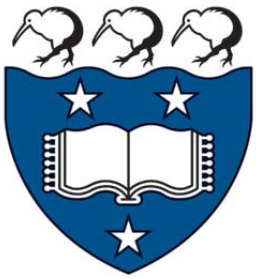
Visualising the Saddlepoint Approximation

Alice Miranda Hankin



Master of Science
Department of Statistics
The University of Auckland
New Zealand

Abstract

The saddlepoint approximation is a tool that approximates the probability density function of a random variable, given only the moment generating function. Though developed to estimate the density function for estimators, it has implementation in many different areas of science. This report details the development of three applications that allow the user to visualise the saddlepoint approximation via comparison to a simulated approximation of the true density. The applications are designed to allow the user to explore, using an interactive user interface, how the saddlepoint approximation is generated, particularly with reference to exponential tilting. The first application is to be used primarily for educational purposes (particularly for statistics students at the University of Auckland) and exhibits several well-known distributions. The second application demonstrates the saddlepoint approximation to a sum of two known distributions for which the true density function is non-trivial. The final application is designed for those with experience in R and moment generating functions; this allows the visualisation of bivariate probability functions. All three applications are written using the Shiny package in R.

Contents

Chapter 1

Introduction

When considering a probability distribution, it is intuitive to think about the probability mass or density function. The phrase "normal distribution" is almost synonymous with the "bell-shaped curve" visual. However, this is not the only way to define a probability distribution. As well as the more widely known cumulative mass or density function, we can also characterise a distribution by its moment generating function.

This raises the question of whether it is possible to approximate the probability density or mass function if we only have access to the moment generating function. Such an approximation is possible with the use of the saddlepoint approximation.

In this project, a series of data visualisations will be created to help intuitively demonstrate the underlying mathematics behind the saddlepoint approximation. Although these will be primarily designed for teaching purposes, they should also benefit those in the field wishing to visualise their work.

1.1 A Brief History of the Saddlepoint Approximation Method

Henry Daniels first developed the saddlepoint approximation in 1954. Compared to other approximations at the time, such as those suggested by Edgeworth and Pearson, the saddlepoint approximation had much lower errors in the tail regions. It was, therefore, a good choice for a tool to approximate the probability distribution of estimators [1][2]. As well as use in maximum likelihood, the saddlepoint approximation can be applied (and often is) to the Bayesian framework [3]. More recently, we have seen its applications in many different areas, such as financial engineering [4], astrophysics [5], and cell biology [6].

At the University of Auckland, Rachel Fewster has utilised saddlepoint approximations in her work on capture-recapture models for wildlife abundance approximation, and Godrick Oketch is writing code to generate saddlepoint approximations to maximum likelihood estimates and approximate their errors. Rishika Chopara has work in progress with deviances of the saddlepoint approximation and Jesse Goodman has also been researching its asymptotic accuracy [7].

Having an intuitive method to visualise the saddlepoint approximation would benefit numerous people working in many areas of science, hence the need for this project.

1.2 Aims of this Project

This project aims to develop a series of applications in R [8] Shiny [9] that visually demonstrate, particularly with reference to exponential tilting, how the saddlepoint approximation to a distribution is calculated. The saddlepoint approximations in each case are used as a point of comparison for the probability density/mass function approximated by simulation. One application is to be designed specifically for teaching purposes; this will present approximations to known distributions (such as the Poisson, Gaussian, and binomial) that students are more likely to be familiar with. A second application will be designed to present approximations to sums of known distributions, for which the true distribution is non-trivial. The final application will be designed for bivariate distributions with complex moment generating functions, where it is much less intuitive what the saddlepoint approximation is doing. The target audience for this application is academics or those with experience in R who wish to visualise much more complicated approximations.

Chapter 2

Exponential Tilting, the Saddlepoint Approximation, and Poisson Processes

In order to understand the outcomes of this project, it is first necessary to introduce some relevant statistical theory. This chapter introduces the concept of exponential tilting; some context for where the saddlepoint approximation comes from.

2.1 Moment Generating Functions

The moment generating function is a function that is uniquely determined by its probability function and vice versa; they can fully characterise a distribution of a random variable [10]. Moment generating functions allow us to find the mean and variance of a random variable easily via what are called "moments".

2.1.1 Moments

We define the n^{th} moment about 0, m'_n, as:

$$m'_n := \mathbb{E}(X^n)$$

This is also known as the n^{th} raw moment or the n^{th} crude moment [11].

By the definition of an expected value, for a continuous random variable X with probability density function $f(x)$,

$$m'_n = \int_{-\infty}^{\infty} x^n f(x) dx$$

Similarly, for a discrete random variable X with probability mass function $f(x)$,

$$m'_n = \sum_x x^n f(x)$$

2.1.2 Defining the Moment Generating Function

Given a random variable X with probability density/mass function $f(x)$, if there exists an $h > 0$ such that $M(s) = \mathbb{E}(e^{sX})$ for $|s| < h$, then $M(s)$ is called the moment generating function, or MGF [12].

For a continuous random variable X with probability density function $f(x)$, under certain conditions, we have the equality:

$$\begin{aligned} M(s) &= \int_{-\infty}^{\infty} e^{sx} f(x)dx \\ &= \int_{\infty}^{\infty} \left(1 + sx + \frac{1}{2!}s^2x^2 + \dots\right) f(x)dx && \text{by Taylor series expansion} \\ &= 1 + sm_1' + \frac{1}{2!}s^2m_2' + \dots && \text{by the definition of a moment, section 2.1.1} \end{aligned} \tag{2.1}$$

The equality above holds for discrete random variables, as well as continuous random variables, the integrals being replaced by summations.

It is interesting to note, and apparent from the formula above, that taking the n^{th} derivative with respect to s of $M(s)$ at 0 gives $\mathbb{E}(X^n)$.

2.1.3 Defining the Cumulant Generating Function

The cumulant generating function (CGF) of a random variable X, $K(s)$, is calculated by taking the natural log of its moment generating function;

$$K(s) = \log M(s) = \log \mathbb{E}(e^{sX})$$

.

Cumulant generating functions will be used many times throughtout this project, and we will always use the capital letter K to refer to them.

2.2 Exponential Tilting

We have seen in section 2.1.3 that $K(s) := \log \mathbb{E}(e^{sX})$. Using the definition of the expected value, we can rearrange this to look like:

$$\int_{-\infty}^{\infty} \exp\{sx - K(s)\} f(x)dx = 1$$

The integrand $\exp\{sx - K(s)\}f(x)$ can be viewed as a density function, since the area below the curve is 1. This is called the s-tilted density [3].

Let's plot the standard normal density function and try tilting it.

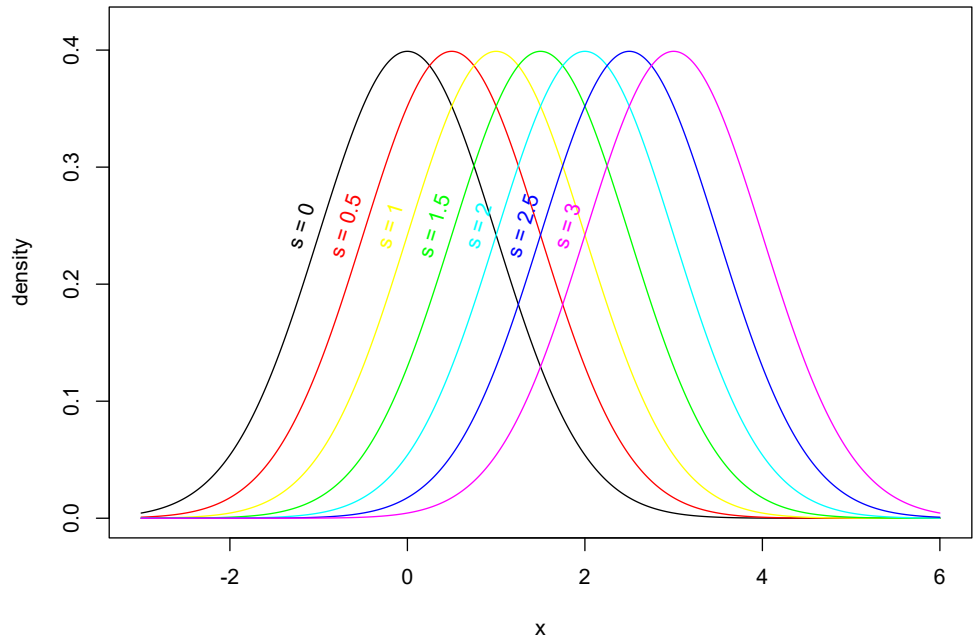


Figure 2.1: The standard normal distribution, as tilted by factor $s \in (0, 0.5, ..., 3)$

In the figure above, the black line labelled $s = 0$ is the untilted probability density function of a standard normal random variable. The standard normal density is tilted by each factor in $(0, 0.5, 1, ..., 2.5, 3)$, and these new densities are plotted.

It is the case that, when we shift the Normal(0,1) distribution by s, we end up with the Normal(s,1) distribution. In fact, in shifting the Normal(μ, σ) distribution by s, we end up with the Normal($\mu + s\sigma^2, \sigma$) distribution; see Appendix A.1.2 for proof.

Let's try the same, this time with an exponential distribution with parameter 0.2; take tilting parameter $s \in (-1, -0.8, ..., 0.2)$.

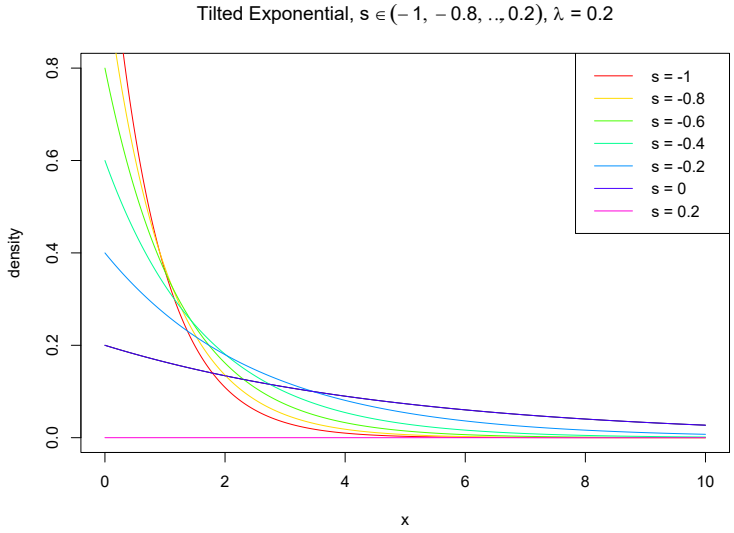


Figure 2.2: The exponential distribution with fixed rate parameter of $\lambda = 0.2$, tilted by factor $s \in (-1, -0.8, ..., 0, 0.2)$

Now, let's plot an Exponential distribution with rates $\lambda \in (0, 0.2, ..., 1.2)$

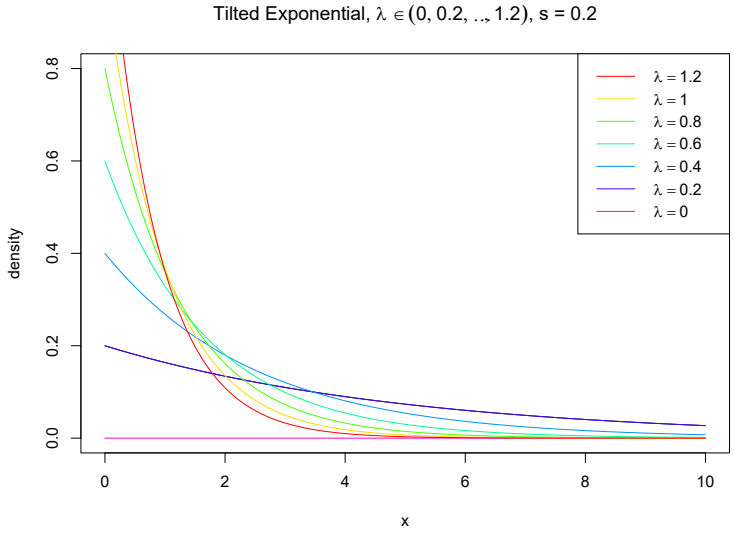


Figure 2.3: The (untilted) exponential distribution with rate parameters $\lambda \in (0, 0.2, ..., 1, 1.2)$

Figures 2.2 and 2.3 are identical. If we plot an s-tilted Exponential(λ) distribution, we get the same as if we plotted an Exponential($\lambda - s$) distribution. The proof can be found in Appendix A.1.3.

Although both examples that are included here have the characteristic that tilting the distributions results in a member of the same distribution with a different parameter, this is not always the case. In fact, the normal and exponential distributions belong to a particular family of probability distributions which are to be discussed in the following

section.

2.3 The Natural Exponential Family

Suppose a family of probability density functions can be written in the form:

$$f(x \mid \theta) = h(x) \exp\{\theta x - A(\theta)\} \tag{2.2}$$

Then - as long as $h(x)$ and $A(\theta)$ are defined, and the support of f does not depend on θ - we say that f is a natural exponential family of distributions [13].

Many well-known distributions are natural exponential families. The normal distribution with known variance, the Poisson distribution, and the binomial distribution with known number of trials are some of many [14].

When a probability density function belongs to an exponential family, the tilted distribution belongs to the same family as the original. In fact, the natural exponential family is alternatively defined as the different exponential tiltings of a random variable [15].

2.4 The Saddlepoint Approximation

The saddlepoint approximation is a formula for approximating a probability density/mass function from its cumulant generating function [3].

For continuous random variable X, the saddlepoint approximation to its density function f at x is:

$$\hat{f}(x) = \frac{1}{\sqrt{2\pi K''(s)}} \cdot \exp\{K(s) - sx\}$$

where s is defined as being the solution to the equation $K'(s) = x$. Note that s is a function of x and is unique.

It is trivial to calculate the saddlepoint approximation for the normal distribution; the saddlepoint approximation for a Normal(μ, σ^2) distribution is also a Normal(μ, σ^2) distribution.

Often, however, the saddlepoint approximation of a distribution is not itself. For the Exponential(λ) distribution, for example, it is easy to show that the saddlepoint approximation is $\frac{\lambda}{\sqrt{2\pi}} e^{(1-x\lambda)}$. Not only is this not the probability density function for an exponential distribution, but also it does not integrate to 1; it is not a probability distribution at all. In order to turn it into such, we can normalise it.

2.4.1 Normalised Saddlepoint Approximation

The normalised saddlepoint density is equal to the saddlepoint density $\hat{f}(x)$ multiplied by a constant c such that $\int_{-\infty}^{\infty} c\hat{f}(x)dx = 1$ (or, in the discrete case, $\sum_x c\hat{f}(x) = 1$). This ensures that the normalised saddlepoint density is, in fact, a probability function.

Recall from the previous section (2.4) that we had the saddlepoint approximation to the exponential distribution as being $\frac{\lambda}{\sqrt{2\pi}}e^{(1-x\lambda)}$. If this is integrated, the result is $\frac{e}{\sqrt{2\pi}}$; we require $c = \frac{\sqrt{2\pi}}{e}$. The normalised saddlepoint approximation is therefore $\frac{\sqrt{2\pi}}{e} \times \frac{\lambda}{\sqrt{2\pi}}e^{(1-x\lambda)} = \lambda e^{-x\lambda}$. We have demonstrated that the normalised saddlepoint approximation to the exponential distribution is itself.

Although both examples here have the feature that the normalised saddlepoint approximation to the distribution function is itself, this is not always the case. For most probability functions, the saddlepoint approximation is not exact. However, it does tend to be a good approximation for the tails of the probability function [2] [16].

The normalised saddlepoint approximation is not often used in practice, and so we will not be using it from hereon out, but it is worth mentioning for completeness.

2.4.2 Relation to Exponential Tilting

This section will discuss how the saddlepoint approximation links to exponential tilting. Rather than the formula existing as a given, hopefully this section will give some insight into how Daniels would have developed the approximation in the first place.

Given true density function $f(x)$, suppose we are interested in finding the value of the saddlepoint approximation $\hat{f}$ at $x = x_0$. This is equivalent to the following:

1. Consider the tilted density function – this is tilted by some s_0 such that the mean of the tilted distribution is x_0.
2. Approximate this tilted density by the Gaussian distribution around x_0
3. Put back the tilting factor $\exp\{K(s_0) - s_0x_0\}$ to get an approximation for $f(x_0)$

Of course, we don't know the true tilted density. The way we approximate it is to find the Gaussian distribution with the same mean and variance. We have chosen the tilting factor s_0 such that the mean is x_0 so this is known. It can be shown that the variance is given by $K''(s_0)$. For more details, see appendix A.2.

The main takeaway from this proof is that there is a relationship between x_0 and s_0 - namely that $x_0 = K'(s_0)$. This is a incredibly important fact, and will be referenced time and time again throughout this report.

Of course, these three steps have to be done for each value of x to get the approximation of the full probability function.

2.4.3 The Multivariate Case

Suppose we have a random vector $\mathbf{X} = (X_1, ..., X_n)$. Then we define the multivariate moment generating function $M(s)$ for $s = (s_1, ..., s_n)$ as

$$\mathbb{E}(e^{s_1X_1+\dot{\cdot}\dot{\cdot}\dot{\cdot}+s_nX_n})$$

.

Similarly, the cumulant generating function $K(s)$ is defined as $\log(M(s))$. The first and second derivatives of the cumulant generating function are defined in the natural way; $K'(s)$ is the vector of partial derivatives, and $K''(s)$ is the Hessian matrix.

The saddlepoint approximation is then given by:

$$\hat{f}(\mathbf{x}) = \frac{\exp(K(s) - s\mathbf{x})}{\sqrt{\det(2\pi K''(s))}} \quad [7]$$

Here, just as in the univariate case, $K'(s) = x$. The proof for this directly follows from the univariate proof we've seen in A.2.

2.5 Poisson Point Processes

Although point processes are not strictly relevant to the statistical theory behind the saddlepoint approximation, they will be utilised in the application, so it is worth giving some brief background information here. Note that some of this information is adapted from my honours project which can be found on my GitHub [17].

2.5.1 Point Patterns

A point pattern is a collection of events occurring in time and/or space. This space could be one dimensional (for example, a period of one hour of time), two dimensional (for example, the country of New Zealand), or multi-dimensional.

For example, if the events we are interested in are the times of bus arrivals in a one hour period, first consider the hour-long period of time as a one-dimensional space. Denoting the space, S, by a line and each arrival as a $\times$ symbol, one eventuality might look like Figure 2.4 below.

Figure 2.4: An example of a one-dimensional point pattern; the $\times$ symbols represent events in the one-dimensional space represented by the line

We can use the function $N(B)$, for $B \subseteq S$, to denote to the number of points within region B. For example, in Figure 2.4, $N(S) = 5$ since there are five points within the entirety of the space S.

In the case that no two points of the point process are coincident, it is useful to think of a point process as a random set of points in S, which can denote $\mathbf{P}$ [18].

2.5.2 Intensity Measures and Functions

Let $\mathbf{P}$ be a point process on the space $S = \mathbb{R}^d$. We define the intensity measure ν on S as:

$$\nu(B) = \mathbb{E}[N(B)], \quad B \subseteq S \quad [18]$$

It's worth noting that for this to make sense we must have some conditions - in particular the constraint that $\nu(B) < \infty$ for all compact B. The elaboration on this is outside the scope of this project.

If the intensity measure of $\mathbf{P}$ in $\mathbb{R}^d$ satisfies

$$\nu(B) = \int_B \lambda(u)du$$

for some function λ, then we call λ the intensity function of $\mathbf{X}$.

2.5.3 Poisson Point Processes

Let X be a discrete random variable such that $X \sim \text{Poisson}(\lambda)$. Then X has the following probability mass function:

$$\mathbb{P}(X = x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

λ is called the rate or the intensity of the process, and is also the mean and variance of the distribution.

A one-dimensional Poisson point process on $S \subset \mathbb{R}$ with intensity measure Λ is a point process that fits the criteria [18]:

- For every bounded interval $(a, b]$, the count $N(a, b]$ has a Poisson distribution with mean $\beta(b - a)$
- If $(a_1, b_1], ..., (a_m, b_m]$ are disjoint bounded intervals, then the counts $N(a_1, b_1], ..., N(a_m, b_m]$ are independent random variables

It is worth noting that the intensity function or measure characterises a Poisson point pattern.

2.5.4 Mapping

Suppose we have a one-dimensional Poisson point pattern. We can map this to another point pattern by applying a fixed transformation $g : \mathbb{R} \to \mathbb{R}$ to each individual point [18]. We can see what this looks like in Figure 2.5.

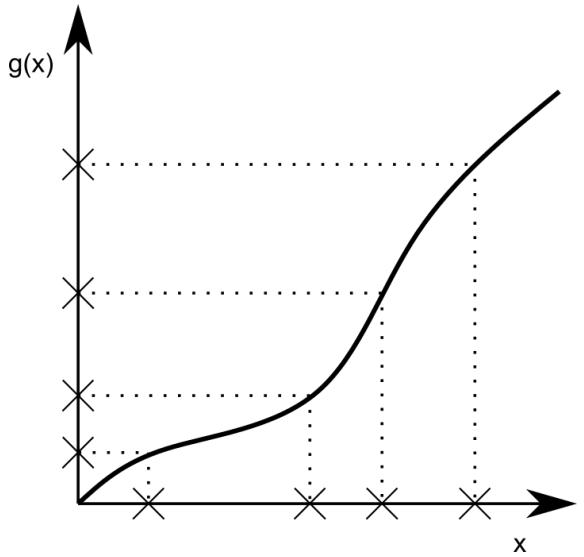


Figure 2.5: Visual representation of the mapping of a one-dimensional point process

Any mapping which has a continuous inverse transforms one Poisson process into another [18].

2.6 The Mathematics Underlying The Simulation

This section will discuss the mathematical basis for the applications. The concept of Poisson point processes is to be utilised in a particular way to illustrate exponential tilting. This can then be used, as we've seen in section 2.4.2, to create the saddlepoint approximation.

2.6.1 How Poisson Processes Can Be Used To Demonstrate Exponential Tilting

Suppose Y is an infinitely long list of points corresponding to a one dimensional homogeneous Poisson process with rate 1.

We can assign each point a random value X, which comes from some other distribution $f(x)$. In this example, X is normally distributed with mean 0 and variance 1.

Plotting x against y, one simulated outcome might be:

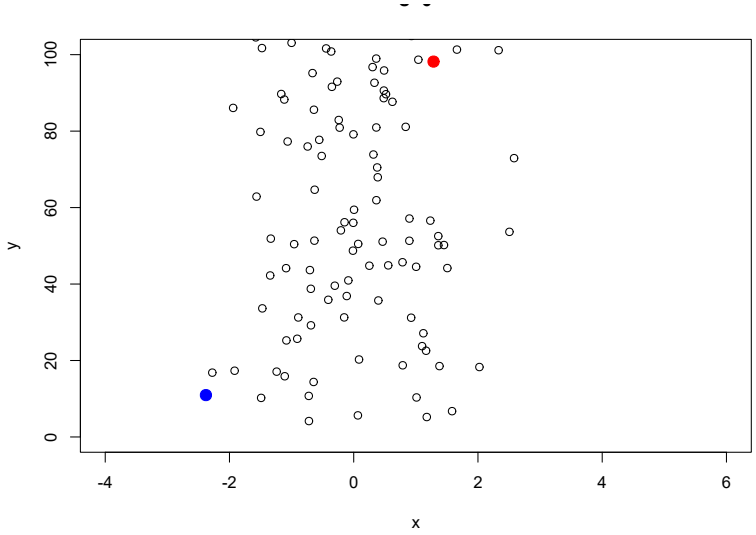


Figure 2.6: Scatterplot; the x axis has a Normal(0,1) distribution, and the y axis is a Poisson point process with rate 1

This scatterplot has been drawn using the default plotting function in R [8].

Let's say we apply a transformation/mapping to this where $(x, y) \mapsto (x, ye^{-sx})$ for some s. Note that the x coordinates do not change, but the amount the y coordinate changes does depend on its value of x.

We can see what happens if we let s be $0.5, 1, 1.5$, and 2 in our example below:

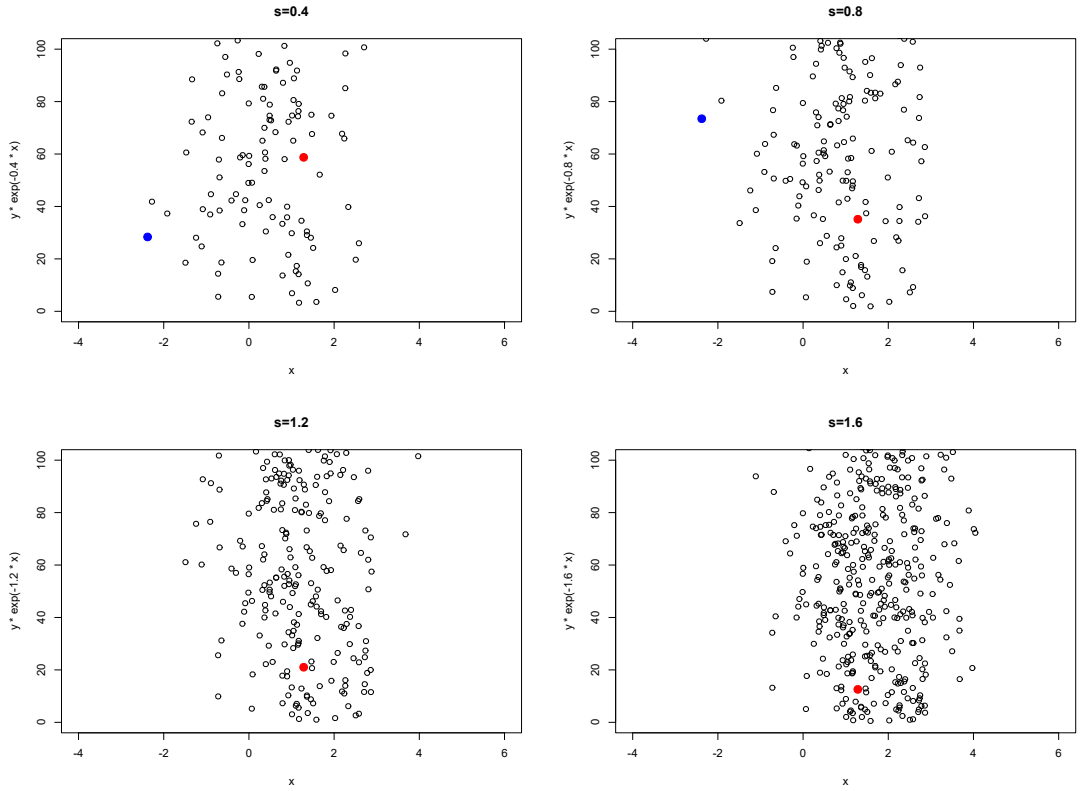


Figure 2.7: The same points as in figure 2.6; this time a mapping from (x, y) to (x, ye^{-sx}) has been applied for $s \in \{0.5, 1, 1.5, 2\}$

Points with low values of x will move upwards (y will increase) and high values of x will move downwards (y will decrease). The same points are coloured in in each of these plots, as well as in figure 2.6, to illustrate this. The red point decreases as s increases, whilst the blue point increases. Note that the blue point is only visible for s values below 0.93; it is out of sight in the bottom two plots.

We can see from figure 2.7 that the number of points on screen changes as we change s; the density of points increases as s increases. We are interested in normalising these plots so that the intensity of points appears the same for any s. It is possible to show (see proof in appendix A.3) that the normalising factor is the moment generating function of x.

