R. However, these values can easily be computed from the model estimated by the lme4 package using the `confint()` function.

$$95\%CI = \begin{cases} confint(model, method = "Wald") \text{\#For fixed effects only} \\ confint(model, method = "profile") \text{\#For both fixed and random effects} \\ confint(model, method = "boot") \text{\#Bootstrap method} \end{cases}$$

In order to compute the variance accounted for (R^2), you will likely need to install and load the "performance" package. Calling the `r2()` function will then yield the conditional and the marginal R^2 (see code block below). The conditional R^2 includes both fixed and random effects and the marginal R^2 only accounts for the fixed effects. The percent variance accounted for by only the random effects is the difference between the two, $R^2_{conditional} - R^2_{marginal}$.

R Code

```
# Install additional packages \% variance accounted for
install.package(''performance'')
library(performance)

# Get percent variance accounted for
> r2(model)
 Conditional R2: 0.513
 Marginal R2: 0.501

# Get 95\% confidence intervals
> confint(model, method="profile")
                                    2.5 \%     97.5 \%
.sig01                           0.0000000  2.0622077
.sigma                           4.1585789  5.2367328
(Intercept)                     42.5573209 46.3951265
therapy_typeMindfulness          2.2543649  7.4398386
time3                           -5.9670212 -0.7815476
time6                          -11.3975347 -6.2120611
therapy_typeMindfulness:time3   -0.1156872  7.2176799
therapy_typeMindfulness:time6    0.6924522  8.0258193
```

Code block 3.2 Getting the R^2 and 95% CIs from the LMM model.

3.9 Visualizing the Results of a LMM

Linear mixed effect models can be visualized in several effective ways to understand both fixed and random effects. Here are the key visualization approaches:

Geometric diagrams are to geometers what board and pieces are to chessmasters: visual aids, helpful but not indispensable.

— Richard J. Trudeau

4.1 Introduction to Graph Theory

Historical Significance

Graph theory's origins can be traced to 1736 when Leonhard Euler solved the famous Seven Bridges of Königsberg problem, marking the first formal proof in the field. By abstracting the physical layout of bridges and landmasses into vertices and edges, Euler created a new mathematical approach that would eventually revolutionize how we analyze networks and relationships. While the field developed slowly at first, it gained significant momentum in the 19th century when mathematicians like Arthur Cayley connected it to chemical compositions and James Joseph Sylvester introduced the term "graph" in 1878. The publication of the first textbook on graph theory by Dénes Kőnig in 1936 officially established it as a distinct branch of mathematics6. Today, graph theory serves as a cornerstone for solving complex problems across diverse fields, from computer science and biology to sociology and transportation networks5, demonstrating how a solution to a recreational puzzle evolved into a fundamental mathematical framework for understanding and optimizing interconnected systems.

Overview

Today, graph theory provides a mathematical framework for analyzing networks of interconnected entities, where nodes (also called vertices) represent discrete objects and edges represent the relationships between them. A graph G is formally defined as an ordered pair $G = (V, E)$, where V is a set of vertices and E is a set of edges connecting pairs of vertices. Graphs can be categorized into several fundamental types based on their edge properties. In **undirected graphs**, edges represent bidirectional relationships, such as friendship connections in social networks or pedestrian paths that can be traversed in both directions. **Directed graphs** (digraphs) contain edges with specific orientations, indicating one-way relationships like hyperlinks between web pages or parent-child relationships. Additionally, graphs can be weighted, where edges carry numerical values representing costs, distances, or strengths of relationships, or unweighted, where all connections are considered equal.

Several key metrics help characterize graph structure and identify important network features. **Degree** measures the number of connections a node has, while **centrality** metrics like betweenness and closeness identify nodes that are crucial for network flow or information spread. The **clustering coefficient** quantifies how tightly connected groups of nodes are, with values ranging from 0 (no clustering) to 1 (complete clustering). This measure is particularly important in social networks, where nodes tend to form dense local communities. The shortest path length between nodes helps evaluate network efficiency and can be used to identify optimal routes through the network, making it essential for applications in transportation and communication systems.

