Visualisation of Multidimensional Physics Data

A thesis submitted for the degree of

Bachelor of Commerce (Honours)

by

Michael Kipp

B.Ec., Monash University

Department of Econometrics and Business Statistics

Monash University

Australia

October 2017

Contents

Declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma in any university or equivalent institution, and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Michael Kipp

Motivation

Particle physics involves multivariate model specification. Different theorists propose different explanatory models, and it is important to provide tools to compare the models. For the most part, this is done by plotting pairs of variables. However, the differences between models may involve more than two variables. The goal of this research is to provide a new tool to help explore differences between multivariate data using linear combinations of more than two variables.

Figure 1: *All pairs of variables for showing differences between three groups.*

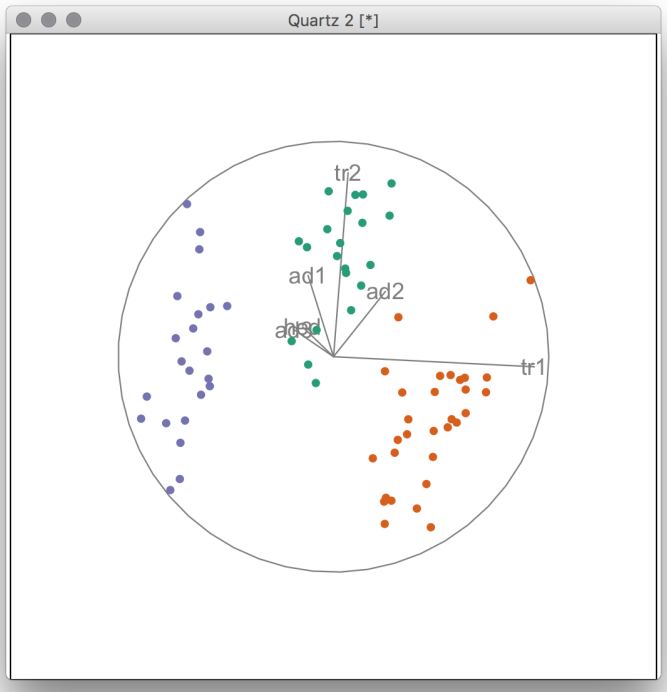

Figure 2: *Linear combination of all variables revealing a bigger difference between groups than can be seen in pairs of variables alone.*

Chapter 1

Literature Review

1.1 Terminology and introduction

Suppose we have some model $f : \mathbf{R}^d \to Y$ of particle physics data, trained on some d-dimensional dataset $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_d)$. We partition Y into subsets Y_i called *groups*, and then examine the pre-images $f^{-1}(Y)$ to try to identify differences. These Y_i could be outcomes of an experiment, or percentile ranges in a histogram. For example, we could split Y into the upper 90% and lower 10% of some $f : \mathbf{R}^d \to \mathbf{R}$, and see how the shape of the data that gives rise to each of those changes.

When $d = 2$, for example in the case of temperature measurements on the surface of the earth, we can plot the datapoints as a scatterplot (in this case, on a map) directly and immediately see where the differences are. When $d = 3$, we can project the data onto two dimensions via a 3D scatterplot with a little imagination, or by adding additional information to a two-dimensional graphic; but for higher dimensions, many more projections may be needed in order to expose meaningful shape information and allow the viewer of the data to discern differences between the groups.

The tour algorithm, due to (Cook et al., 1995), is a way of systematically generating and displaying such projections on a visual display in order to allow the user to tell apart groups of data based on differences in the shape of their projections. It can do this either

randomly, or by picking projections judged interesting according to some criterion or index function.

Current supervised classification methods, such as Linear Discriminant Analysis, Principal Component Analysis, Support Vector Machines and others, can be used to discriminate between groups with different centers, and can be adapted for use with the tour; but they are not good at discriminating between groups of similar centres, regardless of shape differences. Moreover, the current implementation of the tour algorithm is outdated, requiring multiple dependencies on the viewer's computer, including Java, and needs to be updated.

The aim of this work is twofold: to develop an index function for use with the tour algorithm to better discriminate between groups of different shapes, and to write a new graphical user interface for the tour algorithm which implements this function.

1.2 Tour

A complete description of projection interpolation and the tour algorithms is described in detail in Cook et al. (2007). What follows is a brief summary of that description, along with some discussion of projection pursuit and applications to the current problem. \

The tour is a way to visualise multi-dimensional data by projecting it to two dimensionals for visual display on a computer display. In the same way that a three-dimensional object projects different shadows when viewed from different angles, a multi-dimensional data can be projected onto two dimensions. A *tour* is a collection of different such projections strung together and interpolated to display like a movie, giving the user a "tour" of the data from different directions.

When selecting projections to view, various methods can be used. The *grand tour* picks and interpolates between projections at random, while the *guided tour* uses a criterion function, called a *projection pursuit* index, to pick new target planes. If left to run long enough, the grand tour will show all possible projections of the data, while the guided tour will try to show only "interesting" projections, as judged by the projection pursuit index.

