Computational Statistics with R

Niels Richard Hansen

2021-11-08, Git version: c532a37

Contents

Preface

This is a draft of a book on computational statistics, which is being developed specifically for the master's education in statistics at University of Copenhagen.

A solid mathematical background is assumed throughout, while the computer science prerequisites are more modest. To be specific, the reader is expected to have a reasonable command of mathematical analysis, linear algebra and mathematical statistics, as exemplified by maximum likelihood estimation of multivariate parameters and asymptotic properties of multivariate estimators. The reader is expected to have an understanding of what an algorithm is, how numerical computations differ from symbolic computations, and be able to write small computer programs.

The intention of the material is to serve as a pedagogical introduction to computational statistics. No claim is made that the material is comprehensive, nor does it purport computational statistics as a single coherent field with a unifying theoretical foundation. The presentation is driven by statistical examples with the unifying theme being an experimental approach to solving computational problems in statistics.

Contemporary challenges in computational statistics revolve around large scale computations, either because the amount of data is massive or because we want to apply ever more complicated and sophisticated models and methods for the analysis and visualization of data. The examples treated are all of a rather modest complexity compared to these challenges. This is deliberate! A solid understanding of how to solve simpler problems is seen as a prerequisite for solving complex problems. It is the hope that this material provides the reader with a foundation in computational statistics that will subsequently make him or her able to develop good solutions to problems in computational statistics.

The book is based on R for several reasons. First of all, the target audience of statisticians is expected to be familiar with R, and they should learn how to use their programming language in an optimal way. This includes knowledge of the infrastructure supported by R and RStudio for supporting good software development and for testing, benchmarking and profiling code. In addition, this infrastructure combined with R Markdown and bookdown makes it a bliss to write a book that systematically integrates code and software development with the theory. Finally, though the reference CRAN implementation of R itself is not a fast interpreter, it is possible to write performant code by a proper use of R as a high-level programming language or by interfacing compiled code via the Rcpp package. Statisticians who use R should master these skills.

Chapter 1

Introduction

Computational statistics is about turning theory and methods into algorithms and actual numerical computations with data. It is about solving real computational problems that arise when we visualize, analyze and model data.

Computational statistics is not a single coherent topic but a large number of vaguely related computational techniques that we use in statistics. This book is not attempting to be comprehensive. Instead, a few selected statistical topics and methodologies are treated in some detail with the intention that good computational practice can be learned from these topics and transferred to other statistical methodologies as needed. Though the topics are arguably fundamental, they reflect the knowledge and interests of the author, and different topics could clearly have been chosen.

The demarcation line between statistical methodology and computational statistics is, moreover, blurred. Most methodology involves mathematical formulas and even algorithms for computing estimates and other statistics of interest from data, or for evaluating probabilities or integrals numerically or via simulations. In this book, the transition from methodology to computational statistics happens when the methodology is to be implemented. That is, when formulas, algorithms and pseudo code are transformed into actual code and statistical software. It is during this transition that a number of practical challenges reveal themselves, such as actual run time and memory usage, and the limitations of finite precision arithmetic. And when dealing with actual implementations we learn to appreciate the value of an approximate solution that may be theoretically suboptimal but sufficiently accurate for any practical purpose.

Statistical software development also requires some basic software engineering skills and knowledge of the most common programming paradigms. Implementing a single algorithm for a specific problem is one thing, but developing a piece of statistical software for others to use is something quite different. This book is *not* an introduction to statistical software development as such, but the process of developing good software plays a role throughout. Good implementations are not presented as code that manifests itself from divine insight, but rather as code that is derived through experimental and analytic cycles – somewhat resembling how software development actually takes place.

There is a notable practical and experimental component to software development. However important theoretical considerations are regarding correctness and complexity of algorithms, say, the actual code has to strike a balance between generality, readability, efficiency, accuracy, ease of usage and ease of development among other things. Finding a good balance requires that one is able to reason about benefits and deficiencies of different implementations. It is an

important point of this book that such reasoning should rely on experiments and empirical facts and not speculations.

R and RStudio is used throughout, and the reader is expected to have some basic knowledge of R programming. While RStudio is not a requirement for most of the book, it is a recommendable IDE (integrated development environment) for R, which offers a convenient framework for developing, benchmarking, profiling, testing, documenting and experimenting with statistical software. Appendix A covers some basic and important aspects of R programming and can serve as a survey and quick reference. For an in-depth treatment of R as a programming language the reader is referred to Advanced R by Hadley Wickham. In fact, direct references to that book are given throughout for detailed explanations of many R programming language concepts, while Appendix @ref{app-R} contains a brief overview of core R concepts used.

This book is organized into three parts on smoothing, Monte Carlo methods and optimization. Each part is introduced in the following three sections to give the reader an overview of the topics covered, how they are related to each other and how they are related to some main trends and challenges in contemporary computational statistics. In this introduction, several R functions from various packages are used to illustrate how smoothing, simulation of random variables and optimization play roles in statistics. Thus the introduction relies largely on already implemented solutions, and some data analysts will never want to move beyond that use of R. However, the remaining part of the book is written for those who want to move on and learn how to develop their own solutions and not just how to use interfaces to the plethora of already existing implementations.

1.1 Smoothing

Smoothing is a descriptive statistical tool for summarizing data, a practical visualization technique, as well as a nonparametric estimation methodology. The basic idea is that data is representative of an underlying distribution with some smoothness properties, and we would like to approximate or estimate this underlying distribution from data.

There are two related but slightly different approaches. Either we attempt to estimate a smooth density of the observed variables, or we attempt to estimate a smooth conditional density of one variable given others. The latter can in principle be done by computing the conditional density from a smooth estimate of the joint density. Thus it appears that we really just need a way of computing smooth density estimates. In practice it may, however, be better to solve the conditional smoothing problem directly instead of solving a strictly more complicated problem. This is particularly so, if the conditioning variables are fixed e.g. by a design, or if our main interest is in the conditional mean or median, say, and not the entire conditional distribution. Conditional smoothing is dealt with in Chapter 3.

In this introduction we focus on the univariate case, where there really only is one problem: smooth density estimation. Moreover, this is a very basic problem, and one viewpoint is that we simply need to "smooth out the jumps of the histogram". Indeed, it does not need to be made more sophisticated than that! Humans are able to do this quite well using just a pen and a printed histogram, but it is a bit more complicated to automatize such a smoothing procedure. Moreover, an automatized procedure is likely to need calibration to yield a good tradeoff between smoothness and data fit. This is again something that humans can do quite well by eyeballing visualizations, but that approach does not scale, neither in terms of the number of density estimates we want to consider, nor in terms of going from univariate to multivariate densities.

If we want to really discuss how a smoothing procedure works not just as a heuristic but

also as an estimator of an underlying density, it is necessary to formalize how to quantify the performance of the procedure. This increases the level of mathematical sophistication, but it allows us to discuss optimality, and it lets us develop fully automatized procedures that do not rely on human calibration. While human inspection of visualizations is always a good idea, computational statistics is also about offloading humans from all computational tasks that can be automatized. This is true for smoothing as well, hence the need for automatic and robust smoothing procedures that produce well calibrated results with a minimum of human effort.

1.1.1 Angle distributions in proteins

We will illustrate smoothing using a small data set on angles formed between two subsequent peptide planes in 3D protein structures. This data set is selected because the angle distributions are multimodal and slightly non-standard, and these properties are well suited for illustrating fundamental considerations regarding smooth density estimation in practice.

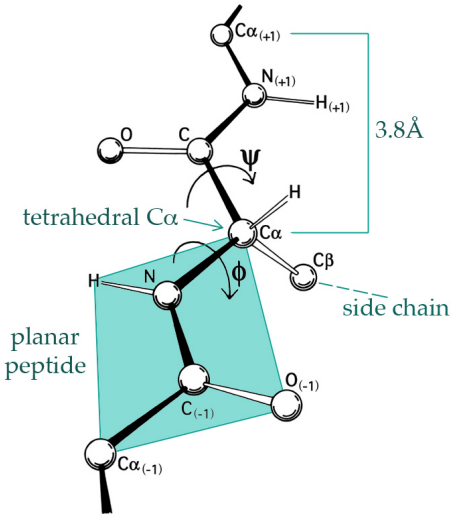
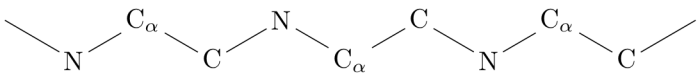


Figure 1.1: The 3D structure of proteins is largely given by the ϕ- and ψ-angles of the peptide planes. (By Dcrjsr, CC BY 3.0 via Wikimedia Commons.)

A protein is a large molecule consisting of a backbone with carbon and nitrogen atoms arranged sequentially:



A hydrogen atom binds to each nitrogen (N) and an oxygen atom binds to each carbon without the α subscript (C), see Figure 1.1, and such four atoms form together what is known as a peptide bond between two alpha-carbon atoms (C_α). Each C_α atom binds a hydrogen atom and an amino acid *side chain*. There are 20 naturally occurring amino acids in genetically encoded proteins, each having a three letter code (such as Gly for Glycine, Pro for Proline, etc.). The protein will typically form a complicated 3D structure determined by the amino acids, which in turn determine the ϕ- and the ψ-angles between the peptide planes as shown on Figure 1.1.

We will consider a small data set, `phipsi`, of experimentally determined angles from a single protein, the human protein 1HMP, which is composed of two chains (denoted A and B). Figure 1.2 shows the 3D structure of the protein.

```
head(phipsi)
```

```
##   chain  AA pos        phi        psi
## 1     A Pro   5 -1.6218794  0.2258685
## 2     A Gly   6  1.1483709 -2.8314426
## 3     A Val   7 -1.4160220  2.1190570
## 4     A Val   8 -1.4926720  2.3941331
## 5     A Ile   9 -2.1814653  1.4877618
## 6     A Ser  10 -0.7525375  2.5676186
```

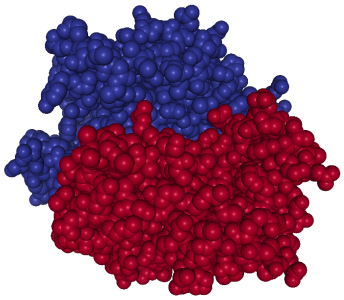


Figure 1.2: The 3D structure of the atoms constituting the protein 1HMP. The colors indicate the two different chains.

We can use base R functions such as `hist` and `density` to visualize the marginal distributions of the two angles.

```
hist(phipsi$phi, prob = TRUE)
rug(phipsi$phi)
density(phipsi$phi) %>% lines(col = "red", lwd = 2)

hist(phipsi$psi, prob = TRUE)
rug(phipsi$psi)
density(phipsi$psi) %>% lines(col = "red", lwd = 2)
```

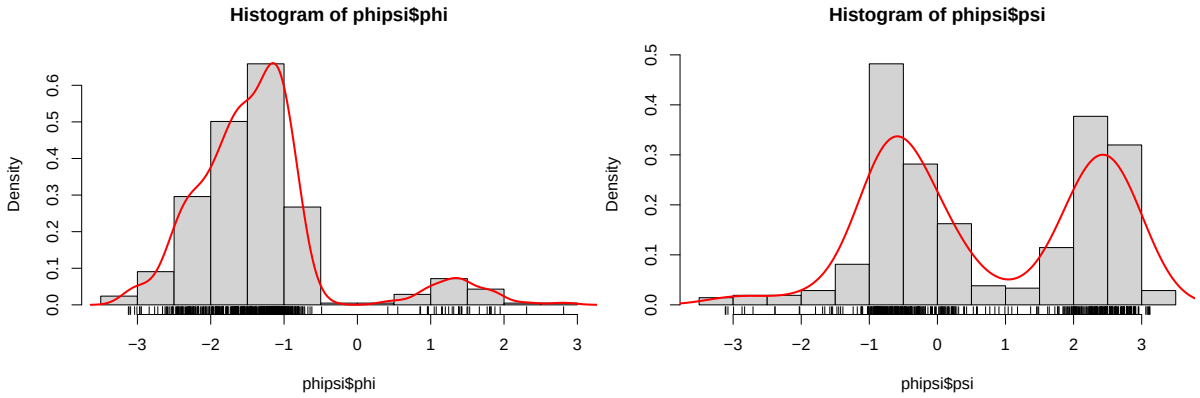


Figure 1.3: Histograms equipped with a rug plot and smoothed density estimate (red line) of the distribution of ϕ-angles (left) and ψ-angles (right).

The smooth red density curve shown in Figure 1.3 can be thought of as a smooth version of a histogram. It is surprisingly difficult to find automatic smoothing procedures that perform uniformly well – it is even quite difficult to automatically select the number and positions of the breaks used for histograms. This is one of the important points that is taken up in this book: how to implement good default choices of various *tuning parameters* that are required by

any smoothing procedure.

1.1.2 Using ggplot2

It is also possible to use `ggplot2` to achieve similar results.

```
library(ggplot2)
ggplot(phipsi, aes(x = phi)) +
  geom_histogram(aes(y = ..density..), bins = 13) +
  geom_density(col = "red", size = 1) +
  geom_rug()

ggplot(phipsi, aes(x = psi)) +
  geom_histogram(aes(y = ..density..), bins = 13) +
  geom_density(col = "red", size = 1) +
  geom_rug()
```

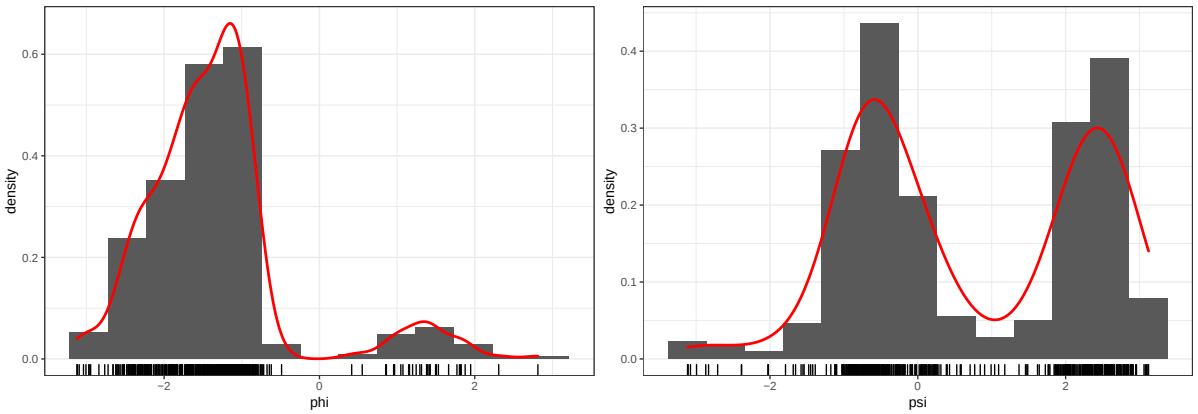


Figure 1.4: Histograms and density estimates of ϕ-angles (left) and ψ-angles (right) made with ggplot2.

Histograms produced by ggplot2 have a non adaptive default number of bins equal to 30 (number of breaks equal to 31), which is different from `hist` that uses Sturges' formula

$$\text{number of breaks} = \lceil \log_2(n) + 1 \rceil$$

with n the number of observations in the data set. In addition, this number is further modified by the function `pretty` that generates "nice" breaks, which results in 14 breaks for the angle data. For easier comparison, the number of bins used by `geom_histogram` above is set to 13, though it should be noticed that the breaks are not chosen in exactly the same way by `geom_histogram` and `hist`. Automatic and data adaptive bin selection is difficult, and `geom_histogram` implements a simple and fixed, but likely suboptimal, default while notifying the user that this default choice can be improved by setting `binwidth`.

For the density, `geom_density` actually relies on the `density` function and its default choices of how and how much to smooth. Thus the figure may have a slightly different appearance, but the estimated density obtained by `geom_density` is identical to the one obtained by `density`.

1.1.3 Changing the defaults

The range of the angle data is known to be $(-\pi, \pi]$, which neither the histogram nor the density smoother take advantage of. The `pretty` function, used by `hist` chooses, for instance,

breaks in -3 and 3, which results in the two extreme bars in the histogram to be misleading. Note also that for the ψ-angle it appears that the defaults result in oversmoothing of the density estimate. That is, the density is more smoothed out than the data (and the histogram) appears to support.

To obtain different – and perhaps better – results, we can try to change some of the defaults of the histogram and density functions. The two most important defaults to consider are the *bandwidth* and the *kernel*. Postponing the mathematics to Chapter 2, the kernel controls how neighboring data points are weighted relatively to each other, and the bandwidth controls the size of neighborhoods. A bandwidth can be specified manually as a specific numerical value, but for a fully automatic procedure, it is selected by a bandwidth selection algorithm. The `density` default is a rather simplistic algorithm known as Silverman's rule-of-thumb.

```
hist(phipsi$psi, breaks = seq(-pi, pi, length.out = 15), prob = TRUE)
rug(phipsi$psi)
density(phipsi$psi, adjust = 1, cut = 0) %>% lines(col = "red", lwd = 2)
density(phipsi$psi, adjust = 0.5, cut = 0) %>% lines(col = "blue", lwd = 2)
density(phipsi$psi, adjust = 2, cut = 0) %>% lines(col = "purple", lwd = 2)

hist(phipsi$psi, breaks = seq(-pi, pi, length.out = 15), prob = TRUE)
rug(phipsi$psi)
# Default kernel is "gaussian"
density(phipsi$psi, bw = "SJ", cut = 0) %>% lines(col = "red", lwd = 2)
density(phipsi$psi, kernel = "epanechnikov", bw = "SJ", cut = 0) %>%
  lines(col = "blue", lwd = 2)
density(phipsi$psi, kernel = "rectangular", bw = "SJ", cut = 0) %>%
  lines(col = "purple", lwd = 2)
```

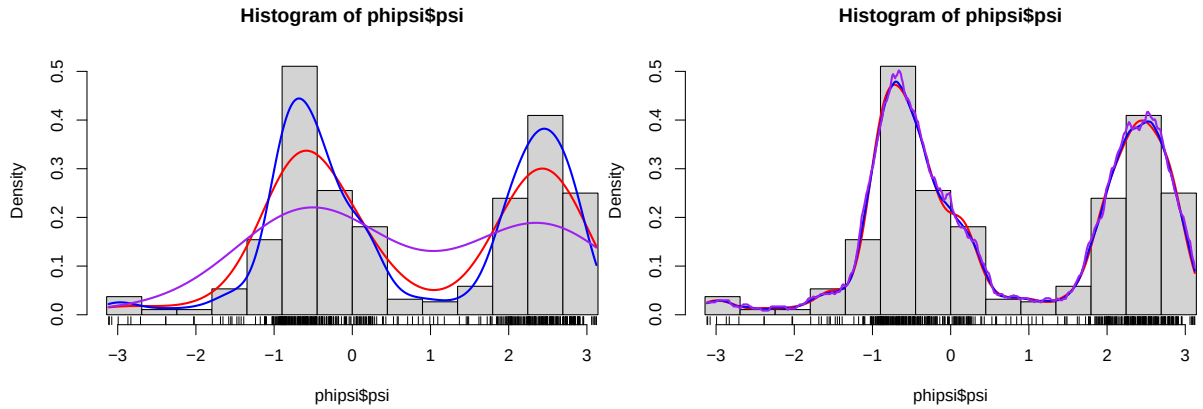


Figure 1.5: Histograms and various density estimates for the ψ-angles. The colors indicate different choices of bandwidth adjustments using the otherwise default bandwidth selection (left) and different choices of kernels using Sheather-Jones bandwidth selection (right).

Figure 1.5 shows examples of several different density estimates that can be obtained by changing the defaults of `density`. The breaks for the histogram have also been chosen manually to make sure that they match the range of the data. Note, in particular, that Sheather-Jones bandwidth selection appears to work better than the default bandwidth for this example. This is generally the case for multimodal distributions, where the default bandwidth tends to oversmooth. Note also that the choice of bandwidth is far more consequential than the choice of kernel, the latter mostly affecting how wiggly the density estimate is locally.

It should be noted that defaults arise as a combination of historically sensible choices and

backward compatibility. Thought should go into choosing a good, robust default, but once a default is chosen, it should not be changed haphazardly, as this might break existing code. That is why not all defaults used in R are by today's standards the best known choices. You see this argument made in the documentation of `density` regarding the default for bandwidth selection, where Sheather-Jones is suggested as a better default than the current, but for compatibility reasons Silverman's rule-of-thumb is the default and is likely to remain being so.

1.1.4 Multivariate methods

This section provides a single illustration of how to use the bivariate kernel smoother `kde2d` from the MASS package for bivariate density estimation of the (ϕ, ψ)-angle distribution. A scatter plot of ϕ and ψ angles is known as a Ramachandran plot, and it provides a classical and important way of visualizing local structural constraints of proteins in structural biochemistry. The density estimate can be understood as an estimate of the distribution of (ϕ, ψ)-angles in naturally occuring proteins from the small sample of angles in our data set.

We compute the density estimate in a grid of size 100 by 100 using a bandwidth of 2 and using the `kde2d` function that uses a bivariate normal kernel.

```
denshat <- MASS::kde2d(phipsi$phi, phipsi$psi, h = 2, n = 100)
```

```
denshat <- data.frame(
  cbind(
    denshat$x,
    rep(denshat$y, each = length(denshat$x)),
    as.vector(denshat$z)
    )
)
```

```
colnames(denshat) <- c("phi", "psi", "dens")
p <- ggplot(denshat, aes(phi, psi)) +
  geom_tile(aes(fill = dens), alpha = 0.5) +
  geom_contour(aes(z = sqrt(dens))) +
  geom_point(data = phipsi, aes(fill = NULL)) +
  scale_fill_gradient(low = "white", high = "darkblue", trans = "sqrt")
```

We then recompute the density estimate in the same grid of size using the smaller bandwidth of 0.5.

```
denshat <- MASS::kde2d(phipsi$phi, phipsi$psi, h = 0.5, n = 100)
```

The Ramachandran plot in Figure 1.6 shows how structural constraints of a protein, such as steric effects, induce a non-standard bivariate distribution of (ϕ, ψ)-angles.

1.1.5 Large scale smoothing

With small data sets of less than 10,000 data points, say, univariate smooth density estimation requires a very modest amount of computation. That is true even with rather naive implementations of the standard methods. The R function `density` is implemented using a number of computational tricks like binning and the fast Fourier transform, and it can compute density estimates with a million data points (around 8 MB) within a fraction of a second.

It is unclear if we ever need truly large scale *univariate* density estimation with terabytes of data points, say. If we have that amount of (heterogeneous) data it is likely that we are better off breaking the data down into smaller and more homogeneous groups. That is, we should

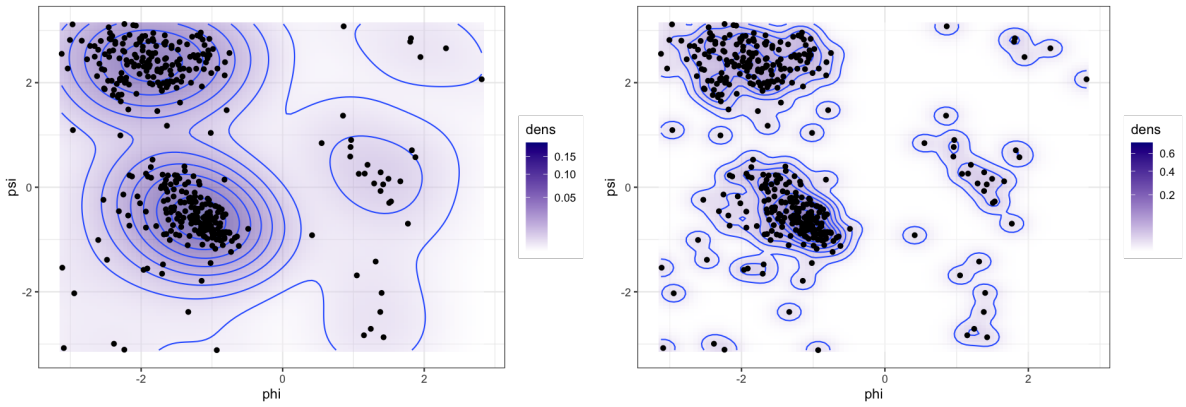


Figure 1.6: Bivariate density estimates of protein backbone angles using a bivariate Gaussian kernel with bandwiths 2 (left) and 0.5 (right).

turn a big data computation into a large number of small data computations. That does not remove the computational challenge but it does diminish it somewhat e.g. by parallelization.

Deng and Wickham did a review in 2011 on Density estimation in R, where they assessed the performance of a number of R packages including the `density` function. The KernSmooth package was singled out in terms of speed as well as accuracy for computing *smooth* density estimates with `density` performing quite well too. (Histograms are non-smooth density estimates and generally faster to compute). The assessment was based on using defaults for the different packages, which is meaningful in the sense of representing the performance that the occasional user will experience. It is, however, also an evaluation of the combination of default choices and the implementation, and as different packages rely on e.g. different bandwidth selection algorithms, this assessment is not the complete story. The `bkde` function from the KernSmooth package, as well as `density`, are solid choices, but the point is that performance assessment is a multifaceted problem.

To be a little more specific about the computational complexity of density estimators, suppose that we have n data points and want to evaluate the density in m points. A naive implementation of kernel smoothing, Section 2.2, has $O(mn)$ time complexity, while a naive implementation of the best bandwidth selection algorithms have $O(n^2)$ time complexity. As a simple rule-of-thumb, anything beyond $O(n)$ will not scale to very large data sets. A quadratic time complexity for bandwidth selection will, in particular, be a serious bottleneck. Kernel smoothing illustrates perfectly that a literal implementation of the mathematics behind a statistical method may not always be computationally viable. Even the $O(mn)$ time complexity may be quite a bottleneck as it reflects mn kernel evaluations, each being potentially a computationally relatively expensive operation.

The binning trick, with the number of bins set to m, is a grouping of the data points into m sets of neighbor points (bins) with each bin representing the points in the bin via a single point and a weight. If $m \ll n$, this can reduce the time complexity substantially to $O(m^2) + O(n)$. The fast Fourier transform may reduce the $O(m^2)$ term even further to $O(m \log(m))$. Some approximations are involved, and it is of importance to evaluate the tradeoff between time and memory complexity on one side and accuracy on the other side.

Multivariate smoothing is a different story. While it is possible to generalize the basic ideas of univariate density estimation to arbitrary dimensions, the curse-of-dimensionality hits unconstrained smoothing hard – statistically as well as computationally. Multivariate smoothing is therefore still an active research area developing computationally tractable and

novel ways of fitting smooth densities or conditional densities to multivariate or even high-dimensional data. A key technique is to make structural assumptions to alleviate the challenge of a large dimension, but there are many different assumptions possible, which makes the body of methods and theory richer and the practical choices much more difficult.

1.2 Monte Carlo methods

Broadly speaking, Monte Carlo methods are computations that rely on some form of random input in order to carry out a computation. The actual random input will be generated by a (pseudo)random number generator according to some distributional specifications, and the precise value of the computation will depend on the precise value of the random input but in a way where we understand the magnitude of the dependence quite well. In most cases we can make the dependence diminish by increasing the amount of random input.

In statistics it is quite interesting in itself that we can make the computer simulate data so that we can draw an example data set from a statistical model. However, the real usage of such simulations is almost always as a part of a Monte Carlo computation, where we repeat the simulation of data sets a large number of times and compute distributional properties of various statistics. This is just one of the most obvious applications of Monte Carlo methods in statistics and there are many others. Disregarding the specific computation of interest in applications, a core problem of Monte Carlo methods is efficient simulation of random variables from a given target distribution.

1.2.1 Univariate von Mises distributions

We will exemplify Monte Carlo computations by considering angle distributions just as in Section 1.1. The angles take values in the interval $(-\pi, \pi]$, and we will consider models based on the von Mises distribution on this interval, which has density

$$f(x) = \frac{1}{\varphi(\theta)} e^{\theta_1 \cos(x) + \theta_2 \sin(x)}$$

for $\theta = (\theta_1, \theta_2)^T \in \mathbb{R}^2$. A common alternative parametrization is obtained by introducing $\kappa = \|\theta\|_2 = \sqrt{\theta_1^2 + \theta_2^2}$, and (whenever $\kappa \neq 0$) $\nu = \theta/\kappa = (\cos(\mu), \sin(\mu))^T$ for $\mu \in (-\pi, \pi]$. Using the (κ, μ)-parametrization the density becomes

$$f(x) = \frac{1}{\varphi(\kappa\nu)} e^{\kappa \cos(x-\mu)}.$$

The former parametrization in terms of θ is, however, the canonical parametrization of the family of distributions as an exponential family, which is particularly useful for various likelihood estimation algorithms. The normalization constant

$$\begin{aligned} \varphi(\kappa\nu) &= \int_{-\pi}^{\pi} e^{\kappa \cos(x-\mu)} \mathrm{d}x \\ &= 2\pi \int_0^1 e^{\kappa \cos(\pi x)} \mathrm{d}x = 2\pi I_0(\kappa) \end{aligned}$$

is given in terms of the modified Bessel function I_0. We can easily compute and plot the density using R's `besselI` implementation of the modified Bessel function.

```
phi <- function(k) 2 * pi * besselI(k, 0)
curve(exp(cos(x)) / phi(1), -pi, pi, lwd = 2, ylab = "density", ylim = c(0, 0.52))
curve(exp(2 * cos(x - 1)) / phi(2), col = "red", lwd = 2, add = TRUE)
curve(exp(0.5 * cos(x + 1.5)) / phi(0.5), col = "blue", lwd = 2, add = TRUE)
```

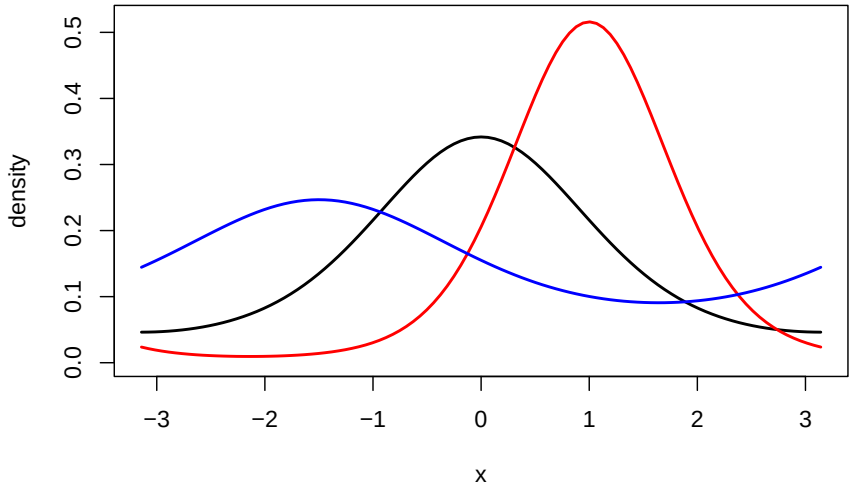


Figure 1.7: Density for the von Mises distribution with parameters $\kappa = 1$ and $\nu = 0$ (black), $\kappa = 2$ and $\nu = 1$ (red), and $\kappa = 0.5$ and $\nu = -1.5$ (blue).

It is not entirely obvious how we should go about simulating data points from the von Mises distribution. It will be demonstrated in Section 4.3 how to implement a *rejection sampler*, which is one useful algorithm for simulating samples from a distribution with a density.

In this section we simply use the `rmovMF` function from the `movMF` package, which implements a few functions for working with (finite mixtures of) von Mises distributions, and even the general von Mises-Fisher distributions that are generalizations of the von Mises distribution to p-dimensional unit spheres.

```
library("movMF")
xy <- rmovMF(500, 0.5 * c(cos(-1.5), sin(-1.5)))
# rmovMF represents samples as elements on the unit circle
x <- acos(xy[, 1]) * sign(xy[, 2])
```

```
hist(x, breaks = seq(-pi, pi, length.out = 15), prob = TRUE)
rug(x)
density(x, bw = "SJ", cut = 0) %>% lines(col = "red", lwd = 2)
curve(exp(0.5 * cos(x + 1.5)) / phi(0.5), col = "blue", lwd = 2, add = TRUE)
```

1.2.2 Mixtures of von Mises distributions

The von Mises distributions are unimodal distributions on $(-\pi, \pi]$. Thus to find a good model of the bimodal angle data, say, we have do move beyond these distributions. A standard approach for constructing multimodal distributions is as *mixtures* of unimodal distributions. A mixture of two von Mises distributions can be constructed by flipping a (biased) coin to decide which of the two distributions to sample from. We will use the exponential family parametrization in the following.

```
thetaA <- c(3.5, -2)
thetaB <- c(-4.5, 4)
```

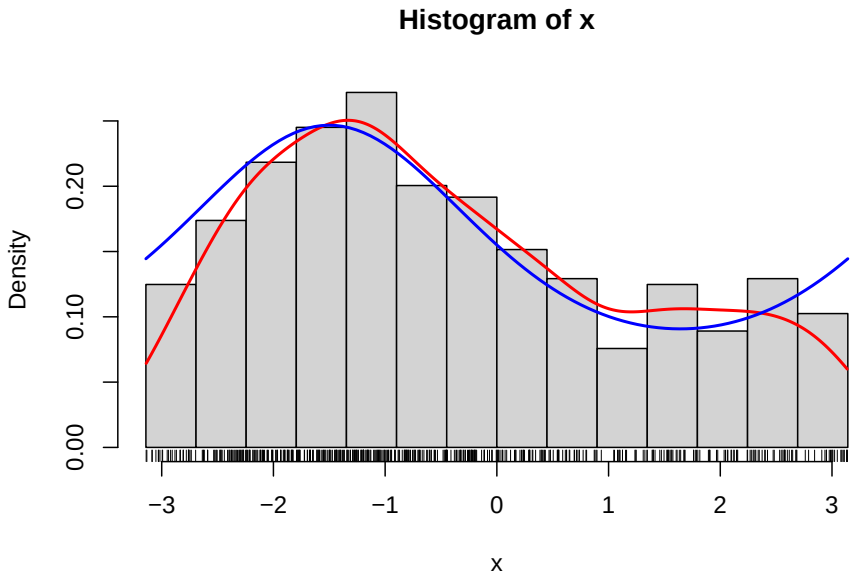


Figure 1.8: Histogram of 500 simulated data points from a von Mises distribution with parameters $\kappa = 0.5$ and $\nu = -1.5$. A smoothed density estimate (red) and the true density (blue) are added to the plot.

```
alpha <- 0.55 # Probability of von Mises distribution A
# The sample function implements the "coin flips"
u <- sample(c(1, 0), 500, replace = TRUE, prob = c(alpha, 1 - alpha))
xy <- rmovMF(500, thetaA) * u + rmovMF(500, thetaB) * (1 - u)
x <- acos(xy[, 1]) * sign(xy[, 2])
```

The `rmovMF` actually implements simulation from a mixture distribution directly, thus there is no need to construct the "coin flips" explicitly.

```
theta <- rbind(thetaA, thetaB)
xy <- rmovMF(length(x), theta, c(alpha, 1 - alpha))
x_alt <- acos(xy[, 1]) * sign(xy[, 2])
```

To compare the simulated data with two mixture components to the model and a smoothed density, we implement an R function that computes the density for an angle argument using the function `dmovMF` that takes a unit circle argument.

```
dvM <- function(x, theta, alpha) {
  xx <- cbind(cos(x), sin(x))
  dmovMF(xx, theta, c(alpha, 1 - alpha)) / (2 * pi)
}
```

Note that `dmovMF` uses normalized spherical measure on the unit circle as reference measure, thus the need for the 2π division if we want the result to be comparable to histograms and density estimates that use Lebesgue measure on $(-\pi, \pi]$ as the reference measure.

```
hist(x, breaks = seq(-pi, pi, length.out = 15), prob = TRUE, ylim = c(0, 0.5))
rug(x)
density(x, bw = "SJ", cut = 0) %>% lines(col = "red", lwd = 2)
curve(dvM(x, theta, alpha), col = "blue", lwd = 2, add = TRUE)

hist(x_alt, breaks = seq(-pi, pi, length.out = 15), prob = TRUE, ylim = c(0, 0.5))
```

```
rug(x_alt)
density(x_alt, bw = "SJ", cut = 0) %>% lines(col = "red", lwd = 2)
curve(dvM(x, theta, alpha), col = "blue", lwd = 2, add = TRUE)
```

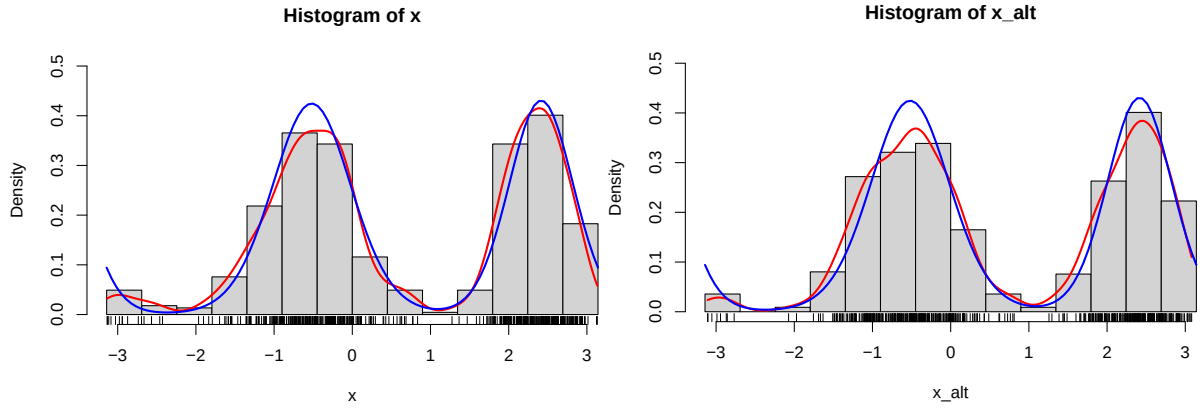


Figure 1.9: Histograms of 500 simulated data points from a mixture of two von Mises distributions using either the explicit construction of the mixture (left) or the functionality in `rmovMF` to simulate mixtures directly (right). A smoothed density estimate (red) and the true density (blue) are added to the plot.

Simulation of data from a distribution finds many applications. The technique is widely used whenever we want to investigate a statistical methodology in terms of its frequentistic performance under various data sampling models, and simulation is a tool of fundamental importance for the practical application of Bayesian statistical methods. Another important application is as a tool for computing approximations of integrals. This is usually called Monte Carlo integration and is a form of numerical integration. Computing probabilities or distribution functions, say, are notable examples of integrals, and we consider here the computation of the probability of the interval $(0, 1)$ for the above mixture of two von Mises distributions.

It is straightforward to compute this probability via Monte Carlo integration as a simple average. Note that we will use a large number of samples, 50,000 in this case, of simulated angles for this computation. Increasing the number even further will make the result more accurate. Chapter 5 deals with the assessment of the accuracy of Monte Carlo integrals, and how this random error can be estimated, bounded and minimized.

```
xy <- rmovMF(50000, theta, c(alpha, 1 - alpha))
x <- acos(xy[, 1]) * sign(xy[, 2])
mean(x > 0 & x < 1)  # Estimate of the probability of the interval (0, 1)
```

```
## [1] 0.08444
```

The probability above could, of course, be expressed using the distribution function of the mixture of von Mises distributions, which in turn can be computed in terms of integrals of von Mises densities. Specifically, the probability is

$$p = \frac{\alpha}{\varphi(\theta_A)} \int_0^1 e^{\theta_{A,1}\cos(x)+\theta_{A,2}\sin(x)} \mathrm{d}x + \frac{1-\alpha}{\varphi(\theta_B)} \int_0^1 e^{\theta_{B,1}\cos(x)+\theta_{B,2}\sin(x)} \mathrm{d}x,$$

but these integrals do not have a simple analytic representation – just as the distribution function of the von Mises distribution doesn't have a simple analytic expression. Thus the computation of the probability requires numerical computation of the integrals.

The R function `integrate` can be used for numerical integration of univariate functions using standard numerical integration techniques. We can thus compute the probability by integrating the density of the mixture, as implemented above as the R function `dvM`. Note the arguments passed to `integrate` below. The first argument is the density function, then follows the lower and the upper limits of the integration, and then follows additional arguments to the density – in this case parameter values.

```
integrate(dvM, 0, 1, theta = theta, alpha = alpha)
```

```
## 0.08635171 with absolute error < 9.6e-16
```

The `integrate` function in R is an interface to a couple of classical QUADPACK Fortran routines for numerical integration via adaptive quadrature. Specifically, the computations rely on approximations of the form

$$\int_a^b f(x)\mathrm{d}x \simeq \sum_i w_i f(x_i)$$

for certain *grid points* x_i and weights w_i, which are computed using Gauss-Kronrod quadrature. This method provides an estimate of the approximation error in addition to the numerical approximation of the integral itself.

It is noteworthy that `integrate` as a function implemented in R takes another function, in this case the density `dvM`, as an argument. R is a functional programming language and functions are first-class citizens. This implies, for instance, that functions can be passed as arguments to other functions using a variable name – just like any other variable can be passed as an argument to a function. In the parlance of functional programming, `integrate` is a functional: a higher-order function that takes a function as argument and returns a numerical value. One of the themes of this book is how to make good use of functional (and object oriented) programming features in R to write clear, expressive and modular code without sacrificing computational efficiency.

Returning to the specific problem of the computation of an integral, we may ask what the purpose of Monte Carlo integration is? Apparently we can just do numerical integration using e.g. `integrate`. There are at least two reasons why Monte Carlo integration is sometimes preferable. First, it is straightforward to implement and often works quite well for multivariate and even high-dimensional integrals, whereas grid-based numerical integration schemes scale poorly with the dimension. Second, it does not require that we have an analytic representation of the density. It is common in statistical applications that we are interested in the distribution of a statistic, which is a complicated transformation of data, and whose density is difficult or impossible to find analytically. Yet if we can just simulate data, we can simulate from the distribution of the statistic, and we can then use Monte Carlo integration to compute whatever probability or integral w.r.t. the distribution of the statistic that we are interested in.

1.2.3 Large scale Monte Carlo methods

Monte Carlo methods are used pervasively in statistics and in many other sciences today. It is nowadays trivial to simulate millions of data points from a simple univariate distribution like the von Mises distribution, and Monte Carlo methods are generally attractive because they allow us to solve computational problems approximately in many cases where exact or analytic computations are impossible and alternative deterministic numerical computations require more adaptation to the specific problem.

Monte Carlo methods are thus really good off-the-shelf methods, but scaling the methods up can be a challenge and is a contemporary research topic. For the simulation of multivariate

p-dimensional data it can, for instance, make a big difference whether the algorithms scale linearly in p or like $O(p^a)$ for some $a > 1$. Likewise, some Monte Carlo methods (e.g. Bayesian computations) depend on a data set of size n, and for large scale data sets, algorithms that scale like $O(n)$ are really the only algorithms that can be used in practice.

1.3 Optimization

The third part of the book is on optimization. This is a huge research field in itself and the focus of the book is on its relatively narrow application to parameter estimation in statistics. That is, when a statistical model is given by a parametrized family of probability distributions, optimization of some criterion – often the likelihood function – is used to find a model that fits the data.

Classical and generic optimization algorithms based on first and second order derivatives can often be used, but its important to understand how convergence can be measured and monitored, and how the different algorithms scale with the size of the data set and the dimension of the parameter space. The choice of the right data structure, such as sparse matrices, can be pivotal for efficient implementations of the criterion function that is to be optimized.

For a mixture of von Mises distributions it is possible to compute the likelihood and optimize it using standard algorithms. However, it can also be optimized using the EM-algorithm, which is a clever algorithm for optimizing likelihood functions for models that can be formulated in terms of latent variables.

This section demonstrates how to fit a mixture of von Mises distributions to the angle data using the EM-algorithm as implemented in the R package movMF. This will illustrate some of the many practical choices we have to make when using numerical optimization such as: the choice of starting value; the stopping criterion; the precise specification of the steps that the algorithm should use; and even whether it should use randomized steps.

1.3.1 The EM-algorithm

The `movMF()` function implements the EM-algorithm for mixtures of von Mises distributions. The code below shows one example call of `movMF()` with one of the algorithmic control arguments specified. It is common in R that such arguments are all bundled together in a single list argument called `control`. It can make the function call appear a bit more complicated than it needed to be, but it allows for easier reuse of the – sometimes fairly long – list of control arguments. Note that as the movMF package works with data as elements on the unit circle, the angle data must be transformed using cos and sin.

```
psi_circle <- cbind(cos(phipsi$psi), sin(phipsi$psi))
vM_fit <- movMF(
  x = psi_circle,
  k = 2,              # The number of mixture components
  control = list(
    start = "S"       # Determines how starting values are chosen
  )
)
```

The function `movMF()` returns *an object of class `movMF`* as the documentation says (see `?movMF`). In this case it means that `vM_fit` is a list with a class label `movMF`, which controls how generic

functions work for this particular list. When the object is printed, for instance, some of the the content of the list is formatted and printed as follows.

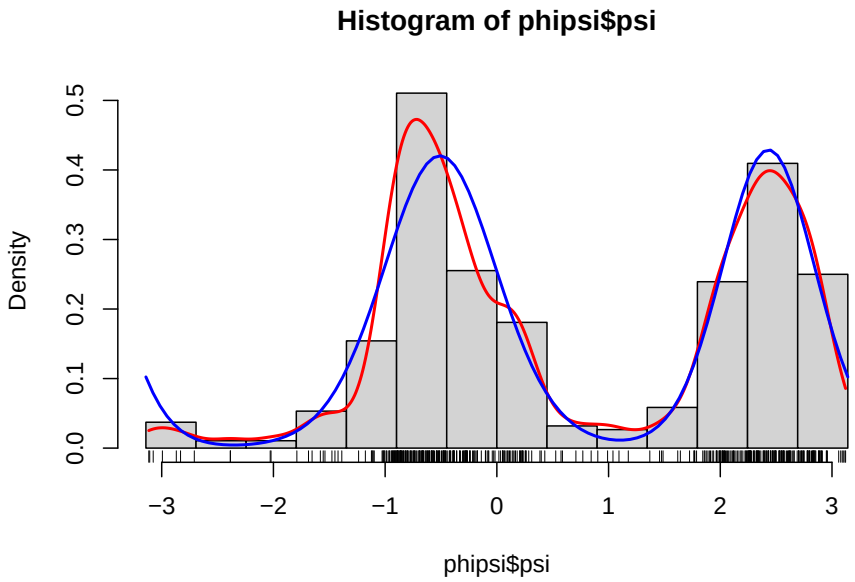
```
vM_fit
```

```
## theta:
##         [,1]      [,2]
## 1  3.472846 -1.935807
## 2 -4.508098  3.872847
## alpha:
## [1] 0.5486586 0.4513414
## L:
## [1] 193.3014
```

What we see above is the estimated θ-parameters for each of the two mixture components printed as a matrix, the mixture proportions (`alpha`) and the value of the log-likelihood function (`L`) in the estimated parameters.

We can compare the fitted model to the data using the density function as implemented above and the parameters estimated by the EM-algorithm.

```
hist(phipsi$psi, breaks = seq(-pi, pi, length.out = 15), prob = TRUE)
rug(phipsi$psi)
density(phipsi$psi, bw = "SJ", cut = 0) %>% lines(col = "red", lwd = 2)
curve(dvM(x, vM_fit$theta, vM_fit$alpha[1]), add = TRUE, col = "blue", lwd = 2)
```



As we can see from the figure, this looks like a fairly good fit to the data, and this is reassuring for two reasons. First, it shows that the two-component mixture of von Mises distributions is a good model. This is a statistical reassurance. Second, it shows that the optimization over the parameter space found a good fit. This is a numerical reassurance. If we optimize over a parameter space and subsequently find that the model does not fit the data well, it is either because the numerical optimization failed or because the model is wrong.

Numerical optimization can fail for a number of reasons. For once, the algorithm can get stuck in a local optimum, but it can also stop prematurely either because the maximal number of iterations is reached or because the stopping criterion is fulfilled (even though the algorithm has not reached an optimum). Some information related to the two last problems can be gathered from the algorithm itself, and for `movMF()` such information is found in a list entry of

vM_fit.

```
vM_fit$details
```

```
## $reltol
## [1] 1.490116e-08
##
## $iter
##    iter maxiter
##      16     100
##
## $logLiks
## [1] 193.3014
##
## $E
## [1] "softmax"
##
## $kappa
## NULL
##
## $minalpha
## [1] 0
##
## $converge
## [1] TRUE
```

We see that the number of iterations used was 16 (and less than the maximal number of 100), and the stopping criterion (small relative improvement) was active (`converge` is `TRUE`, if `FALSE` the algorithm is run for a fixed number of iterations). The tolerance parameter used for the stopping criterion was 1.49×10^{-8}.

The small relative improvement criterion for stopping in iteration n is

$$|L(\theta_{n-1}) - L(\theta_n)| < \varepsilon(|L(\theta_{n-1})| + \varepsilon)$$

where L is the log-likelihood and ε is the tolerance parameter above. The default tolerance parameter is the square root of the machine epsilon, see also `?.Machine`. This is a commonly encountered default for a "small-but-not-too-small-number", but outside of numerical differentiation this default may not be supported by much theory.

By choosing different control arguments we can change how the numerical optimization proceeds. A different method for setting the starting value is chosen below, which contains a random component. Here we consider the results in four different runs of the entire algorithm with different random starting values. We also decrease the number of maximal iterations to 10 and make the algorithm print out information about its progress along the way.

```
vM_control <- list(
    verbose = TRUE,    # Print output showing algorithmic progress
    maxiter = 10,
    nruns = 4          # Effectively 4 runs with randomized starting values
  )
vM_fit <- movMF(psi_circle, 2, control = vM_control)
## Run: 1
## Iteration: 0 *** L: 158.572
## Iteration: 1 *** L: 190.999
```

```
## Iteration: 2 *** L: 193.118
## Iteration: 3 *** L: 193.238
## Iteration: 4 *** L: 193.279
## Iteration: 5 *** L: 193.293
## Iteration: 6 *** L: 193.299
## Iteration: 7 *** L: 193.3
## Iteration: 8 *** L: 193.301
## Iteration: 9 *** L: 193.301
## Run: 2
## Iteration: 0 *** L: 148.59
## Iteration: 1 *** L: 188.737
## Iteration: 2 *** L: 192.989
## Iteration: 3 *** L: 193.197
## Iteration: 4 *** L: 193.264
## Iteration: 5 *** L: 193.288
## Iteration: 6 *** L: 193.297
## Iteration: 7 *** L: 193.3
## Iteration: 8 *** L: 193.301
## Iteration: 9 *** L: 193.301
## Run: 3
## Iteration: 0 *** L: 168.643
## Iteration: 1 *** L: 189.946
## Iteration: 2 *** L: 192.272
## Iteration: 3 *** L: 192.914
## Iteration: 4 *** L: 193.157
## Iteration: 5 *** L: 193.249
## Iteration: 6 *** L: 193.282
## Iteration: 7 *** L: 193.295
## Iteration: 8 *** L: 193.299
## Iteration: 9 *** L: 193.301
## Run: 4
## Iteration: 0 *** L: 4.43876
## Iteration: 1 *** L: 5.33851
## Iteration: 2 *** L: 6.27046
## Iteration: 3 *** L: 8.54826
## Iteration: 4 *** L: 14.4429
## Iteration: 5 *** L: 29.0476
## Iteration: 6 *** L: 61.3225
## Iteration: 7 *** L: 116.819
## Iteration: 8 *** L: 172.812
## Iteration: 9 *** L: 190.518
```

In all four cases it appears that the algorithm is approaching the same value of the log-likelihood as what we found above, though the last run starts out in a much lower value and takes more iterations to reach a large log-likelihood value. Note also that in all runs the log-likelihood is increasing. It is a feature of the EM-algorithm that every step of the algorithm will increase the likelihood.

Variations of the EM-algorithm are possible, like in `movMF` where the control argument `E` determines the so-called E-step of the algorithm. Here the default (`softmax`) gives the actual EM-algorithm whereas `hardmax` and `stochmax` give alternatives. While the alternatives do not guarantee that the log-likelihood increases in every iteration they can be numerically beneficial.

We rerun the algorithm from the same four starting values as above but using `hardmax` as the E-step. Note how we can reuse the control list by just adding a single element to it.

```
vM_control$E <- "hardmax"
vM_fit <- movMF(psi_circle, 2, control = vM_control)
## Run: 1
## Iteration: 0 *** L: 158.572
## Iteration: 1 *** L: 193.052
## Iteration: 2 *** L: 193.052
## Run: 2
## Iteration: 0 *** L: 148.59
## Iteration: 1 *** L: 192.443
## Iteration: 2 *** L: 192.812
## Iteration: 3 *** L: 193.052
## Iteration: 4 *** L: 193.052
## Run: 3
## Iteration: 0 *** L: 168.643
## Iteration: 1 *** L: 191.953
## Iteration: 2 *** L: 192.812
## Iteration: 3 *** L: 193.052
## Iteration: 4 *** L: 193.052
## Run: 4
## Iteration: 0 *** L: 4.43876
## Iteration: 1 *** L: 115.34
## Iteration: 2 *** L: 191.839
## Iteration: 3 *** L: 192.912
## Iteration: 4 *** L: 192.912
```

It is striking that all runs now stopped before the maximal number of iterations was reached, and run four is particularly noteworthy as it jumps from its low starting value to above 190 in just two iterations. However, `hardmax` is a heuristic algorithm, whose fixed points are not necessarily stationary points of the log-likelihood. We can also see above that all four runs stopped at values that are clearly smaller than the log-likelihood of about 193.30 that was reached using the real EM-algorithm. Whether this is a problem from a statistical viewpoint is a different matter; using `hardmax` could give an estimator that is just as efficient as the maximum likelihood estimator.

Which optimization algorithm should we then use? This is in general a very difficult question to answer, and it is non-trivial to correctly assess which algorithm is "the best". As the application of `movMF()` above illustrates, optimization algorithms may have a number of different parameters that can be tweaked, and when it comes to actually implementing an optimization algorithm we need to: make it easy to tweak parameters; investigate and quantify the effect of the parameters on the algorithm; and choose sensible defaults. Without this work it is impossible to have a meaningful discussion about the benefits and deficits of various algorithms.

1.3.2 Large scale optimization

Numerical optimization is the tool that has made statistical models useful for real data analysis – most importantly by making it possible to compute maximum likelihood estimators in practice. This works very well when the model can be given a parametrization with a concave log-likelihood, while optimization of more complicated log-likelihood surfaces can be numerically challenging and lead to statistically dubious results.

In contemporary statistics and machine learning, numerical optimization has come to play an even more important role as structural model constraints are now often also part of the optimization, for instance via penalty terms in the criterion function. Instead of working with simple models selected for each specific problem and with few parameters, we use complex models with thousands or millions of parameters. They are flexible and adaptable but need careful, regularized optimization to not overfit the data. With large amounts of data the regularization can be turned down and we can discover aspects of data that the simple models would never show. However, we need to pay a computational price.

When the dimension of the parameter space, p, and the number of data points, n, are both large, evaluation of the log-likelihood or its gradient can become prohibitively costly. Optimization algorithms based on higher order derivatives are then completely out of the question, and even if the (penalized) log-likelihood and its gradient can be computed in $O(pn)$-operations this may still be too much for an optimization algorithm that does this in each iteration. Such large scale optimization problems have spurred a substantial development of stochastic gradient methods and related stochastic optimization algorithms that only use parts of data in each iteration, and which are currently the only way to fit sufficiently complicated models to data.

Part I: Smoothing

Chapter 2

Density estimation

This chapter is on nonparametric density estimation. A classical nonparametric estimator of a density is the histogram, which provides discontinuous and piecewise constant estimates. The focus in this chapter is on some of the alternatives that provide continuous or even smooth estimates instead.

Kernel methods form an important class of smooth density estimators as implemented by the R function `density()`. These estimators are essentially just locally weighted averages, and their computation is relatively straightforward in theory. In practice, different choices of how to implement the computations can, however, have a big effect on the actual computation time, and the implementation of kernel density estimators will illustrate three points:

- if possible, choose vectorized implementations in R,
- if a small loss in accuracy is acceptable, an approximate solution can be orders of magnitudes faster than a literal implementation,
- the time it takes to numerically evaluate different elementary functions can depend a lot on the function and how you implement the computation.

The first point is emphasized because it results in implementations that are short, expressive and easier to understand just as much as it typically results in computationally more efficient implementations. Note also that not every computation can be vectorized in a beneficial way, and one should never go through hoops to vectorize a computation.

Kernel methods rely on one or more *regularization parameters* that must be selected to achieve the right balance of adapting to data without adapting too much to the random variation in the data. Choosing the right amount of regularization is just as important as choosing the method to use in the first place. It may, in fact, be more important. We actually do not have a complete implementation of a nonparametric estimator until we have implemented a data driven and automatic way of choosing the amount of regularization. Implementing only the computations for evaluating a kernel estimator, say, and leaving it completely to the user to choose the bandwidth is a job half done. Methods and implementations for choosing the bandwidth are therefore treated in some detail in this chapter.

In the final section a likelihood analysis is carried out. This is done to further clarify why regularized estimators are needed to avoid overfitting to the data, and why there is in general no nonparametric maximum likelihood estimator of a density. Regularization of the likelihood can be achieved by constraining the density estimates to belong to a family of increasingly flexible parametric densities that are fitted to data. This is known as the *method of sieves*. Another approach is based on basis expansions, but in either case, automatic selection of the amount of regularization is just as important as for kernel methods.

2.1 Univariate density estimation

Recall the data on ϕ- and ψ-angles in polypeptide backbone structures, as considered in Section 1.1.1.

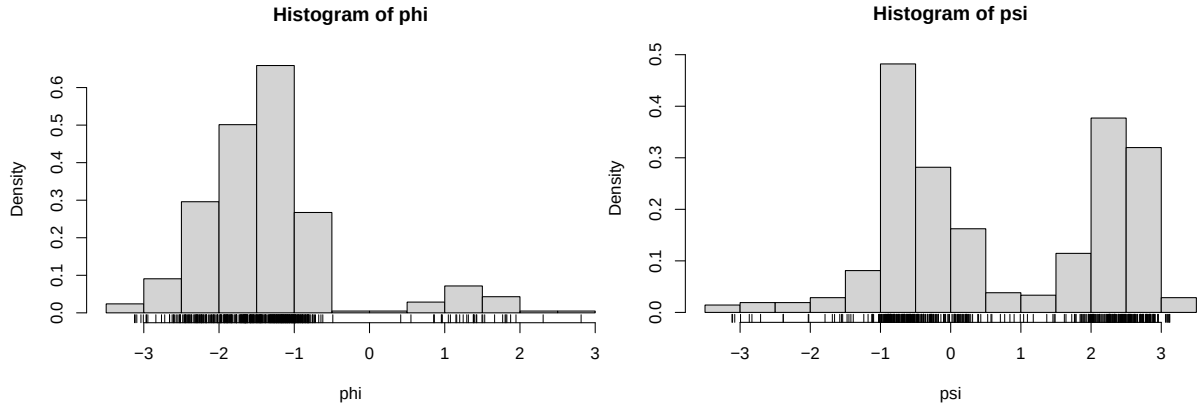


Figure 2.1: Histograms equipped with a rug plot of the distribution of ϕ-angles (left) and ψ-angles (right) of the peptide planes in the protein human protein 1HMP.

We will in this section treat methods for smooth density estimation for univariate data such as data on either the ϕ- or the ψ-angle.

We let f_0 denote the unknown density that we want to estimate. That is, we imagine that the data points $x_1, \ldots, x_n$ are all observations drawn from the probability measure with density f_0 w.r.t. Lebesgue measure on $\mathbb{R}$.

Suppose first that f_0 belongs to a parametrized statistical model $(f_\theta)_\theta$, where f_θ is a density w.r.t. Lebesgue measure on $\mathbb{R}$. If $\hat{\theta}$ is an estimate of the parameter, $f_{\hat{\theta}}$ is an estimate of the unknown density f_0. For a parametric family we can always try to use the MLE

$$\hat{\theta} = \arg\max_\theta \sum_{j=1}^{n} \log f_\theta(x_j)$$

as an estimate of θ. Likewise, we might compute the empirical mean and variance for the data and plug those numbers into the density for the Gaussian distribution, and in this way obtain a Gaussian density estimate of f_0.

```
psi_mean <- mean(psi)
psi_sd <- sd(psi)
hist(psi, prob = TRUE)
rug(psi)
curve(dnorm(x, psi_mean, psi_sd), add = TRUE, col = "red")
```

As Figure 2.2 shows, if we fit a Gaussian distribution to the ψ-angle data we get a density estimate that clearly does not match the histogram. The Gaussian density matches the data on the first and second moments, but the histogram shows a clear bimodality that the Gaussian distribution by definition cannot match. Thus we need a more flexible parametric model than the Gaussian if we want to fit a density to this data set.

In nonparametric density estimating we want to estimate the target density, f_0, without assuming that it belongs to a particular parametrized family of densities.

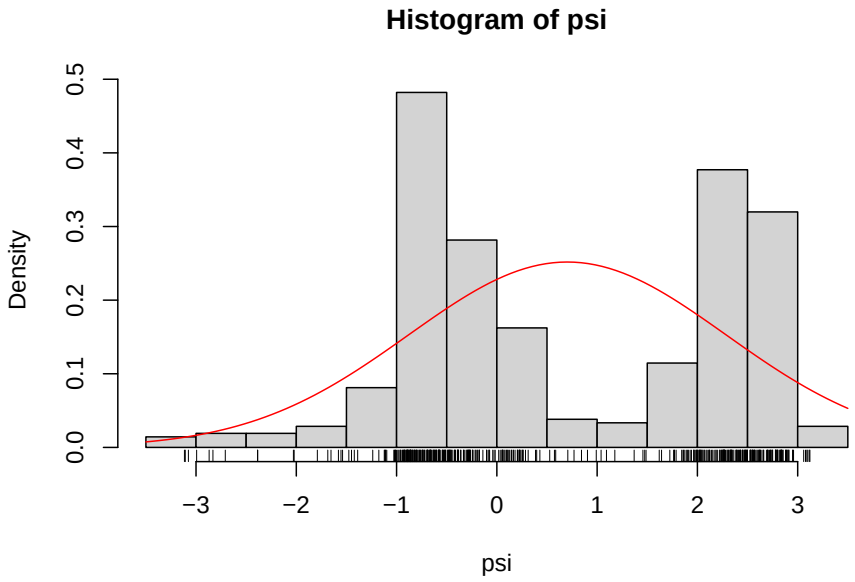


Figure 2.2: Gaussian density (red) fitted to the ψ-angles.

2.2 Kernel methods

A simple approach to nonparametric density estimation relies on the approximation

$$P(X \in (x-h, x+h)) = \int_{x-h}^{x+h} f_0(z)\, dz \simeq f_0(x) 2h,$$

which is valid for any continuous density f_0. Inverting this approximation and using the law of large numbers,

$$\begin{aligned} f_0(x) &\simeq \frac{1}{2h} P(X \in (x-h, x+h)) \\ &\simeq \frac{1}{2hn} \sum_{j=1}^{n} 1_{(x-h,x+h)}(x_j) \\ &= \underbrace{\frac{1}{2hn} \sum_{j=1}^{n} 1_{(-h,h)}(x - x_j)}_{\hat{f}_h(x)} \end{aligned}$$

for i.i.d. observations $x_1, \ldots, x_n$ from the distribution $f_0 \cdot m$. The function $\hat{f}_h$ defined as above is an example of a kernel density estimator with a rectangular kernel. We immediately note that h has to be chosen appropriately. If h is large, $\hat{f}_h$ will be flat and close to a constant. If h is small, $\hat{f}_h$ will make large jumps close to the observations.

What do we then mean by an "appropriate" choice of h above? To answer this we must have some prior assumptions about what we expect f_0 to look like. Typically, we expect f_0 to have few oscillations and to be fairly smooth, and we want $\hat{f}_h$ to reflect that. A too large h will oversmooth the data relative to f_0 by effectively ignoring the data, while a too small h will undersmooth the data relative to f_0 by allowing individual data points to have large local effects that make the estimate wiggly. More formally, we can look at the mean and variance of $\hat{f}_h$. Letting $p(x,h) = P(X \in (x-h, x+h))$, it follows that $f_h(x) = E(\hat{f}_h(x)) = p(x,h)/(2h)$ while

$$V(\hat{f}_h(x)) = \frac{p(x,h)(1-p(x,h))}{4h^2n} \simeq f_h(x)\frac{1}{2hn}. \tag{2.1}$$

We see from these computations that for $\hat{f}_h(x)$ to be approximately unbiased for any x we need h to be small – ideally letting $h \to 0$ since then $f_h(x) \to f_0(x)$. However, this will make the variance blow up, and to minimize variance we should instead choose h as large as possible. One way to define "appropriate" is then to strike a balance between the bias and the variance as a function of h so as to minimize the mean squared error of $\hat{f}_h(x)$.

We will find the optimal tradeoff for the rectangular kernel in Section 2.3 on bandwidth selection. It's not difficult, and you are encouraged to try finding it yourself at this point. In this section we will focus on computational aspects of kernel density estimation, but first we will generalize the estimator by allowing for other kernels.

The estimate $\hat{f}_h(x)$ will be unbiased if f_0 is constantly equal to $f_0(x)$ in the entire interval $(x-h, x+h)$. This is atypical and can only happen for all x if f_0 is constant. We expect the typical situation to be that f_0 deviates the most from $f_0(x)$ close to $x \pm h$, and that this causes a bias of $\hat{f}_h(x)$. Observations falling close to $x+h$, say, should thus count less than observations falling close to x? The rectangular kernel makes a sharp cut; either a data point is in or it is out. If we use a smooth weighting function instead of a sharp cut, we might be able to include more data points and lower the variance while keeping the bias small. This is precisely the idea of *kernel estimators*, defined generally as

$$\hat{f}_h(x) = \frac{1}{hn}\sum_{j=1}^{n} K\left(\frac{x - x_j}{h}\right) \tag{2.2}$$

for a kernel $K : \mathbb{R} \to \mathbb{R}$. The parameter $h > 0$ is known as the *bandwidth*. Examples of kernels include the *uniform* or *rectangular kernel*

$$K(x) = \frac{1}{2}1_{(-1,1)}(x),$$

and the *Gaussian kernel*

$$K(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}.$$

One direct benefit of considering other kernels than the rectangular is that $\hat{f}_h$ inherits all smoothness properties from K. Whereas the rectangular kernel is not even continuous, the Gaussian kernel is C^∞ and so is the resulting kernel density estimate.

2.2.1 Implementation

What should be computed to compute a kernel density estimate? That is, in fact, a good question, because the definition actually just specifies how to evaluate $\hat{f}_h$ in any given point x, but there is really not anything to compute until we need to evaluate $\hat{f}_h$. Thus when we implement kernel density estimation we really implement algorithms for evaluating a density estimate in a finite number of points.

Our first implementation is a fairly low-level implementation that returns the evaluation of the density estimate in a given number of equidistant points. The function mimics some of the defaults of `density()` so that it actually evaluates the estimate in the same points as `density()`.

```r
# This is an implementation of the function 'kern_dens' that computes
# evaluations of Gaussian kernel density estimates in a grid of points.
#
# The function has three formal arguments: 'x' is the numeric vector of data
# points, 'h' is the bandwidth and 'm' is the number of grid points.
# The default value of 512 is chosen to match the default of 'density()'.
kern_dens <- function(x, h, m = 512) {
  rg <- range(x)
  # xx is equivalent to grid points in 'density()'
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  y <- numeric(m) # The evaluations, initialized as a vector of zeros
  # The actual computation is done using nested for-loops. The outer loop
  # is over the grid points, and the inner loop is over the data points.
  for (i in seq_along(xx))
    for (j in seq_along(x))
      y[i] <- y[i] + exp(- (xx[i] - x[j])^2 / (2 * h^2))
  y <- y / (sqrt(2 * pi) * h * length(x))
  list(x = xx, y = y)
}
```

Note that the function returns a list containing the grid points (`x`) where the density estimate is evaluated as well as the estimated density evaluations (`y`). Note also that the argument `m` above sets the number of grid points, whereas `density()` uses the argument `n` for that. The latter can be a bit confusing as n is often used to denote the number of data points.

We will immediately test if the implementation works as expected – in this case by comparing it to our reference implementation `density()`.

```r
f_hat <- kern_dens(psi, 0.2)
f_hat_dens <- density(psi, 0.2)
plot(f_hat, type = "l", lwd = 4, xlab = "x", ylab = "Density")
lines(f_hat_dens, col = "red", lwd = 2)
plot(f_hat$x, f_hat$y - f_hat_dens$y,
  type = "l",
  lwd = 2,
  xlab = "x",
  ylab = "Difference"
)
```

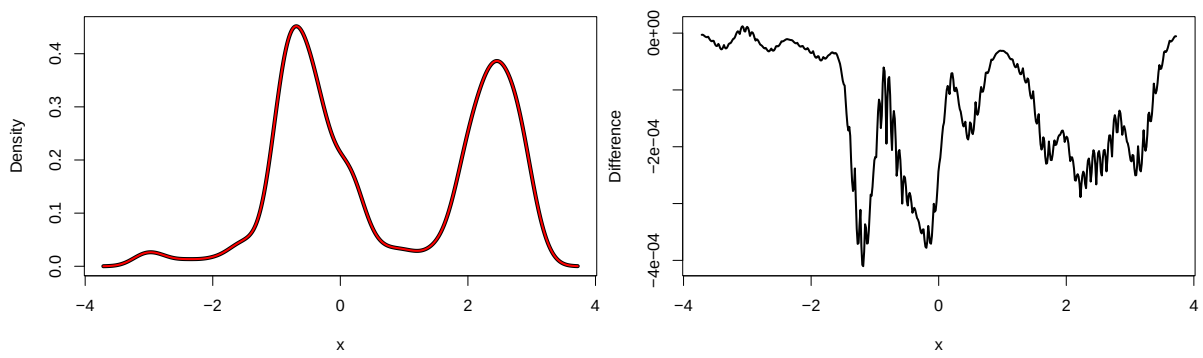


Figure 2.3: Kernel density estimates with the Gaussian kernel (left) using R's implementation (black) and our implementation (red) together with differences of the estimates (right).

Figure 2.3 suggests that the estimates computed by our implementation and by `density()` are the same when we just visually compare the plotted densities. However, if we look at the differences instead, we see that they are as large as 4×10^{-4} in absolute value. This is way above what we should expect from rounding errors alone when using double precision arithmetic. Thus the two implementations only compute *approximately* the same, which is, in fact, because `density()` relies on certain approximations for run time efficiency.

In R we can often beneficially implement computations in a vectorized way instead of using an explicit loop. It is fairly easy to change the implementation to be more vectorized by computing each evaluation in one single line using the `sum()` function and the fact that `exp()` and squaring are vectorized.

```
kern_dens_vec <- function(x, h, m = 512) {
  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  y <- numeric(m)
  # The inner loop from 'kern_dens' has been vectorized, and only the
  # outer loop over the grid points remains.
  const <- (sqrt(2 * pi) * h * length(x))
  for (i in seq_along(xx))
      y[i] <- sum(exp(-(xx[i] - x)^2 / (2 * h^2))) / const
  list(x = xx, y = y)
}
```

We test this new implementation by comparing it to our previous implementation.

```
range(kern_dens(psi, 0.2)$y - kern_dens_vec(psi, 0.2)$y)
```

```
## [1] -5.551115e-16  3.885781e-16
```

The magnitude of the differences are of order at most 10^{-16}, which is what can be expected due to rounding errors. Thus we conclude that up to rounding errors, `kern_dens()` and `kern_dens_vec()` return the same on this data set. This is, of course, not a comprehensive test, but it is an example of one among a number of tests that should be considered.

There are several ways to get completely rid of the explicit loops and write an entirely vectorized implementation in R. One of the solutions will use the `sapply()` function, which belongs to the family of `*apply()` functions that apply a function to each element in a vector or a list. In the parlance of functional programming the `*apply()` functions are variations of the functional, or higher-order-function, known as map.

```
kern_dens_apply <- function(x, h, m = 512) {
  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  const <- sqrt(2 * pi) * h * length(x)
  y <- sapply(xx, function(z) sum(exp(-(z - x)^2 / (2 * h^2))) / const)
  list(x = xx, y = y)
}
```

The `sapply()` call above will apply the function `function(z) sum(dnorm(...` to every element in the vector `xx` and return the result as a vector. The function is an example of an *anonymous function* that does not get a name and exists only during the `sapply()` evaluation. Instead of `sapply()` it is possible to use `lapply()` that returns a list. In fact, `sapply()` is a simple wrapper around `lapply()` that attempts to "simplify" the result from a list to an array (and in this case to a vector).

An alternative, and also completely vectorized, solution can be based on the functions `outer()` and `rowMeans()`.

```
kern_dens_outer <- function(x, h, m = 512) {
  rg <- range(x)
  xx <- seq(rg[1] - 3 * h, rg[2] + 3 * h, length.out = m)
  y <- outer(xx, x, function(zz, z) exp(-(zz - z)^2 / (2 * h^2)))
  y <- rowMeans(y) / (sqrt(2 * pi) * h)
  list(x = xx, y = y)
}
```

The `outer()` function evaluates the kernel in all combinations of the grid and data points and returns a matrix of dimensions $m \times n$. The function `rowMeans()` computes the means of each row and returns a vector of length m.

We should, of course, also remember to test these two last implementations.

```
range(kern_dens(psi, 0.2)$y - kern_dens_apply(psi, 0.2)$y)
```

```
## [1] -5.551115e-16  3.885781e-16
```

```
range(kern_dens(psi, 0.2)$y - kern_dens_outer(psi, 0.2)$y)
```

```
## [1] -4.996004e-16  3.885781e-16
```

The natural question is then how to choose between the different implementations? Besides being correct it is important that the code is easy to read and understand. Which of the four implementations above that is best in this respect may depend a lot on the background of the reader. If you strip the implementations for comments, all four are arguably quite readable, but `kern_dens()` with the double loop might appeal a bit more to people with a background in imperative programming, while `kern_dens_apply()` might appeal more to people with a preference for functional programming. This functional and vectorized solution is also a bit closer to the mathematical notation with e.g. the sum sign Σ being mapped directly to the `sum()` function instead of the incremental addition in the for-loop. For these specific implementations these differences are mostly aesthetic nuances and preferences may be more subjective than substantial.