These fundamental concepts of graph theory find applications across diverse fields, from analyzing social media networks and optimizing transportation systems to modeling biological networks and designing computer algorithms. The versatility of graph theory as an analytical tool stems from its ability to represent and quantify complex relationships in a mathematically rigorous yet intuitively understandable way.

Applications of Graph Theory in Neuroscience

In neuroscience, brain networks are represented as graphs where nodes typically represent brain regions, electrodes, or voxels, while edges represent structural or functional connections between these elements. This abstraction allows researchers to quantify and analyze the organizational principles of neural systems at multiple scales, from microscopic neural circuits to macroscopic brain regions.

In structural connectivity analysis, diffusion MRI data is used to construct graphs where edges represent physical white matter connections between brain regions. These structural networks provide insights into the anatomical architecture of the brain, revealing important organizational principles such as the presence of highly connected hub regions and hierarchical modular structure. Graph metrics like clustering coefficient and path length help characterize the efficiency of information transfer through these anatomical networks, while centrality measures identify critical brain regions that serve as important relay stations for neural communication.

Functional connectivity studies, whether using fMRI or EEG data, create graphs based on statistical dependencies between neural signals recorded from different brain areas. In EEG studies, for example, electrodes serve as nodes while edges represent measures of synchronization or coherence between electrode signals. These functional

networks are dynamic, changing with cognitive states and tasks, and can reveal how different brain regions coordinate their activity during various cognitive processes. Graph theoretical analyses of these networks have revealed important properties like small-world organization, which balances efficient global communication with robust local processing, and hub regions that play crucial roles in integrating information across different functional modules.

Recent advances in the field have moved beyond static network analyses to study dynamic changes in graph properties over time, particularly in EEG and MEG studies where temporal resolution is high. These dynamic network analyses have revealed how brain network organization flexibly adapts to changing cognitive demands and how disruptions in these dynamic patterns may relate to various neurological and psychiatric conditions. The application of graph theory to neuroscience has thus provided not only powerful analytical tools but also new conceptual frameworks for understanding how the brain's structural and functional organization supports cognitive function.

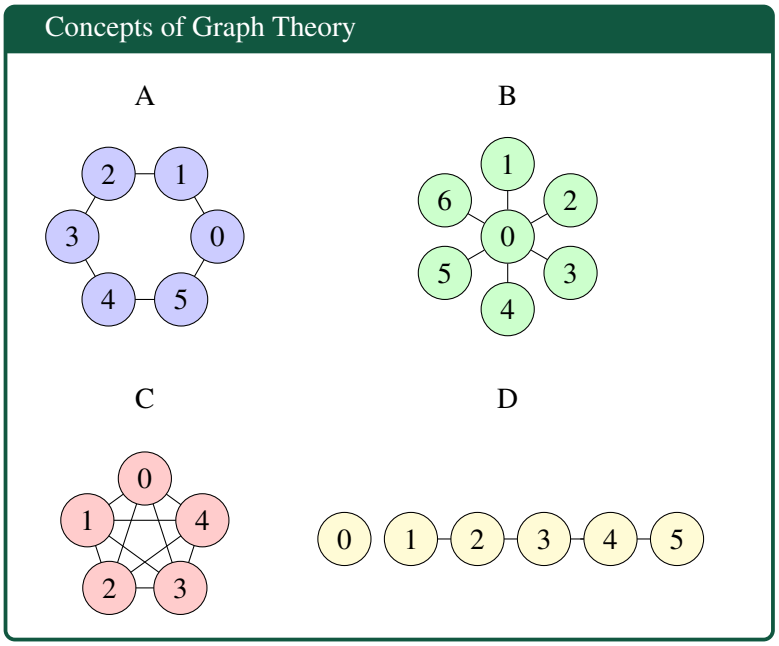


Figure 4.1: Graph theory concepts illustrated: (A) The cycle graph demonstrates degree, where each vertex has exactly two connections, showing uniform degree distribution. (B) The star graph highlights centrality, with node 0 having maximum centrality as it connects to all other nodes. (C) The hexagon graph shows maximum clustering coefficient, where all possible connections between neighbors exist. (D) The path graph illustrates shortest path concepts, where the distance between any two nodes is the minimum number of edges that must be traversed.