1.2.1 Projections, planes and frames

A *projection* from p- to 2-dimensional Euclidean space is the result of pre-multiplying a $p \times 2$ orthonormal matrix $\mathbf{A}$ by an $n \times p$ data matrix, $\mathbf{X}$. For our purposes, the data matrix $\mathbf{X}$ is constant, so we also simply refer to the matrix $\mathbf{A}$ as the projection, or *frame*. The resulting projected data matrix $\mathbf{XA}$ has dimension $n \times 2$ and this is rendered as a 2-dimensional scatterplot of n points.

The orthonormal matrix, or frame, $\mathbf{A}$, describes a 2-dimensional plane in p- space. There are infinitely many such frames which describe the same plane – for example, rotating a frame $\mathbf{A}$ within the plane represents a new . The tour shows different projections of our data $\mathbf{X}$ by *rotating* these planes $\mathbf{A}$, calculating new projections of interest $\mathbf{XA}$, and interpolating between them. Note: we should avoid calculating rotations which lie in the plane of view, as this would result simply in a 2D rotation of the scatterplot and serve no other purpose than to distract the viewer.

1.2.2 Interpolating between planes

The core of the tour algorithm, as originally implemented in the visualisation software (Swayne et al., 2011) and given in (Cook et al., 2007), is the following:

1. Given a starting $p \times d$ projection $\mathbf{A}_a$ describging the starting plane, create a new target projection $\mathbf{A}_z$, describing the target plane. Recall these projections may also be called orthonormal frames. To find the optimal rotation of the starting plane into the target plane, we need to find the frames in each plane which are the closest.
2. Determine the shortest path between the frames using singular value decomposition. $\mathbf{A}_a'\mathbf{A}_z = \mathbf{V}_a\Lambda\mathbf{A}_z'$, $\Lambda = \text{diag}(\lambda_1 \geq \ldots \geq \lambda d)$, and the principal directions in each plane are $\mathbf{B}_a = \mathbf{A}_a$, $\mathbf{B}_z = \mathbf{A}_z\mathbf{V}_z$. the principal directions are the frames describing the startnig and target planes which have the shortest distance between them. The rotation si definsed with respect to these principal directions. Ths singular values, $\lambda_i, i = 1, \ldots, d$, define the smallest angles between the principal directions.
3. Orthonormalize $\mathbf{B}_z$ on $\mathbf{B}_a$, giving $\mathbf{B}_*$ to create a rotation framework.
4. Calculate the principal angles, $\tau_i = \cos^{-1}\lambda_i, i = 1, \ldots, d$.

5. Rotate the frames by dividins the angles into increments, $\tau_i(t)$, for $t \in (0, 1]$ and create the i^{th} column of the new frame, $\mathbf{b}_i$, from the i^{th} columns of $\mathbf{B}_a$ and $\mathbf{B}_*$, by $\mathbf{b}_i(t) = \cos(\tau_i(t))\mathbf{b}_{ai} + \sin(\tau_i(t))\mathbf{b}_{*i}$. When $t = 1$, the frame will be $\mathbf{B}_z$.
6. Project the data into $\mathbf{A}(t) = \mathbf{B}(t)\mathbf{V}_a$.
7. Continue the rotation until $t = 1$. Set the current projection to be $\mathbf{A}_a$ and go back to step 1.

This algorithm is used in calculating all different types of tour path. The grand toyr picks the target frames $\mathbf{A}_z$ at random, while the guided tour picks them based on a so-called *projection pursuit index function*.

1.3 Projection pursuit

Projection pursuit is a statistical technique which is used to generate "interesting" low-dimensional – in our case, 2-dimensional – projections of a high-dimensional point cloud by numerically maximising a certain objective function, which we call a *projection index*. It was first implemented by Friedman and Tukey (1974). Projection pursuit can be thought of as a method to increase the likelihood of finding interesting projections.

Formally, projection optimises a criterion or index function $f(\mathbf{X}A)$ over the space of all projections A. However, the purpose of this optimisation is not simply to find the global maximum, since there may be local maxima which expose interesting features of the point cloud, which is the purpose of the visualisation. Hence there must be some method which allows the exposure of both local and global maxima.

In the Guided tour, projection pursuit is implemented using a modified simulated annealing method due to Lee et al. (2005). They use a modified simulated annealing method using two different temperatures. One is used for neighbourhood definition, and the other allows the algoirthm to visit a local minimum and then jump out and explore for other minima. The temperature for neighbourhood definition is re-scaled by a cooling patameter, determining how many iterations are needed to converge and whether the maximum is likely to be a local maximum or global maximum.:

1. From the current projection $\mathbf{A}_a$, calculate the initial projection pursuit index value, I_0.
2. For $i = 1, 2, \ldots$:
   a) Generate new projections, $\mathbf{A}_i = \mathbf{A}_a + c^i\mathbf{B}$, from a neighbourhood of the current projection where the size of the neighbourhood is specified by the cooling parameter, c and a random projection, $\mathbf{B}$.
   b) Calculate the index value for the new projection, I_i, the difference, $\Delta I_i = I_i - I_0$ and $T_i = \frac{T_0}{\log(i+1)}$, where T_0 is an initial temperature
   c) Take the new projection $\mathbf{A}_i$ to be the target, A_z with probability $\rho = \min\left(\exp\left(\frac{\Delta I_i}{T_i}\right), 1\right)$ and interpolate from the current projection to the target projection $\mathbf{A}_z$ using the interpolation method described previously
   d) Set $\mathbf{A}_a = \mathbf{A}_i$
3. Repeat a) – d) until ΔI_i is small.