If we instead plot mye^{-sx} where $m = M(s)$ is the normalising function, we get the following:

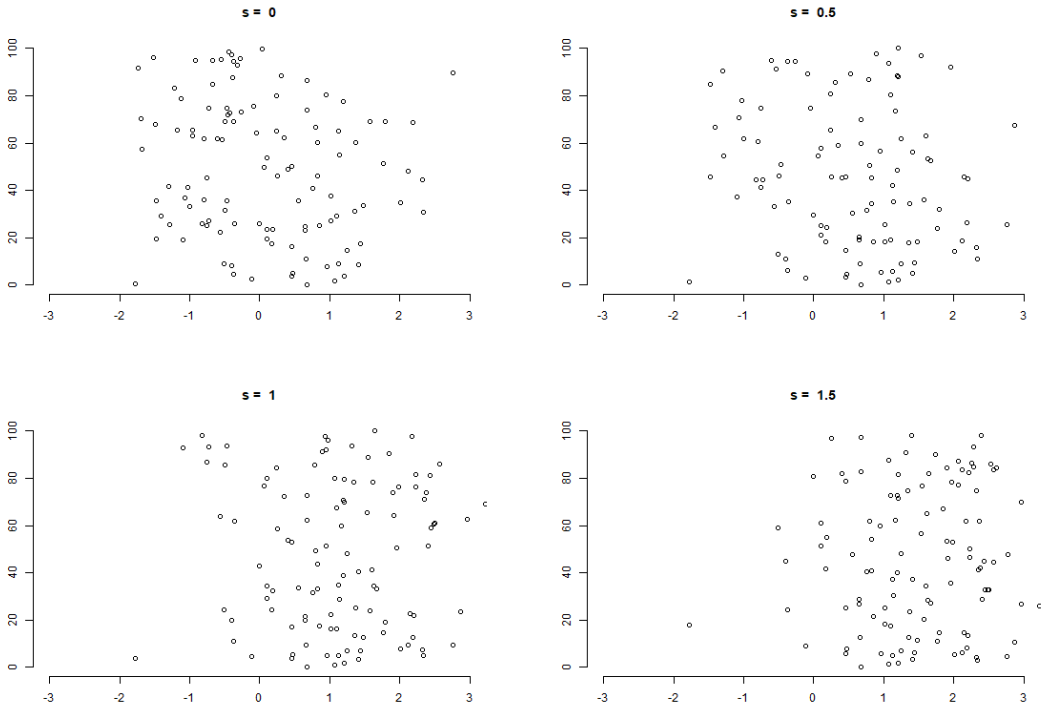


Figure 2.8: The same points as in figure 2.6; this time a mapping from y to $yM(s)e^{-sx}$ has been applied for $s \in \{0.5, 1, 1.5, 2\}$

Now, since we have infinite points, we require some sampling mechanism in order to understand the new density of points on the x-axis. It is intuitive to be looking only at points within in some finite y range. Doing so leaves us with the s-tilted density on the x-axis.

In this example, we started with x being distributed as per a standard normal distribution. This means that if we set s to be equal to one, we can see that the points are distributed like a Normal$(1, 1)$ distribution. This is because, as we have seen in 2.2, a standard normal density tilted by 1 is a Normal(1,1) density. Similarly, when $s = s_0$, the points have a Normal$(s_0, 1)$ distribution.

The proof of this follows directly from the previous one, but can be found in more detail in appendix A.4.

2.6.2 The Infinite Points Assumption

In the previous section, the discussion was premised on the idea that we have an infinitely long list of coordinates. In a simulation, this cannot be the case. Suppose we have only a finite number of points. If these points are tilted by a large enough value of s, there will simply not be enough points in the tails to accurately represent the true tilted density there.

For example, let's generate n points from a $N(0, 1)$ distribution, where $n \sim$Poisson(100).

We will let the y-axis range from 0 to 200. A red line will illustrate the maximum y value (at each x-value) that points can have.

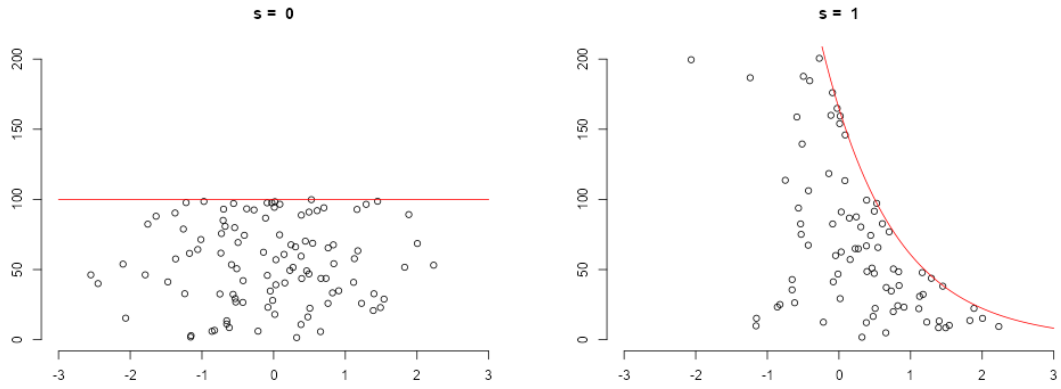


Figure 2.9: 100 points tilted by $s \in \{0, 1\}$. A red line shows their upper limit in y

At $s = 0$, the red line is at $y = 100$. If we tilt the distribution by factor s then every point on the line $(x_0, 100)$ will map to $(x_0, 100me^{-sx_0})$; at $s = 1$, we have $y = 100me^{-x}$ plotted in red. This is the maximum value that a point can have after tilting. If we increase the number of points to 1000, like in the figure below, we can see than the red line is no longer cutting off the distribution as severely, and the density looks closer to the $N(1, 1)$ distribution, like we would expect.

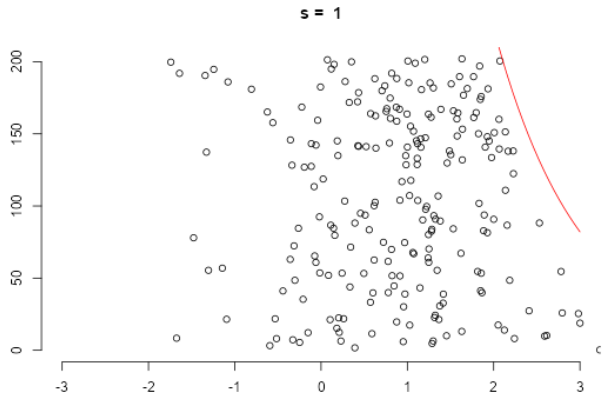


Figure 2.10: 1000 points tilted by $s = 1$. A red line shows their upper limit in y.

The density of simulated points on the screen approaches the true tilted density as the number of simulated points approaches infinity.

Before getting into how we can use these concepts in the application, it is first necessary to discuss what makes a good data visualisation; this is the topic of the next chapter.

Chapter 3

Effective Data Visualisation

This project aims to communicate the processes underlying the saddlepoint approximation; we want our data visualisation to be as effective at conveying understanding as possible. Therefore, it is worth looking at some literature that can help guide us towards creating the best possible visualisation.

Thanks to Paul Murrell for introducing me to the ideas in this chapter. More information about effective data visualisation can be found on his website [19].

3.1 Gestalt Theory

Given a completely random pattern, we as humans tend to see patterns and structure that doesn't exist in the underlying model. Gestalt rules describe the "strong inferences we make about relationships between visual elements from relatively sparse visual information" [20]. Essentially, these rules describe the way in which we organise and group elements of an image together subconsciously, viewing them as a cohesive system, rather than as disparate parts [21].

There are seven main principles that are associated with Gestalt theory [20][21]:

- Closure: Incomplete shapes are perceived as complete.
- Connection: Elements that are visually tied to one another seem to be related.
- Continuity: Partially hidden elements are completed into familiar shapes.
- Common Fate: Elements sharing a direction of movement seem to be related.
- Figure and Ground: Visual elements are taken to be either in the foreground or the background.
- Proximity: Elements that are near to one another seem to be related.
- Similarity: Elements that look similar seem to be related.

One has to be careful when creating a data visualisation; since the viewer will always follow these rules, you have to be careful to make elements appear related if and only if it

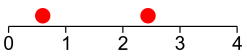
contributes to the understanding of the data.

3.2 Visual Channels

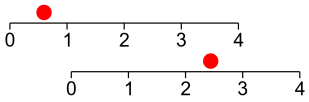
The same data can be displayed in many different ways. A bar chart, line graph, or pie charts could all be used to show the exact same dataset. These charts are not, however, equally as effective when it comes to the viewer's interpretation of the relative size of each of the data points. There are numerous articles that can be found online admonishing the pie chart [22] [23], and for good reason - humans are simply not as good at comparing two areas as they are at comparing two lengths.

William Cleveland and Robert McGill did experimental studies on graphical perception in the 1980s [24]. They were interested in finding which "visual channels" or "elementary codes" were more effective at communicating data to the viewer. This was done by asking the participants in the study to estimate values in a chart when encoded by various visual channels. The results of these experiments was replicated later on by Tamara Munzner in 2014 [**?**].

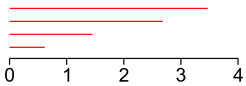
Cleveland and McGill ended up with a list of visual channels for displaying continuous data, which they could then rank in order of effectiveness. This list is included here, with the most effective at the top, and least effective at the bottom.



Position on a common scale



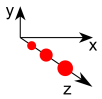
Position on unaligned scales



Length

Tilt

Area



Depth

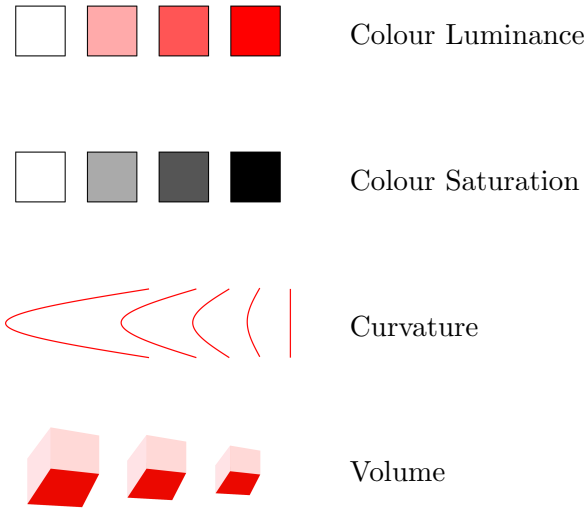


Table 3.1: Visual channels for continuous data in order of effectiveness

The same was done for discrete data; that table can be found below.

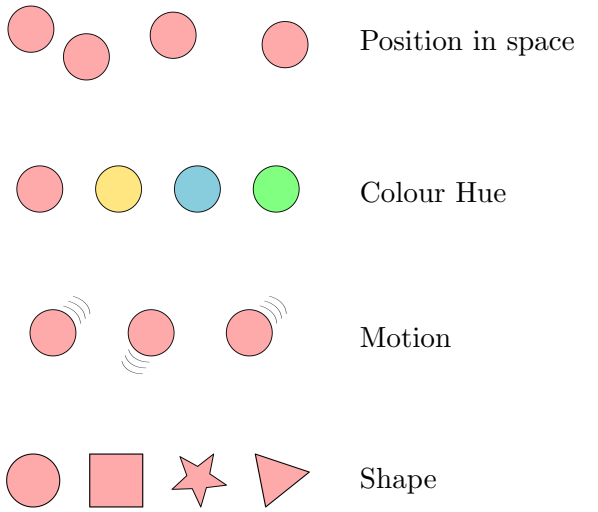


Table 3.2: Visual channels for discrete data in order of effectiveness

3.3 The CRAP Guidelines

In his book, "presentationzen", Garr Reynolds talks about the "big four" principles of graphic design: contrast, repetition, alignment, and proximity [25]. Although he mostly discusses these in context of creating presentations, they can also be applied to data visualisations [26].

Here, I talk a little about each of the big four.

Contrast
To make good use of contrast involves leveraging the viewer's innate ability to scan and look for differences. If one item is clearly dominant in an image, it gives the viewer a focal point or a "place to begin" [25]. We want the viewer to be drawn to the most important part of the visualisation (i.e. the data itself) rather than the less important parts (such as subtitles, axis ticks, or sources).

Colour choices (such as warm vs cool colours, or dark vs light colours) are just one way of creating contrast. One can also use changes in typography, such as differences in font family (i.e. serif vs sans serif), font weight (i.e. bold vs light), or font size (i.e. large vs small) [26]. Reynolds mentions that manipulation of space (i.e. near vs far, or empty vs filled elements) or positioning (i.e. top vs bottom, or isolated vs grouped elements) can also be used as a means to create contrast [25].

In the figure below, it is apparent that the image on the left, which contains little variation in line weight, font face/size, and colour, is less effective of a data visualisation than the image on the right.

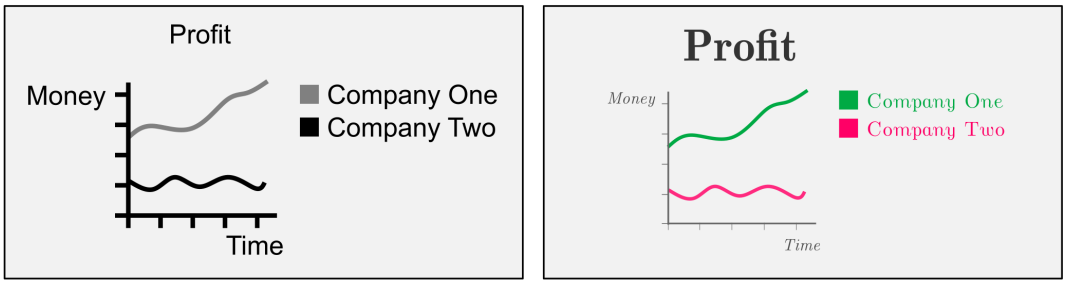


Figure 3.1: An image with low contrast (left) and an image with high contrast (right)

Repetition
Repetition involves the use of similar elements throughout the data visualisation. Like contrast, these elements can be colours, fonts, graphical elements, alignments, and so on. Having a lot of repetition means that the elements of the visualisation are tied together to form one cohesive image.

The two images below, sourced from *Openfield* [27], illustrate the importance of repetition. The top image has created disharmony due to its differing colour palettes, whereas the lower image utilises repetition to create cohesion - the plots look like they belong together.

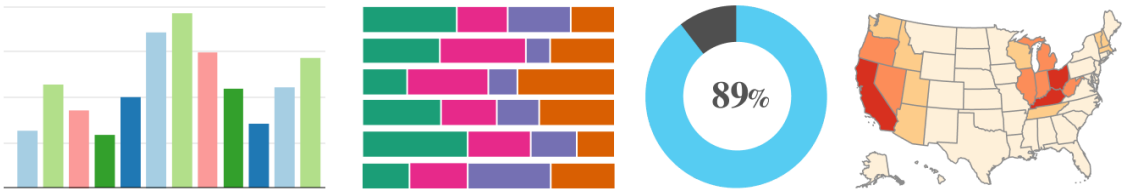


Figure 3.2: A series of graphs with little repetition of colour

Figure 3.3: A series of graphs with much repetition of colour

Alignment
Andrew Heiss describes the principle of alignment: "every item should have a visual connection with something else on the page"[26]. Alignment makes it easy to navigate within the data visualisation and creates a structured final image [28]. Just like with repetition, alignment creates a sense of unity and order among the elements of the visualisation [25]. In data visualisation, alignment refers to having the elements (such as the title, legend, and axis labels) "line up" along vertical or horizontal lines; they should not be randomly floating in space.

In the figure below, we can see an image with lots of aligned elements on the right and few aligned elements on the left. The right image is more cohesive and visually pleasing.

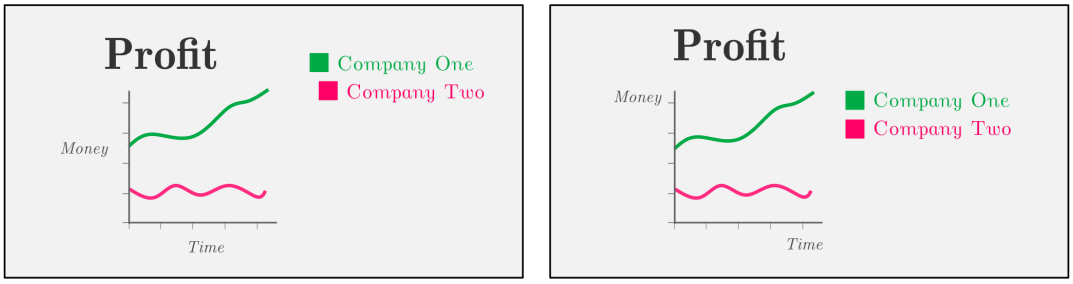


Figure 3.4: An image with little alignment (left) and an image where all the elements are well-aligned (right)

Proximity
Proximity involves elements of an image being placed near each other so they will appear

as a group; this principle is the same as the proximity rule we have seen in the Gestalt theory section (3.1). If objects are placed close together, we are likely to see them as connected, and if they are placed far apart, we are not inclined to associate them with each other. In data visualisation, this rule tells us that axis labels should be near their corresponding axes, and chart descriptions should be near their corresponding charts.

3.4 Use of Colour

There are three main ways that colour is used in data visualisations; to distinguish data from each other, to highlight particular data points, and to represent data values [29]. As we discuss in section 5.2.10, the last two use cases are not needed in our data visualisation, and so here I will discuss the use of colour to distinguish. This involves what is known as qualitative colour scales [29].

3.4.1 Colourblindness and the Okabe-Ito Palette

When choosing colours for data visualisations, it is important to choose colours for the application that are colourblind friendly [29]. It is estimated that 4.5% of the world's population has some form of colourblindness [30]; these people have trouble distinguishing colours that look clearly different to non-colourblind people [29].

In 2002, Masataka Okabe and Kei Ito developed a set of colours that is unambiguous to both non-colourblind and colourblind people, along with some guidelines for assigning colours to drawings [31][32].

The palette can be seen below:

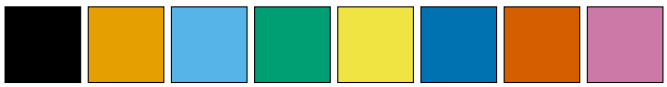


Figure 3.5: The Okabe-Ito Palette

Along with this colour palette, a few of the main points in their presentation were:

- Use redundant coding; show differences in shape as well as colour
- Use warm and cool colours alternately
- Keep contrast in brightness/saturation as well as hue

Additionally, in 2014, Stone, Szafir and Setlur noted that colour distinguishability changes as a function of size [33]; it is easier to distinguish colours as applied to large areas than small ones.

I was keen to use the Okabe-Ito palette and to follow these guidelines when developing my applications - you can see which colours were chosen for the first application and why in section 5.2.10.

Chapter 4

R Shiny

For this project, I will be using the `shiny` [9] package in the R [8] programming environment to develop the applications. Shiny allows us the web-app capabilities and attractive user interface of business intelligence tools such as Power BI or Tableau, but also the code-based nature allows complex code on the back-end [34]. This short chapter will briefly discuss how Shiny works.

4.1 Structure of a Shiny App

To understand the applications produced in this project, it is necessary first to understand the components of an R Shiny application and some of the jargon used when discussing interactive data visualisations.

There are two sections of a Shiny application; the user interface and the server. The user interface (also known as the UI or the front-end) comprises everything the user can see. Elements of the user interface include sliders and buttons with which the user can interact; these are called widgets. A different input value is recorded depending on the state of the widget. For example, if we want to make the user choose between two options, this could be visualised as either a 'radio buttons' widget or a 'select box' widget. In both cases, the result of the user's choice will be recorded in a named variable as an input.

Other parts of the user interface include text, backgrounds, sidebars, graphs and tables. Some of these might update to reflect changes in the inputs - we use the word "reactive" to describe these. For example, a histogram might update reactively so that the number of bins matches a number that is inputted by the user on a slider widget.

The other part of the Shiny app is the server or the back-end. This is all the computations that R is running in the background. In the above example, the server will take the input value and create a plot. This value will be sent back to the UI, where the plot can be placed on the user's screen. This plot, like any plots or text that are passed from the server to the UI, is called an output.

4.2 HTML and CSS in the context of a Shiny application

HTML is the programming language that is used to provide an outline for every webpage on the internet. Using HTML, one can create elements such as headings, paragraphs, images, lists and links in order to structure a very basic webpage. What HTML cannot do is specify *how* to display these elements; this is where CSS comes in.

CSS defines styles for HTML web pages [35]. This refers to the appearance of HTML elements - for example, their colour or their shape. Together, they provide the basis of every website.

In the image below, sourced from MDN Web Docs [36], we can see the difference between a webpage made purely with HTML, and the same webpage with added CSS.

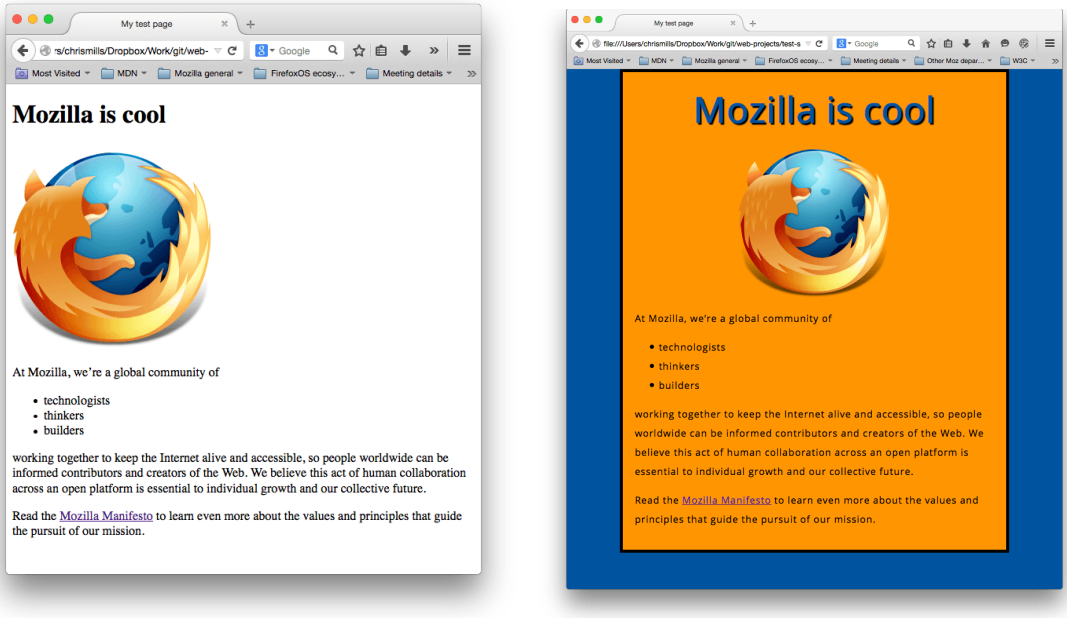


Figure 4.1: Webpage made with pure HTML (left) and with both HTML and CSS (right)

Shiny takes code written in R, and generates HTML/CSS code for us so we don't need to worry about web development. However, it is also possible to use custom CSS code to change the styling of the application. I discuss how custom CSS is used in my first application in section 5.2.11.

Chapter 5

Application - Known Distribution

For this first application, my main goal was to create an application that could be used as an effective learning tool for teaching the concepts of exponential tilting and the saddlepoint approximation. It demonstrates the tilting of several well-known distributions to best show users who might be familiar with them just what the saddlepoint approximation to these distributions would look like and how it is calculated.

5.1 The Initial Application

This rudimentary application was written by Jesse Goodman with help from Rachel Fewster. It was designed initially to give a visual for the process of exponential tilting. In this section, I will discuss how the application works - the code written will act as the jumping-off point for my application. Note that the mathematics behind this code is the same as we have discussed in section 2.6.

5.1.1 The Front End

A figure displaying the appearance of the front-end of the initial application can be seen below.

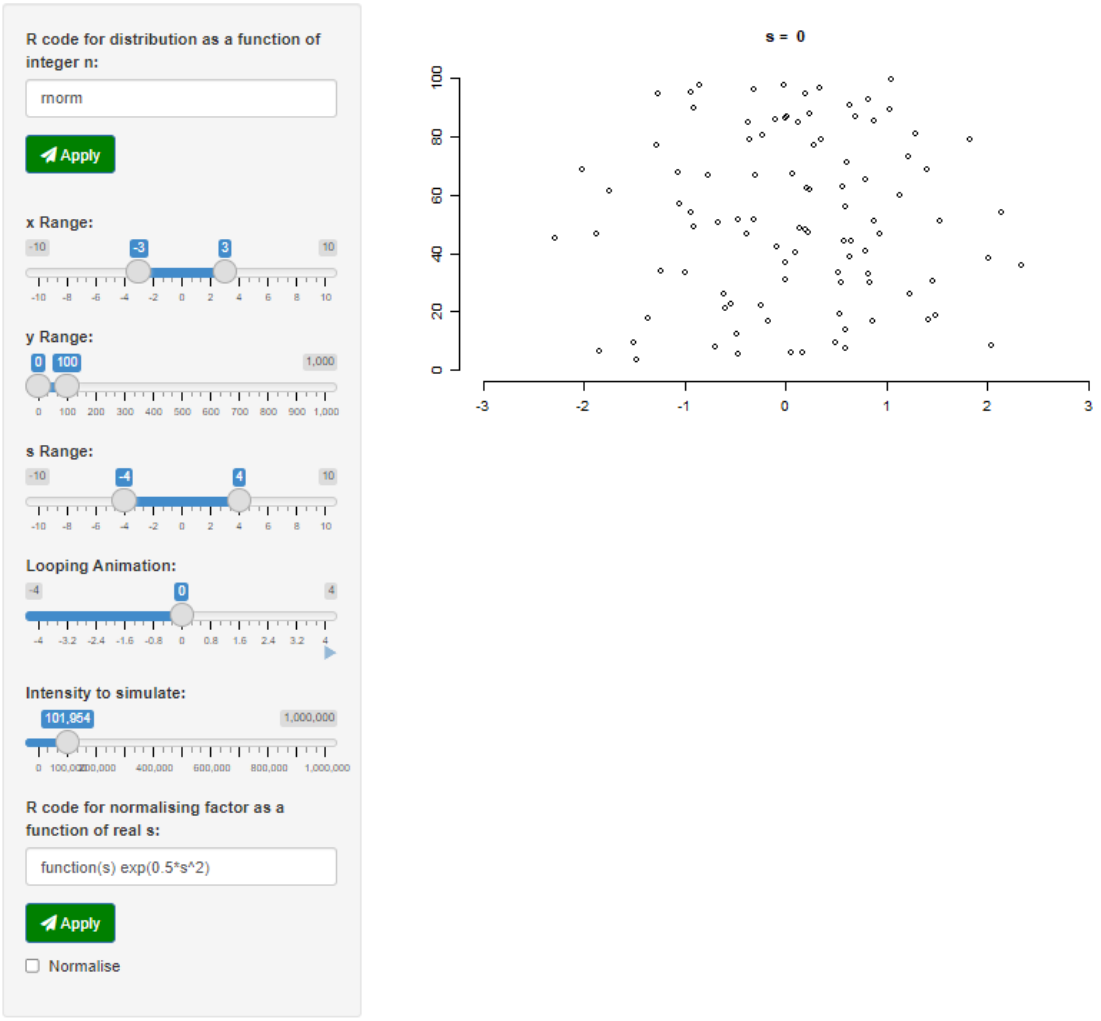


Figure 5.1: The GUI of the initial application

This application produces output similar to that seen in section 2.6. On the left-hand side, we have a panel which contains various widgets where the user can input parameters, and on the right-hand side is the resulting output. Here, I will go through each of the widgets the user can control, and discuss what they do.

