Graphical Representation of Longitudinal Data

Sophie Nicola Jones
Department of Statistics
The University of Auckland

Supervisor: Chris Wild

A dissertation submitted in partial fulfilment of the requirements for the degree of BSc(Hons) in Statistics, The University of Auckland, 2020.

Abstract

This project set out to add easy-to-use longitudinal data visualisation methods that could handle large or complex datasets to the iNZight software package. Longitudinal data requires unique methods to handle multiple responses over time from the same study unit but still suffers from problems that are common in other areas of data visualisation such as overprinting. The project places emphasis on understanding the best methods for visualising these complex datasets and producing clear, understandable graphics. The development of these plot ideas has led to the development of a dedicated Longitudinal Data Module for iNZight, an example Shiny web application, and an R package, `longZight` which provides functions for the iNZight module but also generic tools that allow users to create and customise their own plots. Additional commentary is provided on issues that developers may face when working with iNZight modules and R packaging based on experiences during this project.

Acknowledgements

I wish to thank my project supervisor, Chris Wild, for the support and encouragement for this project. Without his passion, iNZight would not exist. Additionally Tom Elliot, lead developer for iNZight for his valuable advice and the assistance given in debugging several problems with the iNZight module, and finally Avinesh Pillai from the Statistical Consulting Centre and Growing Up in New Zealand for his assistance, and making me realise the impact this project would have.

I'd also like to thank my family, friends, and the staff from the Department of Statistics for their support and encouragement as I navigated both Honours and significant life changes this year. Kia kaha, thank you!

Finally, I dedicate this dissertation to my late father, Ivan who passed in 2018. He was always backed my academic aspirations but sadly didn't get the opportunity to see the hard work pay off.

Contents

List of Code Listings

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Intent

The project is designed to investigate appropriate methods of graphically representing longtitudinal data in a clear and concise manner. The investigation is to then guide the development of an R package that can be integrated with iNZight (by the means of a Longitudinal Data Visualisation module), or other tools such as Shiny, to allow users a simple method of creating their own longitudinal visualisations.

This dissertation concentrates on the ideas behind, and the development process of, the software produced. Demonstrations of the resulting applications and and their capabilities will be demonstrated in the project talk.

1.2 Definition of Longitudinal Data

Although a variety of descriptions for longitudinal data exist – the broad definition been any data with a time element – a focus for this project was placed on a specific unambiguous definition.

Longitudinal data involves repeated individual-level observations taken on a set of study units over time (in general, individuals can begin, and cease participation in a study at any time, and study units may be replaced over time - a rotating panel design). For the computer representations of longitudinal data, this project states that the data must hold the following characteristics:

- Each study unit is assigned a unique identifier, index or key. Additionally, there is a one-to-one map between study units and identifiers (a study unit cannot have two identifiers and there is a restriction that they must not be shared between units).

- Each observation is associated to the unique identifier for the study unit the observation was collected from
- Each observation is associated to the time it was recorded

It is important to note that the definition does not dictate a particular data-storage format (for instance long-form or wide-form are equally acceptable). The main condition is that each observation must be unambiguously mapped to a point of time and an individual identifier.

1.2.1 Time in Longitudinal Data

Time may be either continuous or categorical in longitudinal data depending on the nature of time for the dataset.

The anticipated common use of the completed iNZight visualisation module is for longitudinal surveys and/or panel datasets where time is recorded as categorical time points. In these situations the time is often recorded as the ordinal count or year in which a specific iteration of a survey was responded to, rather than a specific time on a continuous scale. Often these categorical measures of time are referred to as survey waves or time points.

However, we must also cater for datasets where the precision of time matters (i.e. the timing of state transitions) and is stored as continuous measurements from a chosen epoch.

1.2.2 Example Longitudinal Data

Table 1.1 below, shows an example long-form dataset that complies with this definition, but other forms of data storage (such as wide formats) are equally compatible with the core characteristics described.

ID <integer>	**WAVE** <factor>	**Industry** <factor>	**Salary** <integer>
43	2000	Communications	54,000
43	2005	Finance	80,000
44	2000	Communications	50,000
44	2005	Communications	64,000

Table 1.1: **Example Longitudinal Dataset** - A synthetic example of compliant longitudinal dataset

1.3 Use of iNZight

iNZight is a data analytics and visualisation program developed at the University of Auckland and provides a variety of specialised tools for a variety of data types including multiple response, GIS/mapping, surveys and time series. The iNZight application is used by a wide variety of groups such as students, journalists, public servants, and researchers.

By creating a module for iNZight to perform longitudinal data exploration, existing users will have the option to enter the module and using the Graphical User Interface (GUI) select the variables they wish to visualise. The existence of a GUI will also provide a step-by-step process that enforces the definition of longitudinal data from Section 1.2 when setting up the data. The advantages of building on top of the pre-existing iNZight system reduces the complexity of creating a new GUI but also allows users to harness the powerful data manipulation tools already built into the application in the event that the users' data is not already in the expected form.

1.4 Data

Several datasets have been used during this project to provide applied examples of longitudinal visualisations. Provided in this section are descriptions of the datasets that feature heavily in the project.

1.4.1 Chinese Health and Nutritional Survey (CHNS)

Data from the Chinese Health and Nutritional Survey (CHNS) a longitudinal study managed by the University of North Carolina at Chapel Hill's Carolina Population Centre was used extensively. The primary attraction of the CHNS datasets was the well formatted structure and documentation of the datasets which made the initial stages of the project, including testing plotting ideas much easier. Additionally, the CHNS data is quite broad covering many topics and a wide range of time - for some topics data was available from 1989 to 2015 (waves were typically separated by 2-3 years).

Focus was placed on the *Individual Roster dataset* (`rst_12`), and the *Individual Health Care dataset* (`hlth_12`). Both of these datasets have common variables that are of interest from the previous definition of Longtudinal Data (Section 1.2):

Variable Name	Description
`IDind`	Individual Identifier Variable
`WAVE`	Year the record was observed during

Table 1.2: The Identifier and Time variables in the CHNS datasets

1.4.1.1 Individual Roster Dataset

The Individual Roster dataset provided 180,582 observations across 67 variables.

The **`A5E`** variable relates to a question in which respondents were asked if they were "Living in the home?", the response was recorded as: Missing, Unknown or Invalid, "Yes, still lives in household" (1), "No, gone to school" (2), "No, military service" (3), "No, sought employment elsewhere" (4), "No, gone abroad" (5) and "No, Other" (6). This variable will feature heavily in examples of Alluvial and Lasagna plots as they do assist in showing how these plots can enable storytelling, even in the absence of statistical analysis.

1.4.1.2 Individual Health Care Dataset

The Individual Health Care Dataset provided 127,761 observations across 78 variables.

We were most interested in this dataset was the **`M23`** variable which relates to a question that asked if respondents had "been sick or injured in [the] last 4 weeks?" with the response options of Missing, No (0), Yes (1) or Unknown (9). The use of this variable provided a very simple dataset with only several options allowing the creation of simple Alluvial and Lasagna plots for initial testing and concept building that were not visually or computationally complex.

1.4.2 Wages data from National Longitudinal Survey of Youth (NLSY)

Data describing hourly wages of workers from the National Longitudinal Survey of Youth (NLSY) was provided by the `brolgar`[17] package. The dataset provided 6,402 observations from 888 subjects across 8 variables.

Variable Name	**Description**
`id`	Individual Identifier Variable
`xp` (numeric)	Amount of time/experience in the workforce at time data was collected
`xp_since_ged` (numeric)	Amount of time/experience in the workforce at time data was collected (measured from time a GED was obtained)
`ln_wages` (numeric)	Natural log value of inflation adjusted wages (normalised to 1990)
`ged` (binary)	If the subject had obtained a graduate equivalency diploma
`black` (bianry)	If the subject's race was recorded as black
`hispanic` (bianry)	If the subject's race was recorded as hispanic
`high_grade` (discrete/ordinal)	The last grade level of schooling attended by the subject

Table 1.3: Variables as provided in the `brolgar`[17] Wages Dataset

The numeric nature of **`xp`** and **`xp_since_ged`** which meet the defined criteria in Section 1.2 for measurements of time - a measurement from an epoch (birth or obtaining a Graduate Equivalency Diploma respectively) and the numeric nature of the response variable **`ln_wages`** mean that the dataset was applicable to Spaghetti plots.

1.4.3 Synthetic Transitional Dataset

In the absence of readily available data that had a categorical response but a continuous time variable, a synthetic dataset was created with the purposes of matching the following criteria:

- Each individual followed a predictable progression through states (i.e. $A \rightarrow B \rightarrow C$)
- The time was continuous, and individuals did not change states at predictable intervals, allowing some individuals to progress multiple states in the same time span as another individual may progress only one state.

The primary purpose of this dataset was to investigate alternatives to Lasagna plots with continuous responses, and to investigate mechanisms to overcome overprinting in traditional Spaghetti plots.

The code used to generate the synthetic dataset is provided in Code A.2 (Appendix A.2).

1.5 Existing Work

The most common visualisation methods seen in the public domain are plots that have been colloquially named after the pasta dishes Spaghetti and Lasagna due to their appearances. Additionally, transitional diagrams and Sankey plots (also known as Alluvial plots) were found to be commonplace.

1.5.1 Spaghetti Plots

Spaghetti plots generally take the form of a line plot with individual lines for each study unit displaying the trajectory across time points. A simplistic example, based on the dataset in Code A.1, is provided below as Figure 1.1. In this example there are only three study units, with a y-value that was measured at the three points of time (in this example all units were observed at the same time points). Each unit has their observations joined by a line following the ordered progression of time, each line segment forms the trajectory of the subject across time. For this example, points at each observation were added to demonstrate the construction of the plot and would generally be excluded from Spaghetti plots due to the noise they add.

It is important to note that while this plot is relatively easy to read, primarily due to the limited number of observations, other similar plots can become unwieldy and difficult to read such as Figure 1.2. The overprinting of lines due to the increase in the number of observations results in a plot that generally speaking begins to resemble a lump of spaghetti. Another common cause for overprinting in Spaghetti plots is when one or both of the axis variables is categorical in nature such as Figure 1.3 which uses the Synthetic Transitional Dataset described in Section 1.4.3.

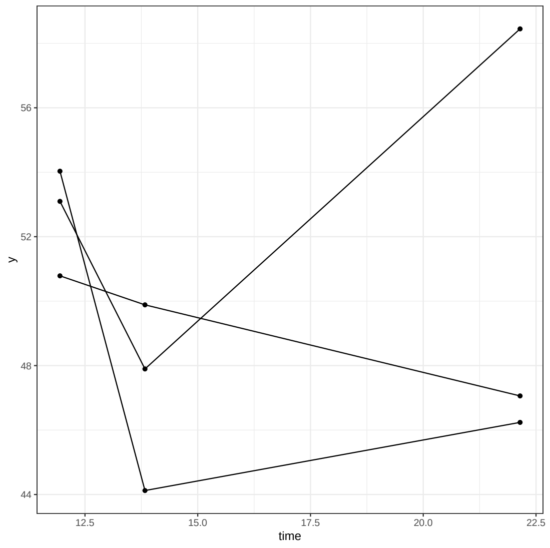


Figure 1.1: Example Spaghetti Plot; Data from Listing A.1

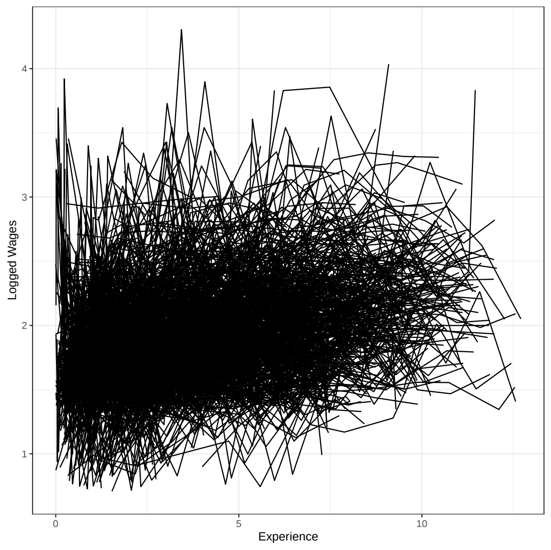


Figure 1.2: Spaghetti Plot from Brolgar Wages Dataset (Section 1.4.2)

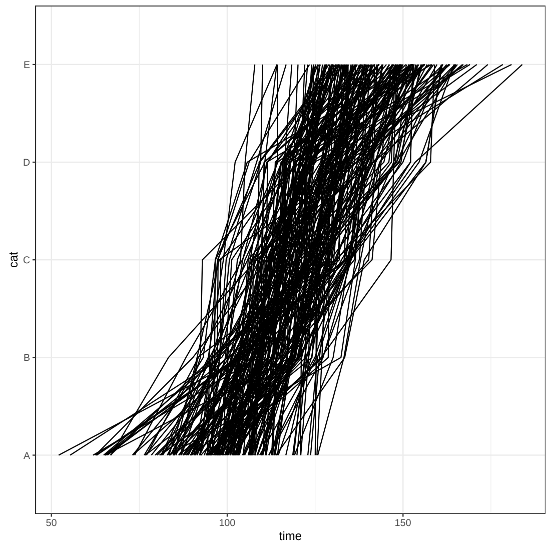


Figure 1.3: Spaghetti Plot Using the Synthetic Transitional Dataset (Section 1.4.3)

1.5.2 Lasagna Plots

A common method of displaying categorical response data has been the Lasagna plot, also frequently called a Fettucine plot, or a strip-plot. The names for the plot derive from the use of coloured parallel strips or blocks across the plots' x-axis, which each strip represented a set number of study units, or a proportion of study units, that match the common criteria.

While several packages exist for creating Lasagna plots, the most common has been the `lasagnar`[16] package, which provides tools for building Lasagna plots for both categorical and continuous time datasets as evidenced in the package's vignettes hosted on its GitHub repository[1]. The use of the `lasagnar` package for this project was discounted at an early stage as the code appeared to be now unmaintained and dated. Most of the plots are still generated using the R `graphics` package, while the focus of iNZight has begun moving to the use of `ggplot2`[19]-generated plots.

However, the use of `lasagnar` to produce Lasagna plots have been used with great success, prior to the commencement of this project, I had the privilege to see Victoria King present her work to date on understanding sleep cycles in pregnant women, during the presentation several `lasagnar`-based plots where presented with different sorting mechanisms. The sorting mechanisms used help reduce the problems created by a large quantity of observations, and the use of continuous (measured time) in her data.

An example of a Lasagna plot (generated using the `lasagnar` package, with example

[1]`https://github.com/swihart/lasagnar`

code) is provided below in Figure 1.4.

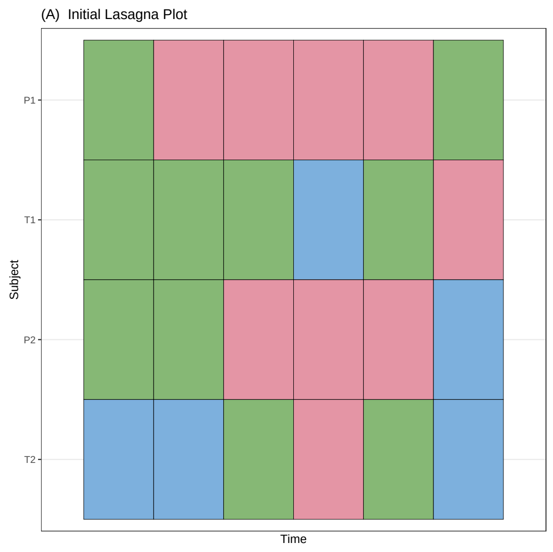


Figure 1.4: Lasagna plot generated with `lasagnar`, using provided example code

1.5.3 Transitional Diagrams

1.5.3.1 SLIDER Transitional Diagrams

A group of French researchers, Commenges, Pistre & Cura[4] developed the SLIDER (Software for LongItudinal Data Exploration with R) application as an alternative method for presenting data that features both categorical time and response. At an initial glance from reading the journal article it appeared the application made excellent plots for conveying the rates of transitions (slides) between levels of a response category through time. This was achieved by using the thickness of lines to convey the relative proportions of how many study units transitioned between states. I managed to replicate, using their software, a SLIDER plot similar to that presented in Figure 5 of [4] using a reduced version the Individual Roster Data from the CHNS (Section 1.4.1.1) as Figure 1.5.

The manipulations on the data used in Figure 1.5 had resulted in the exclusion of all individuals that had responded "Yes, still lives at home" in all of the last 4 waves of the survey. This version of the dataset was created to ease testing Alluvial and Lasagna plots. We noticed that when we included a fuller version of the CHNS Individual Roster dataset that the transitional diagrams became unclear (see Figure 1.6), in particular, the default settings for generating slider plots resulted in what can only be described as "a black blob on a screen" due to the heavy weighting of the "Yes" response category on the total frequencies/proportions.

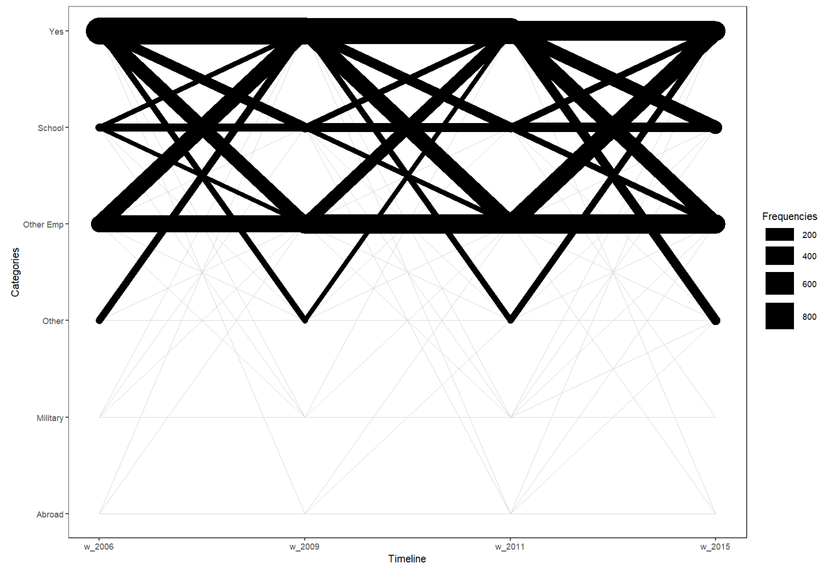


Figure 1.5: Plot generated with the SLIDER Shiny application with 4 Waves of CHNS Individual Roster Data (Reduced Dataset)

This results in an overprinting problem similar to other plot types discussed in this project.

While the software does provide some controls to the graphical parameters by way of a "*Threshold*" and "*Minimal thickness*" slider control, I feel that these methods are not suitable and can be misleading to users of the plots. These settings were required for the production of both Figures 1.5 & 1.6 and apart from the thicker line at the top and slightly thinner lines in the middle section, I feel the two plots could be mistaken as telling the same story. The biggest hint as to what is happening is the Axis, which suggests alongside the source code, that the threshold results in a log-like transformation which means large differences in frequencies are scaled down.

Additionally, the usefulness is restricted as, due to overwhelming data in the Yes/School/Other Employment categories, the patterns for people moving abroad or to the Military are not clear, and subtle changes in frequencies is not immediately clear. As an example, it is barely noticeable that there appears to be a decrease in frequencies for the "Yes, still lives at home" category between 2011 and 2015.

Overall, while the SLIDER plots do appear to be an excellent concept, the usefulness of them for conveying transition states is not as immediately clear as Alluvial and Lasagnar plots that were evaluated in parallel. Finally, there are concerns that the code has not been maintained since the original publication in 2014 and it's resulting maintainability, and that the AGPL-3 license may cause complexities in licensing if it were bundled or included in the GPL-3 licensed iNZight.

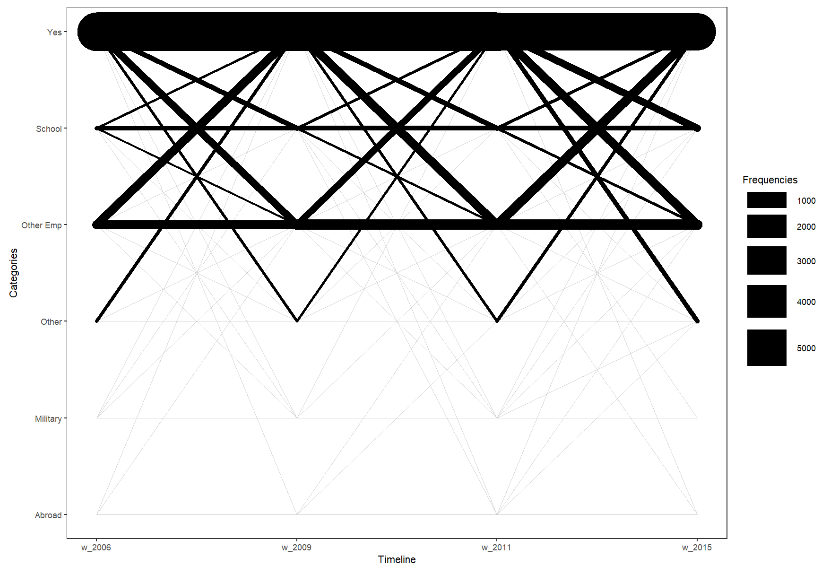


Figure 1.6: Plot generated with the SLIDER Shiny application with 4 Waves of CHNS Individual Roster Data

1.5.3.2 Alluvial/Sankey Plots

Another option that became apparent for the project was the use of Alluvial plots (sometimes referred to by the name Sankey) as a means of presenting changes in state through discrete time periods. The primary inspiration for the storytelling nature of Alluvial plots for time-domain data was from a study investigating how fire and flood events affected the composition of the floodplains in Southeastern British Columbia[10]. In the Alluvial plot presented in this article, the individual flows were coloured by the primary reason for the change in cover type (for instance fire, logging, etc) and it was possible to build a picture of many years of habitat changes from the single plot.

The `ggalluvial`[2] package was found in the CRAN repository and while it has a primary focus on non-longitudinal uses, such as process flows, the package appeared to be suitable for longitudinal data, which was confirmed with testing using the Individual Health Care dataset 1.4.1.2 – as can be seen in Figure 1.7.

One current disadvantage of the `ggalluvial` package is that while it does use `ggplot2`[19] routines, to achieve the flows between time periods, custom geometries were created by the package author and these are yet to be supported in the `plotly`[15] package, so the `plotly` conversion of these Alluvial plots for interactive visualisations is, at present, not possible.

As observed with Spaghetti and Lasagna plots, large quantities of data and/or categories can mean that the flows can be difficult to follow.