Any function f which characterises somethign interesting about the projected data can be used in this procedure. As part of this work, a new projection pursuit index was implemented using Scagnostics.

1.4 Scagnostics

Scagnostics is a neologism for the term *scatterplot diagnostics*, first introduced by John and Paul Tukey during the mid-1980s. It was developed as a method to characterise 2D distributions of orthogonal projections of sets of points in multi-dimensional space. Wilkinson and Willis (2008) created a set of nine scagnostics measures – Outlying, Skewed, Clumpy, Sparse, Striated, Convex, Skinny, Stringy, and Monotonic – which give a number between zero and one to each scatter plot.

The idea was that, while scatterplot matrices of all variables in a dataset can become overwhelming and difficult to interpret, a scatterplot matrix of the *scagnostics* can provide information on differences between groups, exceptions and patterns in a much more concise view.

In the paper by Wilkinson and Willis (2008), the calculation of the 9 scagnostics measures is carried out by creating a Euclidean graph, removing outliers to render the computations more robust, and calculating the scagnostics using various features of those graphs. For example, the Outlying measure is calculated by dividing the length of the graph before removing the outliers by its length afterwards.

A C++ version of Leland Wilkson's original Java code (Wilkinson and Willis, 2008) has been implemented by Hadley Wickham as an R package which calculates all nine scagnostics without

Chapter 2

Software base

The software used to implement the new tour interface was built using the programming languages R and JavaScript (or JS). The computation engine and web framework is written in R using the `Shiny` package, while the visualisation rendering is handled in JS.

2.1 Shiny apps

Shiny is developed by the team behind RStudio, a popular integrated development environment. The description at the Shiny homepage (https://shiny.rstudio.com/) reads: "Shiny is an R package that makes it easy to build interactive web apps straight from R. You can host standalone apps on a webpage or enable them in R Markdown documents or build dashboards. You can also extend your Shiny apps with CSS themes, htmlwidgets, and JavaScript actions."

For our purposes, the features that made Shiny an attractive platform on which to build the new tour interface were:

- an extensive catologue of UI widgets to use, with simple R function calls
- the JS frontend, which means anyone with a web browser can access the app
- a large user base and extensive documentation on the website, including premade examples
- the backend is based on R, which is an industry standard.

2.2 Tourr

The `tourr` package (Wickham et al., 2011) is an R implementation of the tour algorithms discussed in @ref{litreview}. It includes methods for geodesic interpolation and basis generation, as well as an implementation of the simulated annealing algorithm to optimise projection pursuit indices for the guided tour.

Each tour is initialised with the `new_tour()` method, which instantiates a `tour` object and takes as arguments the data **X**, the tour method including projection pursuit index if applicable, and the starting basis (if any). Once initialised, a new target plane is chosen. When a floating point argument between 0 and 1 is given to the `tour` object, a projection is returned using geodesic interpolation. For example, a value of 0.5 would be "half-way" between the two target planes, and a value of 1 would return the next target plane without any interpolation.

This series of calls to the `tour` object produces a series of projections. The value given can be understood as the ratio ω/f, where ω denotes the angular velocity of the geodesic interpolation, and f is a parameter denoting the rendering speed of the device in use, per unit time. f is a function of the particular device and can be thought of as the `frames per second`, while ω affects the speed at which the tour moves through the projection space. For our purposes, f, or `fps` in the code, is set at 25, while the ω can be manipulated by the user.

The projections can either be saved as a series of matrices **XA** or delivered to a rendering device immediately for display to the user as a scatter plot. Our implementation of the tour does not save any projections but displays them immediately, using D3.js.

2.3 D3.js

D3.js is a JavaScript library for manipulating documents based on data. D3 stands for Data-Driven Documents and its home on the web is at `https://d3js.org/`. D3 uses HTML, JS, Scalable Vector Graphics (SVG) and Cascading Style Sheets (CSS) technology to render interactive data visualisations for display on a webpage.

The advantages of D3 are similar to those provided by Shiny: namely, an industry standard with rich array of powerful, easy to use methods and widgets that can be displayed on a wide variety of devices, with a large user base. D3 works on data objects in the JavaScript Object Notation (JSON) format, which are then parsed and used to display customisable data visualisations.