To make a qualified choice between the implementations we should investigate if they differ in terms of run time and memory consumption.

2.2.2 Benchmarking

Benchmarking is about measuring and comparing performance. For software this often means measuring run time and memory usage, though there are clearly many other aspects of software that should be benchmarked in general. This includes user experience, energy consumption and implementation and maintenance time. In this section we focus on benchmarking run time.

The function `system.time` in R provides a simple way of benchmarking run time measured in seconds.

```
system.time(kern_dens(psi, 0.2))
system.time(kern_dens_vec(psi, 0.2))
system.time(kern_dens_apply(psi, 0.2))
system.time(kern_dens_outer(psi, 0.2))
```

```
## kern_dens:
```

```
##    user  system elapsed
##   0.036   0.000   0.036
## kern_dens_vec:
##    user  system elapsed
##   0.004   0.000   0.004
## kern_dens_apply:
##    user  system elapsed
##   0.005   0.001   0.005
## kern_dens_outer:
##    user  system elapsed
##   0.005   0.000   0.005
```

The "elapsed" time is the total run time as experienced, while the "user" and "system" times are how long the CPU spent on executing your code and operating system code on behalf of your code, respectively.

From this simple benchmark, `kern_dens()` is clearly substantially slower than the three other implementations. For more systematic benchmarking of run time, the R package microbenchmark is useful.

```
library(microbenchmark)
```

```
kern_bench <- microbenchmark(
  kern_dens(psi, 0.2),
  kern_dens_vec(psi, 0.2),
  kern_dens_apply(psi, 0.2),
  kern_dens_outer(psi, 0.2)
)
```

The result stored in `kern_bench()` is a data frame with two columns. The first contains the R expressions evaluated, and the second is the evaluation time measured in nanoseconds. Each of the four expressions were evaluated 100 times, and the data frame thus has 400 rows. The `times` argument to `microbenchmark()` can be used to change the number of evaluations per expression if needed.

It may not be immediately obvious that `kern_bench()` is a data frame, because printing will automatically summarize the data, but the actual data structure is revealed by the R function `str()`.

```
str(kern_bench)
```

```
## Classes 'microbenchmark' and 'data.frame':   400 obs. of  2 variables:
##  $ expr: Factor w/ 4 levels "kern_dens(psi, 0.2)",..: 3 3 3 4 3 2 1 1 2 2 ...
##  $ time: num  3474883 3467712 3355875 4488063 4729016 ...
```

A total of 400 evaluations were done for the above benchmark, and `microbenchmark()` does the evaluations in a random order by default. Measuring evaluation time on a complex system like a modern computer is an empirical science, and the order of evaluation can potentially affect the results as the conditions for the evaluation change over time. The purpose of the randomization is to avoid that the ordering causes systematically misleading results.

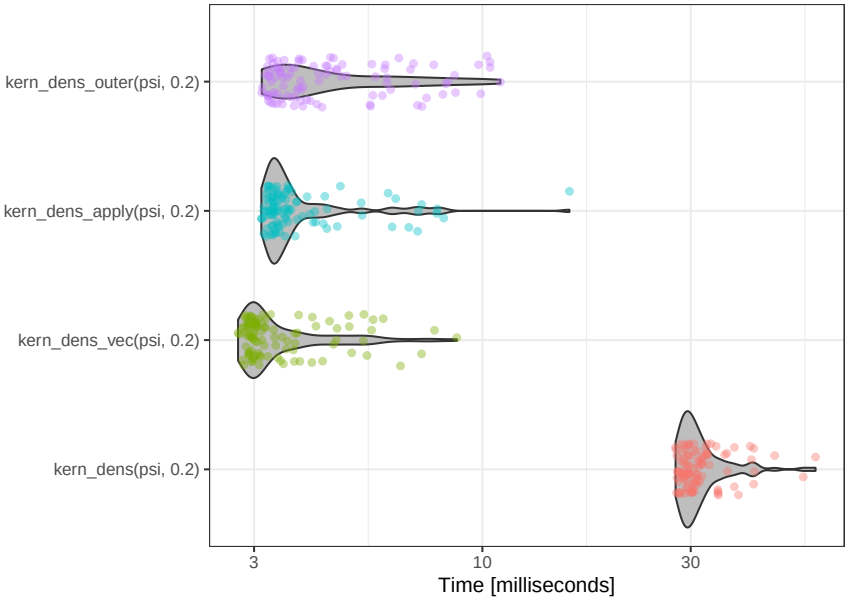
The microbenchmark package implements some methods for summarizing and printing the results such as the following summary table with times in milliseconds.

```
## Unit: milliseconds
##                       expr    min     lq   mean median     uq    max neval cld
```

```
##        kern_dens(psi, 0.2) 27.70 29.05 31.92  30.37 32.51 57.94   100   c
##    kern_dens_vec(psi, 0.2)  2.76  2.95  3.59   3.08  3.72  8.74   100 a
##  kern_dens_apply(psi, 0.2)  3.12  3.31  4.03   3.43  3.82 15.82   100 ab
##  kern_dens_outer(psi, 0.2)  3.12  3.41  4.85   3.88  5.80 11.00   100  b
```

The summary table shows some key statistics like median and mean evaluation times but also extremes and upper and lower quartiles. The distributions of run times can be investigated further using the `autoplot()` function, which is based on ggplot2 and thus easy to modify.

```
autoplot(kern_bench) +
  geom_jitter(position = position_jitter(0.2, 0),
              aes(color = expr), alpha = 0.4) +
  aes(fill = I("gray")) +
  theme(legend.position = "none")
```



This more refined benchmark study does not change our initial impression from using `system.time()` substantially. The function `kern_dens()` is notably slower than the three vectorized implementations, but we are now able to more clearly see the minor differences among them. For instance, `kern_dens_vec()` and `kern_dens_apply()` have very similar run time distributions, while `kern_dens_outer()` clearly has a larger median run time and also a run time distribution that is more spread out to the right.

In many cases when we benchmark run time it is of interest to investigate how run time depends on various parameters. This is so for kernel density estimation, where we want to understand how changes in the number of data points, n, and the number of grid points, m, affect run time. We can still use `microbenchmark()` for running the benchmark experiment, but we will typically process and plot the benchmark data afterwards in a customized way.

Figure 2.4 shows median run times for an experiment with 28 combinations of parameters for each of the four different implementations yielding a total of 112 different R expressions being benchmarked. The number of replications for each expression was set to 40. The results confirm that `kern_dens()` is substantially slower than the vectorized implementations for all combinations of n and m. However, Figure 2.4 also reveals a new pattern; `kern_dens_outer()` appears to scale with n in a slightly different way than the two other vectorized implementations for small n. It is comparable to or even a bit faster than `kern_dens_vec()` and `kern_dens_apply()` for very small data sets, while it becomes slower for the larger data sets.

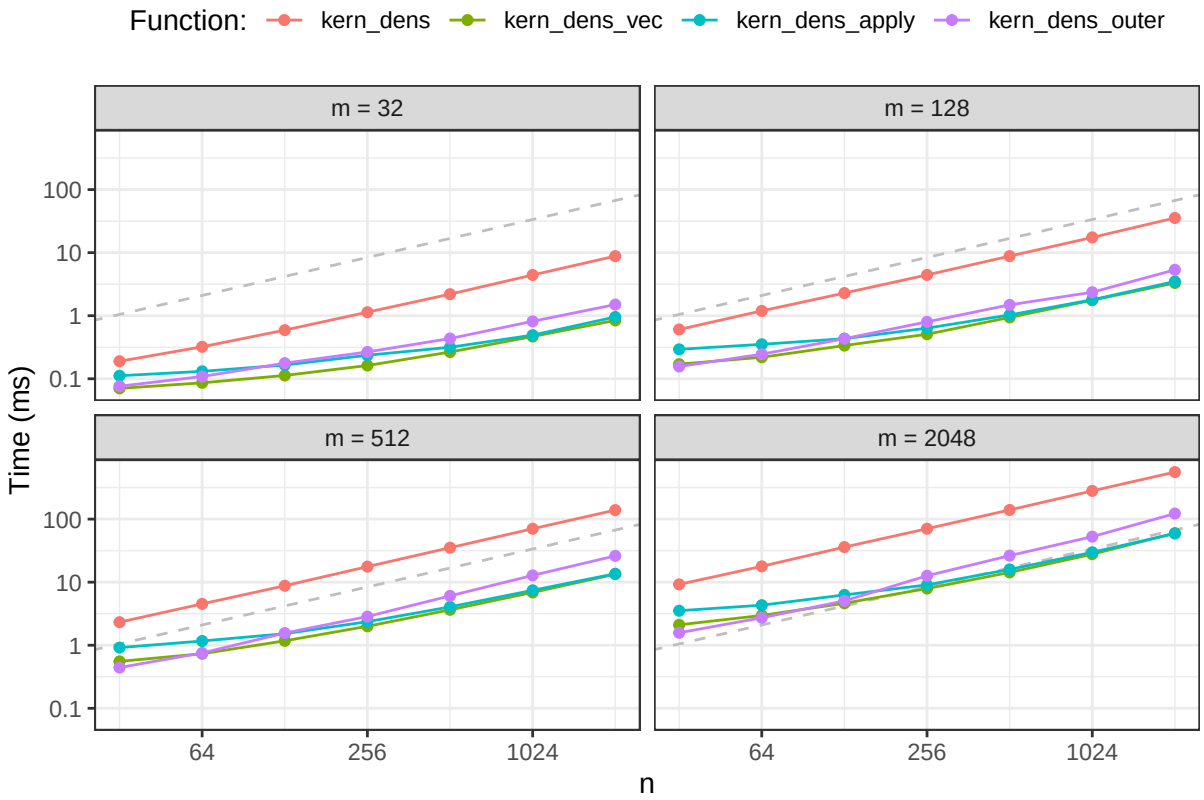


Figure 2.4: Median run times for the four different implementations of kernel density estimation. The dashed gray line is a reference line with slope 1.

Run time for many algorithms have to a good approximation a dominating power law behavior as a function of typical size parameters, that is, the run time will scale approximately like $n \mapsto Cn^a$ for constants C and a and with n denoting a generic size parameter. Therefore it is beneficial to plot run time using log-log scales and to design benchmark studies with size parameters being equidistant on a log-scale. With approximate power law scaling, the log run time behaves like

$$\log(C) + a \log(n),$$

that is, on a log-log scale we see approximate straight lines. The slope reveals the exponent a, and two different algorithms for solving the same problem might have different exponents and thus different slopes on the log-log-scale. Two different implementations of the same algorithm should have approximately the same slope but may differ in the constant C depending upon how efficient the particular implementation is in the particular programming language used. Differences in C correspond to vertical translations on the log-log scale.

In practice, we will see some deviations from straight lines on the log-log plot for a number of reasons. Writing the run time as $Cn^a + R(n)$, the residual term $R(n)$ will often be noticeable or even dominating and positive for small n. It is only for large enough n that the power law term, Cn^a, will dominate. In addition, run time can be affected by hardware constraints such as cache and memory sizes, which can cause abrupt jumps in run time.

Using `microbenchmark()` over `system.time()` has two main benefits. First, it handles the replication and randomization automatically, which is convenient. Second, it attempts to provide more accurate timings. The latter is mostly important when we benchmark very fast computations.

It can be debated if a median summary of randomly ordered evaluations is the best way to

summarize run time. This is due to the way R does memory management. R allocates and deallocates memory automatically and uses garbage collection for the deallocation. This means that computations occasionally, and in a somewhat unpredictable manner, trigger the garbage collector, and as a result a small fraction of the evaluations may take substantially longer time than the rest. The median will typically be almost unaffected, and memory deallocation is thus effectively (and wrongly) disregarded from run time when the median summary is used. This is an argument for using the mean instead of the median, but due to the randomization the computation that triggered the garbage collector might not be the one that caused the memory allocation in the first place. Using the mean instead of the median will therefore smear out the garbage collection run time on all benchmarked expressions. Setting the argument `control = list(order = "block")` for `microbenchmark()` will evaluate the expressions in blocks, which in combination with a mean summary more correctly accounts for memory allocation and deallocation in the run time. The downside is that without the randomization the results might suffer from other artefacts. This book will use randomization and median summaries throughout, but we keep in mind that this could underestimate actual average run time depending upon how much memory a given computation requires. Memory usage and how it affects run time by triggering garbage collection will be dealt with via code profiling tools instead.

2.3 Bandwidth selection

2.3.1 Revisiting the rectangular kernel

We return to the rectangular kernel and compute the mean squared error. In the analysis it may be helpful to think about n large and h small. Indeed, we will eventually choose $h = h_n$ as a function of n such that as $n \to \infty$ we have $h_n \to 0$. We should also note the $f_h(x) = E(\hat{f}_h(x))$ is a density, thus $\int f_h(x)\mathrm{d}x = 1$.

We will assume that f_0 is sufficiently differentiable and use a Taylor expansion of the distribution function F_0 to get that

$$\begin{aligned} f_h(x) &= \frac{1}{2h}\left(F_0(x+h) - F_0(x-h)\right) \\ &= \frac{1}{2h}\left(2hf_0(x) + \frac{h^3}{3}f_0''(x) + R_0(x,h)\right) \\ &= f_0(x) + \frac{h^2}{6}f_0''(x) + R_1(x,h) \end{aligned}$$

where $R_1(x,h) = o(h^2)$. One should note how the quadratic terms in h in the Taylor expansion canceled. This gives the following formula for the squared bias of $\hat{f}_h$.

$$\begin{aligned} \mathrm{bias}(\hat{f}_h(x))^2 &= (f_h(x) - f_0(x))^2 \\ &= \left(\frac{h^2}{6}f_0''(x) + R_1(x,h)\right)^2 \\ &= \frac{h^4}{36}f_0''(x)^2 + R(x,h) \end{aligned}$$

where $R(x,h) = o(h^4)$. For the variance we see from (2.1) that

$$V(\hat{f}_h(x)) = f_h(x)\frac{1}{2hn} - f_h(x)^2\frac{1}{n}.$$

Integrating the sum of the bias and the variance over x gives the integrated mean squared error

$$\begin{aligned}\mathrm{MISE}(h) &= \int \mathrm{bias}(\hat{f}_h(x))^2 + V(\hat{f}_h(x))\mathrm{d}x \\ &= \frac{h^4}{36}\|f_0''\|_2^2 + \frac{1}{2hn} + \int R(x,h)\mathrm{d}x - \frac{1}{n}\int f_h(x)^2\mathrm{d}x.\end{aligned}$$

If $f_h(x) \leq C$ (which happens if f_0 is bounded),

$$\int f_h(x)^2\mathrm{d}x \leq C\int f_h(x)\mathrm{d}x = C,$$

and the last term is $o((nh)^{-1})$. The second last term is $o(h^4)$ if we can interchange the limit and integration order. It is conceivable that we can do so under suitable assumptions on f_0, but we will not pursue those at this place. The sum of the two remaining and asymptotically dominating terms in the formula for MISE is

$$\mathrm{AMISE}(h) = \frac{h^4}{36}\|f_0''\|_2^2 + \frac{1}{2hn},$$

which is known as the asymptotic mean integrated squared error. Clearly, for this to be a useful formula, we must assume $\|f_0''\|_2^2 < \infty$. In this case the formula for AMISE can be used to find the asymptotic optimal tradeoff between (integrated) bias and variance. Differentiating w.r.t. h we find that

$$\mathrm{AMISE}'(h) = \frac{h^3}{9}\|f_0''\|_2^2 - \frac{1}{2h^2n},$$

and solving for $\mathrm{AMISE}'(h) = 0$ yields

$$h_n = \left(\frac{9}{2\|f_0''\|_2^2}\right)^{1/5} n^{-1/5}.$$

When AMISE is regarded as a function of h we observe that it tends to ∞ for $h \to 0$ as well as for $h \to \infty$, thus the unique stationary point h_n is a unique global minimizer. Choosing the bandwidth to be h_n will therefore minimize the asympototic mean integrated squared error, and it is in this sense an optimal choice of bandwidth.

We see how "wiggliness" of f_0 enters into the formula for the optimal bandwidth h_n via $\|f_0''\|_2$. This norm of the second derivative is precisely a quantification of how much f_0 oscillates. A large value, indicating a wiggly f_0, will drive the optimal bandwidth down whereas a small value will drive the optimal bandwidth up.

We should also observe that if we plug the optimal bandwidth into the formula for AMISE, we get

$$\begin{aligned}\mathrm{AMISE}(h_n) &= \frac{h_n^4}{36}\|f_0''\|_2^2 + \frac{1}{2h_nn} \\ &= Cn^{-4/5},\end{aligned}$$

which indicates that in terms of integrated mean squared error the rate at which we can nonparametrically estimate f_0 is $n^{-4/5}$. This should be contrasted to the common parametric rate of n^{-1} for mean squared error.

From a practical viewpoint there is one major problem with the optimal bandwidth h_n; it depends via $\|f_0''\|_2^2$ upon the unknown f_0 that we are trying to estimate. We therefore refer to

h_n as an *oracle* bandwidth – it is the bandwidth that an oracle that knows f_0 would tell us to use. In practice, we will have to come up with an estimate of $\|f_0''\|_2^2$ and plug that estimate into the formula for h_n. We pursue a couple of different options for doing so for general kernel density estimators below together with methods that do not rely on the AMISE formula.

2.3.2 ISE, MISE and MSE for kernel estimators

Bandwidth selection for general kernel estimators can be studied asymptotically just as above. To this end it is useful to formalize how we quantify the *quality* of an estimate $\hat{f}_h$. One natural quantification is the *integrated squared error*,

$$\mathrm{ISE}(\hat{f}_h) = \int (\hat{f}_h(x) - f_0(x))^2 \, \mathrm{d}x = \|\hat{f}_h - f_0\|_2^2.$$

The quality of the estimation procedure producing $\hat{f}_h$ from data can then be quantified by taking the mean ISE,

$$\mathrm{MISE}(h) = E(\mathrm{ISE}(\hat{f}_h)),$$

where the expectation integral is over the data. Using Tonelli's theorem we may interchange the expectation and the integration over x to get

$$\mathrm{MISE}(h) = \int \mathrm{MSE}_x(h) \, \mathrm{d}x$$

where

$$\mathrm{MSE}_h(x) = \mathrm{var}(\hat{f}_h(x)) + \mathrm{bias}(\hat{f}_h(x))^2.$$

is the pointwise mean squared error.

Using the same kind of Taylor expansion argument as above we can show that if K is a square integrable probability density with mean o and

$$\sigma_K^2 = \int z^2 K(z) \, \mathrm{d}z > 0,$$

then

$$\mathrm{MISE}(h) = \mathrm{AMISE}(h) + o((nh)^{-1} + h^4)$$

where the *asymptotic mean integrated squared error* is

$$\mathrm{AMISE}(h) = \frac{\|K\|_2^2}{nh} + \frac{h^4 \sigma_K^4 \|f_0''\|_2^2}{4}$$

with

$$\|g\|_2^2 = \int g(z)^2 \, \mathrm{d}z \quad (\text{squared } L_2\text{−norm}).$$

Some regularity assumptions on f_0 are necessary, and from the result we clearly need to require that f_0'' is meaningful and square integrable. However, that is also enough. See Proposition A.1 in Tsybakov (2009) for a rigorous proof.

By minimizing $\mathrm{AMISE}(h)$ we derive the optimal oracle bandwidth

$$h_n = \left(\frac{\|K\|_2^2}{\|f_0''\|_2^2 \sigma_K^4} \right)^{1/5} n^{-1/5}. \tag{2.3}$$

If we plug this formula into the formula for AMISE we arrive at the asymptotic error rate $\mathrm{AMISE}(h_n) = Cn^{-4/5}$ with a constant C depending on f_0'' and the kernel. It is noteworthy that

the asymptotic analysis can be carried out even if K is allowed to take negative values, though the resulting estimate may not be a valid density as it is. Tsybakov (2009) demonstrates how to improve on the rate $n^{-4/5}$ by allowing for kernels whose moments of order two or above vanish. Necessarily, such kernels must take negative values.

We observe that for the rectangular kernel,

$$\sigma_K^4 = \left(\frac{1}{2}\int_{-1}^{1} z^2 \, \mathrm{d}z\right)^2 = \frac{1}{9}$$

and

$$\|K\|_2^2 = \frac{1}{2^2}\int_{-1}^{1} \mathrm{d}z = \frac{1}{2}.$$

Plugging these numbers into (2.3) we find the oracle bandwidth for the rectangular kernel as derived in Section 2.3.1. For the Gaussian kernel we find that $\sigma_K^4 = 1$, while

$$\|K\|_2^2 = \frac{1}{2\pi}\int e^{-x^2} \, \mathrm{d}x = \frac{1}{2\sqrt{\pi}}.$$

2.3.3 Plug-in estimation of the oracle bandwidth

To compute $\|f_0''\|_2^2$ that enters into the formula for the asymptotically optimal bandwidth we have to know f_0 that we are trying to estimate in the first place. To resolve the circularity we will make a first guess of what f_0 is and plug that guess into the formula for the oracle bandwidth.

Our first guess is that f_0 is Gaussian with mean 0 and variance σ^2. Then

$$\begin{aligned}
\|f_0''\|_2^2 &= \frac{1}{2\pi\sigma^2}\int \left(\frac{x^2}{\sigma^4} - \frac{1}{\sigma^2}\right)^2 e^{-x^2/\sigma^2} \, \mathrm{d}x \\
&= \frac{1}{2\sigma^9\sqrt{\pi}}\frac{1}{\sqrt{\pi\sigma^2}}\int (x^4 - 2\sigma^2x^2 + \sigma^4)e^{-x^2/\sigma^2} \, \mathrm{d}x \\
&= \frac{1}{2\sigma^9\sqrt{\pi}}(\frac{3}{4}\sigma^4 - \sigma^4 + \sigma^4) \\
&= \frac{3}{8\sigma^5\sqrt{\pi}}.
\end{aligned}$$

Plugging this expression for the squared 2-norm of the second derivative of the density into the formula for the oracle bandwidth gives

$$h_n = \left(\frac{8\sqrt{\pi}\|K\|_2^2}{3\sigma_K^4}\right)^{1/5} \sigma n^{-1/5}, \tag{2.4}$$

with σ the only quantity depending on the unknown density f_0. We can now simply estimate σ, using e.g. the empirical standard deviation $\hat{\sigma}$, and plug this estimate into the formula above to get an estimate of the oracle bandwidth.

It is well known that the empirical standard deviation is sensitive to outliers, and if σ is overestimated for that reason, the bandwidth will be too large and the resulting estimate will be oversmoothed. To get more robust bandwidth selection, Silverman (1986) suggested using the interquartile range to estimate σ. In fact, he suggested estimating σ by

$$\tilde{\sigma} = \min\{\hat{\sigma}, \mathrm{IQR}/1.34\}.$$

In this estimator, IQR denotes the empirical interquartile range, and 1.34 is approximately the interquartile range, $\Phi^{-1}(0.75) - \Phi^{-1}(0.25)$, of the standard Gaussian distribution. Curiously, the interquartile range for the standard Gaussian distribution is 1.35 to two decimals accuracy, but the use of 1.34 in the estimator $\tilde{\sigma}$ has prevailed. Silverman, moreover, suggested to reduce the kernel-dependent constant in the formula (2.4) for h_n to further reduce oversmoothing.

If we specialize to the Gaussian kernel, formula (2.4) simplifies to

$$\hat{h}_n = \left(\frac{4}{3}\right)^{1/5} \tilde{\sigma} n^{-1/5},$$

with $\tilde{\sigma}$ plugged in as a robust estimate of σ. Silverman (1986) made the ad hoc suggestion to reduce the factor $(4/3)^{1/5} \simeq 1.06$ to 0.9. This results in the bandwidth estimate

$$\hat{h}_n = 0.9\tilde{\sigma} n^{-1/5},$$

which has become known as *Silverman's rule of thumb*.

Silverman's rule of thumb is the default bandwidth estimator for `density()` as implemented by the function `bw.nrd0()`. The `bw.nrd()` function implements bandwidth selection using the factor 1.06 instead of 0.9. Though the theoretical derivations behind these implementations assume that f_0 is Gaussian, either will give reasonable bandwidth selection for a range of unimodal distributions. If f_0 is multimodal, Silverman's rule of thumb is known to oversmooth the density estimate.

Instead of computing $\|f_0''\|_2^2$ assuming that the distribution is Gaussian, we can compute the norm for a pilot estimate, $\tilde{f}$, and plug the result into the formula for h_n. If the pilot estimate is a kernel estimate with kernel H and bandwidth r we get

$$\|\tilde{f}''\|_2^2 = \frac{1}{n^2 r^6} \sum_{i=1}^{n} \sum_{j=1}^{n} \int H''\left(\frac{x - x_i}{r}\right) H''\left(\frac{x - x_j}{r}\right) \mathrm{d}x.$$

The problem is, of course, that now we have to choose the pilot bandwidth r. But doing so using a simple method like Silverman's rule of thumb at this stage is typically not too bad an idea. Thus we arrive at the following plug-in procedure using the Gaussian kernel for the pilot estimate:

- Compute an estimate, $\hat{r}$, of the pilot bandwidth using Silverman's rule of thumb.
- Compute $\|\tilde{f}''\|_2^2$ using the Gaussian kernel as pilot kernel H and using the estimated pilot bandwidth $\hat{r}$.
- Plug $\|\tilde{f}''\|_2^2$ into the oracle bandwidth formula (2.3) to compute $\hat{h}_n$ for the kernel K.

Note that to use a pilot kernel different from the Gaussian we have to adjust the constant 0.9 (or 1.06) in Silverman's rule of thumb accordingly by computing $\|H\|_2^2$ and σ_H^4 and using (2.4).

Sheather and Jones (1991) took these plug-in ideas a step further and analyzed in detail how to choose the pilot bandwidth in a good and data adaptive way. The resulting method is somewhat complicated but implementable. We skip the details but simply observe that their method is implemented in R in the function `bw.SJ()`, and it can be selected when using `density()` by setting the argument `bw = "SJ"`. This plug-in method is regarded as a solid default that performs well for many different data generating densities f_0.

2.3.4 Cross-validation

An alternative to relying on the asymptotic optimality arguments for integrated mean squared error and the corresponding plug-in estimates of the bandwidth is known as *cross-validation*.

The method mimics the idea of setting aside a subset of the data set, which is then *not* used for computing an estimate but only for validating the estimator's performance. The benefit of cross-validation over simply setting aside a validation data set is that we do not "waste" any of the data points on validation only. All data points are used for the ultimate computation of the estimate. The deficit is that cross-validation is usually computationally more demanding.

Suppose that $I_1, \ldots, I_k$ form a partition of the index set $\{1, \ldots, n\}$ and define

$$I^{-i} = \bigcup_{l: i \notin I_l} I_l.$$

That is, I^{-i} contains all indices but those that belong to the set I_l containing i. In particular, $i \notin I^{-i}$. Define also $n_i = |I^{-i}|$ and

$$\hat{f}_h^{-i} = \frac{1}{h n_i} \sum_{j \in I^{-i}} K\left(\frac{x_i - x_j}{h}\right).$$

That is, $\hat{f}_h^{-i}$ is the kernel density estimate based on data with indices in I^{-i} and evaluated in x_i. Since the density estimate evaluated in x_i is not based on x_i, the quantity $\hat{f}_h^{-i}$ can be used to assess how well the density estimate computed using a bandwidth h concur with the data point x_i. This can be summarized using the log-likelihood

$$\ell_{\mathrm{CV}}(h) = \sum_{i=1}^{n} \log(\hat{f}_h^{-i}),$$

that we will refer to as the cross-validated log-likelihood, and we define the bandwidth estimate as

$$\hat{h}_{\mathrm{CV}} = \arg\max_h \ \ell_{\mathrm{CV}}(h).$$

This cross-validation based bandwidth can then be used for computing kernel density estimates using the entire data set.

If the partition of indices consists of k subsets we usually talk about k-fold cross-validation. If $k = n$ so that all subsets consist of just a single index we talk about leave-one-out cross-validation. For leave-one-out cross-validation there is only one possible partition, while for $k < n$ there are many possible partitions. Which should be chosen then? In practice, we choose the partition by sampling indices randomly without replacement into k sets of size roughly n/k.

It is also possible to use cross-validation in combination with MISE. Rewriting we find that

$$\mathrm{MISE}(h) = E(\|\hat{f}_h\|_2^2) - 2E(\hat{f}_h(X)) + E(\|f_0^2\|_2^2)$$

for X a random variable independent of the data and with distribution having density f_0. The last term does not depend upon h and we can ignore it from the point of view of minimizing MISE. For the first term we have an unbiased estimate in $\|\hat{f}_h\|_2^2$. The middle term can be estimated without bias by

$$\frac{2}{n} \sum_{i=1}^{n} \hat{f}_h^{-i},$$

and this leads to the statistic

$$\mathrm{UCV}(h) = \|\hat{f}_h\|_2^2 - \frac{2}{n} \sum_{i=1}^{n} \hat{f}_h^{-i}$$

known as the unbiased cross-validation criterion. The corresponding bandwidth estimate is

$$\hat{h}_{\text{UCV}} = \arg\min_h \ \text{UCV}(h).$$

Contrary to the log-likelihood based criterion, this criterion requires the computation of $\|\hat{f}_h\|_2^2$. Bandwidth selection using UCV is implemented in R in the function `bw.ucv()` and can be used with `density()` by setting the argument `bw = "ucv"`.

2.4 Likelihood considerations

In Section 2.3.4 the cross-validated likelihood was used for bandwidth selection. It is natural to ask why we did not go all-in and simply maximized the likelihood over all possible densities to find a maximum likelihood estimator instead of using the ad hoc idea behind kernel density estimation.

The log-likelihood

$$\ell(f) = \sum_{i=j}^{n} \log f(x_j)$$

is well defined as a function of the density f – even when f is not restricted to belong to a finite-dimensional parametric model. To investigate if a nonparametric MLE is meaningful we consider how the likelihood behaves for the densities

$$\overline{f}_h(x) = \frac{1}{nh\sqrt{2\pi}} \sum_{j=1}^{n} e^{-\frac{(x-x_j)^2}{2h^2}}$$

for different choices of h. Note that $\overline{f}_h$ is simply the kernel density estimator with the Gaussian kernel and bandwidth h.

```
## The densities can easily be implemented using the density implementation
## of the Gaussian density in R
f_h <- function(x, h) mean(dnorm(x, psi, h))
## This function does not work as a vectorized function as it is, but there is
## a convenience function, 'Vectorize', in R that turns the function into a
## function that can actually be applied correctly to a vector.
f_h <- Vectorize(f_h)
```

Figure 2.5 shows what some of these densities look like compared to the histogram of the ψ-angle data. Clearly, for large h these densities are smooth and slowly oscillating, while as h gets smaller the densities become more and more wiggly. As $h \to 0$ the densities become increasingly dominated by tall narrow peaks around the individual data points.

The way that these densities adapt to the data points is reflected in the log-likelihood as well.

```
# To plot the log-likelihood we need to evaluate it in a grid of h-values.
hseq <- seq(1, 0.001, -0.001)
# For the following computation of the log-likelihood it is necessary
# that f_h is vectorized. There are other ways to implement this computation
# in R, and some are more efficient, but the computation of the log-likelihood
# for each h scales as O(n^2) with n the number of data points.
ll <- sapply(hseq, function(h) sum(log(f_h(psi, h))))
qplot(hseq, ll, geom = "line") + xlab("h") + ylab("Log-likelihood")
qplot(hseq, ll, geom = "line") + scale_x_log10("h") + ylab("")
```

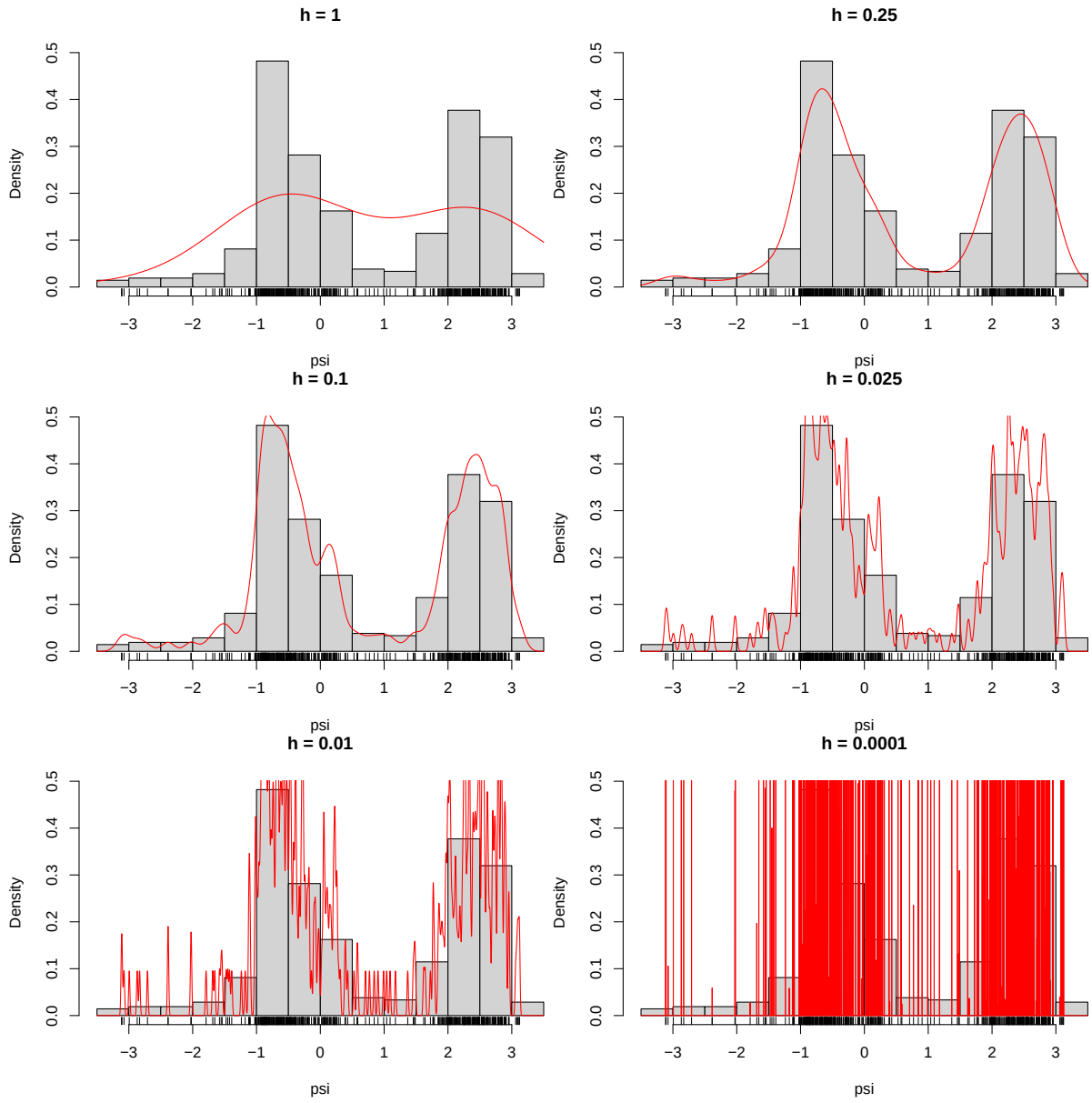


Figure 2.5: The densities $\overline{f}_h$ for different choices of h.

From Figure 2.6 it is clear that the likelihood is decreasing in h, and it appears that it is unbounded as $h \to 0$. This is most clearly seen on the figure when h is plotted on the log-scale because then it appears that the log-likelihood approximately behaves as $-\log(h)$ for $h \to 0$.

We can show that that is, indeed, the case. If $x_i \neq x_j$ when $i \neq j$

$$\begin{aligned}\ell(\overline{f}_h) &= \sum_i \log\left(1 + \sum_{j \neq i} e^{-(x_i - x_j)^2/(2h^2)}\right) - n\log(nh\sqrt{2\pi}) \\ &\sim -n\log(nh\sqrt{2\pi})\end{aligned}$$

for $h \to 0$. Hence, $\ell(\overline{f}_h) \to \infty$ for $h \to 0$. This demonstrates that the MLE of the density does not exist in the set of all distributions with densities.

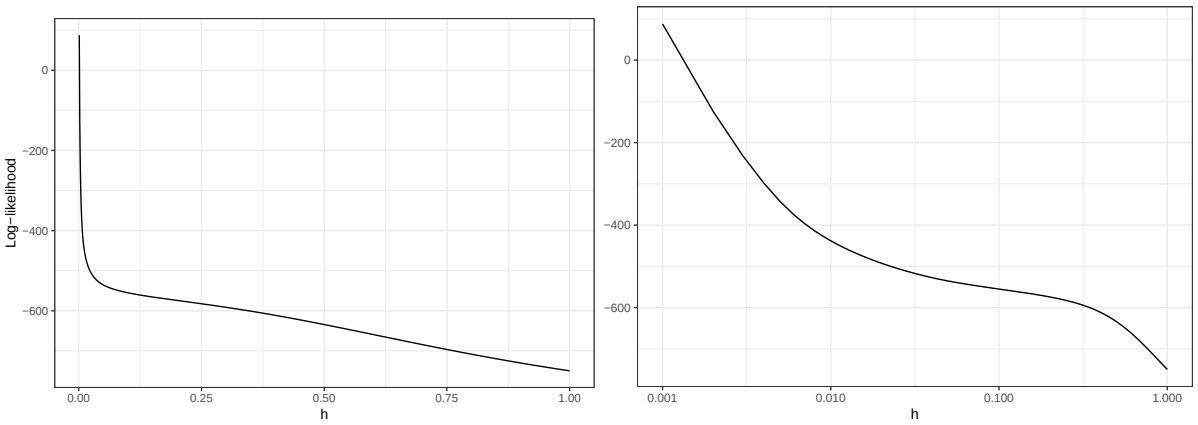


Figure 2.6: Log-likelihood, $\ell(\overline{f}_h)$, for the densities $\overline{f}_h$ as a function of h. Note the log-scale on the right.

In the sense of weak convergence it actually holds that

$$\overline{f}_h \cdot m \xrightarrow{\text{wk}} \varepsilon_n = \frac{1}{n}\sum_{j=1}^{n} \delta_{x_j}$$

for $h \to 0$. The *empirical measure* ε_n can sensibly be regarded as the nonparametric MLE of the distribution, but the empirical measure does not have a density. We conclude that we cannot directly define a sensible density estimator as a maximum-likelihood estimator.

2.4.1 Method of sieves

A sieve is a family of models, Θ_h, indexed by a real valued parameter, $h \in \mathbb{R}$, such that $\Theta_{h_1} \subseteq \Theta_{h_2}$ for $h_1 \leq h_2$. In this chapter Θ_h will denote a set of probability densities. If the increasing family of models is chosen in a sensible way, we may be able to compute the MLE

$$\hat{f}_h = \arg\max_{f \in \Theta_h} \ell(f),$$

and we may even be able to choose $h = h_n$ as a function of the sample size n such that $\hat{f}_{h_n}$ becomes a consistent estimator of f_0.

It is possible to take

$$\Theta_h = \{\overline{f}_{h'} \mid h' \leq h\}$$

with $\overline{f}_{h'}$ as defined above, in which case $\hat{f}_h = \overline{f}_h$. We will see in the following section that this is simply a kernel estimator.

A more interesting example is obtained by letting

$$\Theta_h = \left\{ x \mapsto \frac{1}{h\sqrt{2\pi}} \int e^{-\frac{(x-z)^2}{2h^2}} \mathrm{d}\mu(z) \;\middle|\; \mu \text{ a probability measure} \right\},$$

which is known as the convolution sieve. We note that $\overline{f}_h \in \Theta_h$ by taking $\mu = \varepsilon_n$, but generally $\hat{f}_h$ will be different from $\overline{f}_h$.

We will not pursue the general theory of sieve estimators, but refer to the paper Nonparametric Maximum Likelihood Estimation by the Method of Sieves by Geman and Hwang. In the following section we will work out some more practical details for a particular sieve estimator based on a basis expansion of the log-density.

2.4.2 Basis expansions

We suppose in this section that the data points are all contained in the interval $[a, b]$ for $a, b \in \mathbb{R}$. This is true for the angle data with $a = -\pi$ and $b = \pi$ no matter the size of the data set, but if f_0 does not have a bounded support it may be necessary to let a and b change with the data. However, for any fixed data set we can choose some sufficiently large a and b.

In this section the sieve will be indexed by integers, and for $h \in \mathbb{N}$ we suppose that we have chosen continuous functions $b_1, \ldots, b_h : [a, b] \to \mathbb{R}$. These will be called *basis functions*. We then define

$$\Theta_h = \left\{ x \mapsto \varphi(\boldsymbol{\beta})^{-1} \exp\left(\sum_{k=1}^{h} \beta_k b_k(x) \right) \;\middle|\; \boldsymbol{\beta} = (\beta_1, \ldots, \beta_h)^T \in \mathbb{R}^h \right\},$$

where

$$\varphi(\boldsymbol{\beta}) = \int_a^b \exp\left(\sum_{k=1}^{h} \beta_k b_k(x) \right) \mathrm{d}x.$$

The MLE over Θ_h is then given as

$$\hat{f}_h(x) = \varphi(\hat{\boldsymbol{\beta}})^{-1} \exp\left(\sum_{k=1}^{h} \hat{\beta}_k b_k(x) \right),$$

where

$$\begin{aligned} \hat{\boldsymbol{\beta}} &= \arg\max_{\boldsymbol{\beta} \in \mathbb{R}^h} \sum_{j=1}^{n} \sum_{k=1}^{h} \beta_k b_k(x_j) - \log \varphi(\boldsymbol{\beta}) \\ &= \arg\max_{\boldsymbol{\beta} \in \mathbb{R}^h} \mathbf{1}^T \mathbf{B} \boldsymbol{\beta} - \log \varphi(\boldsymbol{\beta}). \end{aligned}$$

Here $\mathbf{B}$ is the $n \times h$ matrix with $B_{jk} = b_k(x_j)$. Thus for any fixed h the model is, in fact, just an ordinary parametric exponential family, though it may not be entirely straightforward how to compute $\varphi(\boldsymbol{\beta})$.

Many basis functions are possible. Polynomials may be used, but splines are often preferred. An alternative is a selection of trigonometric functions, for instance

$$b_1(x) = \cos(x), b_2(x) = \sin(x), \ldots, b_{2h-1}(x) = \cos(hx), b_{2h}(x) = \sin(hx)$$

on the interval $[-\pi, \pi]$. In Section 1.2.1 a simple special case was actually treated corresponding to $h = 2$, where the normalization constant was identified in terms of a modified Bessel function.

It is worth remembering the following:

> "With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."
>
> — John von Neumann

The Normal-inverse Gaussian distribution has four parameters and the generalized hyperbolic distribution is an extension with five, but von Neumann was probably thinking more in terms of a spline or a polynomial expansion as above with four or five suitably chosen basis functions.

The quote is not a mathematical statement but an empirical observation. With a handful of parameters you already have a quite flexible class of densities that will fit many real data sets well. But remember that a reasonably good fit does not mean that you have found the "true" data generating model. Though data is in some situations not as scarce a resource today as when von Neumann made elephants wiggle their trunks, the quote still suggests that h should grow rather slowly with n to avoid overfitting.

2.5 Exercises

Kernel density estimation

Exercise 2.1. The Epanechnikov kernel is given by

$$K(x) = \frac{3}{4}(1 - x^2)$$

for $x \in [-1, 1]$ and 0 elsewhere. Show that this is a probability density with mean zero and compute σ_K^2 as well as $\|K\|_2^2$.

For the following exercises use the log(F12) variable as considered in Exercise A.4.

Exercise 2.2. Use `density()` to compute the kernel density estimate with the Epanechnikov kernel to the log(F12) data. Try different bandwidths.

Exercise 2.3. Implement kernel density estimation yourself using the Epanechnikov kernel. Test your implementation by comparing it to `density()` using the log(F12) data.

Benchmarking

Exercise 2.4. Construct the following vector

```
x <- rnorm(2^13)
```

Then use `microbenchmark` to benchmark the computation of

```
density(x[1:k], 0.2)
```

for `k` ranging from 2^5 to 2^{13}. Summarize the benchmarking results.

Exercise 2.5. Benchmark your own implementation of kernel density estimation using the Epanechnikov kernel. Compare the results to those obtained for `density()`.

Exercise 2.6. Experiment with different implementations of kernel evaluation in R using the Gaussian kernel and the Epanechnikov kernel. Use `microbenchmark()` to compare the different implementations.

Chapter 3

Bivariate smoothing

The focus of this chapter is on estimating how one variable, Y, is smoothly related to another, X. Thus we are directly aiming for an estimate of (aspects of) the conditional distribution of Y given X. If both variables are real valued, we can get a pretty good idea of their relation by simply looking at a scatter plot, and what we are aiming for is also often referred to as *scatter plot smoothing*. In some cases X represents a random variable, while in other cases, as the temperature example below, X represents a deterministic variable. In the example below X is time, and in other applications X could be fixed by an experimental design.

One of the examples that will be used throughout is the monthly and yearly temperatures in Nuuk, Greenland, see Vinther et al. (2006). The updated data is available from the site SW Greenland temperature data.

```
## Warning: `read_table2()` was deprecated in readr 2.0.0.
## Please use `read_table()` instead.
```

```
p_Nuuk <- ggplot(Nuuk_year, aes(x = Year, y = Temperature)) + geom_point()
p_Nuuk + geom_smooth(se = FALSE) +
  geom_smooth(
    method = "lm",
    formula = y ~ poly(x, 10),      # A degree-10 polynomial expansion
    color = "red",
    se = FALSE
  ) +
  geom_smooth(
    method = "gam",
    formula = y ~ s(x, bs = "cr"),    # A spline smoother via 'mgcv::gam()'
    color = "purple",
    se = FALSE
  )

ggplot(Nuuk, aes(x = Month, y = Temperature)) +
  geom_line(aes(group = Year), alpha = 0.3) +
  geom_point(alpha = 0.3) +
  geom_smooth(color = "red", se = FALSE)  # A spline smoother
```

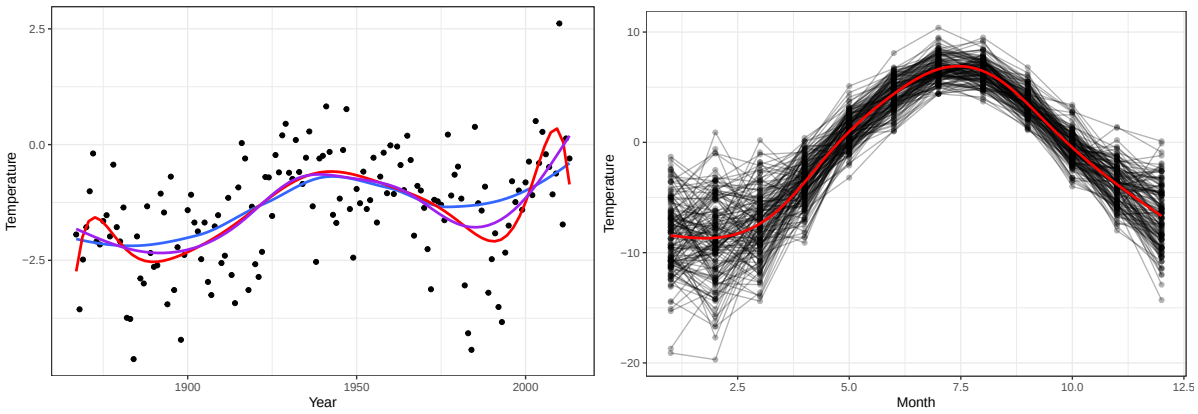


Figure 3.1: Nuuk average yearly temperature in degrees Celsius (left) smoothed using loess (black), a degree 10 polynomial (red) and a smooth spline (purple). Nuuk annual temperature cycles (right) smoothed using a spline.

3.1 Nearest neighbor smoothers

One of the most basic ideas on smoothing bivariate data is to use a running mean or moving average. This is particularly sensible when the x-values are equidistant, e.g. when the observations constitute a time series such as the Nuuk temperature data. The running mean is an example of the more general nearest neighbor smoothers.

Mathematically, the k nearest neighbor smoother in x_i is defined as

$$\hat{f}_i = \frac{1}{k} \sum_{j \in N_i} y_j$$

where N_i is the set of indices for the k nearest neighbors of x_i. This simple idea is actually very general and powerful. It works as long as the x-values lie in a metric space, and by letting k grow with n it is possible to construct consistent nonparametric estimators of regression functions, $f(x) = E(Y \mid X = x)$, under minimal assumptions. The practical problem is that k must grow slowly in high dimensions, and the estimator is not a panacea.

In this chapter we focus exclusively on x being real valued with the ordinary metric used to define the nearest neighbors. The total ordering of the real line adds a couple of extra possibilities to the definition of N_i. When k is odd, the *symmetric* nearest neighbor smoother takes N_i to consist of x_i together with the $(k-1)/2$ smaller x_j-s closest to x_i and the $(k-1)/2$ larger x_j-s closest to x_i. It is also possible to choose a one-sided smoother with N_i corresponding to the k smaller x_j-s closest to x_i, in which case the smoother would be known as a causal filter.

The symmetric definition of neighbors makes it very easy to handle the neighbors computationally; we don't need to compute and keep track of the n^2 pairwise distances between the x_i-s, we only need to sort data according to the x-values. Once data is sorted,

$$N_i = \{i - (k-1)/2, i - (k-1)/2 + 1, \ldots, i-1, i, i+1, \ldots, i + (k-1)/2\}$$

for $(k-1)/2 \leq i \leq n - (k-1)/2$. The symmetric k nearest neighbor smoother is thus a running mean of the y-values when sorted according to the x-values. There are a couple of possibilities for handling the boundaries, one being simply to not define a value of $\hat{f}_i$ outside of the interval above.

With $\hat{\mathbf{f}}$ denoting the vector of smoothed values by a nearest neighbor smoother we can observe that it is always possible to write $\hat{\mathbf{f}} = \mathbf{S}\mathbf{y}$ for a matrix $\mathbf{S}$. For the symmetric nearest neighbor

smoother and with data sorted according to the x-values, the matrix has the following band diagonal form

$$\mathbf{S} = \begin{pmatrix} \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & 0 & 0 & \dots & 0 & 0 \\ 0 & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & 0 & \dots & 0 & 0 \\ 0 & 0 & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \frac{1}{5} & \frac{1}{5} \end{pmatrix}$$

here given for $k = 5$ and with dimensions $(n-4) \times n$ due to the undefined boundary values.

3.1.1 Linear smoothers

A smoother of the form $\hat{\mathbf{f}} = \mathbf{S}\mathbf{y}$ for a *smoother matrix* $\mathbf{S}$, such as the nearest neighbor smoother, is known as a *linear smoother*. The linear form is often beneficial for theoretical arguments, and many smoothers considered in this chapter will be linear smoothers. For computing $\mathbf{f}$ there may, however, be many alternatives to forming the matrix $\mathbf{S}$ and computing the matrix-vector product. Indeed, this is often not the best way to compute the smoothed values.

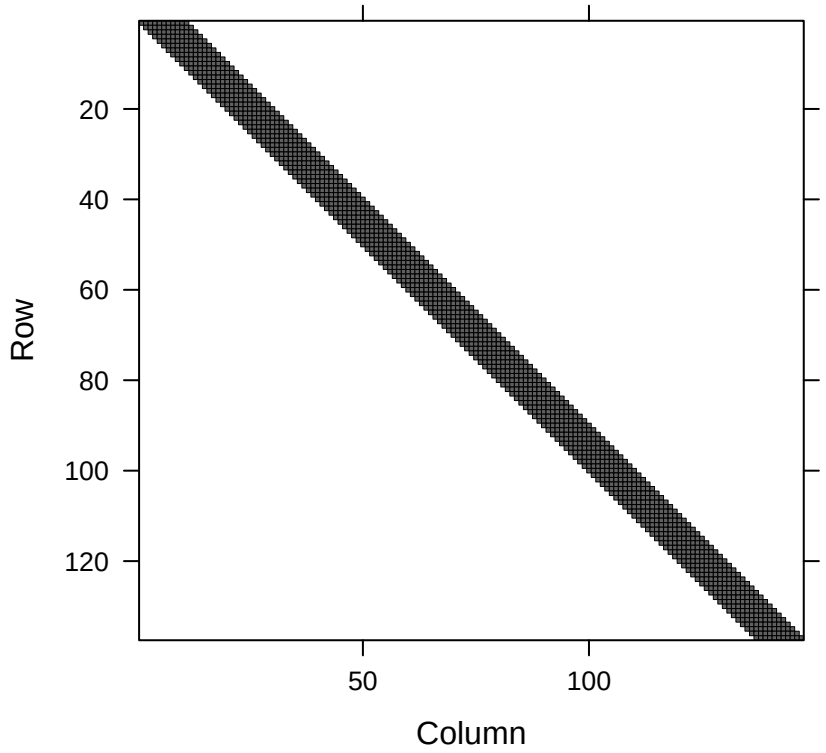
It is, on the other hand, useful to see how $\mathbf{S}$ can be constructed for the symmetric nearest neighbor smoother.

```
w <- c(rep(1/11, 11), rep(0, 147 - 10))
S <- matrix(w, 147 - 10, 147, byrow = TRUE)
```

```
## Warning in matrix(w, 147 - 10, 147, byrow = TRUE): data length [148] is not a
## sub-multiple or multiple of the number of rows [137]
```

The construction above relies on vector recycling of `w` in the construction of `S` and the fact that `w` has length $147 + 1$, which will effectively cause `w` to be translated by one to the right every time it is recycled for a new row. As seen, the code triggers a warning by R, but in this case we get what we want.

```
S
```



We can use the matrix to smooth the annual average temperature in Nuuk using a running mean with a window of $k = 11$ years. That is, the smoothed temperature at a given year is the average of the temperatures in the period from five years before to five years after. Note that to add the smoothed values to the previous plot we need to pad the values at the boundaries with NAs to get a vector of length 147.

```
# Check first if data is sorted correctly.
# The test is backwards, but confirms that data isn't unsorted :-)
is.unsorted(Nuuk_year$Year)
```

```
## [1] FALSE
```

```
f_hat <- c(rep(NA, 5), S %*% Nuuk_year$Temperature, rep(NA, 5))
p_Nuuk + geom_line(aes(y = f_hat), color = "blue")
```

3.1.2 Implementing the running mean

The running mean smoother fulfills the following identity

$$\hat{f}_{i+1} = \hat{f}_i - y_{i-(k-1)/2}/k + y_{i+(k+1)/2}/k,$$

which can be used for a much more efficient implementation than the matrix-vector multiplication. It should be emphasized again that the identity above and the implementation below assume that data is sorted according to x-values.

```
# The vector 'y' must be sorted according to the x-values
run_mean <- function(y, k) {
  n <- length(y)
  m <- floor((k - 1) / 2)
  k <- 2 * m + 1            # Ensures k to be odd and m = (k - 1) / 2
  y <- y / k
  s <- rep(NA, n)
```

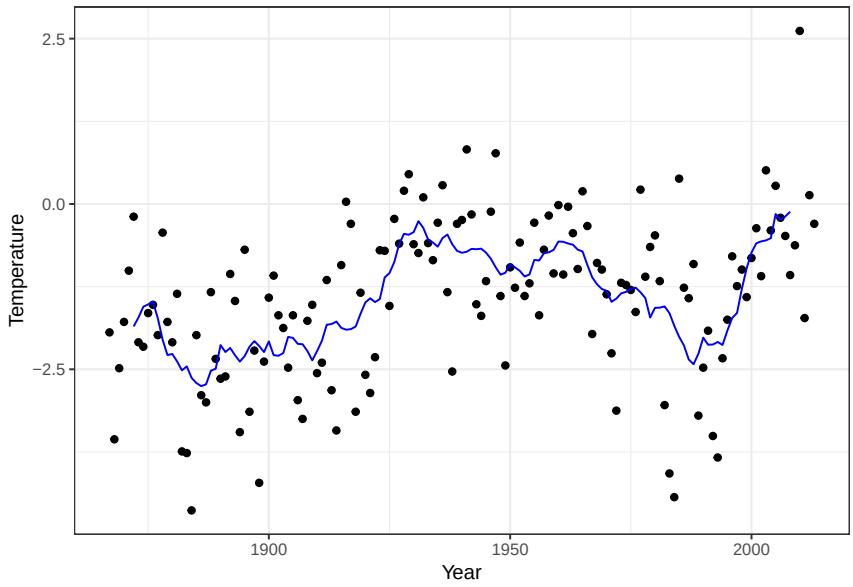


Figure 3.2: Annual average temperature in Nuuk smoothed using the running mean with $k = 11$ neighbors.

```
  s[m + 1] <- sum(y[1:k])
  for(i in (m + 1):(n - m - 1))
    s[i + 1] <- s[i] - y[i - m] + y[i + 1 + m]
  s
}
```

```
p_Nuuk + geom_line(aes(y = run_mean(Nuuk_year$Temperature, 11)), color = "blue")
```

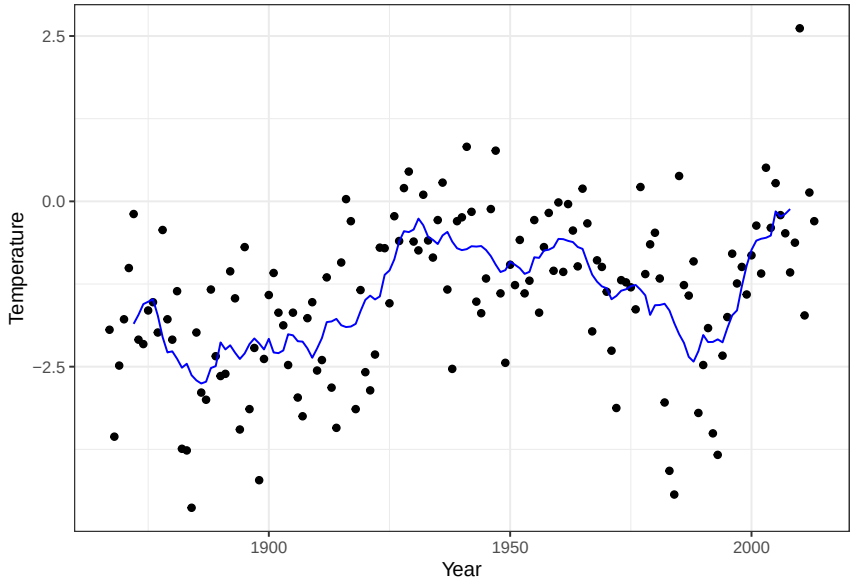


Figure 3.3: Annual average temperature in Nuuk smoothed using the running mean with $k = 11$ neighbors. This time using a different implementation than in Figure 3.2.

The R function `filter()` (from the stats package) can be used to compute running means and general moving averages using any weight vector. We compare our two implementations to `filter()`.

```
f_hat_filter <- stats::filter(Nuuk_year$Temperature, rep(1/11, 11))
range(f_hat_filter - f_hat, na.rm = TRUE)
```

```
## [1] -4.440892e-16  4.440892e-16
```

```
range(f_hat_filter - run_mean(Nuuk_year$Temperature, 11), na.rm = TRUE)
```

```
## [1] -1.332268e-15  4.440892e-16
```

Note that `filter()` uses the same boundary convention as used in `run_mean()`.

A benchmark comparison between matrix-vector multiplication, `run_mean()` and `filter()` gives the following table with median run time in microseconds.

```
##                                        expr     median
## 1                          S1 %*% y[1:512]   467.6490
## 2                         S2 %*% y[1:1024]  1928.7875
## 3                         S3 %*% y[1:2048]  7639.1670
## 4                         S4 %*% y[1:4096] 30905.4715
## 5              run_mean(y[1:512], k = 11)    91.5475
## 6             run_mean(y[1:1024], k = 11)   177.4830
## 7             run_mean(y[1:2048], k = 11)   341.4590
## 8             run_mean(y[1:4096], k = 11)   748.3645
## 9   stats::filter(y[1:512], rep(1/11, 11))    98.6885
## 10 stats::filter(y[1:1024], rep(1/11, 11))   141.6700
## 11 stats::filter(y[1:2048], rep(1/11, 11))   178.7065
## 12 stats::filter(y[1:4096], rep(1/11, 11))   374.5390
```

The matrix-vector computation is clearly much slower than the two alternatives, and the time to construct the **S**-matrix has not even been included in the benchmark above. There is also a difference in how the matrix-vector multiplication scales with the size of data compared to the alternatives. Whenever the data size doubles the run time approximately doubles for both `filter()` and `run_mean()`, while it quadruples for the matrix-vector multiplication. This shows the difference between an algorithm that scales like $O(n)$ and an algorithm that scales like $O(n^2)$ as the matrix-vector product does.

Despite that `filter()` is implementing a more general algorithm than `run_mean()`, it is still faster, which reflects that it is implemented in C and compiled.

3.1.3 Choose k by cross-validation

Cross-validation relies predictions of y_i from x_i for data points (x_i, y_i) left out of the data set when the predictor is fitted to data. Many (linear) smoothers have a natural definition of an "out-of-sample" prediction, that is, how $\hat{f}(x)$ is computed for x not in the data. If so, it becomes possible to define

$$\hat{f}_i^{-i} = \hat{f}^{-i}(x_i)$$

as the prediction at x_i using the smoother computed from data with (x_i, y_i) excluded. However, here we derectly *define*

$$\hat{f}_i^{-i} = \sum_{j \neq i} \frac{S_{ij} y_j}{1 - S_{ii}}$$

for any linear smoother. This definition concurs with the "out-of-sample" predictor in x_i for most smoothers, but this has to be verified case-by-case.

The running mean is a little special in this respect. In the previous section, the running mean was only considered for odd k and using a symmetric neighbor definition. This is convenient when considering the running mean *in the observations* x_i. When considering the running mean in any other point, a symmetric neighbor definition works better with an even k. This is exactly what the definition of $\hat{f}_i^{-i}$ above amounts to. If $\mathbf{S}$ is the running mean smoother matrix for an odd k, then $\hat{f}_i^{-i}$ corresponds to symmetric $(k-1)$-nearest neighbor smoothing excluding (x_i, y_i) from the data.

Using the definition above, we get that the *leave-one-out cross-validation* squared error criterion becomes

$$\text{LOOCV} = \sum_{i=1}^{n} (y_i - \hat{f}_i^{-i})^2 = \sum_{i=1}^{n} \left(\frac{y_i - \hat{f}_i}{1 - S_{ii}} \right)^2 .$$

The important observation from the identity above is that LOOCV can be computed without actually computing all the $\hat{f}_i^{-i}$.

For the running mean, all diagonal elements of the smoother matrix are identical. We disregard boundary values (with the `NA` value), so to get a comparable quantity across different choices of k we use `mean()` instead of `sum()` in the implementation.

```
loocv <- function(k, y) {
  f_hat <- run_mean(y, k)
  mean(((y - f_hat) / (1 - 1/k))^2, na.rm = TRUE)
}
```

```
k <- seq(3, 40, 2)
CV <- sapply(k, loocv, y = Nuuk_year$Temperature)
k_opt <- k[which.min(CV)]
qplot(k, CV) + geom_line() + geom_vline(xintercept = k_opt, color = "red")
```

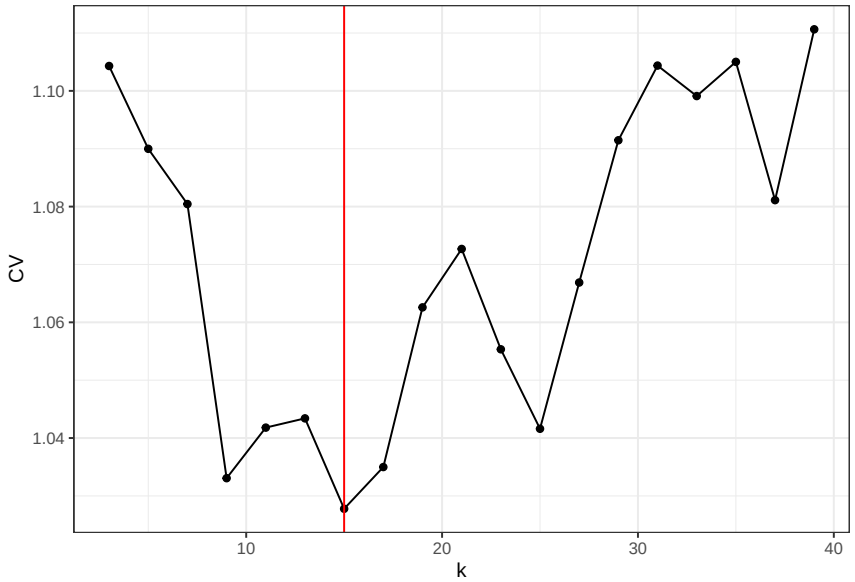


Figure 3.4: The leave-one-out cross-validation criterion for the running mean as a function of the number of neighbors k.

The optimal choice of k is 15, but the LOOCV criterion jumps quite a lot up and down with changing neighbor size, and $k = 9$ as well as $k = 25$ give rather low values as well.

```
p_Nuuk +
  geom_line(aes(y = run_mean(Nuuk_year$Temperature, 9)), color = "red") +
  geom_line(aes(y = run_mean(Nuuk_year$Temperature, k_opt)), color = "blue") +
  geom_line(aes(y = run_mean(Nuuk_year$Temperature, 25)), color = "purple")
```

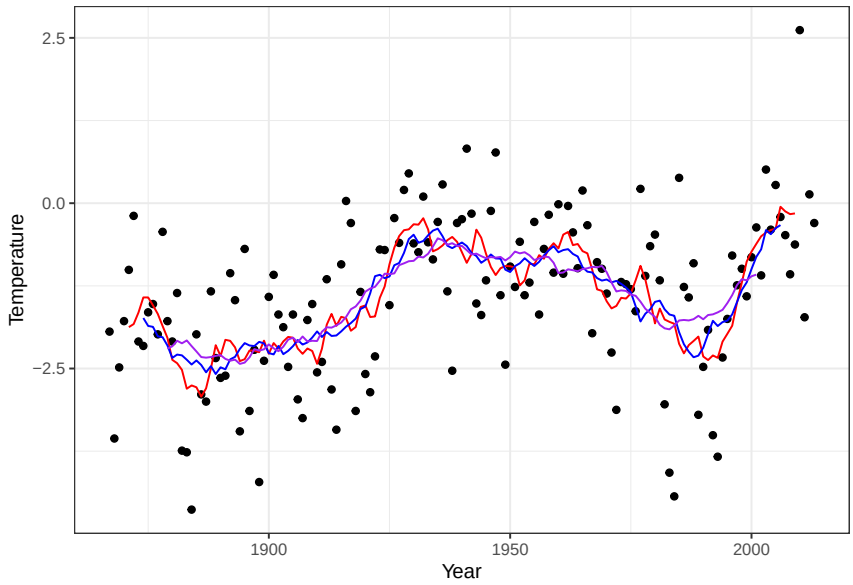


Figure 3.5: The k-nearest neighbor smoother with the optimal choice of k based on LOOCV (blue) and with $k = 9$ (red) and $k = 25$ (purple).

3.2 Kernel methods

3.2.1 Nadaraya–Watson kernel smoothing

The section before introduced the basic idea of nearest neighbor smoothing using a fixed number of neighbors. A very similar idea is to use a fixed neighborhood, B_i, say, around x_i. This leads to the estimator

$$\hat{f}_i = \frac{\sum_{j=1}^n y_j 1_{B_i}(x_j)}{\sum_{j=1}^n 1_{B_i}(x_j)},$$

which is simply the average of the y-s for which the corresponding x-s fall in the neighborhood B_i of x_i. Note that contrary to the nearest neighbor estimator, the denominator now also depends on the x-s.

In a metric space the natural choice of B_i is the ball, $B(x_i, h)$, around x_i with some radius h. In $\mathbb{R}$ with the usual metric (or even $\mathbb{R}^p$ equipped with any norm-induced metric) we have that

$$1_{B(x_i,h)}(x) = 1_{B(0,1)}\left(\frac{x - x_i}{h}\right),$$

thus since $B(0,1) = [-1,1]$ in $\mathbb{R}$

$$\hat{f}_i = \frac{\sum_{j=1}^n y_j 1_{[-1,1]}\left(\frac{x_j - x_i}{h}\right)}{\sum_{l=1}^n 1_{[-1,1]}\left(\frac{x_l - x_i}{h}\right)}.$$

This nonparametric estimator of the conditional expectation $E(Y \mid X = x_i)$ is closely related to the kernel density estimator with the rectangular kernel, see Section 2.2, and just as for

this estimator there is a natural generalization allowing for arbitrary kernels instead of the indicator function $1_{[-1,1]}$.

With $K : \mathbb{R} \mapsto \mathbb{R}$ a fixed kernel the corresponding kernel smoother with bandwidth h becomes

$$\hat{f}_i = \frac{\sum_{j=1}^n y_j K\left(\frac{x_j - x_i}{h}\right)}{\sum_{l=1}^n K\left(\frac{x_l - x_i}{h}\right)}.$$

This smoother is known as the Nadaraya–Watson kernel smoother or Nadaraya–Watson estimator. See Exercise 3.1 for an additional perspective based on bivariate kernel density estimation.

The Nadaraya–Watson kernel smoother can be implemented in much the same way as the kernel density estimator, and the run time will inevitably scale like $O(n^2)$ unless we exploit special properties of the kernel or use approximation techniques such as binning. In this section we will focus on implementing bandwidth selection rather than the kernel smoother itself. The basic kernel smoothing computation is also implemented in the `ksmooth` function from the stats package.

```
f_hat <- ksmooth(
  Nuuk_year$Year,
  Nuuk_year$Temperature,
  kernel = "normal",
  bandwidth = 10,
  x.points = Nuuk_year$Year
)
p_Nuuk + geom_line(aes(y = f_hat$y), color = "blue")
```

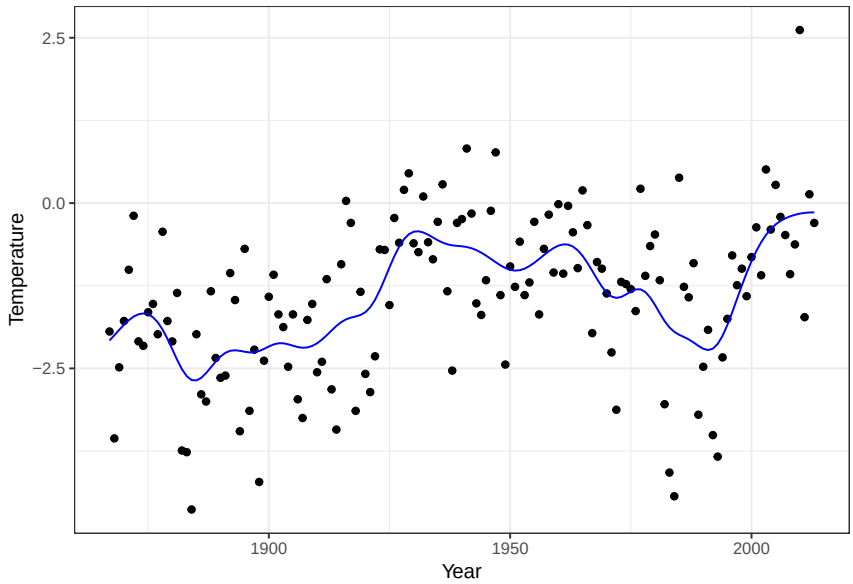


Figure 3.6: Nadaraya–Watson kernel smoother of the annual average temperature in Nuuk computed using 'ksmooth' with the Gaussian kernel and bandwidth 10.

The kernel smoother is clearly a linear smoother, and we will implement its computation and the bandwidth selection using LOOCV by direct computation of the the smoother matrix $\mathbf{S}$, which is given by

$$S_{ij} = \frac{K\left(\frac{x_j - x_i}{h}\right)}{\sum_{l=1}^n K\left(\frac{x_l - x_i}{h}\right)}.$$

The rows sum to 1 by construction, and the diagonal elements are

$$S_{ii} = \frac{K(0)}{\sum_{l=1}^{n} K\left(\frac{x_l - x_i}{h}\right)}.$$

The computation of $\mathbf{S}$ for the Gaussian kernel is implemented using `outer` and `rowSums`.

```
kern <- function(x) exp(- x^2 / 2)  # The Gaussian kernel
Kij <- outer(Nuuk_year$Year, Nuuk_year$Year, function(x, y) kern((x - y) / 10))
S <- Kij / rowSums(Kij)
f_hat <- S %*% Nuuk_year$Temperature
p_Nuuk + geom_line(aes(y = f_hat), color = "blue")
```

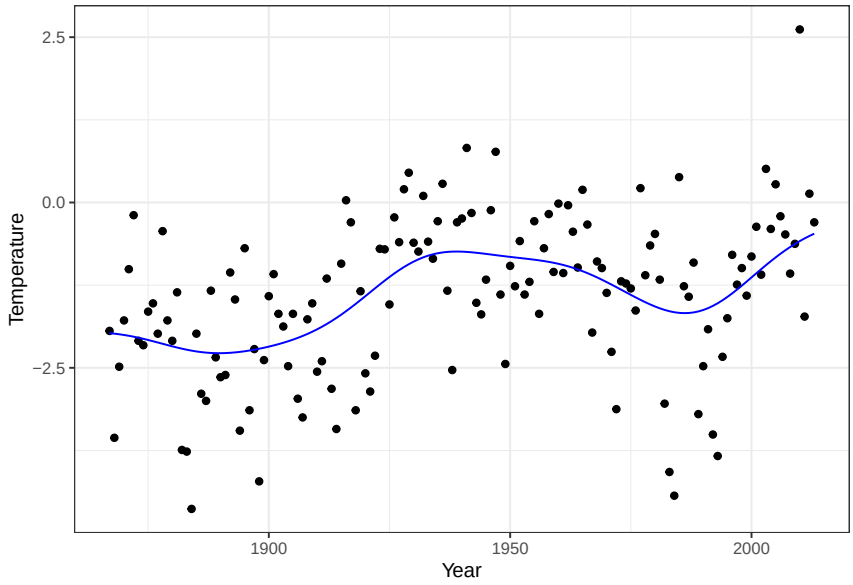


Figure 3.7: Nadaraya–Watson kernel smoother of the annual average temperature in Nuuk computed using the smoother matrix and the Gaussian kernel with bandwidth 10.