- The x-axis and y-axis limits of the plot are controlled by the sliders labelled **x Range** and **y Range**.

- The distribution of points on the x-axis are controlled by the uppermost textbox; in this case, they are normally distributed with mean 0 and variance 1. On the y-axis, the points *always* follow a Poisson point process with parameter 1; the user does not control this.

- The **s Range** slider controls the upper and lower limits of the **Looping Animation** slider; the latter changes the value of s, the amount we wish to tilt the distribution. This is an animation - the user can press the ▶ button which will animate the

distribution tilting, taking values every 0.1 over the whole **s Range**.

- The **Intensity to simulate** slider changes the number of points that are simulated by changing the upper limit of what y-values (before tilting) that points can have.

- Finally, the user must manually enter the moment generating function corresponding to the x-distribution in the lower textbox. The points are normalised if and only if the **Normalise** checkbox is ticked. Recall from 2.6 that normalising the points means that we ensure that the same number of points are on screen at all times.

5.2 Coding Decisions

I will not be including large sections of code in this report, however it is of interest to discuss some of the major decisions made whilst creating the application. Starting with the initial application discussed in section 5.1, I will touch on the steps that were taken (in roughly chronological order) to develop the new application.

5.2.1 The Flexibility-Accessibility Trade-Off

In the original application, the user must enter R code for both the distribution and the normalising factor. Not only does this leave the application open to malicious code, but it also means that the user interface is impossible to use for people who do not have experience in R. On top of this, the user must manually calculate the moment generating function for the distribution in which they are interested.

Allowing user input does mean that the application is much more flexible, though; the user can input any distribution they like. The alternative is allowing the user access to toggles, drop-down menus, and sliders which means they can only choose from a discrete number of options.

We must consider the target audience to decide where to sit on the flexibility-accessibility spectrum. An application used predominantly for educational purposes is best designed with high accessibility; one used by those already knowledgeable about R and moment generating functions is best designed with high flexibility.

Changing the Distribution Input

This first application is designed for educational purposes and, therefore, should be able to be used by anyone interested in learning about the saddlepoint approximation, whether or not they have experience with R and moment generating functions. Using well-known distributions is a good start since most people will already have some intuition about how they might work. It also makes for an intuitive and easy-to-use user interface.

It was imperative, then, for the existing inputs to be changed. In terms of the user interface, it was decided the user would be able to choose between several pre-specified distributions. There would no longer need to be a box for the normalisation; this would automatically be calculated by the program. The difference in inputs can be seen here:

Figure 5.2: Distribution inputs before (left) and after (right)

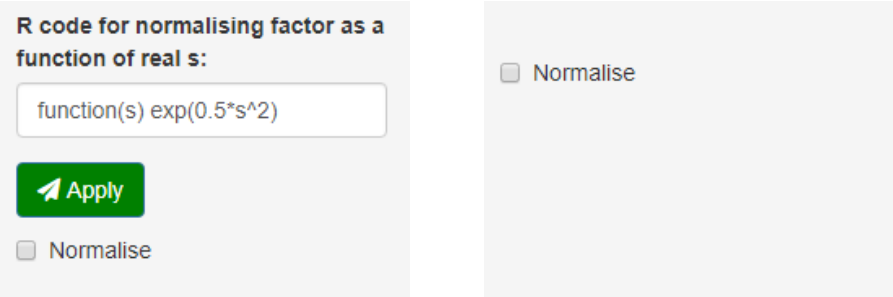


Figure 5.3: Normalising inputs before (left) and after (right)

We have seen that making this change meant that the program was significantly less flexible; I discuss how we can add more flexibility in chapters 6 and 7.

On the server side, this change required a bit of reshaping. Now, all of the moment generating functions must necessarily be hardcoded. At this initial stage, I used two reactive switch statements. When the distribution is changed, it tells the program what distribution to sample the points from, and what the moment generating function is. I further discuss the use of switch statements in the program in section 5.2.4.

5.2.2 Inputting the Parameters

Since the drop-down menu does not have the parameters of the distribution, it was important to create an input for each parameter. For example, the normal distribution will require a "mean" input and a "standard deviation" input.

There were three ways I could have gone about coding the widgets corresponding to these inputs:

1) For each of the eight initial distributions, have unique widgets which appear/disappear reactively.

2) For each of the variable types, have one or two widgets which appear/disappear and change names reactively

3) Have only two widgets which disappear/appear and change names/properties reactively

For example, say we toggle the Poisson distribution on. This has one parameter, rate.

This is restricted to be greater than 0, but it can be a non-integer. Here is what each of the three ways would look like:

1) The "rate" widget would appear. All other widgets (e.g. the "standard deviation" widget) will disappear. The "rate" widget has preset values for the step size, minimum, maximum, and label.

2) The "positive real value" widget would appear. All other widgets (e.g. the "integer between 1 and 10" widget) will disappear. The "positive real value" widget has preset values for step size, minimum, and maximum. The label will react to the distribution selection, so will display "rate".

3) The "first parameter" widget would appear. The "second parameter" widget would disappear; the Poisson distribution only has one parameter. The step size, minimum/maximum values, and the label will update reactively to the distribution selection.

It's worth noting that all of these options would look exactly the same to the user, it is only the back-end that distinguishes them.

The first option is the easiest to code. It solves a lot of problems that the latter two options have where the values of the parameters remain in place when the distribution is changed, raising errors when the value is out of range. The issue with this option is that each existing widget requires its own call to `sliderInput()`, along with a JavaScript expression determining whether it should be displayed. This is not possible to be vectorised, which means that we would have a lot of essentially redundant code.

The third option is the best. Unlike the first option, we don't need to write a whole new call to `sliderInput()`, along with conditional statement - we just need to define parameters for the relevant values for each distribution. This makes the code more streamlined since each of it is all in one place - we don't need to go through each of the sliders' conditional statements if we need to add a new distribution. It is harder to code, but possible, so this is the option that was chosen.

5.2.3 Reactively Updating Parameters

We have already added functionality to change the parameters of the distribution. Initially, the way this worked was that the user would update the parameter slider, then press "Apply" to change the parameters. What would be more interesting, however, would be to allow the user to dynamically change the parameters of the distribution and see a smooth change. We wish to be able to visualise the effect of changing distributional parameters, especially whilst keeping tilting parameter s constant.

Ideally, we want the user to be able to change the parameter slider and the scatterplot should reactively update to reflect the change. No "Apply" button should be necessary. Below, we can see what happens if we code this naïvely.

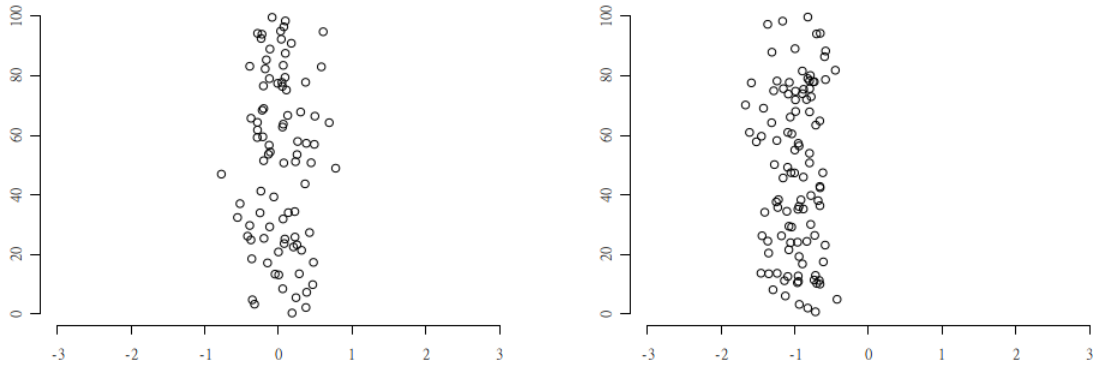


Figure 5.4: Normal distribution with standard deviation 0.3 and mean 0 (left) or mean -1 (right)

You can see in the figure above that the mean has changed from 0 to -1, but the points have completely reshuffled.

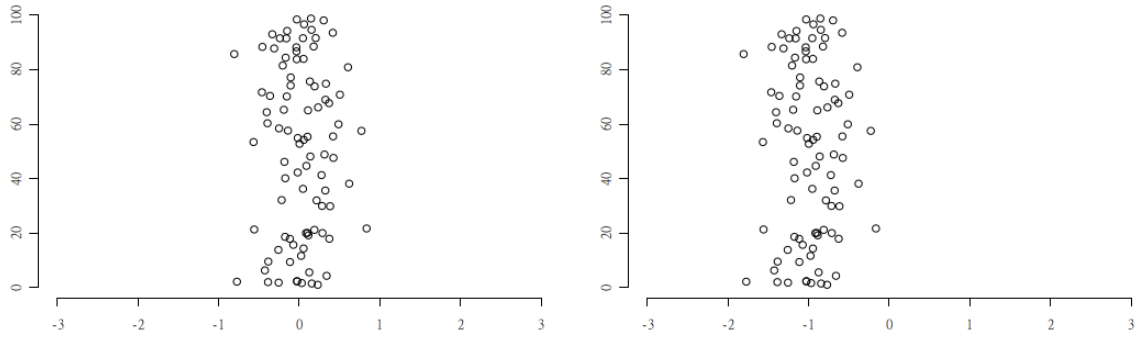


Figure 5.5: Normal distribution with standard deviation 0.3 and mean 0 or mean -1 - this time the structure of the points is retained

In the figure above, even though the mean has changed, the points still have the same pattern. Note that in order to emphasise this, the colours of the points did end up changing; this is further discussed in section 5.2.10.

To implement this, it was necessary to find a way to either remember where the points were before the change, and determine where they move to, or alternatively to remember where the points were in the initial position, and change relative to that. Keep in mind that we also want the ability to change the points if we choose (using a "refresh" button) so we can't just set the seed every time the program is run. As it turns out, the former method is much harder to code than the latter. Yet it remains non-trivial to figure out a way to do this.

For the normal distribution, if we have some $X \sim N(a, b)$ and we would like $Y \sim N(c, d)$, the transformation is trivial since we know $\frac{d}{b}X + (c - \frac{ad}{b}) \sim N(c, d)$. However, for other distributions, the method is not so straightforward.

Consider a X from some continuous distribution with some initial parameters. We wish to transform these to points from that same distribution with new parameters. We can do so by finding the quantiles of the points, then back-transforming them, such as in the following figure.

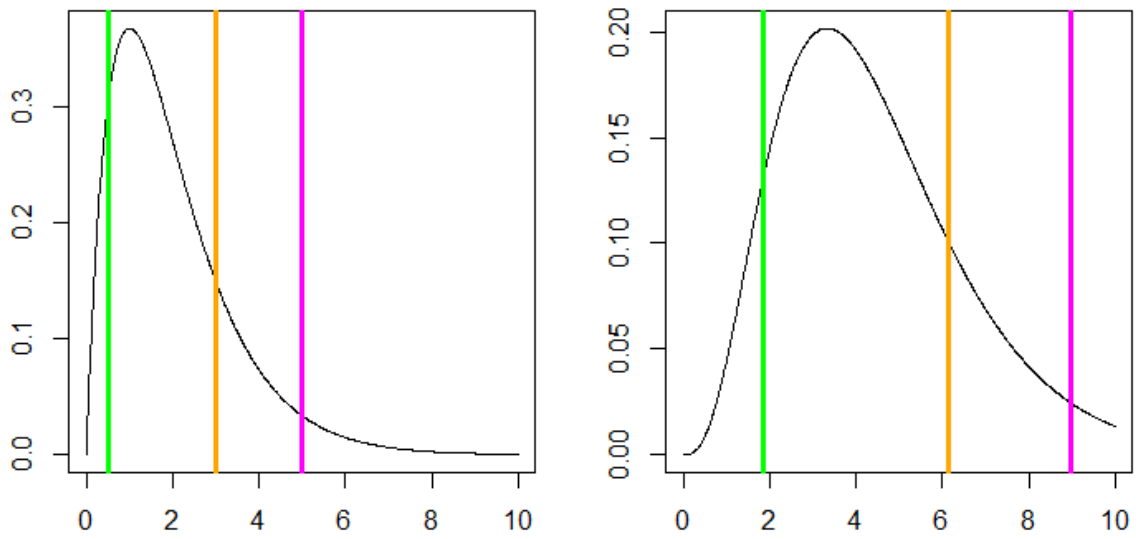


Figure 5.6: Three sampled values from the Gamma$(2, 1)$ distribution (left). Their equivalent values in the Gamma$(4, 0.9)$ distribution on the right have the same quantiles.

If the three vertical lines (say `x`) correspond to points taken from the Gamma$(2, 1)$ distribution, we can place their "equivalent" values in the Gamma$(4, 0.9)$ distribution with the following code:

```
qgamma(pgamma(x, 2, 1), 4, 0.9)
```

Alternatively, we could initially sample from the Uniform$(0, 1)$ distribution, and use these values as quantiles for the new distribution.

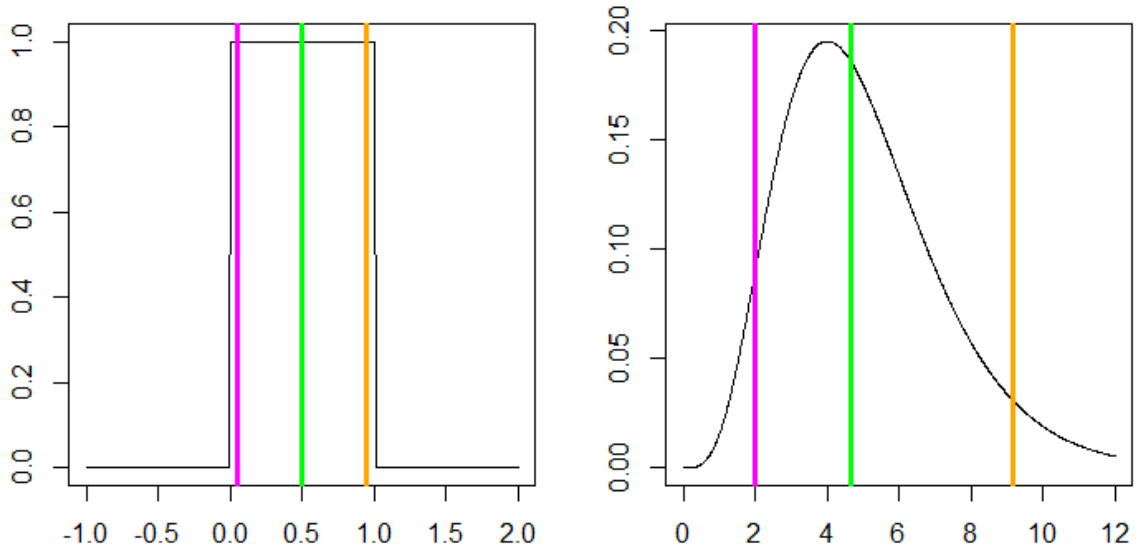


Figure 5.7: Three sampled values from the Uniform$(0, 1)$ distribution (left). Their equivalent values in the Gamma$(5, 1)$ distribution on the right have the same quantiles.

The latter option is more useful for us, since this can be used on both discrete and continuous distributions. It also makes the code slightly more efficient (although this is nowhere near a limiting factor of the code!) since we have one fewer function call.

5.2.4 Use of Switch Functions

Since I have gone down the two slider route (see 5.2.2), I end up with a lot of switch statements that look like this:

```
output$name_second_slider <- renderUI({
  HTML(paste(
    "<b>",switch(input$distrString,
                 "Normal" = "Standard Deviation",
                 "Gamma" = "Rate",
                 "Beta" = "Shape (beta)",
                 "Negative Binomial" = "k",
                 "Binomial" = "n",
                 "Chi-Squared" = "Degrees of Freedom",
                 ""))
  )

})
```

Figure 5.8: A switch statement

This is a statement which looks at the distribution name, and reactively updates based

on it. At the point where I changed this, I had eleven different switch statements dotted around the code which update reactively to the same thing! This is bad coding practice; we want to make it as seamless as possible to add in new distributions.

The solution to this is to consolidate all of the switch statements in one big function which is called at the start of the program

```
switch(distrString,

       "Normal" = {mylist$start = c(1,1);
                   mylist$ddist = function(x, var1, var2) dnorm(x, var1, var2);
                   mylist$rdist = function(n) rnorm(n, mean=mylist$start[1],
                                                    sd = mylist$start[2]);
                   mylist$m = function(s, var1, var2) exp(s*var1+0.5*s^2*var2^2);
                   mylist$firstminmax = c(-10,10);
                   mylist$secondminmax = function(var1, var2) c(0,10);
                   mylist$step = rep(def_step, 2);
                   mylist$max_s = function(var1, var2) 10;
                   mylist$slidernames = c("Mean:", "Standard Deviation:");
                   mylist$scalex = function(new1,new2,x) {a = new2/mylist$start[2];
                                                          b = new1 - a * mylist$start[1];
                                                          a * x + b};
                   mylist$discrete = F
       },

       "Exponential"= {mylist$start = c(1,0);
                       mylist$ddist = function(x, var1, var2) dexp(x, var1);
                       mylist$rdist = function(n) rexp(n, rate=mylist$start[1]);
```

Figure 5.9: Another, longer, switch statement

Now we have one reactive statement which calls this function every time the distribution is changed. The step size, start values and minimum/maximum values for parameter sliders are all updated, as well as the moment generating function of the distribution, amongst other things.

5.2.5 The Saddlepoint Approximation Plot

Next, I wanted to include a graph that compared the true distribution to both the saddlepoint approximation, and also the simulated distribution that is visible in the scatterplot.

For a Gamma$(2.2, 0.9)$ distribution, this is what the end result ends up looking like:

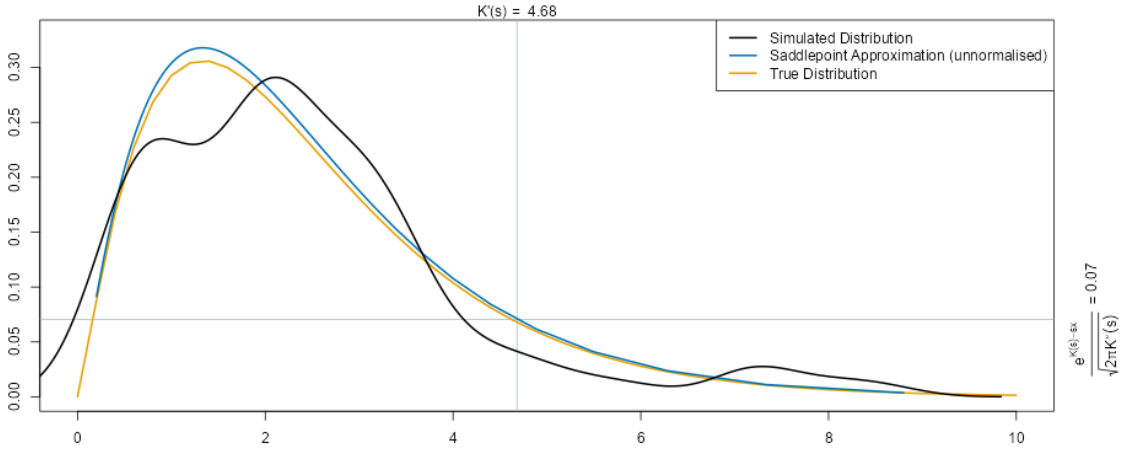


Figure 5.10: The Gamma(2.2,0.9) distribution; the true, saddlepoint, and simulated densities

We can see that the saddlepoint approximation matches the true distribution quite well. The simulated distribution is not as good of a match, but will conform to the true line as the number of simulated points is increased.

I talk about the colours used in section 5.2.10 and I talk about the positioning of the plot in section 5.2.13.

Calculations

Lots of the information about each distribution is hardcoded (i.e. it is not inputted by the user). This includes the true distribution, so this is trivial to plot. In the continuous case, the simulated distribution uses the `density()` function to draw a kernel density plot reflecting the x-coordinates of the sampled points; in the discrete case, a line graph is used instead. Note that since we have finitely many points, it is possible to plot the density of them all; since the tilting does not affect the x-coordinate, this simulated line is not affected by tilting.

Although the moment generating functions are hardcoded, the cumulative generating function and its derivatives are not. This is because it is easy to find these in R from the moment generating function. A benefit of doing it this way is that the code is easily adaptable. In chapter 6, we will see how easy it is to generalise the code so that we can pass it any moment generating function.

The Deriv [37] package does symbolic differentiation of functions. Seeing as all of our moment generating functions are composed of sums, products, and compositions of standard functions, we can very easily obtain the derivatives we need using the `Deriv()` function.

In the univariate case we can simply plot $\left(K'(\mathbf{s}), \frac{\exp\{K(\mathbf{s})-\mathbf{s}K'(\mathbf{s})\}}{\sqrt{2\pi K''(\mathbf{s})}}\right)$ to draw the saddlepoint distribution. Note that this doesn't quite look like the formula we have seen in **??**. All the x values have been replaced by $K'(s)$ values! However, recall from 2.4.2 that we have

shown there is a relationship between x and s - namely that $x = K'(s)$. The reason why we plot $\left(K'(\mathbf{s}), \frac{\exp\{K(\mathbf{s})-\mathbf{s}K'(\mathbf{s})\}}{\sqrt{2\pi K''(\mathbf{s})}}\right)$ instead of $\left(\mathbf{x}, \frac{\exp\{K(\mathbf{s})-\mathbf{s}\mathbf{x}\}}{\sqrt{2\pi K''(\mathbf{s})}}\right)$ is that we do not need to find which x values correspond to which s values.

The vertical line in the crosshairs displays the value of $K'(s_0)$ where s_0 is the tilting factor that is selected by the user. The horizontal line corresponds to the saddlepoint approximation at that value. That is, it plots $y = \frac{\exp\{K(s_0)-s_0K'(s_0)\}}{\sqrt{2\pi K''(s_0)}}$.

Inputs

There are a few widgets that the user has access to that can change the appearance of the graph. In the original application, the x and y axes on the scatterplot were controlled by sliders. This remains true for this application. As I discuss in more detail in section 5.2.13, the saddlepoint approximation plot will be constrained to share the x axis of the scatterplot to make it easy to compare the two images.

Other widgets the user can change are widgets which:

- Specify the minimum and maximum values of the y axis
- Specify whether the y axis of the graph is on a log scale
- Turn each of the three lines on and off independently
- Turn on and off crosshairs

Here are what the controls for the plot look like:

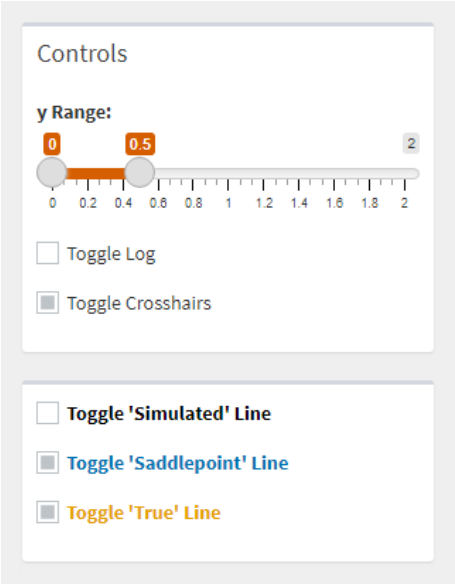


Figure 5.11: Controls for the Saddlepoint Approximation Plot

I talk about the placement of these controls in section 5.2.13.

5.2.6 The Tilted Density Plot

I also wanted to include a plot that would show the tilted distribution and a comparison to its normal approximation.

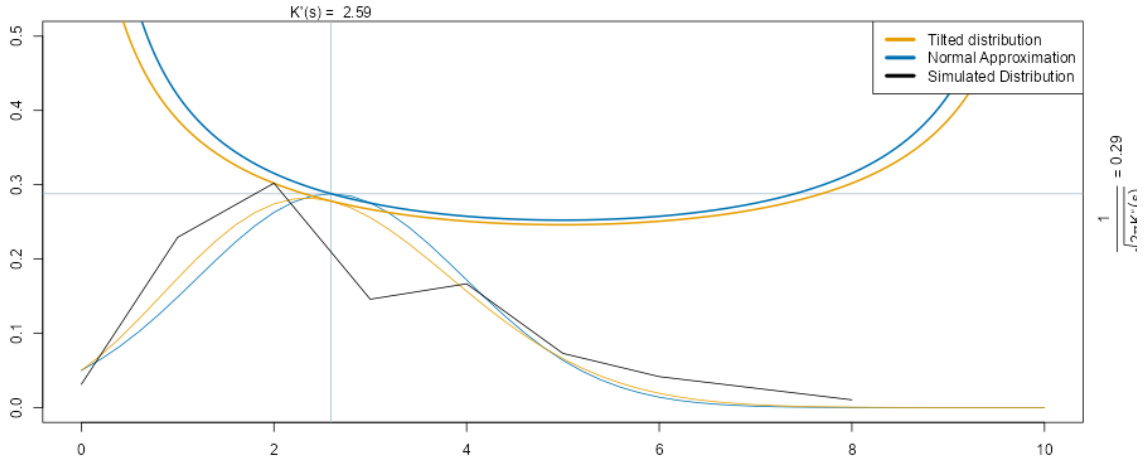


Figure 5.12: The Binomial(0.5,10) distribution at $s = -1$; the true tilted, the normal approximation to the tilted, and simulated densities

In the figure above, the thin orange and blue lines correspond to the tilted density, and the normal approximation to said tilted density at the particular value of s that is selected. The thick orange and blue lines trace out the value of the corresponding thin lines for every value of s. This means that, as we change the value of s using the slider, the thin lines will update, but the thick lines will not. It is worth noting that the thick orange/blue lines correspond to the orange/blue lines from the saddlepoint approximation plot, just on the tilted scale.

The black line shows the density plot for all of the *visible* points in the scatterplot. The crosshairs point to the mode of the normal distribution; this is the point that, when untilted, gives the saddlepoint approximation for that value of s.

Although the plot does look really busy, this is because it is not designed for all the lines to be visible at once - each line will be able to be toggled on and off. With the crosshairs and simulated distribution toggled off, the plot is more readable:

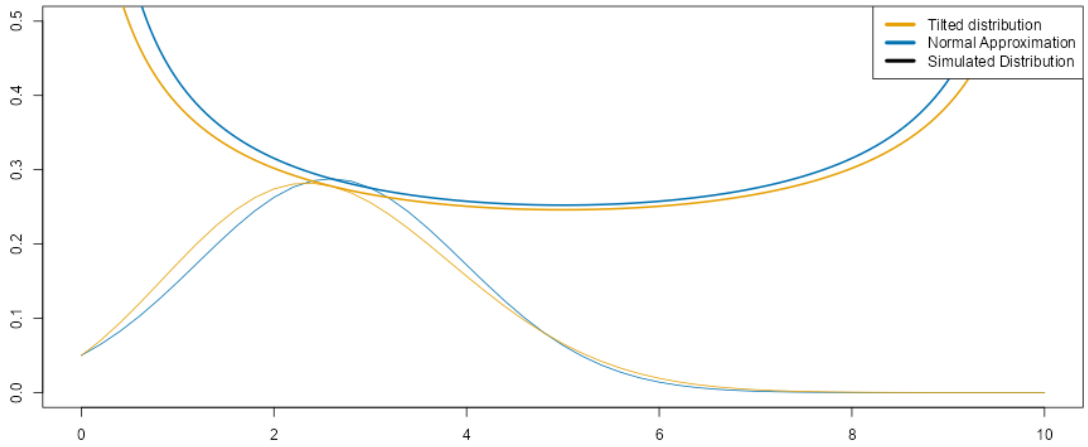


Figure 5.13: The Binomial(0.5,10) distribution at $s = -1$; the crosshairs and simulated density are turned off

I discuss the colours used in this plot in 5.2.10.