The flexibility of D3 means that there are practically limitless possibilities for visualisations, using the various graphical objects and modifiers available. Using JavaScript, visualisations are rendered in D3 and can be updated in real time based on input from the user.

The new implementation of the tour uses D3 to render each projection step returned by R, calculated with the tourr (Wickham et al. (2011)) package, and update them in real-time as the tour traverses the projection space. It does this by drawing and re-drawing a scatterplot with dots (or `circles` in D3 language) and providing SVG objects for the web browser to render.

Chapter 3

Tour GUI

The new GUI for the tour was developed using the programming languages R and JavaScript, leveraging Shiny for the user interface and D3 for visualising the data. The tour paths are calculated using R with the `tourr` package, which are then sent to D3 via the Shiny webserver, which also handles and renders the UI widgets.

The new user interface allows the user to control the tour type, speed, density display toggles and projection index for the guided tour. It currently allows selection between a number of example datasets and in future will allow direct selection and upload of a new dataset using Shiny's `fileInput()` UI widget.

All of the code for the new UI is included in Appendix A. The most up-to-date version of the code can be found under the title `TourrGuiD3` at Github, at https://github.com/makipp/TourrGuiD3.

An example of the app in action is shown in a short video I recorded and uploaded to YouTube, at https://www.youtube.com/watch?v=cD6qkYQMwFU.

3.1 User interface

Shown in Figure 3.1 is a screenshot of the current version of the TourrGuiD3 GUI. It features:

1. Tour type selection

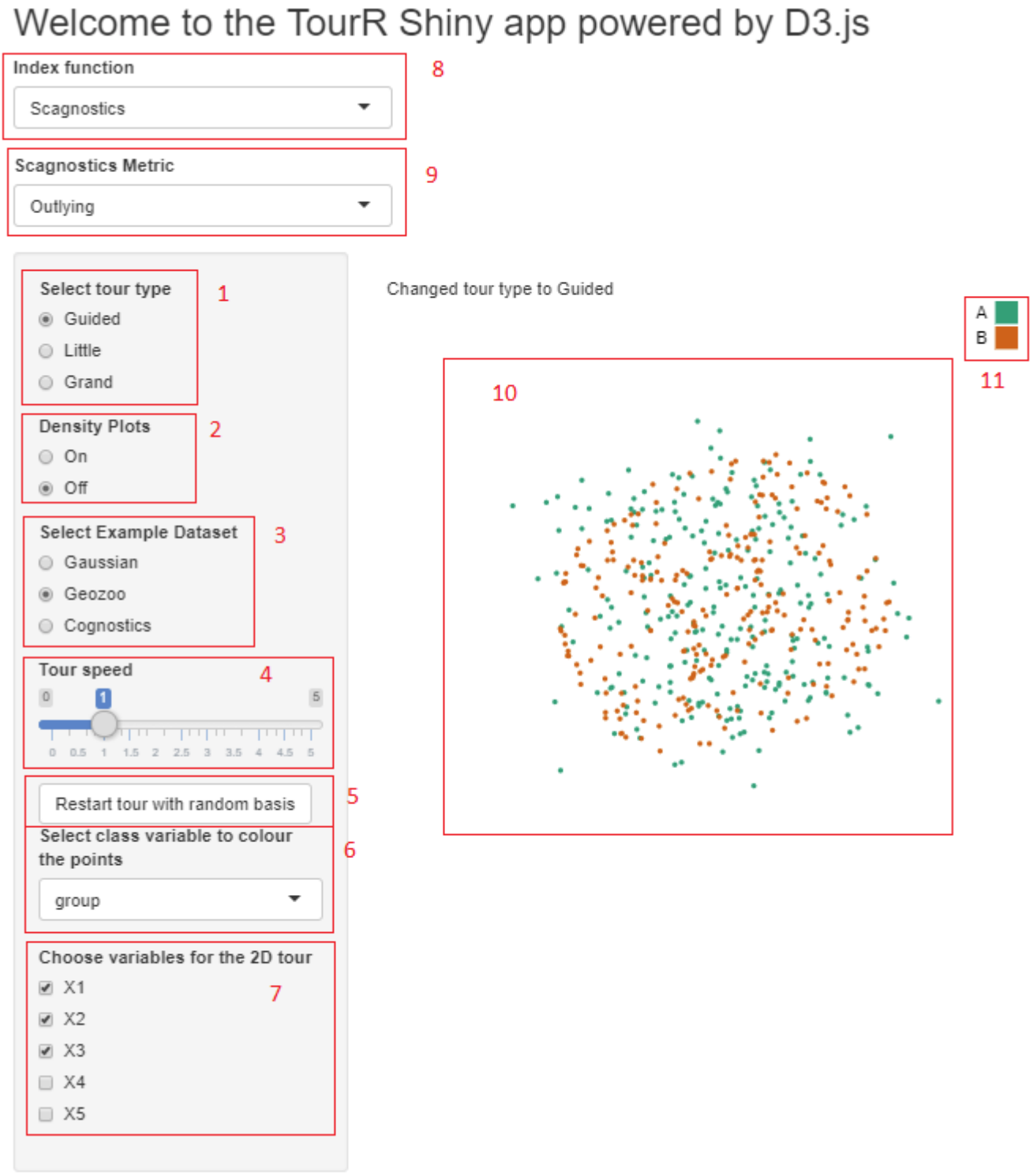

Figure 3.1: *User interface, accessed via Chrome. The UI widgets controlling the display are on the left, with animated scatterplot and legend on the right.*