The implementation above will work with any kernel and any sequence of x-s. In this example, the kernel is symmetric and the x-s are equidistant. Exercise 3.2 explores how to exploit this in the computation of the smoother matrix as well as the diagonal elements of the smoother matrix.

The smoother computed using `ksmooth` with bandwidth 10, as shown in Figure 3.6, is different from the smoother computed directly from the smoother matrix, see Figure 3.7. Though both computations use the Gaussian kernel and allegedly a bandwidth of 10, the resulting smoothers differ because `ksmooth` internally rescales the bandwidth. The rescaling amounts to multiplying h by the factor 0.3706506, which will make the kernel have quartiles in $\pm 0.25h$ (see also `?ksmooth`).

The LOOCV computation is implemented as a function that computes the smoother matrix and the corresponding LOOCV value as a function of the bandwidth. For comparison with previous implementations the mean is computed instead of the sum.

```
loocv <- function(h) {
  Kij <- outer(Nuuk_year$Year, Nuuk_year$Year, function(x, y) kern((x - y) / h))
  S <- Kij / rowSums(Kij)
  mean(((Nuuk_year$Temperature - S %*% Nuuk_year$Temperature) / (1 - diag(S)))^2)
}
```

```
h <- seq(1, 5, 0.05)
CV <- sapply(h, loocv)
h_opt <- h[which.min(CV)]
qplot(h, CV) + geom_line() + geom_vline(xintercept = h_opt, color = "red")
```

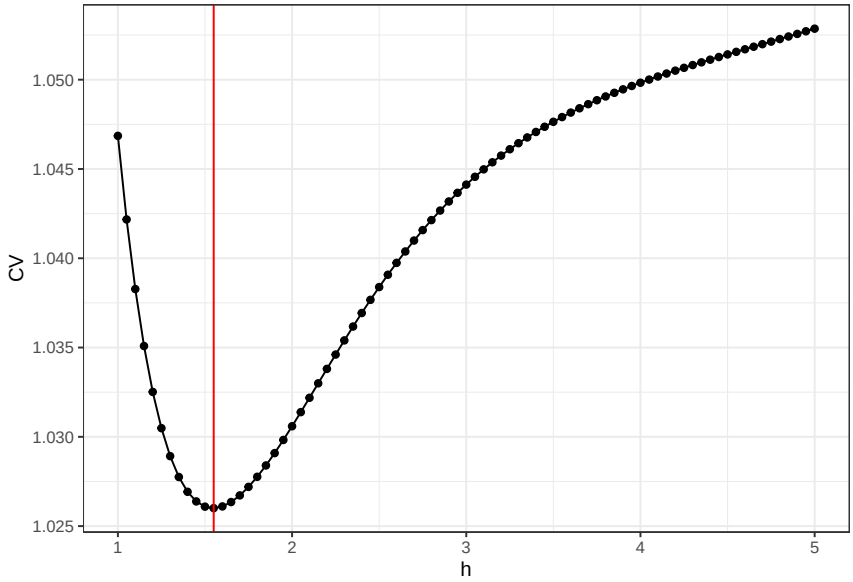


Figure 3.8: The leave-one-out cross-validation criterion for the kernel smoother using the Gaussian kernel as a function of the bandwidth h.

The optimal bandwith is 1.55. We compute the resulting optimal smoother and compare it to the smoother computed using `ksmooth`.

```
Kij <- outer(Nuuk_year$Year, Nuuk_year$Year, function(x, y) kern((x - y) / h_opt))
S <- Kij / rowSums(Kij)
f_hat <- S %*% Nuuk_year$Temperature
f_hat_ksmooth <- ksmooth(
  Nuuk_year$Year, Nuuk_year$Temperature,
  kernel = "normal",
  bandwidth = h_opt /  0.3706506,  # Rescaling!
  x.points = Nuuk_year$Year
)
range(f_hat - f_hat_ksmooth$y)
```

```
## [1] -4.535467e-05  4.598776e-05
```

The differences are of the order 10^{-5}, which is small but larger than can be explained by rounding errors alone. In fact, `ksmooth` truncates the tails of the Gaussian kernel to 0 beyond $4h$. This is where the kernel becomes less than $0.000335 \times K(0) = 0.000134$, which for practical purposes effectively equals zero. The truncation explains the relatively large differences between the two results that should otherwise be equivalent. It is an example where the approximate solution computed by `ksmooth` is acceptable because it substantially reduces the run time.

```
p_Nuuk +
  geom_line(aes(y = run_mean(Nuuk_year$Temperature, 9)), color = "red") +
  geom_line(aes(y = f_hat), color = "blue")
```

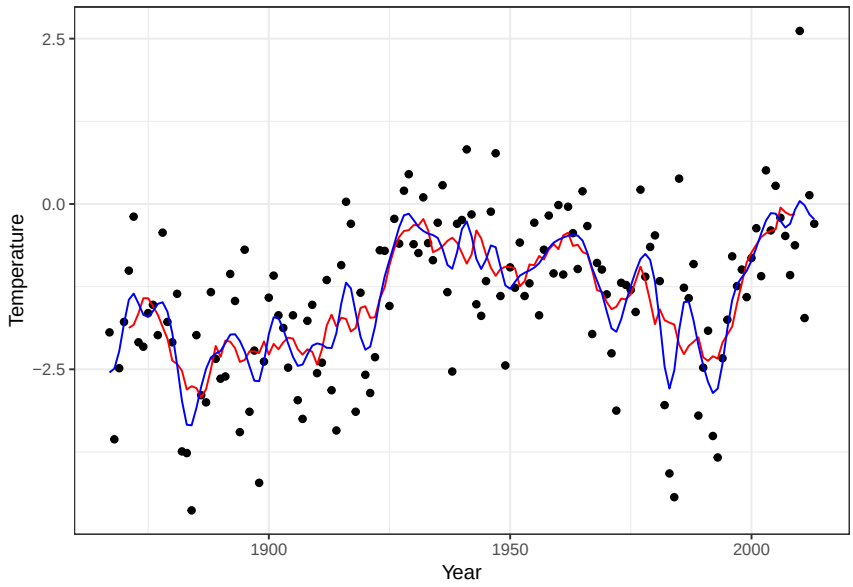


Figure 3.9: Nadaraya–Watson kernel smoother of the annual average temperature in Nuuk for the optimal bandwidth using LOOCV (blue) compared to the k-nearest neighbor smoother with $k = 9$ (red).

Figure 3.9 shows the optimal kernel smoother, which is actually somewhat wiggly. It is locally more smooth than the k-nearest neighbor smoother but overall comparable to this smoother with $k = 9$.

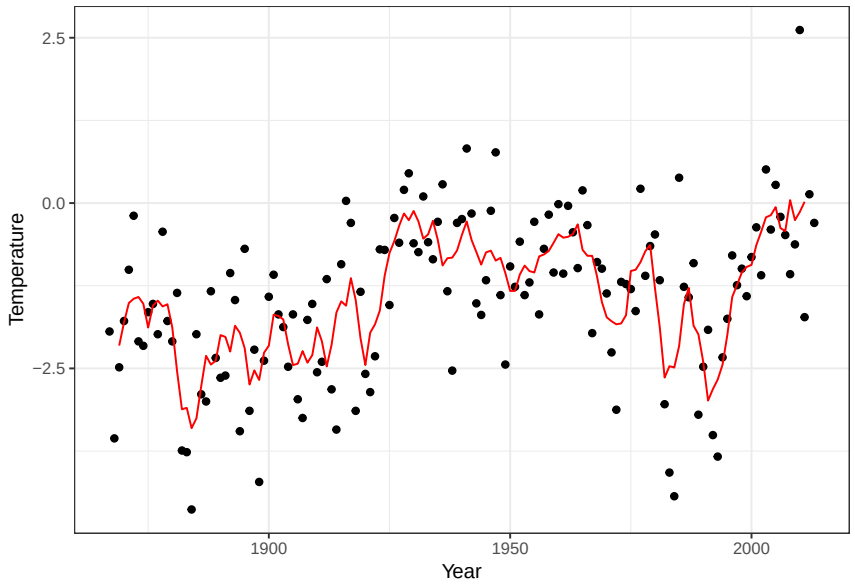
3.2.2 Local regression smoothers

3.3 Sparse linear algebra

```
library(Matrix)
bandSparse(15, 15, seq(-2, 2))
```

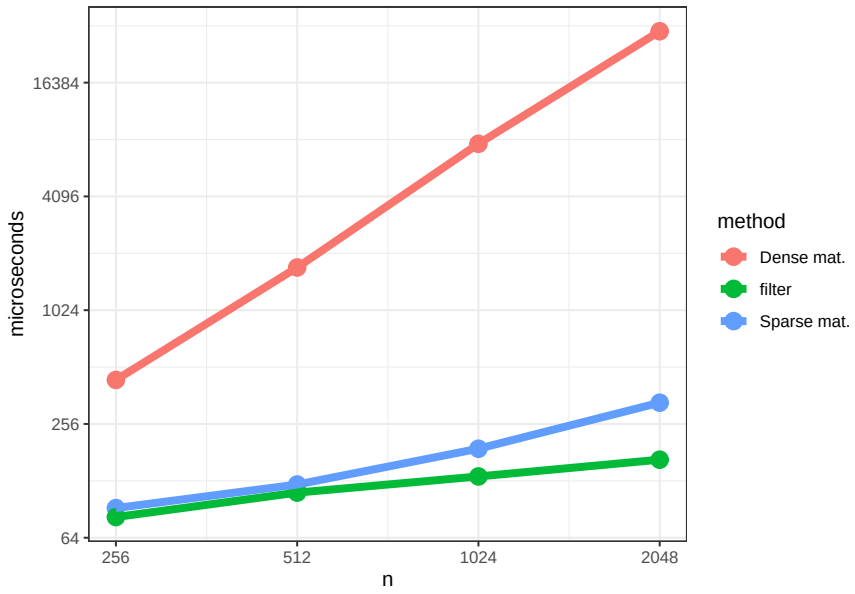
```
## 15 x 15 sparse Matrix of class "ngCMatrix"
##
##  [1,] | | . . . . . . . . . . . . .
##  [2,] | | | . . . . . . . . . . . .
##  [3,] | | | | . . . . . . . . . . .
##  [4,] . | | | | . . . . . . . . . .
##  [5,] . . | | | | . . . . . . . . .
##  [6,] . . . | | | | . . . . . . . .
##  [7,] . . . . | | | | . . . . . . .
##  [8,] . . . . . | | | | . . . . . .
##  [9,] . . . . . . | | | | . . . . .
## [10,] . . . . . . . | | | | . . . .
## [11,] . . . . . . . . | | | | . . .
## [12,] . . . . . . . . . | | | | . .
## [13,] . . . . . . . . . . | | | | .
## [14,] . . . . . . . . . . . | | | |
## [15,] . . . . . . . . . . . . | | |
```

```
K <- bandSparse(n, n, seq(-2, 2))
weights <- c(1/3, 1/4, rep(1/5, n - 4), 1/4, 1/3)
weights <- c(NA, NA, rep(1/5, n - 4), NA, NA)
p_Nuuk <- ggplot(Nuuk_year, aes(Year, Temperature)) + geom_point()
p_Nuuk +
  geom_line(aes(y = as.numeric(K %*% Nuuk_year$Temperature) * weights),
            color = "red")
```



When the smoother matrix is *sparse*, matrix multiplication can be much faster.

We will present some benchmark comparisons below. First we compare the run time for the matrix multiplication `as.numeric(K %*% Nuuk_year$Temperature) * weights` using a sparse matrix (as above) with the run time using a dense matrix. The dense matrix is given as `Kdense = as.matrix(K)`. These run times are compared to using `filter()`. In all computations, $k = 5$.



The difference in slopes between dense and sparse matrix multiplication should be noted. This is the difference between $O(n^2)$ and $O(n)$ run time. The run time for the dense matrix multiplication will not change with k. For the other two it will increase (linearly) with

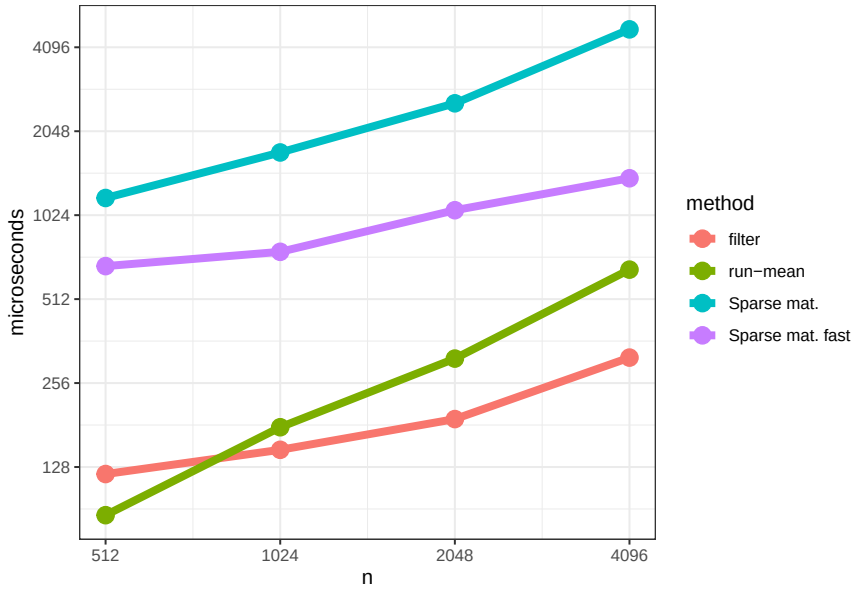
increasing k.

For smoothing only once with a given smoother matrix the time to construct the matrix should also be taken into account for fair comparison with `filter()`. It turns out that the function `bandSparse` is not optimized for the specific running mean banded matrix, and a faster C++ function for this job is given below.

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
List fastBand(int n, int k) {
  int N = (2 * k + 1) * (n - 2 * k) + 3 * k * k + k;
  int iter = 0;
  IntegerVector i(N), p(n + 1);
  for(int col = 0; col < n; ++col) {
    p[col] = iter;
    for(int r = std::max(col - k, 0); r < std::min(col + k + 1, n); ++r) {
      i[iter] = r;
      ++iter;
    }
  }
  p[n] = N;
  return List::create(_["i"] = i, _["p"] = p);
}
```

And then R function.

```
bandSparseFast <- function(n, k) {
  n <- as.integer(n)
  k <- as.integer(k)
  tmp <- fastBand(n, k)
  new("ngCMatrix",
      i = tmp$i,
      p = tmp$p,
      Dim = c(n, n))
}
```

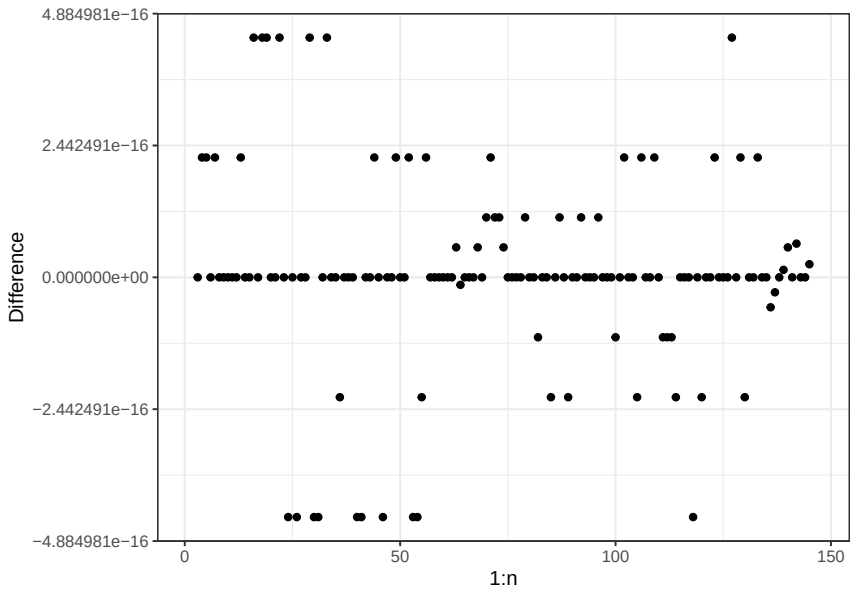


The construction of the sparse matrix turns out to take up much more time than the matrix-vector multiplication. The run time is still $O(n)$, but the constant is of the order of a factor 16 larger than for `filter()`. With the faster construction of the sparse matrix, the constant is reduced to being of the order 5 larger than for `filter()`. For small n there is some overhead from the constructor of the sparse matrix object even for the faster algorithm.

If you implement an algorithm (like a smoother) using linear algebra (e.g. a matrix-vector product) then sparse matrix numerical methods can be useful compared to dense matrix numerical methods. The Matrix package for R implements sparse matrices, and you should always attempt to use methods for constructing the sparse matrix that avoid dense intermediates. But even with a special purpose constructor of a sparse band matrix, sparse linear algebra cannot compete with optimized special purpose algorithms like `filter()` or a C++ implementation of `run_mean()`. The `filter()` function even works more generally for kernels (weights) with *equidistant* data.

We conclude this section by verifying that `filter()` actually computes the running mean up to numerical errors.

```
qplot(1:n,
      as.numeric(K %*% Nuuk_year$Temperature) * weights -
        c(stats::filter(Nuuk_year$Temperature, rep(1/5, 5)))) +
  scale_y_continuous("Difference")
```




```
all(as.numeric(K %*% Nuuk_year$Temperature) * weights ==
      c(stats::filter(Nuuk_year$Temperature, rep(1/5, 5))))
```

```
## [1] FALSE
```

```
all.equal(as.numeric(K %*% Nuuk_year$Temperature) * weights,
          c(stats::filter(Nuuk_year$Temperature, rep(1/5, 5))))
```

```
## [1] TRUE
```

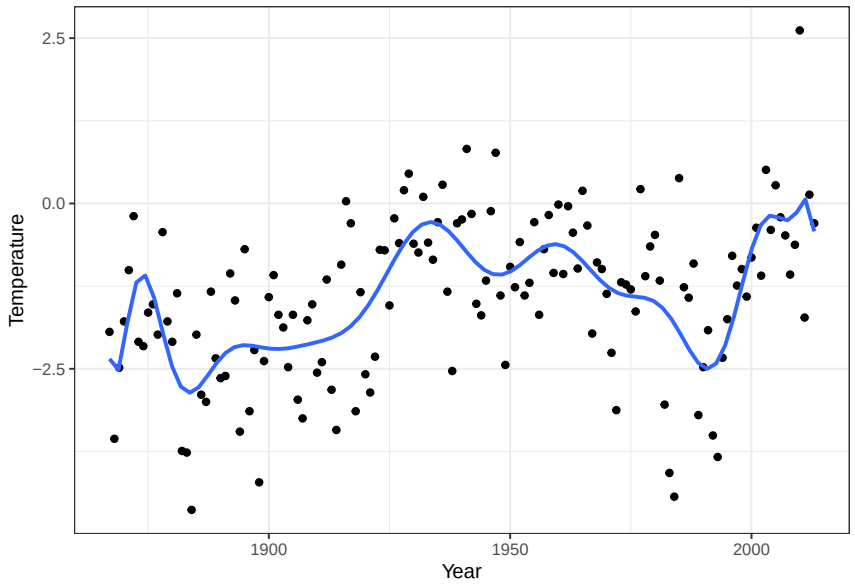
```
identical(as.numeric(K %*% Nuuk_year$Temperature) * weights,
          c(stats::filter(Nuuk_year$Temperature, rep(1/5, 5))))
```

```
## [1] FALSE
```

3.4 Orthogonal basis expansions

3.4.1 Polynomial expansions

Degree 19 polynomial fitted to the temperature data.



We can extract the model matrix from the lm-object.

```
intercept <- rep(1/sqrt(n), n)  # To make intercept column have norm one
polylm <- lm(Temperature ~ intercept + poly(Year, 19) - 1, data = Nuuk_year)
Phi <- model.matrix(polylm)
```

The model matrix is (almost) orthogonal, and estimation becomes quite simple. With an orthogonal model matrix the normal equation reduces to the estimate

$$\hat{\beta} = \Phi^T Y$$

since $\Phi^T\Phi = I$. The predicted (or fitted) values are $\Phi\Phi^T Y$ with smoother matrix $\mathbf{S} = \Phi\Phi^T$ being a projection.

```
(t(Phi) %*% Nuuk_year$Temperature)[1:10, 1]
```

```
##       intercept poly(Year, 19)1 poly(Year, 19)2 poly(Year, 19)3 poly(Year, 19)4
##     -17.2469646       4.9002430      -1.7968913       0.8175400       5.9668689
## poly(Year, 19)5 poly(Year, 19)6 poly(Year, 19)7 poly(Year, 19)8 poly(Year, 19)9
##       1.4265091      -1.9258864      -0.2523581      -2.1355117      -0.8046267
```

```
coef(polylm)[1:10]
```

```
##       intercept poly(Year, 19)1 poly(Year, 19)2 poly(Year, 19)3 poly(Year, 19)4
##     -17.2469646       4.9002430      -1.7968913       0.8175400       5.9668689
## poly(Year, 19)5 poly(Year, 19)6 poly(Year, 19)7 poly(Year, 19)8 poly(Year, 19)9
##       1.4265091      -1.9258864      -0.2523581      -2.1355117      -0.8046267
```

With homogeneous variance

$$\hat{\beta}_i \overset{\text{approx}}{\sim} \mathcal{N}(\beta_i, \sigma^2),$$

and for $\beta_i = 0$ we have $P(|\hat{\beta}_i| \geq 1.96\sigma) \simeq 0.05$.

Thresholding:

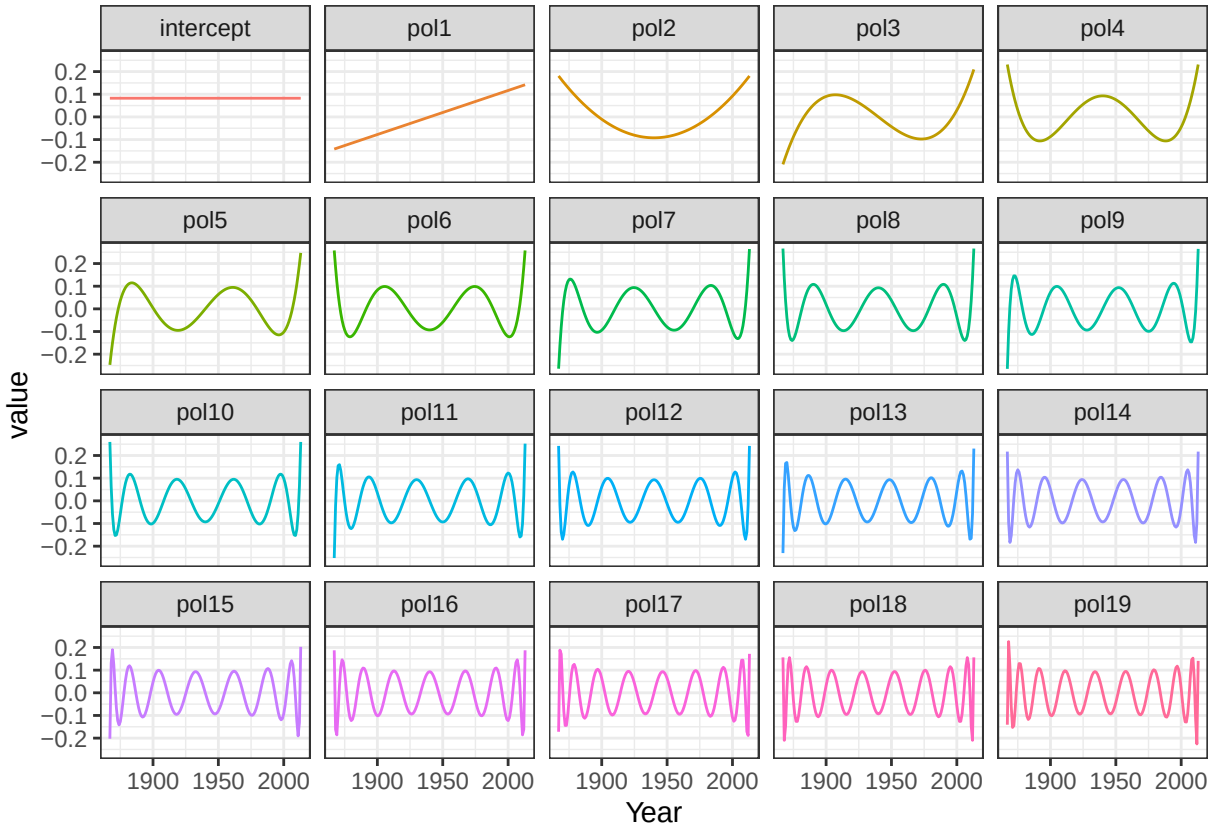
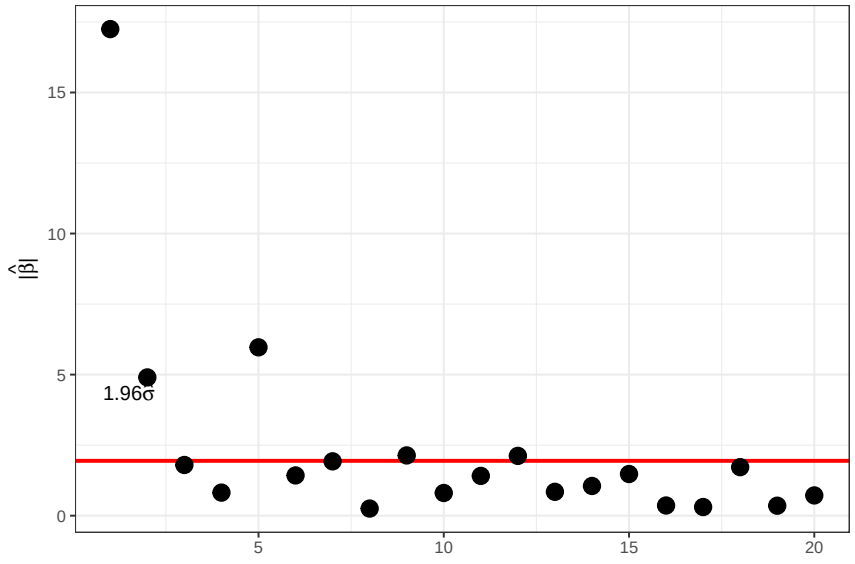


Figure 3.10: The model matrix columns as functions



3.4.2 Fourier expansions

Introducing

$$x_{k,m} = \frac{1}{\sqrt{n}} e^{2\pi i k m/n},$$

then

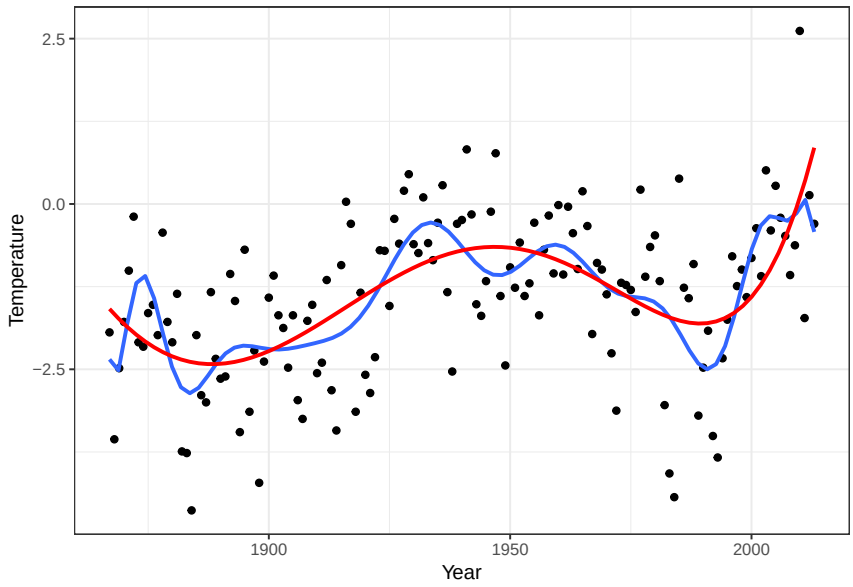


Figure 3.11: Polynomial fit using all 19 basis functions (blue) and using a degree 5 polynomial (red).

$$\sum_{k=0}^{n-1} |x_{k,m}|^2 = 1$$

and for $m_1 \neq m_2$

$$\sum_{k=0}^{n-1} x_{k,m_1} \overline{x_{k,m_2}} = 0$$

Thus $\Phi = (x_{k,m})_{k,m}$ is an $n \times n$ unitary matrix;

$$\Phi^* \Phi = I$$

where Φ^* is the conjugate transposed of Φ.

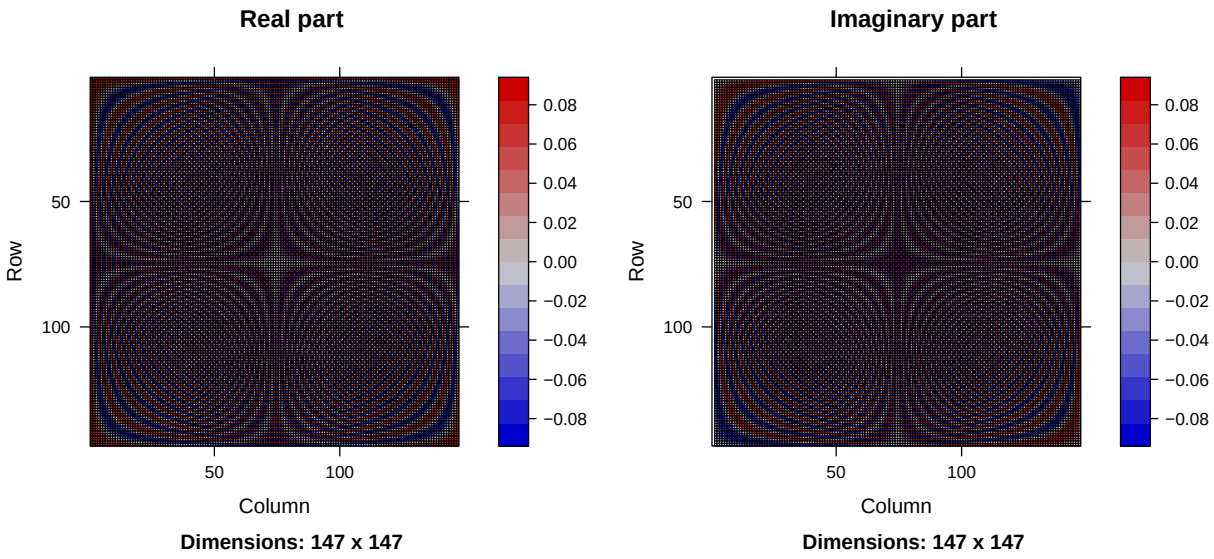
$\hat{\beta} = \Phi^* y$ is the *discrete Fourier transform* of y. It is the basis coefficients in the orthonormal basis given by Φ;

$$y_k = \frac{1}{\sqrt{n}} \sum_{m=0}^{n-1} \hat{\beta}_m e^{2\pi i k m / n}$$

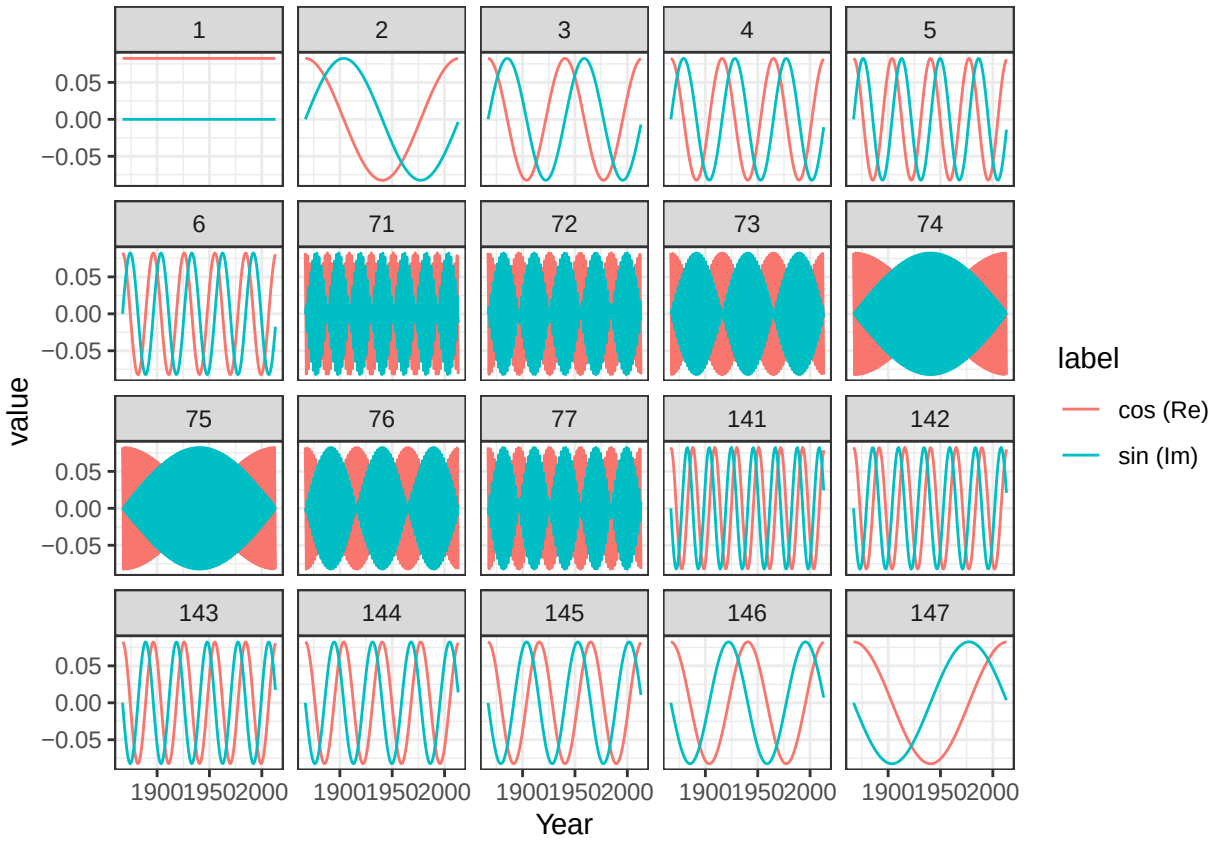
or $y = \Phi \hat{\beta}$.

```
Phi <- outer(
  0:(n - 1),
  0:(n - 1),
  function(k, m) exp(2 * pi * 1i * (k * m) / n) / sqrt(n)
)
```

The matrix Φ generates an interesting pattern.



Columns in the matrix Φ:



We can estimate by matrix multiplication

```
betahat <- Conj(t(Phi)) %*% Nuuk_year$Temperature # t(Phi) = Phi for Fourier bases
betahat[c(1, 2:4, 73, n:(n - 2))]
```

```
## [1] -17.2469646+0.0000000i  -2.4642887+2.3871189i   3.5481329+0.9099226i
## [4]   1.6721444+0.7413580i   0.0321232+0.7089991i  -2.4642887-2.3871189i
## [7]   3.5481329-0.9099226i   1.6721444-0.7413580i
```

For real y it holds that $\hat{\beta}_0$ is real, and the symmetry

$$\hat{\beta}_{n-m} = \hat{\beta}_m^*$$

holds for $m = 1, \ldots, n-1$. (For n even, $\hat{\beta}_{n/2}$ is real too).

Modulus distribution:

Note that for $m \neq 0, n/2$, $\beta_m = 0$ and $y \sim \mathcal{N}(\Phi\beta, \sigma^2 I_n)$ then

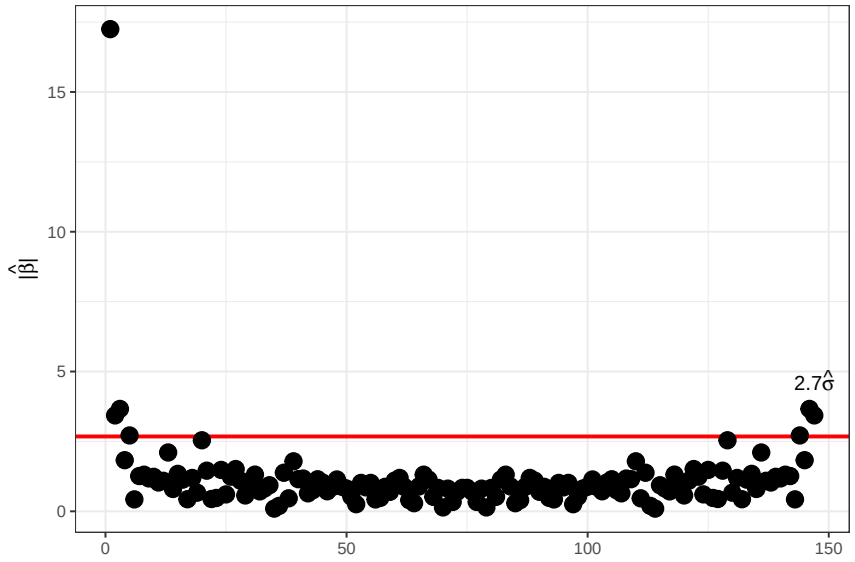
$$(\mathrm{Re}(\hat{\beta}_m), \mathrm{Im}(\hat{\beta}_m))^T \sim \mathcal{N}\left(0, \frac{\sigma^2}{2} I_2\right),$$

hence

$$|\hat{\beta}_m|^2 = \mathrm{Re}(\hat{\beta}_m)^2 + \mathrm{Im}(\hat{\beta}_m)^2 \sim \frac{\sigma^2}{2}\chi_2^2,$$

that is, $P(|\hat{\beta}_m| \geq 1.73\sigma) = 0.05$. There is a clear case of multiple testing if we use this threshold at face value, and we would expect around $0.05 \times n/2$ false positive if there is no signal at all. Lowering the probability using the Bonferroni correction yields a threshold of around 2.7σ instead.

Thresholding Fourier:



The coefficients are not independent (remember the symmetry), and one can alternatively consider

$$\hat{\gamma}_m = \sqrt{2}\mathrm{Re}(\hat{\beta}_m) \quad \text{and} \quad \hat{\gamma}_{n'+m} = -\sqrt{2}\mathrm{Im}(\hat{\beta}_m)$$

for $1 \leq m < n/2$. Here $n' = \lfloor n/2 \rfloor$. Here, $\hat{\gamma}_0 = \hat{\beta}_0$, and $\hat{\gamma}_{n/2} = \hat{\beta}_{n/2}$ for n even.

These coefficients are the coefficients in a real cosine, $\sqrt{2}\cos(2\pi km/n)$, and sine, $\sqrt{2}\sin(2\pi km/n)$, basis expansion, and they are i.i.d. $\mathcal{N}(0, \sigma^2)$ distributed.

Thresholding Fourier:

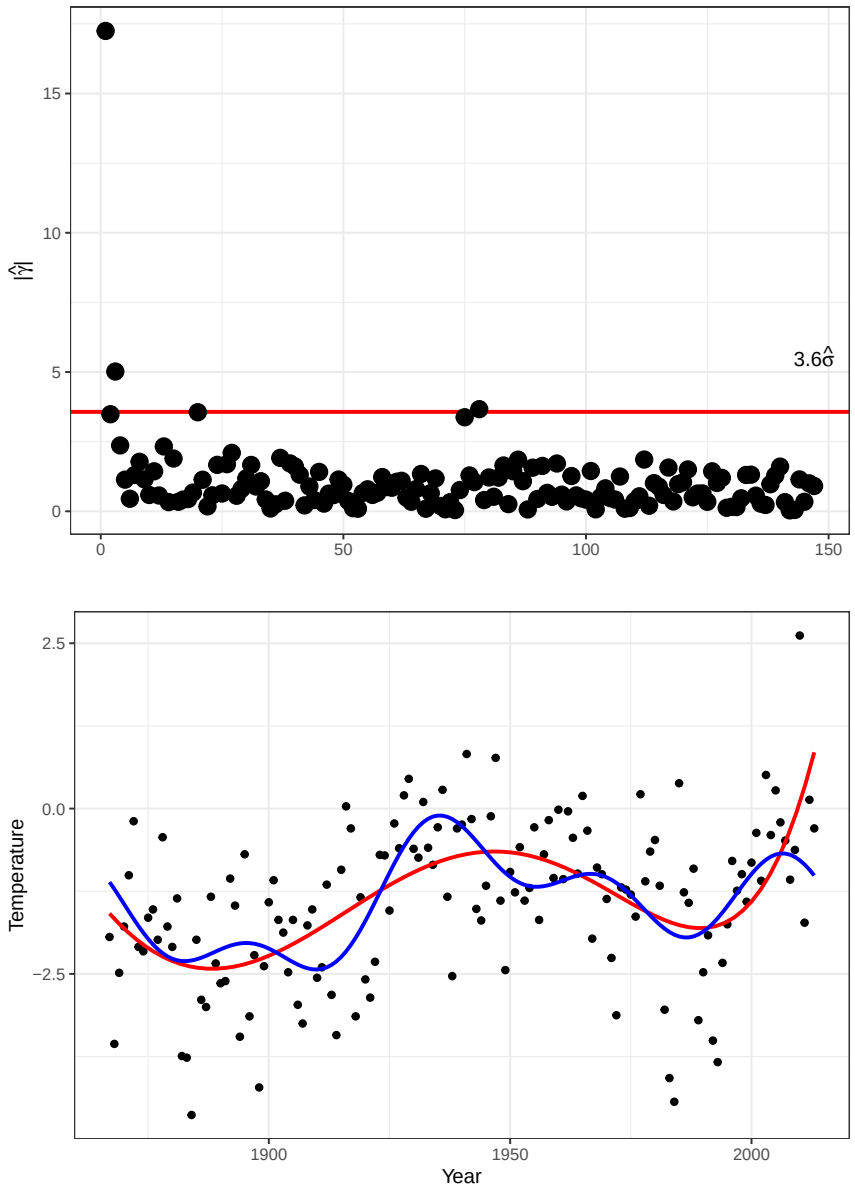


Figure 3.12: Fourier based smoother by thresholding (blue) and polynomial fit of degree 5 (red).

What is the point using the discrete Fourier transform? The point is that the discrete Fourier transform can be computed via the *fast Fourier transform* (FFT), which has an $O(n \log(n))$ time complexity. The FFT works optimally for $n = 2^p$.

```
fft(Nuuk_year$Temperature)[1:4] / sqrt(n)
```

```
## [1] -17.246965+0.000000i  -2.464289+2.387119i   3.548133+0.909923i
## [4]   1.672144+0.741358i
```

```
betahat[1:4]
```

```
## [1] -17.246965+0.000000i  -2.464289+2.387119i   3.548133+0.909923i
## [4]   1.672144+0.741358i
```

3.5 Splines

In the previous section orthogonality of basis functions played an important role for computing basis function expansions efficiently as well as for the statistical assessment of estimated coefficients. This section will deal with bivariate smoothing via basis functions that are not necessarily orthogonal.

Though some of the material of this section will apply to any choice of basis, we restrict attention to splines and consider almost exclusively the widely used B-splines (the "B" is for basis).

3.5.1 Smoothing splines

To motivate splines we briefly consider the following penalized least squares criterion for finding a smooth approximation to bivariate data: minimize

$$L(f) = \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \|f''\|_2^2 \tag{3.1}$$

over all twice differentiable functions f. The first term is the standard squared error, and we can easily find a smooth function interpolating the y-values (if all the x-values are different), which will thus drive the squared error to 0. The squared 2-norm regularizes the minimization problem so that the minimizer finds a balance between interpolation and having a small second derivative (note that $\|f''\|_2 = 0$ if and only if f is an affine function). The tuning parameter λ controls this balance.

It is possible to show that the minimizer of (3.1) is a natural cubic spline with *knots* in the data points x_i. That is, the spline is a C^2-function that equals a third degree polynomial in between the knots. At the knots, the two polynomials that meet fit together up to the second derivative, but they may differ on the third derivative. That the solution is *natural* means that it has zero second and third derivative at and beyond the two boundary knots.

It is not particularly difficult to show that the space of natural cubic splines is a vector space of dimension n if all the x-values are different. It is therefore possible to find a basis of splines, $\varphi_1, \ldots, \varphi_n$, such that the f that minimizes (3.1) is of the form

$$f = \sum_{i=1}^{n} \beta_i \varphi_i.$$

What is remarkable about this is that the basis (and the finite dimensional vector space it spans) doesn't depend upon the y-values. Though the optimization is over an infinite dimensional space, the penalization ensures that the minimizer is always in the same finite dimensional space nomatter what $y_1, \ldots, y_n$ are. Moreover, since (3.1) is a quite natural criterion to minimize to find a smooth function fitting the bivariate data, splines appear as good candidates for producing such smooth fits. On top of that, splines have several computational advantages and are widely used.

If we let $\hat{f}_i = \hat{f}(x_i)$ with $\hat{f}$ the minimizer of (3.1), we have in vector notation that

$$\hat{\mathbf{f}} = \mathbf{\Phi}\hat{\beta}$$

with $\mathbf{\Phi}_{ij} = \varphi_j(x_i)$. The minimizer can be found by observing that

$$L(\mathbf{f}) = (\mathbf{y} - \mathbf{f})^T(\mathbf{y} - \mathbf{f}) + \lambda \|f''\|_2^2 \tag{3.2}$$

$$= (\mathbf{y} - \mathbf{\Phi}\beta)^T(\mathbf{y} - \mathbf{\Phi}\beta) + \lambda \beta^T \mathbf{\Omega} \beta \tag{3.3}$$

where

$$\mathbf{\Omega}_{ij} = \langle \varphi_i'', \varphi_j'' \rangle = \int \varphi_i''(z)\varphi_j''(z)\mathrm{d}z.$$

The matrix $\mathbf{\Omega}$ is positive semidefinite by construction, and we refer to it as the *penalty matrix*. It induces a seminorm on $\mathbb{R}^n$ so that we can express the seminorm, $\|f''\|_2$, of f in terms of the parameters in the basis expansion using φ_i.

This is a standard penalized least squares problem, whose solution is

$$\hat{\beta} = (\mathbf{\Phi}^T\mathbf{\Phi} + \lambda\mathbf{\Omega})^{-1}\mathbf{\Phi}^T\mathbf{y}$$

and with resulting smoother

$$\hat{\mathbf{f}} = \underbrace{\mathbf{\Phi}((\mathbf{\Phi}^T\mathbf{\Phi} + \lambda\mathbf{\Omega})^{-1}\mathbf{\Phi}^T}_{\mathbf{S}_\lambda}\mathbf{y}.$$

This linear smoother with smoothing matrix $\mathbf{S}_\lambda$ based on natural cubic splines gives what is known as a smoothing spline that minimizes (3.1). We will pursue spline based smoothing by minimizing (3.1) but using various B-spline bases that may have more or less than n elements. For the linear algebra, it actually doesn't matter if we use a spline basis or any other basis – as long as $\mathbf{\Phi}_{ij} = \varphi_j(x_i)$ and $\mathbf{\Omega}$ is given in terms of φ_i'' as above.

3.5.2 Splines in R

The splines package in R implements some of the basic functions needed to work with splines. In particular, the `splineDesign()` function that computes evaluations of B-splines and their derivatives.

```
library(splines)
```

```
# Note the specification of repeated boundary knots
knots <- c(0, 0, 0, seq(0, 1, 0.2), 1, 1, 1)
xx <- seq(0, 1, 0.005)
B_splines <- splineDesign(knots, xx)
matplot(xx, B_splines, type = "l", lty = 1)
```

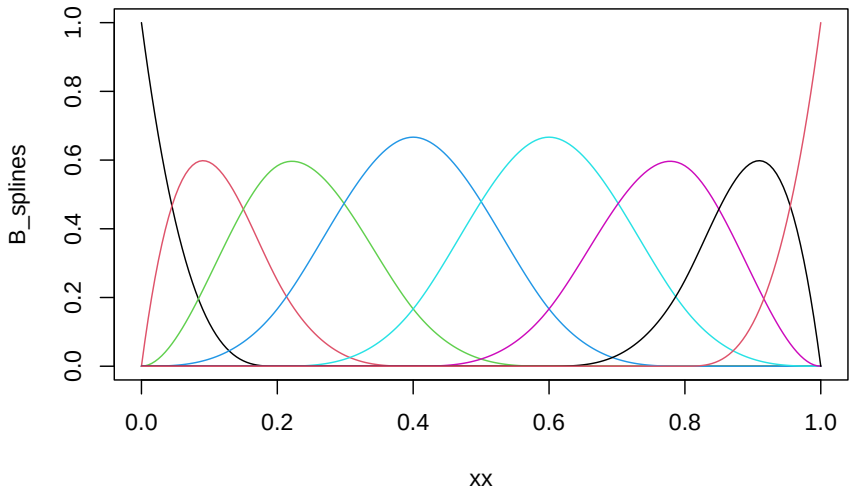


Figure 3.13: B-spline basis as computed by 'splineDesign()'.

The basis shown in Figure 3.13 is an example of a cubic B-spline basis with the 11 inner knots $0, 0.1, \ldots, 0.9, 1$. The repeated boundary knots control how the spline basis behaves close to the

boundaries of the interval. This basis has 13 basis functions, not 11, and spans a larger space than the space of *natural* cubic splines. It is possible to compute a basis based on B-splines for the natural cubic splines using the function `ns`, but for all practical purposes this is not important, and we will work exclusively with the B-spline basis itself.

The computation of the penalty matrix $\mathbf{\Omega}$ constitutes a practical problem, but observing that φ_i'' is an affine function in between knots leads to a simple way of computing $\mathbf{\Omega}_{ij}$. Letting $g_{ij} = \varphi_i''\varphi_j''$ it holds that g_{ij} is quadratic between two consecutive knots a and b, in which case

$$\int_a^b g_{ij}(z)\mathrm{d}z = \frac{b-a}{6}\left(g_{ij}(a) + 4g_{ij}\left(\frac{a+b}{2}\right) + g_{ij}(b)\right).$$

This identity is behind Simpson's rule for numerical integration, and the fact that this is an identity for quadratic polynomials, and not an approximation, means that Simpson's rule applied appropriately leads to exact computation of $\mathbf{\Omega}_{ij}$. All we need is the ability to evaluate φ_i'' at certain points, and `splineDesign()` can be used for that.

```
pen_mat <- function(inner_knots) {
  knots <- sort(c(rep(range(inner_knots), 3), inner_knots))
  d <- diff(inner_knots)  # The vector of knot differences; b - a
  g_ab <- splineDesign(knots, inner_knots, derivs = 2)
  knots_mid <- inner_knots[-length(inner_knots)] + d / 2
  g_ab_mid <- splineDesign(knots, knots_mid, derivs = 2)
  g_a <- g_ab[-nrow(g_ab), ]
  g_b <- g_ab[-1, ]
  (crossprod(d * g_a,  g_a) +
      4 * crossprod(d * g_ab_mid, g_ab_mid) +
      crossprod(d * g_b, g_b)) / 6
}
```

It is laborious to write good tests of `pen_mat()`. We would have to work out a set of example matrices by other means, e.g. by hand. Alternatively, we can compare to a simpler numerical integration technique using Riemann sums.

```
spline_deriv <- splineDesign(
  c(0, 0, 0, 0, 0.5, 1, 1, 1, 1),
  seq(0, 1, 1e-5),
  derivs = 2
)
Omega_numeric <- crossprod(spline_deriv[-1, ]) * 1e-5  # Right Riemann sums
Omega <- pen_mat(c(0, 0.5, 1))
Omega_numeric / Omega
```

```
##           [,1]      [,2]     [,3]     [,4]     [,5]
## [1,] 0.9999700 0.9999673 0.999940 1.000000      NaN
## [2,] 0.9999673 0.9999663 0.999955 1.000000 1.000000
## [3,] 0.9999400 0.9999550 1.000000 1.000045 1.000060
## [4,] 1.0000000 1.0000000 1.000045 1.000034 1.000033
## [5,]       NaN 1.0000000 1.000060 1.000033 1.000030
```

```
range((Omega_numeric - Omega) / (Omega + 0.001))  #  Relative error
```

```
## [1] -5.99967e-05  5.99983e-05
```

And we should also test an example with non-equidistant knots.

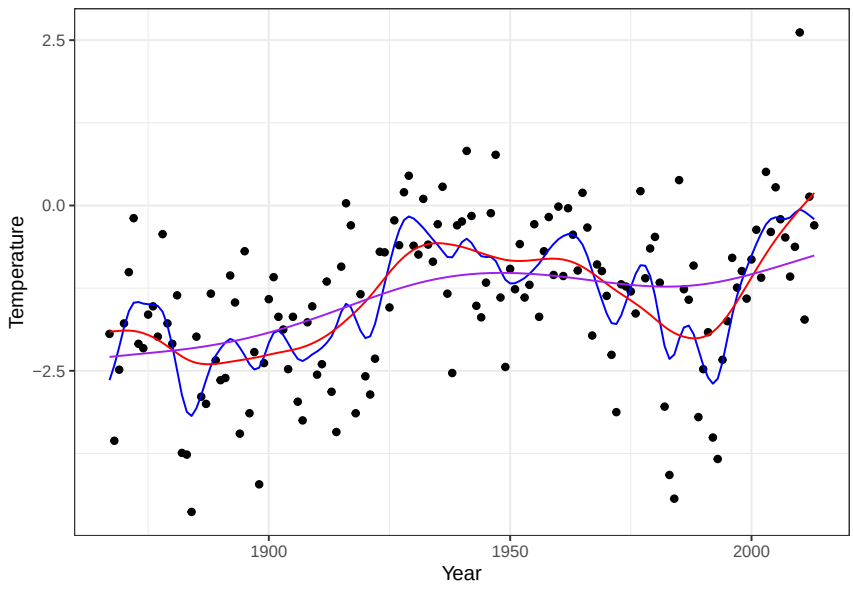
```
spline_deriv <- splineDesign(
  c(0, 0, 0, 0, 0.2, 0.3, 0.5, 0.6, 0.65, 0.7, 1, 1, 1, 1),
  seq(0, 1, 1e-5),
  derivs = 2
)
Omega_numeric <- crossprod(spline_deriv[-1, ]) * 1e-5  # Right Riemann sums
Omega <- pen_mat(c(0, 0.2, 0.3, 0.5, 0.6, 0.65, 0.7, 1))
range((Omega_numeric - Omega) / (Omega + 0.001)) # Relative error
```

```
## [1] -0.0001607084  0.0002545494
```

These examples indicate that `pen_mat()` computes $\mathbf{\Omega}$ correctly, in particular as increasing the Riemann sum precision by lowering the number 10^{-5} will decrease the relative error (not shown). Of course, correctness ultimately depends on `splineDesign()` computing the correct second derivatives, which hasn't been tested here.

We can also test how our implementation of smoothing splines works on data. We do this here by implementing the matrix-algebra directly for computing $\mathbf{S}_\lambda \mathbf{y}$.

```
inner_knots <- Nuuk_year$Year
Phi <- splineDesign(c(rep(range(inner_knots), 3), inner_knots), inner_knots)
Omega <- pen_mat(inner_knots)
smoother <- function(lambda)
  Phi %*% solve(
    crossprod(Phi) + lambda * Omega,
    t(Phi) %*% Nuuk_year$Temperature
  )
p_Nuuk +
  geom_line(aes(y = smoother(10)), color = "blue") +      # Undersmooth
  geom_line(aes(y = smoother(1000)), color = "red") +     # Smooth
  geom_line(aes(y = smoother(100000)), color = "purple")  # Oversmooth
```



Smoothing splines can be computed using the R function `smooth.spline()` from the stats package. It is possible to manually specify the amount of smoothing using one of the arguments `lambda`, `spar` or `df` (the latter being the trace of the smoother matrix). However, due to internal differences from the `splineDesign()` basis above, the `lambda` argument to `smooth.spline()`

does not match the λ parameter above.

If the amount of smoothing is not manually set, `smooth.spline()` chooses λ by *generalized cross validation* (GCV), which minimizes

$$\text{GCV} = \sum_{i=1}^{n} \left(\frac{y_i - \hat{f}_i}{1 - \text{df}/n} \right)^2,$$

where

$$\text{df} = \text{trace}(\mathbf{S}) = \sum_{i=1}^{n} S_{ii}.$$

GCV corresponds to LOOCV with the diagonal entries, S_{ii}, replaced by their average df/n. The main reason for using GCV over LOOCV is that for some smoothers, such as the spline smoother, it is possible to compute the trace df easily without computing $\mathbf{S}$ or even its diagonal elements.

To compare our results to `smooth.spline()` we optimize the GCV criterion. First we implement a function that computes GCV for a fixed value of λ. Here the implementation is relying on computing the smoother matrix, but this is not the most efficient implementation. Section 3.5.3 provides a diagonalization of the smoother matrix jointly in the tuning parameters. This representation allows for efficient computation with splines, and it will become clear why it is not necessary to compute $\mathbf{S}$ or even its diagonal elements. The trace is nevertheless easily computable $\mathbf{S}$.

```
gcv <- function(lambda, y) {
  S <- Phi %*% solve(crossprod(Phi) + lambda * Omega, t(Phi))
  df <- sum(diag(S))  # The trace of the smoother matrix
  sum(((y - S %*% y) / (1 - df / length(y)))^2, na.rm = TRUE)
}
```

Then we apply this function to a grid of λ-values and choose the value of λ that minimizes GCV.

```
lambda <- seq(50, 250, 2)
GCV <- sapply(lambda, gcv, y = Nuuk_year$Temperature)
lambda_opt <- lambda[which.min(GCV)]
qplot(lambda, GCV) + geom_vline(xintercept = lambda_opt, color = "red")
```

Finally, we can visualize the resulting smoothing spline.

```
smooth_opt <- Phi %*% solve(
  crossprod(Phi) + lambda_opt * Omega,
  t(Phi) %*% Nuuk_year$Temperature
)
p_Nuuk + geom_line(aes(y = smooth_opt), color = "blue")
```

The smoothing spline that we found by minimizing GCV can be compared to the smoothing spline that `smooth.spline()` computes by minimizing GCV as well.

```
smooth_splines <- smooth.spline(
  Nuuk_year$Year,
  Nuuk_year$Temperature,
  all.knots = TRUE        # Don't use heuristic
)
range(smooth_splines$y - smooth_opt)
```

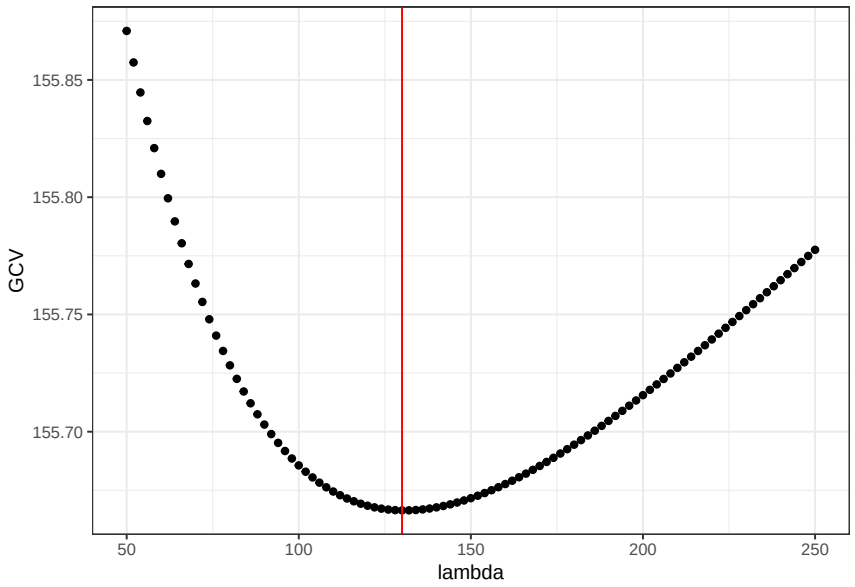


Figure 3.14: The generalized cross-validation criterion for smoothing splines as a function of the tuning parameter λ.

```
## [1] -0.000775662  0.001072587
```

```
p_Nuuk + geom_line(aes(y = smooth_splines$y), color = "blue")
```

The differences between the smoothing spline computed by our implementation and by `smooth.spline()` is hardly detectable visually, and they are at most of the order 10^{-3} as computed above. It is possible to further decrease the differences by finding the optimal value of λ with a higher precision, but we will not pursue this here.

3.5.3 Efficient computation with splines

Using the full B-spline basis with knots in every observation is computationally heavy and from a practical viewpoint unnecessary. Smoothing using B-splines is therefore often done using a knot-selection heuristic that selects much fewer knots than n, in particular if n is large. This is also what `smooth.spline()` does unless `all.knots = TRUE`. The heuristic for selecting the number of knots is a bit complicated, but it is implemented in the function `.nknots.smspl()`, which can be inspected for details. Once the number of knots gets above 200 it grows extremely slowly with n. With the number of knots selected, a common heuristic for selecting their position is to use the quantiles of the distribution of the x-values. That is, with 9 knots, say, the knots are positioned in the deciles (0.1-quantile, 0.2-quantile etc.). This is effectively also what `smooth.spline()` does, and this heuristic places most of the knots where we have most of the data points.

Having implemented a knot-selection heuristic that results in p B-spline basis functions, the matrix Φ will be $n \times p$, typically with $p < n$ and with Φ of full rank p. In this case we derive a way of computing the smoothing spline that is computationally more efficient and numerically more stable than relying on the matrix-algebraic solution above. This is particularly so when we need to compute the smoother for many different λ-s to optimize the smoother. As we will show, we are effectively computing a simultaneous diagonalization of the (symmetric) smoother matrix $\mathbf{S}_\lambda$ for all values of λ.

The matrix Φ has a singular value decomposition

$$\Phi = \mathbf{U}D\mathbf{V}^T$$

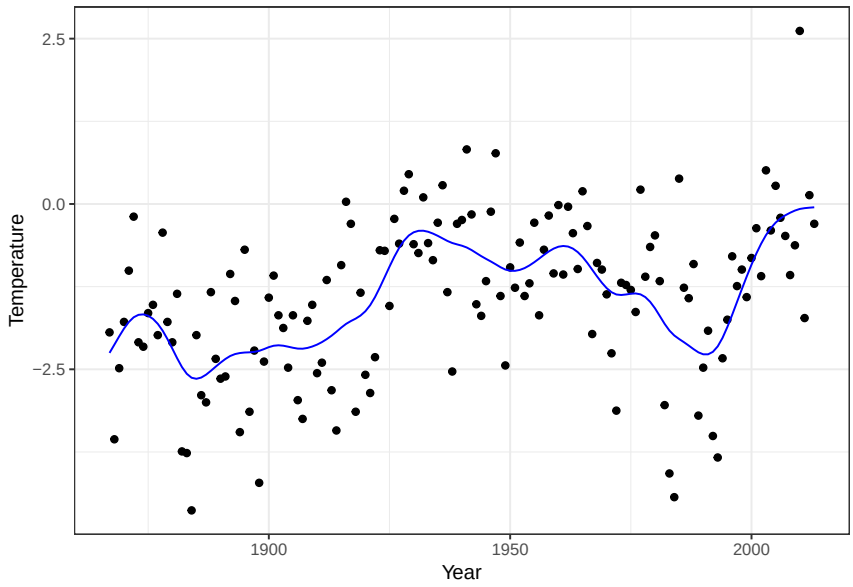


Figure 3.15: The smoothing spline that minimizes GCV over the tuning parameter λ

where D is diagonal with entries $d_1 \geq d_2 \geq \ldots \geq d_p > 0$, $\mathbf{U}$ is $n \times p$, $\mathbf{V}$ is $p \times p$ and both are orthogonal matrices. This means that

$$\mathbf{U}^T\mathbf{U} = \mathbf{V}^T\mathbf{V} = \mathbf{V}\mathbf{V}^T = I$$

is the $p \times p$ dimensional identity matrix. We find that

$$\begin{aligned}
\mathbf{S}_\lambda &= \mathbf{U}D\mathbf{V}^T(\mathbf{V}D^2\mathbf{V}^T + \lambda\mathbf{\Omega})^{-1}\mathbf{V}D\mathbf{U}^T && (3.4)\\
&= \mathbf{U}D(D^2 + \lambda\mathbf{V}^T\mathbf{\Omega}\mathbf{V})^{-1}D\mathbf{U}^T && (3.5)\\
&= \mathbf{U}(I + \lambda D^{-1}\mathbf{V}^T\mathbf{\Omega}\mathbf{V}D^{-1})^{-1}\mathbf{U}^T && (3.6)\\
&= \mathbf{U}(I + \lambda\widetilde{\mathbf{\Omega}})^{-1}\mathbf{U}^T, && (3.7)
\end{aligned}$$

where $\widetilde{\mathbf{\Omega}} = D^{-1}\mathbf{V}^T\mathbf{\Omega}\mathbf{V}D^{-1}$ is a positive semidefinite $p \times p$ matrix. By diagonalization,

$$\widetilde{\mathbf{\Omega}} = \mathbf{W}\Gamma\mathbf{W}^T,$$

where $\mathbf{W}$ is orthogonal and Γ is a diagonal matrix with nonnegative values in the diagonal, and we find that

$$\begin{aligned}
\mathbf{S}_\lambda &= \mathbf{U}\mathbf{W}(I + \lambda\Gamma)^{-1}\mathbf{W}^T\mathbf{U}^T && (3.8)\\
&= \widetilde{\mathbf{U}}(I + \lambda\Gamma)^{-1}\widetilde{\mathbf{U}}^T && (3.9)
\end{aligned}$$

where $\widetilde{\mathbf{U}} = \mathbf{U}\mathbf{W}$ is an orthogonal $n \times p$ matrix.

The interpretation of this representation is as follows.

- First, the coefficients, $\hat{\beta} = \widetilde{\mathbf{U}}^T y$, are computed for expanding y in the basis given by the columns of $\widetilde{\mathbf{U}}$.

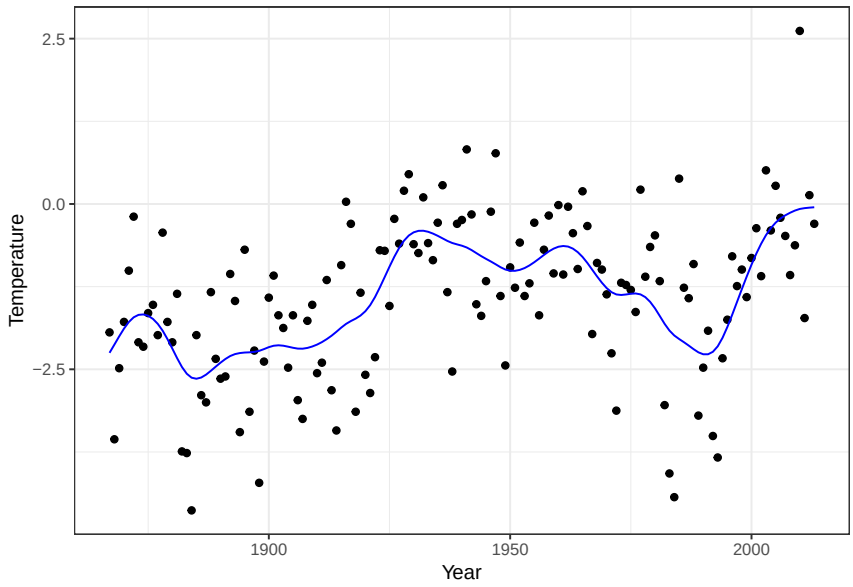


Figure 3.16: The smoothing spline that minimizes GCV as computed by `smooth.spline()`.

- Second, the i-th coefficient is shrunk towards 0,

$$\hat{\beta}_i(\lambda) = \frac{\hat{\beta}_i}{1 + \lambda\gamma_i}.$$

- Third, the smoothed values, $\widetilde{\mathbf{U}}\hat{\beta}(\lambda)$, are computed as an expansion using the shrunken coefficients.

Thus the smoother works by shrinking the coefficients toward zero in the orthonormal basis given by the columns of $\widetilde{\mathbf{U}}$. The coefficients corresponding to the largest eigenvalues γ_i are shrunk relatively more toward zero than those corresponding to the small eigenvalues. The basis formed by the columns of $\widetilde{\mathbf{U}}$ is known as the Demmler-Reinsch basis with reference to Demmler and Reinsch (1975).

We implement the computation of the diagonalization for the Nuuk temperature data using $p = 20$ basis functions (18 inner knots) equidistantly distributed over the range of the years for which we have data.