Calculations

We want to plot each of the lines for a range of x and s values. Let's denote the vector of x-values that we wish to plot over by $\mathbf{x}$, and the range of s values by $\mathbf{s}$. The tilting factor selected by the user is denoted s_0. $dnorm(a, \mu, \sigma)$ is the value of the normal density with mean μ and standard deviation σ at a. $K(s), K'(s)$, and $K''(s)$ denote the cumulative generating function and its first and second derivatives respectively, and $f(x)$ is the value of the density function at x.

Then the coordinates of all the points that we are interested in plotting for our four lines are given by:

- *The tilted density (at $s = s_0$:*
  $(\mathbf{x},\ \ f(\mathbf{x}) \exp\{s_0\mathbf{x} - K(s_0)\})$

  This is the definition of the s_0-tilted density. We wish to plot this for all x values in $\mathbf{x}$. We have seen this defined in 2.2.

- *The mean tilted density (for all s):*
  $(K'(\mathbf{s}),\ \ f(K'(\mathbf{s})) \exp\{\mathbf{s}K'(\mathbf{s}) - K(\mathbf{s})\})$

  This is equivalent to $(\mathbf{x},\ \ f(\mathbf{x}) \exp\{\mathbf{s}\mathbf{x}) - K(\mathbf{s})\})$. We replace $\mathbf{x}$ with $K'(\mathbf{s})$; recall from 2.4.2, that they are equivalent.

  When it comes to the coding, however, we have separate vectors for $\mathbf{s}$ and for $\mathbf{x}$. It is not the case that the i^{th} entry of $\mathbf{s}$ is the same as the derivative of the CGF of the i^{th} entry of $\mathbf{x}$. Rather, both $\mathbf{s}$ and $\mathbf{x}$ are linear sequences, are not linked, and usually have different lengths.

To find **xs**, we do require two vectors of the same length are required to be. It is simpler to do all the calculations in terms of **s** instead; $K''(\mathbf{s}), K'(\mathbf{s}), K(\mathbf{s})$ and **s** are necessarily the same length.

We get $(\mathbf{x},\ f(\mathbf{x})\exp\{\mathbf{sx}) - K(\mathbf{s})\})$ from evaluating the variable tilted density at every value of **s**.

- *The normal approximation to the tilted density (at $s = s_0$):*
  $(\mathbf{x},\ dnorm(\mathbf{x}, K'(s_0), \sqrt{K''(s_0)}))$

  We have seen in 2.4.2 that the s_0-tilted density has mean $K'(s_0)$ and variance $K''(s_0)$.

- *The mean of the normal approximation (for all s):*
  $(K'(\mathbf{s}),\ 1/\sqrt{2\pi K''(\mathbf{s})})$

  The $1/\sqrt{2\pi K''(\mathbf{s})}$ term comes from the fact that $1/\sqrt{2\pi\sigma^2}$ is the height of a normal distribution with standard deviation σ. Since this is calculated in terms of **s**, we can use $K'(\mathbf{s})$ instead of **x** as our x-coordinates.

Sampling Mechanism for the Simulated Distribution

Recall from section 2.6 that when we tilt the points in the scatterplot their x-coordinate does not change. Therefore if we plot a density of the x-coordinates of *all* the points generated, it will be the same as the untilted sampling distribution. We have seen this in the saddlepoint approximation plot in section 5.2.5.

However, if we sample only the points below a certain y-value, the distribution of the x-values should, as long as enough points are generated (see section 2.6.2), follow the tilted distribution. Here, the particular y-value is just the maximum y-value on the scatterplot. This makes it easy to see when enough points are plotted for the simulated density to reflect the tilted density since the we can see the red line which indicates the highest y-value that the simulated points can have.

In the figure below, the simulated line and the true line match closely.

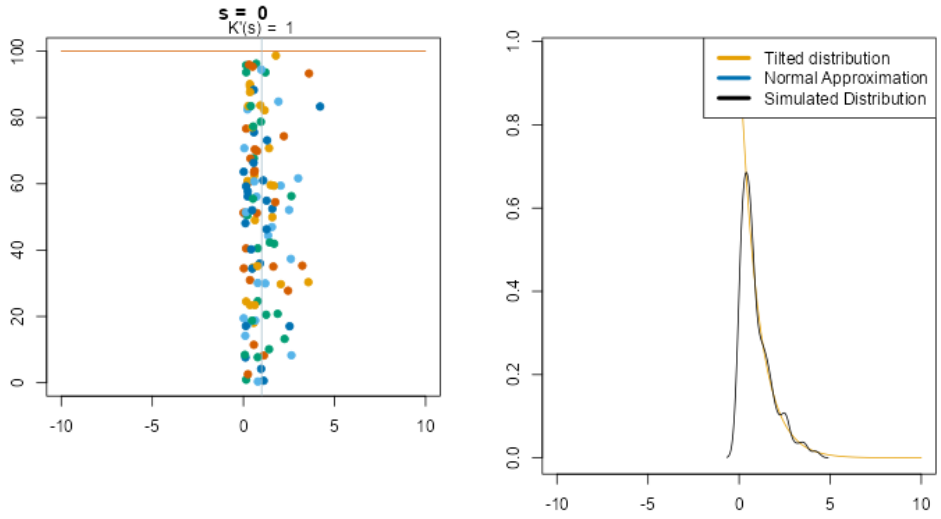


Figure 5.14: The simulated density to the Gamma distribution looks fairly close when $s = 0$

Whereas, in the following image, we do not have enough points for the simulated line to be accurate.

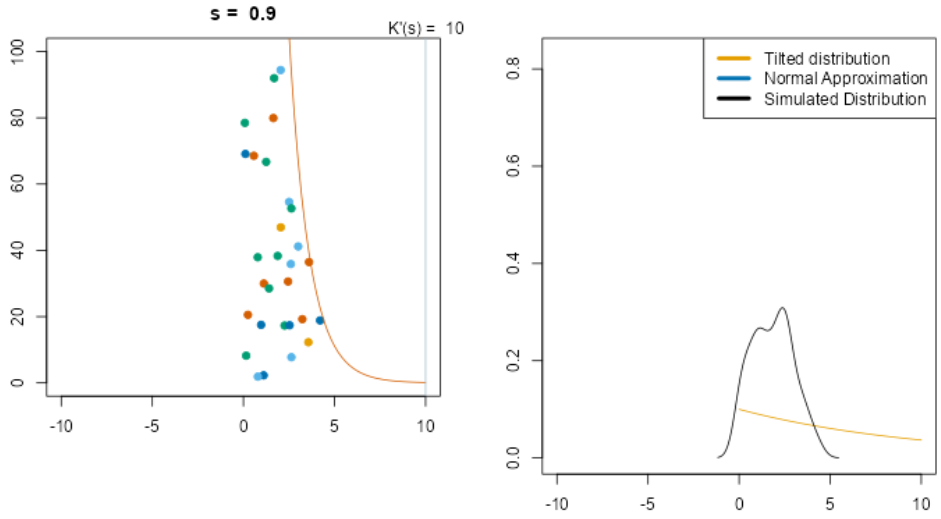


Figure 5.15: The simulated density to the Gamma distribution looks very far off when $s = 0.9$

Inputs

The tilted density plot is on a tab below the scatterplot. Either this plot or the saddlepoint approximation plot can be seen at one time (but not both). I talk more about positioning in section 5.2.13.

The inputs for this plot are the same as for the saddlepoint approximation plot - we can change the x and y axis limits, toggle the y axis to be on the log scale, toggle on and off crosshairs, and toggle each line independently.

The x axis is constrained to be the same as both other plots. The y-limit, log scale, and crosshairs are the same as the saddlepoint plot - the box on the top remains unchanged when we switch between the two tabs but the lower box swaps out.

The controls for this plot can be seen on the figure to the right

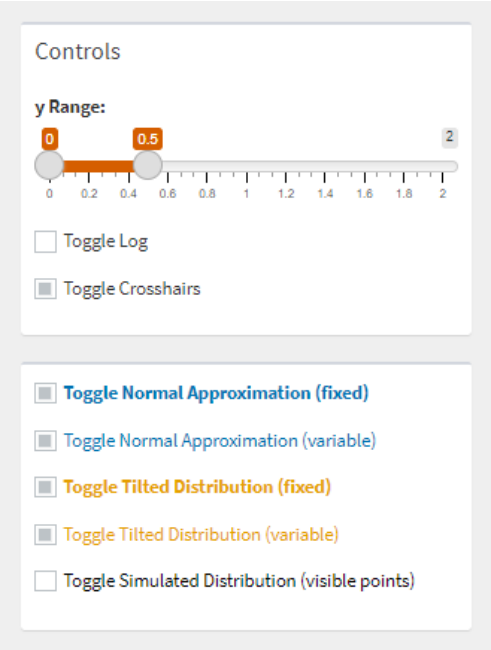


Figure 5.16: Controls for the Tilted Density Plot

5.2.7 Changing $K'(s)$ vs changing s

The application currently has one widget which allows the user to input some value for s. In practice, however, we are usually interested in finding an approximation to the probability function at one specific value of x rather than a value of s. With our current setup, we have to try all values of s until it gives us the x we are interested in.

This indicates that should be two modes; one for allowing the user to input the s value, and one for allowing the user to input the $x = K'(s)$ value. This is relatively easy to implement with use of conditional panels - these are widgets which can appear/disappear reactively. We can have one toggle which allows the user to specify which "mode" they would like to be in, and the widgets update reactively. Here is what it looks like:

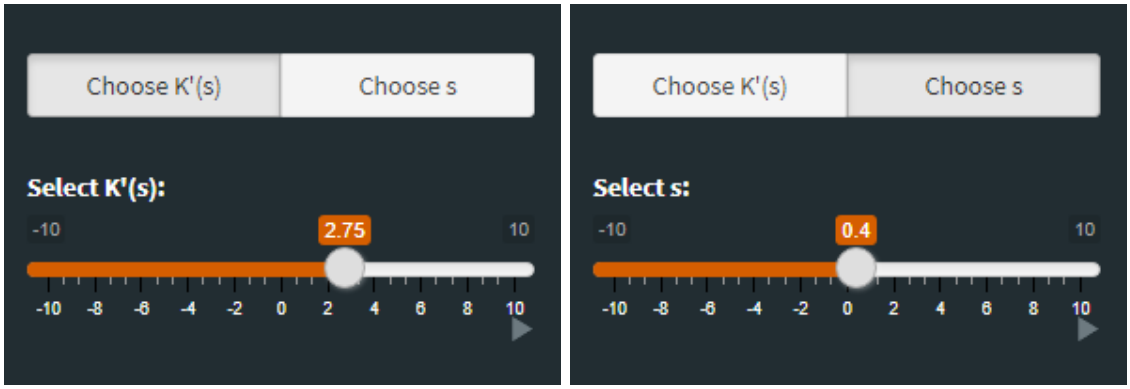


Figure 5.17: Two modes of input; selecting $K'(s)$ (left) and selecting s (right)

The toggle at the top swaps out one widget for another - it is not just the title that is changed, but the slider itself. Each slider has it's own minimum/maximum values, and a value that is completely independent from the other.

A Need to Constrain The Widgets

Currently, our two sliders have no influence on each other. This is not ideal; we would prefer for them to be constrained such that changing one also changes the other to match it. In this section, I use the word "match" in reference to the $K'(s)$ and s values to mean the following:
Say our s slider has a value of a. We say that the $K'(s)$ slider matches the s slider if and only if it has the value $K'(a)$. Similarly, if the $K'(s)$ slider has a value of a then we say that the s slider matches it if and only if it has a value b such that $K'(b) = a$. It is obvious that if our s slider matches the $K'(s)$ slider then the $K'(s)$ matches the s slider and vice versa.

It is important that our sliders match, so that our application can seamlessly switch between the two modes. As it is currently, when we switch modes, all the points in our scatterplot suddenly shift to represent the new values of s or $K'(s)$. If the sliders match then our plots will not change when the user changes modes.
As it turns out, ensuring that our sliders always match is very much non-trivial to implement.

Implementation of the Constraint

`observeEvent()` is a function in R Shiny which executes lines of code when a particular "event" is observed. These events can be any reactive expression, but is usually a change in some input value.

We are interested in using `observeEvent()` to update the value of a slider. In particular, the idea is that when we update the $K'(s)$ slider, it will automatically update the s slider to match, and vice versa.

When we start with matching values of $K'(s)$ and s, this approach works perfectly. However, if the initial values do not match, we end up in an infinite cycle. Both of our sliders

are switching between two states – one to match the value of each slider.
For example, if the sliders are initially set to $s = a$ and $K'(s) = b$ then the application will switch constantly between the $(s, K'(s)) = (a, K'(a))$ and the $(s, K'(s)) = (c \text{ such that } K'(c) = b, b)$ states. Since our sliders don't match, these states are not the same.

This leads us to believe that all we then require is to find initial values for $K'(s)$ and s so that they match. This is not easy since $K'(s)$ and s have a relationship that is determined not only by the distribution itself, but also by the values of the parameters. Even if we come up with initial values for $K'(s)$ and s, if the parameters are changed, then we also end up with an endless loop, since there is an instant where they don't match.

The solution is to make it such that the $K'(s)$ value only updates when we switch to the mode where we input $K'(s)$ and vice versa.

Another Problem Raised, and The Final Fix

Now whilst this does fix the issue, it also raises another one. When we change parameters, the sliders are reset. This is an issue for when we wish to fix $K'(s)/s$ and animate a change in a parameter.

The $K'(s)$ and s sliders are resetting when the parameters change. The reason for this is that the minimum and maximum values of these sliders depend on the parameters. When the parameters change, the slider will reactively update the minimum and maximum, and in doing so, the sliders will be reset to their default.

The fix for this involves exploiting the order that R Shiny carries out commands. Essentially, we can use `observeEvent()` functions to update the value of the slider that is visible to be *equal to itself* whenever either of the parameters change. The code for this looks like:

```
observeEvent(list(input$var1, input$var2),
             if(input$switch=="false"){updateNumericInput(session, inputId = "s", value = input$s)})

observeEvent(list(input$var1, input$var2),
             if(input$switch=="true"){updateNumericInput(session, inputId = "x", value = input$x)})
```

Figure 5.18: Code

This means that when the parameters are changed, instead of resetting back to their default values, they remain as how they were before the parameter change. Now at last our two sliders will always match.

Finding the Inverse of $K'(s)$

We have seen in theory about changing $K'(s)$ vs changing x, however this change requires a bit of additional coding. When we switch from s mode to $K'(s)$ mode, knowing what value to change to is easy, since we know the derivative of the cumulative generating function. However, in switching the other direction, we require the inverse function of $K'(s)$.

There are a few ways to find the inverse of $K'(s)$. We can use `optimize()` to find the value of s which minimises $K(s) - sx$ (where x is the inputted value of $K'(s)$), or that minimises $K'(s) - x$. Alternatively, we can use the `which.min()` function to do the same, only using discrete values of s. It turns out that `optimise()` is not any slower (but is more accurate) and there is little difference between the two functions to minimise.

Our application has a few calculations - namely the ones for the tilted/normal distributions that change according to s - that rely on the inputted value of s. These will have to be adapted to instead take the inputted value of $K'(s)$ instead. This is because, recall, the inputted value of s does not update as we change $K'(s)$ on the slider.

This is an easy fix, however, since we can simply replace all instances of s with our 'inverse-of-k-prime' function of the selected value of $K'(s)$.

5.2.8 Choosing Minima/Maxima of Sliders

As part of our reactive statement that updates with the distribution (see section 5.2.4), we have some hardcoded restrictions on the size of s and x for each distribution.

One of these is the maximum values of s that can be taken - `max_s`. This is a function of the parameters due to the fact that the moment generating functions of some distributions have upper bounds for the values that s can take. For example, the moment generating function of the exponential distribution, $M(s) = \left(1 - s\lambda^{-1}\right)^{-1}$ is defined only for $s < \lambda$.

We also have the minimum values that x can take - `min_x`. This is because, for lots of distributions, the probability function is defined for all real numbers. However, the support of the distribution is not the entire real line.

In the image below, we can see what the plot of the Poisson(2.6) distribution looks like if we do not restrict the x range.

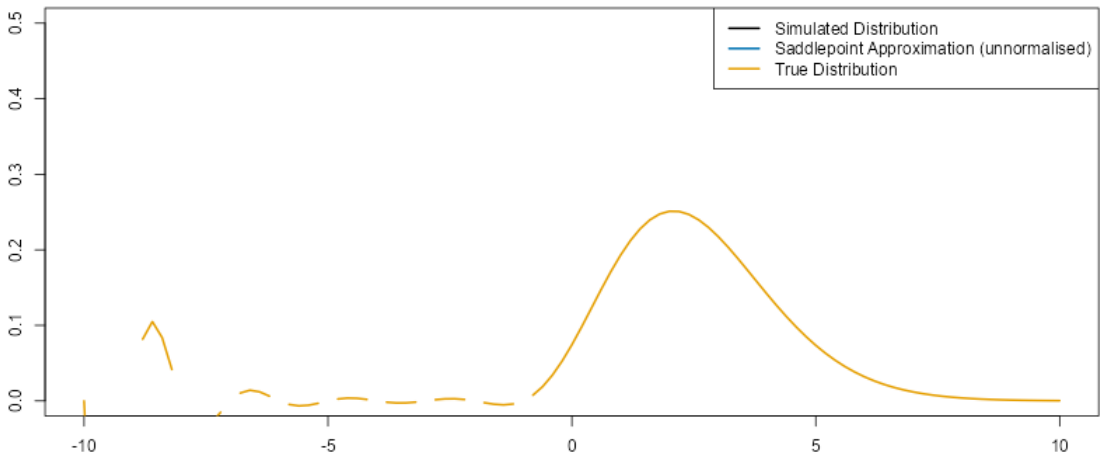


Figure 5.19: The Poisson(2.6) distribution; the curve looks continuous for $x > -1$ but is broken up when $x < -1$

This issue is fixed when we ensure that the distribution is zero for any x values below, in this case, negative one.

Relation to Sliders

In the previous section, 5.2.7, I alluded to there being a relationship between the minimum and maximum allowed values of the sliders which decide s or $K'(s)$, and the parameter values. Although we might have limits for the maximum value of s and minimum value of $x = K'(s)$ as described above, there is also an interaction between the two sliders, as well as some other limitations we need to take into account.

We have the following rules:

1. $K'(s)$ must be between -10 and 10.
   This is due to the fact that, at its largest, the x-axis of our plots ranges from -10 to 10. Of course, these limits are able to be changed, just not by the user.
2. $K'(s)$ cannot be bigger than `min_x`
3. s cannot be smaller than `min_s`
4. We can't have a value of $K'(s)$ which would mean the matching value of s is not allowed, and vice versa

In practice, we also end up with another rule, due to us having to disallow infinite values of s.

5. s must be between -10 and 10

Although we could allow s to have more magnitude, doing so often means that a big change in s only reflects a minute change in x. The "useful" s values are almost always between -10 and 10.

In pseudocode, we end up with:

```
minimum_s = max(k_prime_inverse(min_x), k_prime_inverse(-10), -10)
maximum_s = min(max_s, k_prime_inverse(10), 10)
minimum_x = max(min_x, k_prime(-10), -10)
maximum_x = min(k_prime(max_s), k_prime(10), 10)
```

Where `k_prime` is the function $K'(s)$ and `k_prime_inverse` is its inverse. Note that both of these are dependent on the parameters of our distribution, which is why we had issues with constraining our sliders in section 5.2.7.

5.2.9 Sidebars and shinydashboard

There are a few packages in R which create a more visually pleasing user interface than the default one. Although it is possible to directly edit the CSS, these packages provide an easy way to create an appealing application without having to get into the nitty-gritty of the source code. The one I will be using for this project is `shinydashboard` [38]. The following two figures illustrate the difference between the two.

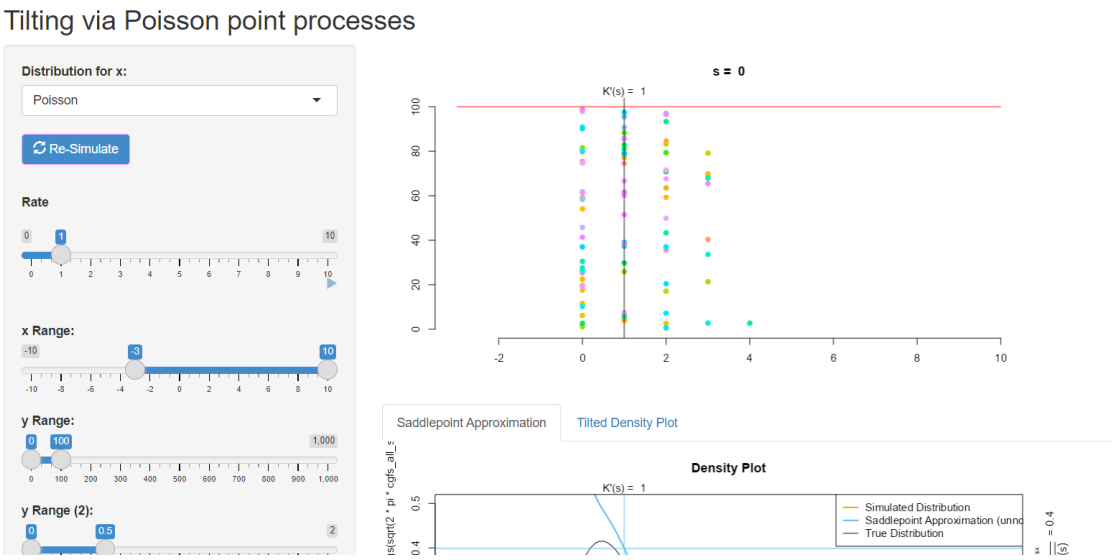


Figure 5.20: An application using the default version of R Shiny only

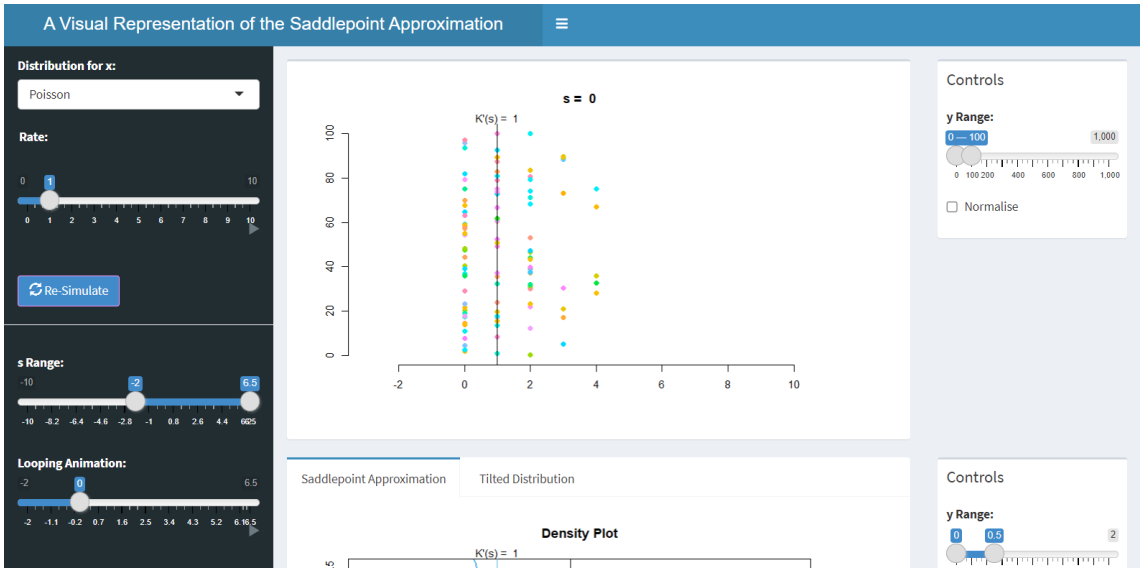


Figure 5.21: An application using the default version of shinydashboard

There are three main things that shinydashboard does for us:

- It allows for a collapsible sidebar. This means that the plot section can be expanded to fill the entire screen; this is not a feature which the default shiny interface allows.

- It also allows boxes (such as the white rectangle labelled "Controls" in figure 5.21) which can contain plots, text, or widgets. These have good contrast against the background (one of our "big four" principles of graphic design; see section 3.3), and ensure that the UI elements are not floating in space.

- Generally, `shinydashboard` has a more polished appearance. It's worth noting that, while it is possible to customise the appearance of the original Shiny application (such as changing the fonts or colours) using CSS and HTML, it would require a lot of additional work. We can see that the principles of repetition (in the slider colours matching the navigation bar) and contrast (in having large areas of dark and light colours) have been thought about for us. This gives the shinydashboard app in figure 5.21 a more visually appealing and cohesive look compared to the default app in figure 5.20.

Now, originally, the application had every widget in one section, in a grey panel on the left hand side. This is bad user interface design. The user cannot know intuitively which widget affects which plot. Despite each widget being labelled, we should aim to make the understanding of the influence of each widget as obvious to the user as possible. Secondly, the page is really quite long. This means that if the user changes a widget at the bottom of the page, which updates a plot at the top of the page reactively, they will have to scroll all the way up to see the effects. Particularly with sliders with arrows, (see section 5.2.3), there is no way to see the value of the slider and the corresponding plot simultaneously.

With this "box" idea in shinydashboard, we can change the layout slightly so that the widgets that are related to particular graphs lie next to them, utilising the design guideline of proximity. I discuss this in more detail in section 5.2.13.

5.2.10 Colours of Graphs

We have seen in section 3.4.1 some guidelines to choosing qualitative colours. In this section, I will discuss the choices I made in selecting colours for the graphs, and the reasons for such, particularly with regard to Okabe and Ito's work.

Scatterplot

There were two reasons to add colours to the scatterplot: firstly to distinguish the points from one another, and secondly to create a more aesthetically pleasing image.

Distinguishing the points from one another is required so that we can see the trajectory of each point when the parameters of the distribution, or the value of s, is changed. The colours are not used to represent different groups, so the number of colours to choose is flexible. Choosing too few colours means that multiple close-by points would be coloured the same and would therefore be hard to distinguish; too many colours would not only be aesthetically unpleasing, but many points would be so close in hue that they too would be hard to tell apart.

We have seen how the Okabe-Ito palette is a good choice for people with colour blindness, so these colours will be used throughout the application. For this scatterplot, selecting five of the eight colours in the palette was a good balance between too many and too few colours; these chosen colours are shown below.

Figure 5.22: A Subset of the Okabe-Ito Palette

These five colours all have high contrast with one another by design. In order to look good together, green-blue and red-orange colours were selected - these are complementary colours, which have been found to be "preferred" colour combinations [39].

Although it isn't easy to see from static images, the colours do make it much easier to see where each point moves to when they are animated. For comparison, see the difference between the following plot and figure 5.4 where there are no colours.