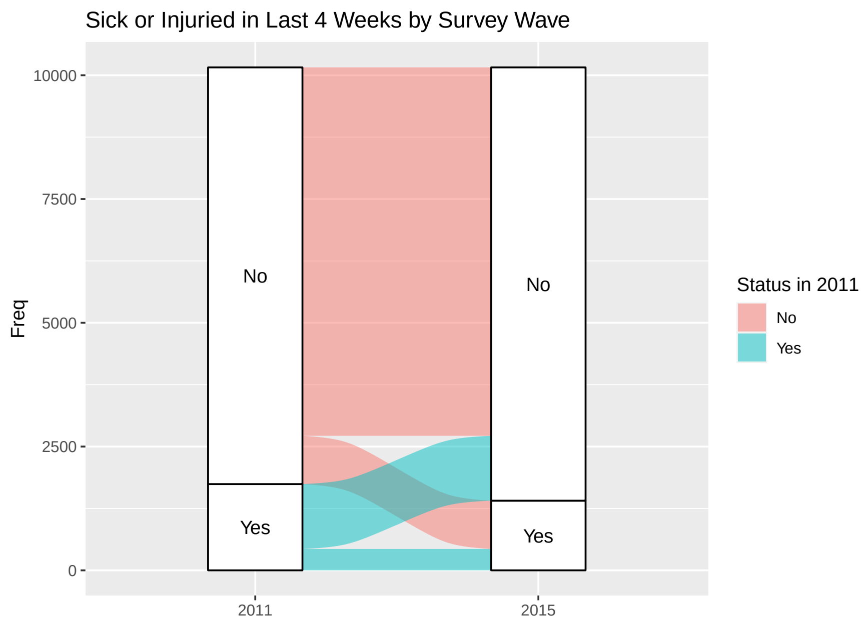


Figure 1.7: Example Alluvial Plot using CHNS Individual Health Care dataset

1.5.4 Additional Software

1.5.4.1 `brolgar`

I was alerted by my project supervisor to the existence of the `brolgar`[17] R package in the early stages of the project. Brolgar which stands for "BRowse Over Longitudinal data Graphically and Analytically in R" is currently in development by Nicholas Tierney, Di Cook & Tania Prvan. At this stage, `brolgar` is not a direct data visualisation tool but rather provides a multitude of helper functions that can be used to manipulate longitudinal datasets for the purposes of exploration.

A set of helper functions included in `brolgar` that were identified as potentially useful in this project was a set of functions that use the `fabletools`[12] `features()` method to identify and extract features from longitudinal data, such features include monotonic observations (observations where the combined trajectories are generally increasing or decreasing) and observations that are mostly similar.

The use of such feature and helper functions will be discussed further in Section 2.3.3.

One potential issue that may arise from the use of `brolgar`, especially with regards to the development of my `longZight` R package (Section 4.2) is the fact that `brolgar` does not yet exist on the Comprehensive R Archive Network (CRAN; the standard R package distribution system) which will inhibit the release of the R package created during this project if `brolgar` features too heavily. The release of the iNZight module should not be affected as iNZight hosts its own package repository and requirements such as `brolgar` can be added there prior to the release on CRAN. Finally, unlike other packages and software mentioned, `brolgar` also has an advantage in that it is presently under active development.

Chapter 2

Plot Development

2.1 Overall Requirements

Given the key outcome of the project is to create a data visualisation module for iNZight, it is important to ensure that plotting options are made available for all combinations of data types for both the selected time variable and response variable. The benefit of this is that the user cannot select a combination of variables that will produce an error or no visualisation, and is congruent with the overarching principle of iNZight that it is easy to use.

To guide the discussion of suitable plot types, the combinations of variables can be viewed in terms of the four quadrants in Table 2.1.

		Time	
		Categorical	Continuous
Response	Categorical	Quadrant 1	Quadrant 2
	Continuous	Quadrant 3	Quadrant 4

Table 2.1: Table showing the types of longitudinal visualisations that must be considered

2.1.1 Quadrant 1 – Categorical Time and Response

The first quadrant, dealing with categorical time and response variables (or values that can be coerced as such) was the easiest to populate with potential plotting mechanisms. Both Alluvial and Lasagna plots have been shown to be suitable to convey information and state transitions clearly. However, both do have handicaps with large quantities of observations that have unbalanced distributions between categories. Additional mechanisms must be considered to help retain some usefulness when this occurs.

Additionally, with large numbers of individuals, problems may occur when attempting to use tools such as `plotly`[15] to create interactive graphics as it is easy to strain the plotting mechanisms if there are too many objects to be displayed.

These issues are discussed further in Sections 2.3.1 & 2.3.2 for Alluvial and Lasagna plots respectively.

2.1.2 Quadrant 2 – Continuous Time and Categorical Response

By design, Alluvial plots are not suitable for handling continuous time as the flows must generally stop at fixed intermediary points to be aggregated. However, Lasagna plots, as discussed in Section 1.5.2, are suitable for conveying a categorical response with continuous time as evidenced in Victoria King's SatRday Auckland conference presentation in February 2020.

Ms King had previously experimented with Spaghetti plots for virtualisation her ECG sleep data (monitoring the progression and time in each state of sleep during a persons' pregnancy), but as independently found during this project, the use of Spaghetti plots is not appropriate due to issues with overplotting.

However, the data presented by Ms King followed a typical progression of states (effectively an ordered categorical response). The predictible progression between states meant she was able to utilise data sorting techniques to arrange the Lasagna plots in a readable manner that enhanced the necessary storytelling abilities. The development of these ideas is discussed in Section 2.3.2.

Although the direct use of Spaghetti plots was dismissed by Ms King, they may be usable with modifications to the display of data in similar ways to the application of sorting mechanisms and altered graphical parameters. While this would be similar to the previously dismissed SLIDER plots (Section 1.5.3.1), the application of graphical parameters will avoid most of the previously discussed issues. The use of alternative graphical parameters to turn Spaghetti plots into Transitional Plots is further discussed in Section 2.3.3.2.

2.1.3 Quadrant 3 – Categorical Time and Continuous Response

With a continuous response variable, the natural method for handling visualisation is to use Spaghetti plots however the same problems described for Quadrant 2 will also exist here to due crowding and overplotting of the data, therefore the use of Transitional Plots (Section 2.3.3.2) appears to be the most appropriate solution for this situation, at this stage.

2.1.4 Quadrant 4 – Continuous Time and Response

The fourth quadrant of our requirements table requires us to solely look at the Spaghetti plot due to the numerical nature of the variables that will be present on both axes. Without aggregating or grouping responses this is the only type of plot that is suitable for directly displaying data of this type.

2.2 Final Requirements Diagram

Based on these observations, the filled requirements diagram follows in Table 2.2

		Time	
		Categorical	Continuous
Response	Categorical	Quadrant 1 Alluvial Lasagna	Quadrant 2 Transitional Spaghetti Lasagna
	Continuous	Quadrant 3 Transitional Spaghetti Spaghetti	Quadrant 4 Spaghetti

Table 2.2: Updated plotting situations table for longitudinal visualisations

2.3 Improving Existing Plots

From the observations of existing plotting mechanisms in Section 1.5, all the methods that have been highlighted as potential solutions to visualising longitudinal data for this project require improvements to effectively handle large quantities of data. The use of large datasets, such as from the CHNS, specifically highlighted this problem and the need to implement sorting, highlighting and faceting mechanisms to help make the data make sense.

2.3.1 Alluvial Plots

As previously discussed, most documented uses of Alluvial plots have focused on tracking units (or monetary allocations) through distinct categorical classes – examples include showing the fate of passengers from the Titanic dataset by Gender, Ticket Class, or displaying how The New York Times derives its income from various sources[5]. The number of examples for longitudinal visualisation are relatively limited, especially when considering the documentation of available Alluvial graphing packages.

However, as shown with Figure 1.7, the `ggalluvial` package is well suited to displaying Alluvial data in a clear manner for time anchored data.

The Alluvial plots are typically split into two components, as shown in Figure 2.1, the flows (or alluviums), highlighted in blue, that depict the transitions across the x-axis from one grouping to another, and the stratum, highlighted in yellow, which are the boxes aligned with set points on the x-axis depicting response categories where flows originate from and travel to. With specific reference to Figures 1.7 (and also, less clearly, Figure 2.1), the Stratum are the response categories representing if someone was injured or sick in the 4 weeks prior to been surveyed at the given time periods, and the flows/alluviums represent the number of people that were previously belonging to one response category and transitioned to a response category in the next iteration of the survey. We see that while the number of sick or injured persons remains similar between survey iterations, very few people remain sick or injured across the two iterations – indicating either bad luck, or a potentially chronic condition.

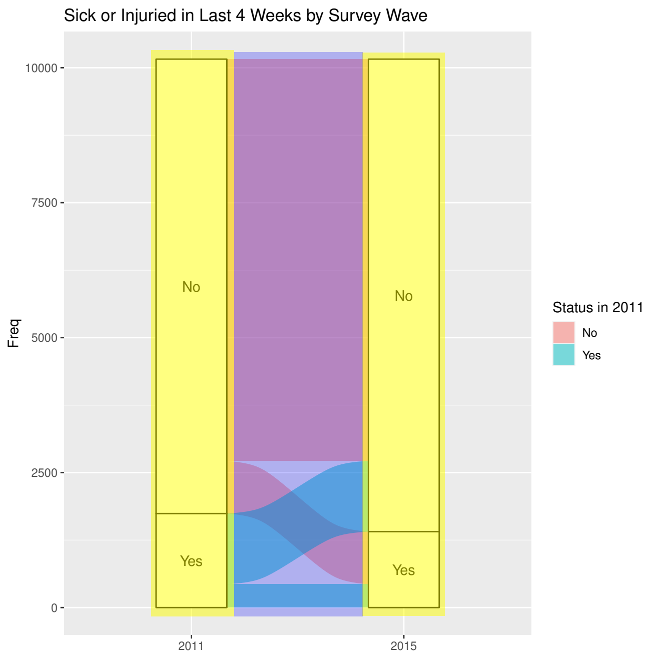


Figure 2.1: Modified version of Figure 1.7 with overlays depicting Stratum (yellow) and flows/alluviums (blue)

If we begin examining a more complicated Alluvial plot, such as the Individual Health Care Dataset with additional waves, such as in Figure 2.2, we start noticing additional issues with the usefulness of the plot, particularly:

- It can become harder – especially with the addition of more time periods and/or response levels – to trace a particular response through time as the flows potentially get thinner or overlaps get fuzzier.
- We may wish to focus on another time period as a baseline – allowing us to tell of story based on the paths taken to get to a particular category in the focus time period, and the paths taken subsequently.
- We may not be interested in the through-time effects, but rather what is happening between a pair of successive time periods.
- We may wish to track the exact paths those that start in a particular response category flow across time, which the re-aggregation loses.

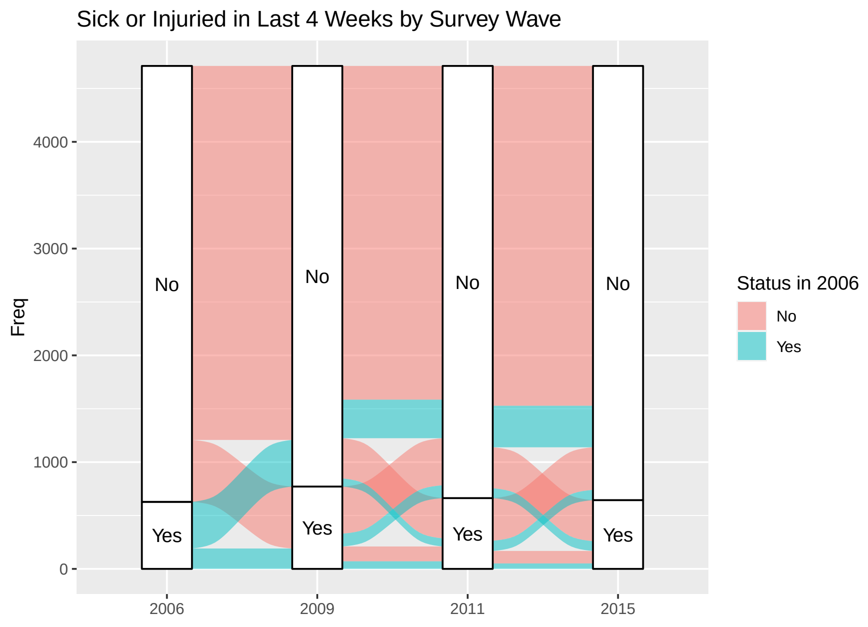


Figure 2.2: Modified version of Figure 1.7 with addition of data from 2006 & 2009

Highlighting variables through time was one of the trickiest parts of the project as unlike with standard R graphics where you can customise the colours at each point, `ggplot2` typically uses a global fill function for the entire plot. The solution to this problem required the modification of the standard `ggplot2` fill functions, the current version of code to achieve this was developed as a result of understanding the R closure system, and reimplementing the standard `ggplot2::scale_colour_discrete` function as an internal function in `longZight` as `alluvial_fill_discrete`, the source code for this function is provided as Code A.3. By specifying the index of the factor level you wish to highlight to the alternative function the colour for all other levels will be made approximately 60% lighter than they would usually be (by using the `colorspace`[26] `lighten` function). Using this on the more complex CHNS Individual Roster dataset (Figure 2.3), we can see the effect in Panel B where the response code 4 (sought employment elsewhere) is highlighted showing the flows between waves clearer.

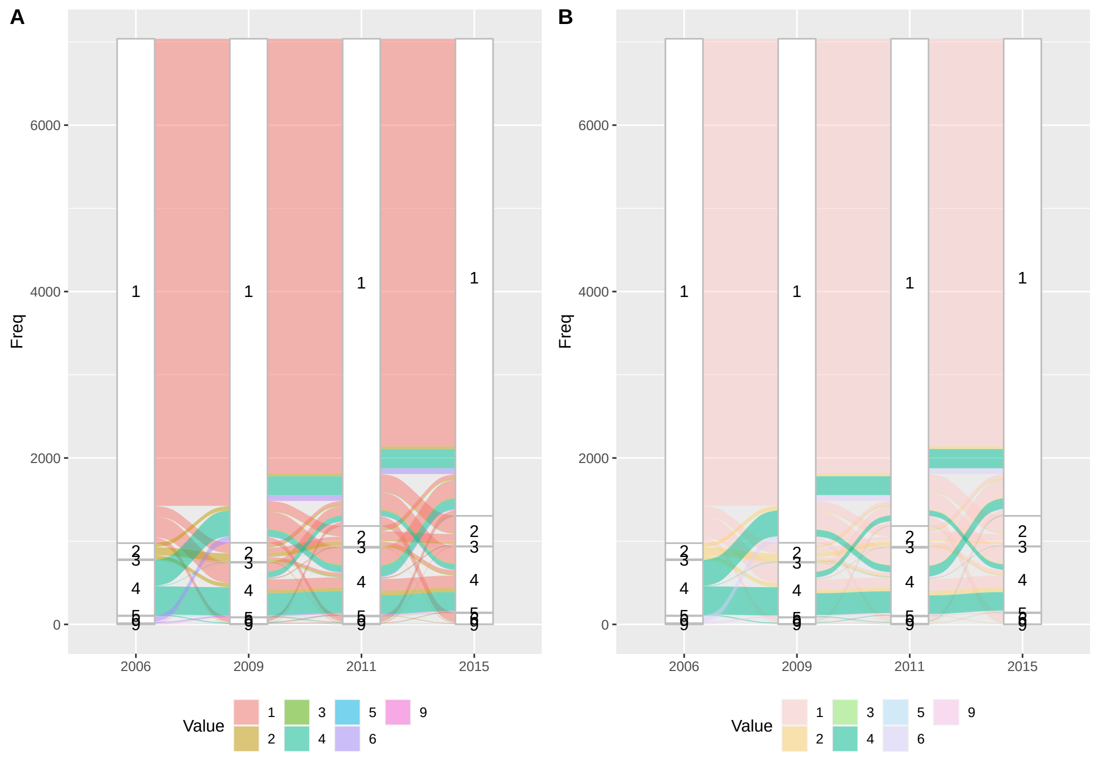


Figure 2.3: Comparison of CHNS Individual Roster dataset without highlight (Panel A) and highlighting level 4 (sought employment elsewhere) in Panel B

Another potential option is to allow users to change which time point serves as the basis for highlight variables, this can be useful for situations where researchers are interested in the progression through a certain point of time. In discussions with my project supervisor I often described this as showing "Where they came from, and where they went" this forms an important part of using these plots for storytelling. In the case of the CHNS Individual Roster dataset, we may be interested in the survey run around the time of the Global Financial Crisis (2008-2009) and the living situation of participants, and how they changed from prior iterations and what happened afterwards. This has been made available in `longZight` by allowing the user to instruct the plotting code which time period to focus on, an example is provided in Figure 2.4.

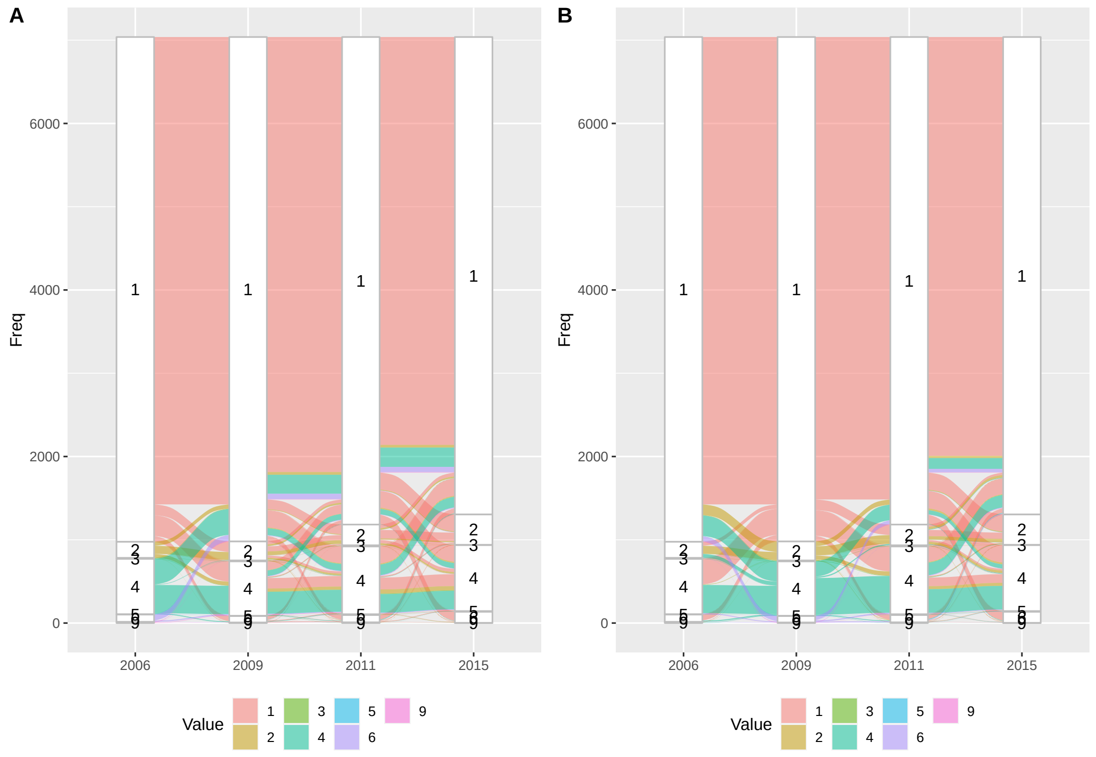


Figure 2.4: Comparison of CHNS Individual Roster dataset with default focus (2006; Panel A) and focus changed to 2009 (Panel B)

To describe the Alluvial plots shown to this point, I began referring to them as having been "Coloured by Origin" with the origin reflecting either the default focus (first wave/time period in the series) or the user selected focus.

Solutions to the remaining problems were found within the documentation for `ggalluvial` but the data structures created to support the initial Alluvial plots shown thus far were not compatible with the alternative geometry and statistic methods provided by the package. However I identified that the data manipulation code that I had developed for the Lasagna plots (which are described in detail below in Section 2.3.2) that were now been worked on in parallel was directly applicable to the alternative geometries.

First, by changing the configuration of the `ggplot2` aesthetic options, I was able to create a version of the Alluvial plots that showed the wave-to-wave transitions as shown in Figure 2.5.

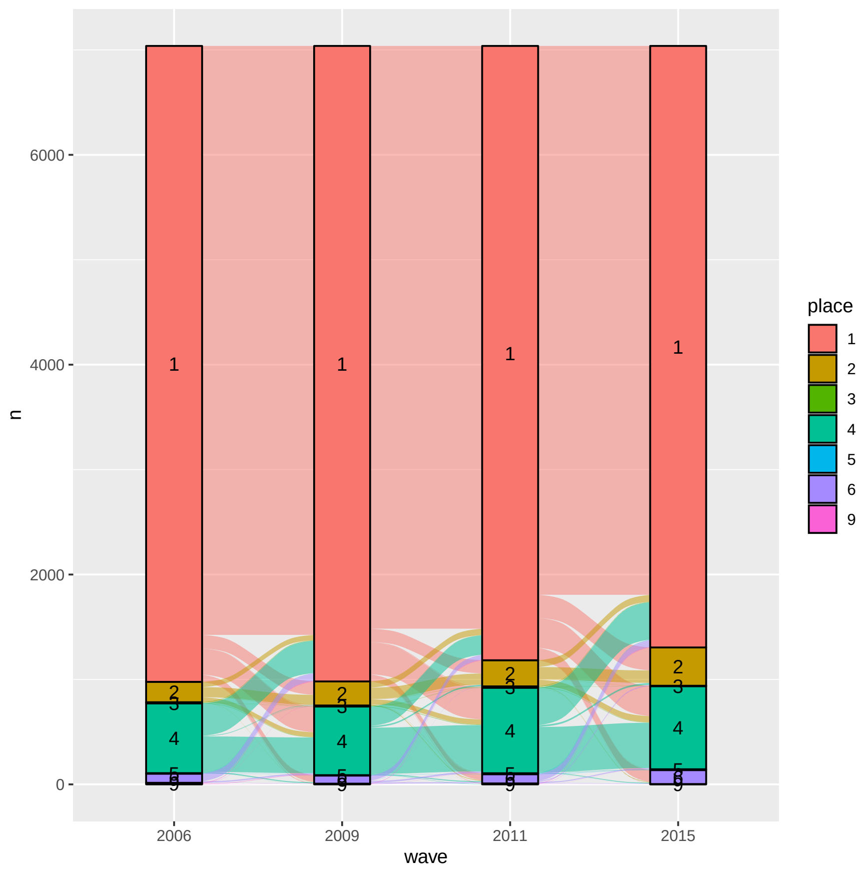


Figure 2.5: Alluvial plot showing wave-to-wave transitions between levels

The design of this plot allows us to compare the number of people that are moving from a particular level of the response variable to another, for instance we can clearly see that the number of people that are moving from home (1) to school (2) between the 2011 and 2015 surveys is close to double the width as previous transitions between surveys. Similarly, we can see that fewer people changed from been away from their household for other employment (4) back to home (1) between the 2009 and 2011 surveys, compared to the other two available transitions. The ability to make comparisons, and visually convey comparisons such as this is of importance to the wider goal of enabling storytelling. This type of plot has been referred to as a "Transition Plot" in other areas of the project.