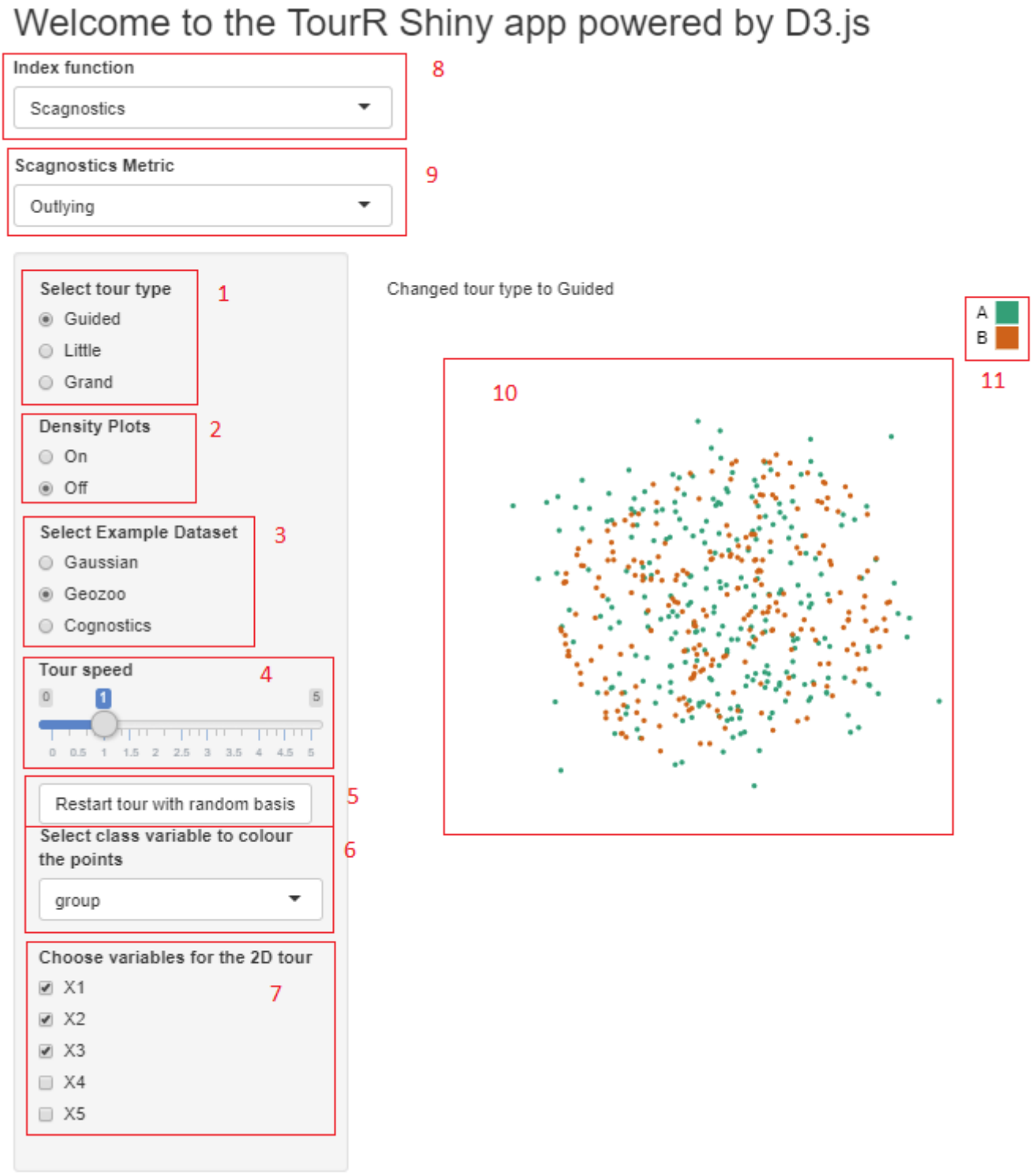
Welcome to the TourR Shiny app powered by D3.js
Index function
Scagnostics
8
Scagnostics Metric
Outlying
9
Select tour type
Guided
Little
Grand
1
Density Plots
On
Off
2
Select Example Dataset
Gaussian
Geozoo
Cognostics
3
Tour speed
4
Restart tour with random basis
5
Select class variable to colour the points
group
6
Choose variables for the 2D tour
X1
X2
X3
X4
X5
7
Changed tour type to Guided
A
B
11
10

Figure 3.2

2. Density display toggle
3. Selection of example dataset
4. Tour speed (ω)
5. Option to restart tour from a new starting basis – particularly useful for the guided tours
6. Selection of a class variable used to define the two (or more) groups for discrimination
7. Variable selection
8. Index function selection (only visible with Guided tour selected)
9. Scagnostics metric selection (only visible with Scatnostics index function selected)
10. Animated scatterplot (and density display, if toggled)
11. Colour legend. The scatterplot will be coloured according to the class variable selected in (5).