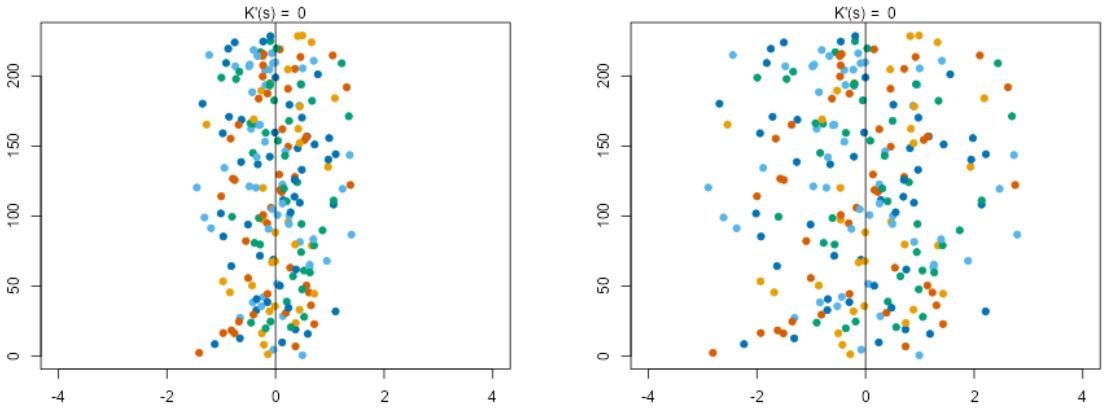


Figure 5.23: Points with the x-axis sampled from the Normal distribution with mean 0 and standard deviation 0.6 (left) or 1.2 (right). The structure of the points has been retained.

The colours selected should be distinguishable to colourblind people. However, Okabe and Ito also stress the importance of redundant coding, so there is also an option for the points to be distinguished by shape. This is illustrated in the figure below.

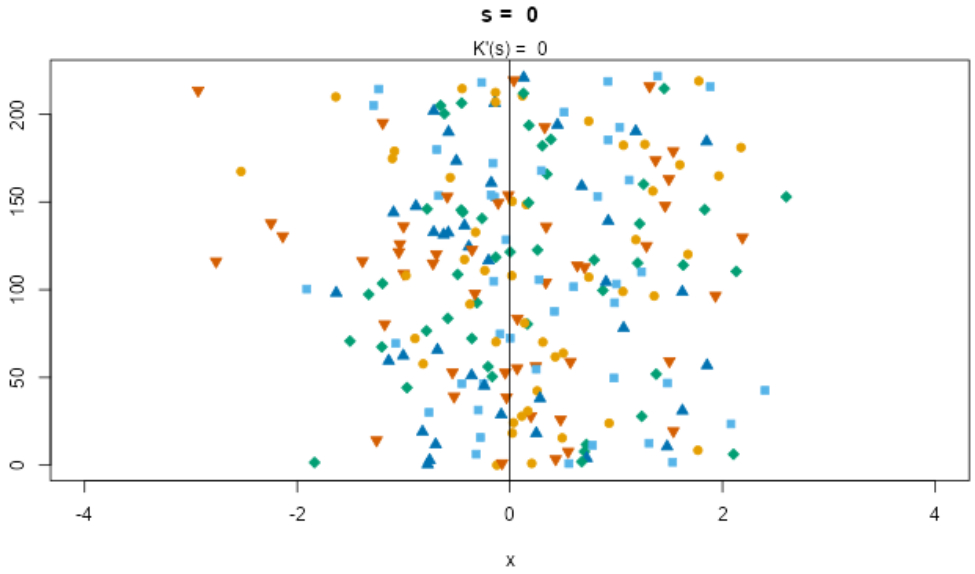


Figure 5.24: A scatterplot when "colourblind mode" has been turned on

Although we have seen in section 3.2 that shape is not as effective of a visual channel as colour, the points are also related by motion. So there are three different visual channels at work, meaning there is sufficient redundant coding.

Saddlepoint Approximation Plot

As per Okabe and Ito's guidelines, the three colours for this plot should have a mixture of hue (i.e. warm and cool tones) and value (i.e. lightness/darkness). I decided to go for a light orange, a black, and a dark blue. The figure below shows what the colour palette looks like to people with the most common forms of colourblindness.

Figure 5.25: Chosen colours for the saddlepoint approximation plot, as viewed by a person with (from left to right) normal vision, deutanomaly, protanomaly, tritanomaly. The final column on the right is in greyscale.

The `colorspace` [40] package in R was used to create this image, which allows us to view colours through the lens *weird metaphor* of people who are colourblind. We can see that in each column, the colours are very distinguishable from one another. Also, the final column shows that the colours have very different values.

To add redundant coding, I also allowed an option ("colourblind mode") which, when toggled on, makes each line types unique.

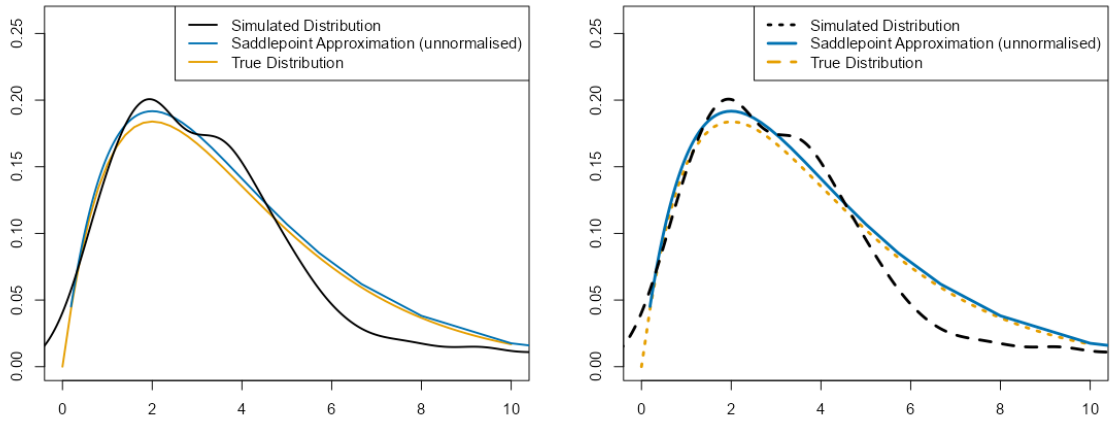


Figure 5.26: The saddlepoint approximation plot with colourblind mode off (left) and on (right)

Note that here I have all three lines toggled "on" which does mean that they aren't all

that easy to see since there is lots of overlap. However, recall that there is the option to toggle the lines on and off; see section 5.2.5 for more details.

This colourblind mode also means that when used in a presentation setting, the presenter can describe the lines using non-colour specific language. For example, the phrase "dotted line" can be used instead of "orange line". Recall from section 3.4.1 that larger areas means that colours are more distinguishable; colourblind mode also makes the line width thicker.

I also had to decide the colour for the crosshairs. Since they pinpoint the height of the saddlepoint approximation line for the selected value of s or $K'(s)$, we want them to be associated with the dark blue line. However, we also don't want them to be as dark/bright as this line, since they are relatively less important; we want them to have less contrast.

The crosshairs were therefore chosen to be coloured in a muted dark blue. The final appearance, with all the other lines toggled off, looks like:

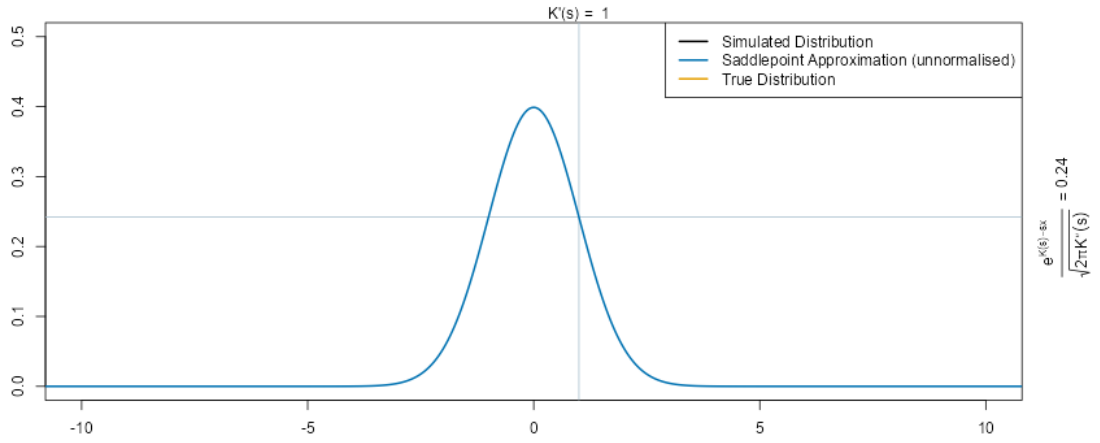


Figure 5.27: A normal density plot; the crosshairs are coloured a muted grey-blue colour

Tilted Distribution Plot

Recall that the tilted density plot shows similar lines to the saddlepoint approximation plot, just on another scale. In particular, the tilted distribution is equivalent to the true distribution line and the normal approximation is equivalent to the saddlepoint approximation line.

Recall from section 3.1 that one of the principals in Gestalt theory is that of similarity; if elements of the design are similar in their size, shape, or colour, we view them as inherently connected [21]. In order to draw the connections between the two graphs, keeping in mind that both are not on the screen simultaneously, colour is a good way of linking the two graphs together. This will use the Gestalt rule of similarity to allow the user to draw a connection themselves without it explicitly being labelled.

This means that choice for colours was to be dark blue for the normal approximation

lines, and orange for the tilted distribution lines. Here is what the plot looks like when colourblind mode is turned on and off.

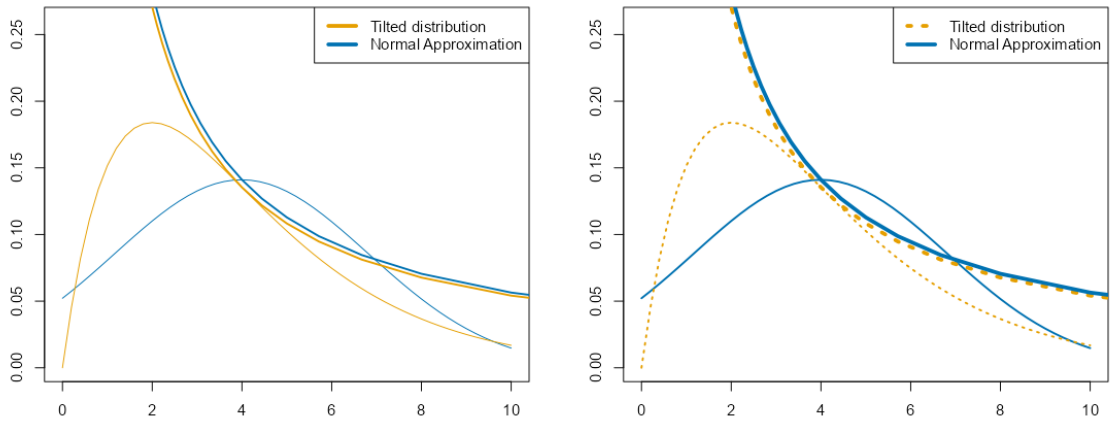


Figure 5.28: The tilted density plot with colourblind mode off (left) and on (right)

Again, each of the lines is individually able to be toggled, so the plot is able to look less busy.

Now, note that the 'variable' lines (i.e. the ones that change with s) have thin line weights and the fixed lines have a thicker line weight. This difference was again to unite the two types of lines using the Gestalt rule of similarity. I felt that the rigidness of the fixed lines corresponded better to a thicker line weight which is the reason it is this way around.

5.2.11 Changing the CSS

We have seen in section 5.2.9 that `shiny` and `shinydashboard` do not have built-in methods to change small details, since they are written to be relatively high-level. For this, it is required to manually alter the theme using custom CSS, which we introduced in section 4.2.

There were a few things that it was important to change using CSS. The first is the colour of the navigation bar at the top.

Main Colour Choice

CSS can be used to change the colour of the navigation bar as well as the sliders. I felt that these should be the same colour, to add repetition (one of our CRAP guidelines 3.3) and cohesiveness to the application. The colour chosen really could have been any of the colours from the Okabe-Ito palette since this is what is used throughout the application. Each of the colours was bright enough to contribute some good contrast against the neutral coloured background, and also look visually pleasing. For this application, I went with the red-orange colour.

The code to do this looks like the following:

```
#defines nav-color
tags$style(HTML(glue(
  ":root {--nav-color: @{'#E69F00'}@}",
  .open = "@{", .close = "}@"
))),

#changing colour of active tab indicator
tags$style(".nav-tabs-custom .nav-tabs li.active {border-top-color: var(--nav-color);}"),

#changing colour of navbar
tags$head(tags$style(HTML('
  .skin-blue .main-header .logo {background-color: var(--nav-color);}
  .skin-blue .main-header .logo:hover {background-color: var(--nav-color);}
  .skin-blue .main-header .navbar .sidebar-toggle:hover{background-color: var(--nav-color);}
  .skin-blue .main-header .navbar {background-color: var(--nav-color);}')))),
```

Figure 5.29: CSS code inline to change the colour of the navigation bar

Note how this is written in CSS, but the `htmltools`[41] package in R allows us to create R objects to represent each of the CSS tags we want to create. In particular, the 'style' tag specifies how that object renders - in this case, its colour. This code is placed in the UI section of our shiny app. Changing the colours of the sliders works in a similar way.

Font Faces and Colours

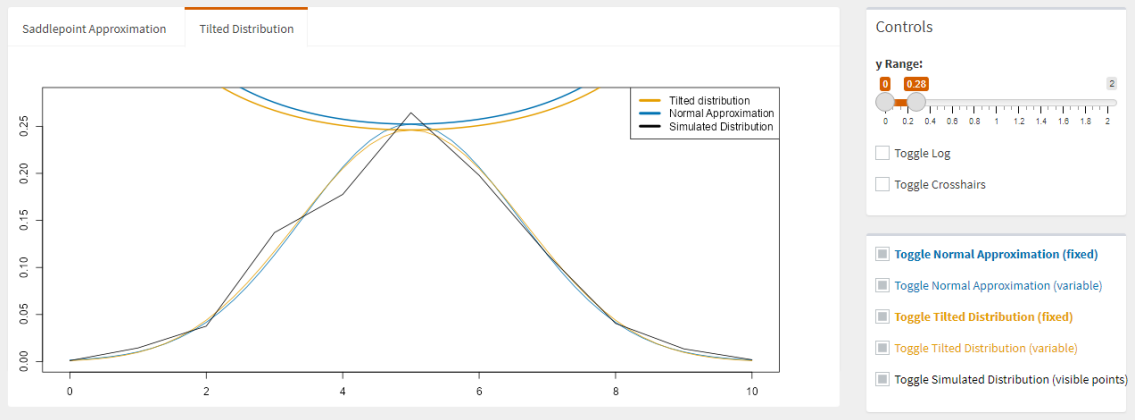


Figure 5.30: The tilted distribution plot; the colours of the toggle labels match their corresponding line

If you note from the image above, we have five different lines, and five different toggles, each corresponding to one of the lines.

The Gestalt rule of similarity has been discussed in detail in section 3.1, as well as the previous section 5.2.10. In order to utilise this, we can use custom CSS code in order to change the font of each of the toggles.

The thin and thick lines will correspond to non-bold and bold font faces respectively. Similarly, the orange and blue lines will correspond to orange and blue font colours.

This will mean that the user has an intuitive understanding of which toggle corresponds to

which line, without having to read the text. This change was made for both the saddlepoint approximation and the tilted density plots.

The `prettyCheckbox()` function, which I am using to create the toggles, doesn't have an easy way to change the colour of the label. Therefore, this had to be done using CSS. The way this works is that we have to create containers around each of our toggles using 'div' tags. Each container is able to be assigned to a particular class. In this case, we want both the 'tilted distribution' toggles to be in the same class, since we want them to be the same colour. We can then change the colour of text per class by using 'style' tags.

Sidebar Toggle Button

In `shinydashboard` applications, there is a button on the navbar which opens and closes the sidebar. By default, it is represented by a 'hamburger' (or 'tribar') symbol. However, this symbol does not do a good job of looking like a clickable button. This is partially because the button had to be moved rightwards in order to accommodate the long title, so it no longer lies just above the edge of the sidebar. It is also because the sidebar doesn't resemble a list like the hamburger button suggests.

The icons I was looking at to replace this icon come from `https://fontawesome.com/`. A few examples can be seen below.

Figure 5.31: A variety of icons from fontawesome

Although most of the options in the figure above look like reasonable options, most of them are unreadable in context for the same reason the original icon was an issue; they don't read as buttons to press. The only one that I thought worked was the left-pointing arrow. This is because the arrow indicates some kind of movement where the other icons appear static.

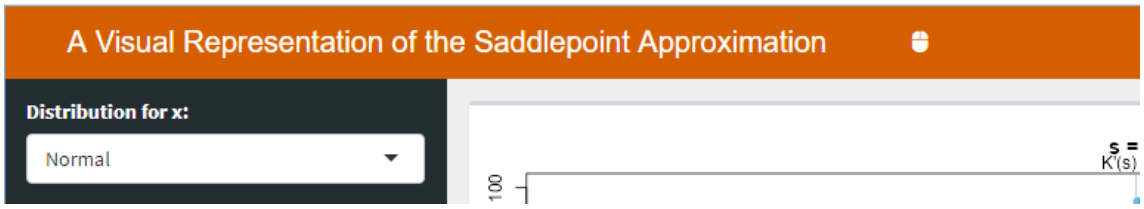


Figure 5.32: The sidebar toggle button labelled with a mouse icon; this looks static and doesn't immediately appear to be clickable

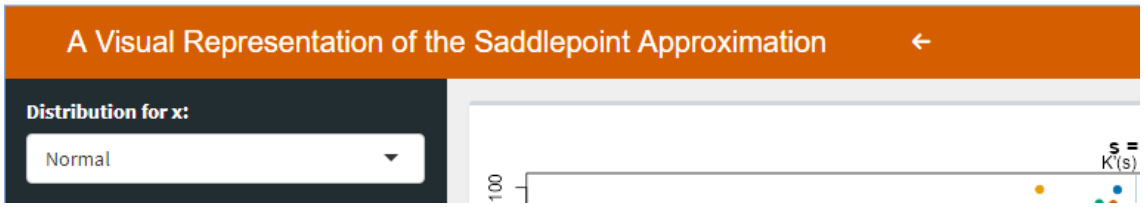


Figure 5.33: The sidebar toggle button labelled with an arrow icon; this indicates movement

It was also possible using CSS to change the direction of the arrow for when the sidebar is collapsed. The direction of the arrow now indicates the direction that the sidebar will move in.

The way that this is done is by counting how many times the sidebar toggle has been clicked. If it is even, we edit the style tags of the button so that the 'content' is a left arrow; if it is odd then then content should be a right arrow.

5.2.12 Choice of Widgets

We have mentioned briefly in section 4.1 that the same input can be entered into Shiny via several different widgets. In this section, I will briefly discuss my choice for which widget I used in which place

Distribution Input

There aren't many options for widget choice when the user is asked to select one of several options, each of which is a word/string. The Shiny Widget Gallery [42] gives examples of radio buttons and select boxes, as you can see below.

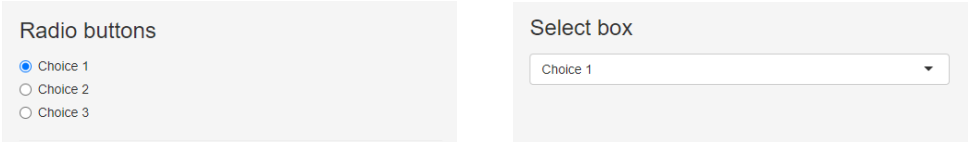


Figure 5.34: Radio Buttons (left) vs Selection Box (right) for string inputs

For the distribution input, the select box works much better. This is because it saves on

space. Not only does the user not need to see all the options all the time, but also there are a lot of them!

Numeric Inputs

Slider widgets are a good choice when we are interested in relative quantities [43]. Recall that we are interested in seeing how a change in parameters or tilting factors influences all three plots. The absolute value of the parameter is not primarily of interest, rather how it changes. And so, sliders are used rather than numeric input widgets throughout the application.

We have seen in section 5.2.3 that we are interested in having a play button which will increase the input value every so often. There are no animation options available for numeric inputs, which is another reason to go for the slider. Also, the slider option denotes exactly what the minimum/maximum available values are; the numeric input does not. This is not only useful for seeing how these change as the parameters are changed, but also it allows us to place the current value in context.

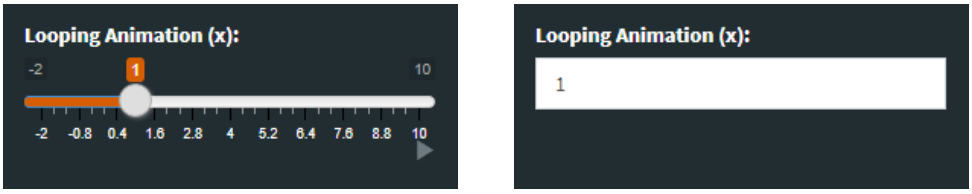


Figure 5.35: Slider with animation (left) vs numeric input (right)

The figure above compares sliders to numeric inputs; the sliders are more visually appealing in my opinion.

Toggles

There are also a number of options for widgets where the input should be one of two options - on or off. We can see a number of these available in the `shinywidgets` package for R [44][45].

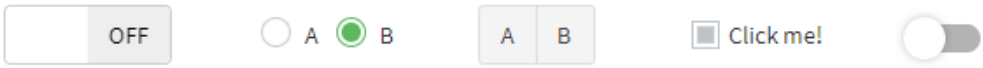


Figure 5.36: A range of widgets with two input choices

The first three designs are useful for when both options are required to be labelled. For example, if the user has to choose between "green" and "blue", these would be used. The latter two designs have one label for the whole widget, so are preferred when they represent two opposite values (e.g. on/off, or true/false).

For most of the toggles in our application (toggling colourblind mode on and off, or showing/hiding lines on graphs), we require only one label. It would make much more sense to have a switch labelled "toggle colourblind mode"; the user will understand that the "true" or "on" state of the switch corresponds to the mode being active, and the "false"/"off" state corresponds to it being inactive. It would be excessive to have a switch where one input is labelled "colourblind mode on" and one labelled "colourblind mode off".

So for all the on/off toggles, the fourth option in figure 5.36 is to be used. We want there to be consistency across the application in order to increase the repetition, one of our "big four" which we saw in section 3.3. The fourth option was used (rather than the fifth) since it is more minimal; we want attention to go towards the data and not the toggles.

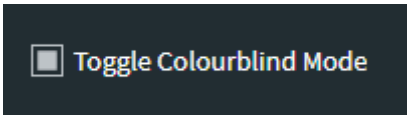


Figure 5.37: Appearance of the on/off toggle

However, there is one toggle which is not an on/off toggle; this is our switch between the application being on the s scale, or the x scale. This is arguably the most important toggle on the page, since it acts globally. And so, it should be given more prominence on the page to increase the contrast, another one of the "big four".

And so, I went for option three, as shown in the figure below. I felt that this took up the most space, and allowed us to see the labels of both options at all times.

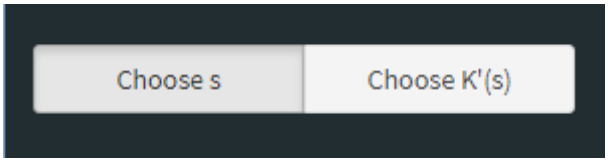


Figure 5.38: Appearance of the 'mode' toggle

5.2.13 Positioning of Elements in the UI

We already have a lot of different elements - plots, widgets, and text - but have not discussed how to arrange these elements on the page. That will be the topic of this section.

Placement of Plots and Per-Plot Widgets

Each of our three plots have the same x-axis, as is controlled by one slider.This was for a reason - we want to highlight the fact that each of these plots is on the same scale. Each of the points in the scatterplot contributes to the "simulated" line in the saddlepoint approximation graph. Similarly, if our tilting factor s is set to 0, then our tilted line and our true line should be exactly equal.

Having the same x scale highlights these similarities. We have seen that the "position on the same scale" is a highly effective visual channel, and connection is one of our design elements, and "alignment" is one of our CRAP guidelines. And so, for many reasons, it makes sense to have the plots aligned vertically. This is so that the comparisons between the x-axes of the plots is as intuitive as possible.

Recall that each plot has a few control widgets which can be used to control their appearance. The Gestalt rule of proximity tells us that these control widgets should be next to each plot. We also should group them together since they are related.

On most monitors, it is only possible to have two plots vertically on screen at any one time, otherwise the size of the elements in the plot are too small. It seems like we have three plots, each with their own controls, but only two spaces on screen for them. Placing all the plots in one column would be possible, however would require a lot of scrolling, making it hard to compare between them.

Recall that the saddlepoint approximation and the tilted density plots both share some control widgets - the y-axis slider, the log toggle, and the crosshairs. What it is possible to do is to put these plots in a tab box. This means that only one will be visible at any one time, but we can switch between them to compare.

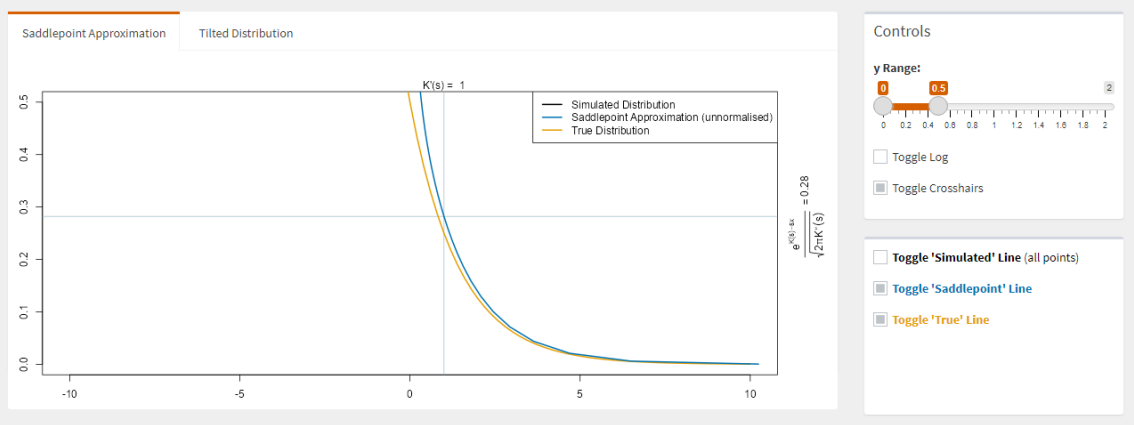


Figure 5.39: A tab box; this allows us to switch between two plots without having to scroll

In the figure above, we can see what the tab box looks like on the left. We can select either plot using the tabs at the top. The controls box on the right at the top has controls that effect both plots, whereas the box on the bottom right has controls that swap out, depending on the plot that is selected.