Finally, the inability in the previous Alluvial plots to distinguish the exact path people take through multiple waves is an important omission that needs to be resolved. In previous Alluvial plots the `ggalluvial::geom_flow()` geometry has been used, but another geometry is available `ggalluvial::geom_alluvium()`, again this requires the data aggregation methods developed for Lasagna plots but was ultimately as simple as changing the geometry used. This leaves us with Figure 2.6.

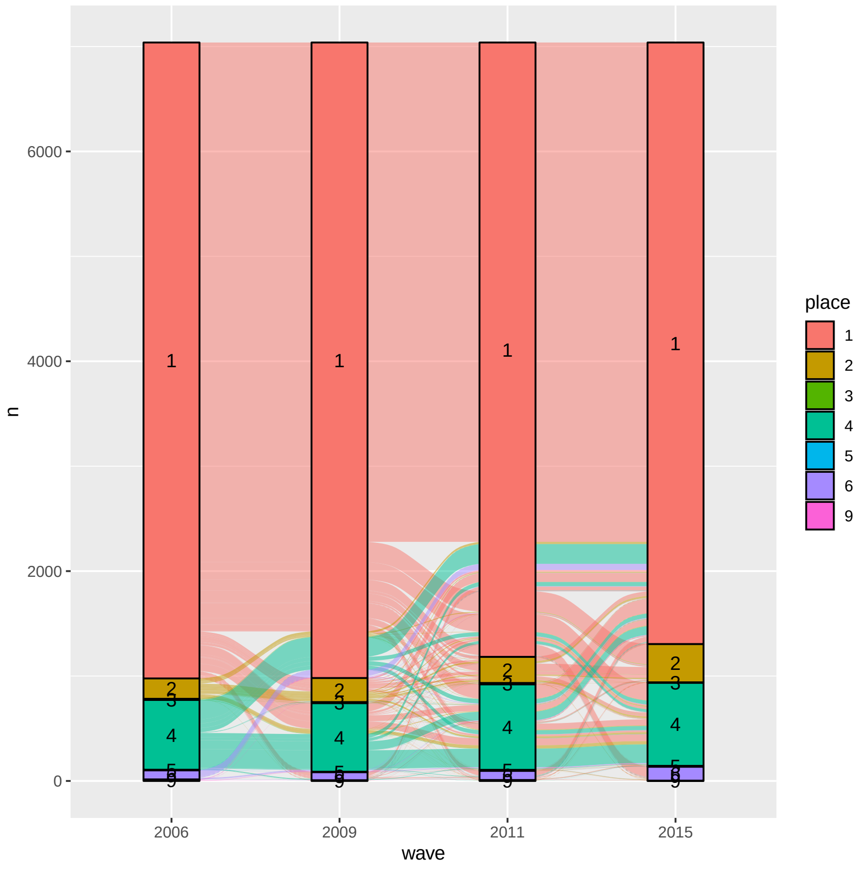


Figure 2.6: Alluvial plot using `geom_alluvium` instead of `geom_flow`

The primary feature of this plot is that the flows (now referred to as alluviums) split out at each stratum (time point) and when exiting the stratum remain positionally accurate to where they entered, this means we can track the exact flow between time periods that a group - regardless of size - took through the survey and may be of use to understanding precise patterns within the data. As these plots allow the tracing of flows across time, these were referred to as a "Trace Flow" plot.

2.3.2 Lasagna Plots

The continued use of `lasagnar`[16] for this project was ruled out at an early stage due to the concerns with the age and maintainability of the code as described in Section 1.5.2. The package did however provide a valuable basis for a modern implementation that could be used in the development and progression of lasagna plots.

The `lasagnar` package provided two plotting mechanisms, a standard `lasagna()` function, and a `ggplot2`-based `gglasagna()` version. The standard version makes use of data stored in a `matrix` structure and can be passed to `image()` (from the R `graphics`[13] package), or `image.plot` (from the R `fields`[6] package), while the `ggplot2` version uses the `geom_tile` geometry to print the coloured squares in the plotting area.

Both mechanisms provide no obvious data aggregation methods, therefore we have a 1:1 mapping between rows of tiles and subjects. Although there are obvious advantages to this approach, especially when dealing with data that uses a continuous time scale, including the previously described work by Victoria King on sleep patterns during pregnancy, there are several obvious disadvantages.

If we were to consider plotting the CHNS Individual Healthcare Dataset (Section 1.4.1.2) using either of the `lasagnar` methods, replicating the Alluvial plot from Figure 2.2 above in Lasagna format, we would be implicitly asking R to draw 18,844 boxes (4,711 unique subjects, each with 4 observations) in the plotting window. Practically this means there is a high chance of loss-of-fidelity in data. Considering that a typical computer display is either 1080 pixels high (Full HD), or 2160 pixels high (4K), it is obviously not possible to fit all 4,711 unique lines on a screen at once. Additionally, when using other technologies such as `plotly`[15], the conversion process from `ggplot2` format will either grind to a halt, or be extremely slow due to the conversion of 18,000+ tiles to the `plotly` format. This issue would prevent the inclusion into iNZight Lite which uses `plotly` where possible.

Two potential solutions to this problem are primarily, sorting (as discussed by Victoria King and described in the introduction to Lasagna plots) or data aggregation. Sorting records so that individuals with the same responses are grouped together would theoretically give a proportionally accurate graph, but if thick black borders are used (as in Figure 1.4) it is likely the presence of 18,000+ borders would hide the data and the user would be left with a black box, even with tile borders disabled there is still a chance the size of boxes will not be proportionally representative due as lines may be omitted depending on how the mechanisms handle boxes smaller than a pixel. This leaves data aggregation prior to plotting as the remaining solution, however once again the `lasagnar` package proves unsuitable for this task as the code for `gglasagna()` enforces equal heights for each row of tiles.

I had also taken some inspiration from the SLIDER Shiny Application (Section 1.5.3.1) which generally aggregated groups and plotted tiles in a proportionately representative manner, however the code would not be easily adaptable for this project.

As a result, work was started to overcome these issues and improve on the plots in general.

One decision made early in the process was to avoid using individually printed tiles for each observation, instead focusing on aggregation. It was found that the `geom_tile` geometry would not be suitable for the data aggregation despite the presence of a `height` parameter. Instead I had to rely on the `geom_rect` geometry which required the computation of precise placements of the rectangles. The calculation of these values required aggregation above and beyond the code required to generate the basic Alluvial plot (as seen in Figure 2.2), that also produced a wider data frame. The additional code for aggregating and sorting data for Lasagna plots proved useful when developing additional Alluvial options as previously described.

Overall three sorting methods were developed, to illustrate Figures 2.7–2.9 show each sorting mechanism used with the CHNS Individual Healthcare dataset (Panel A) and the more complex CHNS Individual Roster dataset (Panel B) as some nuanced differences are not immediately obvious with the simpler dataset.

An example of the default Lasagna plot that is featured in my `longZight` package, which was developed using these principles, is provided in Figure 2.7 below.

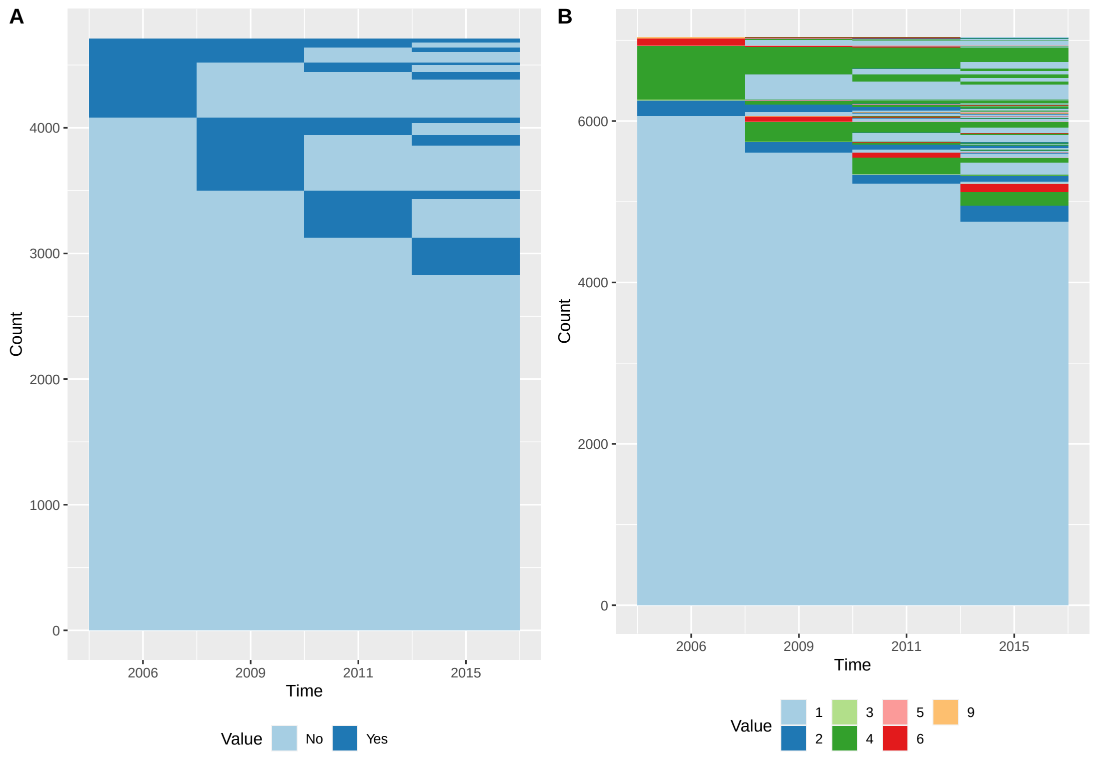


Figure 2.7: Lasagna plots generated using default options from `longZight` (By factor order)

By default, the sorting mechanism sorts by factor order (going left to right), we can see that there are very few people that stay injured in multiple successive sampling frames this way.

With the more complex Individual Roster dataset, we do note that proportions and the importance of across-time patterns is not as well defined, and that alternative methods to sorting data may be appropriate. In order to try and resolve this we consider sorting by the number (or proportion) of respondents that had the same across-time response history to the survey, as shown in Figure 2.8.

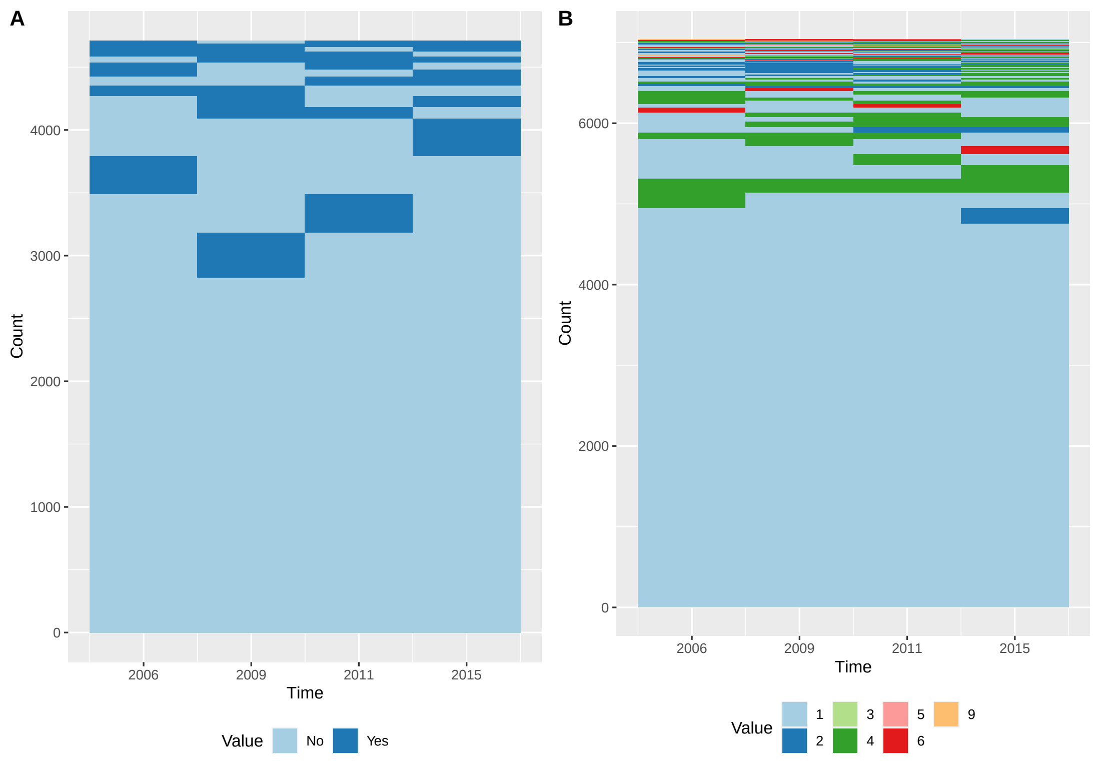


Figure 2.8: The lasagna plot from Figure 2.7 sorted by count of matching response patterns (By Count)

Finally, as a third option, we may wish to combine the sorting effects from the previous two plots, by keeping the contiguous blocks of data as much as possible, but ordering levels by the frequency that they appeared, this is achieved with the third sorting mechanism in Figure 2.9.

It's important to observe that in the case of Figures 2.7 & 2.9 the plots in Panel A mostly stay the same due to the simplicity of the dataset. This is not an unexpected result, and we would expect similar occurrences with other datasets that only have a few response categories.

The general usefulness of lasagna plots is dubious as there can be issues when trying to follow across the x axis, but we feel they are a worthy inclusion at this stage as they are clear

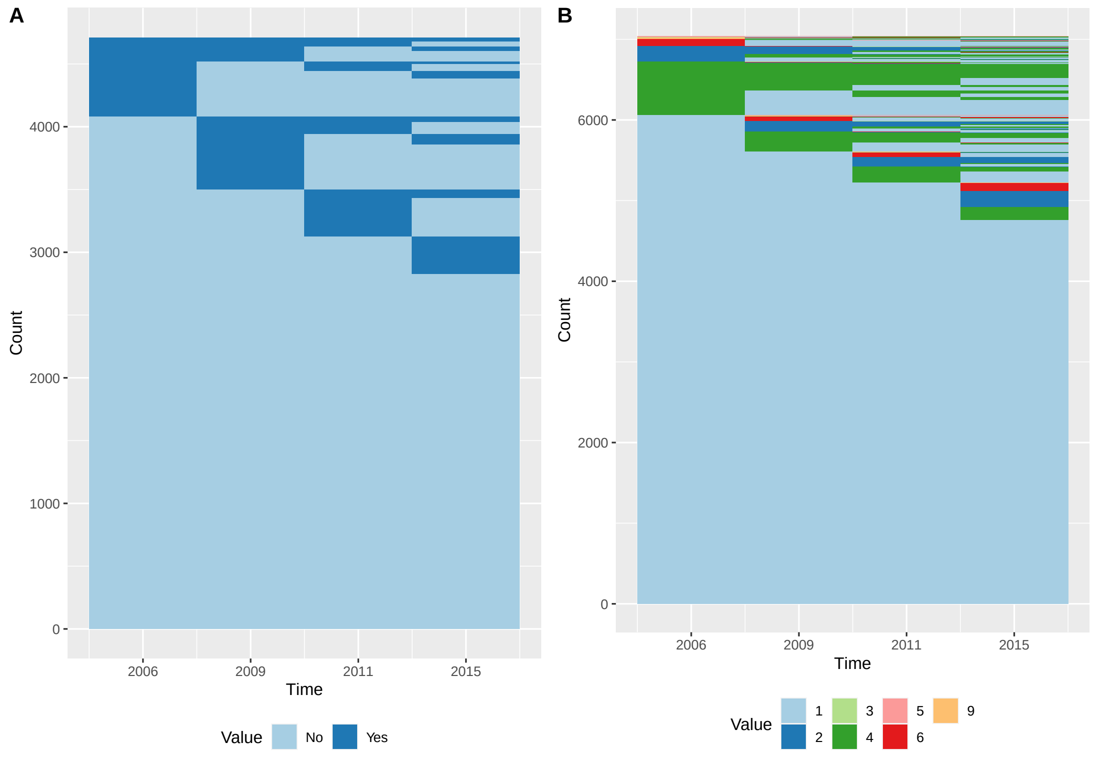


Figure 2.9: The lasagna plot from Figure 2.7 sorted by the largest contiguous blocks of data (By Block)

with simple datasets and I did receive feedback that the plots in their current form would be potentially useful to one organisation currently running a longitudinal study that had invested significant energy in producing plots, in a graphics application, for an annual report that were identical to this format so they could represent participant retention across survey waves.

2.3.3 Spaghetti Plots

As described in Section 1.5.1 the Spaghetti plots are naturally suitable for data where the time and response variables are both continuous. However I've also hypothesised that Spaghetti plots could be adapted to communicate data where one of the axes is categorical, while also adapting to some issues that have already come up with Spaghetti plots or categorical data before, particularly overprinting. The discussion of improvements will be split into two, one for each main use of Spaghetti plots.

2.3.3.1 Continuous Use Cases

The initial plot (Figure 1.2) of the `brolgar` 'Wages' dataset (described in Table 1.3 – Section 1.4.2) demonstrated key issues that exist when trying to plot large datasets – although with just over 6,400 observations it would be unfair to call this dataset as large – in particular the overprinting of lines which results in the indecipherable black 'blob' seen in the middle of the plot.

Traditional mechanisms for dealing with overprinting such as alphaing and colour (representing levels of a relevant teriary variable), are potential solutions as seen in Figure 2.10. However, colour is only typically used to highlight specific groups and not as a mechanism to make plots more readable.

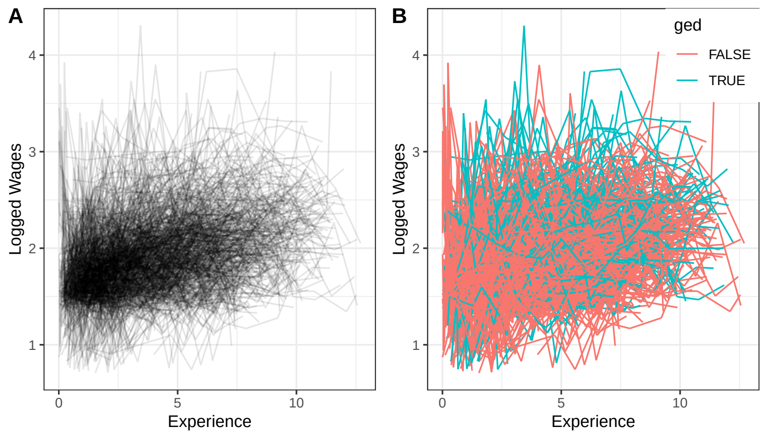


Figure 2.10: Use of Alphaing (Panel A) and Colour (Panel B) with Spaghetti Plots

Although the use of alphaing in the left panel does help to some extent, the individual paths/trajectories are still masked by the sheer quantity of observations that go in a variety of directions. Additionally, because such a low alpha value was required, the individual paths that are distinct are very hard to see and may not even get printed correctly, depending on the manner the plot was generated.

The use of colour in the right panel does not particularly help as the ordering of observations as they are printed onto the plot is not immediately obvious appears random - as shown by the cyan coloured lines disappearing and appearing in amongst orange/red coloured lines and is generally controlled by the `group` aesthetic option to the `ggplot2` `geom_line` call. To achieve this, the factor levels would need to be modified to ensure that the lines were printed in the right order, for variables with a large number of factors this is likely to be undesirable and provide no tangible benefit as existing methods such as faceting exists to draw separate plots based on the value of a variable.

Instead, one feature that was shown in the `brolgar`[17] documentation was the use of binary variables to bring data of relevance to the front of the plot. An example might be where a user only wishes to explore the effects of work experience within the Black community resulting in Figure 2.11. Immediately several points are clear from the plot, it appears that those of Black race (black lines, in foreground) have lower experience and lower logged wages compared to non-Blacks (the grey lines in the background).

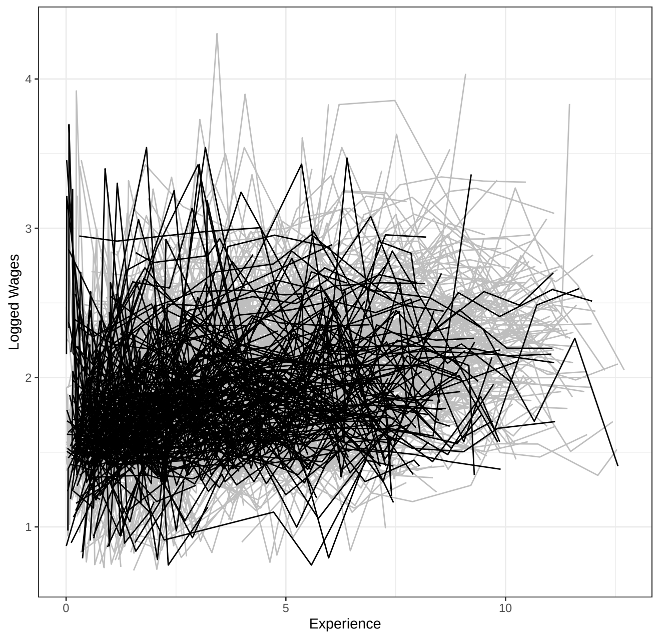


Figure 2.11: Use of `brolgar` technique of only placing relevant data in foreground

The addition of colour or faceting allows the further separation of relevant data for more nuanced comparisons, for instance we can add colour to look at differences within the Black community who have a GED[1] or not (Panel A; Figure 2.12) and then further subset by the highest level of grade-school completed (Panel B; Figure 2.12)

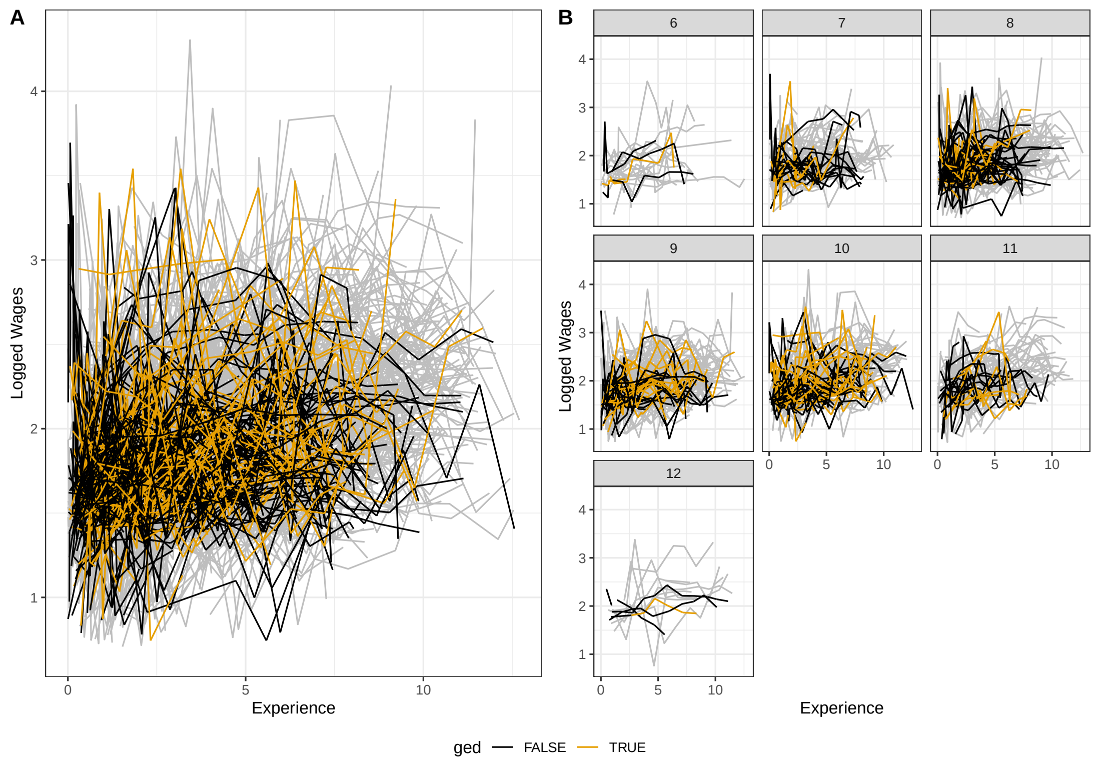


Figure 2.12: Demonstration of adding additional variables to Figure 2.11 such as colour and faceting

We can see by allowing these features, we can aid storytelling significantly, especially when compared to the original black blob of spaghetti when this data was originally plotted as Figure 1.2. For instance we see that very few people enter this dataset at Grade 12, and even fewer Black people, for those in 11th & 12th grade, in the background there appears to be a steady increase in wages, but this isn't reflected to the same extent as those of Black race.