Applications of Graph Theory in Psychology

Graph theory also provides powerful analytical tools for understanding psychological and social phenomena across multiple scales. In the clinical domain, graph theory enables the construction of symptom networks where nodes represent individual symptoms and edges represent their interactions or correlations. This network approach has transformed how we conceptualize mental disorders - rather than viewing them as underlying diseases that cause symptoms, we can analyze them as systems of interacting symptoms. For instance, in depression, insomnia might lead to fatigue, which leads to reduced activity, which in turn reinforces depressive thoughts. By identifying central symptoms that trigger cascading effects, clinicians can target interventions more effectively.

Graph theoretical approaches also inform treatment planning and monitoring. By tracking changes in symptom networks over time, clinicians can assess treatment effectiveness and identify early warning signs of relapse. This dynamic network perspective helps explain why some symptoms are more resistant to treatment than others and why targeting certain symptoms might have cascading beneficial effects throughout the network. The ability to quantify network properties like centrality and clustering has provided new metrics for assessing therapeutic progress and predicting treatment outcomes. These applications demonstrate how graph theory has evolved from a mathematical framework into an essential tool for understanding human behavior and mental health, providing both theoretical insights and practical clinical applications.

In social psychology, graph theory has become instrumental in analyzing group dynamics and social networks. Researchers use these methods to map relationships between individuals (nodes) and their social connections (edges),

revealing important patterns in human interaction. This approach has helped identify how information spreads through social networks, how opinion leaders emerge, and how social support structures function. Social network analysis has revealed important principles about group formation, social influence, and community structure that weren't visible through traditional research methods.

4.2 Conducting Graph Theoretical Analyses

Network data suitable for graph theory analysis can come from many sources, including social connections, neural recordings, communication patterns, or any system with discrete elements that interact. The key is identifying meaningful nodes (the elements) and edges (their relationships). For example, in EEG data, nodes might be electrodes while edges represent functional connectivity between brain regions. Data are typically organized into an adjacency matrix (showing which nodes connect to each other) but can also be organized into an edge list (explicitly listing all connections).

Binary Adjacency Matrix

	A	B	C	D
A	0	1	1	0
B				
C				
D				

Weighted Adjacency Matrix

	A	B	C	D
A	0.0	0.8	0.4	0.0
B				
C				
D				

Binary Edge List

A – B
A – C
B – D
C – D

Weighted Edge List

A – B (0.8)
A – C (0.4)
B – D (0.6)
C – D (0.7)

In EEG research, each electrode recording brain activity serves as a node in the network. The relationships between these nodes are quantified through various measures of functional connectivity. These connections can be measured through phase locking values, which capture the consistency of phase relationships between signals from different electrodes, correlation coefficients that measure the linear relationship between electrode signals, or more sophisticated measures like Granger causality that assess directional influences between brain regions. This approach allows researchers to understand how different brain regions communicate and coordinate during various cognitive tasks or states.

In clinical settings, network analysis provides a framework for understanding how psychological symptoms interact and influence each other. Nodes in these networks represent individual symptoms or psychological variables, while edges capture their relationships. These relationships can be established through various means: symptoms that frequently co-occur, patients' reports of how one symptom influences another, or patterns observed over time in longitudinal studies. This approach has revolutionized our understanding of mental disorders, moving from a disease model to viewing psychological disorders as systems of interacting symptoms.

Social psychology utilizes network analysis to understand human relationships and group dynamics. In these networks, nodes typically represent individuals or social entities (like groups or organizations), while edges represent various types of social relationships. These connections can represent direct communication patterns, friendship ties, or shared group memberships. The strength of these relationships can be represented either as binary connections (present/absent) or weighted edges that indicate the intensity or frequency of interaction. This framework helps researchers understand social influence, group formation, and information flow within communities.