Global Widgets and the Sidebar

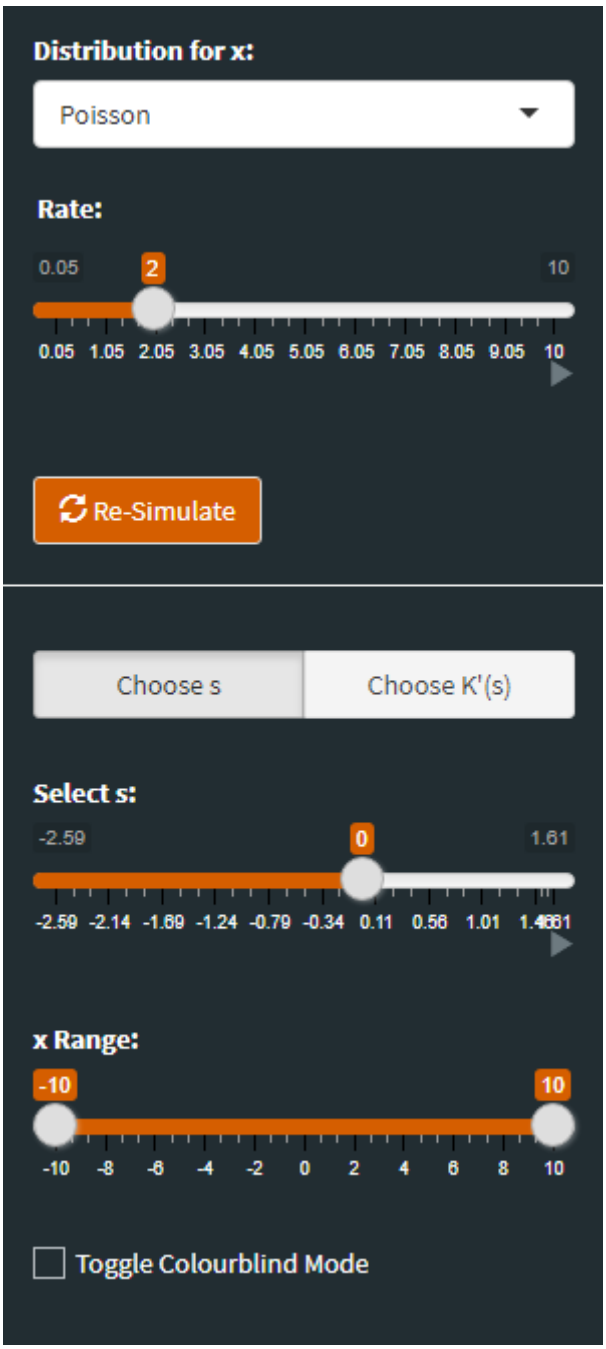


Figure 5.40: The shinydashboard sidebar

We have dealt with per-plot widgets, however there are also a few global widgets that we need to include - these contain the limits of the x axis, the tilting factor, and the parameters of the distribution among others. These are ideal for the sidebar. Since we the sidebar is minimisable, the idea is that these can be set, then the sidebar can be minimised to allow more space for the plots.

It is intuitive to the user that the sidebar is global since it is not in closer proximity to one plot than to the other.

In terms of the arrangement of widgets on the sidebar, it made sense to include the widgets in the order they would likely be filled in. For example, the user is likely to want to change the distribution first, so this is at the top. Since the parameters are related to the distribution, these should be next, followed by the re-simulate button.
A line separates the distributional inputs to the rest; then all of the other widgets are included. The only ordering that was important is that the "select s" or "select $K'(s)$" slider had to be next to the toggle which changes the mode. The toggle should be on top, since this is chosen first.
A figure showing what the sidebar looks like can be seen on the left.

Different Monitors

Different monitors are different shapes and sizes. If one is designing an application that is mostly going to be viewed on mobile devices then different considerations need to be made compared to if the app is to be viewed on an iMac. The way that `shiny` layouts (particularly `fluidRow` and `fluidColumn`) work is that they adapt to the shape of the screen the user is viewing the application from.

We can see from the image on the right that some of the boxes rearrange themselves when the application is being viewed on a mobile (or otherwise vertical) device, compared to how they look on a horizontal monitor (in figure 5.42).

The application should look okay on any device. The difference is that smaller/horizontal screens fit fewer elements on them at once. For example, the image on the left only shows half of the boxes available on the screen, whereas on a large, horizontal computer screen, all the elements are visible.

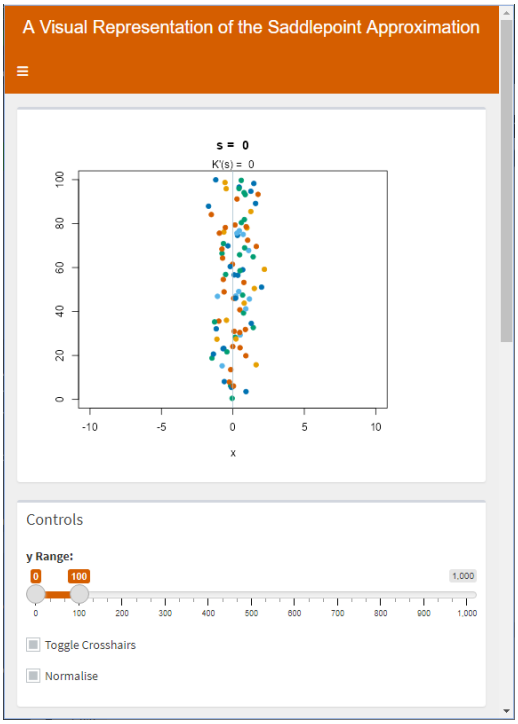


Figure 5.41: A shinydashboard, as viewed on a vertical monitor

Most computer/laptop monitors have similar ratios, so there is no need for significant testing on different monitors. It is worth noting, however, that the application was developed assuming that the end user uses a computer or laptop (as opposed to a mobile device).

5.3 The Final Application

A screenshot of the final application is included below. However, a still image can not demonstrate the full capabilities and user experience of the application. The full application - fully interactive - can be found at `https://saddlepoint.shinyapps.io/visualise/` and the code can be found at `https://github.com/alicemhankin/saddlepoint-visualisation`

Figure 5.42: The Final Application

Chapter 6

Application - The Sum of Two Known Distributions

We have discussed in section 5.2.1 that there is a trade-off between having the application being flexible and being accessible. It is impossible to allow the user to select any distribution without having them directly type in R code. For the first application, I wanted maximum accessibility, but this meant that only a small set of distributions could be used; these are one's that it is easy to find the true density, so the saddlepoint approximation is not needed for these distribution in practice.

For my second application, I wanted to allow a more flexible class of distributions - ones that do not have trivial probability functions. I decided to go with random variables that are the sum of two independent random variables with known distributions. This will mean that the user interface may remain fairly user friendly; the user does not need to input any code and can specify each distribution separately in a similar way to the previous application. The true probability function and tilted distributions will no longer be able to be plotted, making for a tool that is less useful for explaining the saddlepoint approximation, but one that has more practical use.

6.1 Coding Decisions

As before, here I will detail some of the major decisions made in the coding stages of the project. This application is just an extension of the previous one, and therefore most of the hard work had already been done.

6.1.1 Layout - Two Columns

The user is expected to input the distributions and parameters for two different random variables. In order to ensure that the sidebar is not unreasonably long (which would mean that the user may not be able to change a reactive parameter and see its effects simultaneously) the two sets of widgets, each corresponding to one random variable to sum, are placed next to each other. This is achieved with use of the `column()` function.

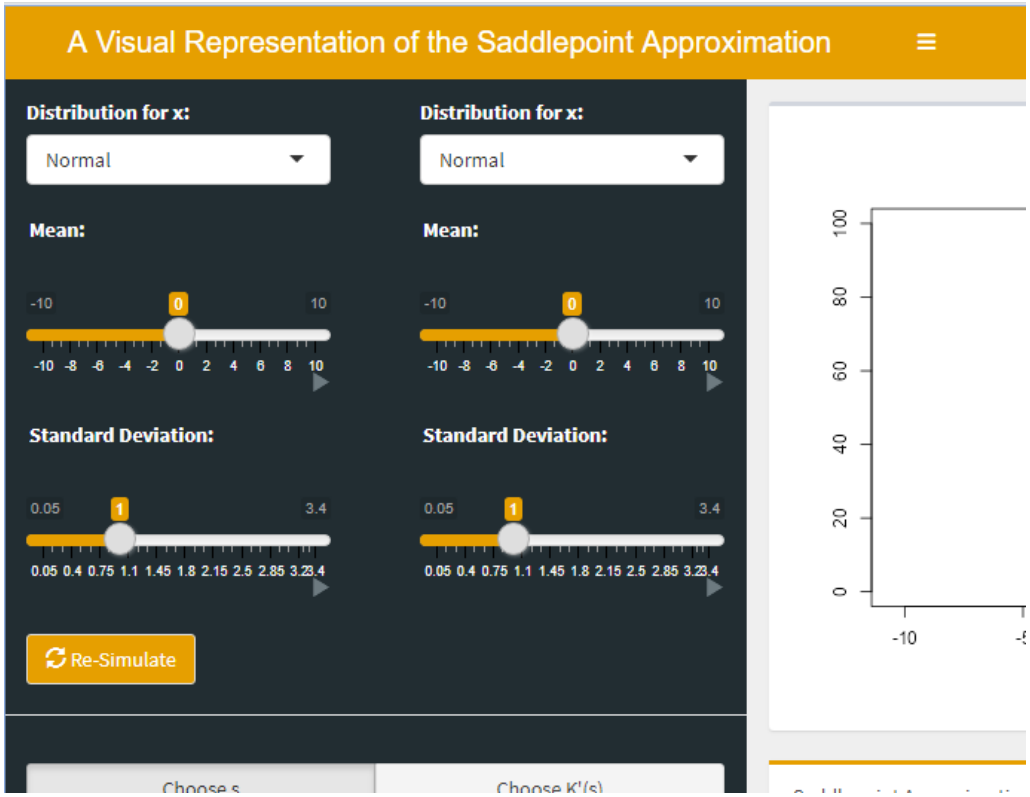


Figure 6.1: Widgets for inputting the distribution and parameters for two random variables

The downside of this is that the sidebar must be wider in order to accommodate the additional widgets, meaning that the plots are smaller. However, it is much improved from the layout made without using columns.

Other than this change, the layout is identical to the previous application.

6.1.2 Adapting the Previous Application

Most of the hard work has already been done. In this application, we are not attempting to show or use the true distribution of the sum of our random variables. So all that is left is to find the moment generating function, and the sampling distribution - these are the only two things that our application requires in order to display everything that is needed.

Sampling Distribution

Just like in the previous application, we would like to have the ability to reactively update the parameters of the distributions (see section 5.2.3) without re-sampling from the distribution. It is not hard to adapt the previous application to do this in the case of a sum of random variables. Here, we simply will have two different lists of random variables saved - x_1 and x_2. We have seen how to use quantile functions to get the new x-coordinate of a point; we can simply do this for each x_1 and x_2. Then, both lists are added together; this gives us all the points we need for the scatterplot, and both the 'simulated distribution'

lines.

Moment Generating Function

The only other mathematics that we need to do is to find the moment generating function. As it turns out, the moment generating function of the sum of two random variables is the product of the individual moment generating functions. This is very easy to show from the definition:

$$M_{X_1+X_2}(s) = \mathbb{E}\left(e^{s(X_1+X_2)}\right) = \mathbb{E}\left(e^{sX_1} \cdot e^{sX_2}\right) = \mathbb{E}\left(e^{sX_1}\right) \cdot \mathbb{E}\left(e^{sX_2}\right) = M_{X_1}(s) \cdot M_{X_2}(s)$$

We can then find the cumulant generating function, and its derivatives, in the same way we have seen previously.

Choosing Minima/Maxima of Sliders

We have seen in section 5.2.8 that determining the minimum and maximum x and s values is not trivial. However, most of the work we have already seen still holds. We only really need to find the minimum value x can have (supposing that its matching s value is allowed, and it is above the lower x limit), and the maximum value s can take (supposing its matching x value is allowed, and it's smaller than, say, 10).

We already know the minimum x value for both x_1 and x_2 (call these `min_x1` and `min_x2`), and the maximum s value for both x_1 and x_2 (call these `max_s1` and `max_s2`). These are hardcoded.

- We let the minimum x value of the sum of x_1 and x_2 be `min_x1` plus `min_x2`. The reason for this is self-explanatory.

- We let the maximum s value of the sum of x_1 and x_2 be the minimum of `max_s1` and `max_s2`. The reason we had these maxima in place was that there were numerical errors when we tried to find the derivatives of the cumulant generating function above these values of s - these errors poisoned the whole vector. And so, we cannot ever have s above either `max_s1` or `max_s2`.

6.1.3 Colour of the Navigation Bar

We have seen in 5.2.11 that we can change the CSS to change the colour of the navbar and the sliders. I wanted this application to look visually different to the other application, and so this application was chosen to have a yellow/orange navbar and sliders. Using any bright colour meant that there was plenty of contrast between the sliders and the background, so this was a reasonable choice. It also matches the rest of the application, since it comes from the Okabe-Ito palette.

6.2 The Final Application

Just as with all the applications made in this project, the code can be found on GitHub at `https://github.com/alicemhankin/saddlepoint-visualisation`. A screenshot of the final application can be seen below.

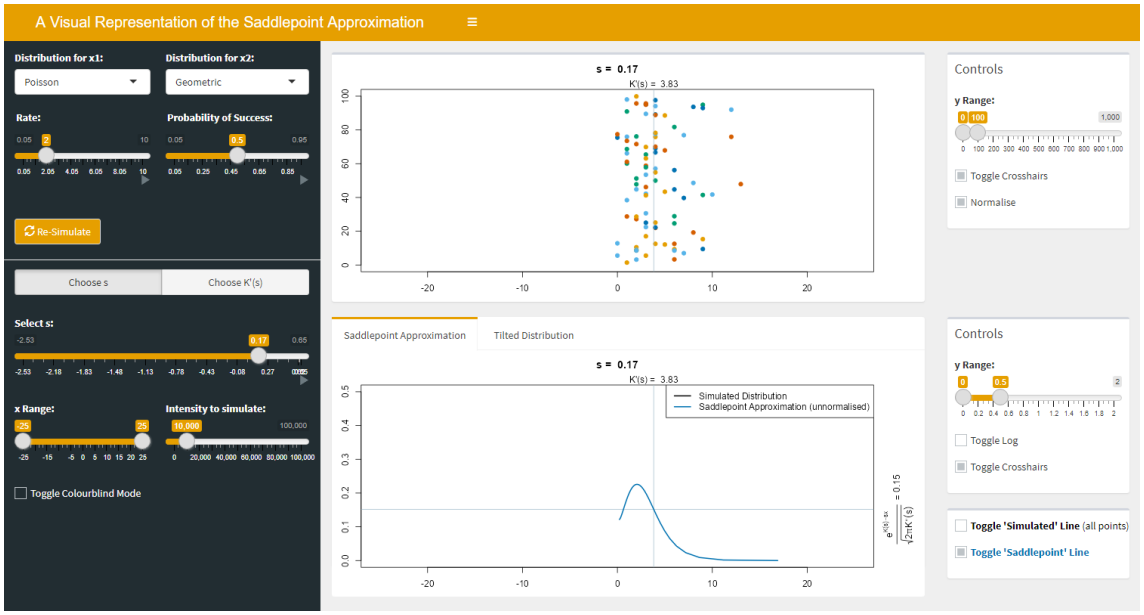


Figure 6.2: The Final Application

Chapter 7

Application - Bivariate Distribution

For my final application, I was interested in creating an app that allows the user to visualise the saddlepoint approximation for bivariate distributions.

7.1 Coding Decisions

As in the previous two chapters, this section will detail some of the coding decisions that were made in creating the application. I assume that the reader will have read the previous two chapters; the code for this application builds off the previous two.

7.1.1 Target Audience

The target audience for this application is those who are already familiar with the saddlepoint approximation and writing R code, who are using this as a tool for their research. The reason for this was twofold:

- People interested in bivariate data are more likely to be knowledgeable about moment generating functions and R code. If these concepts are new to someone, it is unlikely that they require any more functionality than is available in the previous two applications.

- There is no way to make an aesthetically pleasing user interface which allows for the user to choose between many different bivariate distributions. Of course, if we had a discrete number of distributions that it made sense to include in the application - as in the first application we saw in chapter 5 - we could develop some effective UI, however, this is not the case. This means that we the widget used to input the sampling distribution and the moment generating function *must* be textboxes where the user directly inputs R code - the user is required to have some R knowledge.

Although we still want to design an attractive user interface, we can afford to worry less about accessibility to *all* audiences, and focus on the flexibility the application affords.

Since we will have textboxes for directly inputting R code, this application will not be able to be published, due to potential malicious code. However, the code will be open-source on GitHub; anyone interested can run the application on their own machine.

7.1.2 Distribution Inputs

We have discussed in the previous section that this application is designed for maximum flexibility; we want to allow the user to be able to visualise the saddlepoint approximation for any (bivariate) moment generating function. There is only one way to allow the user to specify any moment generating function - they must have to input R code directly into the application.

As well as the moment generating function, the user will input a function to sample n values from the bivariate distribution they are interested in. If we recall from the previous two applications, these are the only two inputs that are required in order to plot both the simulated distribution and the saddlepoint approximation.

The following figure illustrates what the widgets for the two text inputs look like to the user.

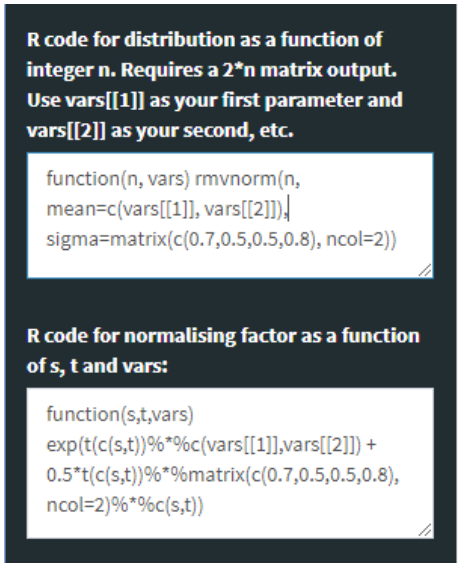


Figure 7.1: Widgets which allow the user to enter the moment generating function and the sampling distribution for the application

For ease of use, the text boxes can also be made larger if the user drags the lower right corner, allowing for arbitrarily long user input.

Note that these two inputs do not tell us the true distribution. Recall that the saddlepoint approximation most often is used in cases when we don't know the true distribution. Especially in the bivariate case, for most interesting scenarios, the true distribution will not be easy (or possible) to find. The true distribution plots will therefore be omitted in this application.

7.1.3 The `vars` list

In figure 7.1, we can see that the user references a list - `vars`. This is designed to contain the parameters of the distribution that the user is interested in changing. For example, in said figure, which represents a bivariate normal distribution, the mean value, as a vector, looks like $\begin{pmatrix} \texttt{vars[[1]]} \\ \texttt{vars[[2]]} \end{pmatrix}$.

The idea is that the user can reactively change the values of the members of the `vars` list. Each of `vars[[1]], vars[[2]], vars[[3]]`, and so on should be controlled by its own widget.

In the rest of this chapter, I use the word "parameters" to refer to the members of the `vars` list.

Changing Parameters

We saw in section 5.2.3 that when we know the true distribution, it is easy to use the quantile function to make it such that when we change the value of a parameter, our randomly sampled points do not completely reshuffle.

However, here we do not have access to the true distribution. It is therefore not possible to develop a method which avoids resampling the points when the parameters are changed. In this application we must use a random seed each time the parameters change.

Generating a reactive number of parameter boxes

The user should be allowed to have as many parameters as they desire. The issue with this is that each new parameter requires its own input box. Whilst it would be possible to allow the user to input a list into a text box, this does not allow them to change each parameter independently in a straightforward manner.

The solution is to have one widget where the user can enter the number of parameters they wish to include in their model. Then, reactively, said number of widgets (either sliders or numeric inputs) will appear, each corresponding to its own parameter.

If we place the code which creates the widgets on the server side, it is possible to reference other inputs. In this case, we are interested in referencing the value that corresponds to the number of widgets the user wants to create, which itself is an inputted value.

It is then simple to create a loop (or in this case, a call to `lapply`) to render n new widgets, where n is the input of another widget (in this case, `slider_val`). We can then create a list, `vars`, which consists of all the user-entered values in each of the dynamically-created widgets. Of course, this is the same list that the user references in both text inputs, as we've seen in figure 7.1.

The code for the creation of n widgets is below.

```
output$dynamic_widget <- renderUI({
  num <- as.integer(input$slider_val)
  column(12,lapply(1:num,function(i) {
      column(4,numericInput(inputId = paste0("var",i),label=paste0("vars[[",i,"]]"),
                            value=0.5, step=0.1), style='padding:0px;')
    }), style='padding:0px;')
})
```

Figure 7.2: Code to create n widgets, where n is controlled by another widget

Sliders or numeric inputs?

For the members of `vars`, it was initially unclear which widget is best used to input the values; a slider or a numeric input. The two options with full width can be seen in the following figure.

Figure 7.3: Inputs for the entries of `vars` - sliders (left) and numeric inputs (right)

Having the widgets full size (i.e. taking up the full width of the box) takes up a lot of space on the screen. If there are more than a few members of `vars`, we have to scroll down to see the end of the list. To deal with this, it is possible to reduce the size of each widget horizontally by two-thirds, as in the following figure.

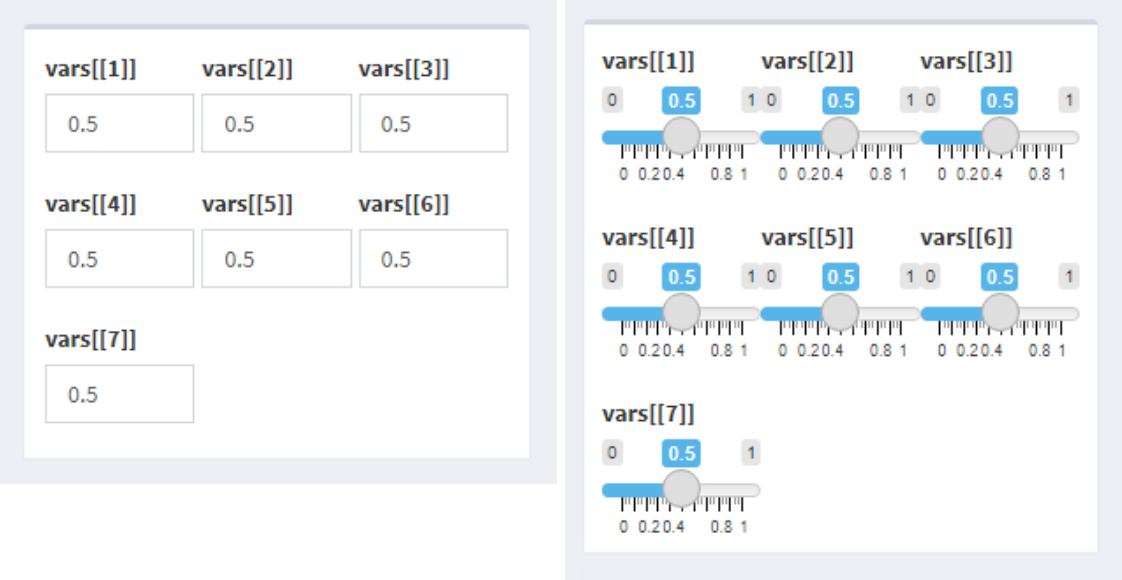


Figure 7.4: Third-size inputs for the entries of `vars` *- sliders (left) and numeric inputs (right)*

Although sliders may look more aesthetically pleasing on the large scale, the user interface (particularly the tick labels) ends up looking very cramped. It is also very hard to make small changes to their values since they are so small on screen.

The other downside is that we have to choose minimum and maximum values for the sliders. Having the user input these values is one possibility, but this creates yet more inputs the user needs to worry about. Having fixed values is also not ideal since it is restrictive - there is no way to know what kind of values a parameter in any given case might be required to take.

Recall that we cannot have animation as we did in the single dimensional case, due to the fact that the points are resampled each time, so the slider option doesn't provide any benefits. The numeric inputs take up less space and look more appealing, so these were chosen.

7.1.4 The Scatterplot

We are interested in plotting a scatterplot for sampled points. In the univariate case, we had values following some distribution on the x-axis, and a Poisson point process with rate 1 on the y-axis. When we change the tilting parameter s, we multiplied the y-coordinate of each point by tilting factor e^{-sx}, or if we want to have the density of points on the screen constant, tilting factor $M(s)\exp(-sx)$.

In the two dimensional case, we have one extra dimension. Each point has three coordinates, x_1, x_2, and y. x_1 and x_2 follow some distribution - in particular, the sampling distribution specified by the user. These could be independent, or not. The y value is unchanged from the one dimensional case; it is a Poisson point process with rate 1. Instead of one tilting factor s, we now have two tilting factors, s and t. s corresponds to the tilting

factor on the x_1 axis and t corresponds to the tilting factor on the x_2 axis.

The user will be able to control the tilting factors s and t, as well as the number of points that are simulated.

When changing s and t, we will multiply the y-coordinate of each point by tilting factor $\exp(-sx_1 - tx_2)$, or, if we want the density of points on the screen constant, tilting factor $M(s,t)\exp(-sx_1 - tx_2)$. Recall that this mapping makes it so that, if we look at the points below a particular y-value, provided enough points are simulated, the points will follow the tilted distribution, as tilted by (s,t).

Finding the scaling factor of the points follows directly from the work we did in appendices A.3.

Drawing the Points

We have a list of points, each with three dimensions - x_1, x_2, and y. There are two ways that we could plot this, as shown below.

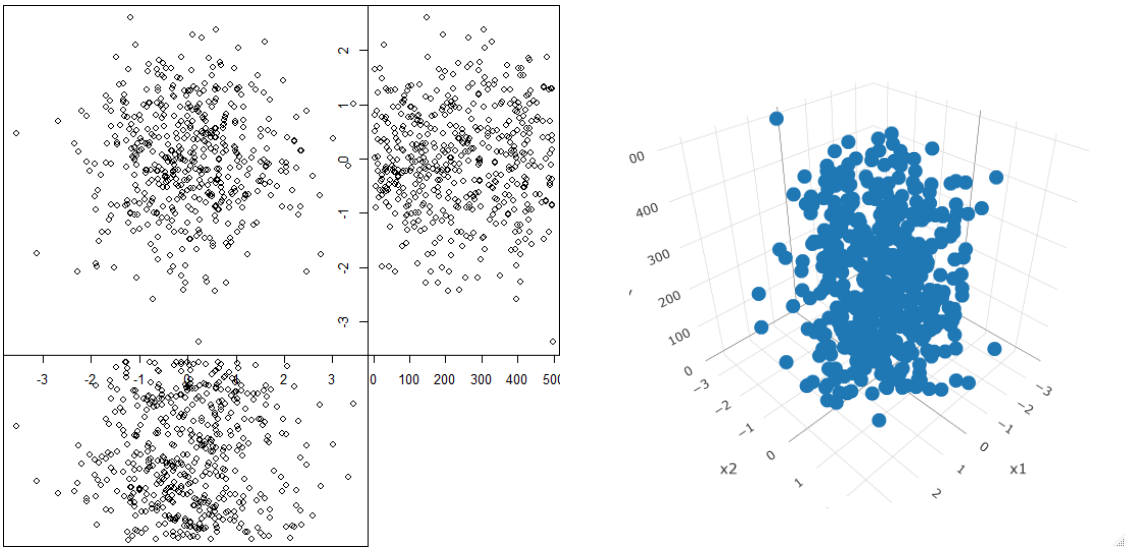


Figure 7.5: Two methods of visualising points in three dimensions. Three 2D scatterplots (left) and one 3D scatterplot made with plotly [46] (right)

The plot on the left has three separate scatterplots. The top-left quadrant shows x_1 and x_2; the top-right quadrant shows y and x_2; the bottom-left quadrant shows x_1 and y. The plot on the right shows one three-dimensional plot. The y values act on the up-down axis, and the x_1 and x_2 values are on the forward-backward and left-right axes.

One benefit of the three-dimensional plot is that it is interactive, so the user can see it from all angles. On the other hand, it looks cluttered and hard to read. The time to generate the plot is also much longer than for the two-dimensional plots.

There isn't much additional information that the three-dimensional plot is giving us. We

aren't all that interested in viewing the three dimensional points from any other angle other than the ones the 2D scatterplots are affording us.

If we want to overlay any other information, it is much easier to see in two dimensions. I discuss this in section **??**. Therefore, I went with the three two-dimensional scatterplots.

Sampling Mechanism

Just like in the univariate case, we can either plot all the points, or just a subsection of the points. If we plot all the points, we are left with a density that reflects the true density. If we plot just the points below a particular y-value, as long as enough points are simulated, the simulated density will reflect the tilted density.

It would be of interest to include both of these sampling mechanisms in the application.

For the distribution that corresponds to the tilted density, the y-value that we will use is simply the y-limit as inputted by the user.