[1]Graduate Equivalency Diploma

2.3.3.2 Transitional Plots

As discussed previously, the use of spaghetti plots where the response variable is measured with a discrete scale will often result in an unreadable plot as shown earlier in 1.3 and repeated in Panel A of Figure 2.13. This is undesirable for many reasons and particularly for using the plots as a storytelling mechanism. The plot suffers from several issues that are common when attempting to plot transitional data where one axis is continuous, particularly we have overprinting and since the transitional pattern is predictable, we have what could easily be mistaken as a linear relationship by people not familiar with the data presented.

The complex tangled mess of lines would not be improved if we were to add more features – such as colouring based on a tertiary variable. Ultimately this means we have a situation much like that with the SLIDER transition plots outlined in Section 1.5.3.1.

Additionally, with a traditional line or spaghetti plot – such as Figure 1.1 – we imply a trajectory from one observation to another. When we have one of the axes on a discrete scale we cannot always assume a linear trend and the drawing is complicated with the fact that there are only limited destinations along the axis that the point can rest or terminate at and the lines become unclear and overprinting becomes a significant issue.

To try and resolve some of these issues, and while considering solutions for the second and third quadrants of the plot types table (Table 2.1), I decided to experiment with R's step-type plotting mechanism. This decision seemed appropriate as there is an inherent implication that observation holds steady at a particular value until it is observed at the new value.

The implication is generally not unreasonable as it is typical that people or systems will treat the currently known truth, as the truth until told otherwise, only in rare cases do changes in knowledge of the state or value of an object change past decisions. This assumption is also consistent with the treatment of Survival data where observational units are treated to be active, or at a particular state until the change is recorded or the point of censorship.

By nature, implementing a step function does not solve the problem of overprinting of lines, in fact as lines are now constrained to a set number of y-values. As can be seen in Panel B of Figure 2.13 the problem has actually worsened. Instead we must also consider other tactics such as jittering of the lines (it's important to note that the jitter offset used should be constant for all observations from the same study unit to ensure consistency) - as demonstrated in Panel C of Figure 2.13.

Finally, as are still plotted with close proximity (and generally the line thickness is comparatively wide compared to the jitter offsets), additional mechanisms just as applying an alpha to the lines printed by the step function are necessary. This means that so that common paths were easily seen in a darker colour compared to where very few observations were in a particular state as shown in Panel D of Figure 2.13.

It's important to note that this tactic can also be inversely applied for categorical time and a continuous y-axis as the same problems and solutions can be applied. Finally it's also important to note that due to the use of a 4-plot panel, and the smaller plotting area, Panel D is potentially undesirably dense, but the overall picture can still be drawn from it, and the common timings of transitions which are equally important are clearly visible.

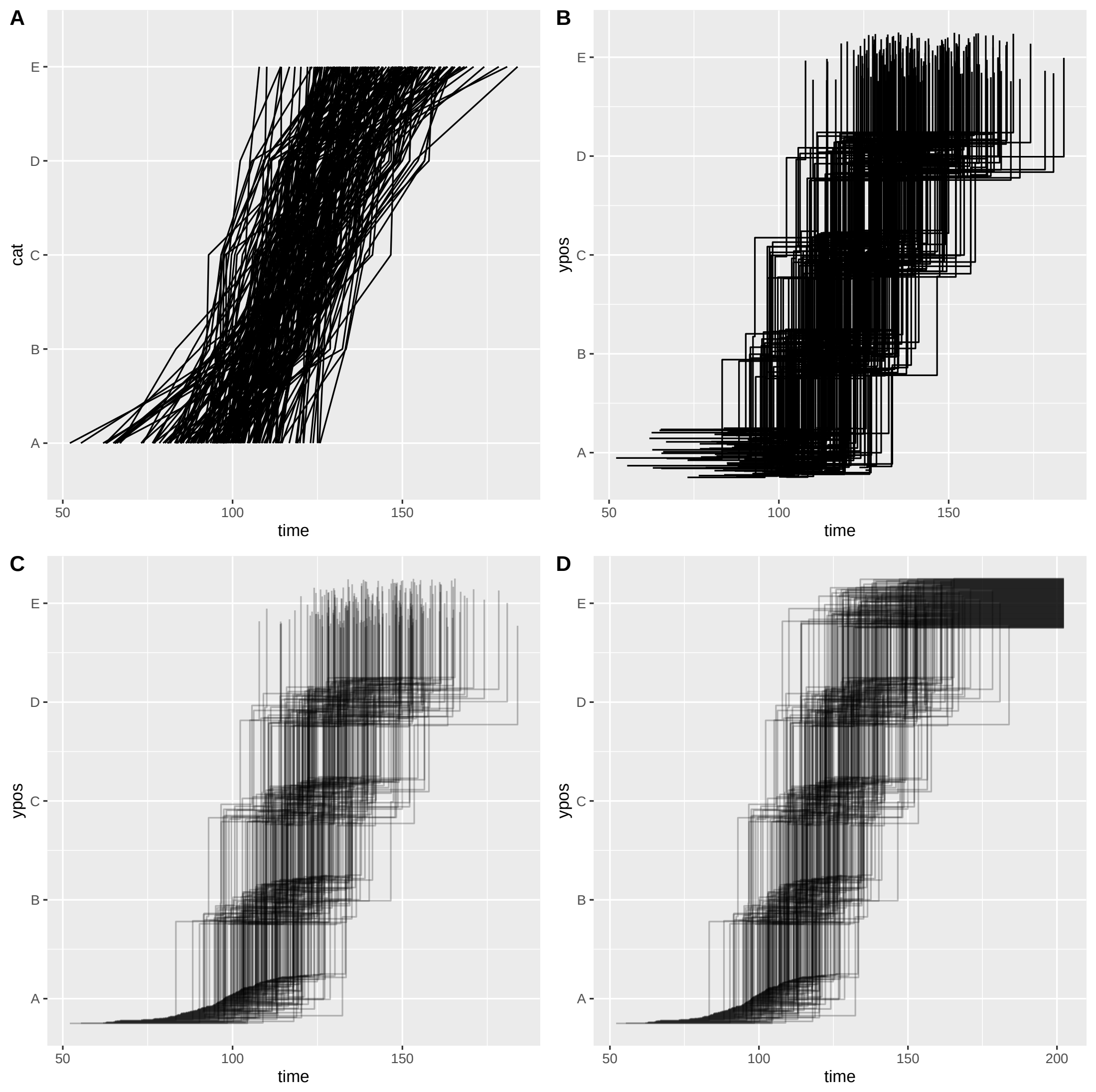


Figure 2.13: Demonstration of development process to creating suitable transitional diagrams

Chapter 3

Longitudinal Data Exploration Module

3.1 Integration with iNZight

A primary outcome of the project, as discussed in 1.3 was to develop a module (a self-contained & independent piece of software that is integrated into a larger application) that may be distributed with iNZight to allow users to generate and customise plots to visually represent longitudinal data.

Integration with iNZight is beneficial in multiple senses, as with other technologies such as Shiny (discussed in Section 4.1) the general user interface scaffold is available, practically this meant that building a new application to handle longitudinal data specifically was not required, and I was able to leverage the existing data manipulation tools available in iNZight. An example of the "Convert to Categorical" tool is provided below in Figure 3.1, but additional tools that allow users to reformat and collapse (or merge) variables is available and is beneficial to both longitudinal data and the visualisation of it.

Two options were available to include the longitudinal visualisation components into iNZight:

1. Include the visualisation code into the `iNZightPlots` code repository and build longitudinal plotting functionality directly into iNZight.
2. Develop the self-contained module with the visualisation code contained in a new R software package.

The selection of the second option was primarily driven by the infancy of both the `ggalluvial`[2] and `brolgar`[17] packages, the latter is not yet available via CRAN. The development of a self-contained module meant that changes to those packages, or the `longZight`

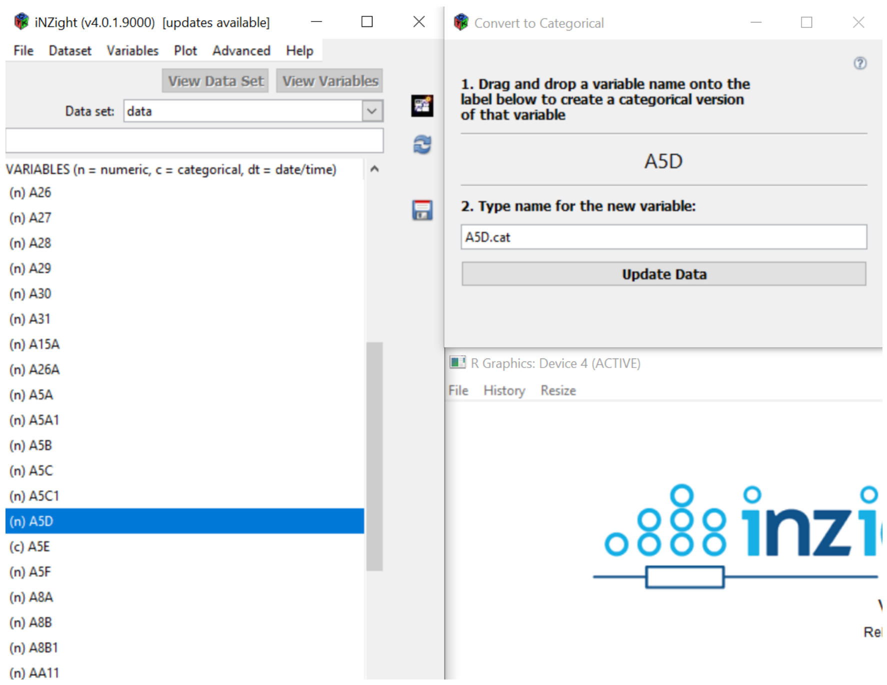

iNZight (v4.0.1.9000) [updates available]
File Dataset Variables Plot Advanced Help
View Data Set View Variables
Data set: data
VARIABLES (n = numeric, c = categorical, dt = date/time)
(n) A26
(n) A27
(n) A28
(n) A29
(n) A30
(n) A31
(n) A15A
(n) A26A
(n) A5A
(n) A5A1
(n) A5B
(n) A5C
(n) A5C1
(n) A5D
(c) A5E
(n) A5F
(n) A8A
(n) A8B
(n) A8B1
(n) AA11
Convert to Categorical
1. Drag and drop a variable name onto the label below to create a categorical version of that variable
A5D
2. Type name for the new variable:
A5D.cat
Update Data
R Graphics: Device 4 (ACTIVE)
File History Resize

Figure 3.1: Example of Pre-existing Data Manipulations Tools in iNZight

package developed as part of this product (Section 4.2) could be quickly adapted into a package update and distributed without requiring an update to iNZight.

Additional advantages of the providing the tools for longitudinal visualisation as a self-contained module include the use of `gWidgets` to create a distinct User Interface (UI), and not exposing the standard data preview tools available in iNZight to users when those tools are not applicable to longitudinal data.

At present, due to the use of the GTK2 framework for the iNZight User Interface, the module will not be compatible with devices running recent versions of the macOS due to well documented incompatibilities between the two pieces of software introduced by successive updates to macOS. The creation of a standard module will however ease porting the Longitudinal Visualisation module to iNZight Lite, a Shiny-based browser version of the application that works on any device that supports modern web technologies.

3.2 User Interface Design

A design principle previously observed in iNZight is the use of a vertically linear process where a user can go from the top of the interface to the bottom to complete an action. An example of linear step-by-step design can be seen in the "Convert to Categorical" interface in Figure 3.1. The Time Series iNZight module provided the most inspiration for the design of the Longitudinal Data module, due to its process of prompting for the required information to configure a time series object, and then options to customise the resulting plot.

Several iterations of design sketches resulted in the user interface depicted in Figure 3.2, which was fairly close to the final design described in Section 3.2.1.

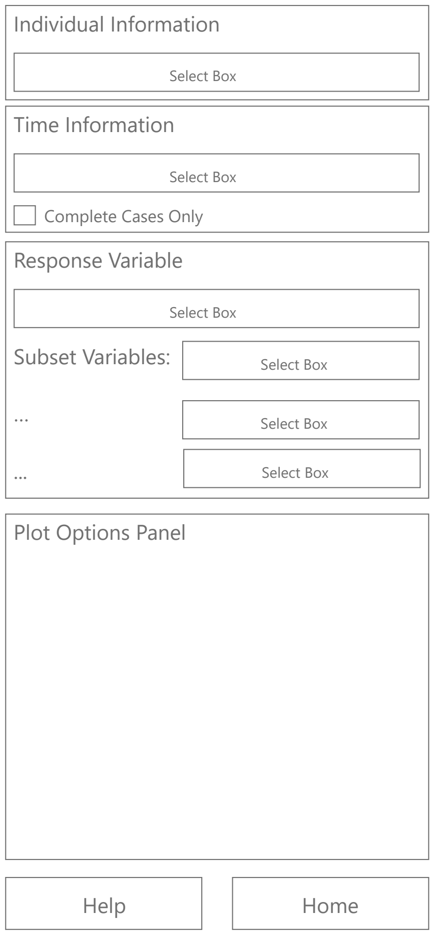
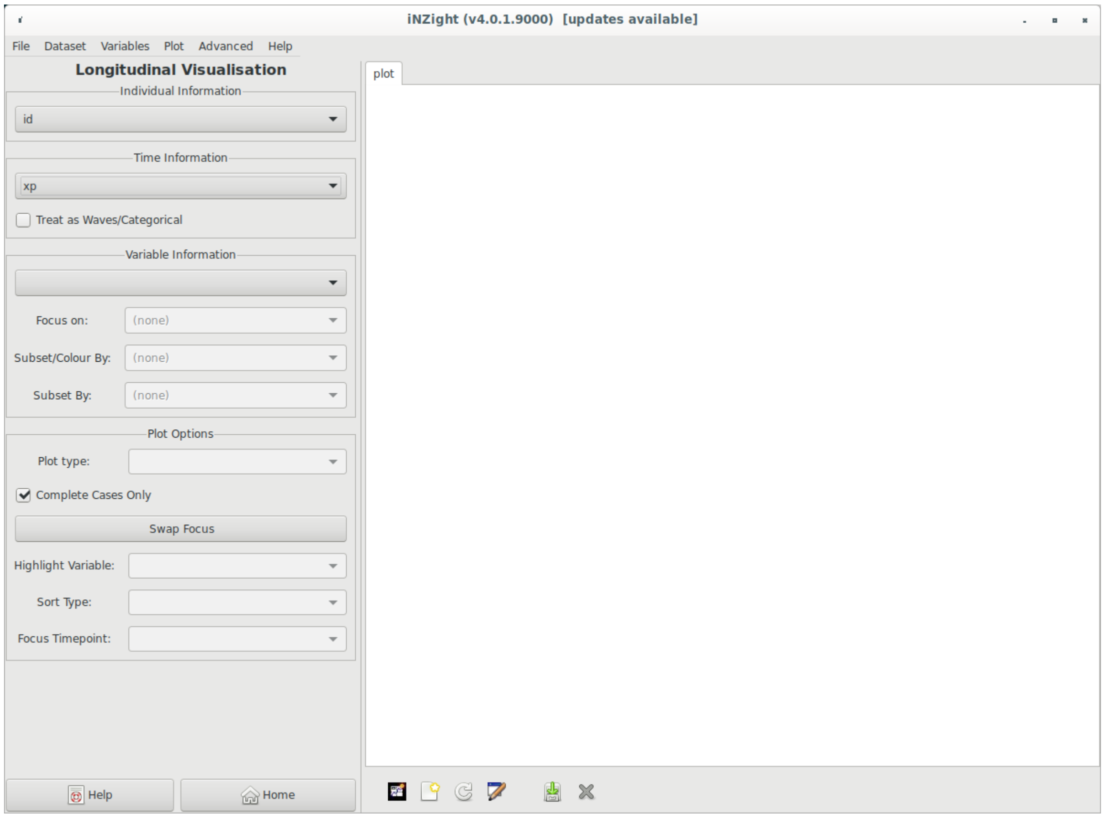


Figure 3.2: Rendered version of final UI sketch

3.2.1 Implemented User Interface

The implementation of modules for iNZight, relies heavily on the use of `gWidgets2`[18] objects (called constructors). The `gWidgets2` objects are an implementation of user interface elements from GTK2 which is the UI framework used by the iNZight application.

The module user interface - at an early stage in variable selection is shown in Figure 3.3, options that are not applicable to the current plot type are generally disabled - by way of greying out.


iNZight (v4.0.1.9000) [updates available]
File Dataset Variables Plot Advanced Help
Longitudinal Visualisation
plot
Swap Focus
Help
Home

Individual Information: id

Time Information: xp

☐ Treat as Waves/Categorical

Variable Information:

Focus on: (none)

Subset/Colour By: (none)

Subset By: (none)

Plot Options

Plot type:

☑ Complete Cases Only

Highlight Variable:

Sort Type:

Focus Timepoint:

Figure 3.3: Demonstration of Implemented User Interface

3.3 Development

The majority of code required for data manipulation and visualisation of the plots had already been developed early in this project as part of the creation of plot prototypes (Section 4.1) and implemented in the `longZight` package (Section 4.2) that I created based on the results of the initial prototype designs and subsequently further improved on. This work serves the basis for the iNZight module with the `longZight` package serving as the core provider of backend functions for data manipulation and plotting.

3.3.1 Use of Wrappers

Creating a plot using `longZight` typically requires the chaining of several functions to process the data, and then output the relevant plot and were developed to work in a similar manner as the code that powered my earlier Shiny prototypes. However, the use of wrapper functions was proposed as a method to reduce the number of external function calls from the module and reduce the likelihood of needing to update the module. Under this scenario, the module would utilise a single function wrapper for each type of plot depending on the type of data and the options selected by the user of the module.

3.3.1.1 Use of a 'One-Shot' Wrapper

For instance, instead of the code shown below in Code 3.1, a 'one-shot' version of the code can be used such as shown in Code 3.2. In the case of a 'one-shot' wrapper function, the entirety of the chained code that would typically be run by the user (Code 3.1) is initiated by the single function call (Code 3.2) without the user been exposed to the full chain of code.

```
  input_data %>%
    longZight::load_long_data(idvar = "id", timevar = "wave") %>%
    longZight::refactor_wave %>%
    longZight::recode_long_alluvial(response = "choice", completecases=TRUE) %>%
    longZight::plot_long_alluvial()
```

Code 3.1: Example of Tidyverse-like syntax for creating a plot using `longZight`

```
longZight::iz_plot_alluvial(input_data, idvar = "id", timevar = "wave", wave.as.
    factor = TRUE, response = "choice", completecases = TRUE)
```

Code 3.2: Example of the using a 'one-shot' wrapper in `longZight`

Including the initial loading of data into the wrapper function did have several downsides that were identified early. This resulted in the decision to separate the loading and validation of data from the plotting wrapper.

The reasons for this decision comprised mainly of the fact that as individual identifier and time variables would be unlikely to change between iterations of plots, re-running the initial validation mechanisms and potentially refactoring the time parameter, seemed unnecessarily wasteful, especially since the underlying data could not change while the module was open.

Additionally, loading the data into `longZight` as soon as possible allows the module to use helper functions to determine the suitable plot types instead of the currently statically coded criteria. This will be particular useful in the future if additional suitable plot types are identified for the inclusion into the `longZight` package, and iNZight module.

For plotting Alluvial and Lasagna plots, the use of a two-stage process was still of concern as the middle stage of the process (line 4 of Code 3.1) is generally computationally expensive, particularly with larger datasets. This was a reason for the multi-stage implementation in longZight as it allowed users to save intermediary results – like with most Tidyverse-inspired designs (Section 4.2.1).

However, despite the concerns, the two-stage process with a single wrapper that handled the appropriate data recoding and final plotting was pushed forward.

3.3.1.2 Use of a Two-Stage Process

For the most part, the separation of initial data loading and refactoring of time, from the recoding and plotting of data, worked well. Especially for Transitional (line) plots and Spaghetti plots, which did not require additional intermediary processing.

However, the two-stage process did introduce, as expected, problems for the creation of Alluvial and Lasagna plots due to the inability to access intermediary results. This meant that generating the required items for some aspects of user interface, such as the levels of the response variable that would be shown on the plot, were complicated. The main source of the complexity was due to the provision of a 'Complete Cases Only' option which was included at an early stage of the project.

If a user ticks the checkbox to only visualise complete cases it is plausible that some options from the response, or even - as seen in the data used during this project - entire waves/time periods, will be removed from the plot.

Predicting which of the response levels or time periods would not be shown in the plot if the "Complete Cases Only" option was selected was not possible, especially with the limitation of minimising the quantity of computational & data manipulation code in the iNZight module.

At this point, two options were available:

1. to capture the intermediary results within the `longZight` wrapper function, and result them alongside the completed plot - requiring code within the module to display the final plot and redirect the intermediary results to the appropriate UI functions.
2. to revert back to a three-stage processing module as used in the original Shiny prototypes.

The second option was both computationally and practically advantageous to the end-users of the module, especially if they wished to make modifications to the plotting options. Therefore the decision was made to adapt the module to separate out the calling of the respective `recode_long_<type>` functions from the wrapper.