```
inner_knots <- seq(1867, 2013, length.out = 18)
Phi <- splineDesign(c(rep(range(inner_knots), 3), inner_knots), Nuuk_year$Year)
Omega <- pen_mat(inner_knots)
Phi_svd <- svd(Phi)
Omega_tilde <- t(crossprod(Phi_svd$v, Omega %*% Phi_svd$v)) / Phi_svd$d
Omega_tilde <- t(Omega_tilde) / Phi_svd$d
# It is safer to use the numerical singular value decomposition ('svd()')
# for diagonalizing a positive semidefinite matrix than to use a
# more general numerical diagonalization implementation such as 'eigen()'.
Omega_tilde_svd <- svd(Omega_tilde)
U_tilde <- Phi_svd$u %*% Omega_tilde_svd$u
```

We observe from Figures 3.17 and 3.18 that there are two relatively large eigenvalues corresponding to the two basis functions with erratic behavior close to the boundaries, and there are two eigenvalues that are effectively zero corresponding to the two affine basis functions. In addition, the more oscillating the basis function is, the larger is the corresponding eigenvalue, and the more is the corresponding coefficient shrunk toward zero by the spline smoother.

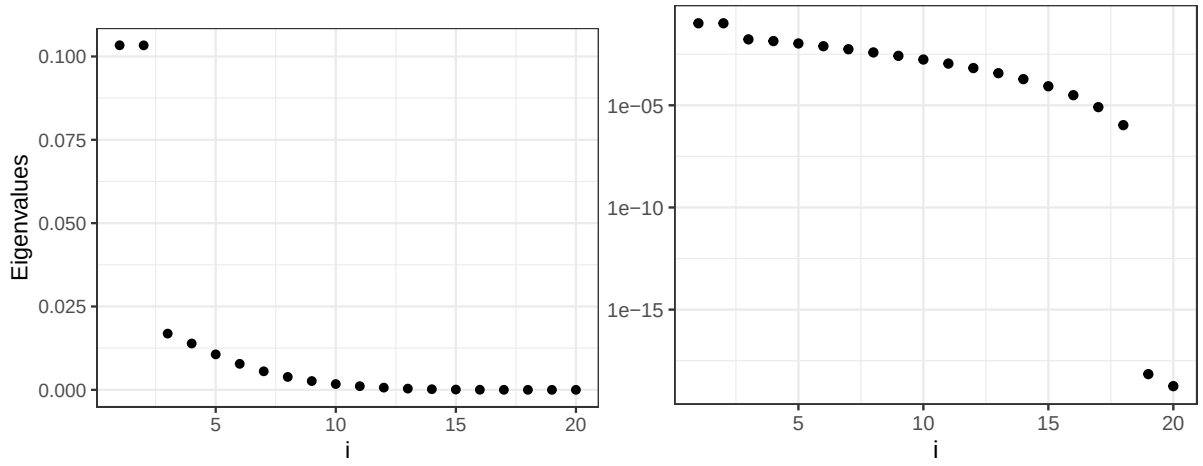


Figure 3.17: The eigenvalues γ_i that determine how much the different basis coefficients in the orthonormal spline expansion are shrunk toward zero. Left plot shows the eigenvalues, and the right plot shows the eigenvalues on a log-scale.

Observe also that

$$\mathrm{df}(\lambda) = \mathrm{trace}(\mathbf{S}_\lambda) = \sum_{i=1}^{p} \frac{1}{1+\lambda\gamma_i},$$

which makes it possible to implement GCV without even computing the diagonal entries of $\mathbf{S}_\lambda$.

3.6 Gaussian processes

Suppose that $X = X_{1:n} \sim \mathcal{N}(\xi_x, \Sigma_x)$ with

$$\mathrm{cov}(X_i, X_j) = K(t_i - t_j)$$

for a kernel function K.

With the *observation equation* $Y_i = X_i + \delta_i$ for $\delta = \delta_{1:n} \sim \mathcal{N}(0, \Omega)$ and $\delta \perp\!\!\!\perp X$ we get

$$(X, Y) \sim \mathcal{N}\left(\begin{pmatrix} \xi_x \\ \xi_x \end{pmatrix}, \begin{pmatrix} \Sigma_x & \Sigma_x \\ \Sigma_x & \Sigma_x + \Omega \end{pmatrix}\right).$$

Hence

$$E(X \mid Y) = \xi_x + \Sigma_x(\Sigma_x + \Omega)^{-1}(Y - \xi_x).$$

Assuming that $\xi_x = 0$ the conditional expectation is a linear smoother with smoother matrix

$$S = \Sigma_x(\Sigma_x + \Omega)^{-1}.$$

This is also true if $\Sigma_x(\Sigma_x + \Omega)^{-1}\xi_x = \xi_x$. If this identity holds approximately, we can argue that for computing $E(X \mid Y)$ we don't need to know ξ_x.

If the observation variance is $\Omega = \sigma^2 I$ then the smoother matrix is

$$\Sigma_x(\Sigma_x + \sigma^2 I)^{-1} = (I + \sigma^2\Sigma_x^{-1})^{-1}.$$

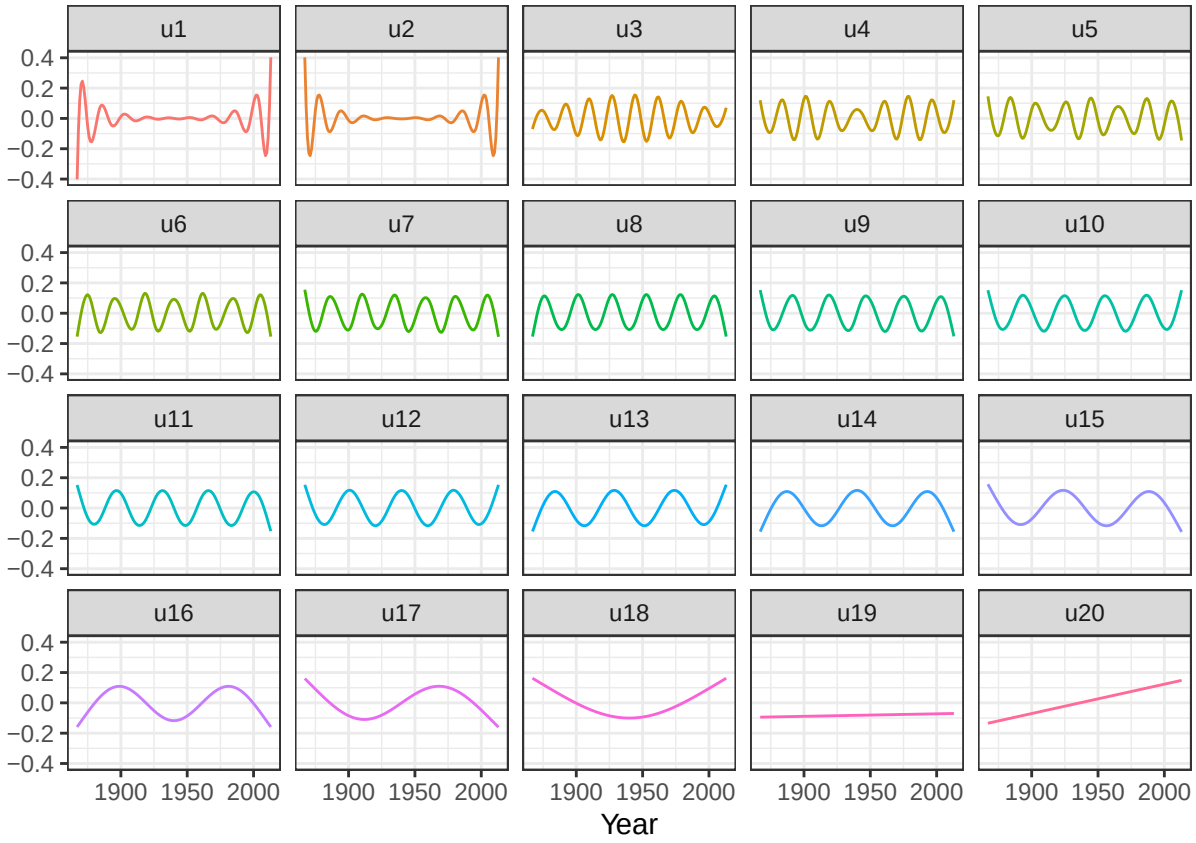


Figure 3.18: The columns of $\widetilde{\mathbf{U}}$ that consitute an orthonormal basis known as the Demmler-Reinsch basis for computing the spline based smoother.

3.7 The Kalman filter

3.7.1 AR(1)-example

Suppose that $|\alpha| < 1$, $X_1 = \epsilon_1/\sqrt{1-\alpha^2}$ and

$$X_i = \alpha X_{i-1} + \epsilon_i$$

for $i = 2, \ldots, n$ with $\epsilon = \epsilon_{1:n} \sim \mathcal{N}(0, \sigma^2 I)$.

We have $\operatorname{cov}(X_i, X_j) = \alpha^{|i-j|}/(1-\alpha^2)$, thus we can find Σ_x and compute

$$E(X_n \mid Y) = ((I + \sigma^2 \Sigma_x^{-1})^{-1} Y)_n$$

3.7.2 The Kalman smoother

From the identity $\epsilon_i = X_i - \alpha X_{i-1}$ it follows that $\epsilon = AX$ where

$$A = \begin{pmatrix} \sqrt{1-\alpha^2} & 0 & 0 & \ldots & 0 & 0 \\ -\alpha & 1 & 0 & \ldots & 0 & 0 \\ 0 & -\alpha & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & 0 & \ldots & -\alpha & 1 \end{pmatrix},$$

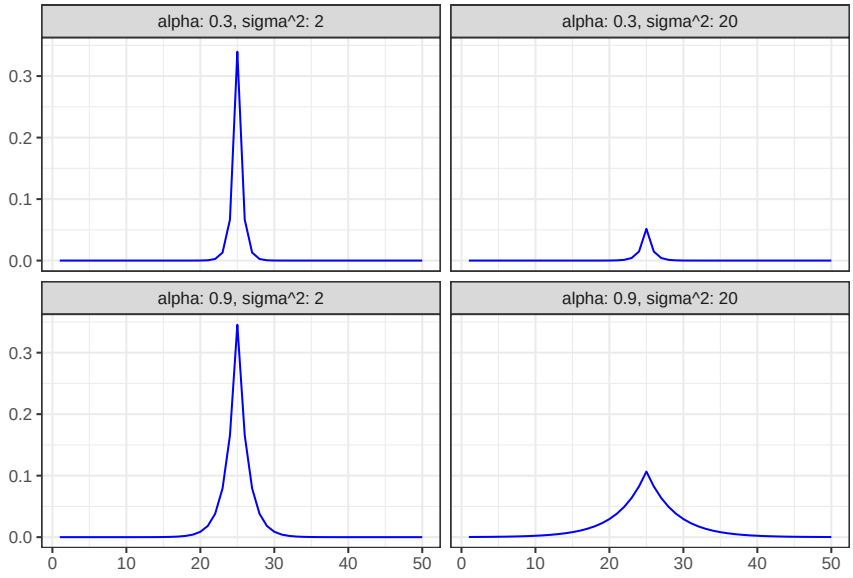


Figure 3.19: Smoothers

This gives $I = V(\epsilon) = A\Sigma_x A^T$, hence

$$\Sigma_x^{-1} = (A^{-1}(A^T)^{-1})^{-1} = A^T A.$$

We have shown that

$$\Sigma_x^{-1} = \begin{pmatrix} 1 & -\alpha & 0 & \dots & 0 & 0 \\ -\alpha & 1+\alpha^2 & -\alpha & \dots & 0 & 0 \\ 0 & -\alpha & 1+\alpha^2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1+\alpha^2 & -\alpha \\ 0 & 0 & 0 & \dots & -\alpha & 1 \end{pmatrix}.$$

Hence

$$I + \sigma^2\Sigma_x^{-1} = \begin{pmatrix} \gamma_0 & \rho & 0 & \dots & 0 & 0 \\ \rho & \gamma & \rho & \dots & 0 & 0 \\ 0 & \rho & \gamma & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \gamma & \rho \\ 0 & 0 & 0 & \dots & \rho & \gamma_0 \end{pmatrix}$$

with $\gamma_0 = 1 + \sigma^2$, $\gamma = 1 + \sigma^2(1 + \alpha^2)$ and $\rho = -\sigma^2\alpha$ is a *tridiagonal matrix.*

The equation

$$\begin{pmatrix} \gamma_0 & \rho & 0 & \dots & 0 & 0 \\ \rho & \gamma & \rho & \dots & 0 & 0 \\ 0 & \rho & \gamma & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \gamma & \rho \\ 0 & 0 & 0 & \dots & \rho & \gamma_0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix}$$

can be solved by a forward and backward sweep.

Forward sweep:

- Set $\rho'_1 = \rho/\gamma_0$ and $y'_1 = y_1/\gamma_0$,
- then recursively

$$\rho'_i = \frac{\rho}{\gamma - \rho\rho'_{i-1}} \quad \text{and} \quad y'_i = \frac{y_i - \rho y'_{i-1}}{\gamma - \rho\rho'_{i-1}}$$

  for $i = 2, \ldots, n-1$
- and finally

$$y'_n = \frac{y_n - \rho y'_{n-1}}{\gamma_0 - \rho\rho'_{n-1}}.$$

By the forward sweep the equation is transformed to

$$\begin{pmatrix} 1 & \rho'_1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \rho'_2 & \ldots & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & \rho'_{n-1} \\ 0 & 0 & 0 & \ldots & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} y'_1 \\ y'_2 \\ y'_3 \\ \vdots \\ y'_{n-1} \\ y'_n \end{pmatrix},$$

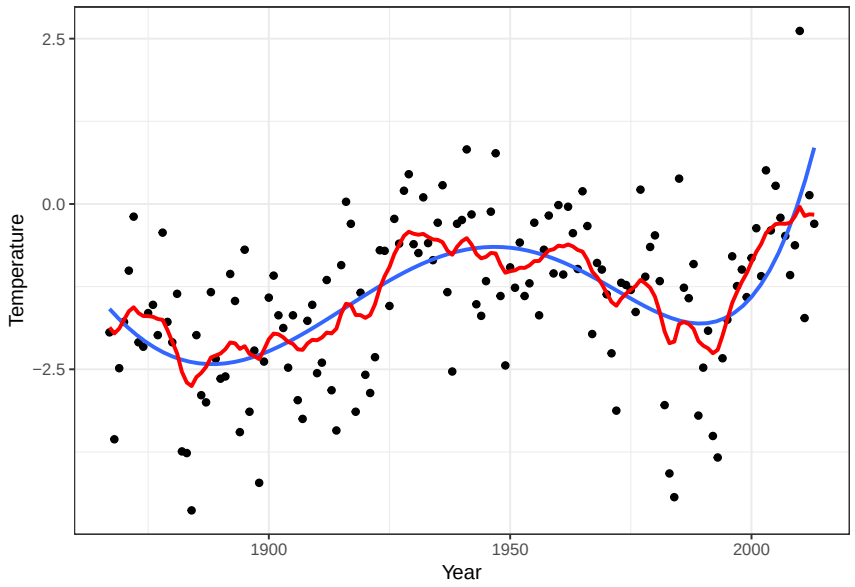
which is then solved by backsubstitution from below; $x_n = y'_n$ and

$$x_i = y'_i - \rho'_i x_{i+1}, \quad i = n-1, \ldots, 1.$$

3.7.3 Implementation

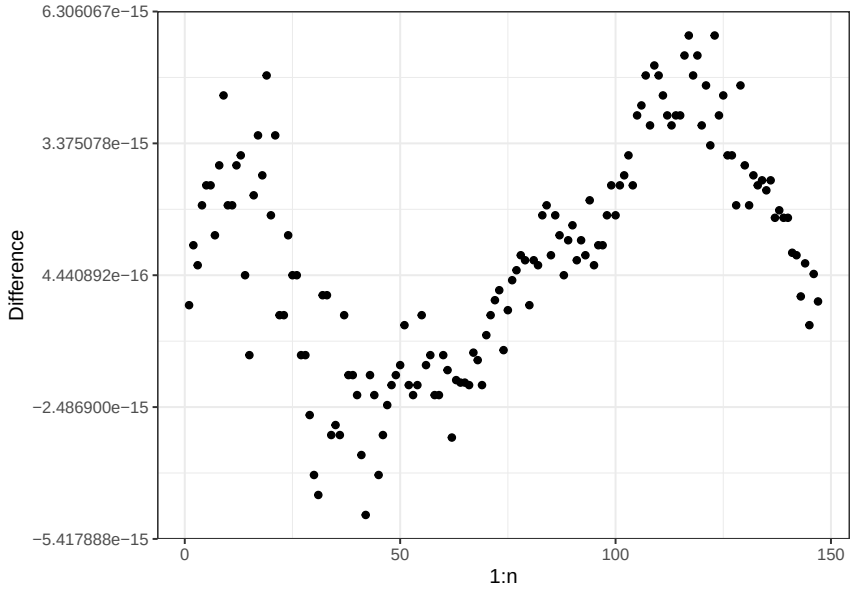
```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector KalmanSmooth(NumericVector y, double alpha, double sigmasq) {
  double tmp, gamma0 = 1 + sigmasq, rho = - sigmasq * alpha;
  double gamma = 1 + sigmasq * (1 + alpha * alpha);
  int n = y.size();
  NumericVector x(n), rhop(n - 1);
  rhop[0] = rho / gamma0;
  x[0] = y[0] / gamma0;
  for(int i = 1; i < n - 1; ++i) { /* Forward sweep */
    tmp = (gamma - rho * rhop[i - 1]);
    rhop[i] = rho / tmp;
    x[i] = (y[i] - rho * x[i - 1]) / tmp;
  }
  x[n - 1] = (y[n - 1] - rho * x[n - 2]) / (gamma0 - rho * rhop[n - 2]);
  for(int i = n - 2; i >= 0; --i) { /* Backsubstitution */
    x[i] = x[i] - rhop[i] * x[i + 1];
  }
  return x;
}
```

Result, $\alpha = 0.95$, $\sigma^2 = 10$



Comparing results

```
Sigma <- outer(1:n, 1:n,
               function(i, j) alpha^(abs(i - j))) / (1 - alpha^2)
Smooth <- Sigma %*% solve(Sigma + sigmasq * diag(n))
qplot(1:n, Smooth %*% Nuuk_year$Temperature - ySmooth) +
  ylab("Difference")
```



Note that the forward sweep computes $x_n = E(X_n \mid Y)$, and from this, the backsubstitution solves the smoothing problem of computing $E(X \mid Y)$.

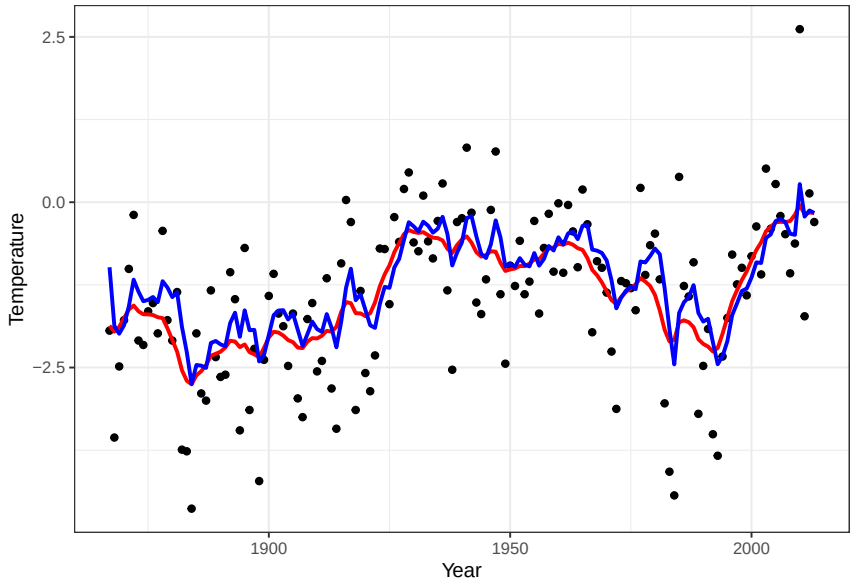
The Gaussian process used here (the AR(1)-process) is not very smooth and nor is the smoothing of the data. This is related to the kernel function $K(s) = \alpha^{|s|}$ being non-differentiable in 0.

Many smoothers are equivalent to a Gaussian process smoother with an appropriate choice of kernel. Not all have a simple inverse covariance matrix and a Kalman filter algorithm.

3.7.4 The Kalman filter

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector KalmanFilt(NumericVector y, double alpha, double sigmasq) {
  double tmp, gamma0 = 1 + sigmasq, rho = - sigmasq * alpha, yp;
  double gamma = 1 + sigmasq * (1 + alpha * alpha);
  int n = y.size();
  NumericVector x(n), rhop(n);
  rhop[0] = rho / gamma0;
  yp = y[0] / gamma0;
  x[0] = y[0] / (1 + sigmasq * (1 - alpha * alpha));
  for(int i = 1; i < n; ++i) {
    tmp = (gamma - rho * rhop[i - 1]);
    rhop[i] = rho / tmp;
    /* Note differences when compared to smoother */
    x[i] = (y[i] - rho * yp) / (gamma0 - rho * rhop[i - 1]);
    yp = (y[i] - rho * yp) / tmp;
  }
  return x;
}
```

Result, $\alpha = 0.95$, $\sigma^2 = 10$



3.8 Exercises

Nearest neighbors

Kernel estimators

Exercise 3.1. Consider a bivariate data set $(x_1, y_1), \ldots, (x_n, y_n)$ and let K be a probability density with mean 0. Then

$$\hat{f}(x,y) = \frac{1}{nh^2} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) K\left(\frac{y - y_i}{h}\right)$$

is a bivariate kernel density estimator of the joint density of x and y. Show that the kernel density estimator

$$\hat{f}_1(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

is also the marginal distribution of x under $\hat{f}$, and that the Nadaraya-Watson kernel smoother is the conditional expectation of y given x under $\hat{f}$.

Exercise 3.2. Suppose that K is a symmetric kernel and the x-s are equidistant. Implement a function that computes the smoother matrix using the `toeplitz` function and $O(n)$ kernel evaluations where n is the number of data points. Implement also a function that computes the diagonal elements of the smoother matrix directly with run time $O(n)$. *Hint: find inspiration in the implementation of the running mean*.

Part II: Monte Carlo Methods

Chapter 4

Univariate random variables

This chapter will deal with algorithms for simulating observations from a distribution on $\mathbb{R}$ or any subset thereof. There can be several purposes of doing so, for instance:

- We want to investigate properties of the distribution.
- We want to simulate independent realizations of univariate random variables to investigate the distribution of a transformation.
- We want to use Monte Carlo integration to compute numerically an integral (which could be a probability).

In this chapter the focus is on the simulation of a single random variable or an i.i.d. sequence of random variables primarily via various transformations of pseudorandom numbers. The pseudo random numbers themselves being approximate simulations of i.i.d. random variables uniformly distributed on $(0,1)$.

4.1 Pseudorandom number generators

Most simulation algorithms are based on algorithms for generating *pseudorandom* uniformly distributed variables in $(0,1)$. They arise from deterministic integer sequences initiated by a *seed*. A classical example of a pseudorandom integer generator is the linear congruential generator. A sequence of numbers from this generator is computed iteratively by

$$x_{n+1} = (ax_n + c) \bmod \mathrm{m}$$

for integer parameters a, c and m. The seed x_1 is a number between 0 and $m-1$, and the resulting sequence is in the set $\{0, \ldots, m-1\}$. The ANSI C standard specifies the choices $m = 2^{31}$, $a = 1,103,515,245$ and $c = 12,345$. The generator is simple to understand and implement but has been superseded by much better generators.

Pseudorandom number generators are generally defined in terms of a finite state space $\mathcal{Z}$ and a one-to-one map $f : \mathcal{Z} \to \mathcal{Z}$. The generator produces a sequence in $\mathcal{Z}$ iteratively from the seed $\mathbf{z}_1 \in \mathcal{Z}$ by

$$\mathbf{z}_n = f(\mathbf{z}_{n-1}).$$

Pseudorandom integers are typically obtained as

$$x_n = h(\mathbf{z}_n)$$

for a transformation $h : \mathcal{Z} \mapsto \mathbb{Z}$. If the image of h is in the set $\{0, 1, \ldots, 2^w - 1\}$ of w-bit integers, pseudorandom numbers in $[0,1)$ are typically obtained as

$$x_n = 2^{-w} h(\mathbf{z}_n).$$

In R, the default pseudorandom number generator is the 32-bit *Mersenne Twister*, which generates integers in the range

$$\{0, 1, \ldots, 2^{32} - 1\}.$$

The state space is

$$\mathcal{Z} = \{0, 1, \ldots, 2^{32} - 1\}^{624},$$

that is, a state is a 624 dimensional vector of 32-bit integers. The function f is of the form

$$f(\mathbf{z}) = (z_2, z_3, \ldots, z_{623}, f_{624}(z_1, z_2, z_{m+1})),$$

for $1 \leq m < 624$, and h is a function of z_{624} only. The standard choice $m = 397$ is used in the R implementation. The function f_{624} is a bit complicated, it includes what is known as the *twist transformation*, and it requires additional parameters. The period of the generator is the astronomical number

$$2^{32 \times 624 - 31} - 1 = 2^{19937} - 1,$$

which is a Mersenne prime. Moreover, all combinations of consecutive integers up to dimension 623 occur equally often in a period, and empirical tests of the generator demonstrate that it has good statistical properties, though it is known to fail some tests.

In R you can set the seed using the function `set.seed` that takes an integer argument and produces an element in the state space. The argument given to `set.seed` is not the actual seed, and `set.seed` computes a valid seed for any pseudorandom number generator that R is using, whether it is the Mersenne Twister or not. Thus the use of `set.seed` is the safe and recommended way of setting a seed.

The actual seed (together with some additional information) can be accessed via the vector `.Random.seed`. Its first entry, `.Random.seed[1]`, encodes the pseudorandom number generator used as well as the generator for Gaussian variables and discrete uniform variables. This information is decoded by `RNGkind()`.

```
RNGkind()
```

```
## [1] "Mersenne-Twister" "Inversion"        "Rejection"
```

For the Mersenne Twister, `.Random.seed[3:626]` contains the vector in the state space, while `.Random.seed[2]` contains the "current position" in the state vector. The implementation needs a position variable because it does 624 updates of the state vector at a time and then runs through those values sequentially before the next update. This is equivalent to but more efficient than e.g. implementing the position shifts explicitly as in the definition of f above.

```
set.seed(27112015)            ## Computes a new seed from an integer
oldseed <- .Random.seed[-1]   ## The actual seed
.Random.seed[1]               ## Encoding of generators used, will stay fixed
```

```
## [1] 10403
```

```
.Random.seed[2]               ## Start position after the seed has been set is 624
```

```
## [1] 624
```

```
tmp <- runif(1)
tmp
```

```
## [1] 0.7793288
```

Every time a random number is generated, e.g. by `runif` above, the same underlying sequence of pseudorandom numbers is used, and the state vector stored in `.Random.seed` is updated accordingly.

```
head(oldseed, 5)
```

```
## [1]         624 -1660633125 -1167670944  1031453153   815285806
```

```
head(.Random.seed[-1], 5)     ## The state vector and position has been updated
```

```
## [1]           1  -696993996 -1035426662  -378189083  -745352065
```

```
c(tmp, runif(1))
```

```
## [1] 0.7793288 0.5613179
```

```
head(.Random.seed[-1], 5)     ## The state vector has not changed, only the position
```

```
## [1]           2  -696993996 -1035426662  -378189083  -745352065
```

Resetting the seed will restart the pseudorandom number generator with the same seed and result in the same sequence of random numbers.

```
set.seed(27112015)
head(.Random.seed[-1], 5)
```

```
## [1]         624 -1660633125 -1167670944  1031453153   815285806
```

```
head(oldseed, 5)              ## Same as current .Random.seed
```

```
## [1]         624 -1660633125 -1167670944  1031453153   815285806
```

```
runif(1)                      ## Same as tmp
```

```
## [1] 0.7793288
```

Note that when using any of the standard R generators, any value of 0 or 1 returned by the underlying pseudorandom uniform generator is adjusted to be in $(0, 1)$. Thus uniform random variables are guaranteed to be in $(0, 1)$.

Some of the random number generators implemented in R use more than one pseudorandom number per variable. This is, for instance, the case when we simulate Gamma distributed random variables.

```
set.seed(27112015)
rgamma(1, 1)                  ## A single Gamma distributed random number
```

```
## [1] 1.192619
```

```
head(.Random.seed[-1], 5)     ## Position changed to 2
```

```
## [1]           2  -696993996 -1035426662  -378189083  -745352065
```

```
rgamma(1, 1)                  ## A single Gamma distributed random number
```

```
## [1] 0.2794622
```

```
head(.Random.seed[-1], 5)     ## Position changed to 5
```

```
## [1]           5  -696993996 -1035426662  -378189083  -745352065
```

In the example above, the first Gamma variable required two pseudorandom numbers, while the second required three pseudorandom numbers. The detailed explanation is given in Section 4.3, where it is shown how to generate random variables from the Gamma distribution via rejection sampling. This requires as a minimum two pseudorandom numbers for every Gamma variable generated.

4.1.1 Implementing a pseudorandom number generator

The development of high quality pseudorandom number generators is a research field in itself. This is particularly true if one needs theoretical guarantees for randomized algorithms or cryptographically secure generators. For scientific computations and simulations correct statistical properties, reproducibility and speed are more important than cryptographic security, but even so, it is not trivial to invent a good generator, and the field is still developing. For a generator to be seriously considered, its mathematical properties should be well understood, and it should pass (most) tests in standardized test suites such as TestU01, see L'Ecuyer and Simard (2007).

R provides a couple of alternatives to the Mersenne Twister, see `?RNG`, but there is no compelling reason to switch to any of those for ordinary use. They are mostly available for historical reasons. One exception is the L'Ecuyer-CMRG generator, which is useful when independent pseudorandom sequences are needed for parallel computations.

Though the Mersenne Twister is a widely used pseudorandom number generator, it has well known shortcomings. There are high quality alternatives that are simpler and faster, such as the family of shift-register generators and their variations, but they are not currently available from the base R package.

Shift-register generators are based on linear transformations of the bit representation of integers. Three particular transformations are typically composed; the Lshift and Rshift operators and the bitwise xor operator. Let $z = [z_{31}, z_{30}, \ldots, z_0]$ with $z_i \in \{0, 1\}$ denote the bit representation of a 32-bit (unsigned) integer z (ordered from most significant bit to least significant bit). That is,

$$z = z_{31}2^{31} + z_{30}2^{30} + \ldots + z_2 2^2 + z_1 2^1 + z_0.$$

Then the left shift operator is defined as

$$\text{Lshift}(z) = [z_{30}, z_{29}, \ldots, z_0, 0],$$

and the right shift operator is defined as

$$\text{Rshift}(z) = [0, z_{31}, z_{30}, \ldots, z_1].$$

The bitwise xor operator is defined as

$$\text{xor}(z, z') = [\text{xor}(z_{31}, z'_{31}), \text{xor}(z_{30}, z'_{30}), \ldots, \text{xor}(z_0, z'_0)]$$

where $\text{xor}(0,0) = \text{xor}(1,1) = 0$ and $\text{xor}(1,0) = \text{xor}(0,1) = 1$. Thus a transformation could be of the form

$$\text{xor}(z, \text{Rshift}^2(z)) = [\text{xor}(z_{31}, 0), \text{xor}(z_{30}, 0), \text{xor}(z_{29}, z_{31}), \ldots, \text{xor}(z_0, z_2)].$$

One example of a shift-register based generator is Marsaglia's xorwow algorithm, Marsaglia (2003). In addition to the shift and xor operations, the output of this generator is perturbed by a sequence of integers with period 2^{32}. The state space of the generator is

$$\{0, 1, \ldots, 2^{32} - 1\}^5$$

with

$$f(\mathbf{z}) = (z_1 + 362437 \pmod{2^{32}}, f_1(z_5, z_2), z_2, z_3, z_4),$$

and

$$h(\mathbf{z}) = 2^{-32}(z_1 + z_2).$$

The number 362437 is Marsaglia's choice for generating what he calls a Weyl sequence, but any odd number will do. The function f_1 is given as

$$f_1(z, z') = \text{xor}(\text{xor}(z, \text{Rshift}^2(z)), \text{xor}(z', \text{xor}(\text{Lshift}^4(z'), \text{Lshift}(\text{xor}(z, \text{Rshift}^2(z)))))).$$

This may look intimidating, but all the operations are very elementary. Take the number $z = 123456$, say, then the intermediate value $\bar{z} = \text{xor}(z, \text{Rshift}^2(z))$ is computed as follows:

z	00000000	00000001	11100010	01000000
$\text{Rshift}^2(z)$	00000000	00000000	01111000	10010000
xor	00000000	00000001	10011010	11010000

And if $z' = 87654321$ the value of $f_1(z, z')$ is computed like this:

$\text{Lshift}^4(z')$	01010011	10010111	11111011	00010000
$\text{Lshift}(\bar{z})$	00000000	00000011	00110101	10100000
xor	01010011	10010100	11001110	10110000
z'	00000101	00111001	01111111	10110001
xor	01010110	10101101	10110001	00000001
$\bar{z}$	00000000	00000001	10011010	11010000
xor	01010110	10101100	00101011	11010001

Converted back to a 32-bit integer, the result is $f_1(z, z') = 1454123985$. The shift and xor operations are tedious to do by hand but extremely fast on modern computer architectures, and shift-register based generators are some of the fastest generators with good statistical properties.

To make R use the xorwow generator we need to implement it as a user supplied generator. This requires writing the C code that implements the generator, compiling the code into a shared object file, loading it into R with the `dyn.load` function, and finally calling `RNGkind("user")` to make R use this pseudorandom number generator. See `?Random.user` for some details and an example.

Using the Rcpp package, and `sourceCpp`, in particular, is usually much preferred over manual compiling and loading. However, in this case we need to make functions available to the internals of R rather than exporting functions to be callable from the R console. That is, nothing needs to be exported from C/C++. If nothing is exported, `sourceCpp` will actually not load the shared object file, so we need to trick `sourceCpp` to do so anyway. In the implementation below we achieve this by simply exporting a direct interface to the xorwow generator.

```
#include <Rcpp.h>
#include <R_ext/Random.h>

/* The Random.h header file contains the function declarations for the functions
 that R rely on internally for a user defined generator, and it also defines
 the type Int32 as an unsigned int. */
```

```
static Int32 z[5];     // The state vector
static double res;
static int nseed = 5;  // Length of the state vector

// Implementation of xorwow from Marsaglia's "Xorshift RNGs"
// modified so as to return a double in [0, 1). The '>>' and '<<'
// operators in C are bitwise right and left shift operators, and
// the caret, '^', is the xor operator.

double * user_unif_rand()
{
  Int32 t = z[4];
  Int32 s = z[1];
  z[0] += 362437;
  z[4] = z[3];
  z[3] = z[2];
  z[2] = s;
  // Right shift t by 2, then bitwise xor between t and its shift
  t ^= t >> 2;
  // Left shift t by 1 and s by 4, xor them, xor with s and xor with t
  t ^= s ^ (s << 4) ^ (t << 1);
  z[1] = t;
  res = (z[0] + t) * 2.32830643653869e-10;
  return &res;
}

// A seed initializer using Marsaglia's congruential PRNG

void  user_unif_init(Int32 seed_in) {
  z[0] = seed_in;
  z[1] = 69069 * z[0] + 1;
  z[2] = 69069 * z[1] + 1;
  z[3] = 69069 * z[2] + 1;
  z[4] = 69069 * z[3] + 1;
}

// Two functions to make '.Random.seed' in R reflect the state vector

int * user_unif_nseed() { return &nseed; }
int * user_unif_seedloc() { return (int *) &z; }

// Wrapper to make 'user_unif_rand' callable from R
double xor_runif() {
  return *user_unif_rand();
}

// This module will export two functions to be directly available from R.
// Note: if nothing is exported, `sourceCpp` will not load the shared
// object file generated by the compilation of the code, and
// 'user_unif_rand' will not become available to the internals of R.
RCPP_MODULE(xorwow) {
```

```
  Rcpp::function("xor_set.seed", &user_unif_init, "Seeds Marsaglia's xorwow");
  Rcpp::function("xor_runif", &xor_runif, "A uniform from Marsaglia's xorwow");
}
```

We first test the direct interface to the xorwow algorithm.

```
xor_set.seed(3573076633)
xor_runif()
```

```
## [1] 0.9090892
```

Then we set R's pseudorandom number generator to be our user supplied generator.

```
default_prng <- RNGkind("user")
```

All R's standard random number generators will after the call `RNGkind("user")` rely on the user provided generator, in this case the xorwow generator. Note that R does an "initial scrambling" of the argument given to `set.seed` before it is passed on to our user defined initializer. This scrambling turns 24102019 used below into 3573076633 used above.

```
set.seed(24102019)
.Random.seed[-1]  ## The state vector as seeded
```

```
## [1] -721890663    9136518 -310030769 1191753796  194708085
```

```
runif(1)          ## As above since same unscrambled seed is used
```

```
## [1] 0.9090892
```

```
.Random.seed[-1]  ## The state vector after one update
```

```
## [1] -721528226  331069150    9136518 -310030769 1191753796
```

The code above shows the state vector of the xorwow algorithm when seeded by the `user_unif_init` function, and it also shows the update to the state vector after a single iteration of the xorwow algorithm.

Though the xorwow algorithm is fast and simple, a benchmark study (not shown) reveals that using xorwow instead of the Mersenne Twister doesn't impact the run time in a notable way when using e.g. `runif`. The generator is simply not the bottleneck. As the implementation of xorwow above is experimental and has not been thoroughly tested, we will not rely on it and quickly reset the random number generator to its default value.

```
## Resetting the generator to the default
RNGkind(default_prng[1])
```

4.1.2 Pseudorandom number packages

To benefit from the recent developments in pseudorandom number generators we can turn to R packages such as the dqrng package. It implements pcg64 from the PCG family of generators as well as Xoroshiro128+ and Xoshiro256+ that are shift-register algorithms. Xoroshiro128+ is the default and other generators can be chosen using `dqRNGkind`. The usage of generators from dqrng is similar to the usage of base R generators.

```
library(dqrng)
dqset.seed(24102019)
dqrunif(1)
```

```
## [1] 0.6172152
```

Using the generators from dqrng does not interfere with the base R generators as the state vectors are completely separated.

In addition to uniform pseudorandom variables generated by `dqrunif` the dqrng package can generate exponential (`dqrexp`) and Gaussian (`dqrnorm`) random variables as well as uniform discrete distributions (`dqsample` and `dqsample.int`). All based on the fast pseudorandom integer generators that the package includes. In addition, the package has a C++ interface that makes it possible to use its generators in compiled code as well.

```
microbenchmark(
  runif(1e6),
  dqrunif(1e6)
)
```

```
## Unit: milliseconds
##             expr  min   lq mean median   uq max neval cld
##    runif(1e+06) 25.4 28.6 32.8   29.6 31.9  79   100   b
##  dqrunif(1e+06)  3.6  4.1  6.3    6.6  7.2  11   100  a
```

As the benchmark above shows, `dqrunif` is about six times faster than `runif` when generating one million variables. The other generators provided by the dqrng package show similar improvements over the base R generators.

4.2 Transformation techniques

If $T : \mathcal{Z} \to \mathbb{R}$ is a map and $Z \in \mathcal{Z}$ is a random variable we can simulate, then we can simulate $X = T(Z)$.

Theorem 4.1. *If $F^{\leftarrow} : (0,1) \mapsto \mathbb{R}$ is the (generalized) inverse of a distribution function and U is uniformly distributed on $(0,1)$ then the distribution of*

$$F^{\leftarrow}(U)$$

has distribution function F.

The proof of Theorem 4.1 can be found in many textbooks and will be skipped. It is easiest to use this theorem if we have an analytic formula for the inverse distribution function. But even in cases where we don't it might be useful for simulation anyway if we have a very accurate approximation that is fast to evaluate.

The call `RNGkind()` in the previous section revealed that the default in R for generating samples from $\mathcal{N}(0,1)$ is inversion. That is, Theorem 4.1 is used to transform uniform random variables with the inverse distribution function Φ^{-1}. This function is, however, non-standard, and R implements a technical approximation of Φ^{-1} via rational functions.

4.2.1 Sampling from a t-distribution

Let $Z = (Y, W) \in \mathbb{R} \times (0, \infty)$ with $Z \sim \mathcal{N}(0,1)$ and $W \sim \chi^2_k$ independent.

Define $T : \mathbb{R} \times (0, \infty) \to \mathbb{R}$ by

$$T(z, w) = \frac{z}{\sqrt{w/k}},$$

then

$$X = T(Z, W) = \frac{Z}{\sqrt{W/k}} \sim t_k.$$

This is how R simulates from a t-distribution with W generated from a gamma distribution with shape parameter $k/2$ and scale parameter 2.

4.3 Rejection sampling

This section deals with a general algorithm for simulating variables from a distribution with density f. We call f the target density and the corresponding distribution is called the target distribution. The idea is to simulate *proposals* from a different distribution with density g (the proposal distribution) and then according to a criterion decide to accept or reject the proposals. It is assumed throughout that the proposal density g is a density fulfilling that

$$g(x) = 0 \Rightarrow f(x) = 0. \tag{4.1}$$

Let $Y_1, Y_2, \ldots$ be i.i.d. with density g on $\mathbb{R}$ and $U_1, U_2, \ldots$ be i.i.d. uniformly distributed on $(0, 1)$ and independent of the Y_i-s. Define

$$T(\mathbf{Y}, \mathbf{U}) = Y_\sigma$$

with

$$\sigma = \inf\{n \geq 1 \mid U_n \leq \alpha f(Y_n)/g(Y_n)\},$$

for $\alpha \in (0, 1]$ and f a density. Rejection sampling then consists of simulating independent pairs (Y_n, U_n) as long as we *reject* the proposals Y_n sampled from g, that is, as long as

$$U_n > \alpha f(Y_n)/g(Y_n).$$

The first time we *accept* a proposal is σ, and then we stop the sampling and return the proposal Y_σ. The result is, indeed, a sample from the distribution with density f as the following theorem states.

Theorem 4.2. *If $\alpha f(y) \leq g(y)$ for all $y \in \mathbb{R}$ and $\alpha > 0$ then the distribution of Y_σ has density f.*

Proof. Note that g automatically fulfills (4.1). The formal proof decomposes the event $(Y_\sigma \leq y)$ according to the value of σ as follows

$$P(Y_\sigma \leq y) = \sum_{n=1}^{\infty} P(Y_n \leq y,\ \sigma = n) \tag{4.2}$$

$$= \sum_{n=1}^{\infty} P(Y_n \leq y,\ U_n \leq \alpha f(Y_n)/g(Y_n)) P(\sigma > n-1) \tag{4.3}$$

$$= P(Y_1 \leq y,\ U_1 \leq \alpha f(Y_1)/g(Y_1)) \sum_{n=1}^{\infty} P(\sigma > n-1). \tag{4.4}$$

By independence of the pairs (Y_n, U_n) we find that

$$P(\sigma > n-1) = p^{(n-1)}$$

where $p = P(U_1 > \alpha f(Y_1)/g(Y_1))$, and

$$\sum_{n=1}^{\infty} P(\sigma > n-1) = \sum_{n=1}^{\infty} p^{(n-1)} = \frac{1}{1-p}.$$

We further find using Tonelli's theorem that

$$P(Y_1 \leq y,\ U_1 \leq \alpha f(Y_1)/g(Y_1)) = \int_{-\infty}^{y} \alpha \frac{f(z)}{g(z)} g(z) \mathrm{d}z \tag{4.5}$$

$$= \alpha \int_{-\infty}^{y} f(z) \mathrm{d}z. \tag{4.6}$$

It also follows from this, by taking $y = \infty$, that $1 - p = \alpha$, and we conclude that

$$P(Y_\sigma \leq y) = \int_{-\infty}^{y} f(z) \mathrm{d}z,$$

and the density for the distribution of Y_σ is, indeed, f. □

Note that if $\alpha f \leq g$ for *densities* f and g, then

$$\alpha = \int \alpha f(x) \mathrm{d}x \leq \int g(x) \mathrm{d}x = 1,$$

whence it follows automatically that $\alpha \leq 1$ whenever αf is dominated by g. The function g/α is called the *envelope* of f. The tighter the envelope, the smaller is the probability of rejecting a sample from g, and this is quantified explicitly by α as $1 - \alpha$ is the rejection probability. Thus α should preferably be as close to one as possible.

If $f(y) = cq(y)$ and $g(y) = dp(y)$ for (unknown) normalizing constants $c, d > 0$ and $\alpha' q \leq p$ for $\alpha' > 0$ then

$$\underbrace{\left(\frac{\alpha' d}{c}\right)}_{=\alpha} f \leq g.$$

The constant α' may be larger than 1, but from the argument above we know that $\alpha \leq 1$, and Theorem 4.2 gives that Y_σ has distribution with density f. It appears that we need to compute the normalizing constants to implement rejection sampling. However, observe that

$$u \leq \frac{\alpha f(y)}{g(y)} \Leftrightarrow u \leq \frac{\alpha' q(y)}{p(y)},$$

whence rejection sampling can actually be implemented with knowledge of the unnormalized densities and α' only and without computing c or d. This is one great advantage of rejection sampling. We should note, though, that when we don't know the normalizing constants, α' does not tell us anything about how tight the envelope is, and thus how small the rejection probability is.

Given two functions q and p, how do we then find α' so that $\alpha' q \leq p$? Consider the function

$$y \mapsto \frac{p(y)}{q(y)}$$

for $q(y) > 0$. If this function is lower bounded by a value strictly larger than zero, we can take

$$\alpha' = \inf_{y : q(y) > 0} \frac{p(y)}{q(y)} > 0.$$

We can in practice often find this value by minimizing $p(y)/q(y)$. If the minimum is zero, there is no α', and p cannot be used to construct an envelope. If the minimum is strictly positive it is the best possible choice of α'.

4.3.1 von Mises distribution

Recall the von Mises distribution from Section 1.2.1. It is a distribution on $(-\pi, \pi]$ with density

$$f(x) \propto e^{\kappa \cos(x-\mu)}$$

for parameters $\kappa > 0$ and $\mu \in (-\pi, \pi]$. Clearly, μ is a location parameter, and we fix $\mu = 0$ in the following. Simulating random variables with $\mu \neq 0$ can be achieved by (wrapped) translation of variables with $\mu = 0$.

Thus the target density is $f(x) \propto e^{\kappa \cos(x)}$. In this section we will use the uniform distribution on $(-\pi, \pi)$ as proposal distribution. It has constant density $g(x) = (2\pi)^{-1}$, but all we need is, in fact, that $g(x) \propto 1$. Since $x \mapsto 1/\exp(\kappa \cos(x)) = \exp(-\kappa \cos(x))$ attains its minimum $\exp(-\kappa)$ for $x = 0$, we find that

$$\alpha' e^{\kappa \cos(x)} = e^{\kappa(\cos(x)-1)} \leq 1,$$

with $\alpha' = \exp(-\kappa)$. The rejection test of the proposal $Y \sim g$ can therefore be carried out by testing if a uniformly distributed random variable U on $(0, 1)$ satisfies

$$U > e^{\kappa(\cos(Y)-1)}.$$

```
vMsim_slow <- function(n, kappa) {
  y <- numeric(n)
  for(i in 1:n) {
    reject <- TRUE
    while(reject) {
      y0 <- runif(1, - pi, pi)
      u <- runif(1)
      reject <- u > exp(kappa * (cos(y0) - 1))
    }
    y[i] <- y0
  }
  y
}
```

```
f <- function(x, k) exp(k * cos(x)) / (2 * pi * besselI(k, 0))
x <- vMsim_slow(100000, 0.5)
hist(x, breaks = seq(-pi, pi, length.out = 20), prob = TRUE)
curve(f(x, 0.5), -pi, pi, col = "blue", lwd = 2, add = TRUE)
x <- vMsim_slow(100000, 2)
hist(x, breaks = seq(-pi, pi, length.out = 20), prob = TRUE)
curve(f(x, 2), -pi, pi, col = "blue", lwd = 2, add = TRUE)
```

Figure 4.1 confirms that the implementation simulates from the von Mises distribution.

```
system.time(vMsim_slow(100000, kappa = 5))
```

```
##    user  system elapsed
##   2.493   0.148   2.665
```

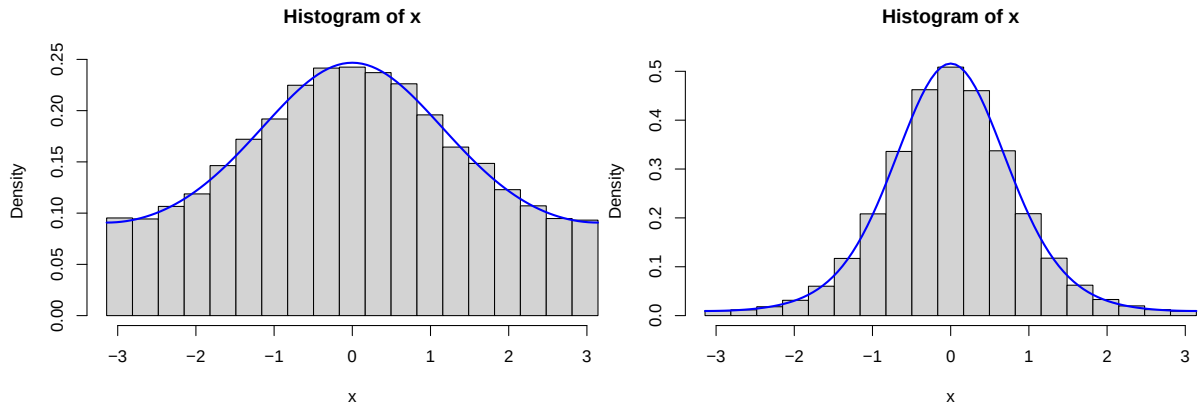
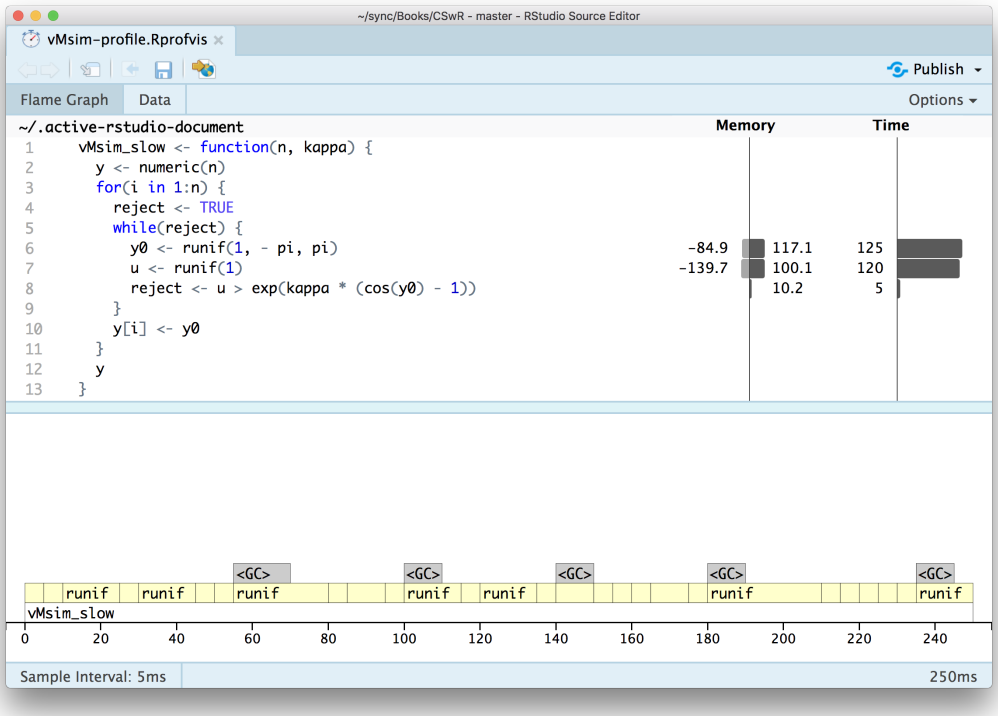


Figure 4.1: Histograms of 100,000 simulated data points from von Mises distributions with parameters $\kappa = 0.5$ (left) and $\kappa = 2$ (right). The true densities (blue) are added to the plots.

Though the implementation can easily simulate 100,000 variables in a couple of seconds, it might still be possible to improve it. To investigate what most of the run time is spent on we use the line profiling tool as implemented in the profvis package.

```
library(profvis)
profvis(vMsim_slow(10000, 5))
```



The profiling result shows that almost all the time is spent on simulating uniformly distributed random variables. It is, perhaps, expected that this should take some time, but that it takes so much more time than computing the ratio, say, used for the rejection test is a bit surprising. What might be even more surprising is the large amount of memory allocation and deallocation associated with the simulation of the variables.

The culprit is `runif` that has some overhead associated with each call. The function performs much better if called once to return a vector than if called repeatedly as above to return

just single numbers. We could rewrite the rejection sampler to make better use of `runif`, but it would make the code a bit more complicated because we don't know upfront how many uniform variables we need. This will introduce some bookkeeping that it is possible to abstract away from the implementation of any rejection sampler. Therefore we implement a generic wrapper of the random number generator that will cache a suitable amount of random variables. This function will take care of some bookkeeping and variables can then be extracted as needed. This also nicely illustrates the use of a function factory.

```
rng_stream <- function(m, rng, ...) {
  args <- list(...)
  cache <- do.call(rng, c(m, args))
  j <- 0
  fact <- 1
  next_rn <- function(r = m) {
    j <<- j + 1
    if(j > m) {
      if(fact == 1 && r < m) fact <<- m / (m - r)
      m <<- floor(fact * (r + 1))
      cache <<- do.call(rng, c(m, args))
      j <<- 1
    }
    cache[j]
  }
  next_rn
}
```

The implementation above is a function that returns a function. The returned function, `next_rn` comes with its own environment, where it stores the cached variables and extracts and returns one variable whenever called. It generates a new vector of random variables whenever it "runs out". The first time it does so, the function estimates a factor of how many variables is needed in total based on the argument `r`, and then it generates the estimated number of variables needed. This may be repeated a couple of times.

We can then reimplement `vMsim` using `rng_stream`. For later usage we add the possibility of printing out some tracing information.

```
vMsim <- function(n, kappa, trace = FALSE) {
  count <- 0
  y <- numeric(n)
  y0 <- rng_stream(n, runif, - pi, pi)
  u <- rng_stream(n, runif)
  for(i in 1:n) {
    reject <- TRUE
    while(reject) {
      count <- count + 1
      z <- y0(n - i)
      reject <- u(n - i) > exp(kappa * (cos(z) - 1))
    }
    y[i] <- z
  }
  if(trace)
    cat("kappa =", kappa, ":", (count - n)/ count, "\n")  ## Rejection frequency
  y
```

```
}
```

We should, of course, remember to test that the new implementation still generates variables from the von Mises distribution.

```
x <- vMsim(100000, 0.5)
hist(x, breaks = seq(-pi, pi, length.out = 20), prob = TRUE)
curve(f(x, 0.5), -pi, pi, col = "blue", lwd = 2, add = TRUE)
x <- vMsim(100000, 2)
hist(x, breaks = seq(-pi, pi, length.out = 20), prob = TRUE)
curve(f(x, 2), -pi, pi, col = "blue", lwd = 2, add = TRUE)
```

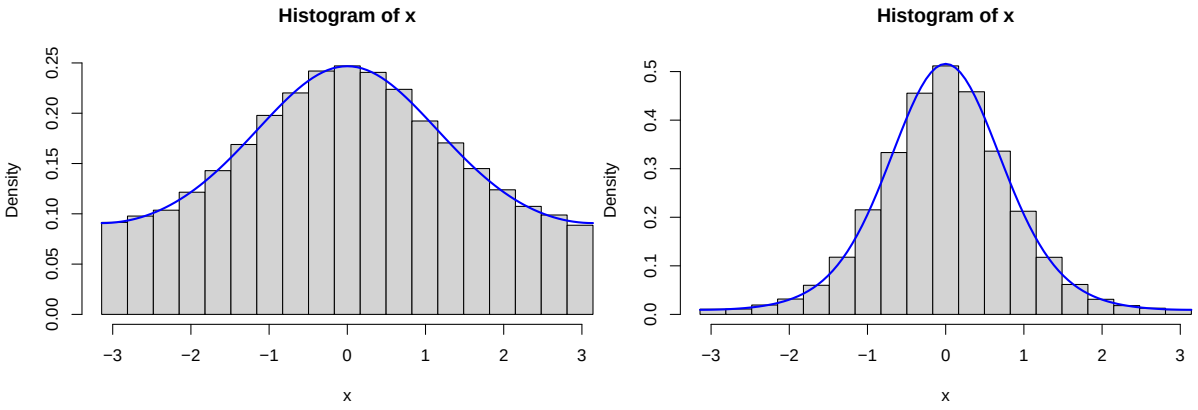


Figure 4.2: Histograms of 100,000 simulated data points from von Mises distributions with parameters $\kappa = 0.5$ (left) and $\kappa = 2$ (right), simulated using vectorized generation of random variables.

Then we can compare the run time of this new implementation to the run time of the first implementation.

```
system.time(vMsim(100000, kappa = 5))
```

```
##    user  system elapsed
##   0.825   0.009   0.843
```

As we see from the time estimate above, using a vectorized call of `runif` reduces the run time by a factor 4-5. It is possible to get a further factor 2-3 run time improvement (not shown) by implementing the computations done by `rng_stream` directly inside `vMsim`. However, we prioritize here to have modular code so that we can reuse `rng_stream` for other rejection samplers without repeating code. A pure R implementation based on a loop will never be able to compete with a C++ implementation anyway when the accept-reject step is such a simple computation.

In fact, to write a pure R function that is run time efficient, we need to turn the entire rejection sampler into a vectorized computation. That is, it is not just the generation of random numbers that need to be vectorized. There is no way around some form of loop as we don't known upfront how many rejections there will be. We can, however, benefit from the ideas in `rng_stream` on how to estimate the fraction of acceptances from a first round, which can be used for subsequent simulations. This is done in the following fully vectorized R implementation.

```
vMsim_vec <- function(n, kappa) {
  fact <- 1
```

```
  j <- 1
  l <- 0  ## The number of accepted samples
  y <- list()
  while(l < n) {
    m <- floor(fact * (n - l))  ## equals n the first time
    y0 <- runif(m, - pi, pi)
    u <- runif(m)
    accept <- u <= exp(kappa * (cos(y0) - 1))
    l <- l + sum(accept)
    y[[j]] <- y0[accept]
    j <- j + 1
    if(fact == 1) fact <- n / l
  }
  unlist(y)[1:n]
}
```

The implementation above incrementally grows a list, whose entries contain vectors of accepted samples. It is usually not advisable to dynamically grow objects (vectors or list), as this will lead to a lot of memory allocation, copying and deallocation. Thus it is better to initialize a vector of the correct size upfront. In this particular case the list will only contain few entries, and it is inconsequential that it is grown dynamically.

Finally, a C++ implementation via Rcpp is given below where the random variables are then again generated one at a time via the C-interface to R's random number generators. There is no (substantial) overhead of doing so in C++.

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector vMsim_cpp(int n, double kappa) {
  NumericVector y(n);
  double y0;
  bool reject;
  for(int i = 0; i < n; ++i) {
    do {
      y0 = R::runif(- M_PI, M_PI);
      reject = R::runif(0, 1) > exp(kappa * (cos(y0) - 1));
    } while(reject);
    y[i] = y0;
  }
  return y;
}
```

Figure 4.3 shows the results from testing the C++ implementation and the fast R implementation, and confirms that the implementations do simulate from the von Mises distribution. We conclude by measuring the run time of the implementations using `system.time` and a combined microbenchmark of all four different implementations.

```
system.time(vMsim_cpp(100000, kappa = 5))
```

```
##    user  system elapsed
##   0.045   0.001   0.045
```

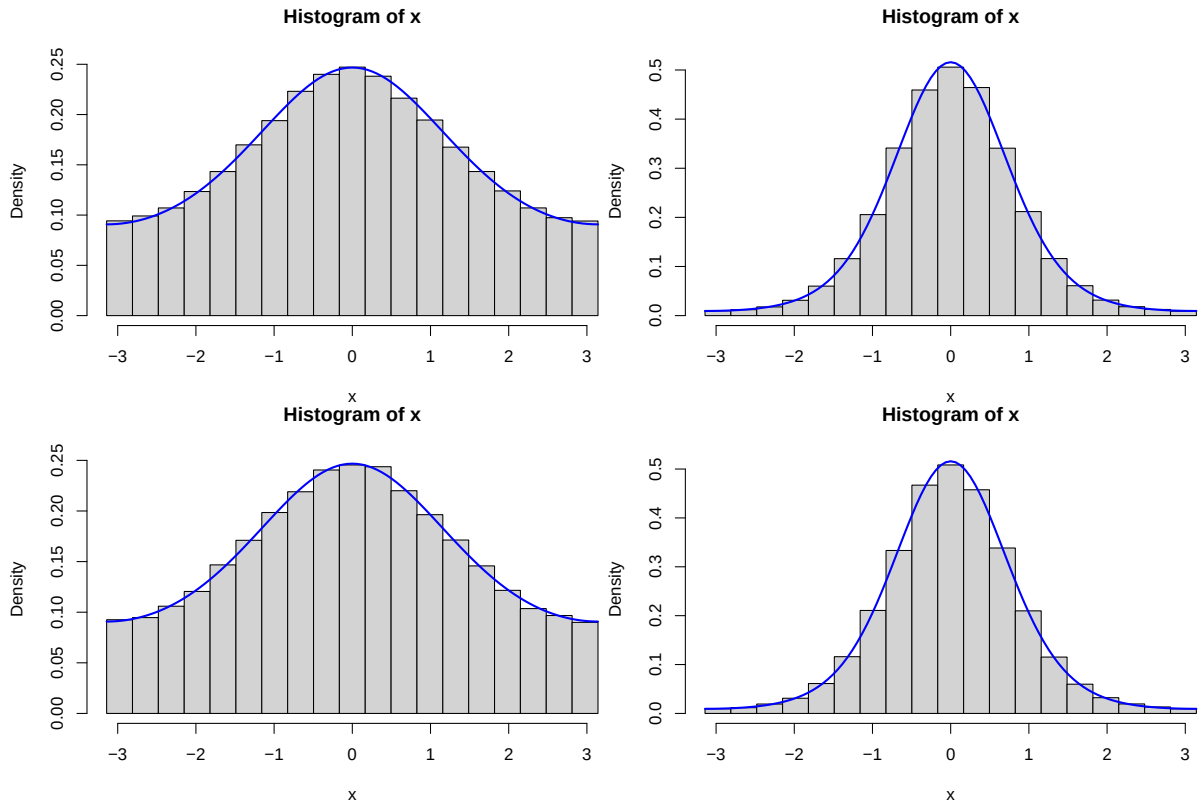


Figure 4.3: Histograms of 100,000 simulated data points from von Mises distributions with parameters $\kappa = 0.5$ (left) and $\kappa = 2$ (right), simulated using the Rcpp implementation (top) and the fully vectorized R implementation (bottom).

```
microbenchmark(
  vMsim_slow(1000, kappa = 5),
  vMsim(1000, kappa = 5),
  vMsim_vec(1000, kappa = 5),
  vMsim_cpp(1000, kappa = 5)
)
```

```
## Unit: microseconds
##                          expr   min    lq  mean median    uq   max neval cld
##  vMsim_slow(1000, kappa = 5) 17247 18671 24982  20095 22266 82899   100   c
##       vMsim(1000, kappa = 5)  6531  7092  7706   7373  7916 12309   100  b
##    vMsim_vec(1000, kappa = 5)   440   519   591    575   632   919   100 a
##    vMsim_cpp(1000, kappa = 5)   317   341   380    362   401   620   100 a
```

The C++ implementation is only a factor 1.5 faster than the fully vectorized R implementation, while it is around a factor 15 faster than the loop-based `vMsim` and a factor 85 or so faster than the first implementation `vMsim_slow`. Rejection sampling is a good example of an algorithm for which a naive loop-based R implementation performs rather poorly in terms of run time, while a completely vectorized implementation is competitive with an Rcpp implementation.

4.3.2 Gamma distribution

It may be possible to find a suitable envelope of the density for the gamma distribution on $(0, \infty)$, but it turns out that there is a very efficient rejection sampler of a non-standard

distribution that can be transformed into a gamma distribution by a simple transformation.

Let $t(y) = a(1+by)^3$ for $y \in (-b^{-1}, \infty)$, then $t(Y) \sim \Gamma(r,1)$ if $r \geq 1$ and Y has density

$$f(y) \propto t(y)^{r-1} t'(y) e^{-t(y)} = e^{(r-1)\log t(y) + \log t'(y) - t(y)}.$$

The proof of this follows from a simple univariate density transformation theorem, but see also the original paper Marsaglia and Tsang (2000) that proposed the rejection sampler discussed in this section. The density f will be the *target density* for a rejection sampler.

With

$$f(y) \propto e^{(r-1)\log t(y) + \log t'(y) - t(y)},$$

$a = r - 1/3$ and $b = 1/(3\sqrt{a})$

$$f(y) \propto e^{a \log t(y)/a - t(y) + a \log a} \propto \underbrace{e^{a \log t(y)/a - t(y) + a}}_{q(y)}.$$

An analysis of $w(y) := -y^2/2 - \log q(y)$ shows that it is convex on $(-b^{-1}, \infty)$ and it attains its minimum in 0 with $w(0) = 0$, whence

$$q(y) \leq e^{-y^2/2}.$$

This gives us an envelope expressed in terms of unnormalized densities with $\alpha' = 1$.

The implementation of a rejection sampler based on this analysis is relatively straightforward. The rejection sampler will simulate from the distribution with density f by simulating from the Gaussian distribution (the envelope). For the rejection step we need to implement q. Finally, we also need to implement t to transform the result from the rejection sampler to be gamma distributed. The rejection sampler is otherwise implemented as for the non-vectorized von Mises distribution. To investigate rejection probabilities below we additionally implement the possibility of printing out some tracing information.

```
## r >= 1
tfun <- function(y, a) {
  b <- 1 / (3 * sqrt(a))
  (y > -1/b) * a * (1 + b * y)^3  ## 0 when y <= -1/b
}

qfun <- function(y, r) {
  a <- r - 1/3
  tval <- tfun(y, a)
  exp(a * log(tval / a) - tval + a)
}

gammasim <- function(n, r, trace = FALSE) {
  count <- 0
  y <- numeric(n)
  y0 <- rng_stream(n, rnorm)
  u <- rng_stream(n, runif)
  for(i in 1:n) {
    reject <- TRUE
    while(reject) {
      count <- count + 1
```

```
      z <- y0(n - i)
      reject <- u(n - i) > qfun(z, r) * exp(z^2/2)
    }
    y[i] <- z
  }
  if(trace)
    cat("r =", r, ":", (count - n)/ count, "\n")  ## Rejection frequency
  tfun(y, r - 1/3)
}
```

We test the implementation by simulating 100,000 values with parameters $r = 8$ as well as $r = 1$ and compare the resulting histograms to the respective theoretical densities.

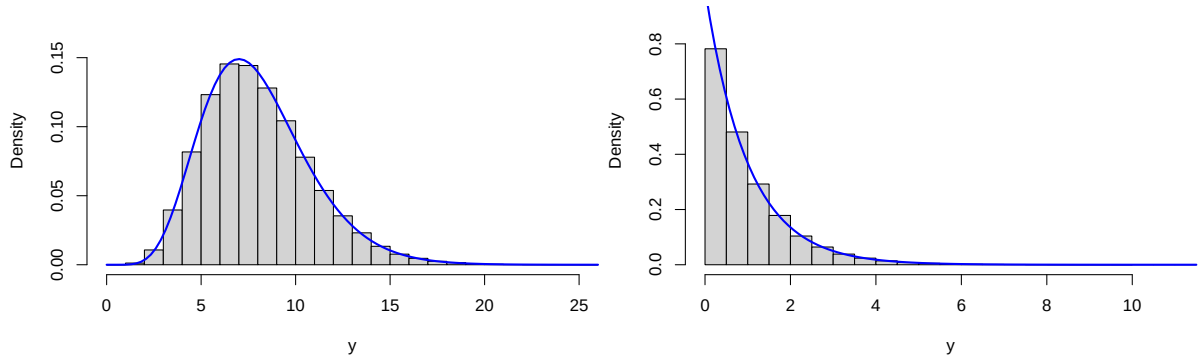


Figure 4.4: Histograms of simulated gamma distributed variables with shape parameters $r = 8$ (left) and $r = 1$ (right) with corresponding theoretical densities (blue).

Though this is only a simple and informal test, it indicates that the implementation correctly simulates from the gamma distribution.

Rejection sampling can be computationally expensive if many samples are rejected. A very tight envelope will lead to fewer rejections, while a loose envelope will lead to many rejections. Using the tracing option as implemented we obtain estimates of the rejection probability and thus a quantification of how tight the envelope is.

```
y <- gammasim(100000, 16, trace = TRUE)
y <- gammasim(100000, 8, trace = TRUE)
y <- gammasim(100000, 4, trace = TRUE)
y <- gammasim(100000, 1, trace = TRUE)
```

```
## r = 16 : 0.001667216
## r = 8 : 0.003497723
## r = 4 : 0.008310359
## r = 1 : 0.04872434
```

We observe that the rejection frequencies are small with $r = 1$ being the worst case with around 5% rejections. For the other cases the rejection frequencies are all below 1%, thus rejection is rare.

A visual comparison of q to the (unnormalized) Gaussian density also shows that the two (unnormalized) densities are very close except in the tails where there is very little probability mass.

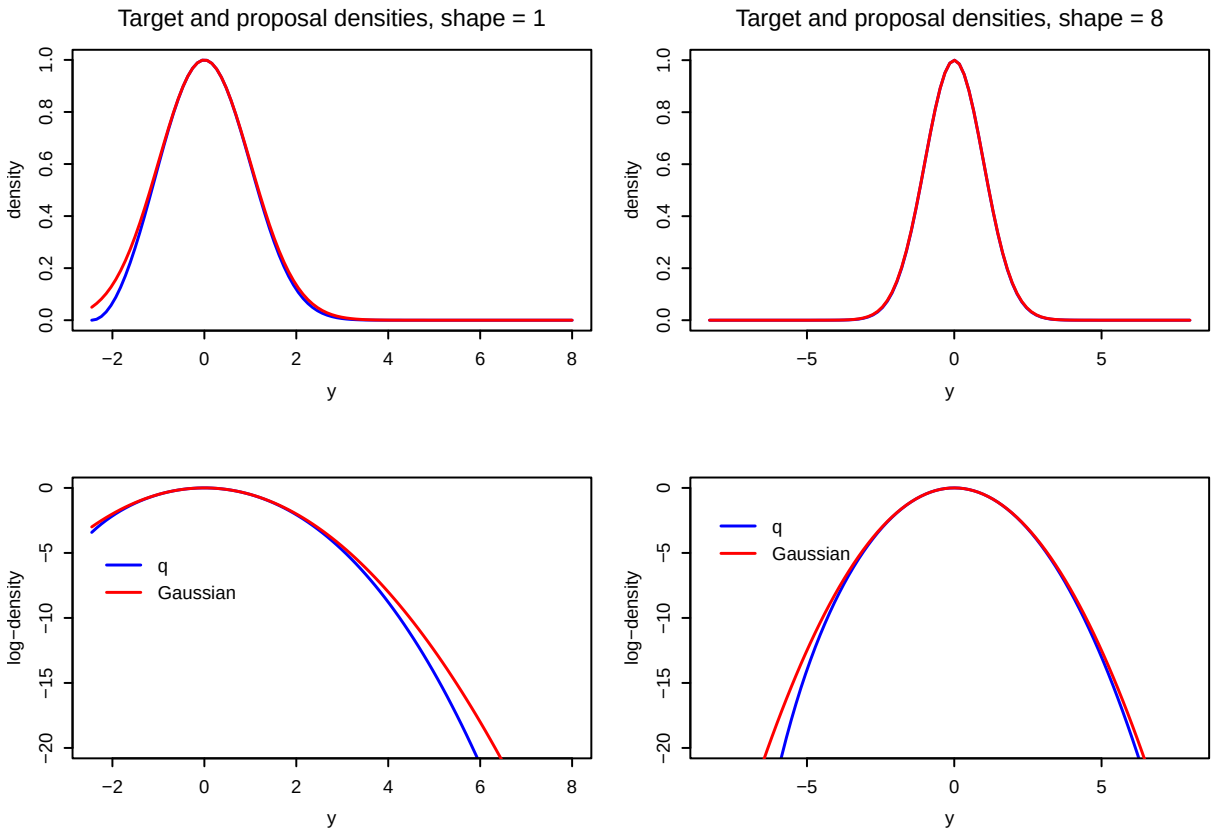


Figure 4.5: Comparisons of the Gaussian proposal (red) and the target density (blue) used for eventually simulating gamma distributed variables via a transformation.

4.4 Adaptive envelopes

A good envelope should be tight, meaning that α is close to one, it should be fast to simulate from and have a density that is fast to evaluate. It is not obvious how to find such an envelope for an arbitrary target density f.

This section develops a general scheme for the construction of envelopes for all log-concave target densities. This is a special class of densities, but it is not uncommon in practice. The scheme can also be extended to work for some densities that have combinations of log-concave and log-convex behaviors. The same idea used for constructing envelopes can be used to bound f from below. The accept-reject step can then avoid many evaluations of f, which is beneficial if f is computationally expensive to evaluate.

The key idea of the scheme is to bound the log-density by piecewise affine functions. This is particularly easy to do if the density is log-concave. The scheme leads to analytically manageable formulas for the envelope, its corresponding distribution function and its inverse, and as a result it is fast to simulate proposals and compute the envelope as needed in the accept-reject step.

The scheme requires the choice of a finite number of points to determine the affine bounds. For any given choice of points the scheme adapts the envelope to the target density automatically. It is possible to implement a *fully* adaptive scheme that doesn't even require the choice of points but initializes and updates the points dynamically as more and more rejection samples are computed. In this section the focus is on the scheme with a given and fixed number of points.

For a continuously differentiable, strictly positive and log-concave target on an open interval $I \subseteq \mathbb{R}$ it holds that

$$\log(f(x)) \leq \frac{f'(x_0)}{f(x_0)}(x - x_0) + \log(f(x_0))$$

for any $x, x_0 \in I$.

Let $x_1 < x_2 < \ldots < x_m \in I$ and let $I_1, \ldots, I_m \subseteq I$ be intervals that form a partition of I such that $x_i \in I_i$. Defining

$$a_i = (\log(f(x_i)))' = \frac{f'(x_i)}{f(x_i)} \quad \text{and} \quad b_i = \log(f(x_i)) - a_i x_i$$

we find the upper bound

$$\log(f(x)) \leq V(x) = \sum_{i=1}^{m} (a_i x + b_i) 1_{I_i}(x),$$

or

$$f(x) \leq e^{V(x)}.$$

Note that by the log-concavity of f, $a_1 \geq a_2 \geq \ldots \geq a_m$. The upper bound is integrable over I if either $a_1 > 0$ and $a_m < 0$, or $a_m < 0$ and I is bounded to the left, or $a_1 > 0$ and I is bounded to the right. In any of these cases we define

$$c = \int_I e^{V(x)} \mathrm{d}x < \infty$$

and $g(x) = c^{-1} \exp(V(x))$, and we find that with $\alpha = c^{-1}$ then $\alpha f \leq g$ and g is an envelope of f. Note that it is actually not necessary to compute c (or α) to implement the rejection step in the rejection sampler, but that c is needed for simulating from g as described below. We will assume in the following that $c < \infty$.

The intervals I_i have not been specified, and we could, in fact, implement rejection sampling with any choice of intervals fulfilling the conditions above. But in the interest of maximizing α (minimizing c) and thus minimizing the rejection frequency, we should choose I_i so that $a_i x + b_i$ is minimal over I_i among all the affine upper bounds. This will result in the tightest envelope. This means that for $i = 1, \ldots, m-1$, $I_i = (z_{i-1}, z_i]$ with z_i the point where $a_i x + b_i$ and $a_{i+1} x + b_{i+1}$ intersect. We find that the solution of

$$a_i x + b_i = a_{i+1} x + b_{i+1}$$

is

$$z_i = \frac{b_{i+1} - b_i}{a_i - a_{i+1}}$$

provided that $a_{i+1} > a_i$. The two extremes, z_0 and z_m, are chosen as the endpoints of I and may be $-\infty$ and $+\infty$, respectively.

One way to simulate from such envelopes is by transformation of uniform random variables by the inverse distribution function. It requires a little bookkeeping, but is otherwise straightforward. Define for $x \in I_i$

$$F_i(x) = \int_{z_{i-1}}^{x} e^{a_i z + b_i} \mathrm{d}z,$$

and let $R_i = F_i(z_i)$. Then $c = \sum_{i=1}^{m} R_i$, and if we define $Q_i = \sum_{k=1}^{i} R_k$ for $i = 0, \ldots, m$ the inverse of the distribution function in q is given as the solution to the equation

$$F_i(x) = cq - Q_{i-1}, \qquad Q_{i-1} < cq \leq Q_i.$$

That is, for a given $q \in (0,1)$, first determine which interval $(Q_{i-1}, Q_i]$ that cq falls into, and then solve the corresponding equation. Observe that when $a_i \neq 0$,

$$F_i(x) = \frac{1}{a_i} e^{b_i} \left(e^{a_i x} - e^{a_i z_{i-1}} \right).$$

4.4.1 Beta distribution

To illustrate the envelope construction above for a simple log-concave density we consider the Beta distribution on $(0,1)$ with shape parameters ≥ 1. This distribution has density

$$f(x) \propto x^{\alpha-1}(1-x)^{\beta-1},$$

which is log-concave (when the shape parameters are greater than one). We implement the rejection sampling algorithm for this density with the adaptive envelope using two points.