The User Interface widgets are displayed on the webpage using JavaScript, but are included as Shiny objects when creating the app. The user interface widgets are stored in the shiny app under in the file `ui.R`. For example, the following code snippet is provided as an argument to the `sidebarPanel()` method in Shiny to render the tour type selector (1):

```
radioButtons(
        "type",
        label = "Select tour type",
        choices = c("Guided", "Little", "Grand"),
        selected = "Grand"
      )
```

The other important function served by the `ui.R` file is to define the structure of the webpage itself, including HTML tags which can be referenced by D3 later, for example:

```
mainPanel(
      tags$div(tags$p(" "),
               ggvisOutput("ggvis")),
```

```
      tags$div(tags$p(textOutput("type"))),
      tags$script(src = "https://d3js.org/d3.v4.min.js"),
      tags$script(src = "https://d3js.org/d3-contour.v1.min.js"),
      tags$script(src = "https://d3js.org/d3-scale-chromatic.v1.min.js"),
      tags$div(id = "d3_output"),
      tags$div(id = "d3_output_2"),
      tags$div(id = "info"),
      tags$div(id = "info_2"),
      tags$script(src = "d3anim.js"))
  )
```

The `script` tags load the required JS libraries to run D3, while the various `div` tags define areas of the webpage required for Shiny and D3 to define and draw the output. The JS code written as part of the app is located in the `d3anim.js` file, seen in the last script tag loaded.

3.2 Server functions

The back-end of the app is defined in the `server.R` file. It is here which the R functions to manipulate the data and calculate the tour path are shown, as well as input handlers for `input` parameters passed through from the `ui.R` code via the `session` object.

The `observeEvent` Shiny method defines a code block to be run whenever some input value changes. The following code snippet restarts a tour using a random basis:

```
observeEvent(input$restart_random,
             {

               p <- length(input$variables)
               b <- matrix(runif(2*p), p, 2)

               rv$tour <- new_tour(as.matrix(rv$d[input$variables]),
                                   choose_tour(input$type, input$guidedIndex,
```

```
                                    c(rv$class[[1]]), input$scagType), b)
                })
```

the `input` object contains all of the input variables set using the UI widgets in the `ui.R` file. This `observeEvent` environment is run whenever the `input$restart_random` button is clicked. It generates a new random basis from a $N(0, 1)$ distribution, and starts a new tour via changing the `rv$tour` variable. Other input variables can be seen in definint the new tour – the matrix, tour type, and class variables.

Another `observeEvent()` environment (line 57 in the `server.R` code) handles the rest of the UI values, and then a different Shiny environment calculates tour projections and sends them to D3.

3.3 Getting projections

The projections are calculated using the tour object in an `observe()` environment, which repeatedly runs the code until a reactive variable is changed, at which point it is is "invalidated" and then re-started. The projections are calculated using the following code block:

```
observe({

    tour <- rv$tour
    aps <- rv$aps

    step <- tour(aps / fps)

    if (!is.null(step)) {
      invalidateLater(1000 / fps)

      j <- center(rv$mat %*% step$proj)
      j <- cbind(j, class = rv$class)
      colnames(j) <- NULL
```

```
      session$sendCustomMessage(type = "data", message = toJSON(j))
    }


      else{


      if (length(rv$mat[1, ]) < 3) {
        session$sendCustomMessage(type = "debug", message =
                                    "Error: Need >2 variables.")
      } else {
        session$sendCustomMessage(type = "debug", message = "Guided tour
                                    finished: no better bases found.")
      }
    }
  })
```

The `step` assignment at the start of the code block requests a new interpolation step from the `tour` path. If it receives a result, it then creates a projection matrix `j`, centers it, adds the class column for colouring, and then sends it to D3 using the `session$sendCustomMessage()` function.

The if-else logic is used to handle error cases: There will be no projection returned in the case that the guided tour has finished (ie. has optimised the index function), or there are insufficient variables to calculate a path.

3.4 R-D3 interface

There two functions provided by the Shiny framework to transport data between R and javascript: `session$sendCustomMessage()` in R, and the corresponding `Shiny.addCustomMessageHandler()` in Javascript. Whenever the former is executed in R, the latter function will execute a code block in JS. There are many examples of such functions being used to pass arbitrary data from an R app to a JS front-end, few examples exist of this basic functionality to update a D3 animation in real-time.