3.3.1.3 Use of a Three-Stage Process

With reference to the example plotting procedure in Code 3.1, the process for handling data in the iNZight module ended up as follows:

1. Lines 1-3: Selection/Update of Individual or Time information variables, or toggling the "Treat Time as Categorical/Waves" checkbox
2. Line 4: Selection/Update of the Response variable, changing the type of plot to be displayed, or toggling the "Complete Cases Only" checkbox
3. Line 5 (wrapper version): Changing any plot options - such as highlighting a response level, or cosmetic plot options. These functions were named in my `longZight` package as `iz_plot_[plottype]()`.

Changes in earlier stages also trigger the updating (if enough information - such as the selected response variable - known), the later stages. For Transition-line & Spaghetti plots, the second and third stages are combined.

Adapting to the three-stage design produced a noticeable increase in speed to generate plots with minor alterations (such as changing the highlighted response level), and resolved the issues that came from an inability to access intermediary results.

3.3.2 Development Challenges

3.3.2.1 Use of `setRefClass()` in Module Structure

The base structure of an iNZight module requires the use of `setRefClass()` function (from the `methods` R package), which defines a `fields` variable with a similar design to the `slots` variable to the R S4 `setClass` function. Variables that are required outside the scope that objects are required under must be defined in advance using the `fields` variable.

As the `longZight` package developed in the course of this project used several custom S3 classes (Section 4.2.2), the `fields` definitions that are intended to store data manipulated by the package must be defined as the `ANY` or `data.frame` classes, rather than the more correct `longZight` class. There are minor advantages to this approach as changes to the class structure and wrapper functions used by the module may not need to change, however, as a counterpoint it's important to note that safeguards to ensure the correct objects are stored in the respective fields are missing.

Additionally, the use of the `fields` definitions for variables that are required outside of the scope they are defined in generally means that at least two variables must be defined for each user interface element. As an example Code 3.3 contains a heavily abridged excerpt from the Longitudinal Module source code, lines 9 & 11 both provide expected variables for storing the current state of `gWidgets2` objects (generally updated by a `handler` function defined as part of the object creation), and lines 10 & 12 both provide variables to store references to the `gWidgets2` objects created in lines 19 & 22.

This contrasts to with other languages where User Interface objects are typically stored as part of another class (such as a `Form` class (C#)) and accessed directly through that class without additional setup, or by dynamically selecting elements by a name or ID (Javascript). Variables created in this manner also require different syntax to work with (such as using the <<− assignment operator (lines 19 to 22) instead of the regular <− operator). The differences between the languages I have used in the past (especially recently) and the expectations for module classes for iNZight caused significant confusion during the development of the module.

```
 1  LongModule <- setRefClass(
 2    "LongModule",
 3    contains = "CustomModule",
 4    fields = list(
 5      GUI = "ANY",
 6      activeData  = "data.frame",
 7      loadedData  = "ANY",
 8      ...
 9      responseVar = "ANY",
10      responseVarSelect = "ANY",
11      completeCases = "logical",
12      completeCasesSelect = "ANY",
13      ...
14
15    ),
16    methods = list(
17    initialize = function(gui, name) {
18      ...
19      responseVarSelect <<- gcombobox(names(activeData), selected = 0, ...)
20      ...
21      completeCases <<- TRUE
22      completeCasesSelect <<- gcheckbox("Complete Cases Only", checked = TRUE, ...)
23      ...
24    },
25    ...
26    )
27  )
```

Code 3.3: Abridged User Interface Code from Longitudinal Module

A theoretical replication of the typical C# style of storing UI elements could potentially be obtained by the use of a `list` object dedicated for storing `gWidgets2` objects, for instance in a field named `UIElements`, but wasn't explored further during this project.

3.3.2.2 Triggering of Handlers

The module has a number of user interface elements where the values available to the user depend on previously selected options (for instance the available subsetting variables may depend on the response, available plot subtypes and sorting options depend on the selected plot type). We could in theory consider this relationship between elements as parent and child – whereby the child elements depend on their parent elements. When a parent element is updated, the handler function for that element will be triggered, and result in successive calls to update the child elements, potentially re-process the data, and update the plot. However, updating the child elements in this manner can result in the triggering of the handler functions for each child element, which may then result in calls to the same helper functions for a second, third or potentially even more times. This can give potentially give the appearance to the user that the tool is malfunctioning or unresponsive and is undesirable from a user experience perspective.

After discussing this problem with the iNZight Lead Developer, I was advised of functions provided by the `gWidgets2` package to disable and re-enable the handler function of an element as needed. These functions are frequently used in iNZight and other modules under similar situations. By blocking the handlers of child variables with `blockHandlers()` prior to updating the available values (and potentially the selected value), I could prevent the repeat calling of the helper routines (such as to plot the data successive times). Once the values had been updated, a second call to `unblockHandlers()` meant that the user interface would be once again responsive to updates to the child elements.

3.3.2.3 Observations in using gWidgets2 for User Interface Design

One observation I made during the implementation of the module, is lack of methods provided by `gWidgets2` to disable, hide, enable or show a number of components at the same time, without using the groups user interface object - an example of which is the Variable Information panel. When defining a user interface group object, borders are typically drawn on the screen, this also does not satisfy the need for objects that may be drawn on screen in distinct locations to share the same logic. Instead, enabling and disabling objects requires individual calls and additional logic to ensure the correct options are enabled or disabled.

3.4 Implementation of Longitudinal Module

The final implementation of the Longitudinal Module in iNZight involved the three-stage data processing model described in Section 3.3.1.3, with the exception of both variants of the Spaghetti Plot (Section 2.3.3) as the `longZight` plotting mechanisms for these plots that were developed during the project did not require any form of data processing apart from loading the data as a `longZight` class object and registering the individual identifier & time variables.

As previously discussed in Section 3.2, the implementation of the module followed the same principles that were identified as existing in the Time Series module and other interfaces in iNZight. Particularly, providing the linear top-to-bottom process flow was considered essential. As I had control of the data processing flow within my `longZight` package, I was able to pair that user interface decision with complementary functions that implemented the exact same process flow. For instance, the conversion of the loaded data to a `longZight` object occurred only when the values for the first two selection boxes had been selected by the user. Similarly the user interface was updated as more about the user's intentions were known (for instance filling in potential response variables, enabling subsetting or highlighting functionalities, and providing appropriate plotting options).

A copy of the source code for the Longitudinal Exploration Module is available in Appendix C.

Chapter 4

Development Process

4.1 Initial Prototyping

The prototyping process for plots discussed during this project initially began as basic R scripts or R Markdown documents to demonstrate the plotting capabilities of the various tools or packages been considered. These plots were generally very rigid and were not easily adaptable to other datasets.

As there were additional people interested in the outcome of the project, work was started on developing a Shiny[3] web application that could be regularly updated to demonstrate the latest work on the project.

Although I had prior experience developing Shiny web applications, in particular having developed an application for generating random ARIMA datasets and visualising them[1], the requirements for this project were significantly greater. However, the familiarity with the common design elements from Shiny did mean that going into the process I had a sense of what was generally achievable with the technologies available and could immediately start of translating the initial mockups into a format that could be dynamically modified, and new features be tested with.

At first, only the initial Alluvial plots were ready to be put into the Shiny application, but they were quickly expanded to include options to highlight variables (the final implementation of this feature is represented in Figure 2.3), gradually further test datasets were added (such as including the more complete versions of the CHNS datasets) to test features. Eventually all development of plotting ideas was performed (after tests in an R Console) directly into the Shiny application which provided the natural advantages of been able to rapidly create, deploy, share and obtain feedback about adjustments made to the plots.

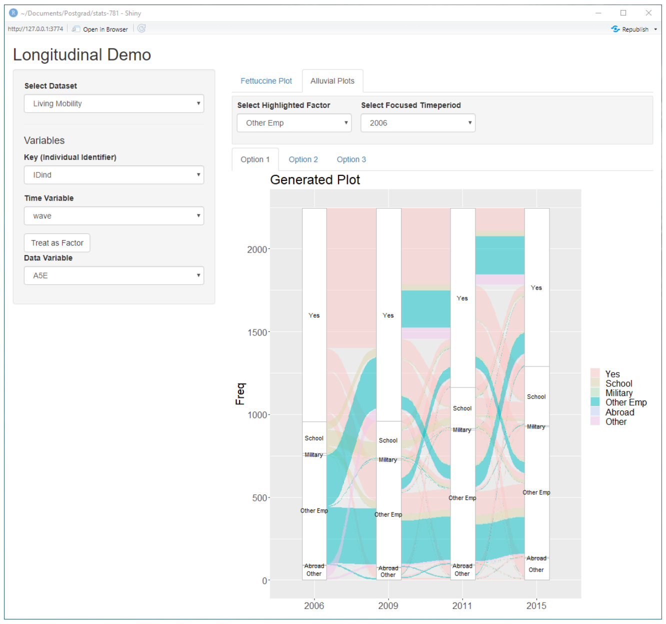
It was also at this point the first experiments with `plotly`[15] were conducted to assess the ease of which the plots could be adapted for iNZight Lite, or the release of a standalone

[1] `https://arsim.docker.stat.auckland.ac.nz/`

Shiny application as it was not known at that stage if the development of a iNZight or iNZight Lite module was going to be attainable by the end of the project due to constraints imposed by time and pressures due to the COVID-19 pandemic.

Mid-way through the development of the Shiny prototype application and at the point where additional plot types (such as Spaghetti) were considered for inclusion it was realised that the application required a significant rewrite to handle the potential for diverging code paths as the original design flow enforced the manipulation of data for both Alluvial and Lasagna plots simultaneously (regardless of the plot that might be requested by the user). At this point I decided that it was advantageous to start a new version of the application with data processing and plotting handled by functions outside of the Shiny `server` environment - instead relying on the dynamic nature of Shiny events and reactive expressions to trigger appropriate updates. A motivation of this was knowing that the likely development of iNZight specific code would require this type of program structure anyway.

Finally, the new version of the prototype application allowed additional focus to be placed on the interface design to replicate what an eventual iNZight Module would look like. The user interfaces from the first version of the Shiny application (Figure 4.1) and the second version (Figure 4.2) are provided below.


~/Documents/Postgrad/stats-781 - Shiny
http://127.0.0.1:3774

Longitudinal Demo

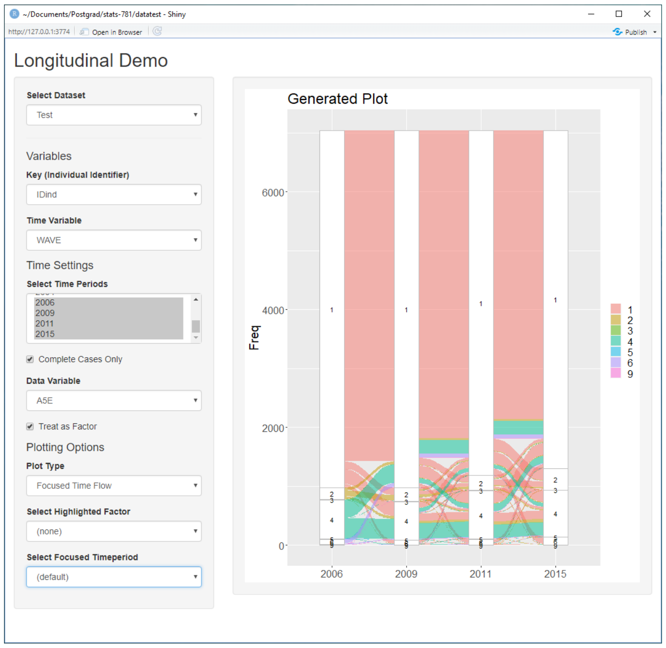
Select Dataset: Living Mobility
Variables
Key (Individual Identifier): IDind
Time Variable: wave
Data Variable: A5E

Select Highlighted Factor: Other Emp
Select Focused Timeperiod: 2006

Generated Plot

Figure 4.1: Screenshot of First Iteration of Shiny Application

To save time, some features were not completely translated over (such as the ability to Highlight specific levels or time periods) and the first application was kept as a reference for



Longitudinal Demo

Select Dataset: Test

Variables

Key (Individual Identifier): IDind

Time Variable: WAVE

Time Settings

Select Time Periods: 2006, 2009, 2011, 2015

✓ Complete Cases Only

Data Variable: A5E

✓ Treat as Factor

Plotting Options

Plot Type: Focused Time Flow

Select Highlighted Factor: (none)

Select Focused Timeperiod: (default)

Figure 4.2: Screenshot of Second Iteration of Shiny Application

their later implementation in either this app, or in the iNZight Module & R package which is what eventually occurred. Instead the time was focused on developing and implementing a proof of concept for the visualisation of Spaghetti plots - which were eventually included in the `longZight` package and module.

For reference, a copy of the second iteration of the Shiny application is provided as Appendix B.

4.2 longZight Package

The development of an R package to support the processing and visualisation of longitudinal data was planned from the beginning of the project. Although I have contributed a patch to an R package in the past, I was not generally familiar with the processes involved in creating or maintaining R packages, to remedy this I started to read the documentation provided by the Core R Team "*Writing R Extensions*"[14], however the documentation felt overly formal and did not provide enough of an overview, so I turned to Hadley Wickham & Jenny Bryan's excellent digital book "*R Packages*"[21].

The book recommended using the tools & functions provided by the `devtools`[24] and `usethis`[22] packages to start a new package, and then maintain it with the latter package providing helper functions such as `use_r()` & `use_test()` to create R source files and test files easily. Additionally the modern template configured the `roxygen2`[23] documentation system which automatically generates package documentation based on syntax provided in the comments of source files instead of needing to manually write and format `.Rd` files.

The vast majority of code that would be required to generate plots already existed but was embedded in the Shiny prototypes, in some cases they made heavy use of the `shiny::req()` function or addressing the `input` list. Additionally, because the structure of a Shiny web application can be rather ad-hoc and not in any particular order, this meant that decisions were required about how to best untangle and separate the code into suitable package ready-functions while also providing users with the most options. Additionally, while deciding on the best way to structure the application a suitable name for the package was somewhat elusive. As a result, I opted to create a test package to experiment and test ideas with while deciding on the package name. In the end I opted for the portmanteau of 'longitudinal' and 'iNZight' to form `longZight`, while this was always intended to be a placeholder the name has remained since and is unlikely to change prior to publication on CRAN.

An advantage of the `devtools` and `usethis` ecosystem was that the basic structure for a Git repository, including appropriate `.gitignore` files, was automatically configured. This meant that it was relatively easier than usual to setup a Git repository and it was used extensively during the development stage[2].

4.2.1 Package Structure

Ultimately, I decided that the best structure for the package was to set it up in a manner similar to the `tidyverse`[25] collection of packages which share a common syntax, a key reason for this is their overall popularity and the familiarity modern R users have with the syntax, practically this meant:

- Ensuring that the package would work with pipes from the `magrittr`[1] package (up until this point pipes had been extensively used in the source code as well).
- Providing syntax for functions in a manner similar to those that use the `tidyselect`[8] style of accepting variable selection.
- Ensuring that dynamic scripted use of the package functions remained accessible for the Longitudinal Visualisation Module.
- Ensuring end users still had the ability to customise plots generated by the package if they were familiar with `ggplot2`.

[2]Unfortunately, due to a configuration issue with the `git` client on one machine, many commits were attributed to my 'dead name'. As rewriting git history is unreliable, I have opted to only release a flattened history of the code.

- Ensuring users could store intermediary results so additional plots or modifications to plots could be generated quickly.

The requirement for the use of `tidyselect` syntax which allows users to refer to column names as if they were locally defined variables (such as by calling `dplyr::select(data, ColumnA, ColumnB)`) did cause problems at multiple stages of the project and are described further in 4.2.3.1, however ultimately the function structure in Code 3.1 (Section 3.3.1.1) was implemented. This allowed a logical chaining of functions by users to load, manipulate (if required) and plot data as required while satisfying all the end-user facing requirements for the package.

To distinguish data that has been loaded into the longZight dataset, and potentially manipulated or aggregated ready for plotting, three S3 classes were used (descriptions are provided in Section 4.2.2).

The naming of functions were generally named using lower snake case following the pattern of `[action]_long_[type]()`, the decision to name functions in this manner was namely to avoid potential namespace conflicts within R, but also so functions were named logically as to what they performed. Adding S3 methods onto existing functions such as `plot()` was considered but as data recoded into the formatted for Lasagna plots could also be used for Transitional and Trace Flow Alluvial plots this would've meant the need for additional options that would not typically be present in functions that provide an S3 `plot()` method. Additionally as the project primarily uses `ggplot2` it was felt that it could be too confusing for users, especially if non-standard syntax is used in the new method - using the `magrittr`-compliant `.data` input as the first parameter would automatically fall into this bucket.

The separation of functions into clearly defined actions for each type of data also meant that the readability of code referring to the package would be relatively clear as the function names become descriptive of what they will do - this is adapting the `tidyverse` design principles.

4.2.2 Data Classes

The `longZight` package provides three S3 classes for encapsulating data, in all cases these classes are placed on top of `tibble` objects from the `tibble`[11] package (these objects are enhanced `data.frame` objects and also inherit that class).

This decision was motivated by the need to ensure that important data was available to future functions downstream in the process and allowing users to theoretically only need to tell the system about variables one - for instance by providing the Individual Identifier and Time variable names in the `load_long_data()` function would result in an object with the primary S3 class of `longZight` that has the identifier and time parameters stored as attributes that can be retrieved by future functions that rely on `longZight` data.

As the primary class, the `longZight` methods for this class are provided by most functions in the function with the exception of the `plot_long_alluvial()` and `plot_long_lasagna()` functions which rely on the `longZightAlluv` and `longZightLasagna` classes which are set by the respective `recode_long_alluvial()` and `recode_long_lasagna()` functions.

4.2.3 Development Challenges

4.2.3.1 Implementing a Tidyverse-like Function Interface

One issue that arose during the development of the package was handling user input in a Tidyverse-like manner, particularly each of the calls to `load_long_data()` in Code 4.1 should be valid and processed correctly.

```
1  data %>% longZight::load_long_data(id, wave)
2  data %>% longZight::load_long_data("id", "wave")
3  data %>% longZight::load_long_data(idvar = "id", timevar = "wave")
4  theidvar = "id"
5  data %>% longZight::load_long_data(idvar = theidvar, timevar = "wave")
```

Code 4.1: Example of syntax that should be acceptable

Unfortunately there was significant confusion over the use of `rlang::ensym()` and `rlang::sym()` functions that are provided by the `rlang`[7] package and their exact manner of operation.

The Shiny web application had previously used code such as `id_param = rlang::sym(input$idvar)` to provide a variable based on a string that could then be referenced in data manipulation code in the manner `!!id_param`. This did not directly translate into the package correctly and after several days of debugging it was found that the `rlang::ensym()` function was more correct if you wanted to encode the value a function parameter was set to, without evaluating it rather than storing the function's defined name for that parameter.

This worked until I struck issues during the implementation of the iNZight module using these functions. The calls to the `longZight` package typically were of the form of lines 4 & 5 of Code 4.1 above. Although the code appeared to be defined correctly the data loading functions from `longZight` were producing errors stating that the variables could not be found - as the lists of variables were defined from the loaded data itself, there the error appeared to be illogical. It was made more difficult by the inability to attach a debugger to a running iNZight module, and I had to resort to placing `warnings()` as debugging printing in various places of both the package code, and the module code to understand the cause of the error.

Additionally, I later discovered that calling the function in the manner described in Line 3 of Code 4.1 can also result in the same issue.

The eventual solution was to add an extra parameter to most functions in `longZight` called `ts.args` which defaults to `TRUE` which allows the user – or the iNZight module –

to switch back to the `rlang::sym()` method of processing values provided to the function. This turned out to be one of the most frustrating problems encountered during the project.

4.2.3.2 Using Special Functions and Objects in Packages

The `longZight` package required the use of two sets of special functions from other packages that do not operate in traditional methods.

In the first case, the code I had originally written for the SHiny application made heavy use of pipes (`%>%` and `%<>%` from the `magrittr` package) for the data handling and manipulation functions. As such I felt that they would've been difficult to revert to more traditional assignment & call methods and I ended up deciding to retain them – while there are performance implications they appear to be relatively minimal and the inclusion of pipe operators in packages is becoming more common.

However, as function calls within packages cannot rely on any particular packages been loaded it was possible that the function calls could fail due to the internal requirement for the pipe operator. Additionally I suspected due to the nature of the pipe operator that calling it using `magrittr::`%>%`` may have caused unexpected problems or would not have worked at all. Instead I discovered the `@importFrom` keyword provided by `roxygen2` which placed an `importFrom(package, function)` entry in the `NAMESPACES` file that defines the functions available both within and from an R package.

An alternative approach was to load the entirety of the `magrittr` package into the environment but the packaging documentation and books had warned against doing this.

A similar issue arose when using `ggplot2` to create Alluvial plots, in one instance I needed to make references to the `stat_stratum` function from `ggalluvial`, however traditional ways of appending the `ggalluvial` namespace to the call did not work due to the way statistic functions are looked up by `ggplot2`. Once again, I had to define special imports for both the statistic function and the related `ggproto` class object.

4.2.4 Package Installation and Documentation

At present both my package and `brolgar` are not available on CRAN. To install my package from the GitHub repository (`https://github.com/snjnz/longzight`) the `remotes` package will be required, along with the calls in Code 4.2.

```r
remotes::install_github('njtierney/brolgar')
remotes::install_github('snjnz/longZight')
```

Code 4.2: Required calls to install `longZight`

Documentation for the package is available at `https://longzight.snj.nz` and is attached as Appendix D.

Chapter 5

Conclusion

5.1 Future Work

There are several tasks and features that I feel are missing from the completed project and was disappointed not to have the time to see to completion. In particular, I feel that in the immediate term the following issues are of the highest importance to resolve as part of the maintenance of the package and module:

- The inclusion of the package onto CRAN.

  At this stage the package will remain mostly unknown due to the lack of exposure on CRAN, there are several things that currently prohibit this, including the fact that the `brolgar` package is also not on CRAN and is defined as an import for the package. This may require reimplementing the calls required into the package.

- Several features from the Shiny Application did not make it into the Longitudinal Module for this report.

  I am particularly disappointed that I had to omit the "Select Time Periods" box - as visible in Figure 4.2 - due to lack of time, combined with uncertainty of the best way in which to implement it. Additionally, the `brolgar` features (that allow the highlighting of always increasing trends) are currently omitted from the Time Plots feature in the iNZight module due to unresolved problems with their final implementation in the `longZight` package - a potential solution to implement these is by making a callback available from the `longZight` function that lists appropriate features for the data provided that can then be included in the interface.

- General tidying of variable selection boxes.

  In most cases the module does not prevent the user selecting variables that are already selected elsewhere. As an example at present it is currently possible to select the same variable as the individual identifier and time period variables which does not make sense given the definition of longitudinal data (Section 1.2).