Data Processing Steps

Before analysis, several preprocessing steps are crucial. First, check for missing or impossible connections in your network data. Consider whether to include weighted edges (showing connection strength) or just binary connections.

Think carefully about thresholding - deciding how strong a relationship must be to count as an edge. For time-series data like EEG, you'll need to address noise and artifacts that could create spurious connections. It's also important to consider whether your network should be directed (connections have a specific direction) or undirected.

Your adjacency matrix should be sparse. A sparse matrix is one in which most elements are zero, with only a small percentage of non-zero entries. In graph theoretical analysis, sparse matrices are crucial because they better reflect real-world networks where nodes typically connect to only a small subset of other nodes rather than everything connecting to everything else. Using sparse matrices provides several key advantages, requiring less computational resources and memory for storage and processing, enabling faster algorithm execution, and often better representing natural phenomena like neural networks where excessive connectivity would be energetically costly and potentially noisy. In neuroscience specifically, sparse connectivity helps support efficient information processing by promoting better pattern separation and more effective stimulus encoding. The sparsity principle is particularly important when analyzing large-scale networks, as it helps maintain computational tractability while preserving the most meaningful connections in the network.

Thresholding is a crucial step in creating sparse networks from correlation matrices for graph theoretical analysis. This process involves making deliberate choices about which connections to maintain and which to eliminate, ultimately creating a more meaningful and computationally manageable network structure. The first step involves selecting a threshold value to determine which correlations are strong enough to be considered meaningful connections. This threshold might be based on statistical significance (e.g., p-values), correlation strength (e.g., keeping only correlations above 0.3 or 0.5), or proportional thresholding (keeping the top percentage of strongest connections). Connections falling below this threshold are set to zero, effectively removing weak or potentially spurious relationships from the network. This step is crucial because weak correlations may represent noise rather than true functional relationships.

The next step involves removing self-correlations by setting the diagonal elements of the matrix to zero. Self-correlations (correlations of a node with itself) are always 1.0 and don't provide meaningful information about network structure. After thresholding and removing self-correlations, researchers must decide whether to maintain the weighted edges (keeping the actual correlation values) or convert to binary connections (where any surviving connection is set to 1). Binary networks are simpler to analyze and interpret, but weighted networks preserve more detailed information about connection strengths. This choice should be guided by the research questions and the specific requirements of the analysis methods being used.

The resulting sparse network retains only the most meaningful connections while eliminating potentially spurious relationships, making it both more interpretable and computationally tractable. This process is particularly important when analyzing large networks, as it helps focus the analysis on the most robust and meaningful patterns in the data.

Analysis Methods

Start analysis with basic graph metrics like degree (number of connections per node), clustering coefficient (how interconnected neighboring nodes are), and various centrality measures (identifying important nodes). These graph metrics provide essential tools for quantifying and understanding network properties.

Clustering Coefficient

The clustering coefficient measures how much nodes tend to cluster together in a network. There are two versions of clustering coefficient; the global and the local. The global coefficient gives an overall indication of the clustering in the network, whereas the local coefficient gives an indication of the embeddedness of single nodes within a cluster.

The global clustering coefficient is based on triplets of nodes. A triplet consists of three connected nodes. A triangle therefore includes three closed triplets, one centered on each of the nodes (n.b. this means the three triplets in a triangle come from overlapping selections of nodes). The global clustering coefficient is the number of closed triplets (or 3 x triangles) over the total number of triplets (both open and closed). The first attempt to measure it was made by Luce and Perry (1949). This measure gives an indication of the clustering in the whole network (global), and can be applied to both undirected and directed networks.