The data format expected by D3 is in JSON format, which combines two basic programming paradigms: a collection of name/value pairs, and an ordered list of values. R's preferred data formats include data frames, vectors and matrices. Every time a new projection has been calculated with the tour path, the resulting matrix needs to be converted to JSON and sent to D3. The code to send the D3 data looks like this:

```
session$sendCustomMessage(type = "data", message = toJSON(j))
```

This code is from the `observe` environment from the `server.R` file. It converts the projection matrix to JSON format, and sends it to JavaScript with the id `data`. When parsed in D3 by its `data()` method, it is converted back into a logical 2D array where the columns are queried first, then the rows. If column names are included in the JSON, the column indices are strings; otherwise they are integers starting from 1.

All of the code required to render the scatterplots and legends, along with colours, is JavaScript code in the file `d3anim.js`. In particular, the data from R is handled with the following code:

```
Shiny.addCustomMessageHandler("data",
    function(message) {

        /* D3 scatterplot is drawn and re-drawn using the
            data sent from the server. */

        }
```

Every time the message is sent (25 times per second), the code-block is run. The full code is omitted here for clarity as it is over seventy lines in length, but is included in appendix A and at GitHub.

3.5 Difficulties

In developing the new solution, I needed to solve problems relating to data transport, animation and user interactivity. The basic code snippets above solved the basic problem

of the data transport, but how to handle the data, render it properly and expose it to user interactivity were also challenges that needed to be overcome.

3.5.1 Animation

Drawing static scatterplots in D3 is simple and well-understood in the D3 community. One needs to define an SVG element, then axis and circle elements (the dots on the scatterplot), add them to the SVG element using the `append()` method, and render the SVG element on the page using the `selectAll()` method.

To update the scatterplot with new values, the circles are removed using the `removeAll()` method and then can be redrawn. If the new values are close enough to the old ones and this redrawing process is done enough times per second, the dots will appear to move across the screen. Alternatively, the `.transition()` method is available, which calculates an interpolation between each step, and the data supplied can be further apart.

The next question to be solved was how to achieve the synchronization of the sending and receipt of the data in such a way that the animation would appear smooth, but remained a faithful reproduction of the tour path as calculated in R. Either the interpolation could be done with the Tour algorithm within R and then sent to JS enough times per second as to make the dots appear animated - for example, 33 times per second, as ultimately chosen or - or the data could be sent less frequently, and animated in D3 using `transition()`.

The tour algorithm interpolates between target planes chosen by the specific tour type along a geodesic tour path. D3's default interpolation is linear, which was not inappropriate for our purpose. It is possible to define a custom interpolation function using D3, using supplementary data about the tour path sent alongside each projection. This has the advantage of reducing the amount of data required to be sent to D3, but the disadvantage of requiring extra logic on the JS side to ensure that the animation is synchronized properly with R.

Ultimately, the geodesic interpolation was done entirely within R and then the draw-redraw method was used within D3 to display the animation. This was because the extra time required to calculate the interpolation caused jerkiness in the animation. The

time for each transition can be set in the argument to the `transition()` function in D3, and the number of times per second data is to be calculated using the tour path can be controlled with an argument to the R `tour()` function; but when matching the two arguments together to provide a target number of frames per second, some frames were dropped and pauses were introduced due to the processing time required in Javascript to calculate the transition.

The final result was that the application calculates (in R) thirty-three 2D projections from the data each second using geodesic interpolation along the tour path, transports these to D3, which then removes the previous scatterplot and draws a new one as required. This created an animation with seamless, uniform transitions that was faithful to the tour algorithm.

3.5.2 Facilitating User Interactivity

The other main challenge was the implementation of the UI design. In the application's user interface, there are two types of UI element. The first type requires the tour to be discarded and a new one started. An example of this the tour type dropdown. The second type does not require the tour to be restarted; an example of this would be the speed slider, which dynamically alters the angle per second argument to the `tour` function.

In a Shiny app, when a user interacts with a UI widget, its value is recorded in the object `input` as `input$variable`, and is passed to the server object to be used in R code. It is not possible to access such input variables outside of so-called ''reactive" environments in Shiny. These environments are essentially code blocks that are run whenever a new value is pushed to the server from the UI, and include `reactive()`, `observe()`, `observeEvent()` and a reactive variable object called `reactiveValues`. `observeEvent()` defines a code block that is run whenever a reactive-type variable is changed, for example input variables from UI widgets.

To enable interactivity, the code that starts a new tour, calculates tour paths and sends the data to the web browser for rendering is included in an `observe()` environment together with an `invalidateLater()` method which keeps the code block running in the background. Variables inside an `observe()` environment are considered to be inside the

scope of that function and can't be accessed outside of it unless the variables are referring to a `reactiveValues()` object; all input variables which require a new tour to be started are set as part of an `observeEvent()` call which updates a `reactiveValues` vector, including defining a new tour. This restarts the tour with the new parameters and the visualisation continues.

UI widgets of the second type simply send parameters to D3 while the tour is running and don't need to be set as part of a `reactiveValues` vector. When these parameters are changed, a message is sent to D3 using `session$sendCustomMessage` which updates the relavant visualisation parameter and the animation continues.