```
Betasim <- function(n, x1, x2, alpha, beta) {
  lf <- function(x) (alpha - 1) * log(x) + (beta - 1) * log(1 - x)
  lf_deriv <- function(x) (alpha - 1)/x - (beta - 1)/(1 - x)
  a1 <- lf_deriv(x1)
  a2 <- lf_deriv(x2)
  if(a1 == 0 || a2 == 0 || a1 - a2 == 0)
    stop("\nThe implementation requires a_1 and a_2 different
and both different from zero. Choose different values of x_1 and x_2.")
  b1 <- lf(x1) - a1 * x1
  b2 <- lf(x2) - a2 * x2
  z1 <- (b2 - b1) / (a1 - a2)
  Q1 <- exp(b1) * (exp(a1 * z1) - 1) / a1
  c <- Q1 + exp(b2) * (exp(a2 * 1) - exp(a2 * z1)) / a2

  y <- numeric(n)
  uy <- rng_stream(n, runif)
  u <- rng_stream(n, runif)
  for(i in 1:n) {
    reject <- TRUE
    while(reject) {
      u0 <- c * uy(n - i)
      if(u0 < Q1) {
        z <- log(a1 * exp(-b1) * u0 + 1) / a1
        reject <- u(n - i) > exp(lf(z) - a1 * z - b1)
      } else {
        z <- log(a2 * exp(-b2) * (u0 - Q1) + exp(a2 * z1)) / a2
        reject <- u(n - i) > exp(lf(z) - a2 * z - b2)
      }
    }
    y[i] <- z
  }
  y
}
```

Note that as a safeguard we implemented a test on the a_i-s to check that the formulas used are actually meaningful, specifically that there are no divisions by zero.

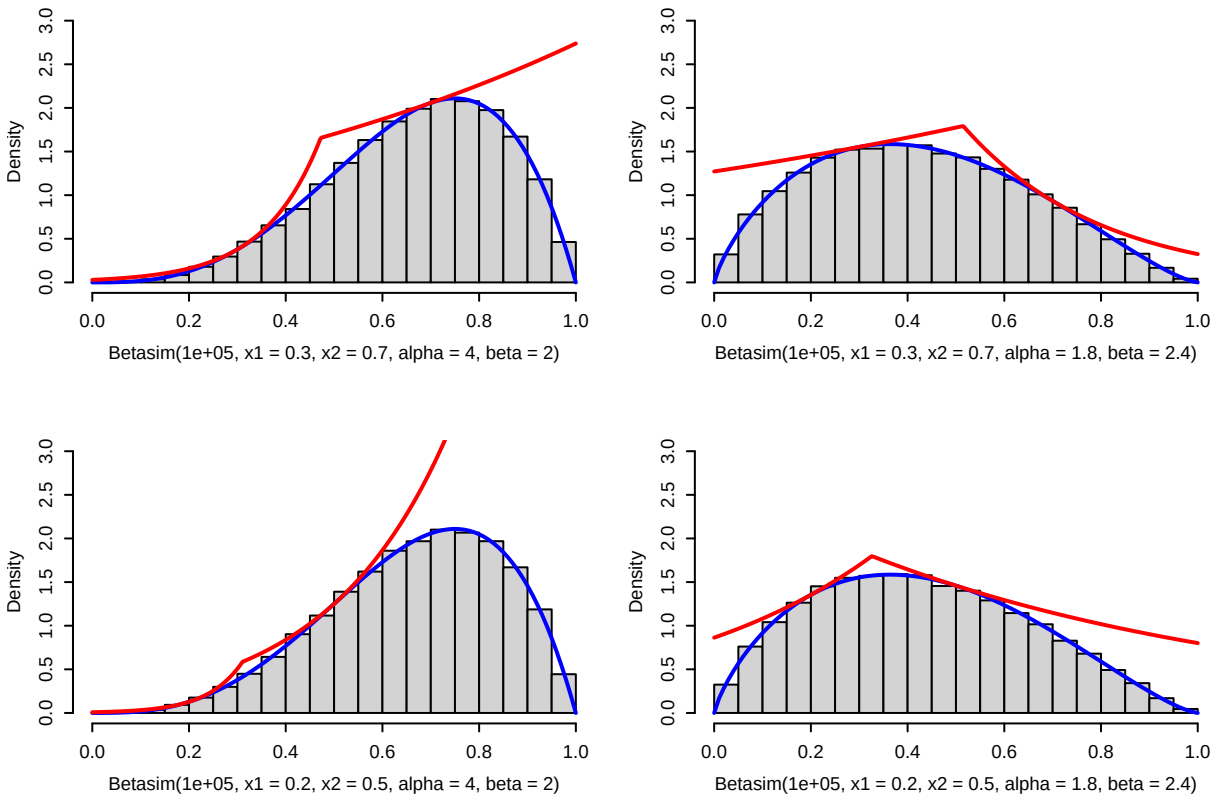


Figure 4.6: Histograms of simulated variables from Beta distributions using the rejection sampler with the adaptive envelope based on log-concavity. The true density (blue) and the envelope (red) are added to the plots.

```r
Betasim(1, x1 = 0.25, x2 = 0.75, alpha = 4, beta = 2)
```

```
## Error in Betasim(1, x1 = 0.25, x2 = 0.75, alpha = 4, beta = 2): ##
The implementation requires a_1 and a_2 different ## and both different
from zero. Choose different values of x_1 and x_2.```
```

```r
Betasim(1, x1 = 0.2, x2 = 0.75, alpha = 4, beta = 2)
```

```
## Error in Betasim(1, x1 = 0.2, x2 = 0.75, alpha = 4, beta = 2): ##
The implementation requires a_1 and a_2 different ## and both different
from zero. Choose different values of x_1 and x_2.```
```

```r
Betasim(1, x1 = 0.2, x2 = 0.8, alpha = 4, beta = 2)
```

```
## [1] 0.6112477
```

4.4.2 von Mises distribution

The von Mises rejection sampler in Section 4.3.1 used the uniform distribution as proposal distribution. As it turns out, the uniform density is not a particularly tight envelope. We illustrate this by studying the proportion of rejections for our previous implementation.

```
y <- vMsim(10000, 0.1, trace = TRUE)
y <- vMsim(10000, 0.5, trace = TRUE)
y <- vMsim(10000, 2, trace = TRUE)
y <- vMsim(10000, 5, trace = TRUE)
```

```
## kappa = 0.1 : 0.08958485
## kappa = 0.5 : 0.3567063
## kappa = 2 : 0.6917386
## kappa = 5 : 0.8161798
```

The rejection frequency is high and increases with κ. For $\kappa = 5$ more than 80% of the proposals are rejected, and simulating $n = 10,000$ von Mises distributed variables thus requires the simulation of around $50,000$ variables from the proposal.

The von Mises density is, unfortunately, not log-concave on $(-\pi, \pi)$, but it is on $(-\pi/2, \pi/2)$. It is, furthermore, log-convex on $(-\pi, -\pi/2)$ as well as $(\pi/2, \pi)$, which implies that on these two intervals the log-density is below the corresponding chords. These chords can be pieced together with tangents to give an envelope.

```
vMsim_adapt <- function(n, x1, x2, kappa, trace = FALSE) {
  lf <- function(x) kappa * cos(x)
  lf_deriv <- function(x) - kappa * sin(x)
  a1 <- 2 * kappa / pi
  a2 <- lf_deriv(x1)
  a3 <- lf_deriv(x2)
  a4 <- - a1

  b1 <- kappa
  b2 <- lf(x1) - a2 * x1
  b3 <- lf(x2) - a3 * x2
  b4 <- kappa

  z0 <- -pi
  z1 <- -pi/2
  z2 <- (b3 - b2) / (a2 - a3)
  z3 <- pi/2
  z4 <- pi

  Q1 <- exp(b1) * (exp(a1 * z1) - exp(a1 * z0)) / a1
  Q2 <- Q1 + exp(b2) * (exp(a2 * z2) - exp(a2 * z1)) / a2
  Q3 <- Q2 + exp(b3) * (exp(a3 * z3) - exp(a3 * z2)) / a3
  c <- Q3 + exp(b4) * (exp(a4 * z4) - exp(a4 * z3)) / a4

  count <- 0
  y <- numeric(n)
  uy <- rng_stream(n, runif)
  u <- rng_stream(n, runif)
  for(i in 1:n) {
    reject <- TRUE
    while(reject) {
      count <- count + 1
      u0 <- c * uy(n - i)
```

```
      if(u0 < Q1) {
        z <- log(a1 * exp(-b1) * u0 + exp(a1 * z0)) / a1
        reject <- u(n - i) > exp(lf(z) - a1 * z - b1)
      } else if(u0 < Q2) {
        z <- log(a2 * exp(-b2) * (u0 - Q1) + exp(a2 * z1)) / a2
        reject <- u(n - i) > exp(lf(z) - a2 * z - b2)
      } else if(u0 < Q3) {
        z <- log(a3 * exp(-b3) * (u0 - Q2) + exp(a3 * z2)) / a3
        reject <- u(n - i) > exp(lf(z) - a3 * z - b3)
      } else {
        z <- log(a4 * exp(-b4) * (u0 - Q3) + exp(a4 * z3)) / a4
        reject <- u(n - i) > exp(lf(z) - a4 * z - b4)
      }
    }
    y[i] <- z
  }
  if(trace)
    cat("kappa =", kappa, ", x1 =", x1,
        ", x2 =", x2, ":", (count - n) / count, "\n")
  y
}
```

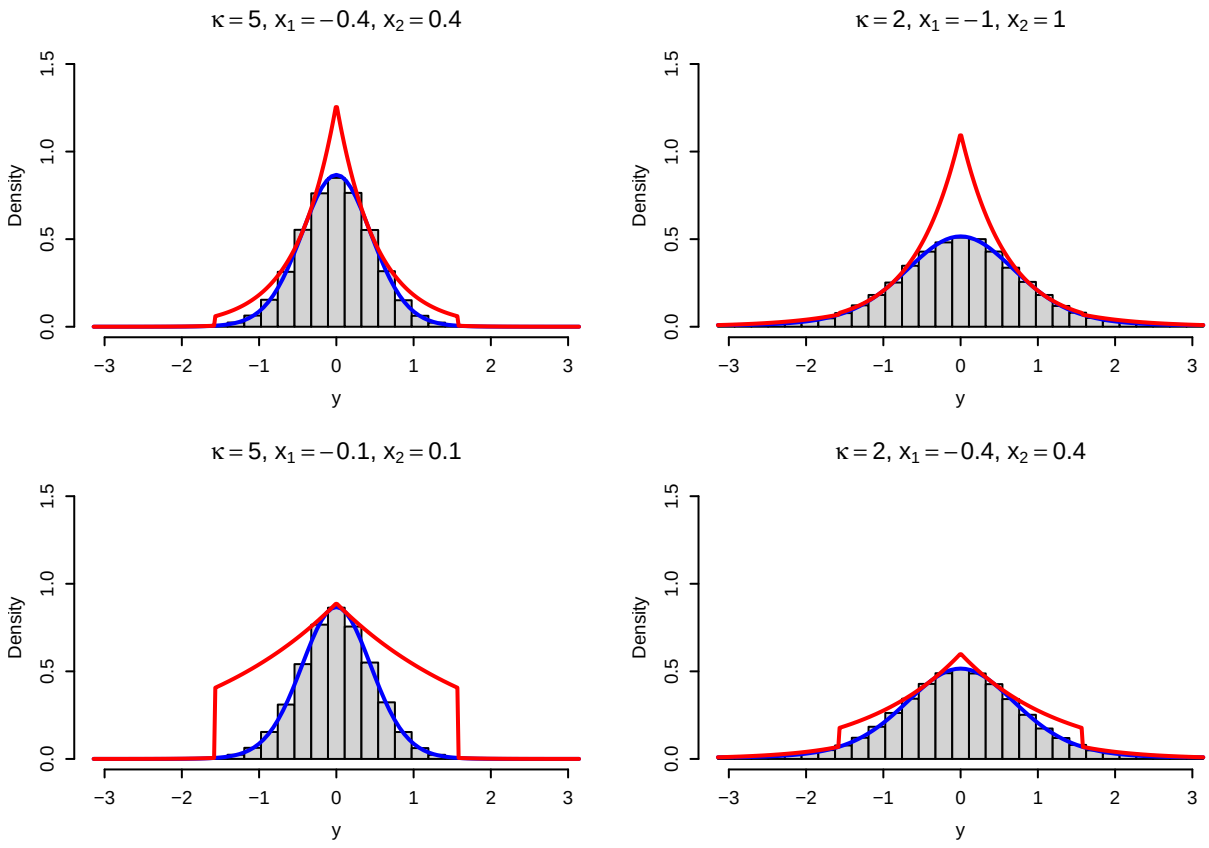


Figure 4.7: Histograms of simulated variables from von Mises distributions using the rejection sampler with the adaptive envelope based on a combination of log-concavity and log-convexity. The true density (blue) and the envelope (red) are added to the plots.

```
y <- vMsim_adapt(100000, -0.4, 0.4, 5, trace = TRUE)
y <- vMsim_adapt(100000, -1, 1, 2, trace = TRUE)
y <- vMsim_adapt(100000, -0.1, 0.1, 5, trace = TRUE)
y <- vMsim_adapt(100000, -0.4, 0.4, 2, trace = TRUE)
```

```
## kappa = 5 , x1 = -0.4 , x2 = 0.4 : 0.198872
## kappa = 2 , x1 = -1 , x2 = 1 : 0.2409752
## kappa = 5 , x1 = -0.1 , x2 = 0.1 : 0.4847459
## kappa = 2 , x1 = -0.4 , x2 = 0.4 : 0.1556621
```

We see that compared to using the uniform density as envelope, these adaptive envelopes are generally tighter and leads to fewer rejections. Even tighter envelopes are possible by using more than four intervals, but it is, of course, always a good question how the added complexity and bookkeeping induced by using more advanced and adaptive envelopes affect run time. It is even a good question if our current adaptive implementation will outperform our first, and much simpler, implementation that used the uniform envelope.

```
microbenchmark(vMsim_adapt(100, -1, 1, 5),
               vMsim_adapt(100, -0.4, 0.4, 5),
               vMsim_adapt(100, -0.2, 0.2, 5),
               vMsim_adapt(100, -0.1, 0.1, 5),
               vMsim(100, 5),
               vMsim_vec(100, 5)
               )
```

```
## Unit: microseconds
##                              expr min  lq mean median  uq   max neval cld
##      vMsim_adapt(100, -1, 1, 5) 544 628  707    677 750  1136   100   b
##  vMsim_adapt(100, -0.4, 0.4, 5) 283 314  338    323 344   512   100  ab
##  vMsim_adapt(100, -0.2, 0.2, 5) 316 369  406    387 415   713   100  ab
##  vMsim_adapt(100, -0.1, 0.1, 5) 388 460  801    490 545 29145   100   b
##                  vMsim(100, 5) 577 664  738    717 799  1077   100   b
##               vMsim_vec(100, 5)  54  68   82     75  88   177   100  a
```

The results from the benchmark show that the adaptive implementation has run time comparable to using the uniform proposal. With $x_1 = -0.4$ and $x_2 = 0.4$ and $\kappa = 5$ we found above that the rejection frequency was about 20% with the adaptive envelope, while it was about 80% when using the uniform envelope. A naive computation would thus suggest a speedup of a factor 4, but using the adaptive envelope there is actually only a speedup of a factor 2. And the vectorized solution is still considerably faster. A completely vectorized solution using the adaptive envelope is possible, but it is not entirely straightforward how to implement the more complicated envelope efficiently, and it may be a better option in this case to implement it using Rcpp.

Even if either implementation can be improved further in terms of run time, it is an important point when comparing algorithms that we don't get too focused on surrogate performance quantities. The probability of rejection is a surrogate for actual run time, and it might be conceptually of interest to bring this probability down. But if it is at the expense of additional computations it might not be worth the effort in terms of real run time.

4.5 Exercises

4.5.1 Rejection sampling of Gaussian random variables

This exercise is on rejection sampling from the Gaussian distribution by using the Laplace distribution as an envelope. Recall that the Laplace distribution has density

$$g(x) = \frac{1}{2} e^{-|x|}$$

for $x \in \mathbb{R}$.

Note that if X and Y are independent and exponentially distributed with mean one, then $X - Y$ has a Laplace distribution. This gives a way to easily sample from the Laplace distribution.

Exercise 4.1. Implement rejection sampling from the standard Gaussian distribution with density

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

by simulating Laplace random variables as differences of exponentially distributed random variables. Test the implementation by computing the variance of the Gaussian distribution as an MC estimate and by comparing directly with the Gaussian distribution using histograms and QQ-plots.

Exercise 4.2. Implement simulation from the Laplace distribution by transforming a uniform random variable by the inverse distribution function. Use this method together with the rejection sampler you implemented in Exercise 4.1

Note: The Laplace distribution can be seen as a simple version of the adaptive envelopes suggested in Section 4.4.

Chapter 5

Monte Carlo integration

A typical usage of simulation of random variables is Monte Carlo integration. With $X_1, \ldots, X_n$ i.i.d. with density f

$$\hat{\mu}_{\text{MC}} := \frac{1}{n} \sum_{i=1}^{n} h(X_i) \to \mu := E(h(X_1)) = \int h(x) f(x) \, \mathrm{d}x$$

for $n \to \infty$ by the law of large numbers (LLN).

Monte Carlo integration is a clever idea, where we use the computer to simulate i.i.d. random variables and compute an average as an approximation of an integral. The idea may be applied in a statistical context, but it way also have applications outside of statistics and be a direct competitor to numerical integration. By increasing n the LLN tells us that the average will eventually become a good approximation of the integral. However, the LLN does not quantify how large n should be, and a fundamental question of Monte Carlo integration is therefore to quantify the precision of the average.

This chapter first deals with the quantification of the precision — mostly via the asymptotic variance in the central limit theorem. This will on the one hand provide us with a quantification of precision for any specific Monte Carlo approximation, and it will on the other hand provide us with a way to compare different Monte Carlo integration techniques. The direct use of the average above requires that we can simulate from the distribution with density f, but that might have low precision or it might just be plain difficult. In the second half of the chapter we will treat importance sampling, which is a technique for simulating from a different distribution and use a *weighted* average to obtain the approximation of the integral.

5.1 Assessment

The error analysis of Monte Carlo integration differs from that of ordinary (deterministic) numerical integration methods. For the latter, error analysis provides bounds on the error of the computable approximation in terms of properties of the function to be integrated. Such bounds provide a guarantee on what the error at most can be. It is generally impossible to provide such a guarantee when using Monte Carlo integration because the computed approximation is by construction (pseudo)random. Thus the error analysis and assessment of the precision of $\hat{\mu}_{\text{MC}}$ as an approximation of μ will be probabilistic.

There are two main approaches. We can use approximations of the distribution of $\hat{\mu}_{\text{MC}}$ to assess the precision by computing a confidence interval, say. Or we can provide finite sample upper bounds, known as concentration inequalities, on the probability that the error of $\hat{\mu}_{\text{MC}}$ is

larger than a given ε. A concentration inequality can be turned into a confidence interval, if needed, or it can be used directly to answer a question such as: if I want the approximation to have an error smaller than $\varepsilon = 10^{-3}$, how large does n need to be to guarantee this error bound with probability at least 99.99%?

Confidence intervals are typically computed using the central limit theorem and an estimated value of the asymptotic variance. The most notable practical problem is the estimation of that asymptotic variance, but otherwise the method is straightforward to use. A major deficit of this method is that the central limit theorem does not provide bounds – only approximations of unknown precision for a finite n. Thus without further analysis, we cannot really be certain that the results from the central limit theorem reflect the accuracy of $\hat{\mu}_{\text{MC}}$. Concentration inequalities provide actual guarantees, albeit probabilistic. They are, however, typically problem specific and harder to derive, they involve constants that are difficult to compute or estimate, and they tend to be pessimistic in real applications. The focus in this chapter is therefore on using the central limit theorem, but we do emphasize the example in Section 5.2.2 that shows how potentially misleading confidence intervals can be when the convergence is slow.

5.1.1 Using the central limit theorem

The CLT gives that

$$\hat{\mu}_{\text{MC}} = \frac{1}{n}\sum_{i=1}^{n} h(X_i) \overset{\text{approx}}{\sim} \mathcal{N}(\mu, \sigma^2_{\text{MC}}/n)$$

where

$$\sigma^2_{\text{MC}} = V(h(X_1)) = \int (h(x) - \mu)^2 f(x)\, \mathrm{d}x.$$

We can estimate σ^2_{MC} using the empirical variance

$$\hat{\sigma}^2_{\text{MC}} = \frac{1}{n-1}\sum_{i=1}^{n} (h(X_i) - \hat{\mu}_{\text{MC}})^2,$$

then the variance of $\hat{\mu}_{\text{MC}}$ is estimated as $\hat{\sigma}^2_{\text{MC}}/n$ and a standard 95% confidence interval for μ is

$$\hat{\mu}_{\text{MC}} \pm 1.96 \frac{\hat{\sigma}_{\text{MC}}}{\sqrt{n}}.$$

```
n <- 1000
x <- rgamma(n, 8) # h(x) = x
mu_hat <- (cumsum(x) / (1:n)) # Cumulative average
sigma_hat <- sd(x)
mu_hat[n]  # Theoretical value 8
```

```
## [1] 8.09556
```

```
sigma_hat  # Theoretical value sqrt(8) = 2.8284
```

```
## [1] 2.861755
```

```
qplot(1:n, mu_hat) +
  geom_ribbon(
  mapping = aes(
    ymin = mu_hat - 1.96 * sigma_hat / sqrt(1:n),
    ymax = mu_hat + 1.96 * sigma_hat / sqrt(1:n)
```

```
  ), fill = "gray") +
  coord_cartesian(ylim = c(6, 10)) +
  geom_line() +
  geom_point()
```

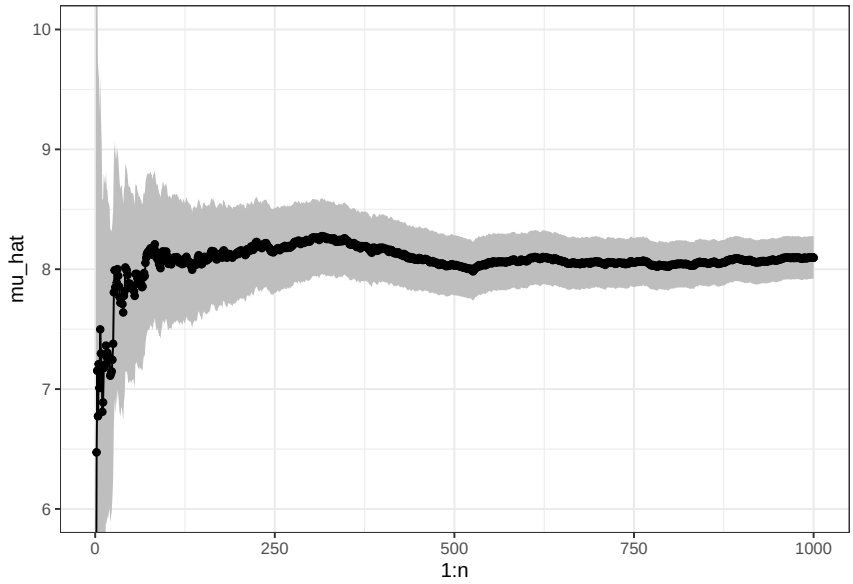


Figure 5.1: Sample path with confidence band for Monte Carlo integration of the mean of a gamma distributed random variable.

5.1.2 Concentration inequalities

If X is a real valued random variable with finite second moment, $\mu = E(X)$ and $\sigma^2 = V(X)$, Chebychev's inequality holds

$$P(|X - \mu| > \varepsilon) \leq \frac{\sigma^2}{\varepsilon^2}$$

for all $\varepsilon > 0$. This inequality implies, for instance, that for the simple Monte Carlo average we have the inequality

$$P(|\hat{\mu}_{\text{MC}} - \mu| > \varepsilon) \leq \frac{\sigma^2_{\text{MC}}}{n\varepsilon^2}.$$

A common usage of this inequality is for the qualitative statement known as the *law of large numbers*: for any $\varepsilon > 0$

$$P(|\hat{\mu}_{\text{MC}} - \mu| > \varepsilon) \to 0$$

for $n \to \infty$. Or $\hat{\mu}_{\text{MC}}$ *converges in probability* towards μ as n tends to infinity. However, the inequality actually also provides a quantitative statement about how accurate $\hat{\mu}_{\text{MC}}$ is as an approximation of μ.

Chebyshev's inequality is useful due to its minimal assumption of a finite second moment. However, it typically doesn't give a very tight bound on the probability $P(|X - \mu| > \varepsilon)$. Much better inequalities can be obtained under stronger assumptions, in particular finite exponential moments.

Assuming that the moment generating function of X is finite, $M(t) = E(e^{tX}) < \infty$, for some suitable $t \in \mathbb{R}$, it follows from Markov's inequality that

$$P(X - \mu > \varepsilon) = P(e^{tX} > e^{t(\varepsilon+\mu)}) \leq e^{-t(\varepsilon+\mu)} M(t),$$

which can provide a very tight upper bound by minimizing the bound over t. This requires some knowledge of the moment generating function. We illustrate the usage of this inequality below by considering the gamma distribution where the moment generating function is well known.

5.1.3 Exponential tail bound for Gamma distributed variables

If X follows a Gamma distribution with shape parameter $\lambda > 0$ and $t < 1$, then

$$M(t) = \frac{1}{\Gamma(\lambda)} \int_0^\infty x^{\lambda-1} e^{-(1-t)x} \, \mathrm{d}x = \frac{1}{(1-t)^\lambda}.$$

Whence

$$P(X - \lambda > \varepsilon) \leq e^{-t(\varepsilon+\lambda)} \frac{1}{(1-t)^\lambda}.$$

Minimization over t of the right hand side gives the minimizer $t = \varepsilon/(\varepsilon + \lambda)$ and the upper bound

$$P(X - \lambda > \varepsilon) \leq e^{-\varepsilon} \left(\frac{\varepsilon + \lambda}{\lambda} \right)^\lambda.$$

Compare this to the bound

$$P(|X - \lambda| > \varepsilon) \leq \frac{\lambda}{\varepsilon^2}$$

from Chebychev's inequality.

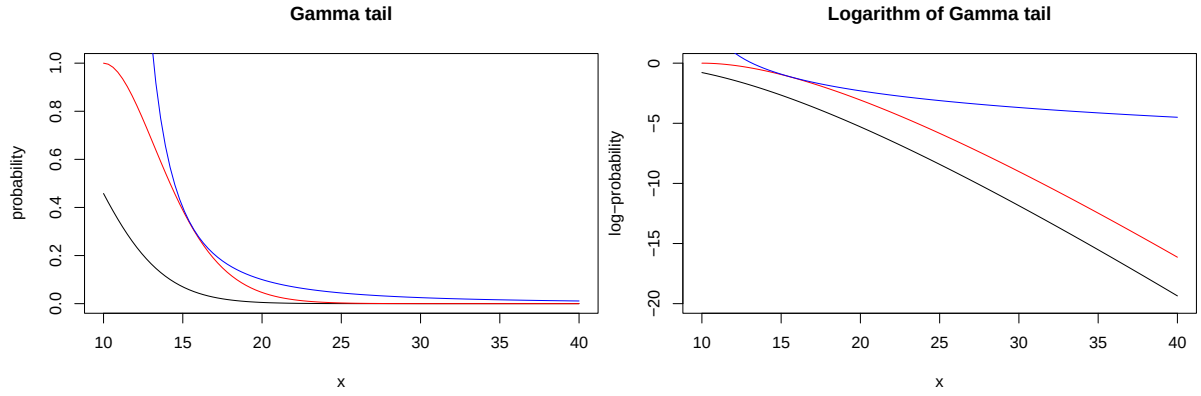


Figure 5.2: Actual tail probabilities (left) for the gamma distribution, computed via the `pgamma` function, compared it to the tight bound (red) and the weaker bound from Chebychev's inequality (blue). The differences in the tail are more clearly seen for the log-probabilities (right)

5.2 Importance sampling

When we are only interested in Monte Carlo integration, we do not need to sample from the target distribution.

Observe that

$$\mu = \int h(x) f(x) \, \mathrm{d}x = \int h(x) \frac{f(x)}{g(x)} g(x) \, \mathrm{d}x \tag{5.1}$$

$$= \int h(x) w^*(x) g(x) \, \mathrm{d}x \tag{5.2}$$

whenever g is a density fulfilling that

$$g(x) = 0 \Rightarrow f(x) = 0.$$

With $X_1, \ldots, X_n$ i.i.d. with density g define the *weights*

$$w^*(X_i) = f(X_i)/g(X_i).$$

The *importance sampling* estimator is

$$\hat{\mu}^*_{\text{IS}} := \frac{1}{n}\sum_{i=1}^{n} h(X_i)w^*(X_i).$$

It has mean μ. Again by the LLN

$$\hat{\mu}^*_{\text{IS}} \to E(h(X_1)w^*(X_1)) = \mu.$$

We will illustrate the use of importance sampling by computing the mean in the gamma distribution via simulations from a Gaussian distribution, cf. also Section 5.1.1.

```
x <- rnorm(n, 10, 3)
w_star <- dgamma(x, 8) / dnorm(x, 10, 3)
mu_hat_IS <- (cumsum(x * w_star) / (1:n))
mu_hat_IS[n]  # Theoretical value 8
```

```
## [1] 7.995228
```

To assess the precision of the importance sampling estimate via the CLT we need the variance of the average just as for plain Monte Carlo integration. By the CLT

$$\hat{\mu}^*_{\text{IS}} \overset{\text{approx}}{\sim} \mathcal{N}(\mu, \sigma^{*2}_{\text{IS}}/n)$$

where

$$\sigma^{*2}_{\text{IS}} = V(h(X_1)w^*(X_1)) = \int (h(x)w^*(x) - \mu)^2 g(x)\, \mathrm{d}x.$$

The importance sampling variance can be estimated similarly as the Monte Carlo variance

$$\hat{\sigma}^{*2}_{\text{IS}} = \frac{1}{n-1}\sum_{i=1}^{n}(h(X_i)w^*(X_i) - \hat{\mu}^*_{\text{IS}})^2,$$

and a 95% standard confidence interval is computed as

$$\hat{\mu}^*_{\text{IS}} \pm 1.96\frac{\hat{\sigma}^*_{\text{IS}}}{\sqrt{n}}.$$

```
sigma_hat_IS <- sd(x * w_star)
sigma_hat_IS  # Theoretical value ??
```

```
## [1] 3.499995
```

It may happen that $\sigma^{*2}_{\text{IS}} > \sigma^2_{\text{MC}}$ or $\sigma^{*2}_{\text{IS}} < \sigma^2_{\text{MC}}$ depending on h and g, but by choosing g cleverly so that $h(x)w^*(x)$ becomes as constant as possible, importance sampling can often reduce the variance compared to plain Monte Carlo integration.

For the mean of the gamma distribution above, σ^{*2}_{IS} is about 50% larger than σ^2_{MC}, so we loose precision by using importance sampling this way when compared to plain Monte Carlo integration. In Section 5.3 we consider a different example where we achieve a considerable variance reduction by using importance sampling.

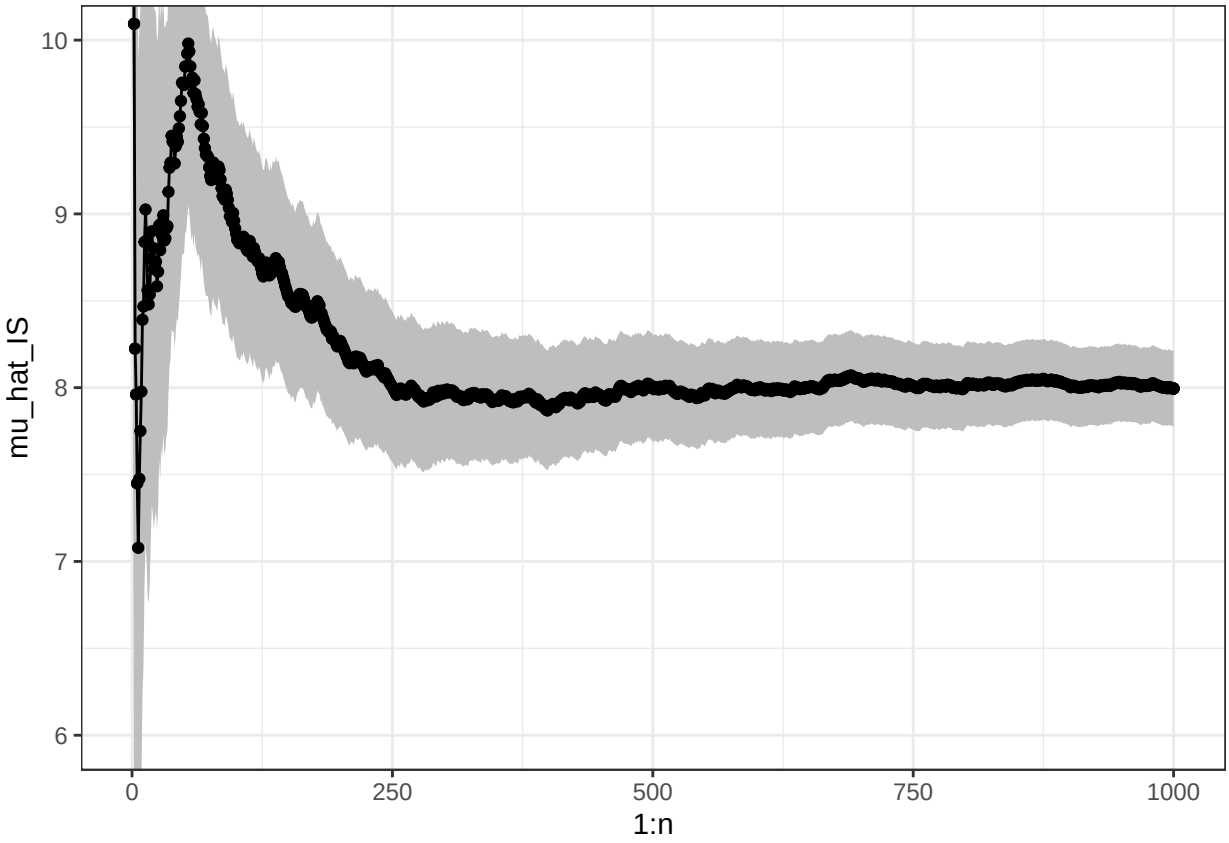


Figure 5.3: Sample path with confidence band for importance sampling Monte Carlo integration of the mean of a gamma distributed random variable via simulations from a Gaussian distribution.

5.2.1 Unknown normalization constants

If $f = c^{-1}q$ with c unknown then

$$c = \int q(x)\ \mathrm{d}x = \int \frac{q(x)}{g(x)} g(x)\ dx,$$

and

$$\mu = \frac{\int h(x)w^*(x)g(x)\ dx}{\int w^*(x)g(x)\ dx},$$

where $w^*(x) = q(x)/g(x)$.

If $X_1, \ldots, X_n$ are then i.i.d. from the distribution with density g, an importance sampling estimate of μ can be computed as

$$\hat{\mu}_{\mathrm{IS}} = \frac{\sum_{i=1}^n h(X_i)w^*(X_i)}{\sum_{i=1}^n w^*(X_i)} = \sum_{i=1}^n h(X_i)w(X_i),$$

where $w^*(X_i) = q(X_i)/g(X_i)$ and

$$w(X_i) = \frac{w^*(X_i)}{\sum_{i=1}^n w^*(X_i)}$$

are the *standardized weights*. This works irrespectively of the value of the normalizing constant c, and it actually works if also g is unnormalized.

Revisiting the mean of the gamma distribution, we can implement importance sampling via samples from a Gaussian distribution but using weights computed without the normalization constants.

```
w_star <- numeric(n)
x_pos <- x[x > 0]
w_star[x > 0] <- exp((x_pos - 10)^2 / 18 - x_pos + 7 * log(x_pos))
mu_hat_IS <- cumsum(x * w_star) / cumsum(w_star)
mu_hat_IS[n]  # Theoretical value 8
```

```
## [1] 8.102177
```

The variance of the IS estimator with standardized weights is a little more complicated, because the estimator is a ratio of random variables. From the multivariate CLT

$$\frac{1}{n}\sum_{i=1}^{n}\begin{pmatrix} h(X_i)w^*(X_i) \\ w^*(X_i) \end{pmatrix} \overset{\text{approx}}{\sim} \mathcal{N}\left(c\begin{pmatrix} \mu \\ 1 \end{pmatrix}, \frac{1}{n}\begin{pmatrix} \sigma^{*2}_{\text{IS}} & \gamma \\ \gamma & \sigma^2_{w^*} \end{pmatrix}\right),$$

where

$$\sigma^{*2}_{\text{IS}} = V(h(X_1)w^*(X_1)) \tag{5.3}$$
$$\gamma = \text{cov}(h(X_1)w^*(X_1), w^*(X_1)) \tag{5.4}$$
$$\sigma^2_{w^*} = V(w^*(X_1)). \tag{5.5}$$

We can then apply the Δ-method with $t(x, y) = x/y$. Note that $Dt(x, y) = (1/y, -x/y^2)$, whence

$$Dt(c\mu, c)\begin{pmatrix} \hat{\sigma}^{*2}_{\text{IS}} & \gamma \\ \gamma & \sigma^2_{w^*} \end{pmatrix} Dt(c\mu, c)^T = c^{-2}(\sigma^{*2}_{\text{IS}} + \mu^2\sigma^2_{w^*} - 2\mu\gamma).$$

By the Δ-method

$$\hat{\mu}_{\text{IS}} \overset{\text{approx}}{\sim} \mathcal{N}(\mu, c^{-2}(\sigma^{*2}_{\text{IS}} + \mu^2\sigma^2_{w^*} - 2\mu\gamma)/n).$$

The unknown quantities in the asymptotic variance must be estimated using e.g. their empirical equivalents, and if $c \neq 1$ (we have used unnormalized densities) it is necessary to estimate c as $\hat{c} = \frac{1}{n}\sum_{i=1}^{n} w^*(X_i)$ to compute an estimate of the variance.

For the example with the mean of the gamma distribution, we find the following estimate of the variance.

```
c_hat <- mean(w_star)
sigma_hat_IS <- sd(x * w_star)
sigma_hat_w_star <- sd(w_star)
gamma_hat <- cov(x * w_star, w_star)
sigma_hat_IS_w <- sqrt(sigma_hat_IS^2 + mu_hat_IS[n]^2 * sigma_hat_w_star^2 -
                         2 * mu_hat_IS[n] * gamma_hat) / c_hat
sigma_hat_IS_w
```

```
## [1] 3.198314
```

In this example, the variance when using standardized weights is a little lower than using unstandardized weights, but still larger than the variance for the plain Monte Carlo average.

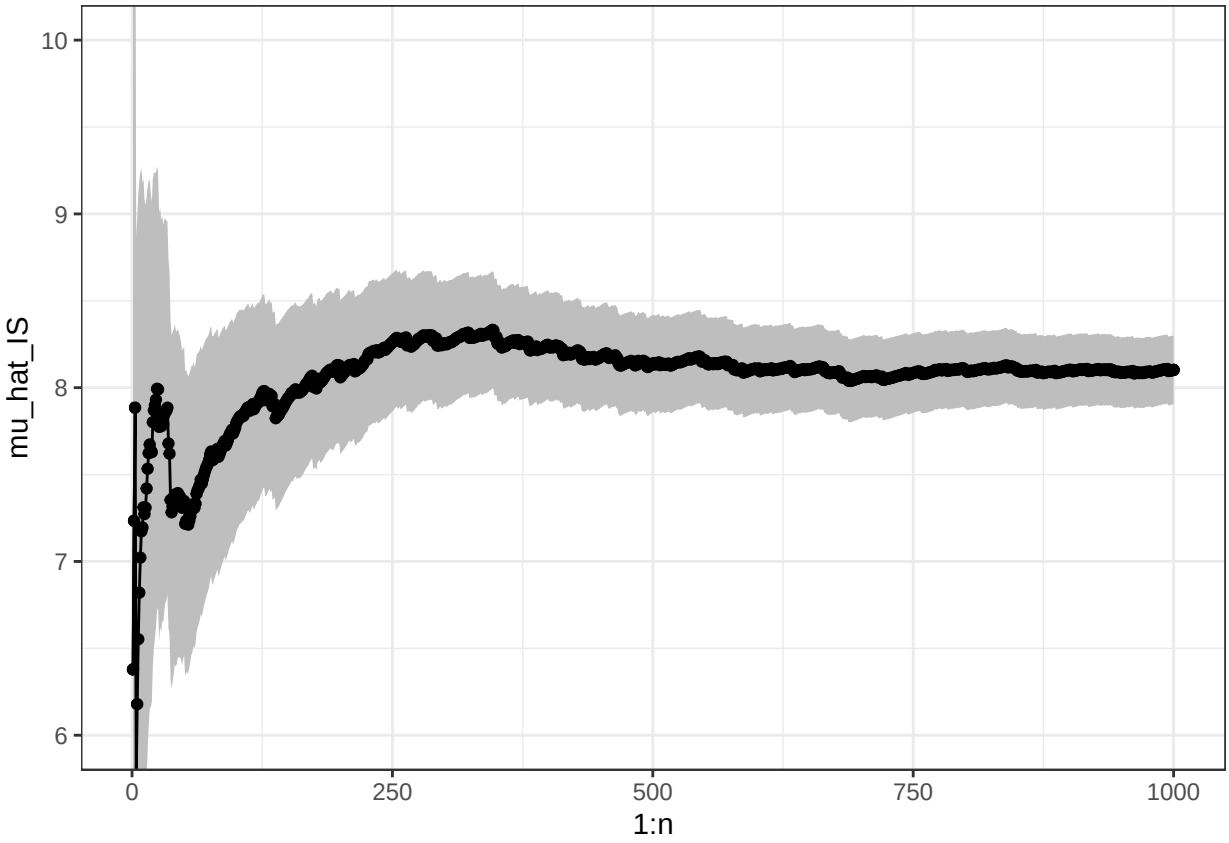


Figure 5.4: Sample path with confidence band for importance sampling Monte Carlo integration of the mean of a gamma distributed random variable via simulations from a Gaussian distribution and using standardized weights.

5.2.2 Computing a high-dimensional integral

To further illustrate the usage but also the limitations of Monte Carlo integration, consider the following p-dimensional integral

$$\int e^{-\frac{1}{2}\left(x_1^2+\sum_{i=2}^{p}(x_i-\alpha x_{i-1})^2\right)}\mathrm{d}x.$$

Now this integral is not even expressed as an expectation w.r.t. any distribution in the first place – it is an integral w.r.t. Lebesgue measure in $\mathbb{R}^p$. We use the same idea as in importance sampling to rewrite the integral as an expectation w.r.t. a probability distribution. There might be many ways to do this, and the following is just one.

Rewrite the exponent as

$$||x||_2^2 + \sum_{i=2}^{p} \alpha^2 x_{i-1}^2 - 2\alpha x_i x_{i-1}$$

so that

$$\begin{aligned}\int e^{-\frac{1}{2}\left(x_1^2+\sum_{i=2}^{p}(x_i-\alpha x_{i-1})^2\right)}\mathrm{d}x &= \int e^{-\frac{1}{2}\sum_{i=2}^{p}\alpha^2 x_{i-1}^2 - 2\alpha x_i x_{i-1}} e^{-\frac{||x||_2^2}{2}}\mathrm{d}x \\ &= (2\pi)^{p/2}\int e^{-\frac{1}{2}\sum_{i=2}^{p}\alpha^2 x_{i-1}^2 - 2\alpha x_i x_{i-1}} f(x)\mathrm{d}x\end{aligned}$$

where f is the density for the $\mathcal{N}(0, I_p)$ distribution. Thus if $X \sim \mathcal{N}(0, I_p)$,

$$\int e^{-\frac{1}{2}\left(x_1^2 + \sum_{i=2}^p (x_i - \alpha x_{i-1})^2\right)} \mathrm{d}x = (2\pi)^{p/2} E\left(e^{-\frac{1}{2}\sum_{i=2}^p \alpha^2 X_{i-1}^2 - 2\alpha X_i X_{i-1}}\right).$$

The Monte Carlo integration below computes

$$\mu = E\left(e^{-\frac{1}{2}\sum_{i=2}^p \alpha^2 X_{i-1}^2 - 2\alpha X_i X_{i-1}}\right)$$

by generating p-dimensional random variables from $\mathcal{N}(0, I_p)$. It can actually be shown that $\mu = 1$, but we skip the proof of that.

First, we implement the function we want to integrate.

```
h <- function(x, alpha = 0.1){
  p <- length(x)
  tmp <- alpha * x[1:(p - 1)]
  exp( - sum((tmp / 2 - x[2:p]) * tmp))
}
```

Then we specify various parameters.

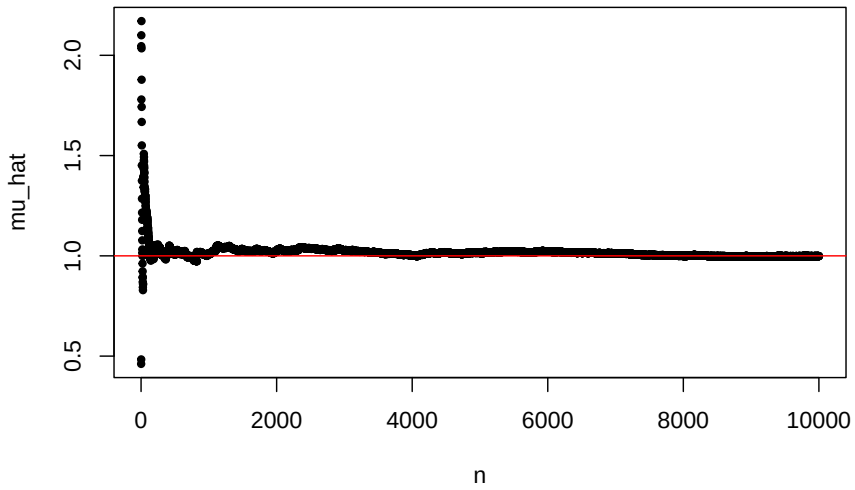
```
n <- 10000 # The number of random variables to generate
p <- 100   # The dimension of each random variable
```

The actual computation is implemented using the `apply` function. We first look at the case with $\alpha = 0.1$.

```
x <- matrix(rnorm(n * p), n, p)
evaluations <- apply(x, 1, h)
```

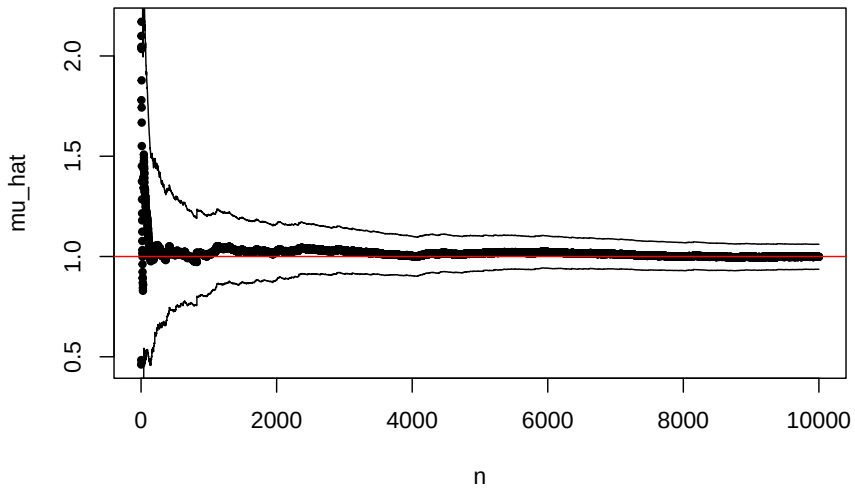
We can then plot the cumulative average and compare it to the actual value of the integral that we know is 1.

```
mu_hat <- cumsum(evaluations) / 1:n
plot(mu_hat, pch = 20, xlab = "n")
abline(h = 1, col = "red")
```



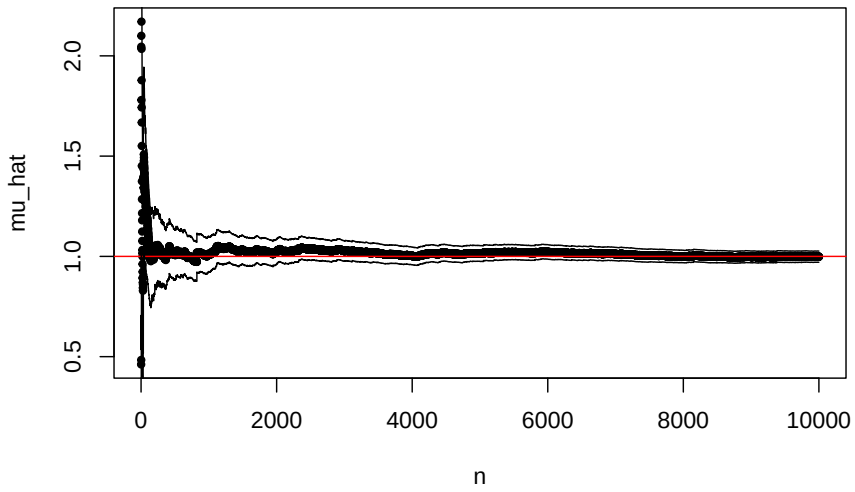
If we want to control the error with probability 0.95 we can use Chebychev's inequality and solve for ε using the estimated variance.

```
plot(mu_hat, pch = 20, xlab = "n")
abline(h = 1, col = "red")
sigma_hat <- sd(evaluations)
epsilon <- sigma_hat / sqrt((1:n) * 0.05)
lines(1:n, mu_hat + epsilon)
lines(1:n, mu_hat - epsilon)
```



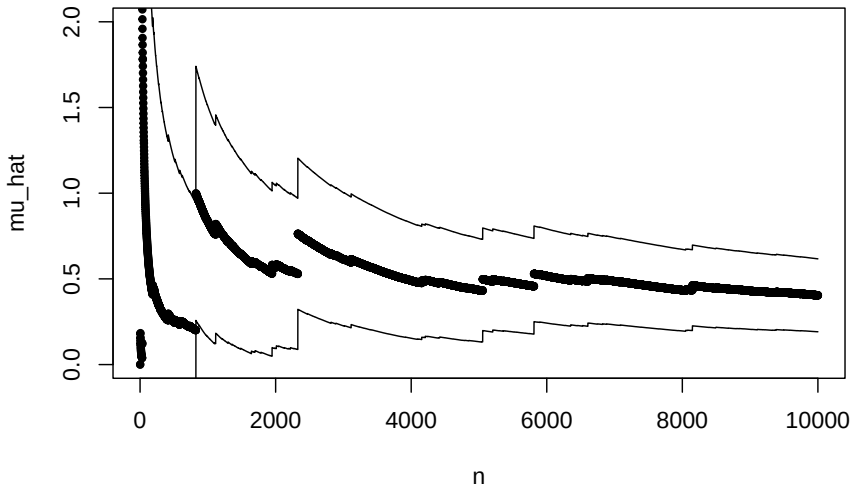
The confidence bands provided by the central limit theorem are typically more accurate estimates of the actual uncertainty than the upper bounds provided by Chebychev's inequality.

```
plot(mu_hat, pch = 20, xlab = "n")
abline(h = 1, col = "red")
lines(1:n, mu_hat + 2 * sigma_hat / sqrt(1:n))
lines(1:n, mu_hat - 2 * sigma_hat / sqrt(1:n))
```



To illustrate the limitations of Monte Carlo integration we increase α to $\alpha = 0.4$.

```
evaluations <- apply(x, 1, h, alpha = 0.4)
```



The sample path above is not carefully selected to be pathological. Due to occasional large values, the typical sample path will show occasional large jumps, and the variance may easily be grossly underestimated.

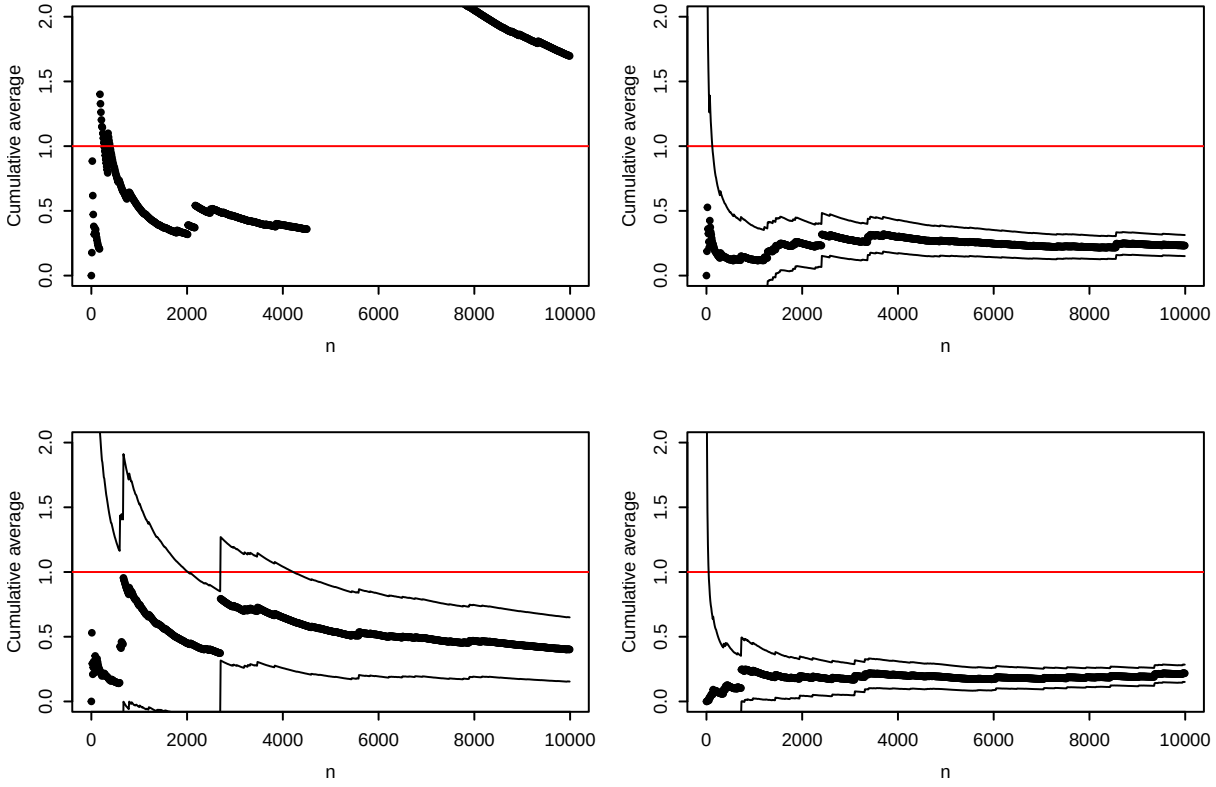


Figure 5.5: Four sample paths of the cumulative average for $\alpha = 0.4$.

To be fair, it is the choice of a standard multivariate normal distribution as the reference distribution for large α that is problematic rather than Monte Carlo integration and importance sampling as such. However, in high dimensions it can be difficult to choose a suitable distribution to sample from.

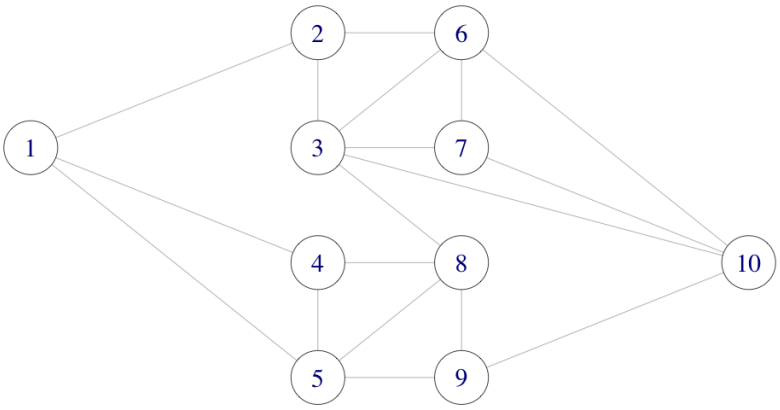
The difficulty for the specific integral is due to the exponent of the integrand, which can become large and positive if $x_i \simeq x_{i-1}$ for enough coordinates. This happens rarely for independent random variables, but large values of rare events can, nevertheless, contribute notably to the integral. The larger $\alpha \in (0, 1)$ is, the more pronounced is the problem with occasional large values of the integrand. It is possible to use importance sampling and instead sample from a

distribution where the large values are more likely. For this particular example we would need to simulate from a distribution where the variables are dependent, and we will not pursue that.

5.3 Network failure

In this section we consider a more serious application of importance sampling. Though still a toy example, where we can find an exact solution, the example illustrates well the type of application where we want to approximate a small probability using a Monte Carlo average. Importance sampling can then increase the probability of the rare event and as a result make the variance of the Monte Carlo average smaller.

We will consider the following network consisting of ten nodes and with some of the nodes connected.



The network could be a computer network with ten computers. The different connections (edges) may "fail" independently with probability p, and we ask the question: what is the probability that node 1 and node 10 are disconnected?

We can answer this question by computing an integral of an indicator function, that is, by computing the sum

$$\mu = \sum_{x \in \{0,1\}^{18}} 1_B(x) f_p(x)$$

where $f_p(x)$ is the point probability of x, with x representing which of the 18 edges in the graph that fail, and B represents the set of edges where node 1 and node 10 are disconnected. By simulating edge failures we can approximate the sum as a Monte Carlo average.

The network of nodes can be represented as a graph adjacency matrix A such that $A_{ij} = 1$ if and only if there is an edge between i and j (and $A_{ij} = 0$ otherwise).

```
A  # Graph adjacency matrix
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    1    0    1    1    0    0    0    0     0
```

```
##  [2,]    1    0    1    0    0    1    0    0    0    0
##  [3,]    0    1    0    0    0    1    1    1    0    1
##  [4,]    1    0    0    0    1    0    0    1    0    0
##  [5,]    1    0    0    1    0    0    0    1    1    0
##  [6,]    0    1    1    0    0    0    1    0    0    1
##  [7,]    0    0    1    0    0    1    0    0    0    1
##  [8,]    0    0    1    1    1    0    0    0    1    0
##  [9,]    0    0    0    0    1    0    0    1    0    1
## [10,]    0    0    1    0    0    1    1    0    1    0
```

To compute the probability that 1 and 10 are disconnected by Monte Carlo integration, we need to sample (sub)graphs by randomly removing some of the edges. This is implemented using the upper triangular part of the (symmetric) adjacency matrix.

```
sim_net <- function(Aup, p) {
  ones <- which(Aup == 1)
  Aup[ones] <- sample(
    c(0, 1),
    length(ones),
    replace = TRUE,
    prob = c(p, 1 - p)
  )
  Aup
}
```

The core of the implementation above uses the `sample()` function, which can sample with replacement from the set $\{0, 1\}$. The vector `ones` contains indices of the (upper triangular part of the) adjacency matrix containing a 1, and these positions are replaced by the sampled values before the matrix is returned.

It is fairly fast to sample even a large number of random graphs this way.

```
Aup <- A
Aup[lower.tri(Aup)] <- 0
system.time(replicate(1e5, {sim_net(Aup, 0.5); NULL}))
```

```
##    user  system elapsed
##   1.518   0.108   1.663
```

The second function we implement checks network connectivity based on the upper triangular part of the adjacency matrix. It relies on the fact that there is a path from node 1 to node 10 consisting of k edges if and only if $(A^k)_{1,10} > 0$. We see directly that such a path needs to consist of at least $k = 3$ edges. Also, we don't need to check paths with more than $k = 9$ edges as they will contain the one node multiple times and can thus be shortened.

```
discon <- function(Aup) {
  A <- Aup + t(Aup)
  i <- 3
  Apow <- A %*% A %*% A # A%^%3
  while(Apow[1, 10] == 0 & i < 9) {
    Apow <- Apow %*% A
    i <- i + 1
  }
  Apow[1, 10] == 0  # TRUE if nodes 1 and 10 not connected
}
```

We then obtain the following estimate of the probability of nodes 1 and 10 being disconnected using Monte Carlo integration.

```
seed <- 27092016
set.seed(seed)
n <- 1e5
tmp <- replicate(n, discon(sim_net(Aup, 0.05)))
mu_hat <- mean(tmp)
```

As this is a random approximation, we should report not only the Monte Carlo estimate but also the confidence interval. Since the estimate is an average of 0-1-variables, we can estimate the variance, σ^2, of the individual terms using that $\sigma^2 = \mu(1-\mu)$. We could just as well have used the empirical variance, which would give almost the same numerical value as $\hat{\mu}(1-\hat{\mu})$, but we use the latter estimator to illustrate that any (good) estimator of σ^2 can be used when estimating the asymptotic variance.

```
mu_hat + 1.96 * sqrt(mu_hat * (1 - mu_hat) / n) * c(-1, 0, 1)
```

```
## [1] 0.000226 0.000340 0.000454
```

The estimated probability is low and only in about 1 of 3000 simulated graphs will node 1 and 10 be disconnected. This suggests that importance sampling can be useful if we sample from a probability distribution with a larger probability of edge failure.

To implement importance sampling we note that the point probabilities (the density w.r.t. counting measure) for sampling the 18 independent 0-1-variables $x = (x_1, \ldots, x_{18})$ with $P(X_i = 0) = p$ is

$$f_p(x) = p^{18-s}(1-p)^s$$

where $s = \sum_{i=1}^{18} x_i$. In the implementation, weights are computed that correspond to using probability p_0 (with density $g = f_{p_0}$) instead of p, and the weights are only computed if node 1 and 10 are disconnected.

```
weights <- function(Aup, Aup0, p0, p) {
  w <- discon(Aup0)
  if (w) {
    s <- sum(Aup0)
    w <- (p / p0)^18 * (p0 * (1 - p) / (p * (1 - p0)))^s
  }
  as.numeric(w)
}
```

The implementation uses the formula

$$w(x) = \frac{f_p(x)}{f_{p_0}(x)} = \frac{p^{18-s}(1-p)^s}{p_0^{18-s}(1-p_0)^s} = \left(\frac{p}{p_0}\right)^{18} \left(\frac{p_0(1-p)}{p(1-p_0)}\right)^s.$$

The importance sampling estimate of μ is then computed.

```
set.seed(seed)
tmp <- replicate(n, weights(Aup, sim_net(Aup, 0.2), 0.2, 0.05))
mu_hat_IS <- mean(tmp)
```

And we obtain the following confidence interval using the empirical variance estimate $\hat{\sigma}^2$.

```
mu_hat_IS + 1.96 * sd(tmp) / sqrt(n) * c(-1, 0, 1)
```

```
## [1] 0.000262 0.000296 0.000330
```

The ratio of estimated variances for the plain Monte Carlo estimate and the importance sampling estimate is

```
mu_hat * (1 - mu_hat) / var(tmp)
```

```
## [1] 11.22476
```

Thus we need around 11 times more samples if using plain Monte Carlo integration when compared to importance sampling to obtain the same precision. A benchmark will show that the extra computing time for importance sampling is small compared to the reduction of variance. It is therefore worth the coding effort if used repeatedly, but not if it is a one-off computation.

The graph is, in fact, small enough for complete enumeration and thus the computation of an exact solution. There are $2^{18} = 262,144$ different networks with any number of the edges failing. To systematically walk through all possible combinations of edges failing, we use the function `intToBits` that converts an integer to its binary representation for integers from 0 to 262,143. This is a quick and convenient way of representing all the different fail and non-fail combinations for the edges.

```
ones <- which(Aup == 1)
Atmp <- Aup
p <- 0.05
prob <- numeric(2^18)
for(i in 0:(2^18 - 1)) {
  on <- as.numeric(intToBits(i)[1:18])
  Atmp[ones] <- on
  if (discon(Atmp)) {
    s <- sum(on)
    prob[i + 1] <- p^(18 - s) * (1 - p)^s
  }
}
```

The probability that nodes 1 and 10 are disconnected can then be computed as the sum of all the probabilities in `prob`.

```
sum(prob)
```

```
## [1] 0.000288295
```

This number should be compared to the estimates computed above. For a more complete comparison, we have used importance sampling with edge fail probability ranging from 0.1 to 0.4, see Figure 5.6. The results show that a failure probability of 0.2 is close to optimal in terms of giving an importance sampling estimate with minimal variance. For smaller values, the event that 1 and 10 become disconnected is too rare, and for larger values the importance weights become too variable. A choice of 0.2 strikes a good balance.

5.3.1 Object oriented implementations

Implementations of algorithms that handle graphs and simulate graphs, as in the Monte Carlo computations above, can benefit from using an object oriented approach. To this end we first

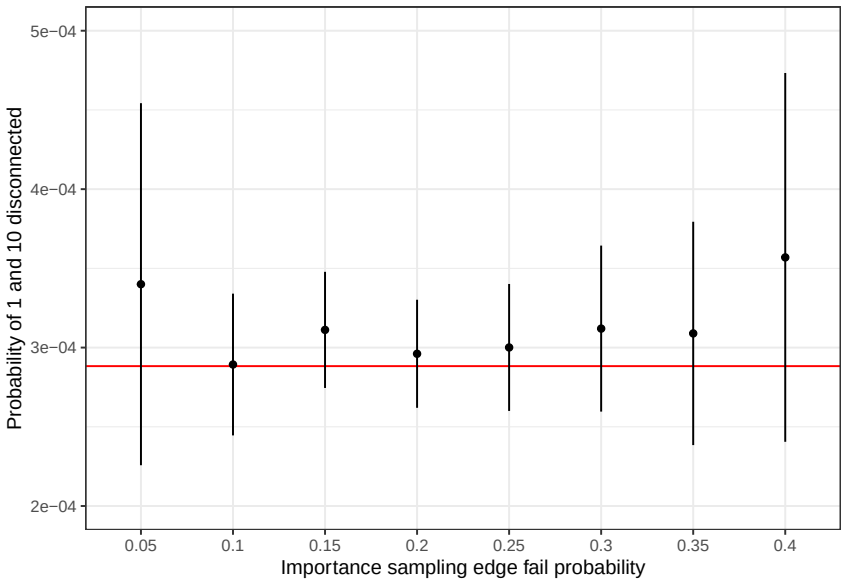


Figure 5.6: Confidence intervals for importance sampling estimates of network nodes 1 and 10 being disconnected under independent edge failures with probability 0.05. The red line is the true probability computed by complete enumeration.

implement a so-called *constructor*, which is a function that takes the adjacency matrix and the edge failure probability as arguments and returns a list with class label `network`.

```
network <- function(A, p) {
  Aup <- A
  Aup[lower.tri(Aup)] <- 0
  ones <- which((Aup == 1))
  structure(
    list(
      A = A,
      Aup = Aup,
      ones = ones,
      p = p
    ),
    class = "network"
  )
}
```

We use the constructor function `network()` to construct and object of class `network` for our specific adjacency matrix.

```
my_net <- network(A, p = 0.05)
str(my_net)
```

```
## List of 4
##  $ A   : num [1:10, 1:10] 0 1 0 1 1 0 0 0 0 0 ...
##  $ Aup : num [1:10, 1:10] 0 0 0 0 0 0 0 0 0 0 ...
##  $ ones: int [1:18] 11 22 31 41 44 52 53 63 66 73 ...
##  $ p   : num 0.05
##  - attr(*, "class")= chr "network"
```

```
class(my_net)
```

```
## [1] "network"
```

The network object contains, in addition to `A` and `p`, two precomputed components: the upper triangular part of the adjacency matrix; and the indices in that matrix containing a `1`.

The intention is then to write two methods for the network class. A method `sim()` that will simulate a graph where some edges have failed, and a method `failure()` that will estimate the probability of node `1` and `10` being disconnected by Monte Carlo integration. To do so we need to define the two corresponding generic functions.

```
sim <- function(x, ...)
  UseMethod("sim")
failure <- function(x, ...)
  UseMethod("failure")
```

The method for simulation is then implemented as a function with name `sim.network`.

```
sim.network <- function(x) {
  Aup <- x$Aup
  Aup[x$ones] <- sample(
    c(0, 1),
    length(x$ones),
    replace = TRUE,
    prob = c(x$p, 1 - x$p)
  )
  Aup
}
```

It is implemented using essentially the same implementation as `sim_net()` except that `Aup` and `p` are extracted as components from the object `x` instead of being arguments, and `ones` is extracted from `x` as well instead of being computed. One could argue that the `sim()` method should return an object of class network — that would be natural. However, then we need to call the
constructor with the full adjacency matrix, this will take some time and we do not want to do that as a default. Thus we simply return the upper triangular part of the adjacency matrix from `sim()`.

The `failure()` method implements plain Monte Carlo integration as well as importance sampling and returns a vector containing the estimate as well as the 95% confidence interval. This implementation relies on the already implemented functions `discon()` and `weights()`.

```
failure.network <- function(x, n, p0 = NULL) {
  if (is.null(p0)) {
    # Plain Monte Carlo simulation
    tmp <- replicate(n, discon(sim(x)))
    mu_hat <- mean(tmp)
    se <- sqrt(mu_hat * (1 - mu_hat) / n)
  } else {
    # Importance sampling
    p <- x$p
    x$p <- p0
    tmp <- replicate(n, weights(x$Aup, sim(x), p0, p))
    se <- sd(tmp) / sqrt(n)
```

```
    mu_hat <- mean(tmp)
  }
  value <- mu_hat + 1.96 * se * c(-1, 0, 1)
  names(value) <- c("low", "estimate", "high")
  value
}
```

We test the implementation against the previously computed results.

```
set.seed(seed)  # Resetting seed
failure(my_net, n)
```

```
##      low estimate     high
## 0.000226 0.000340 0.000454
```

```
set.seed(seed)  # Resetting seed
failure(my_net, n, p0 = 0.2)
```

```
##      low estimate     high
## 0.000262 0.000296 0.000330
```

We find that these are the same numbers as computed above, thus the object oriented implementation concurs with the non-object oriented on this example.

We benchmark the object oriented implementation to measure if there is any run time loss or improvement due to using objects. One should expect a small computational overhead due to method dispatching, that is, the procedure that R uses to look up the appropriate `sim()` method for an object of class `network`. On the other hand, `sim()` does not recompute `ones` every time.

```
microbenchmark(
  sim_net(Aup, 0.05),
  sim(my_net),
  times = 1e4
)
```

```
## Unit: microseconds
##                 expr  min    lq mean median   uq   max neval cld
##  sim_net(Aup, 0.05) 8.04  9.14 12.6   9.94 10.7 20016 10000   a
##         sim(my_net) 9.21 10.33 11.9  11.14 12.0   131 10000   a
```

From the benchmark, the object oriented solution using `sim()` appears to be a bit slower than `sim_net()` despite the latter recomputing `ones`, and this can be explained by method dispatching taking of the order of 1 microsecond during these benchmark computations.

Once we have taken an object oriented approach, we can also implement methods for some standard generic functions, e.g. the `print` function. As this generic function already exists, we simply need to implement a method for class `network`.

```
print.network <- function(x) {
  cat("#vertices: ", nrow(x$A), "\n")
  cat("#edges:", sum(x$Aup), "\n")
  cat("p = ", x$p, "\n")
}
```

```
my_net  # Implicitly calls 'print'
```

```
## #vertices:  10
## #edges: 18
## p =  0.05
```

Our print method now prints out some summary information about the graph instead of just the raw list.

If you want to work more seriously with graphs, it is likely that you want to use an existing R package instead of reimplementing many graph algorithms. One of these packages is igraph, which also illustrates well an object oriented implementation of graph classes in R.

We start by constructing a new graph object from the adjacency matrix.

```
net <- graph_from_adjacency_matrix(A, mode = "undirected")
class(net)
```

```
## [1] "igraph"
```

```
net # Illustrates the print method for objects of class 'igraph'
```

```
## IGRAPH fd0ee3b U--- 10 18 --
## + edges from fd0ee3b:
##  [1] 1-- 2 1-- 4 1-- 5 2-- 3 2-- 6 3-- 6 3-- 7 3-- 8 3--10 4-- 5 4-- 8 5-- 8
## [13] 5-- 9 6-- 7 6--10 7--10 8-- 9 9--10
```

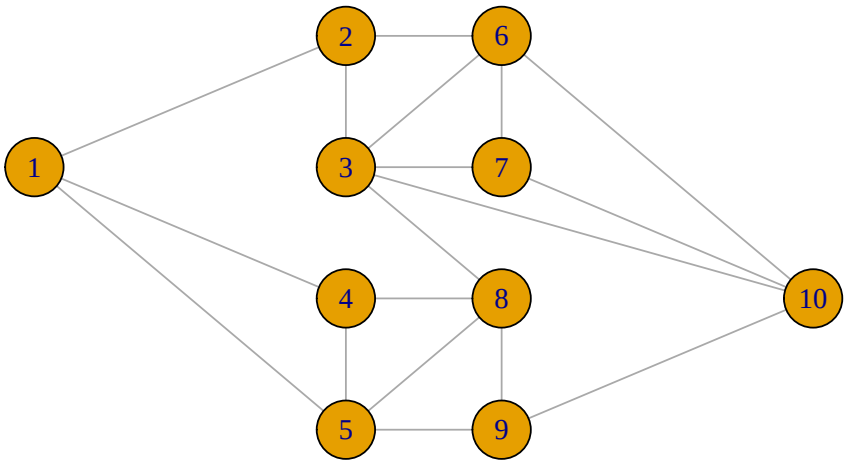
The igraph package supports a vast number of graph computation, manipulation and visualization tools. We will here illustrate how igraph can be used to plot the graph and how we can implement a simulation method for objects of class igraph.

You can use `plot(net)`, which will call the plot method for objects of class igraph. But before doing so, we will specify a layout of the graph.

```
# You can generate a layout ...
net_layout <- layout_(net, nicely())
# ... or you can specify one yourself
net_layout <- matrix(
  c(-20,  1,
    -4,  3,
    -4,  1,
    -4, -1,
    -4, -3,
     4,  3,
     4,  1,
     4, -1,
     4, -3,
    20,  -1),
  ncol = 2, nrow = 10, byrow = TRUE)
```

The layout we have specified makes it easy to recognize the graph.