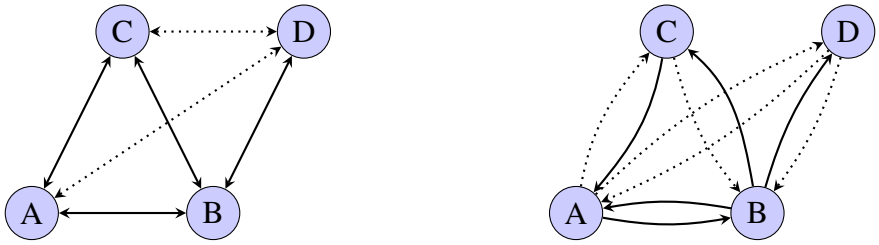
Before delving into the local clustering coefficient, let's formalize the definition of the components of a graph. A graph $G = (V, E)$ formally consists of a set of vertices V (nodes) and a set of edges E between them. An edge

e_{ij} connects vertex v_i with vertex v_j. The neighborhood N_i for a vertex v_i is defined as its immediately connected neighbors as follows: $N_i = \{v_j : e_{ij} \in E \vee e_{ji} \in E\}$. In simpler terms, the neighborhood N_i is the set of vertices j such that there is an edge from node i to node j or from node j to node i in the set of all edges E.

For a directed graph, e_{ij} is distinct from e_{ji}, and therefore for each neighborhood N_i there are $k_i(k_i - 1)$ links that could exist among the vertices within the neighborhood (k_i is the number of neighbors of a vertex). Thus, the local clustering coefficient for directed graphs is given as $C_i = \frac{|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i-1)}$.

An undirected graph has the property that e_{ij} and e_{ji} are considered identical. Therefore, if a vertex v_i has k_i neighbors, $\frac{k_i(k_i-1)}{2}$ edges could exist among the vertices within the neighborhood. Thus, the local clustering coefficient for undirected graphs can be defined as $C_i = \frac{2|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i-1)}$.

The Clustering Coefficient



$C_A = \frac{2|1|}{2(2-1)} = \frac{2}{2} = 1.0$
$C_B = \frac{2|1|}{3(3-1)} = \frac{2}{6} = 0.33$

$C_A = \frac{1}{2(2-1)} = \frac{1}{2} = 0.5$
$C_B = \frac{1}{3(3-1)} = \frac{1}{6} = 0.16$

Figure 4.2: Calculating the clustering coefficient for directed and undirected graphs.

Path Length

Path length measures the efficiency of information flow through the network. The shortest path length between two nodes is the minimum number of edges that must be traversed to get from one node to another. The characteristic path length of a network is the average shortest path length between all possible pairs of nodes. In weighted networks, path lengths consider edge weights, where stronger connections (higher weights) typically represent shorter effective distances. Centrality Measures Centrality identifies influential nodes using several metrics: Degree centrality: Simply counts the number of connections a node has Betweenness centrality: Measures how often a node lies on shortest paths between other nodes Closeness centrality: Calculates how close a node is to all other nodes in the network Eigenvector centrality: Considers both the quantity and quality of connections, where connections to high-scoring nodes contribute more Community Detection Community detection algorithms identify groups of nodes that are more densely connected to each other than to the rest of the network. Common approaches include: Modularity optimization: Maximizes the difference between actual and expected connections within communities Hierarchical clustering: Groups nodes based on similarity or connection strength Spectral clustering: Uses eigenvalues of the network's adjacency matrix to partition nodes into communities These metrics together provide a comprehensive characterization of network structure and organization, revealing important patterns in complex systems.

More advanced analyses might examine community structure, path lengths between nodes, or network efficiency. Always validate your results against null models to ensure findings are meaningful. Visualization tools can help interpret results - consider force-directed layouts for small networks or matrix representations for larger ones. Practical Implementation Begin with simple examples where students can calculate metrics by hand to build intuition. Then progress to using established software packages like NetworkX (Python) or igraph (R) for larger analyses.

Emphasize the importance of understanding what each metric means in the context of your specific data. For instance, high clustering in social networks might indicate tight-knit communities, while in neural networks it might suggest efficient local processing. Remember that graph theory is just a tool - the key is using it to answer meaningful questions about your system of interest. Start simple and gradually build complexity as students gain confidence with the concepts and techniques.