For the simulated distribution that corresponds to the true density, instead of plotting every point, we will just plot all points that have initial y-values (i.e. y-values before tilting has been applied) below the inputted y-limit. This will ensure that both our plots have around the same number of points and we won't have overprinting.

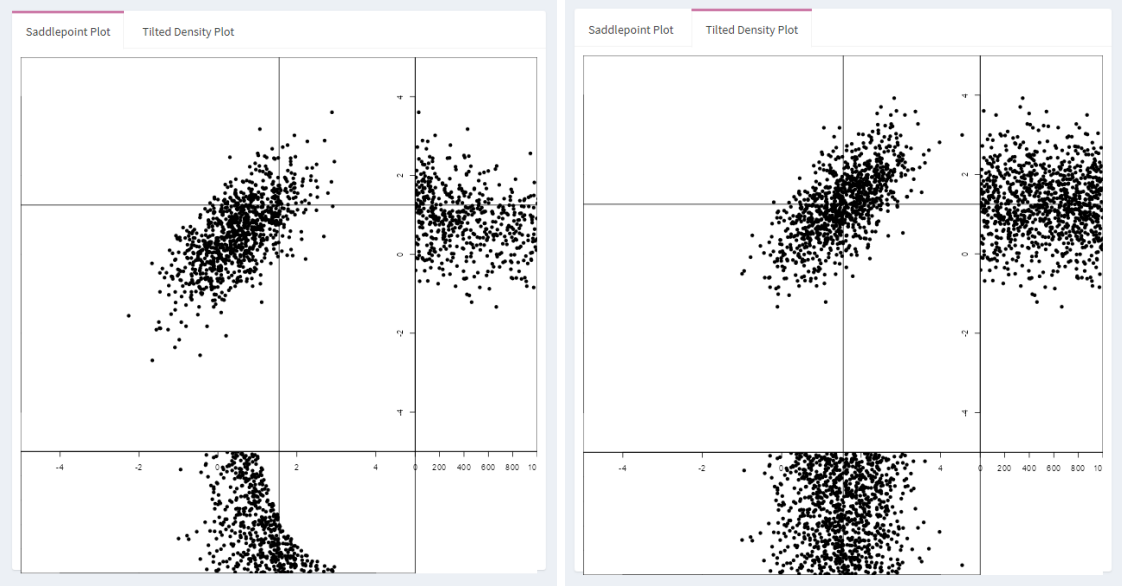


Figure 7.6: The simulated approximation to the true density (left) and tilted density (right) of a bivariate normal distribution tilted by $(s, t) = (1.5, 0)$

Above, we have a bivariate normal distribution with mean $(0.5, 0.5)$ and covariance matrix $\begin{pmatrix} 0.7 & 0.5 \\ 0.5 & 0.8 \end{pmatrix}$, tilted by $(s, t) = (1.5, 0)$. We have the approximation to the true density on the left and the approximation to the tilted density on the right. The crosshairs indicate the mean of the tilted distribution.

7.1.5 The Saddlepoint Approximation Plot

To give our simulated points a point of comparison, it would be useful to compare our points which approximate the true distribution to another approximation. In this section, we discuss how the saddlepoint approximation to a point (x_1, x_2) is found, and how this is visualised.

Mathematics - Finding the Saddlepoint Approximation at (s, t)

Recall that the user has inputted the moment generating function $M(s, t)$. Now, in order to calculate the saddlepoint approximation, we need to first find $K(s, t), K'(s, t)$ and $K''(s, t)$.

This is trivial - the same method as in the univariate case can be used. We can find all first and second derivatives of $M(s, t)$ using the `Deriv()` function from the Deriv [37] package.

Here, I will use notation $f_x(x, y) := \frac{\partial f(x,y)}{\partial x}$, $K := K(s, t)$ and $M := M(s, t)$

Then we can simply find:

$$K'(s,t) = \begin{pmatrix} K_s \\ K_t \end{pmatrix} = \begin{pmatrix} \log(M)_s \\ \log(M)_t \end{pmatrix} = \begin{pmatrix} \frac{M_s}{M} \\ \frac{M_t}{M} \end{pmatrix}$$

Similarly, we get:

$$K''(s,t) = \begin{pmatrix} \frac{M_{ss} \cdot M - M_s^2)}{M^2} & \frac{M_{st} \cdot M - M_s \cdot M_t}{M^2} \\ \frac{M_{st} \cdot M - M_s \cdot M_t}{M^2} & \frac{M_{tt} \cdot M - M_t^2}{M^2} \end{pmatrix}$$

In order to find the saddlepoint approximation at any value of (s, t), we can simply enter these two values into the multivariate saddlepoint approximation formula we've seen in section 2.4.3, that is:

$$\hat{f}(s,t) = \frac{\exp(K(s,t) - (s,t) \cdot K'(s,t))}{\sqrt{\det(2\pi K''(s,t))}}$$

We want to draw the saddlepoint approximation on a plot that has x_1 on one axis and x_2 on the other axis. What we plan to do is split this surface up into a grid, and calculate the saddlepoint approximation at each square on this grid. We can then plot this in a number of ways; a heatmap or a contour plot for example. In short, we need to be able to find the saddlepoint approximation at certain (x_1, x_2) coordinates. Looking at the formula above, we can see that this is all in terms of (s, t) and not (x_1, x_2).

Recall from **??** that there is a relationship between (x_1, x_2) and (s, t) - namely that $K'(s, t) = (x_1, x_2)$. In order to plot the saddlepoint approximation for each point in our grid, we first need to find the corresponding value of (s, t). I discuss this in the following chapter.

Finding (s, t) **given** (x_1, x_2)

We have a list of (x_1, x_2) values that we need to find the saddlepoint approximation for i.e. we need to find the approximation at every point in some grid.

Finding the saddlepoint approximation at a point is harder in the bivariate case than the univariate one. This is because our cumulant generating function, and first/second derivatives are functions of s and t, not functions of x_1 and x_2. We have to first find the (s,t) values that correspond to our (x_1,x_2), and then find the saddlepoint approximation at that (s,t) coordinate.
We did not have to do this in the univariate case since we could plot $(K'(s), \hat{f}(s))$ using the `line()` function. The plotting functions (such as `image()` or `contour()`) in the multidimensional case, however, require grid-like data, so we require the extra step.

Naïvely, we would select each (x_1,x_2) value, then cycle through each (s,t) to see which (s,t) coordinates make $K'(s,t)$ closest to (x_1,x_2). This is highly inefficient due to the fact that we calculate $K'(s,t)$ for each (s,t) many many times over.

Instead, we can simply calculate $K'(s,t)$ for each s, t. Then, for each (x_1,x_2) we use a nearest neighbour function to see which values of s,t make $K'(s,t)$ closest to (x_1,x_2). The nearest neighbour function used in this project is `nn2()` from the RANN package [47]. Then we can easily find the saddlepoint approximation for that (x_1,x_2) coordinate since we know the corresponding (s,t).

Rather than the naïve approach, which involves four for loops, this entire method can be done with no loops, entirely using pre-written and vectorised functions such as `mapply()`. The code is shown below.

```
cc = cgfs()
vars = vars()
x1seq = seq(minmax_x()[1],minmax_x()[2],0.2)
x2seq = seq(minmax_x()[3],minmax_x()[4],0.2)
sseq = seq(st_lims()[1],st_lims()[2],0.2)
tseq = seq(st_lims()[3],st_lims()[4],0.2)

corresponding_t = rep(tseq, length.out = length(sseq)*length(tseq))
corresponding_s = rep(sseq, each=length(tseq))

x_for_each_st = mapply(function(s, t) get_ks(s, t, cc, vars)[[2]], corresponding_s, corresponding_t)
x_for_each_st = t(x_for_each_st)

each_x = cbind(rep(x1seq, 1, each=length(x2seq)), rep(x2seq, length(x1seq)))
indexes = nn2(x_for_each_st,each_x, 1)$nn.idx

s = corresponding_s[indexes]
t = corresponding_t[indexes]

approx = mapply(function(s, t) get_approx(s, t, cc, vars), s, t)

cbind(each_x, approx)
```

Figure 7.7: A fully vectorised method for generating the saddlepoint approximation

Choice of Plot

We have a list of (x_1,x_2) coordinates, along with the saddlepoint approximation at that point. There are a few ways of drawing a distribution for this data; we can see three in the figure below.

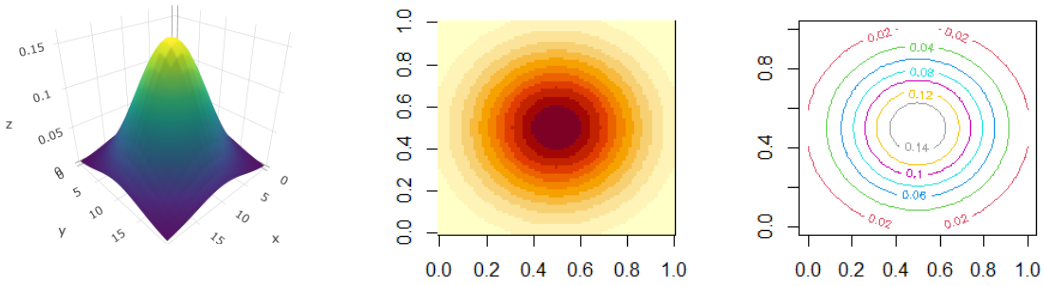


Figure 7.8: Three ways of plotting bivariate data. From left to right, an interactive surface plot made using plotly [46], a heatmap, and a contour plot

We have seen that we are plotting the points in two-dimensions, so the surface plot would not match the aesthetic. It would be very hard to read (and visually unappealing) if the points of the simulated distribution were overlaid on top (we discuss this further in **??**). As well as this, it takes much longer to render and I felt that the side profile didn't contribute any more understanding of what the density looks like (over the other two options).

I felt that the heatmap suits this project the best. The contour plot looks far less visually striking and the values of the contour lines aren't very useful.

We can plot the height of the saddlepoint approximation for each (x_1, x_2) in our grid in the upper-left quadrant of our scatterplot. It makes sense to have the other two quadrants show the marginal distributions of the saddlepoint approximation. We can find this by summing the rows/columns of grid entries and plotting this.

Now, we could have included the heatmap next to the points, however this is not necessary - we can simply overlay the two in order to have a better comparison.

Final Plot

Recall that the saddlepoint approximation is calculated over some grid of (x_1, x_2) values. The finer the grid, the slower but more smooth-looking; the coarser the grid, the quicker but less smooth. I chose a grid size of 0.2 which seemed like a good mix between efficiency and aesthetics.

A plot comparing the simulated points to the saddlepoint approximation heatmap can be seen below.

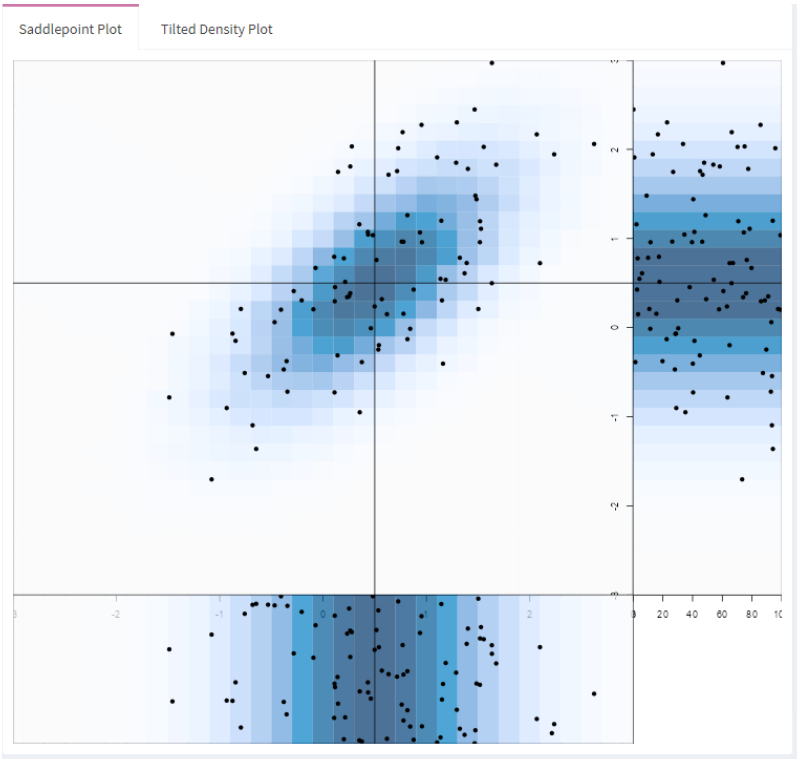


Figure 7.9: cpation

Note that we have crosshairs - these specify the value of s and t that is specified by the user. Changing these will not change the saddlepoint approximation, but they will change the tilted distribution plot which we discuss in the following section.

7.1.6 The Tilted Distribution Plot

Recall, we have a way of sampling points that reflects the tilted density. We don't have access to the true tilted distribution, but we know what the normal approximation to the tilted distribution looks like. We can use this as a point of comparison to our simulated distribution.

Just like in the univariate case, the normal approximation to the tilted distribution has mean $K'(s,t)$ and $K''(s,t)$. And so when we tilt our points by (s,t), it is easy to plot the tilted approximation at that point. This should correspond with our point density. We can see an example below.

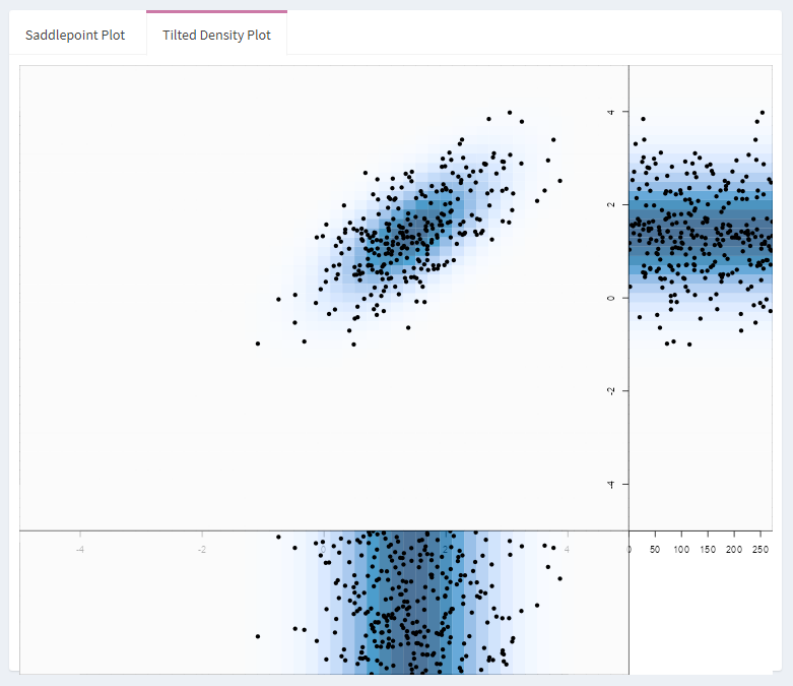


Figure 7.10: Simulated points, and normal approximation to the tilted density of a $Normal\left(\begin{pmatrix}0.5\\0.5\end{pmatrix}, \begin{pmatrix}0.7 & 0.5\\0.5 & 0.8\end{pmatrix}\right)$ *distribution with tilting factor* $(s,t) = (1, 0.5)$

The `mvtnorm` package [48] has been used to find the density of the $N(K'(s,t), K''(s,t))$ function at each point in our grid.

Toggles for the plot

There were a lot of widgets that we want to include in order to control this plot. These widgets will affect both this plot, and the saddlepoint approximation plot discussed in **??**.

- A toggle to turn on and off the crosshairs which show the mean of the tilted distribution
- A slider which specifies the opacity of the points, and another which specifies the opacity of the heatmap. This is because perhaps the user is only interested in one or the other. This also prevents overprinting which will occur depending on the number of points simulated.
- A zoom-in slider for both the x_1 and x_2 axes.
- A widget to select the y-range shown in the graph.
- A widget which allows the user to select the number of points simulated - the "intensity to simulate".
- A selector for each member of the `vars` list. This is discussed in **??**.

- A selector for s and t
- A button which allows the user to toggle on or off normalising.

7.1.7 Point Heatmaps

As well as comparing the simulated points to the saddlepoint approximation (and to the normal approximation to the tilted distribution), I felt it was important to have a direct comparison. The issue with comparing points to a heatmap is that they are representing data in different ways. It would be beneficial to provide a heatmap for the simulated points, so that we can directly compare the simulated distribution with the saddlepoint approximation (or the normal approximation to the tilted distribution).

Below, we can see what the toggle looks like:

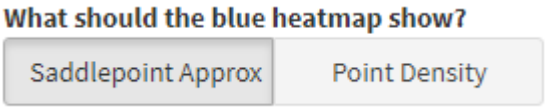


Figure 7.11: aa

And here's what it looks like when the user wants the heatmap to reflect the point density:

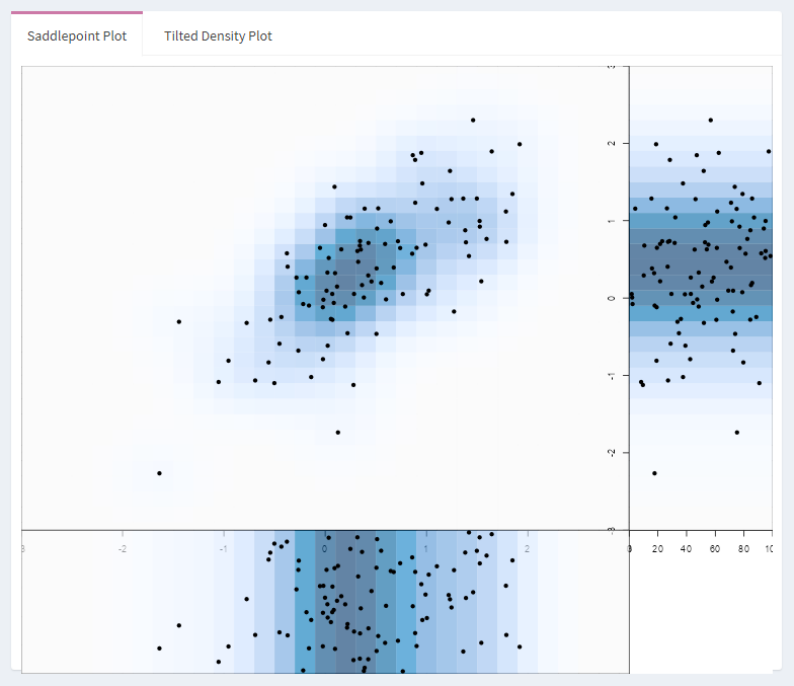


Figure 7.12: bb

7.1.8 Colour Choices

We have two plots - the saddlepoint approximation and the tilted density. Both have three different elements - the points of the simulated distribution, the heatmap corresponding to the saddlepoint approximation, and the heatmap corresponding to the simulation.

It is worth noting that each of these plots have a analogous plot in the univariate applications already made. In those applications, we used a blue line to represent the saddlepoint-related data, and black lines to represent the simulation-related data. We would like to have consistency, and repetition between applications; the same colour scheme is used here.

It is easy to make the points in the scatterplot black, and the saddlepoint-related heatmaps blue. However, the heatmap for the simulated points must also be made blue for two reasons:

- We must be able to compare like-for-like between the two heatmaps
- The black points must stand out from the background

Having a completely consistent colour theme is much less important than having the plot represent the data accurately.

7.1.9 Two Modes and the inverse of the $K'(s)$ function

It would also be useful to have a mode where the user is able to select $K'(s,t)$ (i.e. the mean of the tilted distribution) instead of (s,t) (i.e. how much the distribution is tilted by).

All of the calculations that we currently have are based off of the current tilting factors (s and t) that are inputted by the user. And so, if we allow the user to input $K'(s,t)$, we need to find a way to derive s and t from the input - we need an inverse function of $K'(s,t)$.

This isn't too hard using the work we have already done. Recall from the saddlepoint approximation plot that we had a function which takes all possible values of (s,t) and calculates $K'(s,t)$. Then, we used a nearest neighbour function to compare each of these $K'(s,t)$ values to the relevant value of (x_1, x_2) to see which one was closest, and see which (s,t) gave that value of $K'(s,t)$. It looks like our inverse function is already written for us!

And so, we can allow the user to switch modes, allowing them to input $K'(s,t)$ instead of (s,t). All we have to do on the coding end is use our inverse function to find what values of (s,t) correspond to the inputted $K'(s,t)$ and use that in our calculations.

7.1.10 User Interface

This section will discuss the choices made in creating the user interface. If something is not explicitly stated in this section, such as the use of shinydashboard, it is because it

is carried through from the first two applications made - these decisions were made in chapter 5. It is important to have consistency and repetition, so these UI choices were left unchanged.

Sidebar

There are so many widgets that are required to be on the page. In order to determine the ones that should be on the sidebar, I had to think about how the application would be used in practice. The user would have to specify a few bits of information about the distribution before the application could run. This would be the moment generating function, the sampling distribution, the range that x_1, x_2, s, and t are allowed to take, and how many parameters (i.e. entries in the `vars` list) are needed. The application should *not* update reactively to changes in these widgets. This is because there will be errors if the moment generating function is updated before the range of s/t values are not first decided. It makes a lot of sense to have the user fill in all of these options first, and then press a button to apply them.

These widgets are excellent options to put in the sidebar since they affect the application globally. These only have to be set one time, and then the user can play around with other options such as changing the parameters, the selected value of s, or the y-range for which points are visible. The idea is that the user sets these values, presses an "apply" button, then minimises the sidebar.

In order to ensure the user wouldn't have to scroll in order to see all of these widgets, several of the sliders had to be half-width. However, all widgets did end up fitting on the page. The sidebar can be seen on the right.

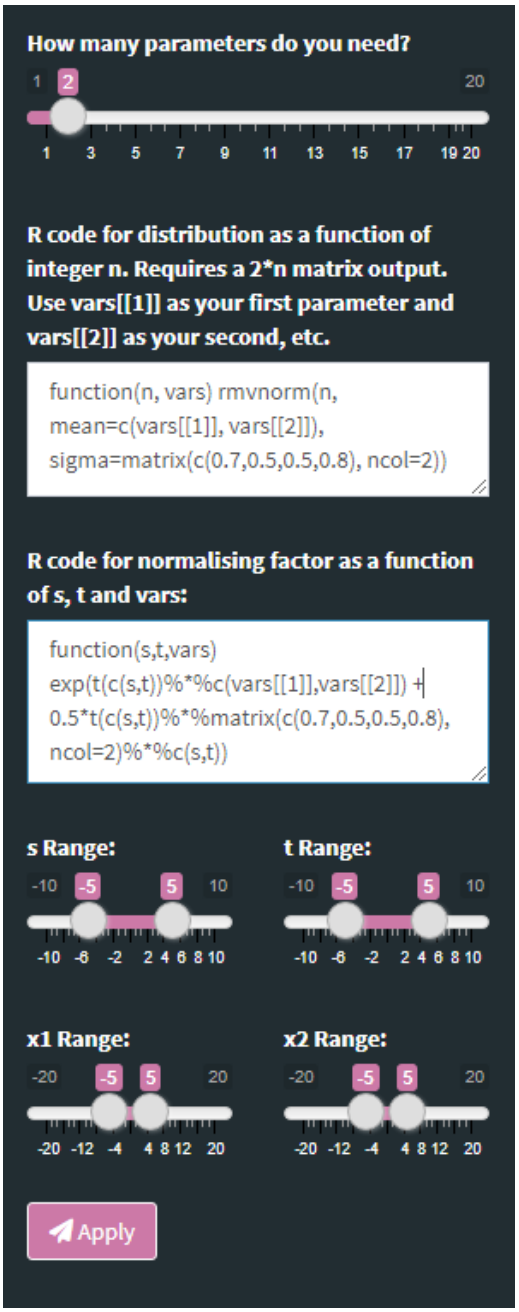


Figure 7.13: The sidebar

The Tab Box

There are two plots we wish to display; the saddlepoint approximation plot and the tilted distribution plot. Whilst it would be possible to fit these both on the same page, it is

not really necessary. We are only really interested in comparing between the sampling distribution and the saddlepoint approximation or approximated tilted distribution i.e. we do not need to see both on one page at the same time.

It is therefore a good idea to put both tabs in a tab box. The user can select which plot they are interested in seeing via tabs at the top. Including the plots in this fashion means that they can take up the entire page, rather than being reduced to half-height.

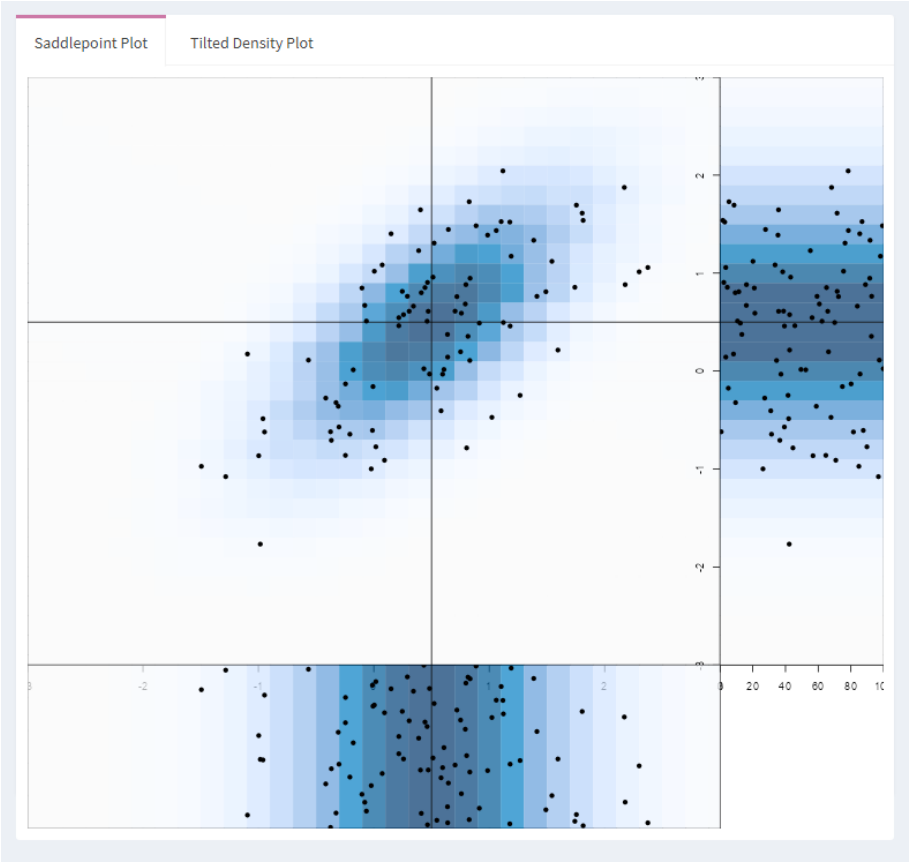


Figure 7.14: A tab box - the user can choose between the tilted density or the saddlepoint approximation plot

Box Layout

We have numerous other widgets to include in the application:

- Choosing each value in the `vars` list
- Selection of "mode" (i.e. choosing whether to input (s,t) or $K'(s,t)$)
- Selection of (s,t) or selection of $K'(s,t)$
- Normalising toggle
- Selecting whether the blue heatmap displays the simulated density or approximation to true/tilted density
- Opacity of heatmap and points
- Zoom of x_1 and zoom of x_2
- y-range
- Intensity to simulate
- Crosshairs toggle

These had to be sorted into two. They all do not fit into one box, without the user having to scroll, so we have to split them into two. The two lists above are a reasonable choice for how this could be done; the list on the right contains widgets which change the visual appearance of the plot, whereas the left left contains widgets which change the underlying mathematics.