Chapter 4

Applications

4.1 Physics

We previously saw how this implementation of the tour can be applied to multivariate physics data. Particle physics models have multiple parameters, and the tour helps separate two groups based on differences in their shape. In particular, it allows more than two variables to be compared at once, something which is difficult to do visually using other techniques.

4.2 Econometrics

The tour technique can be applied to Econometrics by providing a way to see differences between *any* sets of data, that may be difficult to discern using classical visualisation techniques. The example video in @ref{tourgui} features a dataset built by calculating various time series diagnostic measures, or Cognostics. Each point represents an individual time series, and there are two groups: one group of time series of seismic measuremets, and another with music data. You can see how the grand and guided tours immediately show the difference between the two groups, particularly in the ACF1 (first autocorrelation lag) and entropy variables.

The tour can be applied to any situation requiring classification of groups with different centres and can expose interesting features which are not immediately obvious using simple statistical functions.

Chapter 5

Conclusions

The work that has been done is a substantial step forward in providing an easy to use, interactive interface to the tour. However, the work is not finished.

5.1 Speed

Computing and interpolating between bases can be computationally expensive for large datasets, and this was found even with as few as 5000 data points. Some work in optimisation needs to be done, potentially saving projections in a buffer before viewing in order to make it smooth, or parallelisation of the matrix multiplications, or some other optimisation to improve speed.

Likewise, rendering many points using SVG technology can strain the resources of the computer running the web browser, and if the computer is both calculating the bases and displaying them with D3, the problem is compounded. If the bottle-neck in animating scatter plots proves to be too much going forward, alternative rendering technology may be required, such as Canvas or other software.

5.2 Density displays

The density display as implemented in the current working version currently is not separate for each group, and the number of bins used for the kernel density estimate is

variable, according to the D3 library used to produce it. Ideally, there would be multiple density displays for each group, and the number of bins should be fixed.

5.3 External data sets

Currently, the only way to load in a data set for use is to modify the code and add one's own `read_csv` line. Ideally there would be an option for the user to select their own CSV file for analysis, using Shiny's `fileInput()` UI widget; alternatively, external connections could be managed using any of the variety of third-party remote connections available to R, including HTTP download or SQL-type connections.

5.4 Manual tour controls

Sliders to manually manipulate the combinations of the different variables could be added in order to allow the user to manipulate the projection directly. Instead of taking a random or guided tour through the projection space, per-variable sliders could be added as a factor to modify or replace the relevant cells in the projection matrix $\mathbf{A}$. This would allow the user to add or remove specific variables with preselected weights, in order to view the effect of adding/removing variables in clearer detail. Cook et al. (2007) Swayne et al. (2011) Lee et al. (2005) Cook et al. (1995) Cook, Buja, and Cabrera (1993) Wilkinson and Willis (2008) Various (2017) Wickham et al. (2011) Huang, Cook, and Wickham (2012) Wilkinson and Anand (2012)

Appendix A

Supplementary Material

Code for the project is available as a GitHub respository at https://github.com/makipp/TourGuiD3.

This thesis is also available as a GitHub repository, at https://github.com/makipp/thesis.

D

Bibliography

Cook, D, A Buja, and J Cabrera (1993). Projection Pursuit Indexes based on Orthonormal Function Expansions. *Journal of Computational and Graphical Statistics* **2**(3), 225–250.

Cook, D, A Buja, J Cabrera, and C Hurley (1995). Grand Tour and Projection Pursuit. *Journal of Computational and Graphical Statistics* **4**(4), 155–172.

Cook, D, A Buja, EK Lee, and H Wickham (2007). Grand Tours, Projection Pursuit Guided Tours and Manual Controls.

Huang, B, D Cook, and H Wickham (2012). tourrGui: A gWidgets GUI for the Tour to Explore High-Dimensional Data Using Low-Dimensional Projections. *Journal of Statistical Software* **49**(6), 1–12.

Lee, EK, D Cook, S Klinke, and T Lumley (2005). Projection pursuit for exploratory supervised classification. *Journal of Computational and Graphical Statistics* **14**(4), 831–846.

Swayne, DF, A Buja, DT Lang, and D Cook (2011). GGobi: a data visualization system. *Astrophysics Source Code Library*.

Various (2017). *shiny: Web Application Framework for R*. https://cran.r-project.org/web/packages/shiny/index.html (visited on 05/02/2017).

Wickham, H, D Cook, H Hofmann, and A Buja (2011). tourr: An R Package for Exploring Multivariate Data with Projections. *Journal of Statistical Software* **40**(2), 1–18.

Wilkinson, L and A Anand (2012). *scagnostics: Compute scagnostics - scatterplot diagnostics*. https://cran.r-project.org/web/packages/scagnostics/ (visited on 05/02/2017).

Wilkinson, L and G Willis (2008). Scagnostics Distributions. *Journal of Computational and Graphical Statistics* **17**(2), 473–491.