In general these issues and work items are of vital important before widespread distribution of the package and module and will improve the user experience drastically.

In addition, in the longer term of the package development and maintenance cycle I would like to make the following adjustments, additions or improvements:

- The inclusion of Lasagna plots for continuous time cases is still worthwhile despite the potential problems if there are many objects printed on screen (Section 2.3.2)
- Improvements to the Transitional Plots to add parity with the features that are available for Spaghetti plots
- The addition of further plot customisation options and also linking into the defaults that iNZight hold for plot colour schemes and presenting users with R code so they can generate their own plots
- Inclusion into iNZight Lite so that macOS users are able to access the module.
- Instead of greying out irrelevant elements of the user interface (Section 3.2.1), hiding the elements should be considered.
- Additional software tests for the `longZight` package using the `testthat`[20] testing framework are required. The creation of tests during development was mostly omitted and as a result the testing coverage for the package is estimated using `covr`[9] at around 8%. Having reliable tests available in the near future will reduce the risk of introducing bugs or regressions in behaviour as the package matures.

5.2 Summary

The project has resulted in a total of eight different plotting options for a wide range of longitudinal data implemented within an R package and module for iNZight that can in theory be used today. These plots cover an area of data visualisation that has largely been ignored or proven too difficult to provide good, easy-to-use graphics for and fills a niche for users from a wide range of industry sectors. As the adoption of iNZight as a visualisation tool increases, I expect the available of the module will have a positive impact on more and more people's jobs.

5.2.1 Closing Remarks

Adapting existing plotting methods for their inclusion into iNZight took significantly more work than I had expected, and I was sceptical of the initial doubts from my Project Supervisor that we would get to the stage of implementing an actual module for iNZight. At the conclusion of the project and reviewing the code, processes and iterations required to get to this stage I can now understand why.

There are still many items on the Future Work list above, and I am disappointed some of these features had to be omitted, but I also feel that this list pales in comparison to what was implemented. Finally, to me the list actually represents what is the typical final polishing of code and interfaces that comes with developing something new, with minimal impact on the worthiness of what was produced.

I am grateful for the opportunity to work on this small part of iNZight, and to improve my knowledge of a wide area of R including generally, packaging, Shiny. Finally, at the conclusion of the project I now have a different perspective and understanding of the use cases that the plot types I have worked with have and data visualisation in general. Ultimately, this project has had a positive impact on how I view data visualisation and also changed how I tutor Statistics and describe the importance and purposes of visualising data to students.

Chapter 6

References

Cited Works

[4] Hadrien Commenges, Pierre Pistre, and Robin Cura. "SLIDER: Software for LongItudinal Data Exploration with R". en. In: *Cybergeo : European Journal of Geography* (Nov. 2014). Publisher: CNRS-UMR Géographie-cités 8504. ISSN: 1278-3366. DOI: 10.4000/cybergeo.26530. URL: http://journals.openedition.org/cybergeo/26530.

[5] David Crowther. *Old Habits Die Hard*. May 2020. URL: https://www.chartr.co/newsletters/old-habits-die-hard.

[10] W. J. Kleindl, M. C. Rains, L. A. Marshall, and F. R. Hauer. "Fire and flood expand the floodplain shifting habitat mosaic concept". In: *Freshwater Science* 34.4 (Nov. 2015). Publisher: The University of Chicago Press, pp. 1366–1382. ISSN: 2161-9549. DOI: 10.1086/684016. URL: https://www.journals.uchicago.edu/doi/10.1086/684016.

[14] R Core Team. *Writing R Extensions*. 2020. URL: https://cran.r-project.org/doc/manuals/r-release/R-exts.html.

[21] Hadley Wickham and Jennifer Bryan. *Hadley Wickham, Jennifer Bryan*. 2nd ed. 2020. URL: https://r-pkgs.org/.

Cited R Packages

[1] Stefan Milton Bache and Hadley Wickham. *magrittr: A Forward-Pipe Operator for R*. R package version 1.5. 2014. URL: https://CRAN.R-project.org/package=magrittr.

[2] Jason Cory Brunson. "ggalluvial: Layered Grammar for Alluvial Plots". In: *Journal of Open Source Software* 5.49 (2020), p. 2017. DOI: 10.21105/joss.02017.

[3] Winston Chang, Joe Cheng, JJ Allaire, Yihui Xie, and Jonathan McPherson. *shiny: Web Application Framework for R*. R package version 1.5.0. 2020. URL: https://CRAN.R-project.org/package=shiny.

[6] Douglas Nychka, Reinhard Furrer, John Paige, and Stephan Sain. *fields: Tools for spatial data*. R package version 11.6. Boulder, CO, USA: University Corporation for Atmospheric Research, 2017. DOI: 10.5065/D6W957CT. URL: https://github.com/NCAR/Fields.

[7] Lionel Henry and Hadley Wickham. *rlang: Functions for Base Types and Core R and 'Tidyverse' Features*. R package version 0.4.7. 2020. URL: https://CRAN.R-project.org/package=rlang.

[8] Lionel Henry and Hadley Wickham. *tidyselect: Select from a Set of Strings*. R package version 1.1.0. 2020. URL: https://CRAN.R-project.org/package=tidyselect.

[9] Jim Hester. *covr: Test Coverage for Packages*. R package version 3.5.1. 2020. URL: https://CRAN.R-project.org/package=covr.

[11] Kirill Müller and Hadley Wickham. *tibble: Simple Data Frames*. R package version 3.0.4. 2020. URL: https://CRAN.R-project.org/package=tibble.

[12] Mitchell O'Hara-Wild, Rob Hyndman, and Earo Wang. *fabletools: Core Tools for Packages in the 'fable' Framework*. R package version 0.2.1. 2020. URL: https://CRAN.R-project.org/package=fabletools.

[13] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2020. URL: https://www.R-project.org/.

[15] Carson Sievert. *Interactive Web-Based Data Visualization with R, plotly, and shiny*. Chapman and Hall/CRC, 2020. ISBN: 9781138331457. URL: https://plotly-r.com.

[16] Bruce Swihart. *Lasagna plots in base and ggplot graphics*. R package version 0.2. 2016. URL: https://github.com/swihart/lasagnar.

[17] Nicholas Tierney, Di Cook, and Tania Prvan. *brolgar: BRowse Over Longitudinal data Graphically and Analytically in R*. R package version 0.0.6.9100. 2020. URL: https://github.com/njtierney/brolgar.

[18] John Verzani. *gWidgets2: Rewrite of gWidgets API for Simplified GUI Construction*. R package version 1.0-8. 2019. URL: https://CRAN.R-project.org/package=gWidgets2.

[19] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN: 978-3-319-24277-4. URL: `https://ggplot2.tidyverse.org`.

[20] Hadley Wickham. “testthat: Get Started with Testing”. In: *The R Journal* 3 (2011), pp. 5–10. URL: `https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf`.

[22] Hadley Wickham and Jennifer Bryan. *usethis: Automate Package and Project Setup*. R package version 1.6.3. 2020. URL: `https://CRAN.R-project.org/package=usethis`.

[23] Hadley Wickham, Peter Danenberg, Gábor Csárdi, and Manuel Eugster. *roxygen2: In-Line Documentation for R*. R package version 7.1.1. 2020. URL: `https://CRAN.R-project.org/package=roxygen2`.

[24] Hadley Wickham, Jim Hester, and Winston Chang. *devtools: Tools to Make Developing R Packages Easier*. R package version 2.3.2. 2020. URL: `https://CRAN.R-project.org/package=devtools`.

[25] Hadley Wickham et al. “Welcome to the tidyverse”. In: *Journal of Open Source Software* 4.43 (2019), p. 1686. DOI: `10.21105/joss.01686`.

[26] Achim Zeileis, Jason C. Fisher, Kurt Hornik, Ross Ihaka, Claire D. McWhite, Paul Murrell, Reto Stauffer, and Claus O. Wilke. *colorspace: A Toolbox for Manipulating and Assessing Colors and Palettes*. arXiv 1903.06490. arXiv.org E-Print Archive, 2019. URL: `http://arxiv.org/abs/1903.06490`.

6.1 CHNS Data Acknowledgment

This research uses data from China Health and Nutrition Survey (CHNS). We are grateful to research grant funding from the National Institute for Health (NIH), the Eunice Kennedy Shriver National Institute of Child Health and Human Development (NICHD) for R01 HD30880, National Institute of Diabetes and Digestive and Kidney Diseases (NIDDK) for R01DK104371 and R01HL108427, the NIH Fogarty grant D43 TW009077 for financial support for the CHNS data collection and analysis files since 1989, and the China-Japan Friendship Hospital, Ministry of Health for support for CHNS 2009, Chinese National Human Genome Center at Shanghai since 2009, and Beijing Municipal Center for Disease Prevention and Control since 2011. We thank the National Institute for Nutrition and Health, China Center for Disease Control and Prevention, Beijing Municipal Center for Disease Control and Prevention, and the Chinese National Human Genome Center at Shanghai.

More Information about the CHNS is available at `https://www.cpc.unc.edu/projects/china`

Appendix A

Example Datasets

A.1 Data Listing for Example Spaghetti Plot

```
# A tibble: 9 x 3
     id  time     y
  <int> <dbl> <dbl>
1     1  11.9  53.1
2     1  13.8  47.9
3     1  22.1  58.4
4     2  11.9  54.0
5     2  13.8  44.1
6     2  22.1  46.2
7     3  11.9  50.8
8     3  13.8  49.9
9     3  22.1  47.1
```

Code A.1: Data listing for example Spaghetti Plot (Figure 1.1)

A.2 Synthetic Transitional Dataset

```
set.seed(1234)
time <- rnorm(1000, mean = 120, sd = 20)
id <- rep(1:200, each = 5)
cat.d <- factor(rep(LETTERS[1:5], 200))
synth.data <- tibble(id, time) %>%
  arrange(id,time) %>%
  mutate(cat = cat.d)
```

Code A.2: Code to Generate the Synthetic Transitional Dataset

A.3 Additional Code

A.3.1 `longZight::alluvial_fill_discrete`

```r
alluvial_fill_discrete <- function(..., h = c(0, 360) + 15, c = 100, l = 65, h.start = 0, direction = 1, na.value = "grey50", aesthetics = "fill", focus = 1) {
  pal <- scales::hue_pal(h, c, l, h.start, direction)
  ggplot2::discrete_scale(aesthetics, "hue", (function(n) {
    orig.colours <- pal(n)
    lighter.colours <- colorspace::lighten(orig.colours, amount = 0.6, method = "relative")
    lighter.colours[focus] <- orig.colours[focus]
    return(lighter.colours)
  }), na.value = na.value, ...)
}
```

Code A.3: Copy of the `alluvial_fill_discrete` function developed for `longZight`

Appendix B

Shiny Prototype Application Source

For reference purposes the second iteration of the Shiny Application used to prototyping plot development is provided below.

```r
library(shiny)
library(plotly)
library(tidyverse)
library(haven) # SAS
library(ggplot2)
library(ggalluvial)
library(shinyjs)
library(magrittr)
library(ggthemes)
library(brolgar)
library(data.table)

wage_data <- brolgar::wages %>%
    as_tibble %>%
    dplyr::mutate(black = as.logical(black),
                  hispanic = as.logical(hispanic),
                  ged = as.logical(ged),
                  high_grade = as.factor(high_grade))

wage_data2 <- brolgar::wages %>%
  as_tibble  %>%
  dplyr::mutate(black = as.logical(black),
                hispanic = as.logical(hispanic),
                ged = as.logical(ged),
                high_grade = as.factor(high_grade))

set.seed(1234)
time <- rnorm(1000, mean = 120, sd = 20)
id <- rep(1:200, each = 5)
cat.d <- factor(rep(LETTERS[1:5], 200))
my.data <- tibble(id, time) %>%
  arrange(id,time) %>%
  mutate(cat = cat.d)

```

```r
refactor_wave <- function(wave) {
    waves <- sort(unique(wave))
    return(factor(wave, levels = waves))
}

ymin_calc <- function(n) {
    ymins <- lag(cumsum(n))
    if (length(ymins) > 0) {
        ymins[1] <- 0
    }
    return(ymins)
}

loaddata <- function(fileobj) {
    return(readr::read_csv(fileobj))
}

data_rearrange <-
    function(dataobj,
             id_param,
             values_param,
             wave_param) {
        id_param <- dplyr::sym(id_param)
        values_param <- dplyr::sym(values_param)
        wave_param <- dplyr::sym(wave_param)
        dataobj %<>%
            tidyr::drop_na() %>%
            dplyr::arrange(!!wave_param) %>%
            dplyr::mutate(id = !!id_param,
                          wave = refactor_wave(!!wave_param),
                          value = !!values_param) %>%
            dplyr::select(-!!values_param)

        dataobj %<>%
            dplyr::left_join(
                dataobj %>%
                    dplyr::group_by(wave,value) %>%
                    dplyr::summarise(n = n()) %>%
                    dplyr::mutate(sortorder = rank(1/(c(n)+1), ties.method = "last"))
                         %>%
                    dplyr::select(-n)
            )

        waves_vect <- tail(levels(dataobj$wave), 4)
        chart_waves <- syms(paste0("value_", waves_vect))
        chart_sort <- syms(paste0("sortorder_", waves_vect))

        values_levels <- levels(factor(dataobj$value))

        return(list(
            data = dataobj %>%
                tidyr::pivot_wider(names_from = wave, values_from =c(value, sortorder
                    )) %>%
                dplyr::select(id, !!!chart_waves, !!!chart_sort) %>%
                tidyr::drop_na(),
            chart_waves = chart_waves,
            chart_sort = chart_sort,
            waves_vect = waves_vect,
            values_levels = values_levels
        ))
    }

data_fet <- function(dataobj, fet_type) {
    sortorders <- dataobj$data %>%
        select(-id) %>%
        unique

    return(
        dataobj$data %>%
            dplyr::group_by(!!!dataobj$chart_waves) %>%
            dplyr::summarise(n = n()) %>%
            dplyr::ungroup() %>%
            dplyr::left_join(sortorders) %>%
            { if (fet_type == 1) arrange(., desc(n)) else if (fet_type == 2) arrange
                (., !!!dataobj$chart_sort) else . } %>%
            dplyr::mutate(
                ymax = cumsum(n),
                ymin = ymin_calc(n),
                alluvid = row_number()
            ) %>%
            tidyr::pivot_longer(
                cols = starts_with("value_"),
                names_to = "wave",
                names_prefix = "value_",
                values_to = "place"
            ) %>%
            dplyr::mutate(
                wave = as.factor(wave),
                xmin = as.numeric(wave) - 0.5,
                xmax = as.numeric(wave) + 0.5
            )
    )
}

data_alluv <- function(dataobj) {
    return(
        dataobj$data %>%
            dplyr::group_by(!!!dataobj$chart_waves) %>%
            dplyr::summarise(Freq = n()) %>%
            dplyr::ungroup()
    )
}

healthcare_data <- haven::read_sas("../data/hlth_12.sas7bdat") %>%
    dplyr::mutate(
        M23 = factor(M23, levels = c(0, 1), labels = c("No", "Yes")),
        wave = factor(wave)
    ) %>%
    dplyr::select(IDind, M23, wave)

indiv_data <- haven::read_sas("../data/rst_12.sas7bdat") %>%
    dplyr::mutate(
        A5E = factor(
            A5E,
            levels = 1:6,
            labels = c("Yes", "School", "Military", "Other Emp", "Abroad", "Other")
        ),
        wave = factor(WAVE)
    ) %>%
    dplyr::select(IDind, A5E, wave)

indiv_data2 <- haven::read_sas("../data/rst_12.sas7bdat")

rotate <- function(x) {
    (x + h.start) %% 360 * direction
}
force_all <- function(...) {
    list(...)
}
new_huepal <-
    function(h = c(0, 360) + 15,
             c = 100,
             l = 65,
             h.start = 0,
             direction = 1,
             focus = 1) {
        stopifnot(length(h) == 2)
        stopifnot(length(c) == 1)
        stopifnot(length(l) == 1)
        force_all(h, c, l, h.start, direction)
        function(n) {
            if (n == 0) {
                stop("Must request at least one colour from a hue palette.",
                     call. = FALSE
                )
            }
            if ((diff(h) %% 360) < 1) {
                h[2] <- h[2] - 360 / n
            }
            rotate <- function(x) {
                (x + h.start) %% 360 * direction
            }
            hues <- rotate(seq(h[1], h[2], length.out = n))
            c2 <- rep(c / 3, n)
            l2 <- rep(l + 20, n)
            c2[focus] <- c
            l2[focus] <- l
            hcl <- cbind(hues, c2, l2)
            farver::encode_colour(hcl, from = "hcl")
        }
    }

new_scale_fill_discrete <- function(..., h = c(0, 360) + 15, c = 100, l = 65, h.start
     = 0, direction = 1, na.value = "grey50", aesthetics = "fill", focus = 1) {
    ggplot2::discrete_scale(aesthetics, "hue", new_huepal(h, c, l, h.start, direction
        , focus), na.value = na.value, ...)
}