```
plot(net, layout = net_layout, asp = 0)
```



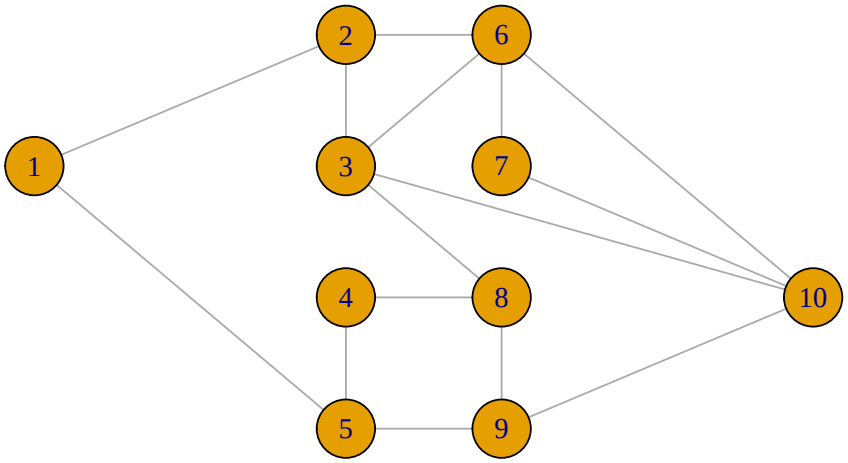
We then use two functions from the igraph package to implement simulation of a new graph. We need `gsize()`, which gives the number of edges in the graph, and we need `delete_edges()`, which removes edges from the graph. Otherwise the simulation is still based on `sample()`.

```
sim.igraph <- function(x, p) {
  deledges <- sample(
    c(TRUE, FALSE),
    gsize(x),         # 'gsize()' returns the number of edges
    replace = TRUE,
    prob = c(p, 1 - p)
  )
  delete_edges(x, which(deledges))
}
```

Note that this method is also called `sim()`, yet there is no conflict here with the method for objects of class network because the generic `sim()` function will delegate the call to the correct method based on the objects class label.

If we combine our new `sim()` method for igraphs with the plot method, we can plot a simulated graph.

```
plot(sim(net, 0.25), layout = net_layout, asp = 0)
```



The implementation using igraph turns out to be a little slower than using the matrix representation alone as in `sim_net()`.

```
system.time(replicate(1e5, {sim(net, 0.05); NULL}))
```

```
##    user  system elapsed
##   5.569   6.556  13.706
```

One could also implement the function for testing if nodes 1 and 10 are disconnected using the `shortest_paths()` function, but this is not faster than the simple matrix multiplications used in `discon()` either. However, one should be careful not to draw any general conclusions from this. Our graph is admittedly a small toy example, and we implemented our solutions largely to handle this particular graph.

Part III: Optimization

Chapter 6

Four Examples

This chapter treats four examples of non-trivial statistical models in some detail. These are all parametric models, and a central computational challenge is to fit the models to data via (penalized) likelihood maximization. The actual optimization algorithms and implementations are the topics of Chapters 7 and 8. The focus of this chapter is on the structure of the statistical models themselves to provide the necessary background for the later chapters.

Statistical models come in all forms and shapes, and it is possible to take a very general and abstract mathematical approach; statistical models are parametrized families of probability distributions. To say anything of interest, we need more structure such as structure on the parameter set, properties of the parametrized distributions, and properties of the mapping from the parameter set to the distributions. For any specific model we have ample of structure but often also an overwhelming amount of irrelevant details that will be more distracting than clarifying. The intention is that the four examples treated will illustrate the breath of statistical models that share important structures without getting lost in a wasteland of abstractions.

If one should emphasize a single abstract idea that is of theoretical value as well as of practical importance, it is the idea of *exponential families*. Statistical models that are exponential families have so much structure that the general theory provides a number of results and details of practical value for individual models. Exponential families are exemplary statistical models, that are widely used as models of data, or as central building blocks of more complicated models of data. For this reason, the treatment of the examples is preceded by a treatment of exponential families.

6.1 Exponential families

This section introduces exponential families in a concise way. The crucial observation is that the log-likelihood is concave, and that we can derive general formulas for derivatives. This will be important for the optimization algorithms developed later for computing maximum-likelihood estimates and for answering standard asymptotic inference questions.

The exponential families are extremely well behaved from a mathematical as well as a computational viewpoint, but they may be inadequate for modeling data in some cases. A typical practical problem is that there is heterogeneous variation in data beyond what can be captured by any single exponential family. A fairly common technique is then to build an exponential family model of the observed variables *as well as some latent variables*. The latent variables then serve the purpose of modeling the heterogeneity. The resulting model of the observed variables is consequently the marginalization of an exponential family, which is generally not

an exponential family and in many ways less well behaved. It is nevertheless possible to exploit the exponential family structure underlying the marginalized model for many computations of statistical importance. The EM-algorithm as treated in Chapter 8 is one particularly good example, but Bayesian computations can in similar ways exploit the structure.

6.1.1 Full exponential families

In this section we consider statistical models on an abstract product sample space

$$\mathcal{Y} = \mathcal{Y}_1 \times \ldots \times \mathcal{Y}_m.$$

We will be interested in models of observations $y_1 \in \mathcal{Y}_1, \ldots, y_m \in \mathcal{Y}_m$ that are independent but not necessarily identically distributed.

An exponential family is defined in terms of two ingredients:

- maps $t_j : \mathcal{Y}_j \to \mathbb{R}^p$ for $j = 1, \ldots, m$,
- and non-trivial σ-finite measures ν_j on $\mathcal{Y}_j$ for $j = 1, \ldots, m$.

The maps t_j are called *sufficient statistics*, and in terms of these and the *base measures* ν_j we define

$$\varphi_j(\theta) = \int e^{\theta^T t_j(u)} \nu_j(\mathrm{d}u).$$

These functions are well defined as functions

$$\varphi_j : \mathbb{R}^p \to (0, \infty].$$

We define

$$\Theta_j = \mathrm{int}(\{\theta \in \mathbb{R}^p \mid \varphi_j(\theta) < \infty\}),$$

which by definition is an open set as. It can be shown that Θ_j is convex and that φ_j is a log-convex function. Defining

$$\Theta = \bigcap_{j=1}^m \Theta_j,$$

then Θ is likewise open and convex, and we define the *exponential family* as the distributions parametrized by $\theta \in \Theta$ that have densities

$$f(\mathbf{y} \mid \theta) = \prod_{j=1}^m \frac{1}{\varphi_j(\theta)} e^{\theta^T t_j(y_j)} = e^{\theta^T \sum_{j=1}^m t_j(y_j) - \sum_{j=1}^m \log \varphi_j(\theta)}, \quad \mathbf{y} \in \mathcal{Y}, \tag{6.1}$$

w.r.t. $\otimes_{j=1}^m \nu_j$. The case where $\Theta = \emptyset$ is of no interest, and we will thus assume that the parameter set Θ is non-empty. The parameter θ is called the *canonical parameter* and Θ is the canonical parameter space. We may also say that the exponential family is canonically parametrized by θ. It is important to realize that an exponential family may come with a non-canonical parametrization that doesn't reveal right away that it is an exponential family. Thus a bit of work is then needed to show that the parametrized family of distributions can, indeed, be reparametrized as an exponential family. In the non-canonical parametrization, the family is then an example of a *curved exponential family* as defined below.

Example 6.1. The von Mises distributions on $\mathcal{Y} = (-\pi, \pi]$ form an exponential family with $m = 1$. The sufficient statistic $t_1 : (-\pi, \pi] \mapsto \mathbb{R}^2$ is

$$t_1(y) = \begin{pmatrix} \cos(y) \\ \sin(y) \end{pmatrix},$$

and

$$\varphi(\theta) = \int_{-\pi}^{\pi} e^{\theta_1 \cos(u) + \theta_2 \sin(u)} \mathrm{d}u < \infty$$

for all $\theta = (\theta_1, \theta_2)^T \in \mathbb{R}^2$. Thus the canonical parameter space is $\Theta = \mathbb{R}^2$.

As mentioned in Section 1.2.1, the function $\varphi(\theta)$ can be expressed in terms of a modified Bessel function, but it doesn't have an expression in terms of elementary functions. Likewise in Section 1.2.1, an alternative parametrization (polar coordinates) was given;

$$(\kappa, \mu) \mapsto \theta = \kappa \begin{pmatrix} \cos(\mu) \\ \sin(\mu) \end{pmatrix}$$

that maps $[0, \infty) \times (-\pi, \pi]$ onto Θ. The von Mises distributions form a curved exponential family in the (κ, μ)-parametrization, but this parametrization has several problems. First, the μ parameter is not identifiable if $\kappa = 0$, which is reflected by the fact that the reparametrization is not a one-to-one map. Second, the parameter space is not open, which can be quite a nuisance for e.g. maxmimum-likelihood estimation. We could circumvent these problems by restricting attention to $(\kappa, \mu) \in (0, \infty) \times (-\pi, \pi)$, but we would then miss some of the distributions in the exponential family – notably the uniform distribution corresponding to $\theta = 0$. In conclusion, the canonical parametrization of the family of distributions as an exponential family is preferable for mathematical and computational reasons.

Example 6.2. The family of Gaussian distributions on $\mathbb{R}$ is an example of an exponential family as defined above with $m = 1$ and $\mathcal{Y} = \mathbb{R}$. The density of the $\mathcal{N}(\mu, \sigma^2)$ distribution is

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right) = \frac{1}{\sqrt{\pi}} \exp\left(\frac{\mu}{\sigma^2} y - \frac{1}{2\sigma^2} y^2 - \frac{\mu^2}{2\sigma^2} - \frac{1}{2} \log(2\sigma^2)\right).$$

Letting the base measure ν_1 be Lebesgue measure scaled by $1/\sqrt{\pi}$, and $t_1 : \mathbb{R} \mapsto \mathbb{R}^2$ be

$$t_1(y) = \begin{pmatrix} y \\ -y^2 \end{pmatrix}$$

we identify this family of distributions as an exponential family with canonical parameter

$$\theta = \begin{pmatrix} \frac{\mu}{\sigma^2} \\ \frac{1}{2\sigma^2} \end{pmatrix}.$$

We can express the mean and variance in terms of θ as

$$\sigma^2 = \frac{1}{2\theta_2} \quad \text{and} \quad \mu = \frac{\theta_1}{2\theta_2},$$

and we find that

$$\log \varphi_1(\theta) = \frac{\mu^2}{2\sigma^2} + \frac{1}{2} \log(2\sigma^2) = \frac{\theta_1^2}{4\theta_2} - \frac{1}{2} \log \theta_2.$$

We note that the reparametrization $(\mu, \sigma^2) \mapsto \theta$ maps $\mathbb{R} \times (0, \infty)$ bijectively onto the open set $\mathbb{R} \times (0, \infty)$, and that the formula above for $\log \varphi_1(\theta)$ only holds on this set. It is natural to ask if the canonical parameter space is actually larger for this exponential family. That is, is $\varphi_1(\theta) < \infty$ for $\theta_2 \leq 0$? To this end observe that if $\theta_2 \leq 0$

$$\varphi_1(\theta) = \frac{1}{\sqrt{2\pi}} \int e^{\theta_1 u - \theta_2 \frac{u^2}{2}} \mathrm{d}u \geq \frac{1}{\sqrt{2\pi}} \int e^{\theta_1 u} \mathrm{d}u = \infty,$$

and we conclude that

$$\Theta = \mathbb{R} \times (0, \infty).$$

The family of Gaussian distributions is an example of a family of distributions whose commonly used parametrization in terms of mean and variance differs from the canonical parametrization as an exponential family. The mean and variance are easy to interpret, but in terms of mean and variance, the Gaussian distributions form a curved exponential family. For mathematical and computational purposes the canonical parametrization is preferable.

For general exponential families it may seem restrictive that all the sufficient statistics, t_j, take values in the same p-dimensional space, and that all marginal distributions share a common parameter vector θ. This is, however, not a restriction. Say we have two distributions with sufficient statistics $\tilde{t}_1 : \mathcal{Y}_1 \to \mathbb{R}^{p_1}$ and $\tilde{t}_2 : \mathcal{Y}_2 \to \mathbb{R}^{p_2}$ and corresponding parameters θ_1 and θ_2, then we construct

$$t_1(y_1) = \begin{pmatrix} \tilde{t}_1(y_1) \\ 0 \end{pmatrix} \quad \text{and} \quad t_2(y_2) = \begin{pmatrix} 0 \\ \tilde{t}_2(y_2) \end{pmatrix}.$$

Now t_1 and t_2 both map into $\mathbb{R}^p$ with $p = p_1 + p_2$, and we can bundle the parameters together into the vector

$$\theta = \begin{pmatrix} \theta_1 \\ \theta_2 \end{pmatrix} \in \mathbb{R}^p.$$

Clearly, this construction can be generalized to any number of distributions and allows us to always assume a common parameter space. The sufficient statistics then take care of selecting which of the coordinates in the parameter vector that is used for any particular marginal distribution.

6.1.2 Exponential family Bayesian networks

An exponential family is defined above as a parametrized family of distributions on $\mathcal{Y}$ of *independent* variables. That is, the joint density in (6.1) factorizes w.r.t. a product measure. Without really changing the notation this assumption can be relaxed considerably.

If the sufficient statistic t_j, instead of being a fixed map, is allowed to depend on $y_1, \ldots, y_{j-1}$, $f(\mathbf{y} \mid \theta)$ as defined in (6.1) is still a joint density. The only difference is that the factor

$$f_j(y_j \mid y_1, \ldots, y_{j-1}, \theta) = e^{\theta^T t_j(y_j) - \log \varphi_j(\theta)}$$

is now the *conditional* density of y_j given $y_1, \ldots, y_{j-1}$. In the notation we let the dependence of t_j on $y_1, \ldots, y_{j-1}$ be implicit as this will not affect the abstract theory. In any concrete example though, it will be clear how t_j actually depends upon all, some or none of these variables. Note that φ_j may now also depend upon the data through $y_1, \ldots, y_{j-1}$.

The observation above is powerful. It allows us to develop a unified approach based on exponential families for a majority of all statistical models that are applied in practice. The models we consider make two essential assumptions

1. the variables that we model can be ordered such that y_j only depends on $y_1, \ldots, y_{j-1}$ for $j = 1, \ldots, m$,
2. all the conditional distributions form exponential families themselves, with the conditioning variables entering through the t_j-maps.

The first of these assumptions is equivalent to the joint distribution being a Bayesian network, that is, a distribution whose density factorizes according to a directed acyclic graph. This includes time series models and hierarchical models. The second assumption is more restrictive,

but is a common practice in applied work. Moreover, as many standard statistical models of univariate discrete and continuous variables are, in fact, exponential families, building up a joint distribution as a Bayesian network via conditional binomial, Poisson, beta, Gamma and Gaussian distributions among others is a rather flexible framework, and yet it will result in a model with a density that factorizes as in (6.1).

Bayesian networks is a large and interesting subject in itself, and it is unfair to gloss over all the details. One of the main points is that for many computations it is possible to develop efficient algorithms by exploiting the graph structure. The seminal paper by Lauritzen and Spiegelhalter (1988) demonstrated this for the computation of conditional distributions. The mere fact that the variables can be ordered in a way that aligns with how the variables depend on each other is useful, but there can be many ways to do this, and just specifying an ordering ignores important details of the graph. It is, however, beyond the scope of this book to get into the graph algorithms required for a thorough general treatment of Bayesian networks.

6.1.3 Likelihood computations

To simplify the notation we introduce

$$t(\mathbf{y}) = \sum_{j=1}^{m} t_j(y_j),$$

which we refer to as the sufficient statistic, and

$$\kappa(\theta) = \sum_{j=1}^{m} \log \varphi_j(\theta),$$

which is a convex C^∞-function on Θ.

Having observed $\mathbf{y} \in \mathcal{Y}$ it is evident that the log-likelihood for the exponential family specified by (6.1) is

$$\ell(\theta) = \log f(\mathbf{y} \mid \theta) = \theta^T t(\mathbf{y}) - \kappa(\theta).$$

From this it follows that the gradient of the log-likelihood is

$$\nabla \ell(\theta) = t(\mathbf{y}) - \nabla \kappa(\theta)$$

and the Hessian is

$$D^2 \ell(\theta) = -D^2 \kappa(\theta),$$

which is always negative semidefinite. The maximum-likelihood estimator exists if and only if there is a solution to the *score equation*

$$t(\mathbf{y}) = \nabla \kappa(\theta),$$

and it is unique if there is such a solution, $\hat{\theta}$, for which $I(\hat{\theta}) = D^2 \kappa(\hat{\theta})$ is positive definite. We call $I(\hat{\theta})$ the *observed Fisher information*.

Note that under the independence assumption,

$$\nabla \log \varphi_j(\theta) = \frac{1}{\varphi_j(\theta)} \int t_j(u) e^{\theta^T t_j(u)} \nu_i(\mathrm{d}u) = E_\theta(t_j(Y)),$$

which means that the score equation can be expressed as

$$t(\mathbf{y}) = \sum_{j=1}^{m} E_\theta(t_j(Y)).$$

In the Bayesian network setup $\nabla \log \varphi_j(\theta) = E_\theta(t_j(Y) \mid Y_1, \ldots, Y_{j-1})$, and the score equation is

$$t(\mathbf{y}) = \sum_{j=1}^{m} E_\theta(t_j(Y) \mid y_1, \ldots, y_{j-1}),$$

which is a bit more complicated as the right-hand-side depends on the observations.

The definition of an exponential family in Section 6.1 encompasses the situation where $y_1, \ldots, y_m$ are i.i.d. by taking t_j to be independent of j. In that case, $\kappa(\theta) = m\kappa_1(\theta)$, the score equation can be rewritten as

$$\frac{1}{m} \sum_{j=1}^{m} t_1(y_j) = \kappa_1(\theta),$$

and the Fisher information becomes

$$I(\hat{\theta}) = mD^2\kappa_1(\hat{\theta}).$$

However, the point of the general formulation is that it includes regression models, and, via the Bayesian networks extension above, models with dependence structures. We could, of course, then have i.i.d. replications $\mathbf{y}_1, \ldots, \mathbf{y}_n$ of the entire m-dimensional vector $\mathbf{y}$, and we would get the score equation

$$\frac{1}{n} \sum_{i=1}^{n} t(\mathbf{y}_i) = \kappa(\theta),$$

and corresponding Fisher information

$$I(\hat{\theta}) = nD^2\kappa(\hat{\theta}).$$

6.1.4 Curved exponential families

A *curved exponential family* consists of an exponential family together with a C^2-map $\theta : \Psi \to \Theta$ from a set $\Psi \subseteq \mathbb{R}^q$. The set Ψ provides a parametrization of the subset $\theta(\Psi) \subseteq \Theta$ of the full exponential family, and the log-likelihood in the ψ-parameter is

$$\ell(\psi) = \theta(\psi)^T t(\mathbf{y}) - \kappa(\theta(\psi)).$$

It has gradient

$$\nabla \ell(\psi) = D\theta(\psi)^T t(\mathbf{y}) - \nabla\kappa(\theta(\psi))D\theta(\psi)$$

and Hessian

$$D^2\ell(\psi) = \sum_{k=1}^{p} D^2\theta_k(\psi)(t(\mathbf{y})_k - \partial_k\kappa(\theta(\psi))) - D\theta(\psi)^T D^2\kappa(\theta(\psi))D\theta(\psi).$$

6.2 Multinomial models

The multinomial model is the model of all probability distributions on a finite set $\mathcal{Y} = \{1, \ldots, K\}$. The model is parametrized by the simplex

$$\Delta_K = \left\{ (p_1, \ldots, p_K)^T \in \mathbb{R}^K \mid p_k \geq 0, \sum_{k=1}^{K} p_k = 1 \right\}.$$

The distributions parametrized by the relative interior of Δ_K form an exponential family by the parametrization

$$p_k = \frac{e^{\theta_k}}{\sum_{l=1}^{K} e^{\theta_l}} \in (0, 1)$$

for $(\theta_1, \ldots, \theta_K)^T \in \mathbb{R}^K$. That is, the sufficient statistic is $k \mapsto (\delta_{1k}, \ldots, \delta_{Kk})^T \in \mathbb{R}^{K-1}$ (where $\delta_{lk} \in \{0, 1\}$ is the Kronecker delta being 1 if and only if $l = k$), and

$$\varphi(\theta_1, \ldots, \theta_K) = \sum_{l=1}^{K} e^{\theta_l}.$$

We call this exponential family parametrization the *symmetric* parametrization. The canonical parameter in the symmetric parametrization is not identifiable, and to resolve the lack of identifiability there is a tradition of fixing the last parameter as $\theta_K = 0$. This results in a canonical parameter $\theta = (\theta_1, \ldots, \theta_{K-1})^T \in \mathbb{R}^{K-1}$, a sufficient statistic $t_1(k) = (\delta_{1k}, \ldots, \delta_{(K-1)k})^T \in \mathbb{R}^p$,

$$\varphi(\theta) = 1 + \sum_{l=1}^{K-1} e^{\theta_l}$$

and probabilities

$$p_k = \begin{cases} \frac{e^{\theta_k}}{1+\sum_{l=1}^{K-1} e^{\theta_l}} & \text{if } k = 1, \ldots, K-1 \\ \frac{1}{1+\sum_{l=1}^{K-1} e^{\theta_l}} & \text{if } k = K. \end{cases}$$

We see that in this parametrization $p_k = e^{\theta_k} p_K$ for $k = 1, \ldots, K-1$, where the probability of the last element K acts as a baseline or reference probability, and the factors e^{θ_k} act as multiplicative modifications of this baseline. Moreover,

$$\frac{p_k}{p_l} = e^{\theta_k - \theta_l},$$

which is independent of the chosen baseline.

In the special case of $K = 2$ the two elements $\mathcal{Y}$ are often given other labels than 1 and 2. The most common labels are $\{0, 1\}$ and $\{-1, 1\}$. If we use the 0-1-labels the convention is to use p_0 as the baseline and thus

$$p_1 = \frac{e^\theta}{1 + e^\theta} = e^\theta p_0 = e^\theta (1 - p_1).$$

parametrized by $\theta \in \mathbb{R}$. As this function of θ is known as the *logistic function*, this parametrization of the probability distributions on $\{0, 1\}$ is often referred to as the logistic model. From the above we see directly that

$$\theta = \log \frac{p_1}{1 - p_1}$$

is the log-odds.

If we use the ± 1-labels, an alternative exponential family parametrization is

$$p_k = \frac{e^{k\theta}}{e^\theta + e^{-\theta}}$$

for $\theta \in \mathbb{R}$ and $k \in \{-1, 1\}$. This gives a symmetric treatment of the two labels while retaining the identifiability.

With i.i.d. observations from a multinomial distribution we find that the log-likelihood is

$$\begin{aligned}\ell(\theta) &= \sum_{i=1}^{n}\sum_{k=1}^{K}\delta_{ky_i}\log(p_k(\theta)) \\ &= \sum_{k=1}^{K}\underbrace{\sum_{i=1}^{n}\delta_{ky_i}}_{=n_k}\log(p_k(\theta)) \\ &= \theta^T\mathbf{n} - n\log\left(1+\sum_{l=1}^{K-1}e^{\theta_l}\right).\end{aligned}$$

where $\mathbf{n} = (n_1, \ldots, n_K)^T = \sum_{i=1}^{n} t(y_i)$ is the sufficient statistic, which is simply a tabulation of the times the different elements in $\{1, \ldots, K\}$ were observed.

6.2.1 Peppered Moths

This example is on the color of the peppered Moth (*Birkemåler* in Danish). The color of the moth is primarily determined by one gene that occur in three different alleles denoted C, I and T. The alleles are ordered in terms of dominance as C > I > T. Moths with genotype including C are dark. Moths with genotype TT are light colored. Moths with genotypes II and IT are mottled. Thus there a total of six different genotypes (CC, CI, CT, II, IT and TT) and three different phenotypes (black, mottled, light colored).

The peppered moth provided an early demonstration of evolution in the 19th century England, where the light colored moth was outnumbered by the dark colored variety. The dark color became dominant due to the increased pollution, where trees were darkened by soot, and had for that reason a selective advantage. There is a nice collection of moth in different colors at the Danish Zoological Museum, and further explanation of the role it played in understanding evolution.

We denote the allele frequencies p_C, p_I, p_T with $p_C + p_I + p_T = 1$. According to the Hardy-Weinberg principle, the genotype frequencies are then

$$p_C^2, 2p_Cp_I, 2p_Cp_T, p_I^2, 2p_Ip_T, p_T^2.$$

If we could observe the genotypes, the complete multinomial log-likelihood would be

$$\begin{aligned}
&2n_{CC}\log(p_C) + n_{CI}\log(2p_Cp_I) + n_{CT}\log(2p_Cp_I) \\
&\quad + 2n_{II}\log(p_I) + n_{IT}\log(2p_Ip_T) + 2n_{TT}\log(p_T) \\
&= 2n_{CC}\log(p_C) + n_{CI}\log(2p_Cp_I) + n_{CT}\log(2p_Cp_I) \\
&\quad + 2n_{II}\log(p_I) + n_{IT}\log(2p_I(1-p_C-p_I)) + 2n_{TT}\log(1-p_C-p_I).
\end{aligned}$$

The log-likelihood is given in terms of the genotype counts and the two probability parameters $p_C, p_I \geq 0$ with $p_C + p_I \leq 1$, and on the interior of this parameter set the model is a curved exponential family.

Collecting moths and determining their color will, however, only identify their phenotype, not their genotype. Thus we observe (n_C, n_T, n_I), where

$$n = \underbrace{n_{CC} + n_{CI} + n_{CT}}_{=n_C} + \underbrace{n_{IT} + n_{II}}_{=n_I} + \underbrace{n_{TT}}_{=n_T}.$$

For maximum-likelihood estimation of the parameters in this model from the observation (n_C, n_T, n_I), we need the likelihood, that is, the likelihood in the marginal distribution of the observed variables.

The peppered Moth example is an example of *cell collapsing* in a multinomial model. In general, let $A_1 \cup \ldots \cup A_{K_0} = \{1, \ldots, K\}$ be a partition and let

$$M : \mathbb{N}_0^K \to \mathbb{N}_0^{K_0}$$

be the map given by

$$M((n_1, \ldots, n_K))_j = \sum_{k \in A_j} n_k.$$

If $Y \sim \text{Mult}(p, n)$ with $p = (p_1, \ldots, p_K)$ then

$$X = M(Y) \sim \text{Mult}(M(p), n).$$

If $\theta \mapsto p(\theta)$ is a parametrization of the cell probabilities the log-likelihood under the collapsed multinomial model is generally

$$\ell(\theta) = \sum_{j=1}^{K_0} x_j \log(M(p(\theta))_j) = \sum_{j=1}^{K_0} x_j \log\left(\sum_{k \in A_j} p_k(\theta)\right). \tag{6.2}$$

For the peppered Moths, $K = 6$ corresponding to the six genotypes, $K_0 = 3$ and the partition corresponding to the phenotypes is

$$\{1, 2, 3\} \cup \{4, 5\} \cup \{6\} = \{1, \ldots, 6\},$$

and

$$M(n_1, \ldots, n_6) = (n_1 + n_2 + n_3, n_4 + n_5, n_6).$$

In terms of the (p_C, p_I) parametrization, $p_T = 1 - p_C - p_I$ and

$$p = (p_C^2, 2p_Cp_I, 2p_Cp_T, p_I^2, 2p_Ip_T, p_T^2).$$

Hence

$$M(p) = (p_C^2 + 2p_Cp_I + 2p_Cp_T, p_I^2 + 2p_Ip_T, p_T^2).$$

The log-likelihood is

$$\ell(p_C, p_I) = n_C \log(p_C^2 + 2p_Cp_I + 2p_Cp_T) + n_I \log(p_I^2 + 2p_Ip_T) + n_T \log(p_T^2). \tag{6.3}$$

We can implement the log-likelihood in a very problem specific way. Note how the parameter constraints are encoded via the return value ∞.

```
## par = c(pC, pI), pT = 1 - pC - pI
## x is the data vector of length 3 of counts
loglik <- function(par, x) {
  pT <- 1 - par[1] - par[2]

  if (par[1] > 1 || par[1] < 0 || par[2] > 1
      || par[2] < 0 || pT < 0)
    return(Inf)

  PC <- par[1]^2 + 2 * par[1] * par[2] + 2 * par[1] * pT
  PI <- par[2]^2 + 2 * par[2] * pT
  PT <- pT^2
  ## The function returns the negative log-likelihood
  - (x[1] * log(PC) + x[2] * log(PI) + x[3]* log(PT))
}
```

It is possible to use `optim` in R with just this implementation to compute the maximum-likelihood estimate of the allele parameters.

```
optim(c(0.3, 0.3), loglik, x = c(85, 196, 341))
```

```
## $par
## [1] 0.07084643 0.18871900
##
## $value
## [1] 600.481
##
## $counts
## function gradient
##       71       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The `optim` function uses an algorithm called Nelder-Mead as the default algorithm that relies on log-likelihood evaluations only. It is slow but fairly robust, though a bit of thought has to go into the initial parameter choice.

```
optim(c(0, 0), loglik, x = c(85, 196, 341))
```

```
## Error in optim(c(0, 0), loglik, x = c(85, 196, 341)): function
cannot be evaluated at initial parameters```
```

```
Starting the algorithm in a boundary value where the negative log-likelihood attains
the value $\infty$ does not work.

The computations can beneficially be implemented in greater
generality. The function `M` sums the cells that are collapsed,
which has to be specified by the `group` argument.


```r
M <- function(x, group)
  as.vector(tapply(x, group, sum))
```

The log-likelihood is then implemented for multinomial cell collapsing via `M` and two problem specific arguments to the `loglik` function. One of these is a vector specifying the grouping structure of the collapsing. The other is a function that determines the parametrization that maps the parameter that we are optimizing over to the cell probabilities. In the implementation it is assumed that this `prob` function in addition encodes parameter constraints by returning `NULL` if parameter constraints are violated.

```
loglik <- function(par, x, prob, group, ...) {
  p <- prob(par)
  if(is.null(p)) return(Inf)
  - sum(x * log(M(prob(par), group)))
}
```

The function `prob` is implemented specifically for the peppered moths as follows.

```
prob <- function(p) {
  p[3] <- 1 - p[1] - p[2]
  if (p[1] > 1 || p[1] < 0 || p[2] > 1 || p[2] < 0 || p[3] < 0)
    return(NULL)
  c(p[1]^2, 2 * p[1] * p[2], 2* p[1] * p[3],
    p[2]^2, 2 * p[2] * p[3], p[3]^2)
}
```

We test that the new implementation gives the same result when optimized as using the more problem specific implementation.

```
optim(c(0.3, 0.3), loglik, x = c(85, 196, 341),
      prob = prob, group = c(1, 1, 1, 2, 2, 3))
```

```
## $par
## [1] 0.07084643 0.18871900
##
## $value
## [1] 600.481
##
## $counts
## function gradient
```
```

```
##       71       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Once we have found an estimate of the parameters, we can compute a prediction of the unobserved genotype counts from the phenotype counts using the conditional distribution of the genotypes given the phenotypes. This is straightforward as the conditional distribution of $Y_{A_j} = (Y_k)_{k \in A_j}$, conditionally on X is also a multinomial distribution;

$$Y_{A_j} \mid X = x \sim \text{Mult}\left(\frac{p_{A_j}}{M(p)_j}, x_j\right).$$

The probability parameters are simply p restricted to A_j and renormalized to a probability vector. Observe that this gives

$$E(Y_k \mid X = x) = \frac{x_j p_k}{M(p)_j}$$

for $k \in A_j$. Using the estimated parameters and the `M` function implemented above, we can compute a prediction of the genotype counts as follows.

```
x <- c(85, 196, 341)
group <- c(1, 1, 1, 2, 2, 3)
p <- prob(c(0.07084643, 0.18871900))
x[group] * p / M(p, group)[group]
```

```
## [1]   3.121549  16.630211  65.248241  22.154520 173.845480 341.000000
```

This is of interest in itself, but computing these conditional expectations will also be central for the EM algorithm in Chapter 8.

6.3 Regression models

Regression models are models of one variable, called the response, conditionally on one or more other variables, called the predictors. Typically we use models that assume conditional independence of the responses given the predictors, and a particularly convenient class of regression models is based on exponential families.

If we let $y_i \in \mathbb{R}$ denote the ith response and $x_i \in \mathbb{R}^p$ the ith predictor we can consider the exponential family of models with joint density

$$f(\mathbf{y} \mid \mathbf{X}, \beta) = \prod_{i=1}^{n} e^{\theta^T t_i(y_i|x_i) - \log \varphi_i(\theta)}$$

for suitably chosen base measures. Here

$$\mathbf{X} = \begin{pmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_{n-1}^T \\ x_n^T \end{pmatrix}$$

is called the *model matrix* and is an $n \times p$ matrix.

Example 6.3. If $y_i \in \mathbb{N}_0$ are counts we often use a Poisson regression model with point probabilities (density w.r.t. counting measure)

$$p_i(y_i \mid x_i) = e^{-\mu(x_i)} \frac{\mu(x_i)^{y_i}}{y_i!}.$$

If the mean depends on the predictors in a log-linear way, $\log(\mu(x_i)) = x_i^T \beta$, then

$$p_i(y_i \mid x_i) = e^{\beta^T x_i y_i - \exp(x_i^T \beta)} \frac{1}{y_i!}.$$

The factor $1/y_i!$ can be absorbed into the base measure, and we recognize this Poisson regression model as an exponential family with sufficient statistics

$$t_i(y_i) = x_i y_i$$

and

$$\log \varphi_i(\beta) = \exp(x_i^T \beta).$$

To implement numerical optimization algorithms for computing the maximum-likelihood estimate we note that

$$t(\mathbf{y}) = \sum_{i=1}^n x_i y_i = \mathbf{X}^T \mathbf{y} \quad \text{and} \quad \kappa(\beta) = \sum_{i=1}^n e^{x_i^T \beta},$$

that

$$\nabla \kappa(\beta) = \sum_{i=1}^n x_i e^{x_i^T \beta},$$

and that

$$D^2 \kappa(\beta) = \sum_{i=1}^n x_i x_i^T e^{x_i^T \beta} = \mathbf{X}^T \mathbf{W}(\beta) \mathbf{X}$$

where $\mathbf{W}(\beta)$ is a diagonal matrix with $\mathbf{W}(\beta)_{ii} = \exp(x_i^T \beta)$. All these formulas follow directly from the general formulas for exponential families.

To illustrate the use of a Poisson regression model we consider the following data set from a major Swedish supermarket chain. It contains the number of bags of frozen vegetables sold in a given week in a given store under a marketing campaign. A predicted number of items sold in a normal week (without a campaign) based on historic sales is included. It is of interest to recalibrate this number to give a good prediction of the number of items actually sold.

```
vegetables <- read_csv("data/vegetables.csv", col_types = cols(store = "c"))
```

```
summary(vegetables)
```

```
##       sale          normalSale        store
##  Min.   :  1.00   Min.   :  0.20   Length:1066
##  1st Qu.: 12.00   1st Qu.:  4.20   Class :character
##  Median : 21.00   Median :  7.25   Mode  :character
##  Mean   : 40.29   Mean   : 11.72
##  3rd Qu.: 40.00   3rd Qu.: 12.25
##  Max.   :571.00   Max.   :102.00
```

It is natural to model the number of sold items using a Poisson regression model, and we will consider the following simple log-linear model:

$$\log(E(\text{sale} \mid \text{normalSale})) = \beta_0 + \beta_1 \log(\text{normalSale}).$$

This is an example of an exponential family regression model as above with a Poisson response distribution. It is a standard regression model that can be fitted to data using the `glm` function and using the formula interface to specify how the response should depend on the normal sale. The model is fitted by computing the maximum-likelihood estimate of the two parameters β_0 and β_1.

```
# A Poisson regression model
pois_model_null <- glm(
  sale ~ log(normalSale),
  data = vegetables,
  family = poisson
)
```

```
summary(pois_model_null)
```

```
##
## Call:
## glm(formula = sale ~ log(normalSale), family = poisson, data = vegetables)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -16.218   -2.677   -0.864    1.324   21.730
##
## Coefficients:
##                 Estimate Std. Error z value Pr(>|z|)
## (Intercept)        1.461      0.015    97.2   <2e-16 ***
## log(normalSale)    0.922      0.005   184.4   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 51177  on 1065  degrees of freedom
## Residual deviance: 17502  on 1064  degrees of freedom
## AIC: 22823
##
## Number of Fisher Scoring iterations: 5
```

The fitted model of the expected sale as a function of the normal sale, x, is

$$x \mapsto e^{1.46+0.92\times\log(x)} = (4.31 \times x^{-0.08}) \times x.$$

This model roughly predicts a four-fold increase of the sale during a campaign, though the effect decays with increasing x. If the normal sale is 10 items the factor is $4.31 \times 10^{-0.08} = 3.58$, and if the normal sale is 100 items the factor reduces to $4.31 \times 100^{-0.08} = 2.98$.

We are not entirely satisfied with this model. It is fitted across all stores in the data set, and it is not obvious that the same effect should apply across all stores. Thus we would like to fit a model of the form

$$\log(E(\text{sale})) = \beta_{\text{store}} + \beta_1 \log(\text{normalSale})$$

instead. Fortunately, this is also straightforward using `glm`.

```
## Note, variable store is a factor! The 'intercept' is removed
## in the formula to obtain a parametrization as above.
pois_model <- glm(
  sale ~ store + log(normalSale) - 1,
  data = vegetables,
  family = poisson
)
```

We will not print all the individual store effects as there are 352 individual stores.

```
summary(pois_model)$coefficients[c(1, 2, 3, 4, 353), ]
```

```
##                 Estimate Std. Error z value   Pr(>|z|)
## store1            2.7182    0.12756  21.309 9.416e-101
## store10           4.2954    0.12043  35.668 1.254e-278
## store100          3.4866    0.12426  28.060 3.055e-173
## store101          3.3007    0.11791  27.993 1.989e-172
## log(normalSale)   0.2025    0.03127   6.474  9.536e-11
```

We should note that the coefficient of log(normalSale) has changed considerably, and the model is a somewhat different model now for the individual stores.

From a computational viewpoint the most important thing that has changed is that the number of parameters has increased a lot. In the first and simple model there are two parameters. In the second model there are 353 parameters. Computing the maximum-likelihood estimate is a considerably more difficult problem in the second case.

6.4 Finite mixture models

A finite mixture model is a model of a pair of random variables (Y, Z) with $Z \in \{1, \ldots, K\}$, $P(Z = k) = p_k$, and the conditional distribution of Y given $Z = k$ having density $f_k(\cdot \mid \theta_k)$. The joint density is then

$$(y, k) \mapsto f_k(y \mid \theta_k) p_k,$$

and the marginal density for the distribution of Y is

$$f(y \mid \theta) = \sum_{k=1}^{K} f_k(y \mid \theta_k) p_k$$

where θ is the vector of all parameters. We say that the model has K mixture components and call $f_k(\cdot \mid \theta_k)$ the mixture distributions and p_k the mixture weights.

The main usage of mixture models is to situations where Z is not observed. In practice, only Y is observed, and parameter estimation has to be based on the marginal distribution of Y with density $f(\cdot \mid \theta)$, which is a weighted sum of the mixture distributions.

The set of all probability measures on $\{1, \ldots, K\}$ is an exponential family with sufficient statistic

$$\tilde{t}_0(k) = (\delta_{1k}, \delta_{2k}, \ldots, \delta_{(K-1)k}) \in \mathbb{R}^{K-1},$$

canonical parameter $\alpha = (\alpha_1, \ldots, \alpha_{K-1}) \in \mathbb{R}^{K-1}$ and

$$p_k = \frac{e^{\alpha_k}}{1 + \sum_{l=1}^{K-1} e^{\alpha_l}}.$$

When f_k is an exponential family as well with sufficient statistic $\tilde{t}_k : \mathcal{Y} \to \mathbb{R}^{p_k}$ and θ_k the canonical parameter, we bundle $\alpha, \theta_1, \ldots, \theta_K$ into θ and define

$$t_1(y) = \begin{pmatrix} \tilde{t}_0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

together with

$$t_2(y \mid k) = \begin{pmatrix} 0 \\ \delta_{1k}\tilde{t}_1(y) \\ \delta_{2k}\tilde{t}_2(y) \\ \vdots \\ \delta_{Kk}\tilde{t}_K(y) \end{pmatrix}$$

we see that we have an exponential family of the joint distribution of (Y, Z) with the $p = K - 1 + p_1 + \ldots + p_K$-dimensional canonical parameter θ, with the sufficient statistic t_1 determining the marginal distribution of Z and with the sufficient statistic t_2 determining the *conditional* distribution of Y given Z. We have here made the conditioning variable explicit.

The marginal density of Y in the exponential family parametrization then becomes

$$f(y \mid \theta) = \sum_{k=1}^{K} e^{\theta^T(t_1(k)+t_2(y|k)) - \log \varphi_1(\theta) - \log \varphi_2(\theta|k)}.$$

For small K it is usually unproblematic to implement the computation of the marginal density using the formula above, and the computation of derivatives can likewise be implemented based on the formulas derived in Section 6.1.3.

6.4.1 Gaussian mixtures

A Gaussian mixture model is a mixture model where all the mixture distributions are Gaussian distributions but potentially with different means and variances. In this section we focus on the simplest Gaussian mixture model with $K = 2$ mixture components.

When $K = 2$, the Gaussian mixture model is parametrized by the five parameters: the mixture weight $p = P(Z = 1) \in (0, 1)$, the two means $\mu_1, \mu_2 \in \mathbb{R}$, and the two variances $\sigma_1, \sigma_2 > 0$. This is *not* the parametrization using canonical exponential family parameters, which we return to below. First we will simply simulate random variables from this model.

```
sigma1 <- 1
sigma2 <- 2
mu1 <- -0.5
mu2 <- 4
p <- 0.5
n <- 5000
z <- sample(c(TRUE, FALSE), n, replace = TRUE, prob = c(p, 1 - p))
## Conditional simulation from the mixture components
y <- numeric(n)
n1 <- sum(z)
y[z] <- rnorm(n1, mu1, sigma1)
y[!z] <- rnorm(n - n1, mu2, sigma2)
```

The simulation above generated 5000 samples from a two-component Gaussian mixture model with mixture distributions $\mathcal{N}(-0.5, 1)$ and $\mathcal{N}(4, 4)$, and with each component having weight 0.5. This gives a bimodal distribution as illustrated by the histogram on Figure 6.1.

```
yy <- seq(-3, 11, 0.1)
dens <- p * dnorm(yy, mu1, sigma1) + (1 - p) * dnorm(yy, mu2, sigma2)
ggplot(mapping = aes(y, ..density..)) +
  geom_histogram(bins = 20) +
  geom_line(aes(yy, dens), color = "blue", size = 1) +
  stat_density(bw = "SJ", geom="line", color = "red", size = 1)
```

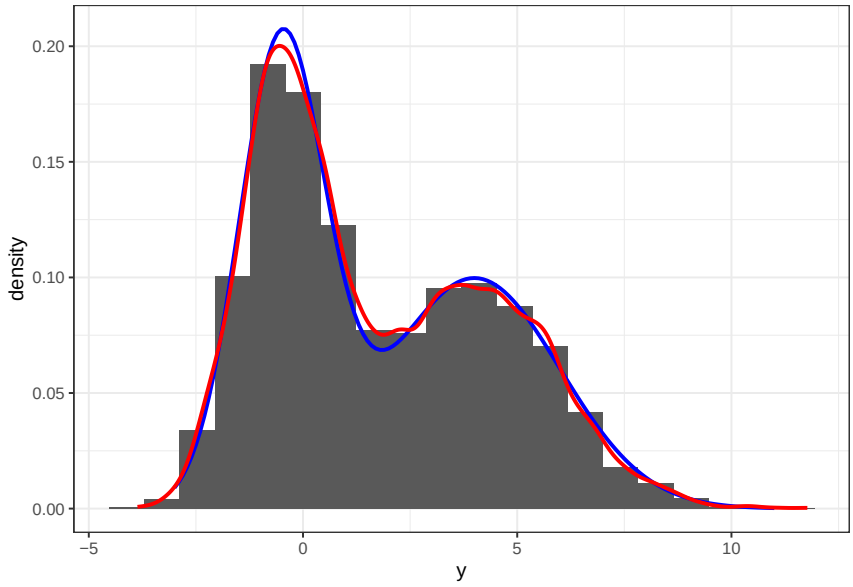


Figure 6.1: Histogram and density estimate (red) of data simulated from a two-component Gaussian mixture distribution. The true mixture distribution has

It is possible to give a mathematically different representation of the marginal distribution of Y that is sometimes useful. Though it gives the same marginal distribution from the same components, it does provide a different interpretation of what a mixture model is a model of, and it does provide a different way of simulating from a mixture distribution.

If we let $Y_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $Y_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$ be independent, and independent of Z, we can define

$$Y = 1(Z = 1)Y_1 + 1(Z = 2)Y_2 = Y_Z. \tag{6.4}$$

Clearly, the conditional distribution of Y given $Z = k$ is f_k. From (6.4) it follows directly that

$$E(Y^n) = P(Z = 1)E(Y_1^n) + P(Z = 2)E(Y_2^n) = pm_1(n) + (1 - p)m_2(n)$$

where $m_k(n)$ denotes the nth non-central moment of the kth mixture component. In particular,

$$E(Y) = p\mu_1 + (1 - p)\mu_2.$$

The variance can be found from the second moment as

$$V(Y) = p(\mu_1^2 + \sigma_1^2) + (1-p)(\mu_2^2 + \sigma_2^2) - (p\mu_1 + (1-p)\mu_2)^2 \tag{6.5}$$
$$= p\sigma_1^2 + (1-p)\sigma_2^2 + p(1-p)(\mu_1^2 + \mu_2^2 - 2\mu_1\mu_2). \tag{6.6}$$

While it is certainly possible to derive this formula by other means, using (6.4) gives a simple argument based on elementary properties of expectation and independence of (Y_1, Y_2) and Z.

The construction via (6.4) has an interpretation that differs from how a mixture model was defined in the first place. Though (Y, Z) has the correct joint distribution, Y is by (6.4) the result of Z *selecting* one out of two possible observations. The difference can best be illustrated by an example. Suppose that we have a large population consisting of married couples entirely. We can draw a sample of individuals (ignoring the marriage relations completely) from this population and let Z denote the sex and Y the height of the individual. Then Y follows a mixture distribution with two components corresponding to males and females according to the definition. At the risk of being heteronormative, suppose that all couples consist of one male and one female. We could then also draw a sample of married couples, and for each couple flip a coin to decide whether to report the male's or the female's height. This corresponds to the construction of Y by (6.4). We get the same marginal mixture distribution of heights though.

Arguably the heights of individuals in a marriage are not independent, but this is actually immaterial. Any dependence structure between Y_1 and Y_2 is lost in the transformation (6.4), and we can just as well assume them independent for mathematical convenience. We won't be able to tell the difference from observing only Y (and Z) anyway.

We illustrate below how (6.4) can be used for alternative implementations of ways to simulate from the mixture model. We compare empirical means and variances to the theoretical values to test if all the implementations actually simulate from the Gaussian mixture model.

```
## Mean
mu <- p * mu1 + (1 - p) * mu2

## Variance
sigmasq <- p * sigma1^2 + (1 - p) * sigma2^2 +
  p * (1-p)*(mu1^2 + mu2^2 - 2 * mu1 * mu2)

## Simulation using the selection formulation via 'ifelse'
y2 <- ifelse(z, rnorm(n, mu1, sigma1), rnorm(n, mu2, sigma2))

## Yet another way of simulating from a mixture model
## using arithmetic instead of 'ifelse' for the selection
## and with Y_1 and Y_2 actually being dependent
y3 <- rnorm(n)
y3 <- z * (sigma1 * y3 + mu1) + (!z) * (sigma2 * y3 + mu2)

## Returning to the definition again, this last method simulates conditionally
## from the mixture components via transformation of an underlying Gaussian
## variable with mean 0 and variance 1
y4 <- rnorm(n)
y4[z] <- sigma1 * y4[z] + mu1
y4[!z] <- sigma2 * y4[!z] + mu2
```

```
## Tests
data.frame(mean = c(mu, mean(y), mean(y2), mean(y3), mean(y4)),
      variance = c(sigmasq, var(y), var(y2), var(y3), var(y4)),
      row.names = c("true", "conditional", "ifelse", "arithmetic",
                    "conditional2" ))
```

```
##                   mean variance
## true          1.750000 7.562500
## conditional   1.728195 7.381843
## ifelse        1.713098 7.513963
## arithmetic    1.708420 7.566312
## conditional2 1.700052 7.463941
```

In terms of run time there is not a big difference between three of the ways of simulating from a mixture model. A benchmark study (not shown) will reveal that the first and third methods are comparable in terms of run time and slightly faster than the fourth, while the second using `ifelse` takes more than twice as much time as the others. This is unsurprising as the `ifelse` method takes (6.4) very literally and generates twice the number of Gaussian variables actually needed.

The marginal density of Y is

$$f(y) = p\frac{1}{\sqrt{2\pi\sigma_1^2}}e^{-\frac{(y-\mu_1)^2}{2\sigma_1^2}} + (1-p)\frac{1}{\sqrt{2\pi\sigma_2^2}}e^{-\frac{(y-\mu_2)^2}{2\sigma_2^2}}$$

as given in terms of the parameters p, μ_1, μ_2, σ_1^2 and σ_2^2.

Returning to the canonical parameters we see that they are given as follows:

$$\theta_1 = \log\frac{p}{1-p}, \quad \theta_2 = \frac{\mu_1}{2\sigma_1^2}, \quad \theta_3 = \frac{1}{2\sigma_1^2}, \quad \theta_4 = \frac{\mu_2}{\sigma_2^2}, \quad \theta_5 = \frac{1}{2\sigma_2^2}.$$

The joint density in this parametrization then becomes

$$(y,k) \mapsto \begin{cases} \exp\left(\theta_1 + \theta_2 y - \theta_3 y^2 - \log(1+e^{\theta_1}) - \frac{\theta_2^2}{4\theta_3} + \frac{1}{2}\log(\theta_3)\right) & \text{if } k = 1 \\ \exp\left(\theta_4 y - \theta_5 y^2 - \log(1+e^{\theta_1}) - \frac{\theta_4^2}{4\theta_5} + \frac{1}{2}\log(\theta_5)\right) & \text{if } k = 2 \end{cases}$$

and the marginal density for Y is

$$f(y \mid \theta) = \exp\left(\theta_1 + \theta_2 y - \theta_3 y^2 - \log(1+e^{\theta_1}) - \frac{\theta_2^2}{4\theta_3} + \frac{1}{2}\log(\theta_3)\right) \tag{6.7}$$

$$+ \exp\left(\theta_4 y - \theta_5 y^2 - \log(1+e^{\theta_1}) - \frac{\theta_4^2}{4\theta_5} + \frac{1}{2}\log(\theta_5)\right). \tag{6.8}$$

There is no apparent benefit to the canonical parametrization when considering the marginal density. It is, however, of value when we need the logarithm of the joint density as we will for the EM-algorithm.

6.5 Mixed models

A mixed model is a regression model of observations that allows for random variation at two different levels. In this section we will focus on mixed models in an exponential family context. Mixed models can be considered in greater generality but there will then be little shared structure and one has to deal with the models much more on a case-by-case manner.

In a mixed model we have observations $y_j \in \mathcal{Y}$ and $z \in \mathcal{Z}$ such that:

- the distribution of z is an exponential family with canonical parameter θ_0
- *conditionally* on z the y_js are independent with a distribution from an exponential family with sufficient statistics $t_j(\cdot \mid z)$.

This definition emphasizes how the y_js have variation at two levels. There is variation in the underlying z, which is the first level of variation (often called the *random effect*), and then there is variation among the y_js given the z, which is the second level of variation. It is a special case of hierarchical models (Bayesian networks with tree graphs) also known as *multilevel* models, with the mixed model having only two levels. When we observe data from such a model we typically observe independent replications, $(y_{ij})_{j=1,\ldots,m_i}$ for $i = 1, \ldots, n$, of the y_js only. Note that we allow for a different number, m_i, of y_js for each i.

The simplest class of mixed models is obtained by $t_j = t$ not depending on j, and

$$t(y_j \mid z) = \begin{pmatrix} t_1(y_j) \\ t_2(y_j, z) \end{pmatrix}$$

for some fixed maps $t_1 : \mathcal{Y} \mapsto \mathbb{R}^{p_1}$ and $t_2 : \mathcal{Y} \times \mathcal{Z} \mapsto \mathbb{R}^{p_2}$. This is called a *random effects* model (this is a model where there are no *fixed effects* in the sense that t_j does not depend on j, and given the random effect z the y_js are i.i.d.). The canonical parameters associated with such a model are θ_0 that enters into the distribution of the random effect, $\theta_1 \in \mathbb{R}^{p_1}$ and $\theta_2 \in \mathbb{R}^{p_2}$ that enter into the conditional distribution of y_j given z.

Example 6.4. The special case of a *Gaussian, linear* random effects model is the model where z is $\mathcal{N}(0, 1)$ distributed, $\mathcal{Y} = \mathbb{R}$ (with base measure proportional to Lebesgue measure) and the sufficient statistic is

$$t(y_j \mid z) = \begin{pmatrix} y_j \\ -y_j^2 \\ z y_j \end{pmatrix}.$$

There are no free parameters in the distribution of z.

From Example 6.2 it follows that the conditional variance of y given z is

$$\sigma^2 = \frac{1}{2\theta_2}$$

and the conditional mean of y given z is

$$\frac{\theta_1 + \theta_3 z}{2\theta_2} = \sigma^2\theta_1 + \sigma^2\theta_3 z.$$

Reparametrizing in terms of σ^2, $\beta_0 = \sigma^2\theta_1$ and $\nu = \sigma^2\theta_3$ we see how the conditional distribution of y_j given z is $\mathcal{N}(\beta_0 + \nu z, \sigma^2)$. From this it is clear how the mixed model of y_j *conditionally on* z is a regression model. However, we do not observe z in practice.

Using the above distributional result we can see that the Gaussian random effects model of observations y_{ij} can equivalently be stated as

$$Y_{ij} = \beta_0 + \nu Z_i + \varepsilon_{ij}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m_i$ where $Z_1, \ldots, Z_n$ are i.i.d. $\mathcal{N}(0,1)$-distributed and independent of $\varepsilon_{11}, \varepsilon_{12}, \ldots, \varepsilon_{1n_1}, \ldots, \varepsilon_{mn_m}$ that are themselves i.i.d. $\mathcal{N}(0,\sigma^2)$-distributed.

Chapter 7

Numerical optimization

The main application of numerical optimization in statistics is for the computation of parameter estimates. Typically by maximizing the likelihood function or by maximizing or minimizing another estimation criterion. The focus of this chapter is on optimization algorithms for (penalized) maximum likelihood estimation. Out of tradition we formulate all results and algorithms in terms of minimization and not maximization.

The generic optimization problem considered is the minimization of $H : \Theta \to \mathbb{R}$ for $\Theta \subseteq \mathbb{R}^p$ an open set and H twice differentiable. In applications, $H = -\ell$, the negative log-likelihood function, or $H = -\ell + J$, where $J : \Theta \to \mathbb{R}$ is a *penalty function*, likewise twice differentiable, that does not depend upon data.

Statistical optimization problems share some properties that we should pay attention to. The term

$$-\ell(\theta) = -\sum_{i=1}^{N} \log(f_\theta(x_i))$$

in the objective function to be minimized is a sum over the data, and the more data we have the more computationally demanding it is to evaluate H and its derivatives. There are exceptions, though, when a sufficient statistic can be computed upfront, but generally we must expect run time to grow with data size. Additionally, high precision in the computed (local) minimizer is typically not necessary. If the numerical error is already orders of magnitudes smaller than the statistical uncertainty of the parameter estimate being computed, further optimization makes no relevant difference.

In fact, blindly pursuing the global maximum of the likelihood can lead you astray if your model is not well behaved. In situations where H is not convex the negative log-likelihood may be unbounded, e.g. for finite mixtures, yet a good estimate can be found as a local minimizer. So even if we phrase the objective as a global optimization objective, we may be equally interested in local minima or even just stationary points; solutions to $\nabla H(\theta) = 0$. Optimization is a computational tool used in statistics for adapting models to data – but the minimizer of H for a particular data set is not intrinsically interesting. There may be little gained from computing a minimizer of H to high numerical precision, and assessment of model fit and quantification of model uncertainty is of greater practical importance than a highly accurately determined minimizer.

For the generic minimization problem considered in this chapter, the practical challenge when implementing algorithms in R is typically to implement efficient evaluation of H and its derivatives. In particular, efficient evaluations of $-\ell$. Several choices of standard optimization algorithms are possible and some are already implemented and available in R. For many

practical purposes the BFGS-algorithms as implemented via `optim()` work well and require only the computation of gradients. It is, of course, paramount that H and ∇H are correctly implemented, and efficiency of the algorithms is largely determined by the efficiency of the implementation of H and ∇H but also by the choice of parametrization.

For some optimization problems, Newton-type algorithms are preferable, and standard implementations are available in R through `nlm()` and `nls()` but with different interfaces than `optim()`.

7.1 Algorithms and convergence

A numerical optimization algorithm computes from an initial value $\theta_0 \in \Theta$ a sequence $\theta_1, \theta_2, \ldots \in \Theta$. One could hope for

$$\theta_n \to \arg\min_\theta H(\theta)$$

for $n \to \infty$, but less can typically be guaranteed. First, the global minimizer may not exist or it may not be unique, in which case the convergence itself is undefined or ambiguous. Second, θ_n can in general only be shown to converge to a *local* minimizer if anything. Third, θ_n may not converge, even if $H(\theta_n)$ converges to a local minimum.

To discuss properties of optimization algorithms in greater detail we need ways to quantify, analyze and monitor their convergence. This can be done in terms of θ_n, $H(\theta_n)$ or $\nabla H(\theta_n)$. Focusing on $H(\theta_n)$ we say that an algorithm is a *descent algorithm* if

$$H(\theta_0) \geq H(\theta_1) \geq H(\theta_2) \geq \ldots.$$

If all inequalities are sharp (except if some θ_i is a local minimizer), the algorithm is called a *strict* descent algorithm. The sequence $H(\theta_n)$ is convergent for any descent algorithm if H is bounded from below, e.g. if the *level set*

$$\text{lev}(\theta_0) = \{\theta \in \Theta \mid H(\theta) \leq H(\theta_0)\}$$

is compact. However, even for a strict descent algorithm, $H(\theta_n)$ may descent in smaller and smaller steps toward a limit that is not a (local) minimum. A good optimization algorithm needs to guarantee more than descent – it needs to guarantee *sufficient* descent in each step.

Many algorithms can be phrased as

$$\theta_n = \Phi(\theta_{n-1})$$

for a map $\Phi : \Theta \to \Theta$. When Φ is continuous and $\theta_n \to \theta_\infty$ it follows that

$$\Phi(\theta_n) \to \Phi(\theta_\infty).$$

Since $\Phi(\theta_n) = \theta_{n+1} \to \theta_\infty$ we see that

$$\Phi(\theta_\infty) = \theta_\infty.$$

That is, θ_∞ is a *fixed point* of Φ. If θ_n is not convergent, any accumulation point of θ_n will still be a fixed point of Φ. Thus we can search for potential accumulation points by searching for fixed points of the map $\Phi : \Theta \to \Theta$.

Mathematics is full of *fixed point theorems* that: i) give conditions under which a map has a fixed point; and ii) in some cases guarantee that the iterates $\Phi^{\circ n}(\theta_0)$ converge to a fixed point.

The most prominent result is Banach's fixed point theorem. It states that if Φ is a *contraction*, that is,

$$\|\Phi(\theta) - \Phi(\theta')\| \leq c\|\theta - \theta'\|$$

for a constant $c \in [0, 1)$ (using any norm), then Φ has a unique fixed point and $\theta_n = \Phi^{\circ n}(\theta_0)$ converges to that fixed point for any starting point $\theta_0 \in \Theta$.

We will show how a simple gradient descent algorithm can be analyzed – both as a descent algorithm and through Banach's fixed point theorem. This will demonstrate basic proof techniques as well as typical conditions on H that can give convergence results for optimization algorithms. We will only scratch the surface here with the intention that it can motivate the algorithms introduced in subsequent sections and chapters as well as empirical techniques for practical assessment of convergence.

Indeed, theory rarely provides us with sharp quantitative results on the rate of convergence and we will need computational techniques to monitor and measure convergence of algorithms in practice. Otherwise we cannot compare the efficiency of different algorithms.

7.1.1 Gradient descent

We will assume in this section that $\Theta = \mathbb{R}^p$. This will simplify the analysis a bit, but with minor modifications it can be generalized to the case where Θ is open and convex.

Suppose that $D^2H(\theta)$ has *numerical radius* uniformly bounded by L, that is,

$$|\gamma^T D^2 H(\theta)\gamma| \leq L\|\gamma\|_2^2$$

for all $\theta \in \Theta$ and $\gamma \in \mathbb{R}^p$. The *gradient descent* algorithm with a fixed step length $1/(L+1)$ is given by

$$\theta_n = \theta_{n-1} - \frac{1}{L+1}\nabla H(\theta_{n-1}). \tag{7.1}$$

Fixing n there is by Taylor's theorem a $\tilde{\theta} = \alpha\theta_n + (1-\alpha)\theta_{n-1}$ (where $\alpha \in [0,1]$) on the line between θ_n and θ_{n-1} such that

$$\begin{aligned}
H(\theta_n) &= H(\theta_{n-1}) - \frac{1}{L+1}\|\nabla H(\theta_{n-1})\|_2^2 + \\
&\qquad \frac{1}{(L+1)^2}\nabla H(\theta_{n-1})^T D^2 H(\tilde{\theta})\nabla H(\theta_{n-1}) \\
&\leq H(\theta_{n-1}) - \frac{1}{L+1}\|\nabla H(\theta_{n-1})\|_2^2 + \frac{L}{(L+1)^2}\|\nabla H(\theta_{n-1})\|_2^2 \\
&= H(\theta_{n-1}) - \frac{1}{(L+1)^2}\|\nabla H(\theta_{n-1})\|_2^2.
\end{aligned}$$

This shows that $H(\theta_n) \leq H(\theta_{n-1})$, and if θ_{n-1} is not a stationary point, $H(\theta_n) < H(\theta_{n-1})$. That is, the algorithm is a descent algorithm, and unless it hits a stationary point it is a strict descent algorithm.

It furthermore follows that

$$H(\theta_n) = (H(\theta_n) - H(\theta_{n-1})) + (H(\theta_{n-1}) - H(\theta_{n-2})) + \ldots \tag{7.2}$$
$$+ (H(\theta_1) - H(\theta_0)) + H(\theta_0) \tag{7.3}$$
$$\leq H(\theta_0) - \frac{1}{(L+1)^2} \sum_{k=1}^{n} \|\nabla H(\theta_{k-1})\|_2^2. \tag{7.4}$$

If H is bounded below, $\sum_{k=1}^{\infty} \|\nabla H(\theta_{k-1})\|_2^2 < \infty$ and hence

$$\|\nabla H(\theta_n)\|_2 \to 0$$

for $n \to \infty$. For any accumulation point, θ_∞, of the sequence θ_n, it follows by continuity of ∇H that

$$\nabla H(\theta_\infty) = 0,$$

and θ_∞ is a stationary point. If H has a *unique* stationary point in $\text{lev}(\theta_0)$, θ_∞ is a (local) minimizer, and

$$\theta_n \to \theta_\infty$$

for $n \to \infty$.

The gradient descent algorithm (7.1) is given by the map

$$\Phi_\nabla(\theta) = \theta - \frac{1}{L+1} \nabla H(\theta).$$

The gradient descent map, Φ_∇, has θ as fixed point if and only if

$$\nabla H(\theta) = 0,$$

that is, if and only if θ is a stationary point.

We will show that Φ_∇ is a contraction on Θ under the assumption that the eigenvalues of the (symmetric) matrix $D^2H(\theta)$ for all $\theta \in \Theta$ are contained in an interval $[l, L]$ with $0 < l \leq L$. If $\theta, \theta' \in \Theta$ we find by Taylor's theorem that

$$\nabla H(\theta) = \nabla H(\theta') + D^2H(\tilde{\theta})(\theta - \theta')$$

for some $\tilde{\theta}$ on the line between θ and θ'. For the gradient descent map this gives that

$$\begin{aligned}
\|\Phi_\nabla(\theta) - \Phi_\nabla(\theta')\|_2 &= \left\| \theta - \theta' - \frac{1}{L+1} \left(\nabla H(\theta) - \nabla H(\theta') \right) \right\|_2 \\
&= \left\| \theta - \theta' - \frac{1}{L+1} \left(D^2H(\tilde{\theta})(\theta - \theta') \right) \right\|_2 \\
&= \left\| \left(I - \frac{1}{L+1} D^2H(\tilde{\theta}) \right) (\theta - \theta') \right\|_2 \\
&\leq \left(1 - \frac{l}{L+1} \right) \|\theta - \theta'\|_2,
\end{aligned}$$

since the eigenvalues of $I - \frac{1}{L+1} D^2H(\tilde{\theta})$ are all between $1 - L/(L+1)$ and $1 - l/(L+1)$. This shows that Φ_∇ is a contraction for the 2-norm with $c = 1 - l/(L+1) < 1$. From Banach's fixed point theorem it follows that there is a unique stationary point, θ_∞, and $\theta_n \to \theta_\infty$. Since $D^2H(\theta_\infty)$ is positive definite, θ_∞ is a (global) minimizer.

To summarize, if $D^2H(\theta)$ has uniformly bounded numerical radius, and if H has a unique stationary point θ_∞ in the compact set $\text{lev}(\theta_0)$, then the gradient descent algorithm is a strict descent algorithm that converges toward that (local) minimizer θ_∞. The fixed step length was key to this analysis. The upper bound on the numerical radius of $D^2H(\theta)$ guarantees descent with a fixed step length, and the fixed step length then guarantees sufficient descent.

When the eigenvalues of $D^2H(\theta)$ are in $[l, L]$ for $0 < l \leq L$, H is a *strongly convex function* with a unique global minimizer θ_∞ and all level sets $\text{lev}(\theta_0)$ compact. Convergence can then also be established via Banach's fixed point theorem. The constant $c = 1 - l/(L+1) = 1 - \kappa^{-1}$ with $\kappa = (L+1)/l$ then quantifies the rate of the convergence, as will be discussed in greater detail in the next section. The constant κ is an upper bound on the conditioning number of the matrix $D^2H(\theta)$ uniformly in θ, and a large value of κ means that c is close to 1 and the convergence can be slow. A large conditioning number for the second derivative indicates that the graph of H looks like a narrow ravine, in which case we can expect the gradient descent algorithm to be slow.

7.1.2 Convergence rate

When $\theta_n = \Phi^{\circ n}(\theta_0)$ for a contraction Φ, Banach's fixed point theorem tells us that there is a unique fixed point θ_∞. That Φ is a contraction further implies that

$$\|\theta_n - \theta_\infty\| = \|\Phi(\theta_{n-1}) - \theta_\infty\| \leq c\|\theta_{n-1} - \theta_\infty\| \leq c^n\|\theta_0 - \theta_\infty\|.$$

That is, $\|\theta_n - \theta_\infty\| \to 0$ with at least geometric rate $c < 1$.

In the numerical optimization literature, convergence at a geometric rate is known as linear convergence (the number of correct digits increases linearly with the number of iterations), and a linearly convergent algorithm is often quantified in terms of its asymptotic convergence rate.

Definition 7.1. Let $(\theta_n)_{n\geq 1}$ be a convergent sequence in $\mathbb{R}^p$ with limit θ_∞. Let $\|\cdot\|$ be a norm on $\mathbb{R}^p$. We say that the convergence is linear if

$$\limsup_{n\to\infty} \frac{\|\theta_n - \theta_\infty\|}{\|\theta_{n-1} - \theta_\infty\|} = r$$

for some $r \in (0, 1)$, in which case r is called the asymptotic rate.

Convergence of an algorithm can be faster or slower than linear. If

$$\limsup_{n\to\infty} \frac{\|\theta_n - \theta_\infty\|}{\|\theta_{n-1} - \theta_\infty\|} = 1$$

we say that the convergence is sublinear, and if

$$\limsup_{n\to\infty} \frac{\|\theta_n - \theta_\infty\|}{\|\theta_{n-1} - \theta_\infty\|} = 0$$

we say that the convergence is superlinear. There is also a refined notion of convergence order for superlinearly convergent algorithms that more precisely describes the convergence.

For a contraction, $\theta_n = \Phi^{\circ n}(\theta_0)$ converges linearly or superlinearly. If it converges linearly, the asymptotic rate is bounded by c, but this is a global constant and may be pessimistic. The following lemma provides a local upper bound on the asymptotic rate in terms of the derivative of Φ in the limit θ_∞.

Lemma 7.1. *Let $\Phi : \mathbb{R}^p \to \mathbb{R}^p$ be twice continuously differentiable. If $\theta_n = \Phi^{\circ n}(\theta_0)$ converges linearly toward a fixed point θ_∞ of Φ then*

$$r_{\max} = \sup_{\gamma:\|\gamma\|=1} \|D\Phi(\theta_\infty)\gamma\|$$

is an upper bound on the asymptotic rate.

Proof. By Taylor's theorem,

$$\theta_n = \theta_\infty + D\Phi(\theta_\infty)(\theta_{n-1} - \theta_\infty) + o(\|\theta_{n-1} - \theta_\infty\|_2).$$

Rearranging yields

$$\begin{aligned}\limsup_{n\to\infty} \frac{\|\theta_n - \theta_\infty\|}{\|\theta_{n-1} - \theta_\infty\|} &= \limsup_{n\to\infty} \frac{\|D\Phi(\theta_\infty)(\theta_{n-1} - \theta_\infty)\|}{\|\theta_{n-1} - \theta_\infty\|} \\ &= \limsup_{n\to\infty} \|D\Phi(\theta_\infty)\left(\frac{\theta_{n-1} - \theta_\infty}{\|\theta_{n-1} - \theta_\infty\|}\right)\| \\ &\leq r_{\max}.\end{aligned}$$

□

Note that it is possible that $r_{\max} \geq 1$ even if the convergence is linear, in which case $r_{\max}$ is a useless upper bound. Moreover, the actual rate can depend on the starting point θ_0, and even when $r_{\max} < 1$ it only quantifies the worst case convergence rate.

To estimate the actual convergence rate, note that linear convergence with rate r implies that for any $\delta > 0$

$$\|\theta_n - \theta_\infty\| \leq (r+\delta)\|\theta_{n-1} - \theta_\infty\| \leq \ldots \leq (r+\delta)^{n-n_0}\|\theta_{n_0} - \theta_\infty\|$$

$n \geq n_0$ with n_0 sufficiently large (depending on δ). That is,

$$\log\|\theta_n - \theta_\infty\| \leq n\log(r+\delta) + d,$$

for $n \geq n_0$. In practice, we run the algorithm for M iterations, so that $\theta_M = \theta_\infty$ up to computer precision, and we plot $\log\|\theta_n - \theta_M\|$ as a function of n. The decay should be approximately linear for $n \geq n_0$ for some n_0, in which case the slope is about $\log(r) < 0$, which can then be estimated by least squares. A slower-than-linear or faster-than-linear decay indicate that the algorithm converges sublinearly or superlinearly, respectively.

It is, of course, possible to quantify convergence in terms of the sequences $H(\theta_n)$ and $\nabla H(\theta_n)$ instead. For a descent algorithm with $H(\theta_n) \to H(\theta_\infty)$ linearly for $n \to \infty$, we can plot

$$\log(H(\theta_n) - H(\theta_M))$$

as a function of n, and use least squares to estimate the asymptotic rate of convergence of $H(\theta_n)$.

Using $\nabla H(\theta_n)$ to quantify convergence is particularly appealing as we know that the limit is 0. This also means that we can monitor its convergence while the algorithm is running and not only after it has converged. If $\nabla H(\theta_n)$ converges linearly in the norm $\|\cdot\|$, that is,

$$\limsup_{n\to\infty} \frac{\|\nabla H(\theta_n)\|}{\|\nabla H(\theta_{n-1})\|} = r,$$

we have for any $\delta > 0$ that

$$\log \|\nabla H(\theta_n)\| \leq n \log(r + \delta) + d$$

for $n \geq n_0$ and n_0 sufficiently large. Again, least squares can be used to estimate the asymptotic rate of convergence of $\|\nabla H(\theta_n)\|$.

The concepts of linear convergence and asymptotic rate are, unfortunately, not independent of the norm used, nor does linear convergence of θ_n in $\|\cdot\|$ imply linear convergence of $H(\theta_n)$ or $\|\nabla H(\theta_n)\|$. We can shown, though, that if θ_n converges linearly in any norm, the other two sequences converge linearly along an arithmetic subsequence. Similarly, it can be shown that in any other norm than $\|\cdot\|$, θ_n is linearly convergent along an arithmetic subsequence.

Theorem 7.1. *Suppose that H is three times continuously differentiable with $\nabla H(\theta_\infty) = 0$ and $D^2 H(\theta_\infty)$ positive definite. If θ_n converges linearly toward θ_∞ in any norm then for some $k \geq 1$, the subsequences $H(\theta_{nk})$ and $\|\nabla H(\theta_{nk})\|$ converge linearly toward $H(\theta_\infty)$ and 0, respectively.*

Proof. Let $G = D^2 H(\theta_\infty)$ and $g_n = \theta_n - \theta_\infty$. Since G is positive definite it defines a norm, and since all norms on $\mathbb{R}^p$ are equivalent there are constants $a, b > 0$ such that

$$a\|x\|^2 \leq x^T G x \leq b\|x\|^2.$$

By Taylor's theorem,

$$H(\theta_{(n+1)k}) - H(\theta_\infty) = g_{(n+1)k}^T G g_{(n+1)k} + o(\|g_{(n+1)k}\|^2).$$

From this,

$$\begin{aligned}\limsup_{n\to\infty} \frac{H(\theta_{(n+1)k}) - H(\theta_\infty)}{H(\theta_{nk}) - H(\theta_\infty)} &= \limsup_{n\to\infty} \frac{g_{(n+1)k}^T G g_{(n+1)k}}{g_{nk}^T G g_{nk}} \\ &\leq \limsup_{n\to\infty} \frac{b\|g_{(n+1)k}\|^2}{a\|g_{nk}\|^2} = \frac{b}{a} r^{2k}\end{aligned}$$

where $r \in (0, 1)$ is the asymptotic rate of $\theta_n \to \theta_\infty$. By choosing k sufficiently large, the right hand side above is strictly smaller than 1, which gives an upper bound on the rate along the subsequence $H(\theta_{nk})$. A similar argument gives a lower bound strictly larger than 0, which shows that the convergence is not superlinear.

Using Taylor's theorem on $\nabla H(\theta_{nk})$ yields a similar proof for the subsequence $\|\nabla H(\theta_{nk})\|$ – with G replaced by G^2 in the bounds. □

The bounds in the proof are often pessimistic since g_n and g_{n+1} can point in completely different directions. Exercise ?? shows, however, that if θ_n approaches θ_∞ nicely along a fixed direction (making g_n and g_{n+1} almost collinear), $H(\theta_n)$ as well as $\|\nabla H(\theta_n)\|$ inherit linear convergence from θ_n and even with the same rate.

All rates discussed hithereto are *per iteration*, which is natural when investigating a single algorithm. However, different algorithms may spend different amounts of time per iteration, and it does not make sense to make a comparison of per iteration rates across different algorithms. We therefore need to be able to convert between per iteration and per time unit rates. If one iteration takes δ time units (seconds, say) the *per time unit* rate is

$$r^{1/\delta}.$$

If t_n denotes the run time of n iterations, we could estimate δ as t_M / M for M sufficiently large.

We will throughout systematically investigate convergence as a function of time t_n instead of iterations n, and we will estimate rates per time unit directly by least squares regression on t_n instead of n.

7.1.3 Stopping criteria

One important practical question remains unanswered even if we understand the theoretical convergence properties of an algorithm well and have good methods for measuring it convergence rates. No algorithm can run for an infinite number of iterations, thus all algorithms need a criterion for when to stop, but the choice of an appropriate stopping criterion is notoriously difficult. We present four of the most commonly used criteria and discuss benefits and deficits for each of them.

Maximal number of iterations: Stop when

$$n = M$$

for a fixed maximal number of iterations M. This is arguably the simplest criterion, but, obviously, reaching a maximal number of iterations provides no evidence in itself that $H(\theta_M)$ is sufficiently close to a (local) minimum. For a specific problem we could from experience know of a sufficiently large M so that the algorithm has typically converged after M iterations, but the most important use of this criterion is in combination with another criterion so that it works as a safeguard against an infinite loop.

The other three stopping criteria all depend on choosing a *tolerance parameter* $\varepsilon > 0$, which will play different roles in the three criteria. The three criteria can be used individually and in combinations, but unfortunately neither of them nor their combination provide convergence guarantees. It is nevertheless common say that an algorithm "has converged" when the stopping criterion is satisfied, but since none of the criteria are sufficient for convergence this can be a bit misleading.

Small relative change: Stop when

$$\|\theta_n - \theta_{n-1}\| \leq \varepsilon(\|\theta_n\| + \varepsilon).$$

The idea is that when $\theta_n \simeq \theta_{n-1}$ the sequence has approximately reached the limit θ_∞ and we can stop. It is possible to use an absolute criterion such as $\|\theta_n - \theta_{n-1}\| < \varepsilon$, but then the criterion would be sensitive to a rescaling of the parameters. Thus fixing a reasonable tolerance parameter across many problems makes more sense for the relative then the absolute criterion. The main reason for adding ε on the right hand size is to make the criterion well behaved even if $\|\theta_n\|$ is close to zero.

The main benefit of this criterion is that it does not require evaluations of the objective function. Some algorithms, such as the EM-algorithm, do not need to evaluate the objective function, and it may even be difficult to do so. In those cases this criterion can be used. The main deficit is that a single iteration with little change in the parameter can happen for many reasons besides convergence, and it does not imply that neither $\|\theta_n - \theta_\infty\|$ nor $H(\theta_n) - H(\theta_\infty)$ are small.

Small relative descent: Stop when

$$H(\theta_{n-1}) - H(\theta_n) \leq \varepsilon(|H(\theta_n)| + \varepsilon).$$

This criterion only makes sense if the algorithm is a descent algorithm. As discussed above, an absolute criterion would be sensitive to rescaling of the objective function, and the added ε on the right hand side is to ensure a reasonable behavior if $H(\theta_n)$ is close to zero.

This criterion is natural for descent algorithms; we stop when the algorithm does not decrease the value of the objective function sufficiently. The use of a relative criterion makes is possible to choose a tolerance parameter that works reasonably well for many problems. A conventional

choice is $\varepsilon \simeq 10^{-8}$ (and often chosen as the square root of the machine epsilon), though the theoretical support for this choice is weak. The deficit of the algorithm is as for the criterion above: a small descent does not imply that $H(\theta_n) - H(\theta_\infty)$ is small, and it could happen if the algorithm enters a part of Θ where H is very flat, say.

Small gradient: Stop when

$$\|\nabla H(\theta_n)\| \leq \varepsilon.$$

This criterion directly measures if θ_n is close to being a stationary point (and not if θ_n is close to a stationary point). A small value of $\|\nabla H(\theta_n)\|$ is still no guarantee that $\|\theta_n - \theta_\infty\|$ or $H(\theta_n) - H(\theta_\infty)$ are small. The criterion also requires the computation of the gradient. In addition, different norms, $\|\cdot\|$, can be used, and if different coordinates of the gradient are of different orders of magnitude this should be reflected by the norm. Alternatively, the parameters should be rescaled.

Neither of the four criteria above gives a theoretical convergence guarantee in terms of how close $H(\theta_n)$ is to $H(\theta_\infty)$. In some special cases it is possible to develop criteria with a stronger theoretical support. If there is a continuous function $\tilde{H}$ satisfying $H(\theta_\infty) = \tilde{H}(\theta_\infty)$ and

$$H(\theta_\infty) \geq \tilde{H}(\theta)$$

for all $\theta \in \Theta$ (or just in $\text{lev}(\theta_0)$ for a descent algorithm) then

$$0 \leq H(\theta_n) - H(\theta_\infty) \leq H(\theta_n) - \tilde{H}(\theta_n),$$

and the right hand side directly quantifies convergence. Convex duality theory gives for many convex optimization problems such a function, $\tilde{H}$, and in those cases a convergence criterion based on $H(\theta_n) - \tilde{H}(\theta_n)$ actually comes with a theoretical guarantee on how close we are to the minimum. We will not pursue the necessary convex duality theory here, but it is useful to know that in some cases we can do better than the ad hoc criteria above.

7.2 Descent direction algorithms

The negative gradient of H in θ is the direction of steepest descent. Since the goal is to minimize H, it is natural to move away from θ in the direction of $-\nabla H(\theta)$. However, other directions than the negative gradient can also be suitable descent directions.

Definition 7.2. A *descent direction* in θ is a vector $\rho \in \mathbb{R}^p$ such that

$$\nabla H(\theta)^T \rho < 0.$$

When θ is not a stationary point,

$$\nabla H(\theta)^T(-\nabla H(\theta)) = -\|\nabla H(\theta)^T\|_2^2 < 0$$

and $-\nabla H(\theta)^T$ is a descent direction in θ according to the definition.

Given θ_n and a descent direction ρ_n in θ_n we can define

$$\theta_{n+1} = \theta_n + \gamma\rho_n$$

for a suitably chosen $\gamma > 0$. By Taylor's theorem

$$H(\theta_{n+1}) = H(\theta_n) + \gamma\nabla H(\theta_n)\rho_n + o(\gamma),$$

which means that

$$H(\theta_{n+1}) < H(\theta_n)$$

if γ is small enough.

One strategy for choosing γ is to minimize the univariate function

$$\gamma \mapsto H(\theta_n + \gamma\rho_n),$$

which is an example of a *line search* method. Such a minimization would give the maximal possible descent in the direction ρ_n, and as we have argued, if ρ_n is a descent direction, a minimizer $\gamma > 0$ guarantees descent of H. However, unless the minimization can be done analytically it is often computationally too expensive. Less will also do, and as shown in Section 7.1.1, if the Hessian has uniformly bounded numerical radius it is possible to fix one (sufficiently small) step length that will guarantee descent.

7.2.1 Line search

We consider algorithms of the form

$$\theta_{n+1} = \theta_n + \gamma_n\rho_n$$

starting in θ_n and with ρ_n a descent direction in θ_n. The step lengths, γ_n, should be chosen so as to give sufficient descent in each iteration.

The function $h(\gamma) = H(\theta_n + \gamma\rho_n)$ is a univariate and differentiable function,

$$h : [0, \infty) \to \mathbb{R},$$

that gives the value of H in the descent direction ρ_n. We find that

$$h'(\gamma) = \nabla H(\theta_n + \gamma\rho_n)^T\rho_n,$$

and maximal descent in direction ρ_n can be found by solving $h'(\gamma) = 0$ for γ. As mentioned above, less will do. First note that

$$h'(0) = \nabla H(\theta_n)^T\rho_n < 0,$$

so h has a negative slope in 0. It descents in a sufficiently small interval $[0, \varepsilon)$, and it is even true that for any $c \in (0, 1)$ there is an $\varepsilon > 0$ such that

$$h(\gamma) \leq h(0) + c\gamma h'(0)$$

for $\gamma \in [0, \varepsilon)$. We note that this inequality can be checked easily for any given $\gamma > 0$, and is known as the *sufficient descent* condition. Sufficient descent is not enough in itself as the step length could be arbitrarily small, and the algorithm could effectively get stuck.

To prevent too small steps we can enforce another condition. Very close to 0, h will have almost the same slope, $h'(0)$, as it has in 0. If we therefore require that the slope in γ should be larger than $\tilde{c}h'(0)$ for some $\tilde{c} \in (0, 1)$, γ is forced away from 0. This is known as the *curvature condition*.

The combined conditions on γ,

$$h(\gamma) \leq h(0) + c\gamma h'(0)$$

for a $c \in (0, 1)$ and

$$h'(\gamma) \geq \tilde{c}h'(0)$$

for a $\tilde{c} \in (c, 1)$ are known collectively as the *Wolfe conditions*. It can be shown that if h is bounded below there exists a step length satisfying the Wolfe conditions (Lemma 3.1 in Nocedal and Wright (2006)).

Even when choosing γ_n to fulfill the Wolfe conditions there is no guarantee that θ_n will converge let alone converge toward a global minimizer. The best we can hope for in general is that

$$\|\nabla H(\theta_n)\|_2 \to 0$$

for $n \to \infty$, and this will happen under some relatively weak conditions on H (Theorem 3.2 Nocedal and Wright (2006)) under the assumption that

$$\frac{\nabla H(\theta_n)^T \rho_n}{\|\nabla H(\theta_n)\|_2 \|\rho_n\|_2} \leq -\delta < 0.$$

That is, the angle between the descent direction and the gradient should be uniformly bounded away from 90°.

A practical way of searching for a step length is via *backtracking*. Choosing a γ_0 and a constant $d \in (0,1)$ we can search through the sequence of step lengths

$$\gamma_0, d\gamma_0, d^2\gamma_0, d^3\gamma_0, \ldots$$

and stop the first time we find a step length satisfying the Wolfe conditions.

Using backtracking, we can actually dispense of the curvature condition and simply check the sufficient descent condition

$$H(\theta_n + d^k \gamma_0 \rho_n) \leq H(\theta_n) + c d^k \gamma_0 \nabla H(\theta_n)^T \rho_n$$

for $c \in (0,1)$. The implementation of backtracking requires the choice of the three parameters: $\gamma_0 > 0$, $d \in (0,1)$ and $c \in (0,1)$. A good choice depends quite a lot on the algorithm used for choosing the descent direction, but choosing c too close to 1 can make the algorithm take too small steps, and taking d too small can likewise generate small step lengths. Thus $d = 0.8$ or $d = 0.9$ and $c = 0.1$ or even smaller are sensible choices. For some algorithms, like the Newton algorithm to be dealt with below, there is a natural choice of $\gamma_0 = 1$. But for other algorithms a good choice depends crucially on the scale of the parameters, and there is then no general advice on choosing γ_0.

7.2.2 Gradient descent

We implement gradient descent with backtracking below as the function `GD()`. For gradient descent, the sufficient descent condition amounts to choosing the smallest $k \geq 0$ such that

$$H(\theta_n - d^k \gamma_0 \nabla H(\theta_n)) \leq H(\theta_n) - c d^k \gamma_0 \|\nabla H(\theta_n)\|_2^2.$$

The `GD()` function takes the starting point, θ_0, the objective function, H, and its gradient, ∇H, as arguments. The four parameters d, c, γ_0 and ε that control the algorithm can also be specified as additional arguments, but are given some reasonable default values. The implementation uses the *squared* norm of the gradient as a stopping criterion, but it also has a maximal number of iterations as a safeguard. Note that if the maximal number is reached, a warning is printed.

Finally, we include a callback argument (the `cb` argument). If a function is passed to this argument, it will be evaluated in each iteration of the algorithm. This gives us the possibility of logging or printing values of variables during evaluation, which can be highly useful for understanding the inner workings of the algorithm. Monitoring or logging intermediate values during the evaluation of code is referred to as *tracing*.