When the sidebar is minimised, we have the tab box, and now two boxes containing widgets. Had the two widget boxes been next to each other, the application looks more crowded - the number of widgets is overwhelming to the user. The best layout is when the tab box containing the plots is in the middle of the two boxes. Since the user is likely to want to select the parameters before the appearance of the plot, this is selected to be on the left. This appearance can be seen below.

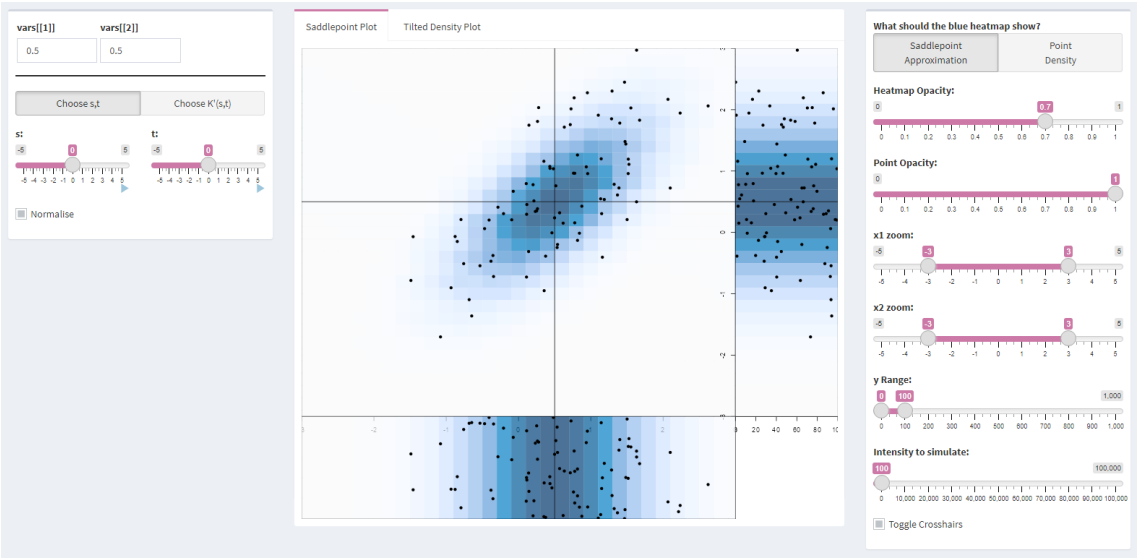


Figure 7.15: Layout of the boxes

Colour of the Application

Just like in the previous two applications, I had to choose a colour for the sliders and the navigation bar. I ended up going with a greyish pink/purple. This colour is bright enough that it will show up against both the light background of the boxes, and dark background of the sidebar. It isn't too garish, however. It also looks significantly different to the colours used for the other two applications.

Widget Choice

The widget choice should be the same as in the univariate applications. Simple square toggles were chosen for the less-important inputs which had a state of being either on or off. Larger radioGroupButtons were used for toggles that were more important, or if both states had to be labelled. Other than the numeric inputs discussed in section **??** and the textboxes discussed in section **??**, every other input is a slider. This is because they all have a defined minimum/maximum - the reason sliders are used throughout is discussed in **??**.

7.1.11 Initial Distribution

The application has to have initial values. This is the state of the application when it is first opened. If the user isn't sure where to start, they could perhaps try changing the parameters on a known distribution.

I decided to make the initial distribution a bivariate normal distribution with a variable mean (i.e. the mean is $\begin{pmatrix} \texttt{vars[[1]]} \\ \texttt{vars[[2]]} \end{pmatrix}$), and fixed variance matrix of $\begin{pmatrix} 0.7 & 0.5 \\ 0.5 & 0.8 \end{pmatrix}$.

Here are the initial inputs:

- We require *two* parameters (members of `vars`). These will correspond to the mean values of x_1 and x_2.
- The sampling distribution is easily inputted via use of the `mvtnorm` package [48]:
  `function(n, vars) rmvnorm(n, mean=c(vars[[1]], vars[[2]]),sigma=matrix(c(0.7,0.5,0.5 ncol=2))`
- The moment generating function is given by:
  `function(s, t, vars) exp(s*vars[[1]] + t*vars[[2]] + 0.5 * (0.7*s^2 + s*t + 0.8*t^2))`
  This is simple to calculate by hand, but a full workthrough by Tamás Linder is also available online [49].
- The x_1, x_2, s and t limits are unrestricted. These can be set to the maximum.

Screenshots of the output of these initial values can be found in the following section - figure 7.16.

7.2 The Final Application

The code for the final application can be found on GitHub at `https://github.com/alicemhankin/saddlepoint-visualisation`. We can see a screenshot of the application below:

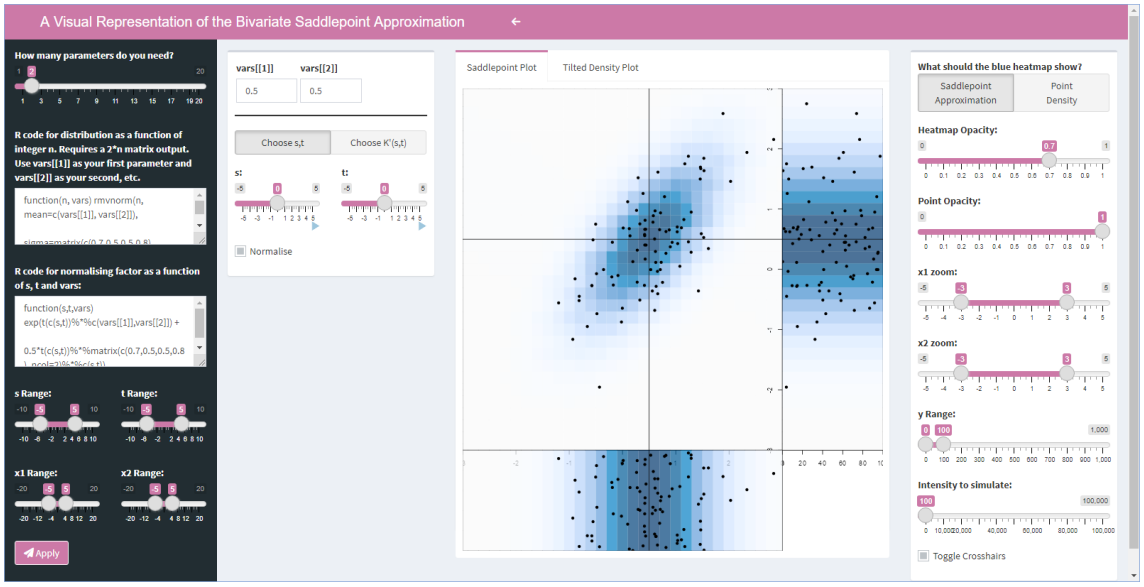


Figure 7.16: The Final Application

Chapter 8

Discussion

This chapter will cover possible future work, then finally a summary of the work completed during this project.

8.1 Limitations/Future Work

All the goals that I set out to complete were achieved, so there is not a whole lot of future work to discuss. However, there are a few minor improvements to the applications, and suggestions for further applications which are worth mentioning here.

Change to the Bivariate Distribution Application: Syncing of the K'(s,t) and the (s,t) sliders
We saw in section **??** that it is possible to have a system where the $K'(s)$ slider and the s sliders match, i.e. when we switch between the two modes, there is no change in the selected tilting factor. This change is not yet implemented in the bivariate distribution application, but would be a slight improvement to the user experience.

Change to the Bivariate Distribution Application: red lines
We saw in section **??** that the initial application written by Jesse Goodman, there were red lines indicating the maximum y-values that any simulated points could have. This feature was kept, for both the univariate distribution applications. However, it was not implemented in the bivariate distribution application. With an extra dimension, this 'red line' would look like a surface, above which points cannot be. It is non-trivial to develop a way in which this surface could be presented in the three two-dimensional scatterplots that are used to display the points. The fact that this is not able to be implemented easily is perhaps a limitation of the two-dimensional plot usage; if one were to require this implemented they may be forced to use three-dimensional plots instead.

New application: a sum of n random variables with known distributions
Another common use for the saddlepoint approximation is for when the random variable we are interested in is the sum of n random variables, each with the same distribution. It might be interested to see this implemented in an application, whether each of these n variables have a known distribution (i.e. one of the distributions in the first application

made) or a complex one that must be directly inputted by the user via R code in a textbox.

8.2 Conclusion

This project aimed to develop applications in R Shiny, the purpose of which was to visualise the saddlepoint approximation via comparison to a simulated approximation of the true density. The idea was that the applications would allow the user to explore, using an interactive user interface, how the saddlepoint approximation is generated, particularly with reference to exponential tilting. This report has detailed the development of three applications which do just that. The first application demonstrates how the saddlepoint approximation is generated for several well-known distributions, such as the Gaussian, Poisson, and binomial. The second application demonstrates the saddlepoint approximation to a sum of two known distributions for which the true density function is non-trivial. The final application allows the visualisation of bivariate probability functions.

The latest version of code for all three applications can be found at `https://github.com/alicemhankin/saddlepoint-visualisation`.

Appendix

A.1 Finding the Density of the Tilted Normal and Tilted Exponential Distributions

A.1.1 List of MGFs and CGFs for common probability distributions

Here is a very brief reference list for some probability distributions, along with their moment/cumulant generating functions, that I reference in this report. Due to the fact that finding the moment generating function requires calculating an infinite sum (as seen in equation 2.1), there are often restrictions on the values of s for which this sum converges. These restrictions have been noted in the fourth column.

Distribution	Moment Generating Function	Cumulant Generating Function	Restriction
Normal(mean = μ, st. dev = σ)	$\exp\{s\mu + \frac{1}{2}\sigma^2 s^2\}$	$s\mu + \frac{1}{2}\sigma^2 s^2$	None
Exponential(rate = λ)	$(1 - \frac{s}{\lambda})^{-1}$	$-\log(1 - \frac{s}{\lambda})$	$s < \lambda$

[50]

A.1.2 Density for a Tilted Normal Distribution

We know that the probability density function for the normal distribution is given by:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\}$$

And that the moment generating function is

$$M(s) = e^{s\mu + \frac{1}{2}\sigma^2 s^2}$$

from section A.1.1

Taking the natural logarithm of the moment generating function gives us the cumulative generating function:
$K(s) = \log(M(s)) = s\mu + \frac{1}{2}\sigma^2 s^2$

And so the density for the tilted normal distribution is:

$$
\begin{aligned}
\hat{f}(x) &= f(x) \cdot \exp\{sx - K(s)\} \\
&= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\} \cdot \exp\{sx - K(s)\} \\
&= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\} \cdot \exp\{sx - s\mu - \frac{1}{2}\sigma^2 s^2\} \\
&= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{\frac{-(x-\mu)^2 + 2\sigma^2 sx - 2\sigma^2 s\mu - \sigma^4 s^2}{2\sigma^2}\right\} \\
&= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{\frac{-(x-\mu-s\sigma^2)^2}{2\sigma^2}\right\}
\end{aligned}
$$

This is the probability density function of a $\mathrm{N}(\mu + s\sigma^2, \sigma)$ distribution.

A.1.3 Density for a Tilted Exponential Distribution

We know that the probability density function for the normal distribution is given by:
$f(x) = \lambda e^{-\lambda x}$

And that the moment generating function is $M(s) = \frac{\lambda}{\lambda - s},\ \text{for } s < \lambda$ from section A.1.1

Taking the natural logarithm of the moment generating function gives us the cumulative generating function:
$K(s) = \log\left(\frac{\lambda}{\lambda - s}\right), s < \lambda$

And so the density for the tilted exponential distribution is:
$\lambda \exp\{-\lambda x\} \cdot \exp\{sx - K(s)\}$

$= \lambda \exp\{-\lambda x\} \cdot \exp\{sx\} \cdot \dfrac{\lambda - s}{\lambda}$

$= (\lambda - s) \exp\{-x(\lambda - s)\}$

This is the probability density function of an Exponential$(\lambda - s)$ distribution.

A.2 The Saddlepoint Approximation is Equivalent to Finding a Normal Approximation to the Tilted Density

We have some density function $f(x)$. We will tilt this by some s_0 so that the mean of the tilted distribution is the value of x that we want to estimate the saddlepoint approximation at (i.e. x_0). In other words, s_0 has the property that $\int_{-\infty}^{\infty} f(x) \exp\{s_0 x - K(s_0)\} dx = x_0$.

Our tilted density has the probability density function $e^{s_0 x - K(s_0)} f(x)$. We are looking to approximate this with a normal density.

From section A.2.1, we have shown that the mean of the tilted density is $K'(s_0)$ and the variance is $K''(s_0)$. The normal approximation to the tilted distribution is therefore $N\left(K'(s_0), \sqrt{K''(s_0)}\right)$.

The height of a $N(\mu, \sigma)$ density is $\frac{1}{\sqrt{2\pi\sigma^2}}$. Therefore, on the tilted scale, at x_0, our approximation is $\frac{1}{\sqrt{2\pi K''(s_0)}}$.

If we put back the tilting factor and evaluate our approximation at x_0, it gives us a result of $\frac{1}{\sqrt{2\pi K''(s_0)}} e^{K(s_0)-s_0 x_0}$. This is exactly equivalent to our saddlepoint approximation at x_0.

A.2.1 Mean and Variance of the Tilted Density

Let's denote the cumulant generating function of the s_0-tilted density by $K_{s_0}(s)$.

Then

$$\begin{aligned} K_{s_0}(s) &= \log\left(\int_{-\infty}^{\infty} e^{sx}\left[e^{s_0 x - K(s_0)} f(x)\right]\, dx\right) \\ &= \log\left(\int_{-\infty}^{\infty} e^{(s+s_0)x} e^{-K(s_0)} f(x)\, dx\right) \\ &= \log\left(\int_{-\infty}^{\infty} e^{(s+s_0)x} f(x)\, dx\right) - K(s_0) \\ &= K(s+s_0) - K(s_0) \end{aligned}$$

The mean of a distribution is the same as the first cumulant, which is also the first derivative of the cumulant generating function evaluated at 0 [51]. And so the mean of the tilted distribution is

$$\frac{d}{ds} K_{s_0}(s)|_{s=0} = K'(s_0)$$

Similarly, the variance of a distribution is the same as the second cumulant, which is also the second derivative of the cumulant generating function evaluated at 0. And so the variance of the tilted distribution is

$$\frac{d^2}{ds^2} K_{s_0}(s)|_{s=0} = K''(s_0)$$

A.3 Finding the Distribution of the Mapped/Tilted Poisson Point Process

A.3.1 Campbell's Formula

Suppose P is a point process on $S = \mathbb{R}^d$ with intensity measure ν. For a function f mapping S to $\mathbb{R}$, then the following holds [18]:

The random sum

$$T_{f,P} := \sum_{x \in P} f(x)$$

is a random variable with expected value of

$$\mathbb{E}\left[T_{f,P}\right] = \int_S f(x)\, d\nu(x)$$

A.3.2 The Intensity function of Transformed Variables

Suppose we have a set of points P in two dimensional space. The x-coordinates of the points have some distribution $f(x)$, while the y-coordinates form a Poisson point process with rate 1. The x and y values are independent of each other.

Let's consider a mapping g that takes each coordinate $(x, y) \in P$ and maps it to $(\tilde{x}, \tilde{y}) \in \tilde{P}$

Note that mappings of Poisson point processes are also Poisson [18]; so $\tilde{P}$ is also a Poisson point process.

If we consider some function on (x, y), say h, we have:
$T_{h \circ g,P} = \sum_{x,y} h(g(x, y)) = \sum_{\tilde{x},\tilde{y}} h(\tilde{x}, \tilde{y}) = T_{h,\tilde{P}}$

Taking expectations, we can see from Campbell's theorem in section A.3.1 that:
$\mathbb{E}(T_{h \circ g,P}) = \int h(g(x, y)) d\nu_{X,Y}(x, y) = \int h(\tilde{x}, \tilde{y}) d\nu_{\tilde{X},\tilde{Y}}(\tilde{x}, \tilde{y}) = \mathbb{E}(T_{h,\tilde{P}})$

We are interested in finding the intensity measure of the new Poisson point process, i.e. we want to find $\nu_{\tilde{X},\tilde{Y}}$

Now, this can be done more generally, but we are interested in the case that $g : (x, y) \to (x, ye^{-sx})$, i.e. $(\tilde{x}, \tilde{y}) = (x, ye^{-sx})$

We have both:

1. $\tilde{y} = ye^{-sx} \implies \frac{d\tilde{y}}{dy} = e^{-sx} \implies dy = d\tilde{y}e^{sx}$

2. P has the intensity measure $f(x)dydx$ - this is contingent on the way we constructed our x and y-values. That is, $d\nu_{X,Y}(x, y) = f(x)dydx$.

Using the two points above, we have the equality:

$$\int\int h(x, ye^{-sx}) f(x)\, dy\, dx = \int\int h(\tilde{x}, \tilde{y}) f(\tilde{x}) e^{s\tilde{x}}\, d\tilde{y}\, d\tilde{x} = \int h(\tilde{x}, \tilde{y}) d\nu_{\tilde{X},\tilde{Y}}(\tilde{x}, \tilde{y})$$

We can read off that the intensity function of our transformed variables is $f(x)e^{sx}$.

A.3.3 Ensuring that the Rate of the Transformed Process Remains Constant

Now, suppose that instead we have $g : (x, y) \to (x, yce^{-sx})$ for some constant c. We want to select c such that the rate of points in our transformed Poisson process $\tilde{P}$ per unit of y

axis is 1, the same as the rate for the untransformed process P.

By the same logic as we've seen above, the intensity for the transformed process is $f(x)c^{-1}e^{sx}$

Having the rate per unit y axis equal to 1 means that we require $\int f(x)c^{-1}e^{sx} = 1$ This implies that $c = \int f(x)e^{sx}$, or $c = \mathbb{E}[e^{sX}] = M(s)$

So in order to achieve our goal of having an unchanged rate on the y-axis, our transformation should be $g : (x, y) \to (x, M(s)ye^{-sx}) = (x, ye^{K(s)-sx})$.

A.4 Mapping Is Equivalent To Tilting

Recall that we have a mapping which multiplies each element of a Poisson point process with rate 1 by $e^{K(s)-sx}$, where x has some density function $f(x)$ (independent of the Poisson process) and cumulant generating function $K(s)$. We have seen from appendix A.3 that applying this mapping results in a density function of $f(x)e^{sx-K(s)}$.

This is exactly the same density function as the s-tilted density.

In the case of the standard normal distribution, we have seen from A.1.2 that tilting a $\mathrm{N}(\mu, \sigma)$ distribution gives us a $\mathrm{N}(\mu + s\sigma^2, \sigma)$ distribution. For a standard normal with $\mu = 0$ and $\sigma = 1$, tilting gives us a $\mathrm{N}(s, 1)$ distribution. This is exactly what we can see in our simulations in 2.6; applying the mapping gives us a distribution on the x-axis identical to a $\mathrm{N}(s, 1)$ distribution.

Bibliography

[1] H. E. Daniels, "Saddlepoint approximations in statistics," *The Annals of Mathematical Statistics*, vol. 25, no. 4, pp. 631 – 650, 1954. https://doi.org/10.1214/aoms/1177728652.

[2] H. E. Daniels, "Exact saddlepoint approximations," *Biometrika*, vol. 67, no. 1, p. 59–63, 1980.

[3] R. W. Butler, *Saddlepoint Approximations with Applications*. Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press, 2007.

[4] Y. K. Kwok and W. Zheng, *Saddlepoint Approximation Methods in Financial Engineering*. Springer International Publishing, 2018.

[5] E. Derezea, A. Kume, and D. Froebrich, "An application of saddlepoint approximation for period detection of stellar light observations," *arxiv.org*, Jan 2022. https://arxiv.org/abs/2201.11762.

[6] O. Hyrien, R. Chen, M. Mayer-Pröschel, and M. Noble, "Saddlepoint approximations to the moments of multitype age-dependent branching processes, with applications," *Biometrics*, vol. 66, p. 567–577, Jun 2009. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2888915/.

[7] J. Goodman, "How accurate is the saddlepoint approximation for MLEs?." https://sites.google.com/view/apsps/previous-speakers.

[8] R Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. http://www.R-project.org/.

[9] W. Chang, J. Cheng, J. Allaire, C. Sievert, B. Schloerke, Y. Xie, J. Allen, J. McPherson, A. Dipert, and B. Borges, *shiny: Web Application Framework for R*, 2021. Available at https://CRAN.R-project.org/package=shiny, R package version 1.7.1.

[10] A. Kim, "Moment-generating functions explained." https://towardsdatascience.com/moment-generating-function-explained-27821a739035, 2019. Towards Data Science.

[11] E. W. Weisstein, "Raw moment." https://mathworld.wolfram.com/RawMoment.html. MathWorld–A Wolfram Web Resource.

[12] E. W. Weisstein, "Moment-generating functions." https://mathworld.wolfram.com/Moment-GeneratingFunction.html. MathWorld–A Wolfram Web Resource.

[13] P. Kempthorne, "Exponential families," MIT OpenCourseWare, 2016. https://ocw.mit.edu/courses/mathematics/18-655-mathematical-statistics-spring-2016/lecture-notes/.

[14] Wikipedia contributors, "Natural exponential family — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Natural_exponential_family&oldid=1078195477, 2022.

[15] Wikipedia contributors, "Exponential tilting — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Exponential_tilting&oldid=1098562094, 2022.

[16] Y. Zhang and Y. Kuen Kwok, "Saddlepoint approximations to tail expectations under non-gaussian base distributions: option pricing applications," *Journal of Applied Statistics*, vol. 47, p. 1936–1956, Dec 2019.

[17] A. Hankin, "Modelling terrorism incidents in iraq as a log-gaussian cox process." https://github.com/alicemhankin/inlabru-terrorism.

[18] W. Weil, ed., *Spatial Point Processes and their Applications*, pp. 1–75. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. https://doi.org/10.1007/978-3-540-38175-4_1.

[19] P. Murrell, "Paul Murrell's Home Page." https://www.stat.auckland.ac.nz/~paul/.

[20] K. J. Healy and P. U. Press, *Data visualization : a practical introduction*. Princeton University Press, 2019.

[21] C. Chapman, "Exploring the gestalt principles of design." https://www.toptal.com/designers/ui/gestalt-principles-of-design, 2018.

[22] Y. Holtz and C. Healy, "The issue with pie charts." https://www.data-to-viz.com/caveat/pie.html. From Data To Viz.

[23] B. Marr, "Why you shouldn't use pie charts in your dashboards and performance reports." https://bernardmarr.com/why-you-shouldnt-use-pie-charts-in-your-dashboards-and-performance-reports/.

[24] W. S. Cleveland and R. McGill, "Graphical perception: The visual decoding of quantitative information on graphical displays of data," *Journal of the Royal Statistical Society. Series A (General)*, vol. 150, no. 3, pp. 192–229, 1987. http:

//www.jstor.org/stable/2981473.

[25] G. Reynolds, *Presentation zen: Simple ideas on presentation design and Delivery*, ch. 6, p. 152–163. New Riders Pub., 2008.

[26] A. Heiss, "Graphic design," Andrew Young School of Policy Studies, 2020. https://datavizm20.classes.andrewheiss.com/slides/02-slides.html.

[27] K. Bentle, "Why your edtech product needs a data visualization style guide." https://openfieldx.com/data-visualization-style-guide-how-to-edtech-software/.

[28] P. Murrell, "Graphic design." https://www.stat.auckland.ac.nz/~paul/stats787/2022/Topics/design.html.

[29] C. O. Wilke, *Fundamentals of data visualization : a primer on making informative and compelling figures*. O'Reilly Media, Inc., 2019.

[30] A. Kosari, "Colorblind people population! statistics." https://www.colorblindguide.com/post/colorblind-people-population-live-counter, Jun 2021.

[31] K. Ito and M. Okabe, "Color universal design (cud)." http://jfly.iam.u-tokyo.ac.jp/color/#pallet, Sep 2008. J*FLY data depository.

[32] M. Mol, "A color-safe palette." https://mikemol.github.io/technique/colorblind/2018/02/11/color-safe-palette.html, Feb 2018. Mike's Notes.

[33] M. C. Stone, D. A. Szafir, and V. Setlur, "An engineering model for color difference as a function of size," 2014.

[34] M. Dubel, "Shiny, Tableau, and PowerBI: Better Business Intelligence." https://www.rstudio.com/blog/shiny-tableau-and-powerbi-better-business-intelligence/, 2019. R Studio Blog.

[35] W3schools, "Css introduction." https://www.w3schools.com/css/css_intro.asp, 2019.

[36] MDN Contributors, "Html basics." https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics, Apr 2019.

[37] A. Clausen and S. Sokol, *Deriv: R-based Symbolic Differentiation*, 2020. Deriv package version 4.1 https://CRAN.R-project.org/package=Deriv.

[38] W. Chang and B. Borges Ribeiro, *shinydashboard: Create Dashboards with 'Shiny'*, 2021. R package version 0.7.2 https://CRAN.R-project.org/package=shinydashboard.

[39] G. W. Granger, "Aesthetic measure applied to color harmony: An experimental test," *The Journal of General Psychology*, vol. 52, no. 2, pp. 205–212, 1955. https://doi.org/10.1080/00221309.1955.9920239.

[40] A. Zeileis, J. C. Fisher, K. Hornik, R. Ihaka, C. D. McWhite, P. Murrell, R. Stauffer, and C. O. Wilke, "colorspace: A toolbox for manipulating and assessing colors and palettes," *Journal of Statistical Software*, vol. 96, no. 1, pp. 1–49, 2020.

[41] J. Cheng, C. Sievert, B. Schloerke, W. Chang, Y. Xie, and J. Allen, *htmltools: Tools for HTML*, 2022. R package version 0.5.3 https://CRAN.R-project.org/package=htmltools.

[42] "Shiny widget gallery." https://shiny.rstudio.com/gallery/widget-gallery.html. Shiny from R Studio.

[43] A. Mikael, "Why numeric sliders are lazy ux." https://blog.prototypr.io/numeric-sliders-are-lazy-ux-46d685a00d62, 2016. Prototypr.

[44] V. Perrier, F. Meyer, and D. Granjon, *shinyWidgets: Custom Inputs Widgets for Shiny*, 2022. R package version 0.7.0 https://CRAN.R-project.org/package=shinyWidgets.

[45] V. Perrier, "Extend widgets available in shiny." https://dreamrs-vic.shinyapps.io/shinyWidgets/.

[46] C. Sievert, *Interactive Web-Based Data Visualization with R, plotly, and shiny*. Chapman and Hall/CRC, 2020. https://plotly-r.com.

[47] S. Arya, D. Mount, S. E. Kemp, and G. Jefferis, *RANN: Fast Nearest Neighbour Search (Wraps ANN Library) Using L2 Metric*, 2019. R package version 2.6.1 https://CRAN.R-project.org/package=RANN.

[48] A. Genz, F. Bretz, T. Miwa, X. Mi, F. Leisch, F. Scheipl, and T. Hothorn, *mvtnorm: Multivariate Normal and t Distributions*, 2021. R package version 1.1-3 https://CRAN.R-project.org/package=mvtnorm.

[49] T. Linder, "Moment generating functions and multivariate normal distribution," Queen's University, 2017. https://mast.queensu.ca/~stat353/slides/5-multivariate_normal17_4.pdf.

[50] C. C. Y. Wang, "Basic statistics and probability for econometrics," Scholars at Havard. https://scholar.harvard.edu/files/charlescywang/files/basic_statistics_and_probability_for_econometrics_econ_270a.pdf.

[51] Wikipedia contributors, "Cumulant — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Cumulant&oldid=1099279214, 2022.