```

```r
data_to_fet <- function(input, proc_data, sort_type = 0) {
  datatypes <- proc_data %>% dplyr::summarise_all(class)
  datavars <- c(input$sel_wavevar, input$sel_datavar, input$sel_indivvar)
  shiny::req(datavars %in% colnames(proc_data))
  shiny::req(pull(datatypes[input$sel_wavevar]) == "factor", pull(datatypes[input$sel_datavar]) == "factor")

  subset_data <- proc_data %>% select(!!!syms(datavars))

  id_param <- dplyr::sym(datavars[3])
  values_param <- dplyr::sym(datavars[2])
  wave_param <- dplyr::sym(datavars[1])
  subset_data %<>%
    rename(id = !!id_param, wave = !!wave_param, value = !!values_param)


  subset_data %<>% dplyr::arrange(wave) %>% drop_na() %>% mutate(wave = fct_drop(wave))
  waves_vect <- levels(pull(subset_data, wave))
  chart_waves <- syms(paste0("value_", waves_vect))
  chart_sort <- syms(paste0("sortorder_", waves_vect))
  values_levels <- levels(pull(subset_data, value))
  subset_data_w <- subset_data %>%
    tidyr::pivot_wider(names_from = wave, values_from= value, names_glue = "value_{wave}") %>%
    {
      if (input$sel_completecases) tidyr::drop_na(.) else mutate_if(., is.factor, forcats::fct_explicit_na)
    } %>%
    dplyr::group_by(!!!chart_waves) %>%
    dplyr::summarise(n = n()) %>%
    ungroup()
  subset_data_l <- subset_data_w %>%
    mutate(id = dplyr::row_number()) %>%
    tidyr::pivot_longer(
      cols = starts_with("value_"),
      names_to = "wave",
      names_prefix = "value_",
      values_to = "value"
    )
  sortorders <- subset_data_l  %>% group_by(wave, value) %>% summarise(n = sum(n)) %>% mutate(sortorder = rank(1/(c(n)+1), ties.method = "last")) %>%
    dplyr::select(-n)
  subset_data_l %>%
    left_join(sortorders) %>%
    tidyr::pivot_wider(names_from = wave, values_from =c(value, sortorder)) %>%
    dplyr::select(id, n, !!!chart_waves, !!!chart_sort) %>%
    {
      if (sort_type == 1) arrange(., desc(n)) else if (sort_type == 2) arrange(., !!!chart_sort) else .
    } %>%
    dplyr::mutate(
      ymax = cumsum(n),
      ymin = ymin_calc(n),
      alluvid = row_number()
    ) %>%
    tidyr::pivot_longer(
      cols = starts_with("value_"),
      names_to = "wave",
      names_prefix = "value_",
      values_to = "place"
    ) %>%
    dplyr::mutate(
      wave = as.factor(wave),
      xmin = as.numeric(wave) - 0.5,
      xmax = as.numeric(wave) + 0.5
    )
}


doAlluvialPlot <- function(input, proc_data) {
  if (input$sel_plottype == 1) {
    datatypes <- proc_data %>% dplyr::summarise_all(class)
    datavars <- c(input$sel_wavevar, input$sel_datavar, input$sel_indivvar)
    shiny::req(datavars %in% colnames(proc_data))
    shiny::req(pull(datatypes[input$sel_wavevar]) == "factor", pull(datatypes[input$
        sel_datavar]) == "factor")

    subset_data <- proc_data %>% select(!!!syms(datavars))

    id_param <- dplyr::sym(datavars[3])
    values_param <- dplyr::sym(datavars[2])
    wave_param <- dplyr::sym(datavars[1])
    subset_data %<>%
      rename(id = !!id_param, wave = !!wave_param, value = !!values_param)

    subset_data %<>%
      dplyr::left_join(
        subset_data %>%
          dplyr::group_by(wave, value) %>%
          dplyr::summarise(n = n()) %>%
          dplyr::mutate(sortorder = rank(1 / (c(n) + 1), ties.method = "last")) %>%
          dplyr::select(-n)
      ) %>%
      dplyr::arrange(wave) %>%
      tidyr::drop_na() %>%
      mutate(wave = forcats::fct_drop(wave))

    waves_vect <- levels(pull(subset_data, wave))
    chart_waves <- syms(paste0("value_", waves_vect))
    chart_sort <- syms(paste0("sortorder_", waves_vect))
    values_levels <- levels(pull(subset_data, value))

    subset_data %<>%
      tidyr::pivot_wider(names_from = wave, values_from = c(value, sortorder)) %>%
      {
        if (input$sel_completecases) tidyr::drop_na(.) else mutate_if(., is.factor,
            forcats::fct_explicit_na)
      } %>%
      dplyr::group_by(!!!chart_waves) %>%
      dplyr::summarise(Freq = n()) %>%
      dplyr::ungroup()


    axes <- sapply(seq_len(length(waves_vect)), function(x) setNames(list(paste0("
        value_", waves_vect[x])), paste0("axis", x)))
    labels <- waves_vect

    subset_data %>%
      ggplot2::ggplot(do.call(ggplot2::aes_string, c(list(y = "Freq"), axes))) +
      ggalluvial::geom_flow(ggplot2::aes(fill = get(paste0("value_", waves_vect[1])))
          ) +
      ggalluvial::geom_stratum(colour = "grey") +
      ggplot2::scale_x_discrete(limits = labels) +
      ggplot2::ggtitle("Generated Plot") +
      ggplot2::scale_fill_discrete() +
      ggplot2::geom_text(stat = "stratum", infer.label = TRUE) +
      ggplot2::theme(
        legend.title = ggplot2::element_blank(),
        text = ggplot2::element_text(size = 20)
      )
  }
  else {
    subset_data <- data_to_fet(input, proc_data)
    if (input$sel_plottype == 2) {
      subset_data %>%
        ggplot2::ggplot(aes(x = wave, stratum = place, alluvium = alluvid, y = n,
            label = place, fill = place)) +
        ggalluvial::geom_flow(aes(fill = place)) +
        ggalluvial::geom_stratum(alpha = 1) +
        ggplot2::geom_text(stat = "stratum") +
        ggplot2::ggtitle("Generated Plot") #+
        #fillfunc
    }
    else if (input$sel_plottype == 3) {
      subset_data %>%
        ggplot2::ggplot(aes(x = wave, stratum = place, alluvium = alluvid, y = n,
            label = place, fill = place)) +
        # geom_flow(aes(fill = place)) +
        ggalluvial::geom_alluvium(aes(fill = place)) +
        ggalluvial::geom_stratum(alpha = 1) +
        ggplot2::geom_text(stat = "stratum") +
        ggplot2::ggtitle("Generated Plot") #+
        #fillfunc
    }
  }
}

doStripPlot <- function(input, proc_data) {
  p <- data_to_fet(input, proc_data, input$sel_fettuccinetype) %>%
    ggplot2::ggplot(aes(x = wave, y = n)) +
    ggplot2::geom_rect(aes(
      x = wave, y = n, xmin = xmin, xmax = xmax,
      ymin = ymin, ymax = ymax, fill = place
    )) +
    ggplot2::scale_fill_brewer(palette = "Paired")
  print(plotly::ggplotly(p))
}
```

```r

doTimePlot <- function(input, proc_data) {
  datatypes <- proc_data %>% dplyr::summarise_all(class)
  datavars <- c(input$sel_wavevar, input$sel_datavar, input$sel_indivvar)
  shiny::req(datavars %in% colnames(proc_data))
  id_param <- dplyr::sym(datavars[3])
  values_param <- dplyr::sym(datavars[2])
  wave_param <- dplyr::sym(datavars[1])

  if ((pull(datatypes[input$sel_wavevar]) == "numeric") && (pull(datatypes[input$sel_
      datavar]) == "numeric")) {
    plotdata <- proc_data %>%
      as_tsibble(key = !!id_param,
               index = !!wave_param,
               regular = FALSE)
    plotdata %<>%
      left_join( plotdata %>%
      features(!!values_param, feat_monotonic))

    aes_gen <- ggplot2::aes_string(x = input$sel_wavevar, y = input$sel_datavar,
        group = input$sel_indivvar)
    if (input$sel_subsetvar1 != "0" && input$sel_subsetvar1 %in% colnames(plotdata))
      aes_ss1 <- ggplot2::aes_string(x = input$sel_wavevar, y = input$sel_datavar,
          group = input$sel_indivvar, col = input$sel_subsetvar1)
    else
      aes_ss1 <- NULL

    tmp_data1 <- plotdata
    if (input$sel_subsetvar2 != "0" && input$sel_subsetvar2 %in% colnames(plotdata))
        {
      tmp_data1 %<>% filter(!(!!dplyr::sym(input$sel_subsetvar2)))
      tmp_data2 <- plotdata %>% filter(!!dplyr::sym(input$sel_subsetvar2))
    } else {
      tmp_data2 <- NULL
    }

    p <- ggplot2::ggplot()
    if (!is.null(aes_ss1)) {
      if (!is.null(tmp_data2)) {
        p <- p + ggplot2::geom_line(aes_gen, tmp_data1, col = "grey") +
          ggplot2::geom_line(aes_ss1, tmp_data2)
      }
      else
        p <- p + ggplot2::geom_line(aes_ss1, tmp_data1)
    } else {
      if (!is.null(tmp_data2)) {
        p <- p + ggplot2::geom_line(aes_gen, tmp_data1, col = "grey") +
          ggplot2::geom_line(aes_gen, tmp_data2)
      }
      else
        p <- p + ggplot2::geom_line(aes_gen, tmp_data1)
    }
    p <- p + ggthemes::scale_color_colorblind()
    return(plotly::ggplotly(p))
    #return(print(p))
  } else {
    value_data <- pull(proc_data, !!values_param)
    p <- proc_data %>%
      left_join(proc_data %>%
                  group_by(!!id_param) %>%
                  summarise(firsttime = min(!!wave_param)) %>%
                  ungroup() %>%
                  arrange(firsttime) %>%
                  select(-firsttime) %>%
                  mutate(int.id = row_number())) %>%
      mutate(dodge = seq(-0.25, 0.25, length.out = max(int.id))[int.id], ypos = as.
          numeric(!!values_param) + dodge) %>%
      ggplot(aes_string(input$sel_wavevar, "ypos", group = input$sel_indivvar)) +
      geom_step(alpha = 0.25, direction = 'hv') +
      scale_y_continuous(breaks = sort(unique(as.numeric(value_data))), labels =
          levels(value_data))
    return(plotly::ggplotly(p))
  }
}

# Define UI for application that draws a histogram
ui <- shiny::fluidPage( # Application title
    useShinyjs(),
    shiny::titlePanel("Longitudinal Demo"),

    # Sidebar with a slider input for number of bins
    shiny::sidebarLayout(
        shiny::sidebarPanel(
            shiny::selectInput(
                "sel_dataset",
                label = "Select Dataset",
                choices = c(
                    "From CSV" = 0,
                    "Injuries" = 1,
                    "Living Mobility" = 2,
                    "Test" = 3,
                    "Test (Wages)" = 4,
                    "Test (Simulated)" = 5,
                    "Wages" = 101
                ),
                selected = 0,
                selectize = F
            ),
            shiny::conditionalPanel(
                condition = "input.sel_dataset == 0",
                shiny::fileInput(
                    "csvupload",
                    "Select File",
                    multiple = FALSE,
                    accept = c(
                        "text/csv",
                        "text/comma-separated-values,text/plain",
                        ".csv"
                    )
                ),
            ),
            shiny::tags$hr(),
            shiny::conditionalPanel(
                condition = "input.sel_dataset <= 100",
                shiny::h4("Variables"),
                shiny::selectInput(
                    "sel_indivvar",
                    label = "Key (Individual Identifier)",
                    choices = c("(pick)" = 0),
                    selected = "",
                    selectize = F
                ),
                shiny::selectInput(
                    "sel_wavevar",
                    label = "Time Variable",
                    choices = c("(pick)" = 0),
                    selected = 0,
                    selectize = F
                ),
                shinyjs::hidden(shiny::checkboxInput(
                    "factorize_wavevar",
                    label = "Treat as Factor",
                    FALSE
                )),
                shinyjs::hidden(shiny::checkboxInput(
                  "show_wavesel",
                  label = "Foo",
                  value = FALSE
                )),
                shiny::conditionalPanel(
                  condition = "input.show_wavesel",
                  shiny::h4("Time Settings"),
                  shiny::selectInput(
                    "sel_waves",
                    label = "Select Time Periods",
                    choices = c("(none)" = 0),
                    selectize = FALSE,
                    multiple = TRUE
                  ),
                  shiny::checkboxInput(
                    "sel_completecases",
                    label = "Complete Cases Only",
                    TRUE
                  )
                ),
                shiny::selectInput(
                    "sel_datavar",
                    label = "Data Variable",
                    choices = c("(pick)" = 0),
                    selected = 0,
                    selectize = F
                ),
                shinyjs::hidden(shiny::checkboxInput(
                  "factorize_datavar",
                  label = "Treat as Factor",
                  FALSE
                )),
            ),
            shiny::h4("Plotting Options"),
            shiny::selectInput(
              "sel_plottype",
              label = "Plot Type",
              choices = c("(none)" = 0),
              selected = "",
              selectize = F
            ),
            shiny::conditionalPanel(
              condition = "input.sel_plottype == 4",
              shiny::selectInput(
                "sel_fettuccinetype",
                label = "Sorting Method",
                choices = c("By Count" = 1,
                            "By 'Block'" = 2,
                            "By Factor" = 3),
                selected = 1,
                selectize = FALSE
              )
            ),
            shiny::conditionalPanel(
              condition = "input.sel_plottype >= 1 && input.sel_plottype < 4",
              shiny::selectInput(
                "sel_highlight",
                label = "Select Highlighted Factor",
                choices = c("(none)" = 0),
                selected = "(none)",
                selectize = F
              ),
              shiny::selectInput(
                "sel_focustime",
                label = "Select Focused Timeperiod",
                choices = c("(default)" = 0),
                selected = "(default)",
                selectize = F
              )
            ),
            shiny::conditionalPanel(
              condition = "input.sel_plottype == 5",
              shiny::selectInput(
                "sel_subsetvar1",
                label = "Subsetting Variable 1 (Colour)",
                choices = c("(pick)" = 0),
                selected = "",
                selectize = F
              ),
              shiny::selectInput(
                "sel_subsetvar2",
                label = "Subsetting Variable 2 (Highlight)",
                choices = c("(pick)" = 0),
                selected = "",
                selectize = F
              )
            )
        ),

        # Show a plot of the generated distribution
        shiny::mainPanel(
          shiny::wellPanel(id = "ggplotOutput",
            shiny::plotOutput("ggplotPlot", height = "800px")
          ),
            shiny::wellPanel(id = "plotlyOutput",
              plotly::plotlyOutput("plotlyPlot", height = "800px")
            )
        )
    )
)

# Define server logic required to draw a histogram
server <- function(session, input, output) {
    dataset <- shiny::reactiveVal()
    raw_data <- shiny::reactiveVal()
    proc_data <- shiny::reactiveVal()
    output$foo <- shiny::renderText(paste(colnames(raw_data)))
    num_data <- shiny::reactiveVal()
    num_tib <- shiny::reactiveVal()
    resetProcess <- shiny::reactiveVal()

    shiny::observeEvent(input$sel_dataset, {
        req(input$sel_dataset)
        ds <- input$sel_dataset
        resetProcess(TRUE)
        proc_data(NULL)

        if (ds == 0) {
            raw_data(NULL)
        } else if (ds == 1) {
            raw_data(healthcare_data)
        } else if (ds == 2) {
            raw_data(indiv_data)
        } else if (ds == 3) {
            raw_data(indiv_data2)
        } else if (ds == 4) {
          raw_data(wage_data2)
        } else if (ds == 5) {
          raw_data(my.data)
        } else if (ds == 101) {
            num_data(wage_data)
        }
        shinyjs::toggleState("csvupload", ds == 0)

        for (i in c("factorize_wavevar", "factorize_datavar")) {
          shiny::updateCheckboxInput(session, i, value=FALSE)
        }
        resetProcess(NULL)
    })

    shiny::observeEvent(input$factorize_wavevar, {
        req(!is.null(raw_data()), input$sel_wavevar != 0)

        wavevar = sym(input$sel_wavevar)
        raw_data(raw_data() %>%
                     mutate(!!wavevar := refactor_wave(!!wavevar)))
   })

    shiny::observeEvent(input$csvupload, {
        tryCatch(
            {
                raw_data(loaddata(input$csvupload$datapath))
            },
            error = function(e) {
                raw_data(NULL)
                stop(safeError(e))
            }
        )
    })

    shiny::observe({
        if (!is.null(raw_data())) {
            rd_names <- colnames(raw_data())
        } else {
            rd_names <- c()
        }
        for (i in c("sel_indivvar", "sel_datavar", "sel_wavevar")) {
            cur <- input[[i]]
            shiny::updateSelectInput(session,
                                     i,
                                     choices = c("(pick)" = 0, rd_names),
                                     selected = (if (cur %in% rd_names) cur else 0)
            )
        }

    })

    shiny::observe({
      req(!is.null(raw_data()))
      d <- raw_data()
      if (input$sel_wavevar != 0) {
        waved <- pull(d,!!sym(input$sel_wavevar))
        shinyjs::toggleElement("factorize_wavevar", condition = !(is.factor(waved)))
        shiny::updateCheckboxInput(session, "show_wavesel", value = is.factor(waved)
            || input$factorize_wavevar)
      } else {
        shiny::updateCheckboxInput(session, "show_wavesel", value = FALSE)
      }
      if (input$sel_datavar != 0) {
        datad <- pull(d, !!sym(input$sel_datavar))
        shinyjs::toggleElement("factorize_datavar", condition = !(is.factor(datad)))
      }
    })

    shiny::observe({
      req(input$show_wavesel, input$sel_wavevar != 0, !is.null(raw_data()))
      vals = sort(unique(pull(raw_data(), !!sym(input$sel_wavevar))))
      shiny::updateSelectInput(session, "sel_waves", choices = vals, selected = vals)
    })

    shiny::observe({
      req(!is.null(raw_data()), is.null(resetProcess()))
      req(input$sel_indivvar != 0, input$sel_datavar != 0, input$sel_wavevar != 0)
      wavevar = sym(input$sel_wavevar)
      datavar = sym(input$sel_datavar)

      proc_data(raw_data() %>%
                { if(input$show_wavesel) dplyr::filter(., !!wavevar %in% input$sel_
                    waves) else . } %>%
                { if(input$factorize_wavevar)  dplyr::mutate(., !!wavevar := refactor
                    _wave(!!wavevar)) else . } %>%
                { if(input$factorize_datavar)  dplyr::mutate(., !!datavar := factor(!
                    !datavar)) else . }

      )
    })

    shiny::observe({
      req(!is.null(proc_data()))
      cat_vars <- select_if(proc_data(), function(x) is.logical(x) | is.factor(x))
          %>% colnames
      logi_vars <- select_if(proc_data(), function(x) is.logical(x) | (is.factor(x) &
           (length(levels(x)) == 2))) %>% colnames
      shiny::updateSelectInput(session,
                               "sel_subsetvar1",
                               choices = c("(pick)" = 0, c(cat_vars, "increase", "
                                   decrease", "unvary", "monotonic")),
                               selected = 0
      )
      shiny::updateSelectInput(session,
                               "sel_subsetvar2",
                               choices = c("(pick)" = 0, c(logi_vars, "increase", "
                                   decrease", "unvary", "monotonic")),
                               selected = 0
      )
    })

    shiny::observe({
      req(!is.null(proc_data()))
      datatypes <- proc_data() %>% dplyr::summarise_all(class)
      cur_type <- input$sel_plottype

      if ((pull(datatypes[input$sel_wavevar]) == "factor") &&
          (pull(datatypes[input$sel_datavar]) == "factor")) {
        shiny::updateSelectInput(session,
          "sel_plottype",
          choices = c("Focused Time Flow" = 1, "Time Flow" = 2,
                      "Transition Flow" = 3, "Strip Plot" = 4),
          selected = (if (cur_type %in% 1:4) cur_type else 1)
        )
      } else {
        shiny::updateSelectInput(session,
                                 "sel_plottype",
                                 choices = c("Time Plot" = 5),
                                 selected = 5)
      }


    })

    shiny::observe({
      if (input$sel_plottype %in% 1:3) {
        shinyjs::showElement(id = "ggplotOutput")
        shinyjs::hideElement(id = "plotlyOutput")
      } else {
        shinyjs::hideElement(id = "ggplotOutput")
        shinyjs::showElement(id = "plotlyOutput")
      }


      shinyjs::toggleState(input$sel_fettuccinetype, input$sel_plottype == 4)
    })

    output$ggplotPlot <- shiny::renderPlot({
      req(input$sel_plottype %in% c(1:3,5), !is.null(proc_data()))
      doAlluvialPlot(input, proc_data())
    })

    output$plotlyPlot <- plotly::renderPlotly({
      req(input$sel_plottype %in% 4:5, !is.null(proc_data()))

      if (input$sel_plottype == 5)
        req(input$sel_subsetvar1, input$sel_subsetvar2)

      if (input$sel_plottype == 4)
        doStripPlot(input, proc_data())
      else if (input$sel_plottype == 5)
        doTimePlot(input, proc_data())
    })
}

# Run the application
shinyApp(ui = ui, server = server)
```