```
GD <- function(
  par,
  H,
  gr,
  d = 0.8,
  c = 0.1,
  gamma0 = 0.01,
  epsilon = 1e-4,
  maxiter = 1000,
  cb = NULL
) {
  for(i in 1:maxiter) {
    value <- H(par)
    grad <- gr(par)
    h_prime <- sum(grad^2)
    if(!is.null(cb)) cb()
    # Convergence criterion based on gradient norm
    if(h_prime <= epsilon) break
    gamma <- gamma0
    # Proposed descent step
    par1 <- par - gamma * grad
    # Backtracking while descent is insufficient
    while(H(par1) > value - c * gamma * h_prime) {
      gamma <- d * gamma
      par1 <- par - gamma * grad
    }
    par <- par1
  }
  if(i == maxiter)
    warning("Maximal number, ", maxiter, ", of iterations reached")
  par
}
```

We will use the Poisson regression example to illustrate the use of gradient descent and other optimization algorithms, and we need to implement functions in R for computing the negative log-likelihood and its gradient.

The implementation below uses a function factory to produce a list containing a parameter vector, the negative log-likelihood function and its gradient. We anticipate that for the Newton algorithm in Section 7.3 we also need an implementation of the Hessian, which is thus included here as well.

We will exploit the `model.matrix()` function to construct the model matrix from the data via a formula, and the sufficient statistic is also computed. The implementations use linear algebra and vectorized computations relying on access to the model matrix and the sufficient statistics in their enclosing environment.

```
poisson_model <- function(form, data, response) {
  X <- model.matrix(form, data)
  y <- data[[response]]
  # The function drop() drops the dim attribute and turns, for instance,
  # a matrix with one column into a vector
  t_map <- drop(crossprod(X, y))  # More efficient than drop(t(X) %*% y)
```

```
  n <- nrow(X)
  p <- ncol(X)

  H <- function(beta)
    drop(sum(exp(X %*% beta)) - beta %*% t_map) /n

  grad_H <- function(beta)
    (drop(crossprod(X, exp(X %*% beta))) - t_map) / n

  Hessian_H <- function(beta)
    crossprod(X, drop(exp(X %*% beta)) * X) / n

  list(par = rep(0, p), H = H, grad_H = grad_H, Hessian_H = Hessian_H)
}
```

We choose to normalize the log-likelihood by the number of observations n (the number of rows in the model matrix). This does have a small computational cost, but the resulting numerical values become less dependent upon n, which makes it easier to choose sensible default values of various parameters for the numerical optimization algorithms. Gradient descent is very slow for the large Poisson model with individual store effects, so we consider only the simple model with two parameters.

```
veg_pois <- poisson_model(~ log(normalSale), vegetables, response = "sale")
```

```
pois_GD <- GD(veg_pois$par, veg_pois$H, veg_pois$grad_H)
```

The gradient descent implementation is tested by comparing the minimizer to the estimated parameters as computed by `glm()`.

```
rbind(pois_glm = coefficients(pois_model_null), pois_GD)
```

```
##          (Intercept) log(normalSale)
## pois_glm    1.461440       0.9215699
## pois_GD     1.460352       0.9219358
```

We get the same result up to the first two decimals. The convergence criterion on our gradient descent algorithm was quite loose ($\varepsilon = 10^{-4}$, which means that the norm of the gradient is smaller than 10^{-2} when the algorithm stops). This choice of ε in combination with $\gamma_0 = 0.01$ implies that the algorithm stops when the gradient is so small that the changes are at most of norm 10^{-4}.

Comparing the resulting values of the negative log-likelihood shows agreement up to the first five decimals, but we notice that the negative log-likelihood for the parameters fitted using `glm()` is just slightly smaller.

```
veg_pois$H(coefficients(pois_model_null))
veg_pois$H(pois_GD)
```

```
## [1] -124.406827879897
## [1] -124.406825325047
```

To investigate what actually went on inside the gradient descent algorithm we will use the callback argument to trace the internals of a call to `GD()`. The `tracer()` function from the CSwR package can be used to construct a tracer object with a `tracer()` function that we can pass as the callback argument. The tracer object and its `tracer()` function work by storing

information in the enclosing environment of `tracer()`. When used as the callback argument to e.g. `GD()` the `tracer()` function will look up variables in the evaluation environment of `GD()`, store them and print them if requested, and store run time information as well.

When `GD()` has returned, the trace information can be accessed via the `summary()` method for the tracer object. The tracer objects and their `tracer()` function should not be confused with the `trace()` function from the R base package, but the tracer object's `tracer()` function can be passed as the `tracer` argument to `trace()` to interactively inject tracing code into any R function. Here, the tracer objects will only be used together with a callback argument.

We use the tracer object with our gradient descent implementation and print trace information every 50th iteration.

```
GD_tracer <- tracer(c("value", "h_prime", "gamma"), N = 50)
pois_GD <- GD(veg_pois$par, veg_pois$H, veg_pois$grad_H, cb = GD_tracer$tracer)
```

```
## n = 1: value = 1; h_prime = 14268.59; gamma = NA;
## n = 50: value = -123.9395; h_prime = 15.45722; gamma = 0.004096;
## n = 100: value = -124.3243; h_prime = 3.133931; gamma = 0.00512;
## n = 150: value = -124.3935; h_prime = 0.601431; gamma = 0.00512;
## n = 200: value = -124.4048; h_prime = 0.09805907; gamma = 0.00512;
## n = 250: value = -124.4065; h_prime = 0.0109742; gamma = 0.00512;
## n = 300: value = -124.4068; h_prime = 0.002375827; gamma = 0.00512;
## n = 350: value = -124.4068; h_prime = 0.000216502; gamma = 0.004096;
```

We see that the gradient descent algorithm runs for a little more than 350 iterations, and we can observe how the value of the negative log-likelihood is descending. We can also see that the step length γ bounces between $0.004096 = 0.8^4 \times 0.01$ and $0.00512 = 0.8^3 \times 0.01$, thus the backtracking takes 3 to 4 iterations to find a step length with sufficient descent.

The printed trace does not reveal the run time information. The run time is measured for each iteration of the algorithm and the cumulative run time is of greater interest. This information can be computed and inspected after the algorithm has converged, and it is returned by the summary method for tracer objects.

```
tail(summary(GD_tracer))
```

```
##         value      h_prime    gamma      .time
## 372 -124.4068 1.125779e-04 0.005120 0.07250408
## 373 -124.4068 1.218925e-04 0.005120 0.07283077
## 374 -124.4068 1.323878e-04 0.005120 0.07306828
## 375 -124.4068 1.441965e-04 0.005120 0.07330968
## 376 -124.4068 1.574572e-04 0.005120 0.07354185
## 377 -124.4068 7.600796e-05 0.004096 0.07382159
```

The trace information is stored in a list. The summary method transforms the trace information into a data frame with one row per iteration. We can also access individual entries of the list of trace information via subsetting.

```
GD_tracer[377]
```

```
## $value
## [1] -124.4068
##
## $h_prime
## [1] 7.600796e-05
```

```
##
## $gamma
## [1] 0.004096
##
## $.time
## [1] 0.0002797358
```

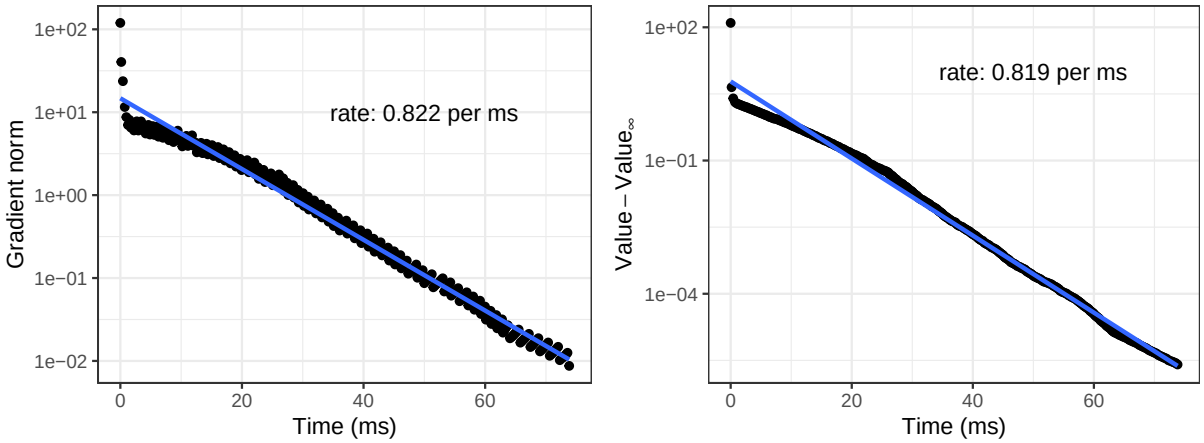


Figure 7.1: Gradient norm (left) and value of the negative log-likelihood (right) above the limit value $H(\theta_\infty)$. The straight line is fitted to the data points except the first ten using least squares, and the rate is computed from this estimate and reported per ms.

7.2.3 Conjugate gradients

The gradient direction is not the best descent direction. It is too local, and convergence can be quite slow. One of the better algorithms that is still a *first order algorithm* (using only gradient information) is the nonlinear conjugate gradient algorithm. In the Fletcher–Reeves version of the algorithm the descent direction is initialized as the negative gradient $\rho_0 = -\nabla H(\theta_0)$ and then updated as

$$\rho_n = -\nabla H(\theta_n) + \frac{\|\nabla H(\theta_n)\|_2^2}{\|\nabla H(\theta_{n-1})\|_2^2}\rho_{n-1}.$$

That is, the descent direction, ρ_n, is the negative gradient but modified according to the previous descent direction. There is plenty of opportunity to vary the prefactor of ρ_{n-1}, and the one presented here is what makes it the Fletcher–Reeves version. Other versions go by the names of their inventors such as Polak–Ribière or Hestenes–Stiefel.

In fact, ρ_n need not be a descent direction unless we put some restrictions on the step lengths. One possibility is to require that the step length γ_n satisfies the *strong* curvature condition

$$|h'(\gamma)| = |\nabla H(\theta_n + \gamma\rho_n)^T\rho_n| \leq \tilde{c}|\nabla H(\theta_n)^T\rho_n| = \tilde{c}|h'(0)|$$

for a $\tilde{c} < \frac{1}{2}$. Then ρ_{n+1} can be shown to be a descent direction if ρ_n is.

We implement the conjugate gradient method in a slightly different way. Instead of introducing the more advanced curvature condition, we simply reset the algorithm to use the gradient direction in any case where a non-descent direction has been chosen. Resets of descent direction every p-th iteration is recommended anyway for the nonlinear conjugate gradient algorithm.

```
CG <- function(
  par,
  H,
  gr,
  d = 0.8,
  c = 0.1,
  gamma0 = 1,
  epsilon = 1e-6,
  maxiter = 1000,
  cb = NULL
) {
  p <- length(par)
  m <- 1
  rho0 <- numeric(p)
  for(i in 1:maxiter) {
    value <- H(par)
    grad <- gr(par)
    grad_norm_sq <- sum(grad^2)
    if(!is.null(cb)) cb()
    if(grad_norm_sq <= epsilon) break
    gamma <- gamma0
    # Descent direction
    rho <- - grad + grad_norm_sq * rho0
    h_prime <- drop(t(grad) %*% rho)
    # Reset to gradient descent if m > p or rho is not a descent direction
    if(m > p || h_prime >= 0) {
      rho <- - grad
      h_prime <- - grad_norm_sq
      m <- 1
    }
    par1 <- par + gamma * rho
    # Backtracking
    while(H(par1) > value + c * gamma * h_prime) {
      gamma <- d * gamma
      par1 <- par + gamma * rho
    }
    rho0 <- rho / grad_norm_sq
    par <- par1
    m <- m + 1
  }
  if(i == maxiter)
    warning("Maximal number, ", maxiter, ", of iterations reached")
  par
}
```

```
CG_tracer <- tracer(c("value", "gamma", "grad_norm_sq"), N = 10)
pois_CG <- CG(veg_pois$par, veg_pois$H, veg_pois$grad_H, cb = CG_tracer$tracer)
```

```
## n = 1: value = 1; gamma = NA; grad_norm_sq = 14269;
## n = 10: value = -123.15; gamma = 0.018014; grad_norm_sq = 129.78;
## n = 20: value = -123.92; gamma = 0.022518; grad_norm_sq = 77.339;
```

```
## n = 30: value = -124.23; gamma = 0.018014; grad_norm_sq = 22.227;
## n = 40: value = -124.41; gamma = 0.0092234; grad_norm_sq = 0.179;
## n = 50: value = -124.41; gamma = 0.10737; grad_norm_sq = 0.028232;
## n = 60: value = -124.41; gamma = 0.0092234; grad_norm_sq = 0.00021747;
## n = 70: value = -124.41; gamma = 0.0092234; grad_norm_sq = 1.7488e-06;
```

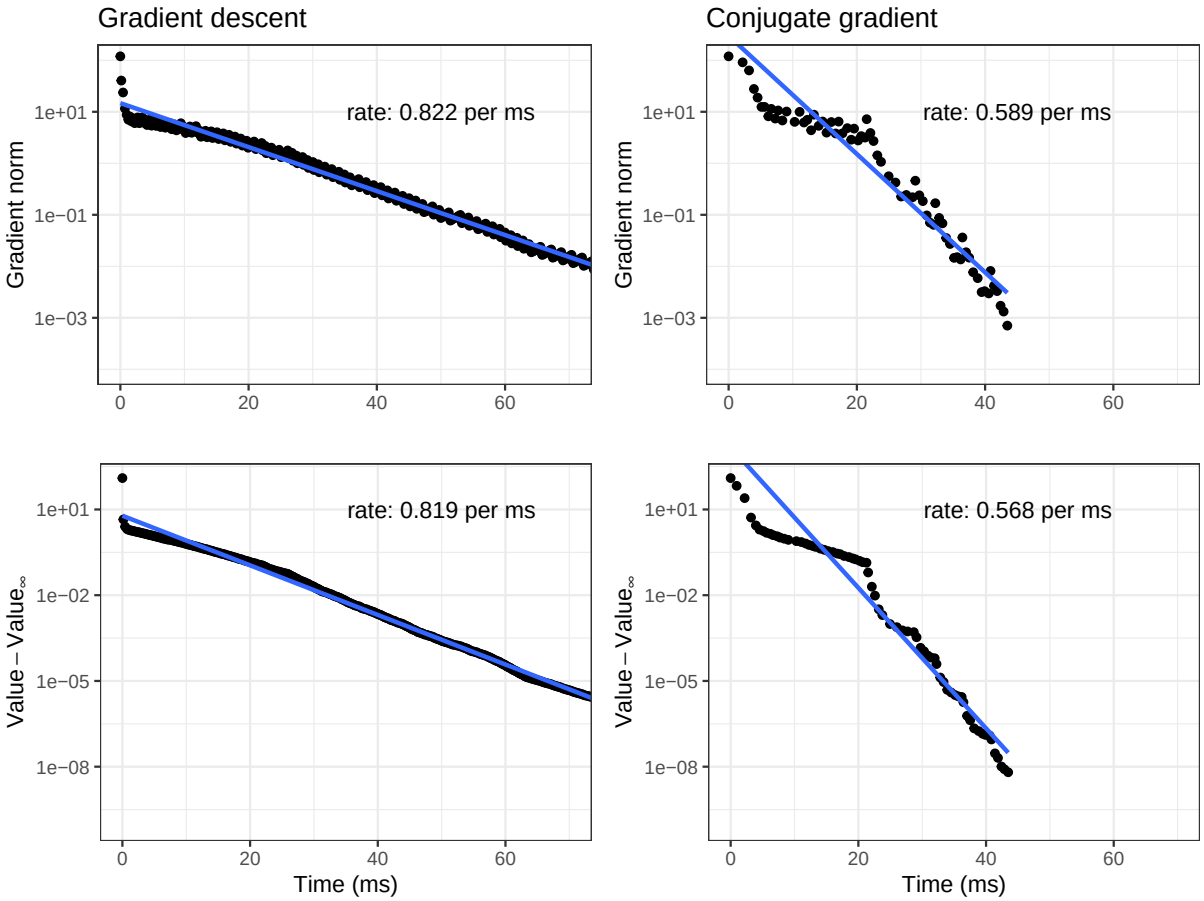


Figure 7.2: Gradient norms (top) and negative log-likelihoods (bottom) for gradient descent (left) and conjugate gradient (right).

This algorithm is fast enough to fit the large Poisson regression model.

```
veg_pois <- poisson_model(~ store + log(normalSale) - 1, vegetables, response = "sale")
```

```
CG_tracer <- tracer(c("value", "gamma", "grad_norm_sq"), N = 100)
pois_CG <- CG(veg_pois$par, veg_pois$H, veg_pois$grad_H, cb = CG_tracer$tracer)
```

```
## n = 1: value = 1; gamma = NA; grad_norm_sq = 12737;
## n = 100: value = -127.9; gamma = 0.018014; grad_norm_sq = 1.676;
## n = 200: value = -128.28; gamma = 0.011529; grad_norm_sq = 2.5128;
## n = 300: value = -128.55; gamma = 0.0037779; grad_norm_sq = 0.068176;
## n = 400: value = -128.59; gamma = 0.022518; grad_norm_sq = 0.0028747;
## n = 500: value = -128.59; gamma = 0.0092234; grad_norm_sq = 0.0652;
## n = 600: value = -128.59; gamma = 0.018014; grad_norm_sq = 0.00020555;
## n = 700: value = -128.59; gamma = 0.0019343; grad_norm_sq = 5.3952e-06;
## n = 800: value = -128.59; gamma = 0.005903; grad_norm_sq = 3.7118e-06;
## n = 900: value = -128.59; gamma = 0.0037779; grad_norm_sq = 1.0621e-06;
```

```
tail(summary(CG_tracer))
```

```
##         value       gamma grad_norm_sq    .time
## 899 -128.5894 0.005902958 5.214667e-06 20.32074
## 900 -128.5894 0.003777893 1.062092e-06 20.34231
## 901 -128.5894 0.068719477 2.119915e-05 20.35352
## 902 -128.5894 0.107374182 2.047029e-04 20.36509
## 903 -128.5894 0.004722366 7.056181e-06 20.38598
## 904 -128.5894 0.003777893 8.881615e-07 20.40910
```

Using `optim()` with the conjugate gradient method.

```
system.time(
  pois_optim_CG <- optim(
    veg_pois$par,
    veg_pois$H,
    veg_pois$grad_H,
    method = "CG",
    control = list(maxiter = 10000)
  )
)
```

```
## Warning in optim(veg_pois$par, veg_pois$H, veg_pois$grad_H, method = "CG", :
## unknown names in control: maxiter
```

```
##    user  system elapsed
##   0.330   0.008   0.347
```

```
pois_optim_CG[c("value", "counts")]
```

```
## $value
## [1] -127.3596
##
## $counts
## function gradient
##      237      101
```

7.2.4 Peppered Moths

Returning to the peppered moth from Section 6.2.1 we implemented in that section the log-likelihood for general multinomial cell collapsing and applied the implementation to compute the maximum-likelihood estimate. In this section we implement the gradient as well. From the expression for the log-likelihood in (6.2) it follows that the gradient equals

$$\nabla \ell(\theta) = \sum_{j=1}^{K_0} \frac{x_j}{M(p(\theta))_j} \nabla M(p(\theta))_j = \sum_{j=1}^{K_0} \sum_{k \in A_j} \frac{x_j}{M(p(\theta))_j} \nabla p_k(\theta).$$

Letting $j(k)$ be defined by $k \in A_{j(k)}$ we see that the gradient can also be written as

$$\nabla \ell(\theta) = \sum_{k=1}^{K} \frac{x_{j(k)}}{M(p(\theta))_{j(k)}} \nabla p_k(\theta) = \tilde{\mathbf{x}}(\theta) \mathrm{D} p(\theta),$$

where $Dp(\theta)$ is the Jacobian of the parametrization $\theta \mapsto p(\theta)$, and $\tilde{\mathbf{x}}(\theta)$ is the vector with

$$\tilde{\mathbf{x}}(\theta)_k = \frac{x_{j(k)}}{M(p(\theta))_{j(k)}}.$$

```
grad_loglik <- function(par, x, prob, Dprob, group) {
  p <- prob(par)
  if(is.null(p)) return(rep(NA, length(par)))
  - (x[group] / M(p, group)[group]) %*% Dprob(par)
}
```

The Jacobian needs to be implemented for the specific example of peppered moths.

```
Dprob <- function(p) {
  p[3] <- 1 - p[1] - p[2]
  matrix(
    c(2 * p[1],             0,
      2 * p[2],             2 * p[1],
      2* p[3] - 2 * p[1],  -2 * p[1],
      0,                    2 * p[2],
      -2 * p[2],            2 * p[3] - 2 * p[2],
      -2 * p[3],           -2 * p[3]),
    ncol = 2, nrow = 6, byrow = TRUE)
}
```

We can then use the conjugate gradient algorithm to compute the maximum-likelihood estimate.

```
optim(c(0.3, 0.3), loglik, grad_loglik, x = c(85, 196, 341),
      prob = prob, Dprob = Dprob, group = c(1, 1, 1, 2, 2, 3),
      method = "CG")
```

```
## $par
## [1] 0.07083691 0.18873652
##
## $value
## [1] 600.481
##
## $counts
## function gradient
##       92       19
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The peppered Moth example is very simple. The log-likelihood can easily be computed, and we used this problem to illustrate ways of implementing a likelihood in R and how to use `optim` to maximize it.

One of the likelihood implementations was very problem specific while the other more abstract and general, and we used the same general and abstract approach to implement the gradient above. The gradient could then be used for other optimization algorithms, still using `optim`,

such as conjugate gradient. In fact, you can use conjugate gradient without computing and implementing the gradient.

```
optim(c(0.3, 0.3), loglik, x = c(85, 196, 341),
      prob = prob, group = c(1, 1, 1, 2, 2, 3),
      method = "CG")
```

```
## $par
## [1] 0.07084109 0.18873718
##
## $value
## [1] 600.481
##
## $counts
## function gradient
##      107       15
##
## $convergence
## [1] 0
##
## $message
## NULL
```

If we do not implement a gradient, a numerical gradient is used by `optim()`. This can result in a slower algorithm than if the gradient is implemented, but more seriously, in can result in convergence problems. This is because there is a subtle tradeoff between numerical accuracy and accuracy of the finite difference approximation used to approximate the gradient. We did not experience convergence problems in the example above, but one way to remedy such problems is to set the `parscale` or `fnscale` entries in the `control` list argument to `optim()`.

In the following chapter the peppered moth example is used to illustrate the EM algorithm. It is important to understand that the EM algorithm does not rely on the ability to compute the likelihood or the gradient of the likelihood for that matter. In many real applications of the EM algorithm the computation of the likelihood is challenging or even impossible, thus most standard optimization algorithms will not be directly applicable.

7.3 Newton-type algorithms

The Newton algorithm is very similar to gradient descent except that the gradient descent direction is replaced by

$$\rho_n = -D^2 H(\theta_n)^{-1} \nabla H(\theta_n).$$

The Newton algorithm is typically much more efficient than gradient descent and will converge in few iterations. However, the storage of the $p \times p$ Hessian, its computation, and the solution of the equation to compute ρ_n all scale like p^2 and this can make the algorithm useless for very large p.

A variety of alternatives to the Newton algorithm exist that replace the Hessian by another matrix that can be easier to compute and update. It should be noted that if we choose a matrix B_n in the n-th iteration, then $-B_n \nabla H(\theta_n)$ is a descent direction whenever B_n is a positive definite matrix.

Newton implementation (with trace).

```
Newton <- function(
  par,
  H,
  gr,
  hess,
  d = 0.8,
  c = 0.1,
  gamma0 = 1,
  epsilon = 1e-10,
  maxiter = 50,
  cb = NULL
) {
  for(i in 1:maxiter) {
    value <- H(par)
    grad <- gr(par)
    if(!is.null(cb)) cb()
    if(sum(grad^2) <= epsilon) break
    Hessian <- hess(par)
    rho <- - drop(solve(Hessian, grad))
    gamma <- gamma0
    par1 <- par + gamma * rho
    h_prime <- t(grad) %*% rho
    while(H(par1) > value +  c * gamma * h_prime) {
      gamma <- d * gamma
      par1 <- par + gamma * rho
    }
    par <- par1
  }
  if(i == maxiter)
    warning("Maximal number, ", maxiter, ", of iterations reached")
  par
}
```

7.3.1 Poisson regression

We use the implementation of the Hessian matrix.

```
Newton_tracer <- tracer(c("value", "h_prime", "gamma"), N = 0)
pois_Newton <- Newton(
  veg_pois$par,
  veg_pois$H,
  veg_pois$grad_H,
  veg_pois$Hessian_H,
  cb = Newton_tracer$tracer
)
```

```
range(pois_Newton - coefficients(pois_model))
```

```
## [1] -4.979266e-10  1.298776e-06
```

```
rbind(
  pois_Newton = veg_pois$H(pois_Newton),
```

```
  pois_glm = veg_pois$H(coefficients(pois_model))
)
```

```
##                               [,1]
## pois_Newton -128.58945047446991339
## pois_glm    -128.58945047447093657
```

```
summary(Newton_tracer)
```

```
##          value       h_prime    gamma     .time
## 1     1.00000            NA       NA 0.0000000
## 2   -14.83270 -4.140563e+03 0.022518 0.2876197
## 3   -64.81635 -4.029847e+02 0.262144 0.5262933
## 4  -111.33647 -7.636275e+01 1.000000 0.7261856
## 5  -124.24937 -2.104160e+01 1.000000 0.9317900
## 6  -127.71116 -5.652483e+00 1.000000 1.1241905
## 7  -128.49729 -1.332600e+00 1.000000 1.3132812
## 8  -128.58733 -1.647034e-01 1.000000 1.5046009
## 9  -128.58945 -4.159696e-03 1.000000 1.7346026
## 10 -128.58945 -3.288913e-06 1.000000 1.9876618
```

The R function `glm.fit()` uses a Newton algorithm (without backtracking) and is about a factor five faster on this example.

```
env_pois <- environment(veg_pois$H)
system.time(glm.fit(env_pois$X, env_pois$y, family = poisson()))
```

```
##    user  system elapsed
##   0.506   0.022   0.593
```

One should be careful when comparing run times for different optimization algorithms, but in this case they have achieved about the same precision with the faster `glm.fit()` that even obtained the smallest negative log-likelihood value of the two.

7.3.2 Quasi-Newton algorithms

We turn to other descent direction algorithms that are more efficient than gradient descent by choosing the descent direction in a more clever way but less computationally demanding than the Newton algorithm that requires the computation of the full Hessian in each iteration.

We will only consider the application of the BFGS algorithm via the implementation in the R function `optim()`.

```
system.time(
  pois_BFGS <- optim(
    veg_pois$par,
    veg_pois$H,
    veg_pois$grad_H,
    method = "BFGS",
    control = list(maxiter = 10000)
  )
)
```

```
## Warning in optim(veg_pois$par, veg_pois$H, veg_pois$grad_H, method = "BFGS", :
## unknown names in control: maxiter
```

```
##    user  system elapsed
##   0.369   0.005   0.388
```

```
range(pois_BFGS$par - coefficients(pois_model))
```

```
## [1] -0.2638641  0.6364195
```

```
pois_BFGS[c("value", "counts")]
```

```
## $value
## [1] -128.5888
##
## $counts
## function gradient
##      104      100
```

7.3.3 Sparsity

One of the benefits of the implementations of H and its derivatives as well as of the descent algorithms is that they can exploit sparsity of $\mathbf{X}$ almost for free. The implementations have not done that in previous computations, because $\mathbf{X}$ has been stored as a dense matrix. In reality, $\mathbf{X}$ is a very sparse matrix (the vast majority of the matrix entries are zero), and if we convert it into a sparse matrix, all the matrix-vector products will be more run time efficient. Sparse matrices are implemented in the R package Matrix.

```
library(Matrix)
```

```
env_pois$X <- Matrix(env_pois$X)
class(env_pois$X)
```

```
## [1] "dgCMatrix"
## attr(,"package")
## [1] "Matrix"
```

Without changing any other code, we get an immediate run time improvement using e.g. `optim()` and the BFGS algorithm.

```
system.time(
  pois_BFGS_sparse <- optim(
    veg_pois$par,
    veg_pois$H,
    veg_pois$grad_H,
    method = "BFGS",
    control = list(maxiter = 10000)
  )
)
```

```
## Warning in optim(veg_pois$par, veg_pois$H, veg_pois$grad_H, method = "BFGS", :
## unknown names in control: maxiter
```

```
##    user  system elapsed
##   0.148   0.003   0.159
```

We should in real applications avoid constructing a dense intermediate model matrix as a step toward constructing a sparse model matrix. This is possible by constructing the sparse model matrix directly using a function from the R package MatrixModels. Ideally, we should

reimplement `pois_model()` to support an option for using sparse matrices, but to focus on the run time benefits of sparse matrices, we simply change the matrix in the appropriate environment directly.

```
env_pois$X <- MatrixModels::model.Matrix(
  ~ store + log(normalSale) - 1,
  data = vegetables,
  sparse = TRUE
)
class(env_pois$X)
```

```
## [1] "dsparseModelMatrix"
## attr(,"package")
## [1] "MatrixModels"
```

The Newton implementation benefits enormously from using sparse matrices because the bottleneck is the computation of the Hessian.

```
Newton_tracer <- tracer(c("value", "h_prime", "gamma"), N = 0)
  pois_Newton <- Newton(
  veg_pois$par,
  veg_pois$H,
  veg_pois$grad_H,
  veg_pois$Hessian_H,
  cb = Newton_tracer$tracer
)
summary(Newton_tracer)
```

```
##          value       h_prime    gamma       .time
## 1     1.00000            NA       NA 0.00000000
## 2   -14.83270 -4.140563e+03 0.022518 0.01121643
## 3   -64.81635 -4.029847e+02 0.262144 0.01964024
## 4  -111.33647 -7.636275e+01 1.000000 0.02380496
## 5  -124.24937 -2.104160e+01 1.000000 0.02923057
## 6  -127.71116 -5.652483e+00 1.000000 0.03485172
## 7  -128.49729 -1.332600e+00 1.000000 0.03948507
## 8  -128.58733 -1.647034e-01 1.000000 0.04417199
## 9  -128.58945 -4.159696e-03 1.000000 0.04727571
## 10 -128.58945 -3.288913e-06 1.000000 0.05146527
```

To summarize the run times we have measured for the Poisson regression example, we found that the conjugate gradient algorithms took of the order of 10 seconds to converge. The Newton-type algorithms in this section were faster and took between 0.3 and 1.7 seconds to converge. The use of sparse matrices reduced the run time of the quasi-Newton algorithm BFGS by a factor 3, but it reduced the run time of the Newton algorithm by a factor 50 to about 0.03 seconds. One could be concerned that the construction of the sparse model matrix takes more time (which we did not measure), but if measured it turns out that for this example it takes about the same time to construct the dense model matrix as it takes to construct the sparse one.

Run time efficiency is not the only argument for using sparse matrices as they are also more memory efficient. It is memory (and time) inefficient to use dense intermediates, and for truly large scale problems impossible. Using sparse model matrices for regression models allows us to work with larger models that have more variables, more factor levels and more observations

than if we use dense model matrices. For the Poisson regression model the memory used by either representation can be found.

```
object.size(env_pois$X)
object.size(as.matrix(env_pois$X))
```

```
## Sparse matrix memory usage:
## 123440 bytes
## Dense matrix memory usage:
## 3103728 bytes
```

We see that the dense matrix uses around a factor 30 more memory than the sparse representation. In this case it means using around 3 MB for storing the dense matrix instead of around 100 kB, which won't be a problem on a contemporary computer. However, going from using 3 GB for storing a matrix to using 100 Mb could be the difference between not being able to work with the matrix on a standard laptop to running the computations with no problems. Using `model.Matrix` makes it possible to construct sparse model matrices directly and avoid all dense intermediates.

The function `glm4()` from the MatrixModels package for fitting regression models, can exploit sparse model matrices direction, and can thus be useful in cases where your model matrix becomes very large but sparse. There are two main applications where the model matrix becomes sparse. When you model the response using one or more factors, and possibly their interactions, the model matrix will become particularly sparse if the factors have many levels. Another case is when you model the response via basis expansions of quantitative predictors and use basis functions with local support. The B-splines form an important example of such a basis with local support that results in a sparse model matrix.

7.4 Misc.

If Φ is just *nonexpansive* (the constant c above is one), this is no longer true, but replacing Φ by $\alpha\Phi + (1-\alpha)I$ for $\alpha \in (0,1)$ we get Krasnoselskii-Mann iterates of the form

$$\theta_n = \alpha\Phi(\theta_{n-1}) + (1-\alpha)\theta_{n-1}$$

that will converge to a fixed point of Φ provided it has one.

Banach's fixed point theorem implies a convergence that is at least as fast as linear convergence with asymptotic rate c. Moreover, if Φ is just a contraction for $n \geq n_0$ for some n_0, then for $n \geq n_0$

$$\|\theta_{n+1} - \theta_n\| \leq c\|\theta_n - \theta_{n-1}\|.$$

The convergence may be superlinear, but if it is linear, the rate is bounded by c. To indicate if Φ is asymptotically a contraction, we can introduce

$$R_n = \frac{\|\theta_{n+1} - \theta_n\|}{\|\theta_n - \theta_{n-1}\|}$$

and monitor its behavior as $n \to \infty$. The constant

$$r = \limsup_{n\to\infty} R_n$$

is then asymptotically the smallest possible contraction constant. If convergence is sublinear and $R_n \to 1$ the sequence is called logarithmically convergent (by definition), while $r \in (0,1)$

is an indication of linear convergence with rate r, and $r = 0$ is an indication of superlinear convergence.

In practice, we can plot the ratio R_n as the algorithm is running, and use R_n as an estimate of the rate r for large n. However, this can be a quite unstable method for estimating the rate.

Finally, it is also possible to estimate the order q as well as the rate r by using that

$$\log \|\theta_n - \theta_N\| \simeq q \log \|\theta_{n-1} - \theta_N\| + \log(r)$$

for $n \geq n_0$. We can estimate q and $\log(r)$ by fitting a linear function by least squares to these log-log transformed norms of errors.

Iteration, fixed points, convergence criteria. Ref to Nonlinear Parameter Optimization Using R Tools.

Chapter 8

Expectation maximization algorithms

Somewhat surprisingly, it is possible to develop an algorithm, known as the *expectation-maximization algorithm*, for computing the maximizer of a likelihood function in situations where computing the likelihood itself is quite difficult. This is possible in situations where the model is defined in terms of certain unobserved components, and where likelihood computations and optimization is relatively easy had we had the complete observation. The EM algorithm exploits this special structure, and is thus not a general optimization algorithm, but the situation where it applies is common enough in statistics that it is one of the core optimization algorithms used for computing maximum-likelihood estimates.

In this chapter it is shown that the algorithm is generally an descent algorithm of the negative log-likelihood, and examples of its implementation are given to multinomial cell collapsing and Gaussian mixtures. The theoretical results needed for the EM algorithm for a special case of mixed models are given as well. Finally, some theoretical results as well as practical implementations for computing estimates of the Fisher information are presented.

8.1 Basic properties

In this section the EM algorithm is formulated and shown to be a descent algorithm for the negative log-likelihood. Allele frequency estimation for the peppered moth is considered as a simple example showing how the algorithm can be implemented.

8.1.1 Incomplete data likelihood

Suppose that Y is a random variable and $X = M(Y)$. Suppose that Y has density $f(\cdot \mid \theta)$ and that X has marginal density $g(x \mid \theta)$.

The marginal density is typically of the form

$$g(x \mid \theta) = \int_{\{y : M(y) = x\}} f(y \mid \theta) \, \mu_x(\mathrm{d}y)$$

for a suitable measure μ_x depending on M and x but not θ. The general argument for the marginal density relies on the coarea formula.

The log-likelihood for observing $X = x$ is

$$\ell(\theta) = \log g(x \mid \theta).$$

The log-likelihood is often impossible to compute analytically and difficult and expensive to compute numerically. The complete log-likelihood, $\log f(y \mid \theta)$, is often easy to compute, but we don't know Y, only that $M(Y) = x$.

In some cases it is possible to compute

$$Q(\theta \mid \theta') := E_{\theta'}(\log f(Y \mid \theta) \mid X = x),$$

which is the conditional expectation of the complete log-likelihood given the observed data and computed using the probability measure given by θ'. Thus for fixed θ' this is a computable function of θ depending only on the observed data x.

One could get the following idea: with an initial guess of $\theta' = \theta_0$ compute iteratively

$$\theta_{n+1} = \arg\max Q(\theta \mid \theta_n)$$

for $n = 0, 1, 2, \ldots$. This idea is the EM algorithm:

- **E-step**: Compute the conditional expectation $Q(\theta \mid \theta_n)$.
- **M-step**: Maximize $\theta \mapsto Q(\theta \mid \theta_n)$.

It is a bit weird to present the algorithm as a two-step algorithm in its abstract formulation. Even though we can regard $Q(\theta \mid \theta_n)$ as something we can compute abstractly for each θ for a given θ_n, the maximization is in practice not really done using all these evaluations. It is computed either by an analytic formula involving x and θ_n, or by a numerical algorithm that computes certain evaluations of $Q(\cdot \mid \theta_n)$ and perhaps its gradient and Hessian. In computing these specific evaluations there is, of course, a need for the computation of conditional expectations, but we would compute these as they are needed and not upfront.

However, in some of the most important applications of the EM algorithm, particularly for exponential families covered in Section 8.2, it makes a lot of sense to regard the algorithm as a two-step algorithm. This is the case whenever $Q(\theta \mid \theta_n) = q(\theta, t(x, \theta_n))$ is given in terms of θ and a function $t(x, \theta_n)$ of x and θ_n that doesn't depend on θ. Then the E-step becomes the computation of $t(x, \theta_n)$, and in the M-step, $Q(\cdot \mid \theta_n)$ is maximized by maximizing $q(\cdot, t(x, \theta_n))$, and the maximum is a function of $t(x, \theta_n)$.

8.1.2 Monotonicity of the EM algorithm

We prove below that the algorithm (weakly) increases the log-likelihood in every step, and thus is a descent algorithm for the negative log-likelihood $H = -\ell$.

It holds in great generality that the conditional distribution of Y given $X = x$ has density

$$h(y \mid x, \theta) = \frac{f(y \mid \theta)}{g(x \mid \theta)} \tag{8.1}$$

w.r.t. the measure μ_x as above (that does not depend upon θ), and where g is the density for the marginal distribution.

This can be verified quite easily for discrete distributions and when $Y = (Z, X)$ with joint density w.r.t. a product measure $\mu \otimes \nu$ that does not depend upon θ. In the latter case, $f(y \mid \theta) = f(z, x \mid \theta)$ and

$$g(x \mid \theta) = \int f(z, x \mid \theta)\, \mu(\mathrm{d}z)$$

is the marginal density w.r.t. ν.

Whenever (8.1) holds it follows that

$$\ell(\theta) = \log g(x \mid \theta) = \log f(y \mid \theta) - \log h(y \mid x, \theta),$$

where $\ell(\theta)$ is the log-likelihood.

Theorem 8.1. *If* $\log f(Y \mid \theta)$ *as well as* $\log h(Y \mid x, \theta)$ *have finite* θ'*-conditional expectation given* $M(Y) = x$ *then*

$$Q(\theta \mid \theta') > Q(\theta' \mid \theta') \quad \Rightarrow \quad \ell(\theta) > \ell(\theta').$$

Proof. Since $\ell(\theta)$ depends on y only through $M(y) = x$,

$$\begin{aligned}
\ell(\theta) &= E_{\theta'}(\ell(\theta) \mid X = x) \\
&= \underbrace{E_{\theta'}(\log f(Y \mid \theta) \mid X = x)}_{Q(\theta|\theta')} + \underbrace{E_{\theta'}(-\log h(Y \mid x, \theta) \mid X = x)}_{H(\theta|\theta')} \\
&= Q(\theta \mid \theta') + H(\theta \mid \theta').
\end{aligned}$$

Now for the second term we find, using Jensen's inequality for the convex function $-\log$, that

$$\begin{aligned}
H(\theta \mid \theta') &= \int -\log(h(y \mid x, \theta)) h(y \mid x, \theta') \mu_x(\mathrm{d}y) \\
&= \int -\log\left(\frac{h(y \mid x, \theta)}{h(y \mid x, \theta')}\right) h(y \mid x, \theta') \mu_x(\mathrm{d}y) \\
&\quad + \int -\log(h(y \mid x, \theta')) h(y \mid x, \theta') \mu_x(\mathrm{d}y) \\
&\geq -\log\left(\int \frac{h(y \mid x, \theta)}{h(y \mid x, \theta')} h(y \mid x, \theta') \mu_x(\mathrm{d}y)\right) + H(\theta' \mid \theta') \\
&= -\log\Big(\underbrace{\int h(y \mid x, \theta) \mu_x(\mathrm{d}y)}_{=1}\Big) + H(\theta' \mid \theta') \\
&= H(\theta' \mid \theta').
\end{aligned}$$

From this we see that

$$\ell(\theta) \geq Q(\theta \mid \theta') + H(\theta' \mid \theta')$$

for all θ and the right hand side is a so-called minorant for the log-likelihood. Observing that

$$\ell(\theta') = Q(\theta' \mid \theta') + H(\theta' \mid \theta')$$

completes the proof of the theorem. □

Note that the proof above can also be given by referring to Gibbs' inequality in information theory stating that the Kullback-Leibler divergence is positive, or equivalently that the cross-entropy $H(\theta \mid \theta')$ is smaller than the entropy $H(\theta' \mid \theta')$, but the proof of this is, in itself, a consequence of Jensen's inequality just as above.

It follows from Theorem 8.1 that if θ_n is computed iteratively starting from θ_0 such that

$$Q(\theta_{n+1} \mid \theta_n) > Q(\theta_n \mid \theta_n),$$

then

$$H(\theta_0) > H(\theta_1) > H(\theta_2) > \dots.$$

This proves that the EM algorithm is a strict descent algorithm for the negative log-likelihood as long as it is possible in each iteration to find a θ such that $Q(\theta \mid \theta_n) > Q(\theta_n \mid \theta_n)$.

The term *EM algorithm* is reserved for the specific algorithm that maximizes $Q(\cdot \mid \theta_n)$ in the M-step, but there is no reason to insist on the M-step being a maximization. A choice of ascent direction of $Q(\cdot \mid \theta_n)$ and a step-length guaranteeing sufficient descent of H (sufficient ascent of $Q(\cdot \mid \theta_n)$) will be enough to give a descent algorithm. Any such variation is usually termed a generalized EM algorithm.

We could imagine that the minorant is a useful lower bound on the difficult-to-compute log-likelihood. The additive constant $H(\theta' \mid \theta')$ in the minorant is, however, not going to be computable in general either, and it is not clear that there is any way to use the bound quantitatively.

8.1.3 Peppered moths

We return in this section to the peppered moths and the implementation of the EM algorithm for multinomial cell collapsing.

The EM algorithm can be implemented by two simple functions that compute the conditional expectations (the E-step) and then maximization of the complete observation log-likelihood.

```
EStep0 <- function(p, x, group) {
  x[group] * p / M(p, group)[group]
}
```

The MLE of the complete log-likelihood is a linear estimator, as is the case in many examples with explicit MLEs.

```
MStep0 <- function(n, X)
  as.vector(X %*% n / (sum(n)))
```

The `EStep0` and `MStep0` functions are abstract implementations. They require specification of the arguments `group` and `X`, respectively, to become concrete.

The M-step is only implemented in the case where the complete-data MLE is a *linear estimator*, that is, a linear map of the complete data vector y that can be expressed in terms of a matrix $\mathbf{X}$.

```
EStep <- function(par, x)
  EStep0(prob(par), x, c(1, 1, 1, 2, 2, 3))

MStep <- function(n) {
  X <- matrix(
  c(2, 1, 1, 0, 0, 0,
    0, 1, 0, 2, 1, 0) / 2,
  2, 6, byrow = TRUE)

  MStep0(n, X)
}
```

The EM algorithm is finally implemented as an iterative, alternating call of `EStep` and `MStep` until convergence as measured in terms of the relative change from iteration to iteration being sufficiently small.

```
EM <- function(par, x, epsilon = 1e-6, trace = NULL) {
  repeat{
    par0 <- par
    par <- MStep(EStep(par, x))
    if(!is.null(trace)) trace()
    if(sum((par - par0)^2) <= epsilon * (sum(par^2) + epsilon))
      break
  }
  par  ## Remember to return the parameter estimate
}

phat <- EM(c(0.3, 0.3), c(85, 196, 341))
phat
```

```
## [1] 0.07083693 0.18877365
```

We check what is going on in each step of the EM algorithm.

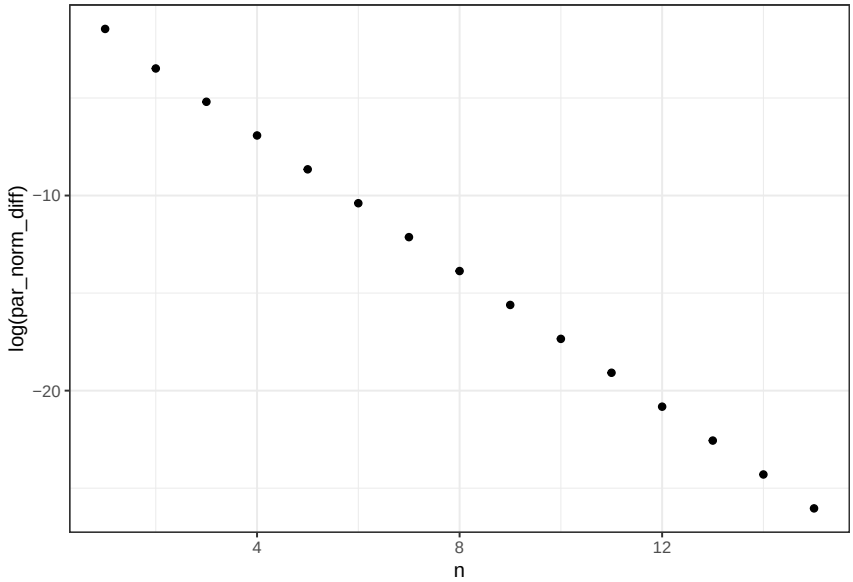
```
EM_tracer <- tracer("par")
EM(c(0.3, 0.3), c(85, 196, 341), trace = EM_tracer$tracer)
```

```
## n = 1: par = 0.08038585, 0.22464192;
## n = 2: par = 0.07118928, 0.19546961;
## n = 3: par = 0.07084985, 0.18993393;
## n = 4: par = 0.07083738, 0.18894757;
## n = 5: par = 0.07083693, 0.18877365;
```

```
## [1] 0.07083693 0.18877365
```

```
EM_tracer <- tracer(c("par0", "par"), N = 0)
phat <- EM(c(0.3, 0.3), c(85, 196, 341), epsilon = 1e-20,
           trace = EM_tracer$tracer)
```

```
EM_trace <- summary(EM_tracer)
  EM_trace <- transform(
  EM_trace,
  n = 1:nrow(EM_trace),
  par_norm_diff = sqrt((par0.1 - par.1)^2 + (par0.2 - par.2)^2)
)
qplot(n, log(par_norm_diff), data = EM_trace)
```



Note the log-axis. The EM-algorithm converges linearly (this is the terminology, see Algorithms and Convergence). The log-rate of the convergence can be estimated by least-squares.

```
log_rate_fit <- lm(log(par_norm_diff) ~ n,  data = EM_trace)
exp(coefficients(log_rate_fit)["n"])
```

```
##         n
## 0.1750251
```

The rate is very small in this case implying fast convergence. This is not always the case. If the log-likelihood is flat, the EM-algorithm can become quite slow with a rate close to 1.

8.2 Exponential families

We consider in this section the special case where the model of $\mathbf{y}$ is given as an exponential family Bayesian network as in Section 6.1.2 and $x = M(\mathbf{y})$ is the observed transformation.

The complete data log-likelihood is

$$\theta \mapsto \theta^T t(\mathbf{y}) - \kappa(\theta) = \theta^T \sum_{j=1}^{m} t_j(y_j) - \kappa(\theta),$$

and we find that

$$Q(\theta \mid \theta') = \theta^T \sum_{j=1}^{m} E_{\theta'}(t_j(Y_j) \mid X = x) - E_{\theta'}(\kappa(\theta) \mid X = x).$$

To maximize Q we differentiate Q and equate the derivative equal to zero. We find that the resulting equation is

$$\sum_{j=1}^{m} E_{\theta'}(t_j(Y_j) \mid X = x) = E_{\theta'}(\nabla\kappa(\theta) \mid X = x).$$

Alternatively, one may also note the following general equation for finding the maximum of $Q(\cdot \mid \theta')$

$$\sum_{j=1}^{m} E_{\theta'}(t_j(Y_j) \mid X = x) = \sum_{j=1}^{m} E_{\theta'}(E_{\theta}(t_j(Y_j) \mid y_1, \ldots, y_{j-1}) \mid X = x),$$

since

$$E_{\theta'}(\nabla\kappa(\theta) \mid X = x) = \sum_{j=1}^{m} E_{\theta'}(\nabla \log \varphi_j(\theta) \mid X = x) = \sum_{j=1}^{m} E_{\theta'}(E_\theta(t_j(Y_j) \mid y_1, \ldots, y_{j-1}) \mid X = x)$$

Example 8.1. Continuing Example 6.4 with M the projection map

$$(\mathbf{y}, \mathbf{z}) \mapsto \mathbf{y}$$

we see that Q is maximized in θ by solving

$$\sum_{i,j} E_{\theta'}(t(Y_{ij} \mid Z_i) \mid \mathbf{Y} = \mathbf{y}) = \sum_i m_i E_{\theta'}(\nabla\kappa(\theta \mid Z_i) \mid \mathbf{Y} = \mathbf{y}).$$

By using Example 6.2 we see that

$$\kappa(\theta \mid Z_i) = \frac{(\theta_1 + \theta_3 Z_i)^2}{4\theta_2} - \frac{1}{2}\log\theta_2,$$

hence

$$\nabla\kappa(\theta \mid Z_i) = \frac{1}{2\theta_2}\begin{pmatrix} \theta_1 + \theta_3 Z_i \\ -\frac{(\theta_1+\theta_3 Z_i)^2}{2\theta_2} - 1 \\ \theta_1 Z_i + \theta_3 Z_i^2 \end{pmatrix} = \begin{pmatrix} \beta_0 + \nu Z_i \\ -(\beta_0 + \nu Z_i)^2 - \sigma^2 \\ \beta_0 Z_i + \nu Z_i^2 \end{pmatrix}.$$

Therefore, Q is maximized by solving the equation

$$\sum_{i,j}\begin{pmatrix} y_{ij} \\ -y_{ij}^2 \\ E_{\theta'}(Z_i \mid \mathbf{Y} = \mathbf{y}) y_{ij} \end{pmatrix} = \sum_i m_i \begin{pmatrix} \beta_0 + \nu E_{\theta'}(Z_i \mid \mathbf{Y}_i = \mathbf{y}_i) \\ -E_{\theta'}((\beta_0 + \nu Z_i)^2 \mid \mathbf{Y} = \mathbf{y}) - \sigma^2 \\ \beta_0 E_{\theta'}(Z_i \mid \mathbf{Y} = \mathbf{y}) + \nu E_{\theta'}(Z_i^2 \mid \mathbf{Y} = \mathbf{y}) \end{pmatrix}.$$

Introducing first $\xi_i = E_{\theta'}(Z_i \mid \mathbf{Y} = \mathbf{y})$ and $\zeta_i = E_{\theta'}(Z_i^2 \mid \mathbf{Y} = \mathbf{y})$ we can rewrite the first and last of the three equations as the linear equation

$$\begin{pmatrix} \sum_i m_i & \sum_i m_i \xi_i \\ \sum_i m_i \xi_i & \sum_i m_i \zeta_i \end{pmatrix}\begin{pmatrix} \beta_0 \\ \nu \end{pmatrix} = \begin{pmatrix} \sum_{i,j} y_{ij} \\ \sum_{i,j} \xi_i y_{ij} \end{pmatrix}.$$

Plugging the solution for β_0 and ν into the second equation we find

$$\sigma^2 = \frac{1}{\sum_i m_i}\left(\sum_{ij} y_{ij}^2 - \sum_i m_i(\beta_0^2 + \nu^2\zeta_i + 2\beta_0\nu\xi_i)\right).$$

This solves the M-step of the EM algorithm for the mixed effects model. What remains is the E-step that amounts to the computation of ξ_i and ζ_i. We know that the joint distribution of $\mathbf{Y}$ and $\mathbf{Z}$ is Gaussian, and we can easily compute the variances and covariances:

$$\operatorname{cov}(Z_i, Z_j) = \delta_{ij}$$

$$\operatorname{cov}(Y_{ij}, Y_{kl}) = \begin{cases} \nu^2 + \sigma^2 & \text{if } i = k, j = l \\ \nu^2 & \text{if } i = k, j \neq l \\ 0 & \text{otherwise} \end{cases}$$

$$\operatorname{cov}(Z_i, Y_{kl}) = \begin{cases} \nu & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

This gives a joint Gaussian distribution

$$\begin{pmatrix} \mathbf{Z} \\ \mathbf{Y} \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mathbf{0} \\ \beta_0 \mathbf{1} \end{pmatrix}, \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix} \right).$$

From this and the general formulas for computing conditional distributions in the multivariate Gaussian distribution:

$$\mathbf{Z} \mid \mathbf{Y} \sim \mathcal{N}\left(\Sigma_{12}\Sigma_{22}^{-1}(\mathbf{Y} - \beta_0 \mathbf{1}), \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21} \right).$$

The conditional means, ξ_i, are thus the coordinates of $\Sigma_{12}\Sigma_{22}^{-1}(\mathbf{Y} - \beta_0 \mathbf{1})$. The conditional second moments, ζ_i, can be found as the diagonal elements of the conditional covariance matrix plus ξ_i^2.

8.3 Fisher information

For statistics relying on classical asymptotic theory we need an estimate of the Fisher information, e.g. the observed Fisher information (Hessian of the negative log-likelihood for the observed data). For numerical optimization of Q or variants of the EM algorithm (like EM gradient or acceleration methods) the gradient and Hessian of Q can be useful. However, these do not directly inform us on the Fisher information. In this section we show some interesting and useful relations between the derivatives of the log-likelihood for the observed data and derivatives of Q with the primary purpose of estimating the Fisher information.

First we look at the peppered moth example, where we note that with $p = p(\theta)$ being some parametrization of the cell probabilities,

$$Q(\theta \mid \theta') = \sum_{k=1}^{K} \frac{x_{j(k)} p_k(\theta')}{M(p(\theta'))_{j(k)}} \log p_k(\theta),$$

where $j(k)$ is defined by $k \in A_{j(k)}$. The gradient of Q w.r.t. θ is therefore

$$\nabla_\theta Q(\theta \mid \theta') = \sum_{k=1}^{K} \frac{x_{j(k)} p_k(\theta')}{M(p(\theta'))_{j(k)} p_k(\theta)} \nabla_\theta p_k(\theta').$$

We recognize from previous computations in Section 7.2.4 that when we evaluate $\nabla_\theta Q(\theta \mid \theta')$ in $\theta = \theta'$ we get

$$\nabla_\theta Q(\theta' \mid \theta') = \sum_{i=1}^{K} \frac{x_{j(i)}}{M(p(\theta'))_{j(i)}} \nabla_\theta p_i(\theta') = \nabla_\theta \ell(\theta'),$$

thus the gradient of ℓ in θ' is actually identical to the gradient of $Q(\cdot \mid \theta')$ in θ'. This is not a coincidence, and it holds generally that

$$\nabla_\theta Q(\theta' \mid \theta') = \nabla_\theta \ell(\theta').$$

This follows from the fact we derived in the proof of Theorem 8.1 that θ' minimizes

$$\theta \mapsto \ell(\theta) - Q(\theta \mid \theta').$$

Another way to phrase this is that the minorant of $\ell(\theta)$ touches ℓ tangentially in θ'.

In the case where the observation $\mathbf{y}$ consists of n i.i.d. observations from the model with parameter θ_0, ℓ as well as $Q(\cdot \mid \theta')$ are sums of terms for which the gradient identity above holds for each term. In particular,

$$\nabla_\theta \ell(\theta_0) = \sum_{i=1}^n \nabla_\theta \ell_i(\theta_0) = \sum_{i=1}^n \nabla_\theta Q_i(\theta_0 \mid \theta_0),$$

and using the second Bartlett identity

$$\mathcal{I}(\theta_0) = V_{\theta_0}(\nabla_\theta \ell(\theta_0))$$

we see that

$$\hat{\mathcal{I}}(\theta_0) = \sum_{i=1}^n \left(\nabla_\theta Q_i(\theta_0 \mid \theta_0) - n^{-1}\nabla_\theta \ell(\theta_0)\right)\left(\nabla_\theta Q_i(\theta_0 \mid \theta_0) - n^{-1}\nabla_\theta \ell(\theta_0)\right)^T$$

is almost an unbiased estimator of the Fisher information. It does have mean $\mathcal{I}(\theta_0)$, but it is not an estimator as θ_0 is not known. Using a plug-in-estimator, $\hat{\theta}$, of θ_0 we get a real estimator

$$\hat{\mathcal{I}} = \hat{\mathcal{I}}(\hat{\theta}) = \sum_{i=1}^n \left(\nabla_\theta Q_i(\hat{\theta} \mid \hat{\theta}) - n^{-1}\nabla_\theta \ell(\hat{\theta})\right)\left(\nabla_\theta Q_i(\hat{\theta} \mid \hat{\theta}) - n^{-1}\nabla_\theta \ell(\hat{\theta})\right)^T,$$

though $\hat{\mathcal{I}}$ will no longer necessarily be unbiased.

We refer to $\hat{\mathcal{I}}$ as the *empirical Fisher information* given by the estimator $\hat{\theta}$. In most cases, $\hat{\theta}$ is the maximum-likelihood estimator, in which case $\nabla_\theta \ell(\hat{\theta}) = 0$ and the empirical Fisher information simplifies to

$$\hat{\mathcal{I}} = \sum_{i=1}^n \nabla_\theta Q_i(\hat{\theta} \mid \hat{\theta}) \nabla_\theta Q_i(\hat{\theta} \mid \hat{\theta})^T.$$

However, $\nabla_\theta \ell(\hat{\theta})$ is in practice only approximately equal to zero, and it is unclear if it should be dropped.

For the peppered moths, where data is collected as i.i.d. samples of n individual specimens and tabulated according to phenotype, we implement the empirical Fisher information with the optional possibility of centering the gradients before computing the information estimate. We note that only three different observations of phenotype are possible, giving rise to three different possible terms in the sum. The implementation works directly on the tabulated data by computing all the three possible terms and then forming a weighted sum according to the number of times each term is present.

```
empFisher <- function(par, x, grad, center = FALSE) {
  grad_MLE <- 0 ## is supposed to be 0 in the MLE
  if (center)
     grad_MLE <-  grad(par, x) / sum(x)
   grad1 <- grad(par, c(1, 0, 0)) - grad_MLE
   grad2 <- grad(par, c(0, 1, 0)) - grad_MLE
```

```
  grad3 <- grad(par, c(0, 0, 1)) - grad_MLE
  x[1] * t(grad1) %*% grad1 +
    x[2] * t(grad2) %*% grad2 +
    x[3] * t(grad3) %*% grad3
}
```

We test the implementation with and without centering and compare the result to a numerically computed hessian using `optimHess` (it is possible to get `optim` to compute the Hessian numerically in the minimizer as a final step, but `optimHess` does this computation separately).

```
## The gradient of Q (equivalently the log-likelihood) was
## implemented earlier as 'grad_loglik'.
grad <- function(par, x) grad_loglik(par, x, prob, Dprob, c(1, 1, 1, 2, 2, 3))
empFisher(phat, c(85, 196, 341), grad)
```

```
##           [,1]     [,2]
## [1,] 18487.558 1384.626
## [2,]  1384.626 6816.612
```

```
empFisher(phat, c(85, 196, 341), grad, center = TRUE)
```

```
##           [,1]     [,2]
## [1,] 18487.558 1384.626
## [2,]  1384.626 6816.612
```

```
optimHess(phat, loglik, grad_loglik, x = c(85, 196, 341),
          prob = prob, Dprob = Dprob, group = c(1, 1, 1, 2, 2, 3))
```

```
##           [,1]     [,2]
## [1,] 18490.938 1384.629
## [2,]  1384.629 6816.769
```

Note that the numerically computed Hessian (the *observed* Fisher information) and the empirical Fisher information are different estimates of the same quantity. Thus they are *not* supposed to be identical on a given data set, but they are supposed to be estimates of the same thing and thus to be similar.