Appendix C

Longitudinal Exploration Module Source

For reference purposes the current version of the Longitudinal Exploration Module for iNZight is provided below.

```r
#' @name Longitudinal Module
#' @author Sophie Jones
#' @desc A module to visualise longitudinal data using the longZight package.
#'
#' @import longZight
#'
#' @export LongModule
#' @exportClass LongModule
LongModule <- setRefClass(
    "Longitudinal Visualisation",
    contains = "CustomModule",
    fields = list(
        GUI = "ANY",
        activeData  = "data.frame",
        loadedData  = "ANY",
        sortedData  = "ANY",
        timeVar = "ANY",
        timeVarSelect = "ANY",
        indivVar = "ANY",
        indivVarSelect = "ANY",
        responseVar = "ANY",
        responseVarSelect = "ANY",
        completeCases = "logical",
        completeCasesSelect = "ANY",
        timeFactor = "logical",
        timeFactorSelect = "ANY",
        colourVar = "ANY",
        colourVarSelect = "ANY",
        subsetVar = "ANY",
        subsetVarSelect = "ANY",
        highlightVar = "ANY",
        highlightVarSelect = "ANY",
        sortTypeVar = "ANY",
        sortTypeVarSelect = "ANY",
        waveFocusVar = "ANY",
        waveFocusVarSelect = "ANY",
        popoutVar = "ANY",
        popoutVarSelect = "ANY",
        plotType = "ANY",
        plotTypeSelect = "ANY",
        reversePopoutVar = "logical",
        reversePopoutButton = "ANY",
        mainGrp = "ANY",
        modwin = "ANY",
        g3 = "ANY",
        g4 = "ANY",
        # The User Interface (UI) should modify these "fields",
        # which can be used by other components of the module
        # (for example, plotting)
        colour = "character"
    ),
    methods = list(
        initialize = function(gui, name) {
            callSuper(gui,
                      name = name,
                      embedded = TRUE
            )

            ## you must specify any necessary packages for the module:
            install_dependencies("longZight")#, optional = nonessential_packages)

            activeData <<- GUI$getActiveData()
            sortedData <<- NULL

            frameFont <- list(weight = "bold")

            modwin <<- GUI$initializeModuleWindow(.self,
                                                  title = "Longitudinal Visualisation
                                                      ", scroll = TRUE)
            mainGrp <<- modwin$body

            g1 <- gframe("Individual Information", pos = 0.5, horizontal = FALSE,
                         container = mainGrp)
            g2 <- gframe("Time Information", pos = 0.5, horizontal = FALSE,
                         container = mainGrp)
            g3 <<- gframe("Variable Information", pos = 0.5, horizontal = FALSE,
                          container = mainGrp)
            g4 <<- gframe("Plot Options", pos = 0.5, horizontal = FALSE,
                          container = mainGrp)
            visible(g3) <<- FALSE
            enabled(g4) <<- FALSE

            g1$set_borderwidth(8)
            g2$set_borderwidth(8)
            g3$set_borderwidth(8)
            g4$set_borderwidth(8)

            g1_layout <- glayout(container = g1)
            indivVar <<- NULL
            indivVarSelect <<- gcombobox(names(activeData), selected = 0,
                                         handler = function(h, ...) {
                                             indivVar <<- svalue(h$obj)
                                             updateLZObject()
                                             updatePlot()
                                         })

            g1_layout[1, 1:2, expand = TRUE] <- indivVarSelect

            g2_layout <- glayout(container = g2)
            timeVar <<- NULL
            timeVarSelect <<- gcombobox(names(activeData), selected = 0,
                                        handler = function(h, ...) {
                                            timeVar <<- svalue(h$obj)
                                            if (sapply(activeData, class)[timeVar] %
                                                in% c("logical", "character")) {
                                                visible(timeFactorSelect) <<- FALSE
                                                timeFactor <<- TRUE
                                            } else {
                                                visible(timeFactorSelect) <<- TRUE
                                                timeFactor <<- svalue(
                                                    timeFactorSelect)
                                            }
                                            updateLZObject()
                                            updateResponse()
                                            updatePlot()
                                        })

            timeFactor <<- FALSE
            timeFactorSelect <<- gcheckbox("Treat as Waves/Categorical", checked =
                FALSE,
                                           handler = function(h, ...) {
                                               timeFactor <<- svalue(h$obj)
                                               updateLZObject()
                                               updateResponse()
                                               updatePlot()
                                           })
            visible(timeFactorSelect) <<- FALSE

            g2_layout[1, 1:2, expand = TRUE] <- timeVarSelect
            g2_layout[2, 1:2, expand = TRUE] <- timeFactorSelect

            g3_layout <- glayout(container = g3)

            responseVar <<- 0
            responseVarSelect <<- gcombobox(names(activeData), selected = 0,
                                            handler = function(h, ...) {
                                                responseVar <<- svalue(h$obj)
                                                updateResponse()
                                                updatePlot()
                                            })

            #time_plot_group <- ggroup(horizontal = FALSE, container = g3_layout)

            colourVar <<- 0
            colourVarSelect <<- gcombobox("(none)", selected = 1,
                                          handler = function(h, ...) {
                                              sval <- svalue(h$obj)
                                              colourVar <<- ifelse(sval == "(none)",
                                                  0, sval)
                                              updatePlot()
                                          })
            enabled(colourVarSelect) <<- FALSE

            popoutVar <<- 0
            popoutVarSelect <<- gcombobox("(none)", selected = 1,
                                          handler = function(h, ...) {
                                              sval <- svalue(h$obj)
                                              popoutVar <<- ifelse(sval == "(none)",
                                                  0, sval)
                                              updatePlot()
                                          })
            enabled(popoutVarSelect) <<- FALSE

            subsetVar <<- 0
            subsetVarSelect <<- gcombobox("(none)", selected = 1,
                                          handler = function(h, ...) {
                                              sval <- svalue(h$obj)
                                              subsetVar <<- ifelse(sval == "(none)",
                                                  0, sval)
                                              updatePlot()
                                          })
            enabled(subsetVarSelect) <<- FALSE

            g3_layout[1, 1:2, expand = TRUE] <- responseVarSelect
            lbl <- glabel("Focus on: ")
            g3_layout[2, 1] <- lbl
            g3_layout[2, 2, expand = TRUE] <- popoutVarSelect
            lbl <- glabel("Subset/Colour By: ")
            g3_layout[3, 1] <- lbl
            g3_layout[3, 2, expand = TRUE] <- colourVarSelect
            lbl <- glabel("Subset By: ")
            g3_layout[4, 1] <- lbl
            g3_layout[4, 2, expand = TRUE] <- subsetVarSelect


            g4_layout <- glayout(container = g4)
            lbl <- glabel("Plot type: ")
            g4_layout[1, 1] <- lbl

            plotTypeSelect <<- gcombobox(character(0), selected = 0,
                                         handler = function(h, ...) {
                                             plotType <<- svalue(h$obj)
                                             updateSortOptions()
                                             updateSortedData()
                                             updatePlot()
                                         })
            g4_layout[1, 2, expand = TRUE] <- plotTypeSelect

            completeCases <<- TRUE
```

```
        completeCasesSelect <<- gcheckbox("Complete Cases Only", checked = TRUE,
                                          handler = function(h, ...) {
                                              completeCases <<- svalue(h$obj)
                                              updatePlot()
                                          })
        g4_layout[2, 1:2, expand = TRUE] <- completeCasesSelect

        reversePopoutVar <<- FALSE
        reversePopoutButton <<- gbutton("Swap Focus",
                                        handler = function(h, ...) {
                                            reversePopoutVar <<- !
                                                reversePopoutVar
                                            updatePlot()
                                        })
        #reversePopoutButton <- gcheckbox("Swap Focus", checked =
            reversePopoutVar,
        #                           handler = function(h, ...) {
        #                               reversePopoutVar <<- svalue(h$obj)
        #                               updatePlot()
        #                           })
        g4_layout[2, 1:2, expand = TRUE] <- completeCasesSelect
        g4_layout[3, 1:2, expand = TRUE] <- reversePopoutButton

        lbl <- glabel("Highlight Variable: ")
        g4_layout[4, 1] <- lbl
        highlightVar <<- 0
        highlightVarSelect <<- gcombobox(character(0), selected = 0,
                                         handler = function(h, ...) {
                                             highlightVar <<- svalue(h$obj)
                                             updatePlot()
                                         })

        g4_layout[4, 2, expand = TRUE] <- highlightVarSelect

        lbl <- glabel("Sort Type: ")
        g4_layout[5, 1] <- lbl
        sortTypeVar <<- 0
        sortTypeVarSelect <<- gcombobox(character(0), selected = 0,
                                        handler = function(h, ...) {
                                            sortTypeVar <<- svalue(h$obj)
                                            updateSortedData()
                                            updatePlot()
                                        })

        g4_layout[5, 2, expand = TRUE] <- sortTypeVarSelect

        lbl <- glabel("Focus Timepoint: ")
        g4_layout[6, 1] <- lbl
        waveFocusVar <<- 0
        waveFocusVarSelect <<- gcombobox(character(0), selected = 0,
                                         handler = function(h, ...) {
                                             waveFocusVar <<- svalue(h$obj)
                                             updatePlot()
                                         })

        g4_layout[6, 2, expand = TRUE] <- waveFocusVarSelect
```

```


            addHandlerSelectionChanged(indivVarSelect, function(h, ...) {
                updateLZObject()
            })

            addHandlerSelectionChanged(timeVarSelect, function(h, ...) {
                updateLZObject()
            })



            ## Footer
            btmGrp <- modwin$footer

            helpButton <- gbutton("Help",
                                  expand = TRUE,
                                  fill = TRUE,
                                  cont = btmGrp,
                                  handler = function(h, ...) {
                                      browseURL(
                                          "https://www.stat.auckland.ac.nz/~wild/
                                              iNZight/user_guides/add_ons/?topic=time
                                              _series"
                                      )
                                  }
            )
            homeButton <<- gbutton("Home",
                                   expand = TRUE,
                                   fill = TRUE,
                                   cont = btmGrp,
                                   handler = function(h, ...) {
                                       close()
                                   }
            )

            cat("Running new module\n")
        },
        updateLZObject = function() {
            if (length(indivVar) == 0 || length(timeVar) == 0 ||
                is.null(indivVar) || is.null(timeVar) ||
                (!any(indivVar %in% names(activeData))) ||
                (!any(timeVar %in% names(activeData))))
                return()
            if (timeFactor)
                loadedData <<- longZight::refactor_wave(longZight::load_long_data(
                    activeData,
                                                                                              id_
                                                                                                  col

                                                                                                  =

                                                                                                  indivVar
                                                                                                  ,

                                                                                              time
```

```
                                                                                       _
                                                                                       col
                                                                                       =
                                                                                       timeVar
                                                                                       ,
                                                                                   ts.
                                                                                       args
                                                                                       =
                                                                                       FALSE
                                                                                       )
                                                                                       )
            else
                loadedData <<- longZight::load_long_data(activeData,
                                                          id_col = indivVar,
                                                          time_col = timeVar,
                                                          ts.args = FALSE)
            visible(g3) <<- TRUE
            enabled(g4) <<- TRUE
        },
        updateResponse = function() {
            if (length(responseVar) == 0 ||
                (!any(responseVar %in% names(activeData))))
                return()
            responseType <- sapply(activeData, class)[responseVar]
            timeType <- sapply(activeData, class)[timeVar]

            blockHandlers(plotTypeSelect)
            if (responseType %in% c("factor", "character")) {
                if (responseType %in% c("factor", "character")) {
                    plotTypeSelect[] <- c("Alluvial", "Lasagna")
                    svalue(plotTypeSelect) <- "Alluvial"
                    plotType <<- "Alluvial"
                    enabled(completeCasesSelect) <<- TRUE

                    highlightVarSelect[] <- c("(none)",
                                              levels(activeData[, responseVar, drop =
                                                   TRUE]))
                    svalue(highlightVarSelect) <- "(none)"

                    enabled(reversePopoutButton) <<- FALSE
                } else {
                    plotTypeSelect[] <- c("Transition Plot")
                    svalue(plotTypeSelect) <- "Transition Plot"
                    plotType <<- "Transition Plot"
                    enabled(completeCasesSelect) <<- FALSE
                    enabled(reversePopoutButton) <<- FALSE
                }
            } else {
                plotTypeSelect[] <- c("Time Plot")
                svalue(plotTypeSelect) <- "Time Plot"
                plotType <<- "Time Plot"
                enabled(completeCasesSelect) <<- FALSE
                enabled(reversePopoutButton) <<- TRUE
            }
            updateSortOptions()
            unblockHandlers(plotTypeSelect)

            colour_names <- c("(none)", names(activeData)[
                (sapply(activeData, class) %in%
                     c("logical", "factor", "categorical") &
                     !(names(activeData) %in%
                           c(indivVar, timeVar, responseVar)))
            ])

            blockHandlers(colourVarSelect)
            blockHandlers(subsetVarSelect)
            blockHandlers(popoutVarSelect)

            colourVarSelect[] <- colour_names
            svalue(colourVarSelect) <- "(none)"

            subsetVarSelect[] <- colour_names
            svalue(subsetVarSelect) <- "(none)"

            popoutVarSelect[] <- c("(none)", names(activeData)[
                (sapply(activeData, class) %in% c("logical") &
                     !(names(activeData) %in%
                           c(indivVar, timeVar, responseVar)))
            ])
            svalue(popoutVarSelect) <- "(none)"

            unblockHandlers(colourVarSelect)
            unblockHandlers(subsetVarSelect)
            unblockHandlers(popoutVarSelect)

            updateSortedData()
        },
        updateSortedData = function() {
            if (plotType == "Alluvial") {
                plotMode <- which(sortTypeVar == sortTypeVarSelect[])
                if (plotMode == 1) {
                    sortedData <<- longZight::recode_long_alluvial(loadedData,
                        responseVar, completecases = completeCases, ts.args = FALSE)
                } else {
                    sortedData <<- longZight::recode_long_lasagna(loadedData,
                        responseVar, completecases = completeCases, ts.args = FALSE)
                }
            } else if (plotType == "Lasagna") {
                sort_types <- c("By Level", "By Count", "By Block")
                sortorder <- ifelse(
                    length(sortTypeVar) > 0 && sortTypeVar %in% sort_types,
                    which.max(sortTypeVar == sort_types)-1,
                    sort_types[1])
                warning(sortTypeVar)
                warning(sortorder)
                sortedData <<- longZight::recode_long_lasagna(loadedData, responseVar
                    , sortorder = sortorder, completecases = completeCases, ts.args =
                     FALSE)
            }
            updateSortOptions()
        },
        updateSortOptions = function() {
            if (plotType %in% c("Alluvial", "Lasagna")) {
                enabled(sortTypeVarSelect) <<- TRUE
                enabled(waveFocusVarSelect) <<- TRUE

                blockHandlers(sortTypeVarSelect)
                blockHandlers(waveFocusVarSelect)

                if (plotType == "Alluvial")
                    sort_types <- c("Colour by Origin", "Transitions", "Trace Flow")
                else
                    sort_types <- c("By Level", "By Count", "By Block")
                old_sort_type <- svalue(sortTypeVarSelect)
                sortTypeVarSelect[] <- sort_types
                svalue(sortTypeVarSelect) <- ifelse(
                    length(old_sort_type) > 0 && old_sort_type %in% sort_types,
                    old_sort_type,
                    sort_types[1])
                sortTypeVar <<- svalue(sortTypeVarSelect)

                if (!is.null(sortedData)) {
                    waves_list <- attr(sortedData,  "wave_list")
                    old_focus <- svalue(waveFocusVarSelect)
                    waveFocusVarSelect[] <- waves_list
                    svalue(waveFocusVarSelect) <- ifelse(
                        length(old_focus) > 0 && old_focus %in% waves_list,
                        old_focus,
                        waves_list[1]
                    )
                    waveFocusVar <<- svalue(waveFocusVarSelect)
                }

                unblockHandlers(sortTypeVarSelect)
                unblockHandlers(waveFocusVarSelect)
            }

            if (plotType == "Time Plot") {
                enabled(colourVarSelect) <<- TRUE
                enabled(popoutVarSelect) <<- TRUE
                enabled(subsetVarSelect) <<- TRUE
            } else {
                enabled(colourVarSelect) <<- FALSE
                enabled(popoutVarSelect) <<- FALSE
                enabled(subsetVarSelect) <<- FALSE
            }
        },
        ## add new methods to simplify your code
        updatePlot = function() {
            if (length(indivVar) == 0 || length(timeVar) == 0 ||
                length(responseVar) == 0 ||
                (!any(indivVar %in% names(activeData)) ||
                (!any(timeVar %in% names(activeData))) ||
                (!any(responseVar %in% names(activeData))) ||
                (is.null(sortedData) && !(plotType %in% c("Time Plot", "Transition
                    Plot"))))
                return()
            # This function is linked to the "refresh plot" button.
            # It should generate a plot purely from the values
            # of variables stored in "fields".
            # The UI then alters the values of the fields and
            # calls this function.
            if (inherits(loadedData, "longZight") &&
                responseVar %in% names(activeData))
                if (plotType == "Alluvial" && (inherits(sortedData, "longZightAlluv")
                     || inherits(sortedData, "longZightLasagna"))) {
                    warning(waveFocusVar)
                    longZight::iz_plot_alluvial(sortedData, responseVar,
                        completecases = completeCases,
                                                highlight = if (highlightVar == 0 ||
                                                    highlightVar == "(none)") NULL
                                                    else highlightVar,
                                                plot.mode = which(sortTypeVar ==
                                                    sortTypeVarSelect[]),
                                                focus = waveFocusVar)
                } else if (plotType == "Lasagna" && inherits(sortedData, "
                    longZightLasagna")) {
                    longZight::iz_plot_lasagna(sortedData, responseVar, completecases
                         = completeCases)
                } else if (plotType == "Time Plot") {
                    print(longZight::plot_long_line(loadedData, responseVar,
                                                    subsetvar1 = if (colourVar != 0)
                                                        colourVar else NULL,
                                                    subsetvar2 = if (popoutVar != 0)
                                                        popoutVar else NULL,
                                                    subsetvar2rev = reversePopoutVar,
                                                    facetvar = if (subsetVar != 0)
                                                        subsetVar else NULL,
                                                    ts.arg = FALSE))
                }
        },
        close = function() {
            cat("Closing module\n")

            callSuper()
        }
    ),
    ## This is currently required to get around namespace locking
    where = e
)
```

Appendix D

`longZight` Package Manual

Package ‘longZight’

November 20, 2020

Title Longitudinal Data Visualisation for iNZight

Version 0.1.0.9001

Description Provides tools for iNZight and other users to
easily visualise longtiduinal data using Tidyverse-like
syntax.

License MIT + file LICENSE

URL https://github.com/snjnz/longZight

BugReports https://github.com/snjnz/longZight/issues

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.1

Suggests testthat

Imports tibble,
rlang,
forcats,
ggalluvial,
ggplot2,
tidyr,
dplyr,
magrittr,
tidyselect,
brolgar

R topics documented:

`alluvial_fill_discrete` *Alternative fill function for Alluvial objects*

Description

Enables the use of a highlight variable in `plot_long_alluvial` to enable easier reading of a standard Alluvial plot.

Usage

```
alluvial_fill_discrete(
  ...,
  h = c(0, 360) + 15,
  c = 100,
  l = 65,
  h.start = 0,
  direction = 1,
  na.value = "grey50",
  aesthetics = "fill",
  focus = 1
)
```

Arguments

`...`	Standard options to be passed on depending on scale type and
`focus`	The index of the factor that should be made lighter

Details

Modifies output of the standard `discrete_scale` palette so levels that are not defined by `focus` are made comparatively lighter.

`getDatatypes` *Internal function to get data types in a data table*

Description

Internal function to get data types in a data table

Usage

```
getDatatypes(.data)
```

Arguments

`.data`	A longZight object generated by `load_long_data`

Examples

```
## Not run:
getDataTypes(dataobject)

## End(Not run)
```

`iz_plot_alluvial` *iNZight Helper Function to generate Alluvial plots*

Description

Will, (if required) recode the data to Alluvial or Lasagna format using `recode_long_alluvial` or `recode_long_lasagna` as required, and provide a complete ready-to-output plot for iNZight or other purposes.

Usage

```
iz_plot_alluvial(...)

## S3 method for class 'longZightAlluv'
iz_plot_alluvial(
  .data,
  response,
  xlab = "Wave",
  title = paste0("Plot of ", response),
  completecases = FALSE,
  highlight = NULL,
  plot.mode = 1,
  focus = NULL,
  ...
)

## S3 method for class 'longZightLasagna'
iz_plot_alluvial(
  .data,
  response,
  xlab = "Wave",
  title = paste0("Plot of ", response),
  completecases = FALSE,
  highlight = NULL,
  plot.mode = 1,
  ...
)

## S3 method for class 'longZight'
iz_plot_alluvial(
  .data,
    response,
    xlab = "Wave",
    title = paste0("Plot of ", response),
    completecases = FALSE,
    highlight = NULL,
    plot.mode = 1,
    focus = NULL,
    ...
  )
```

Arguments

`.data`	A longZight object generated by `load_long_data`
`response`	The response variable to be plotted as an Alluvial plot
`xlab`	Optional x-axis label
`title`	Optional title
`completecases`	Logical variable to exclude incomplete cases
`highlight`	Response level to be highlighted
`plot.mode`	Integer between 1 and 3. 1 for "Colour by Origin" mode, 2 for "Transitional" mode and 3 for "Trace Flow" mode.
`focus`	Time period to use as focus/origin point (only for `plot.mode = 1` or if `.data` inherits the `longZightAlluv` class)

Value

`ggplot2` plot with the data represented as an Alluvial diagram

Examples

```
## Not run:
mydata %>%
  load_long_data(IDind, WAVE) %>%
  refactor_wave %>%
  iz_plot_alluvial(A5E, completecases = TRUE)

## End(Not run)
```

`iz_plot_lasagna` *iNZight Helper Function to generate Lasagna plots*

Description

Will, (if required) recode the data to Lasagna format using `recode_long_lasagna` and provide a complete ready-to-output plot for iNZight or other purposes.

Usage

```
iz_plot_lasagna(...)

## S3 method for class 'longZightLasagna'
iz_plot_lasagna(
  .data,
  response,
  xlab = "Wave",
  title = paste0("Plot of ", response),
  completecases = FALSE,
  ...
)

## S3 method for class 'longZight'
iz_plot_lasagna(
  .data,
  response,
  sortorder = 1,
  xlab = "Wave",
  title = paste0("Plot of ", response),
  completecases = FALSE,
  ...
)
```

Arguments

`.data`	A longZight object generated by `load_long_data`
`response`	The response variable to be plotted as a Lasagna plot
`xlab`	Optional x-axis label
`title`	Optional title
`completecases`	Logical variable to exclude incomplete cases
`sortorder`	(if `.data` is a `longZight` object) the sort order for `recode_long_lasagna`

Examples

```
## Not run:
mydata %>%
  load_long_data(IDind, WAVE) %>%
  refactor_wave %>%
  iz_plot_lasagna(A5E, completecases = TRUE)

## End(Not run)
```

`load_long_data` *Load data into longZight form*

Description

Load data into longZight form

Usage

```
load_long_data(.data, id_col = "id", time_col = "time", ts.args = TRUE)
```

Arguments

`.data`	A `tbl_df` or `data.frame` object storing the data
`id_col`	Name, or `sym` object for the individual identifier column
`time_col`	Name, or `sym` object for the time/wave column

Value

`longZight` object with individual identifer and time information encoded

Examples

```
## Not run:
mydata %>%
  load_long_data(IDind, WAVE)

## End(Not run)
```

`plot_long_alluvial` *Plotting function to generate Alluvial plots*

Description

Generates a `ggplot2` plot of the response variable encoded in the `longZightAlluv` or `longZightLasagna` object, plots are returned allowing further customisation using standard ggplot2 geometries.

Usage

```
plot_long_alluvial(.data, ...)

## S3 method for class 'longZightAlluv'
plot_long_alluvial(
  .data,
  fillfunc = ggplot2::scale_fill_discrete(),
  highlight = NULL,
  focus = NULL
)

## S3 method for class 'longZightLasagna'
plot_long_alluvial(
  .data,
  alluvium = FALSE,
  fillfunc = ggplot2::scale_fill_discrete(),
  highlight = NULL,
  ...
)
```

Arguments

`.data`	A `longZightAlluv` or `longZightLasagna` object generated by `recode_long_alluvial` or `recode_long_lasagna` respectively
`fillfunc`	The fill function to use for colouring Alluvial flows, defaults to `ggplot2::scale_fill_discrete()`, but `alluvial_fill_discrete` must be used if `highlight` is set.
`highlight`	Response level to be highlighted
`focus`	Time period to use as focus/origin point (only if `.data` inherits the `longZightAlluv` class)
`alluvium`	Logical, if `.data` inherits the `longZightLasagna` class, whether to plot transitions (`FALSE`) or trace flows (`TRUE`)

Value

`ggplot2` plot with the data represented as an Alluvial diagram

Examples

```
## Not run:
mydata %>%
  load_long_data(IDind, WAVE) %>%
  refactor_wave %>%
  recode_long_alluvial(A5E) %>%
  plot_long_alluvial

## End(Not run)
```

`plot_long_lasagna` *Plotting function to generate Lasagna plots*

Description

Generates a `ggplot2` plot of the response variable encoded in the `longZightLasagna` object, plots are returned allowing further customisation using standard ggplot2 geometries.

Usage

```
plot_long_lasagna(.data, ...)

## S3 method for class 'longZightLasagna'
plot_long_lasagna(.data, ...)
```

Arguments

`.data`	A `longZightLasagna` object generated by `recode_long_lasagna`

Value

`ggplot2` plot with the data represented as a Lasagna diagram

Examples

```
## Not run:
mydata %>%
  load_long_data(IDind, WAVE) %>%
  refactor_wave %>%
  recode_long_lasagna(A5E) %>%
  plot_long_lasagna

## End(Not run)
```

---

plot_long_line *Plotting function for Spaghetti and Transitional plots*

Description

Plotting function for Spaghetti and Transitional plots

Plotting function for Spaghetti and Transitional plots for `longZight` objects

Usage

```
plot_long_line(
  .data,
  response,
  subsetvar1 = NULL,
  subsetvar2 = NULL,
  subsetvar2rev = FALSE,
  facetvar = NULL,
  ts.args = TRUE
)

## S3 method for class 'longZight'
plot_long_line(
  .data,
  response,
  subsetvar1 = NULL,
  subsetvar2 = NULL,
  subsetvar2rev = FALSE,
  facetvar = NULL,
  ts.args = TRUE
)
```

Arguments

`.data`	A longZight object generated by `load_long_data`
`response`	The response variable to be plotted by an alluvial
`subsetvar1`	Name of variable to be used for colouring Spaghetti lines
`subsetvar2`	Name of variable to be used to bring forward `TRUE` values and grey out `FALSE` variables
`subsetvar2rev`	Logical, reverse logic/processing of `subsetvar2`

`facetvar`	Name of variable to be used for faceting
`ts.args`	Logical control to switch pharsing method for variable options `subsetvar1`, `subsetvar2` and `facetvar` default to `NULL`

Examples

```
## Not run:
mydata %>%
  load_long_data(id, xp) %>%
  plot_long_line(ln_wages, ged, hispanic)

## End(Not run)
```

`plot_types` *Helper function to return matrix of appropriate plot types for data frames.*

Description

Helper function to return matrix of appropriate plot types for data frames.

Usage

```
plot_types(.data)

## S3 method for class 'longZight'
plot_types(.data)
```

Arguments

`.data`	A longZight object generated by `load_long_data`

Examples

```
## Not run:
mydata %>%
  load_long_data(IDind, WAVE) %>%
  plot_types

## End(Not run)
```

recode_long_alluvial — *Function to convert* longZight *objects for an Alluvial plot*

Description

Function to convert longZight objects for an Alluvial plot

Usage

```
recode_long_alluvial(.data, response, completecases, ts.args)

## S3 method for class 'longZight'
recode_long_alluvial(.data, response, completecases = FALSE, ts.args = TRUE)
```

Arguments

.data	A longZight object generated by load_long_data
response	The response variable to be plotted as an alluvial
completecases	Logical variable to exclude incomplete cases
ts.args	Logical, control to switch pharsing method for variable options

Value

A longZightAlluv object

Examples

```
## Not run:
mydata %>%
  load_long_data(IDind, WAVE) %>%
  refactor_wave %>%
  recode_long_alluvial(A5E)

## End(Not run)
```

recode_long_lasagna — *Function to convert* longZight *objects to a format suitable for Lasagna plots*

Description

Note: the data structure created by recode_long_lasagna can also be used by plot_long_alluvial for alternative display methods.

Usage

```
recode_long_lasagna(.data, response, sortorder, completecases, ts.args)

## S3 method for class 'longZight'
recode_long_lasagna(
  .data,
  response,
  sortorder = 0,
  completecases = FALSE,
  ts.args = TRUE
)
```

Arguments

`.data`	A `longZight` object generated by `load_long_data`
`response`	The response variable to be plotted as an alluvial
`sortorder`	Integer between 1 and 3. 1 for "By (factor) Level" mode, 2 for "By Count" mode and 3 for "By (contiguous) Block" mode.
`completecases`	Logical variable to exclude incomplete cases
`ts.args`	Logical, control to switch pharsing method for variable options

Value

A `longZightLasagna` object

Examples

```
## Not run:
mydata %>%
  load_long_data(IDind, WAVE) %>%
  refactor_wave %>%
  recode_long_lasagna(A5E, sortorder = 2, completecases = TRUE)

## End(Not run)
```

refactor_wave *Helper function to convert a wave variable into an ordered factor*

Description

Helper function to convert a wave variable into an ordered factor

Helper function to convert a wave variable into an ordered factor for longZight objects.

Usage

```
refactor_wave(.data, subset = NULL)

## S3 method for class 'longZight'
refactor_wave(.data, subset = NULL)
```

Arguments

`.data`	A longZight object generated by `load_long_data`
`subset`	An optional vector of values to retain

Value

An `longZight` object with a modified wave variable.

Examples

```
## Not run:
mydata %>%
  load_long_data(IDind, WAVE) %>%
  refactor_wave(c(2008, 2010, 2012))

## End(Not run)
```

`ymin_calc` *Internal function to calculate y-values for Lasagna plots*

Description

Internal function to calculate y-values for Lasagna plots

Usage

```
ymin_calc(n)
```

Arguments

`n`	vector of counts.

Index