An alternative to the empirical Fisher information or a direct computation of the observed Fisher information is supplemented EM (SEM). This is a general method for computing the observed Fisher information that relies only on EM steps and a numerical differentiation scheme. Define the EM map $\Phi : \Theta \mapsto \Theta$ by

$$\Phi(\theta') = \arg\max_\theta Q(\theta \mid \theta').$$

A global maximum of the likelihood is a fixed point of Φ, and the EM algorithm searches for a fixed point for Φ, that is, a solution to

$$\Phi(\theta) = \theta.$$

Variations of the EM-algorithm can often be seen as other ways to find a fixed point for Φ. From

$$\ell(\theta) = Q(\theta \mid \theta') + H(\theta \mid \theta')$$

it follows that the observed Fisher information equals

$$\hat{i}_X := -D^2_\theta \ell(\hat{\theta}) = \underbrace{-D^2_\theta Q(\hat{\theta} \mid \theta')}_{=\hat{i}_Y(\theta')} \underbrace{-D^2_\theta H(\hat{\theta} \mid \theta')}_{=\hat{i}_{Y|X}(\theta')}.$$

It is possible to compute $\hat{i}_Y := \hat{i}_Y(\hat{\theta})$. For peppered moths (and exponential families) it is as difficult as computing the Fisher information for complete observations.

We want to compute $\hat{i}_X$ but $\hat{i}_{Y|X} := \hat{i}_{Y|X}(\hat{\theta})$ is not computable either. It can, however, be shown that

$$D_\theta \Phi(\hat{\theta})^T = \hat{i}_{Y|X} \left(\hat{i}_Y\right)^{-1}.$$

Hence

$$\hat{i}_X = \left(I - \hat{i}_{Y|X} \left(\hat{i}_Y\right)^{-1}\right) \hat{i}_Y \tag{8.2}$$

$$= \left(I - D_\theta \Phi(\hat{\theta})^T\right) \hat{i}_Y. \tag{8.3}$$

Though the EM map Φ might not have a simple analytic expression, its Jacobian, $D_\theta \Phi(\hat{\theta})$, can be computed via numerical differentiation once we have implemented Φ. We also need the hessian of the map Q, which we implement as an R function as well.

```
Q <- function(p, pp, x = c(85, 196, 341), group) {
  p[3] <- 1 - p[1] - p[2]
  pp[3] <- 1 - pp[1] - pp[2]
  - (x[group] * prob(pp) / M(prob(pp), group)[group]) %*% log(prob(p))
}
```

The R package numDeriv contains functions that compute numerical derivatives.

```
library(numDeriv)
```

The Hessian of Q can be computed using this package.

```
iY <- hessian(Q, phat, pp = phat, group = c(1, 1, 1, 2, 2, 3))
```

Supplemented EM can then be implemented by computing the Jacobian of Φ using numDeriv as well.

```
Phi <- function(pp) MStep(EStep(pp, x = c(85, 196, 341)))
DPhi <- jacobian(Phi, phat)  ## Using numDeriv function 'jacobian'
iX <- (diag(1, 2) - t(DPhi)) %*% iY
iX
```

```
##           [,1]     [,2]
## [1,] 18487.558 1384.626
## [2,]  1384.626 6816.612
```

For statistics, we actually need the inverse Fisher information, which can be computed by inverting $\hat{i}_X$, but we also have the following interesting identity

$$\hat{i}_X^{-1} = \hat{i}_Y^{-1}\left(I - D_\theta\Phi(\hat{\theta})^T\right)^{-1} \tag{8.4}$$

$$= \hat{i}_Y^{-1}\left(I + \sum_{n=1}^{\infty}\left(D_\theta\Phi(\hat{\theta})^T\right)^n\right) \tag{8.5}$$

$$= \hat{i}_Y^{-1} + \hat{i}_Y^{-1}D_\theta\Phi(\hat{\theta})^T\left(I - D_\theta\Phi(\hat{\theta})^T\right)^{-1} \tag{8.6}$$

where the second identity follows by the Neumann series.

The last formula above explicitly gives the asymptotic variance for the incomplete observation X as the asymptotic variance for the complete observation Y plus a correction term.

```
iYinv <- solve(iY)
iYinv + iYinv %*% t(solve(diag(1, 2) - DPhi, DPhi))
```

```
##                [,1]          [,2]
## [1,]  5.492602e-05 -1.115686e-05
## [2,] -1.115686e-05  1.489667e-04
```

```
solve(iX) ## SEM-based, but different use of inversion
```

```
##                [,1]          [,2]
## [1,]  5.492602e-05 -1.115686e-05
## [2,] -1.115686e-05  1.489667e-04
```

The SEM implementation above relies on the `hessian` and `jacobian` functions from the numDeriv package for numerical differentiation.

It is possible to implement the computation of the hessian of Q analytically for the peppered moths, but to illustrate functionality of the numDeriv package we implemented the computation numerically above.

Variants on the strategy for computing $D_\theta\Phi(\hat{\theta})$ via numerical differentiation have been suggested in the literature, specifically using difference quotient approximations along the sequence of EM steps. This is not going to work as well as standard numerical differentiation since this method ignores numerical errors, and when the algorithm gets sufficiently close to the MLE, the numerical errors will dominate in the difference quotients.

8.4 Revisiting Gaussian mixtures

In a two-component Gaussian mixture model the marginal density of the distribution of Y is

$$f(y) = p\frac{1}{\sqrt{2\pi\sigma_1^2}}e^{-\frac{(y-\mu_1)^2}{2\sigma_1^2}} + (1-p)\frac{1}{\sqrt{2\pi\sigma_2^2}}e^{-\frac{(y-\mu_2)^2}{2\sigma_2^2}}.$$

The following is a simulation of data from such a mixture model.

```
sigma1 <- 1
sigma2 <- 2
mu1 <- -0.5
mu2 <- 4
p <- 0.5
n <- 1000
```

```
z <- sample(c(TRUE, FALSE), n, replace = TRUE, prob = c(p, 1 - p))
y <- numeric(n)
n1 <- sum(z)
y[z] <- rnorm(n1, mu1, sigma1)
y[!z] <- rnorm(n - n1, mu2, sigma2)
```

We implement the log-likelihood assuming that the variances are known. Note that the implementation takes just one single parameter argument, which is then supposed to be a vector of all parameters in the model. Internally to the function one has to decide for each entry in the parameter vector what parameter in the model it corresponds to.

```
loglik <- function(par, y) {
  p <- par[1]
  if(p < 0 || p > 1)
    return(Inf)

  mu1 <- par[2]
  mu2 <- par[3]
  -sum(log(p * exp(-(y - mu1)^2 / (2 * sigma1^2)) / sigma1 +
             (1 - p) * exp(-(y - mu2)^2 / (2 * sigma2^2)) / sigma2))
}
```

Without further implementations, `optim` can find the maximum-likelihood estimate if we have a sensible initial parameter guess. In this case we use the true parameters, which can be used when algorithms are tested, but they are, of course, not available for real applications.

```
optim(c(0.5, -0.5, 4), loglik, y = y)[c(1, 2)]
```

```
## $par
## [1]  0.4504834 -0.5614815  3.9283827
##
## $value
## [1] 1392.366
```

However, if we initialize the optimization badly, it does not find the maximum but a local maximum instead.

```
optim(c(0.9, 3, 1), loglik, y = y)[c(1, 2)]
```

```
## $par
## [1] 0.2497462 5.4915698 0.7122072
##
## $value
## [1] 1486.624
```

We will implement the EM algorithm for the Gaussian mixture model by implementing and E-step and an M-step function. We know from Section 6.4.1 how the complete log-likelihood looks, and the E-step becomes a matter of computing

$$p_i(\mathbf{y}) = E(1(Z_i = 1) \mid \mathbf{Y} = \mathbf{y}) = P(Z_i = 1 \mid \mathbf{Y} = \mathbf{y}).$$

The M-step becomes identical to the MLE, which can be found explicitly, but where the indicators $1(Z_i = 1)$ and $1(Z_i = 2) = 1 - 1(Z_i = 1)$ are replaced by the conditional probabilities $p_i(\mathbf{y})$ and $1 - p_i(\mathbf{y})$, respectively.

```
EStep <- function(par, y) {
  p <- par[1]
  mu1 <- par[2]
  mu2 <- par[3]
  a <- p * exp(- (y - mu1)^2 / (2 * sigma1^2)) / sigma1
  b <- (1 - p) * exp(- (y - mu2)^2 / (2 * sigma2^2)) / sigma2
  b / (a + b)
}

MStep <- function(y, pz) {
  n <- length(y)
  N2 <- sum(pz)
  N1 <- n - N2
  c(N1 / n, sum((1 - pz) * y) / N1, sum(pz * y) / N2)
}

EM <- function(par, y, epsilon = 1e-12) {
  repeat{
    par0 <- par
    par <- MStep(y, EStep(par, y))
    if(sum((par - par0)^2) <= epsilon * (sum(par^2) + epsilon))
      break
  }
  par  ## Remember to return the parameter estimate
}

EM(c(0.5, -0.5, 4), y)
```

```
## [1]  0.4505106 -0.5614569  3.9281823
```

The EM algorithm may, just as any other optimization algorithm, end up in a *local* maximum, if it is started wrongly.

```
EM(c(0.9, 3, 1), y)
```

```
## [1] 0.2496783 5.4911923 0.7123882
```

Chapter 9

Stochastic Optimization

Numerical optimization involves different tradeoffs such as an *exploration-exploitation* tradeoff. On the one hand, the objective function must be thoroughly explored to build an adequate model of it. On the other hand, the model should be exploited so as to find the minimum quickly. Another tradeoff is between the accuracy of the model and the time it takes to compute with it.

The optimization algorithms considered in Chapters 7 and 8 work on all available data and take deterministic steps in each iteration. The models are based on accurate local computations of derivatives that can be greedily exploit the local model obtained from derivatives, but they do little exploration.

By including randomness into optimization algorithms it is possible to lower the computational costs and make the algorithms more exploratory. This can be done in various ways. Examples of stochastic optimization algorithms include simulated annealing and evolutionary algorithms that incorporate randomness into the iterative steps with the purpose of exploring the objective function better than a deterministic algorithm is able to. In particular, to avoid getting stuck in saddle points and to escape local minima. Stochastic gradient algorithms form another example, where descent directions are approximated by gradients from random subsets of the data.

The literature on stochastic optimization is huge, and this chapter will only cover some examples of particular relevance to statistics and machine learning. The most prominent applications are to large scale optimization, where stochastic gradient algorithms have become the standard solution. When the dimension of the optimization problem becomes very large, second order methods become prohibitively slow, and if the number of observations is also large, even one computation of the gradient for the entire data batch becomes time consuming. In those cases, stochastic gradient algorithms, that originate from online learning, can make progress more quickly while still using the entire batch of data.

9.1 Stochastic gradient algorithms

Stochastic gradient algorithms have their origin in an online learning framework, where data arrives sequentially as a stream of data points and where the objective function is an expected loss. Robbins and Monro (1951) introduced in their seminal paper a variant of the online stochastic gradient algorithm that they called the *stochastic approximation method*, and they established the first convergence result for such algorithms. To understand what stochastic gradient algorithms are supposed to optimize, we introduce the general framework of a

population model and give conditions that ensure that the basic online algorithm converges. Subsequently, the basic online algorithm is turned into an algorithm for batch data, which is the algorithm of primary interest. In the following sections, various beneficial extensions of the basic batch algorithm are explored.

9.1.1 Population models and loss functions

We will consider observations from the sample space $\mathcal{X}$, and we will be interested in estimating parameters from the parameter space Θ. A loss function

$$L : \mathcal{X} \times \Theta \to \mathbb{R}$$

is fixed throughout, and we want to minimize the expected loss also known as the *risk*. That is, we want to minimize

$$H(\theta) = E(L(X, \theta)) = \int L(x, \theta)^{-}(dx)$$

with $X \in \mathcal{X}$ having distribution μ. Of course, it is implicitly understood that the expectation has to be well defined for all θ.

Example 9.1. Suppose that $X = (Y, Z)$ with Y a real valued random variable, and that $\mu(z, \theta)$ denotes a parametrized mean value conditionally on $Z = z$. With the squared error loss,

$$L((y, z), \theta) = \frac{1}{2}(y - \mu(z, \theta))^2,$$

the risk is the mean squared error

$$\text{MSE}(\theta) = \frac{1}{2}E(Y - \mu(Z, \theta))^2.$$

From the definition of $\text{MSE}(\theta)$ we see that

$$2\text{MSE}(\theta) = E(Y - E(Y \mid Z))^2 + E(E(Y \mid Z) - \mu(Z, \theta))^2,$$

where the first term does not depend upon θ. Thus minimizing the mean squared error is the same as finding θ_0 such that $\mu(z, \theta_0)$ is the optimal approximation of $E(Y \mid Z = z)$ in a squared error sense.

Note how the link between the distribution of X and the parameter is defined in the example above by the choice of loss function. There is no upfront assumption that $E(Y \mid Z = z) = \mu(z, \theta_0)$ for any $\theta_0 \in \Theta$, but if there is such a θ_0, it will clearly be a minimizer. In general, the optimal θ_0 is simply the θ that minimizes the risk.

An alternative to the squared error loss is the log-likelihood loss, which can be used when we have a parametrized family of distributions.

Example 9.2. Suppose that f_θ denotes a density on $\mathcal{X}$ parametrized by θ. Then the log-likelihood loss is

$$L(x, \theta) = -\log f_\theta(x).$$

The corresponding risk,

$$H(\theta) = -E \log(f_\theta(X)),$$

is known as the cross-entropy. If the distribution of X has density f^0 then

$$\begin{aligned} H(\theta) &= -E \log(f^0(X)) - E \log(f_\theta(X)/f^0(X)) \\ &= H(f^0) + D(f^0 \mid\mid f_\theta) \end{aligned}$$

where the first term is the entropy of f^0, and the second is the Kullback-Leibler divergence of f_θ from f^0. The entropy does not depend upon θ and minimizing the cross-entropy is thus the same as finding a θ with f_θ being the optimal approximation of f^0 in a Kullback-Leibler sense.

We consider now the same regression setup as in Example 9.1 with $X = (Y, Z)$, but we let

$$f_\theta(y|z) = e^{-\mu(z,\theta)} \frac{y^{\mu(z,\theta)}}{y!}$$

denote the Poisson point probabilities for the Poisson distribution with mean $\mu(z, \theta)$ conditionally on $Z = z$. Then the log-likelihood loss is

$$-\log f_\theta(y|z) = \mu(z, \theta) - y \log(\mu(z, \theta))$$

up to an additive constant not depending on θ, and the cross-entropy is

$$H(\theta) = E\big(\mu(Z, \theta) - E(Y \mid Z) \log(\mu(Z, \theta))\big),$$

again up to an additive constant. This risk function quantifies how $\mu(Z, \theta)$ deviates from $E(Y \mid Z)$ in a different way than the risk based on the squared error loss. However, if $E(Y \mid Z = z) = \mu(z, \theta_0)$ for some θ_0, it is still true that θ_0 is a minimizer, cf. Exercise 9.1.

The log-likelihood loss is an appropriate loss function if the parametrized family of distributions fits data well. For a Gaussian (conditional) mean value model the log-likelihood loss gives the same risk as the squared error loss – but the squared error loss can also be suitable even if data is not Gaussian. It can be a suitable loss whenever we just want to fit a model of the conditional mean of Y. If we want to fit the (conditional) median instead, we can use the absolute deviation

$$L((y, z), \theta) = |y - \mu(z, \theta)|.$$

This is a special case of the check loss functions used for quantile regression. Thus by choosing the loss function we decide which aspects of the distribution we model, that is, whether we want a good global fit as for the log-likelihood loss, a fit of the mean value as for the squared error loss, or a fit of the the median or another quantile as for the check loss.

9.1.2 Online stochastic gradient algorithm

The classical stochastic gradient algorithm is an example of an online learning algorithm. It is based on the simple observation that if we can interchange differentiation and expectation then

$$\nabla H(\theta) = E\left(\nabla_\theta L(X, \theta)\right),$$

thus if $X_1, X_2, \ldots$ form an i.i.d. sequence then $\nabla_\theta L(X_i, \theta)$ is unbiased as an estimate of the gradient of H for any θ and any i. With inspiration from gradient descent algorithms it is natural to suggest stochastic parameter updates of the form

$$\theta_{n+1} = \theta_n - \gamma_n \nabla_\theta L(X_{n+1}, \theta_n)$$

starting from some initial value θ_0. The direction, $\nabla_\theta L(X_{n+1}, \theta_n)$, is, however, not guaranteed to be a descent direction for H, and even if it is, γ_n is typically not chosen to guarantee descent.

The sequence of step size parameters $\gamma_n \geq 0$ are known collectively as the *learning rate*. It can be a deterministic sequence, but γ_n may also depend on $X_1, \ldots, X_n$ and $\theta_0, \ldots, \theta_n$. For stochastic gradient algorithms, convergence can be shown under global conditions on the decay of the learning rate rather than local conditions on the individual step lengths.

Theorem 9.1. *Suppose H is strongly convex and*

$$E(\|\nabla_\theta L(X,\theta))\|^2) \leq A + B\|\theta\|^2.$$

If θ^* is the global minimizer of H then θ_n converges almost surely toward θ^* if

$$\sum_{n=1}^{\infty} \gamma_n^2 < \infty \quad \textit{and} \quad \sum_{n=1}^{\infty} \gamma_n = \infty. \tag{9.1}$$

From the above result, convergence of the algorithm is guaranteed if the learning rate, γ_n, converges to 0 but does so sufficiently slowly. Though formulated in a slightly different way, Robbins and Monro (1951) were the first to demonstrate convergence of an online learning algorithm under conditions as above on the learning rate. Following their terminology, much has since been written on online learning and adaptive control theory under the name *stochastic approximation*, Lai (2003).

The precise way that the learning rate decays is known as the *decay schedule*, and a flexible three-parameter power law family of decay schedules is given by

$$\gamma_n = \frac{\gamma_0 K}{K + n^a} = \frac{\gamma_0}{1 + K^{-1} n^a}$$

for some initial learning rate $\gamma_0 > 0$ and constants $K, a > 0$. If $a \in (0.5, 1]$ the resulting learning rate satisfies the convergence conditions (9.1).

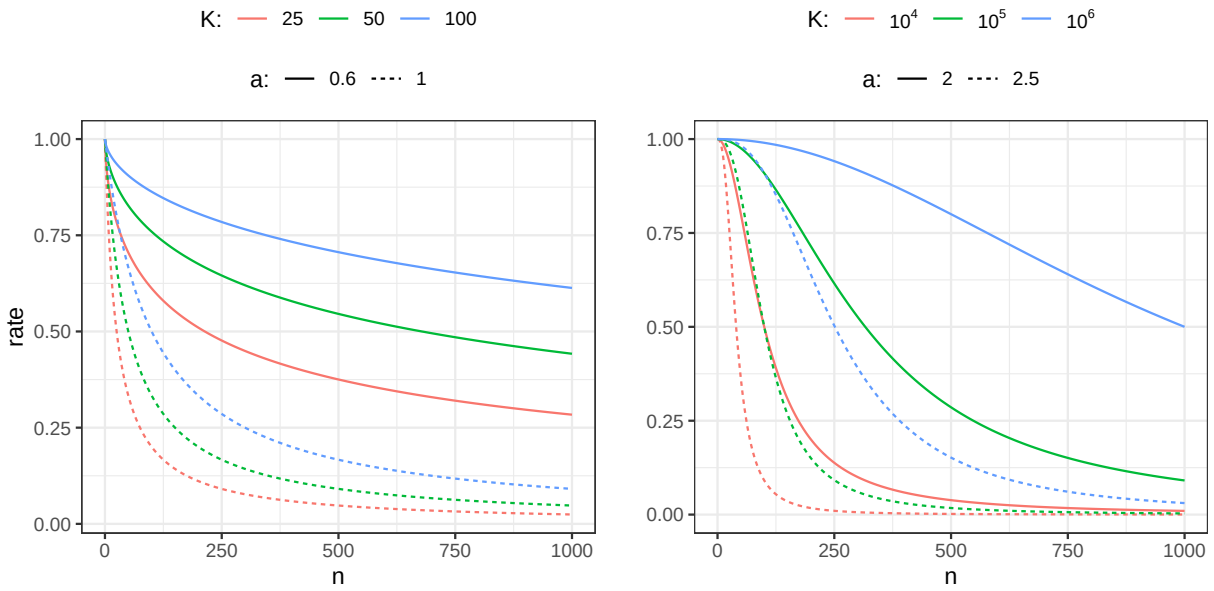


Figure 9.1: Power law decay schedules as a function of n for $\gamma_0 = 1$ and for different choices of K and a. The left figure shows decay schedules with a chosen so that the convergence conditions are fulfilled, whereas the right figure shows decay schedules for which the convergence conditions are not fulfilled.

The parameter γ_0 determines the initial baseline rate, and Figure 9.1 illustrates the effect of the parameters K and a on the decay. The parameter a is the asymptotic exponent of $\gamma_n \sim \gamma_0 K n^{-a}$, and K determines how quickly the rate will turn into a pure power law decay. Moreover, if we have a target rate, γ_1, that we want to hit after n_1 iterations, and we fix the exponent a, we can also solve for K to find

$$K = \frac{n_1^a \gamma_1}{\gamma_0 - \gamma_1}.$$

This gives us a decay schedule that interpolates between γ_0 and γ_1 over the range $0, \ldots, n_1$ of iterations.

We implement `decay_scheduler()` as a function that returns a particular decay schedule, with the possibility to determine K automatically from a target rate.

```
decay_scheduler <- function(gamma0 = 1, a = 1, K = 1, gamma1, n1) {
  force(a)
  if (!missing(gamma1) && !missing(n1))
    K <- n1^a * gamma1 / (gamma0 - gamma1)
  b <- gamma0 * K
  function(n) b / (K + n^a)
}
```

The following example of online Poisson regression illustrates the general ideas.

Example 9.3. In this example $Y_i \mid Z_i = z_i \sim \text{Pois}(\varphi(\beta_0 + \beta_1 z_i))$ for $\beta = (\beta_0, \beta_1)^T$ the parameter vector and $\varphi : \mathbb{R} \to (0, \infty)$ a continuously differentiable function. We let the Z_i-s be uniformly distributed in $(-1, 1)$, but this choice is not particularly important. The conditional mean of Y_i given $Z_i = z_i$ is

$$\mu(z_i, \beta) = \varphi(\beta_0 + \beta_1 z_i)$$

and we will first consider the squared error loss. To this end, observe that

$$\nabla_\beta \mu(z_i, \beta) = \varphi'(\beta_0 + \beta_1 z_i) \begin{pmatrix} 1 \\ z_i \end{pmatrix},$$

which for the squared error loss results in the gradient

$$\nabla_\beta \frac{1}{2}(y_i - \mu(z_i, \beta))^2 = \varphi'(\beta_0 + \beta_1 z_i)(\mu(z_i, \beta) - y_i) \begin{pmatrix} 1 \\ z_i \end{pmatrix}.$$

We simulate data and explore the learning algorithm in the special case with $\varphi = \exp$. To clearly emulate the online nature of the algorithm, the implementation below generates the observations sequentially in the loop.

```
N <- 5000
beta_true = c(2, 3)
mu <- function(z, beta) exp(beta[1] + beta[2] * z)
beta <- vector("list", N)

rate <- decay_scheduler(gamma0 = 0.0004, K = 100)
beta[[1]] <- c(beta0 = 1, beta1 = 1)

for(i in 2:N) {
  # Simulating a new data point
  z <- runif(1, -1, 1)
  y <- rpois(1, mu(z, beta_true))
  # Update via squared error gradient
  mu_old <- mu(z, beta[[i - 1]])
  beta[[i]] <- beta[[i - 1]]  - rate(i) * mu_old * (mu_old - y) * c(1, z)
}
beta[[N]]  # This is close to beta_true; the algorithm works!
```

```
##    beta0    beta1
## 2.037990 2.987941
```

For the log-likelihood loss we instead find the gradient

$$\nabla_\beta\big(\mu(z_i,\beta) - y_i \log(\mu(z_i,\beta))\big) = \frac{\varphi'(\beta_0+\beta_1 z_i)}{\mu(z_i,\beta)}(\mu(z_i,\beta)-y_i)\begin{pmatrix}1\\ z_i\end{pmatrix},$$

which leads to a slightly different but equally valid algorithm. In the special case with $\varphi = \exp$, the derivative is $\varphi'(\beta_0+\beta_1 z_i) = \mu(z_i,\beta)$,

$$\nabla_\beta\big(\mu(z_i,\beta) - y_i \log(\mu(z_i,\beta))\big) = (\mu(z_i,\beta)-y_i)\begin{pmatrix}1\\ z_i\end{pmatrix},$$

and the log-likelihood gradient differs from the squared error gradient by lacking the factor $\mu(z_i,\beta)$. With Z uniformly distributed on $(-1,1)$, the distribution of $\mu(Z,(2,3))$ has range between $e^{-1} \simeq 0.3679$ and $e^5 \simeq 148.4$ and is right skewed, that is, it is concentrated toward the smaller values but with a long right tail. Its median is $e^2 \simeq 7.389$, while its mean is $(e^5 - e)/6 \simeq 24.67$.

The squared error gradient is typically longer than the log-likelihood gradient due to the factor $\mu(z_i,\beta)$ — and sometimes by a large factor. In the implementation below with the gradient of the log-likelihood we therefore choose γ_0 a factor 25 larger than the $\gamma_0 = 0.0004$ that was used with the squared error gradient.

```
rate <- decay_scheduler(gamma0 = 0.01, K = 100)
beta[[1]] <- c(beta0 = 1, beta1 = 1)

for(i in 2:N) {
  # Simulating a new data point
  z <- runif(1, -1, 1)
  y <- rpois(1, mu(z, beta_true))
  # Update via log-likelihood gradient
  mu_old <- mu(z, beta[[i - 1]])
  beta[[i]] <- beta[[i - 1]]  - rate(i) * (mu_old - y) * c(1, z)
}
beta[[N]]  # This is close to beta_true; this algorithm also works!
```

```
##    beta0    beta1
## 2.008452 2.987035
```

Figure 9.2 shows how the estimates of β_0 and β_1 converge toward the true values for the two different choices of loss functions. With $\varphi = \exp$ and either loss function, the risk, $H(\beta)$, is strongly convex and attains its unique minimum in $\beta^* = (2,3)^T$. Theorem 9.1 suggests convergence for appropriate learning rates – except that the growth condition on the gradient is not fulfilled with $\varphi = \exp$. The growth condition *is* fulfilled if we replace the exponential function by softplus, $\varphi(w) = \log(1+e^w)$, which for small values of w behaves like the exponential function, but grows linearly with w for $w \to \infty$.

The gradient from the squared error loss resulted in slower convergence and a more jagged sample path than the gradient from the log-likelihood. This is explained by the random factor $\mu(z_i,\beta)$ for the squared error gradient, which makes the step sizes somewhat irregular when data is from a Poisson distribution. It also makes choosing γ_0 a delicate balance. A small increase of γ_0 will make the algorithm unstable, while decreasing γ_0 will make the convergence even slower, though the fluctuations will also be damped. Making the right choice – or even a suitable choice – of decay schedule depends heavily on the problem considered and the gradient used. It is a problem specific challenge to find a good schedule – and even just to choose the three parameters when we use the power law schedule.

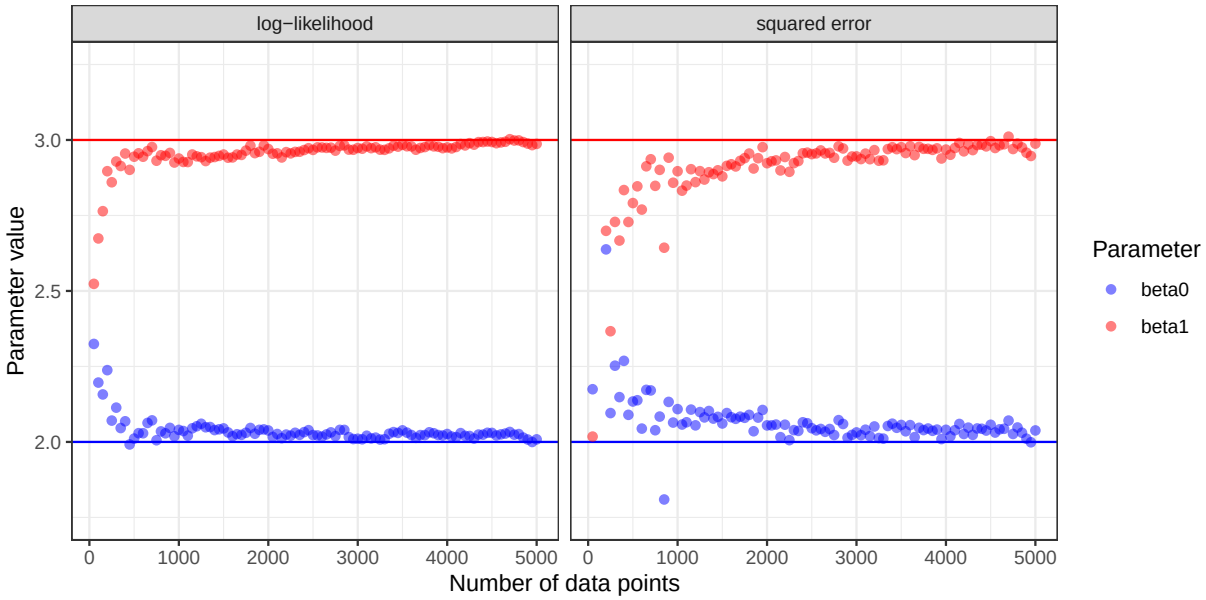


Figure 9.2: Estimated parameter values for the two parameters β_0 (true value 2) and β_1 (true value 3) in the Poisson regression model as a function of the number of data points in the online stochastic gradient algorithm.

The example illustrates that the loss function not only determines what we model, but also how well the learning algorithm works. Both loss functions are appropriate, but since the data is from the Poisson distribution, the log-likelihood loss leads to faster convergence. Exercise 9.2 explores the opposite situation where the data is from a Gaussian distribution.

9.1.3 Batch stochastic gradient algorithms

The online algorithm does not store data, and once a data point is used it is forgotten. The online algorithm is working in a context where data arrives as a stream of data points and the model is updated continually. In statistics, we more frequently encounter batch algorithms, where an entire batch of data points is stored and processed by the algorithm, and where each data point in the batch can be accessed as many times as we like. However, when the batch is sufficiently large, many standard batch algorithms are slow, and some ideas from online algorithms can beneficially be transferred to batch processing of data.

Within the population model framework in Section 9.1.1 the objective is to minimize the population risk, $H(\theta)$, defined as the expected loss w.r.t. the probability distribution μ. For the online algorithms we imagine an endless stream of data points from μ, which can be used to ultimately minimize $H(\theta)$. Batch algorithms replace the population quantity by an empirical surrogate — the average loss on the batch

$$H_N(\theta) = \frac{1}{N}\sum_{i=1}^{N} L(x_i, \theta).$$

Minimizing H_N as a surrogate of minimizing H is known as empirical risk minimization.

If data is i.i.d. the standard deviation of $H_N(\theta)$ decays as $N^{-1/2}$ with N, while the run time of its computation increases as N. Thus as we invest more computation time by using more data we get diminishing returns; doubling the run time will only lower the precision of H_N as an approximation of H by a factor $1/\sqrt{2} \simeq 0.7$. The same is, of course, true if we look at the gradient, $\nabla H_N(\theta)$, or higher derivatives of H_N.

It is not obvious that the computational costs of using the entire data set to compute the gradient, say, is worth the effort compared to using only a fraction of the data. Thus we ask if there is a better tradeoff between run time and precision by using a fraction of data points each time we compute the gradient? The stochastic gradient algorithm is one such algorithm that "cycles through the data" and uses only a random fraction of data points for each computation. In its basic version presented first, it uses only a single data point in each iteration, and it is really the same algorithm as presented in Section 9.1.2 except that the population risk is replaced by the empirical risk defined in terms of the batch data.

With $\hat{\mu}_N = \frac{1}{N}\sum_{i=1}^N \delta_{x_i}$ denoting the empirical distribution of the batch data set, we see that

$$H_N(\theta) = \int L(x,\theta)\bar{}_N(dx),$$

that is, the empirical risk is simply the expected loss with respect to the empirical distribution. Thus to minimize H_N we can use the online approach by sampling observations from $\hat{\mu}_N$. This leads to the following basic version of the stochastic gradient algorithm applied to a data batch.

From an initial parameter value, θ_0, we iteratively compute the parameters as follows: given θ_n

- sample an index i uniformly from $\{1,\ldots,N\}$
- compute $\rho_n = \nabla_\theta L(x_i,\theta_n)$
- update the parameter $\theta_{n+1} = \theta_n - \gamma_n\rho_n$.

Note that sampling the index i is equivalent to sampling an observation from $\hat{\mu}_N$, which in turn is the same as nonparametric bootstrapping. Just as in the online setting, the sequence of learning rates, γ_n, is a tuning parameter of the algorithm.

We implement the basic stochastic gradient algorithm below, allowing for a user defined decay schedule of the learning rate. However, instead of implementing one long loop, we divide the iterations into *epochs* with each epoch consisting of N iterations. In the implementation, the maximal number of iterations is also given in terms of epochs, and the decay schedule is applied on a per epoch basis.

We also introduce a small twist on the sampling from the empirical distribution; instead of sampling with replacement (bootstrapping) we sample without replacement. Sampling N indices from $\{1,\ldots,N\}$ without replacement is the same as sampling a permutation of the indices.

```
SG <- function(
  par,
  grad,               # Function of parameter and observation index
  N,                  # Sample size
  gamma,              # Decay schedule or a fixed learning rate
  maxiter = 100,      # Max epoch iterations
  sampler = sample,   # How data is resampled. Default is a random permutation
  cb = NULL,
  ...
) {
  gamma <- if (is.function(gamma)) gamma(1:maxiter) else rep(gamma, maxiter)
  for(k in 1:maxiter) {
    if(!is.null(cb)) cb()
    samp <- sampler(N)
    for(j in 1:N) {
      i <-  samp[j]
```

```
      par <- par - gamma[k] * grad(par, i, ...)
    }
  }
  par
}
```

One epoch in the algorithm above is exactly one pass through the entire batch of data points, but in a random order. The default value of `sampler = sample` means that resampling is done without replacement. If we call `SG()` with `sampler = function(N) sample(N, replace = TRUE)` we would get sampling with replacement, in which case an epoch would be a pass through N data points sampled independently from the batch. Sampling with replacement will feed the stochastic gradient algorithm with i.i.d. samples from the empirical distribution. Sampling without replacement introduces some dependence. Curiously, sampling without replacement has turned out to be empirically superior to sampling with replacement, and recent theoretical results, Gürbüzbalaban et al. (2019), support that it leads to a faster rate of convergence.

We may ask if the sampling actually matters, and whether we could just leave out that part of the algorithm? In practice, data sets may come in a "bad order", for instance in an unfortunate ordering according to one or more of its variables, and cycling through the data points in such an ordering can easily lead the algorithm astray. *It is therefore important to always randomize the order of the data points somehow.* A minimal amount of randomization in common use is to just do one initial random permutation, corresponding to moving `samp <- sampler(N)` outside of the outer for-loop above. This may be enough randomization for the algorithm to work in some cases, but the link to the convergence result for the online algorithm is lost.

Example 9.4. As a continuation of Example 9.3 we consider the batch version of Poisson regression. We will only use the log-likelihood gradient, and we first simulate a small data set with $N = 50$ data points.

```
N <- 50
z <- runif(N, -1, 1)
y <- rpois(N, mu(z, beta_true))
grad_pois <- function(par, i) (mu(z[i], par) - y[i]) * c(1, z[i])
```

Using the `grad_pois()` function above, we run the stochastic gradient algorithm for 1000 epochs with a decay schedule that interpolates between $\gamma_0 = 0.02$ and $\gamma_1 = 0.001$.

```
pois_SG_tracer <- tracer("par", N = 0)
SG(
  c(0, 0),
  grad_pois,
  N = N,
  gamma = decay_scheduler(gamma0 = 0.02, gamma1 = 0.001, n1 = 1000),
  maxiter = 1000,
  cb = pois_SG_tracer$tracer
)
```

```
## [1] 1.905394 3.162559
```

The resulting parameter estimate should be compared to the maximum-likelihood estimate from an ordinary Poisson regression.

```
beta_hat <- coefficients(glm(y ~ z, family = poisson))
```

```
## beta_hat: 1.898305 3.15986
```

The batch version of stochastic gradient descent converges toward the minimizer, $(\hat{\beta}_0, \hat{\beta}_1)$, of the empirical risk. This is contrary to the online version that converges toward the minimizer of the theoretical risk, which in this case is $(\beta_0, \beta_1) = (2, 3)$. With a larger batch size, $(\hat{\beta}_0, \hat{\beta}_1)$ will come closer to $(2, 3)$. Figure 9.3 shows clearly how the algorithms converge toward a limit that depends on the batch size, and for $N = 500$, this limit is much closer to the theoretical minimizer.

```
N <- 500
z <- runif(N, -1, 1)
y <- rpois(N, mu(z, beta_true))
pois_SG_tracer_2 <- tracer("par", N = 0)
SG(
  par = c(0, 0),
  grad = grad_pois,
  N = N,
  gamma = decay_scheduler(gamma0 = 0.02, gamma1 = 0.001, n1 = 100),
  cb = pois_SG_tracer_2$tracer
)
```

```
## [1] 1.994688 3.022137
```

```
beta_hat_2 <- coefficients(glm(y ~ z, family = poisson))
```

```
## beta_hat_2: 1.98893 3.027715
```

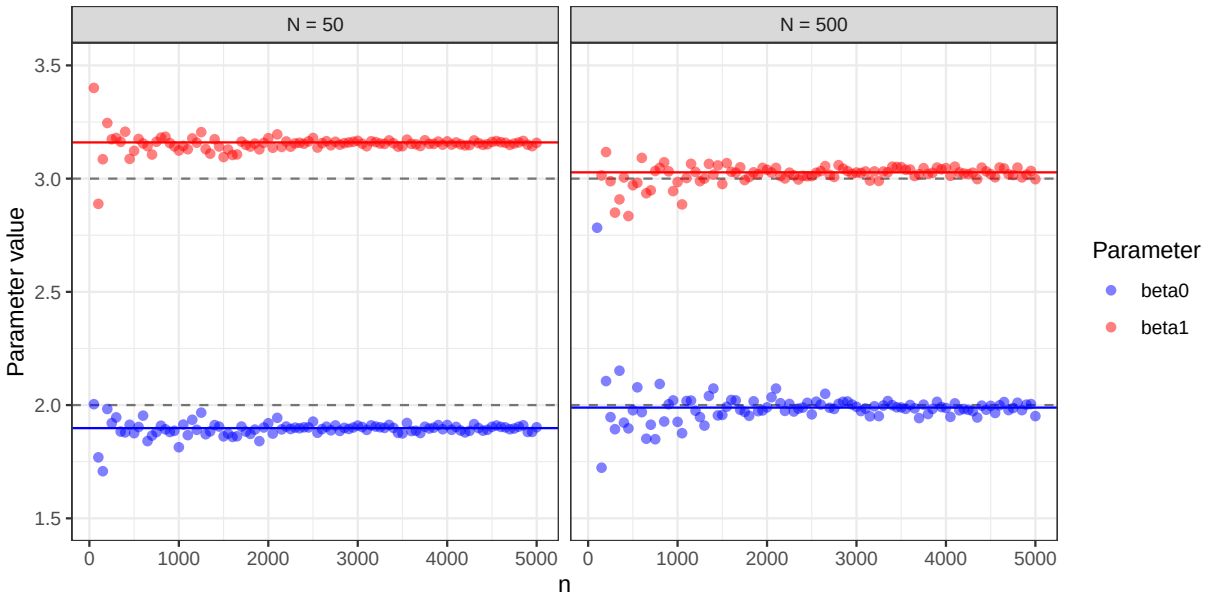


Figure 9.3: Estimated parameter values for the two parameters $\beta_0 = 2$ and $\beta_1 = 3$ in the Poisson regression model as a function of the number of iterations in the stochastic gradient algorithm. For batch size $N = 50$, the algorithm converges to a parameter clearly different from the theoretically optimal one (gray dashed lines), while for batch size $N = 500$ the limit is closer to $(2, 3)$.

Example 9.4 and Figure 9.3, in particular, illustrate that if the data set is relatively small, the algorithm quickly attains a precision smaller than the statistical uncertainty, and further optimization is therefore futile. However, for larger data sets, optimization to a greater precision can be beneficial.

9.1.4 Predicting news article sharing on social media

In this section we will illustrate the use of the basic stochastic gradient algorithm for learning a model that predicts how many times a news article is shared on social media. The data will be subjected to transformations and normalizations to make the use of a linear model and the squared error loss reasonable.

The basic stochastic gradient algorithm for the linear model was introduced to the early machine learning community in 1960 via ADALINE (Adaptive Linear Neuron) by Bernard Widrow and Ted Hoff. ADALINE was implemented as a physical device capable of learning patterns via stochastic gradient updates. The math is the same today, but the implementation has fortunately become somewhat easier.

With a linear model and the squared error loss, $L((y,x),\beta) = \frac{1}{2}(y - \beta^T x)^2$, the gradient becomes

$$\nabla_\beta L((y,x),\beta) = -x(y - \beta^T x) = x(\beta^T x - y),$$

which results in updates of the form

$$\beta_{n+1} = \beta_n - \gamma_n x_i(\beta_n^T x_i - y_i).$$

That is, the parameter moves in the direction of x_i if $\beta_n^T x_i < y_i$ and in the direction of $-x_i$ if $\beta_n^T x_i > y_i$. The amount by which it moves is controlled partly by the learning rate, γ_n, and partly by the size of the residual, $y_i - \beta_n^T x_i$. A larger residual gives a larger move.

The following function factory for linear models takes a model matrix and a response vector for the complete data batch as arguments and implements the squared error loss function on the entire batch as well as the gradient in a single observation. The returned list also contains a parameter vector of the correct dimension, which can be used for initialization of optimization algorithms.

```
ls_model <- function(X, y) {
  N <- length(y)
  X <- unname(X) # Strips X of names
  list(
    # Initial parameter value
    par0 = rep(0, ncol(X)),
    # Objective function
    H = function(beta)
      drop(crossprod(y - X %*% beta)) / (2 * N),
    # Gradient in a single observation
    grad = function(beta, i) {
      xi <- X[i, ]
      xi * drop(xi %*% beta - y[i])
    }
  )
}
```

The data, originally collected by Fernandes et al. (2015), was obtained from the UCI Machine Learning Repository and contains 39,644 observations on 61 variables. One variable is the integer valued `shares`, which will be the target variable of our predictions.

```
News <- readr::read_csv("data/OnlineNewsPopularity.csv")
```

Two of the variables, `timedelta` and `url`, are not relevant predictors, and `is_weekend` is redundant given the other weekday variables, so we exclude those three variables. Some of

the predictors are also highly correlated, and we exclude four additional predictors before the model matrix is constructed.

```
News <- dplyr::select(
  News,
  - url,
  - timedelta,
  - is_weekend,
  - n_non_stop_words,
  - n_non_stop_unique_tokens,
  - self_reference_max_shares,
  - kw_min_max
)
# The model matrix without an explicit intercept is constructed from all
# variables remaining in the data set but the target variable 'shares'
X <- model.matrix(shares ~ . - 1, data = News)
```

This data set is by current standards not a large data set – the dense model matrix takes only about 20 MB of memory – and the linear model (with the target variable `shares` log-transformed) can easily be fitted by simply solving the least squares problem. It takes about 0.2 seconds on a standard laptop to compute the solution. The residual plot in Figure 9.4 shows that the model is actually not a poor fit to data, though there is a considerable unexplained residual variance.

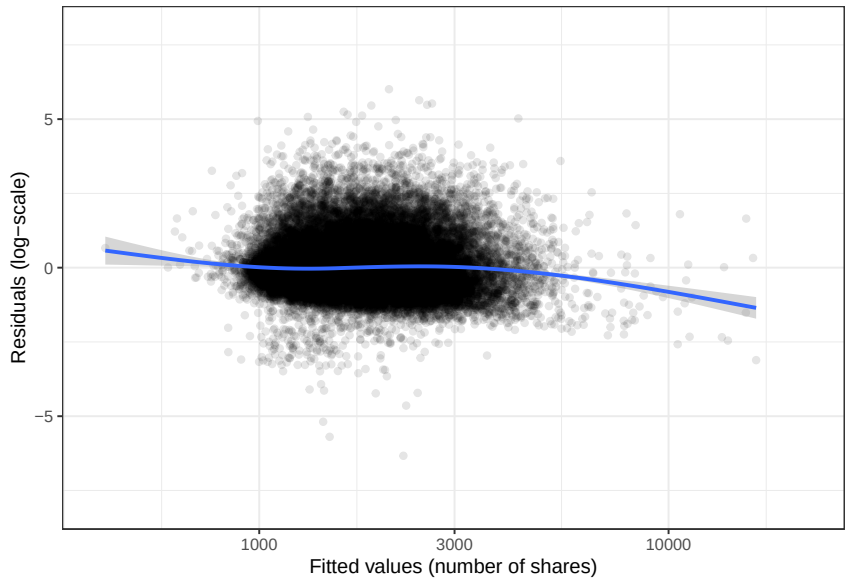


Figure 9.4: Residual plot for the linear model of the logarithm of news article shares.

Optimization of the squared error loss using this data set will be used to illustrate a number of stochastic gradient algorithms even though we can compute the optimizer easily by other means. The real practical benefit of stochastic gradient algorithms comes when applied to large scale problems that are difficult to treat as textbook examples. But using a toy problem makes it easier to understand in detail how the different algorithms behave.

We will standardize all the columns of X to have the same norm. Specifically to have non-central second moment 1. This does not change the optimization problem but corresponds to a reparametrization where all parameters are rescaled. The rescaling brings the parameters on a comparable scale, which is typically a good idea for optimization algorithms based on

gradients only, see also Exercise 9.3. After rescaling we initialize the linear model with a call to `ls_model()` and refit the model using the new parametrization.

```
X_raw <- X
# Standardization and log-transforming the target variable
X <- scale(X, center = FALSE)
y <- log(News$shares)
# The '%<-%' destructure assignment operator is from the zeallot package
c(par0, H, ls_grad) %<-% ls_model(X, y)
# Fitting the model using standard linear model computations
lm_News <- lm.fit(X, y)
par_hat <- lm_News$coefficients  # Will be used below for comparisons
```

We first run the stochastic gradient algorithm with a fixed learning rate of $\gamma = 10^{-5}$ for 50 epochs with a tracer that computes and stores the value of the objective function at each epoch.

```
SG_tracer <- tracer("value", expr = quote(value <- H(par)))
SG(
  par = par0,
  grad = ls_grad,
  N = nrow(X),
  gamma = 1e-5,
  maxiter = 50,
  cb = SG_tracer$tracer
)
```

Using the trace from the last epochs, we can compare the objective function values to the minimum found using `lm.fit()` above. The minimum was not reached completely after the 50 epochs that took some time to compute.

```
SG_trace_low <- summary(SG_tracer)
tail(SG_trace_low)
```

```
##        value    .time
## 45 0.4268483 16.22107
## 46 0.4262983 16.52924
## 47 0.4259705 16.86827
## 48 0.4251088 17.23979
## 49 0.4245424 17.63807
## 50 0.4240064 18.07972
```

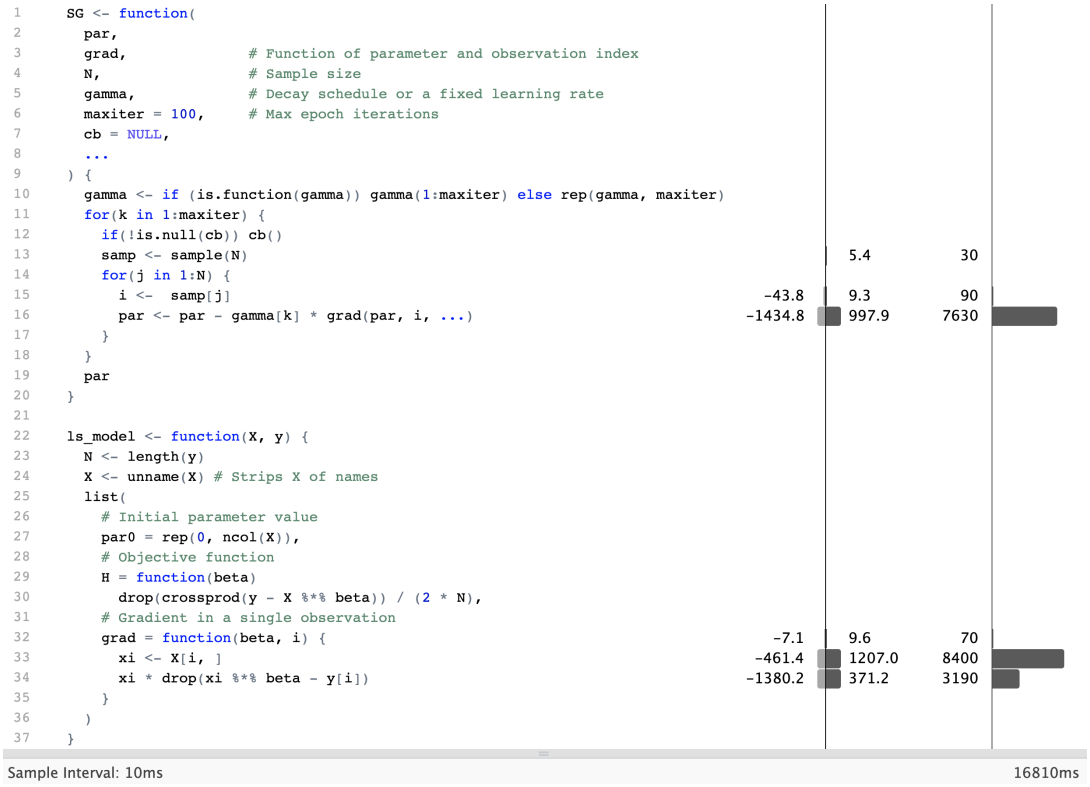
```
H(par_hat)
```

```
## [1] 0.3781595
```

We will use profiling to investigate what most of the time was spent on in the stochastic gradient algorithm.

```
profvis(
  SG(
    par = par0,
    grad = ls_grad,
    N = nrow(X),
    gamma = 1e-5,
    maxiter = 50
```

```
  )
)
```



The profiling result shows, unsurprisingly, that the computation of the gradient and the update of the parameter vector takes up most of the run time. But if we look closer at the implementation of the gradient, we see that the innocuously looking subsetting `X[i, ]` to the i-th row is actually responsible for about half of the run time. We also see a substantial allocation and deallocation of memory associated with this line. It is a bottleneck of the R implementation that slicing out a row from a bigger matrix cannot be done without creating a copy of that row, and this is why this particular line takes up so much time.

To further investigate the convergence of the basic stochastic gradient algorithm we run it with a larger learning rate of $\gamma = 5 \times 10^{-5}$ and then with a power law decay schedule, which interpolates from an initial learning rate of 10^{-3} to a learning rate of 10^{-5} after 50 epochs.

```
SG_tracer$clear()
SG(
  par = par0,
  grad = ls_grad,
  N = nrow(X),
  gamma = 5e-5,
  maxiter= 50,
  cb = SG_tracer$tracer
)
SG_trace_high <- summary(SG_tracer)
SG_tracer$clear()
SG(
  par = par0,
  grad = ls_grad,
  N = nrow(X),
```

```
  gamma = decay_scheduler(gamma0 = 7e-5, gamma1 = 4e-5, a = 0.5, n1 = 50),
  maxiter= 50,
  cb = SG_tracer$tracer
)
SG_trace_decay <- summary(SG_tracer)
```

We will compare the convergence of the three stochastic gradient algorithms with the convergence of gradient descent with backtracking. For gradient descent we choose $\gamma = 8 \times 10^{-2}$, which results in only a few initial backtracking steps and then all subsequent steps will use the step length $\gamma = 8 \times 10^{-2}$. Choosing a larger γ for this particular optimization resulted in backtracking until a step length around 8×10^{-2} was reached, thus this choice of γ will use a minimal amount of time on the backtracking step of the algorithm.

```
grad_H <- function(beta) crossprod(X, X %*% beta - y) / nrow(X)
GD_tracer <- tracer("value", N = 10)
GD(
  par = par0,
  H = H,
  gr = grad_H,
  gamma = 8e-2,
  maxiter = 800,
  cb = GD_tracer$tracer
)
GD_trace <- summary(GD_tracer)
```

Figure 9.5 shows how the four algorithms converge. The gradient descent algorithm converges faster than the stochastic gradient algorithm with the low learning rate $\gamma = 10^{-5}$, but with the high learning rate $\gamma = 5 \times 10^{-5}$ and the power law decay schedule the stochastic gradient algorithm converges about as fast as gradient descent. For this particular run, the power law decay schedule shows marginally faster convergence than the fixed high learning rate, and this was ensured after some tuning of the decay schedule parameters. Despite theoretical guarantees of convergence with the power law decay schedule, the practical choice of a suitable learning rate or decay schedule is really an empirical art. With very large data, the tuning of learning rate parameters can be done on a small subset of data before the algorithms are run using the full data set.

For comparison purposes it is typically better to monitor convergence of optimization algorithms as a function of real time than iterations, but when comparing algorithms in terms of real time we are admittedly comparing their specific implementations. The R implementation of the stochastic gradient algorithm has some shortcomings that are not shared by the gradient descent algorithm. One epoch of the stochastic gradient algorithm should be about as computationally demanding as one iteration of gradient descent as both will compute the gradient in each data point exactly once and add them up. The vectorized batch gradient computation is fairly efficient in R, but the iterative looping over data in the stochastic gradient algorithm is not, and this inefficiency is compounded by the inefficiency of matrix slicing that results in data copying as the profiling revealed. Thus the comparisons in Figure 9.5 are arguably not entirely fair to the stochastic gradient algorithm – only the specific R implementation.

In the subsequent section we will see alternatives to the basic stochastic gradient algorithm, which will diminish the shortcomings of a pure R implementation somewhat. Another way to circumvent some of the shortcomings of the R implementation is to rewrite the algorithm using Rcpp. We will pursue Rcpp implementations in Section 9.3, but we will first consider some beneficial modifications of the basic algorithm.

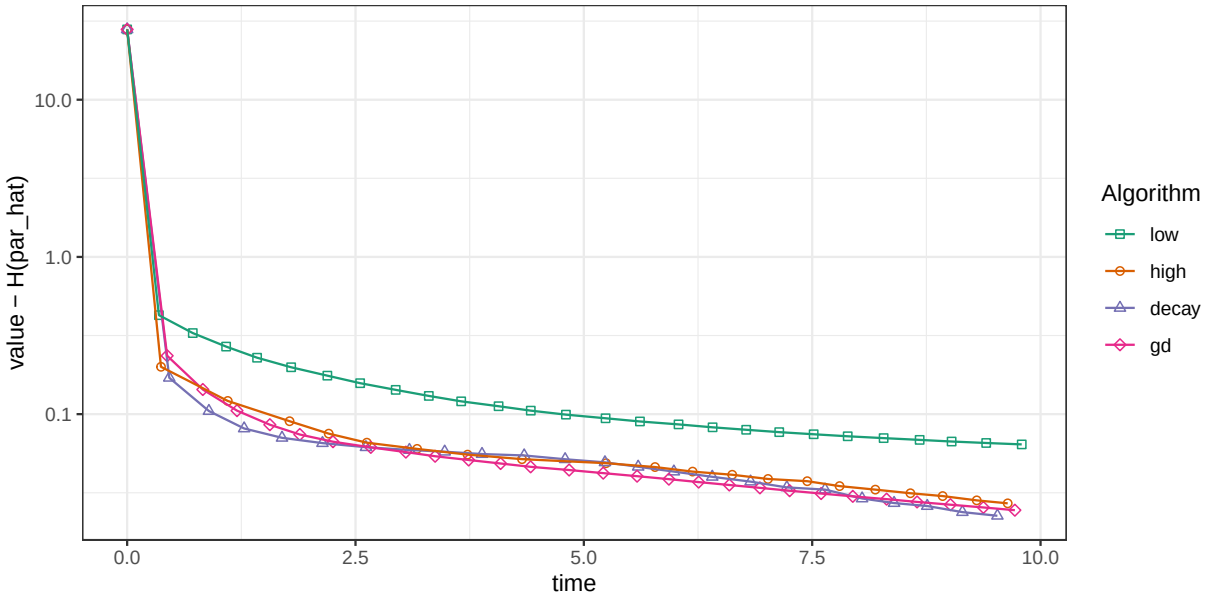


Figure 9.5: Convergence of squared error loss on the news article data set for four algorithms: gradient descent (gd) and three basic stochastic gradient algorithms with a low learning rate, a high learning rate and a power law decay schedule.

9.2 Beyond basic stochastic gradient algorithms

The gradient $\nabla_\theta L(x_i, \theta)$ for a single random data point is quickly computed, and though unbiased as an estimate of $\nabla_\theta H_N(\theta)$ it has a large variance. This affects the basic stochastic gradient algorithm negatively as the directions of each update can oscillate quite wildly from iteration to iteration. This section covers some techniques that yield a better tradeoff between run time and variability.

The most obvious technique is to use more than one observation per computation of the gradient, which gives us *mini-batch* stochastic gradient algorithms. A second technique is to incorporate some memory about previous directions into the movements of the algorithms — in the same spirit as how the conjugate gradient algorithm uses the previous gradient to modify the descent direction.

The literature on deep learning has recently exploded with variations on the stochastic gradient algorithm. Performance is mostly studied empirically and applied in practice to the highly non-convex optimization problem of learning a neural network. A comprehensive coverage of all the different ideas will not be attempted, and only three of the most solidified variations will be treated. The use of mini-batches is ubiquitous, and momentum will be introduced to illustrate a variation with memory. Finally, the Adam algorithm uses memory in combination with adaptive learning rates to achieve both speed and robustness.

9.2.1 Mini-batches

The three steps of the mini-batch stochastic gradient algorithm with mini-batch size m are: given θ_n

- sample m indices, $I_n = \{i_1, \ldots, i_m\}$, from $\{1, \ldots, N\}$
- compute $\rho_n = \frac{1}{m} \sum_{i \in I_n} \nabla_\theta L(x_i, \theta_n)$
- update the parameter $\theta_{n+1} = \theta_n - \gamma_n \rho_n$.

Of course, the mini-batch algorithm with $m = 1$ is the basic stochastic gradient algorithm. As

for the basic algorithm, we implement the variation where we sample a *partition*

$$I_1 \cup \ldots \cup I_M \subseteq \{1, \ldots, N\}$$

for $M = \lfloor N/m \rfloor$ and in one epoch loop through the mini-batches $I_1, \ldots, I_M$.

In the following sections we will develop a couple of modifications to the basic stochastic gradient algorithm, and we will therefore implement a more generic version of the algorithm. What is common to all the modifications is that they differ in the details of the epoch loop, thus we take out that loop as a separate function.

```
SG <- function(
  par,
  N,                  # Sample size
  gamma,              # Decay schedule or a fixed learning rate
  epoch = batch,      # Epoch update function
  ...,                # Other arguments passed to epoch updates
  maxiter = 100,      # Max epoch iterations
  sampler = sample,   # How data is resampled. Default is a random permutation
  cb = NULL
) {
  gamma <- if (is.function(gamma)) gamma(1:maxiter) else rep(gamma, maxiter)
  for(k in 1:maxiter) {
    if(!is.null(cb)) cb()
    samp <- sampler(N)
    par <- epoch(par, samp, gamma[k], ...)
  }
  par
}
```

The implementation uses `batch()` as the default update function, and we implement this function below. It uses the random permutation to generate the M mini-batches, and it contains a loop through the mini-batches containing a call of `grad()` for the computation of the average gradient for each mini-batch in the loop. Note that the `grad` argument to `batch()` will be captured by and passed on from a call of `SG()` via the `...` argument.

```
batch <- function(
    par,
    samp,
    gamma,
    grad,               # Function of parameter and observation index
    m = 50,             # Mini-batch size
    ...
  ) {
    M <- floor(length(samp) / m)
    for(j in 0:(M - 1)) {
      i <- samp[(j * m + 1):(j * m + m)]
      par <- par - gamma * grad(par, i, ...)
    }
    par
}
```

The `grad()` function implemented in `ls_model()` in Section 9.1.4 assumes that the index argument i is a single number and not a vector. It computes for a vector i a matrix containing

the different gradients as columns. We therefore reimplement `ls_model()` so that `grad()` computes the average of the gradients as it is supposed to for a mini-batch.

```
ls_model <- function(X, y) {
  N <- length(y)
  X <- unname(X)
  list(
    par0 = rep(0, ncol(X)),
    H = function(beta)
      drop(crossprod(y - X %*% beta)) / (2 * N),
    # Gradient that works for a mini-batch indexed by i
    grad = function(beta, i) {
      xi <- X[i, , drop = FALSE]
      drop(crossprod(xi, xi %*% beta - y[i])) / length(i)
    }
  )
}
```

We initialize the linear model using the new implementation of `ls_model()`.

```
c(par0, H, ls_grad) %<-% ls_model(X, y)
```

With increased flexibility of the algorithms comes more tuning parameters, and making a good choice of all of them becomes increasingly difficult. When introducing mini-batches we need to choose the mini-batch size in addition to the learning rate, and a good choice of learning rate or decay schedule will depend on the size of the mini-batch. To simplify matters, a mini-batch size of 1000 and a fixed learning rate are used in the subsequent applications. The learning rate will generally be chosen as large as possible without making the algorithms diverge, and with a mini-batch size of 1000 it is possible to run the algorithm with a learning rate of $\gamma = 0.05$.

```
SG_tracer$clear()
SG(
  par = par0,
  N = nrow(X),
  gamma = 5e-2,
  grad = ls_grad,
  m = 1000,
  maxiter = 200,
  cb = SG_tracer$tracer
)
```

Figure 9.6 shows that this implementation of the mini-batch stochastic gradient algorithm converges faster than the basic stochastic gradient algorithm with a power law decay schedule as well as the gradient descent algorithm. Eventually, it begins to fluctuate due to the fixed learning rate, but it quickly gets close to the minimum. We should not jump to conclusions from this assessment. It only really shows the improvement to the specific R implementation of using a mini-batch algorithm, and this implementation clearly benefits from the more vectorized computations with mini-batches of size 1000. We will return to this discussion in Section 9.3.1.

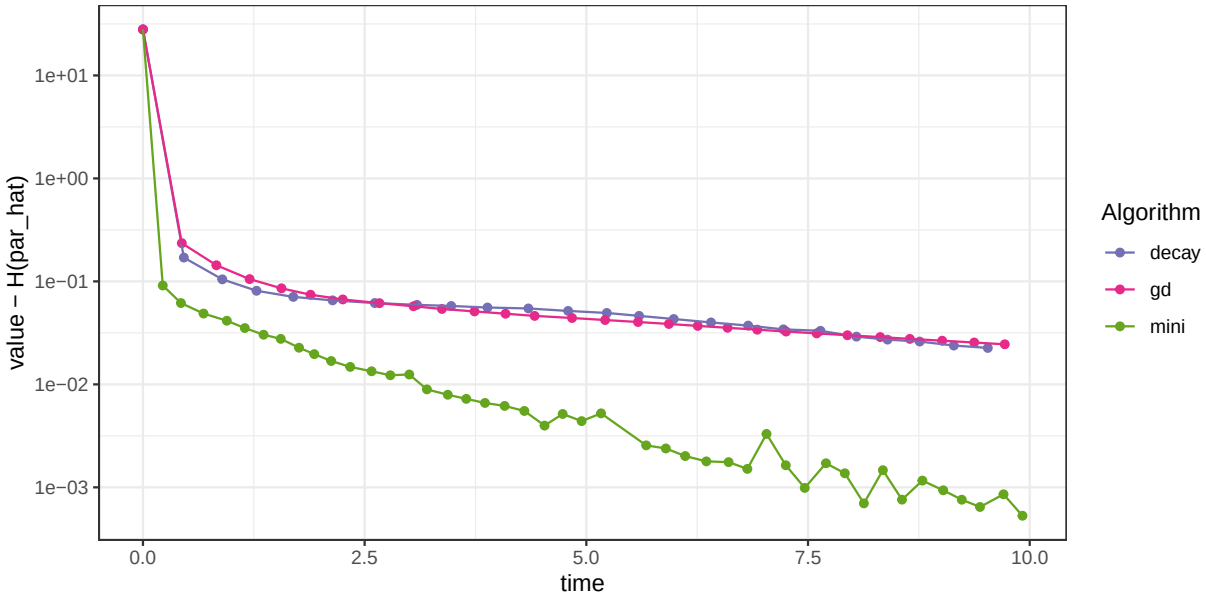


Figure 9.6: Convergence of squared error loss on the news article data set for three algorithms: basic stochastic gradient with a power law decay schedule (decay), gradient descent (gd), and mini-batch stochastic gradient with a batch size of 1000 and a fixed learning rate (mini).

9.2.2 Momentum

Mini-batches stabilize the gradients, and so does momentum. Both techniques can be used in combination, and the momentum update of a mini-batch stochastic gradient algorithm is as follows: Given θ_n and a batch $I_n \subseteq \{1, \ldots, N\}$ with $|I_n| = m$

- compute $g_n = \frac{1}{m} \sum_{i \in I_n} \nabla_\theta L(x_i, \theta_n)$
- compute $\rho_n = \beta \rho_{n-1} + (1 - \beta) g_n$
- update the parameter $\theta_{n+1} = \theta_n - \gamma_n \rho_n$.

The memory of the algorithm is in the second step, where the direction, ρ_n, is updated using a convex combination of the previous direction, ρ_{n-1}, and the mini-batch gradient, g_n. Usually, the initial direction is chosen as $\rho_0 = 0$. The parameter $\beta \in [0, 1)$ is a tuning parameter determining how long the memory is. A value like $\beta = 0.9$ or $\beta = 0.95$ is often recommended – otherwise the memory in the algorithm will be rather short, and the effect of using momentum will be small. A choice of $\beta = 0$ corresponds to the mini-batch algorithm without memory.

Contrary to the batch epoch function, the momentum epoch function needs to store the previous direction between updates. It is not immediately clear how to achieve this between two epochs using the generic `SG()` implementation, but by implementing momentum epochs using a function factory, we can easily use an enclosing environment of the epoch function for storage.

```
momentum <- function() {
  rho <- 0
  function(
    par,
    samp,
    gamma,
    grad,
    m = 50,              # Mini-batch size
    beta = 0.95,         # Momentum memory
```

```
    ...
  ) {
    M <- floor(length(samp) / m)
    for(j in 0:(M - 1)) {
      i <- samp[(j * m + 1):(j * m + m)]
      # Using '<<-' assigns the value to rho in the enclosing environment
      rho <<- beta * rho + (1 - beta) * grad(par, i, ...)
      par <- par - gamma * rho
    }
    par
  }
}
```

When calling `SG()` below with `epoch = momentum()`, the evaluation of the function factory `momentum()` returns the momentum epoch function with is own local environment used to store ρ. With momentum, we can increase the learning rate to $\gamma = 7 \times 10^{-2}$.

```
SG_tracer$clear()
SG(
  par = par0,
  N = nrow(X),
  gamma = 7e-2,
  epoch = momentum(),
  grad = ls_grad,
  m = 1000,
  maxiter = 150,
  cb = SG_tracer$tracer
)
```

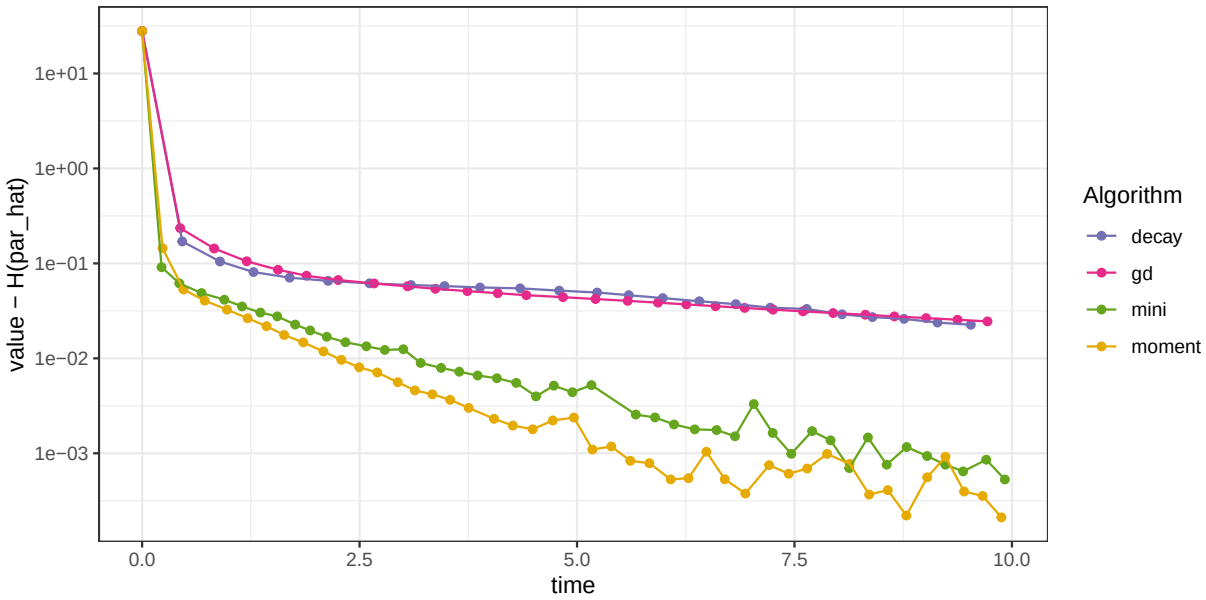


Figure 9.7: Convergence of squared error loss on the news article data set for four algorithms: gradient descent (gd), basic stochastic gradient with a power law decay schedule (decay), and mini-batch stochastic gradient with a batch size of 1000 and a fixed learning rate either without momentum (mini) or with momentum (moment).

Figure 9.7 shows that the momentum algorithm converges a little faster than the mini-batch

algorithm without momentum. This can be explained by the slightly larger learning rate made possible by the use of momentum. With enough memory, momentum dampens rapid fluctuations, which allows for a larger choice of learning rate and a speedier convergence. However, too much memory results in low frequency oscillations and slow convergence. The excellent article Why Momentum Really Works contains many more details about momentum algorithms and beautiful illustrations.

9.2.3 Adaptive learning rates

One difficulty with optimization algorithms based only on gradients is that gradients are not invariant to reparametrizations. In fact, using gradients implicitly assumes that all parameters are on comparable scales. For our news article example, we standardized the model matrix to achieve this, but for many other optimization problems it is not so easy to choose a parametrization with all parameters on comparable scales. And even when we can reparametrize so that all parameters are on a common scale, this common scale can change from problem to problem making it impossible to recommend a good default choice of learning rate. The practical implication is that a considerable amount of tuning is necessary, when the algorithms are applied.

Algorithms that implement adaptive learning rates are alternatives to extensive tuning. They include schemes for adjusting the learning rate to the specific optimization problem. Adapting the learning rate is equivalent to scaling the gradient adaptively, and to achieve a form of automatic standardization of parameter scales, we will consider algorithms that adaptively scale each coordinate of the gradient separately.

To gain some intuition on how to sensibly adapt the scales of the gradient, we will analyze the typical scale of the mini-batch gradient for the linear model. Introduce first the normalized squared column norms

$$\zeta_j = \frac{1}{N}\sum_{i=1}^{N} x_{ij}^2 = \frac{1}{N}\|x_{\cdot j}\|_2^2,$$

and note that with a standardized X, all the ζ_j-s are equal. With $\hat{\beta}$ the least squares estimate we also have that

$$\frac{1}{N}\sum_{i=1}^{N} x_{ij}(y_i - \hat{\beta}^T x_i) = 0$$

for $j = 1, \ldots, p$. Thus if we sample a random index, ι, from $\{1, \ldots, N\}$ it holds that $E(x_{\iota j}(y_\iota - \hat{\beta}^T x_\iota)) = 0$, where the expectation is w.r.t. ι. With

$$\hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{\beta}^T x_i)^2$$

denoting the residual variance, we also have that

$$V(x_{\iota j}(y_\iota - \hat{\beta}^T x_\iota)) = E(x_{\iota j}^2(y_\iota - \hat{\beta}^T x_\iota)^2) \simeq \zeta_j \hat{\sigma}^2$$

For the above approximation to hold, the squared residual, $(y_\iota - \hat{\beta}^T x_\iota)^2$, must be roughly independent of x_ι, which is not guaranteed by the least squares fit alone, but holds approximately if data is from a population with homogeneous residual variance.

If $I \subseteq \{1, \ldots, N\}$ is a random subset of size m sampled *with* replacement, the averaged gradient

$$g = -\frac{1}{m}\sum_{i \in I} x_i(y_i - \hat{\beta}^T x_i)$$

is an average of m i.i.d. random variables with mean 0, thus

$$E(g_j^2) = V(g_j) \simeq \zeta_j \frac{\hat{\sigma}^2}{m}.$$

If $\odot$ denotes the coordinate wise product of vectors (aka the Hadamard product), this can also be written as

$$E(g \odot g) \simeq \zeta \frac{\hat{\sigma}^2}{m}.$$

The computations suggest that by estimating $v = E(g \odot g)$ for a mini-batch gradient evaluated in $\beta = \hat{\beta}$, we are in fact estimating ζ up to a scale factor. We extrapolate this insight to the general case and standardize the j-th coordinate of the descent direction by an estimate of $1/\sqrt{v_j}$ to bring the coordinates on a (more) common scale. We will implement adaptive estimation of v using a similar update scheme as for momentum, where the estimate in iteration n is updated as a convex combination of the current value of $g_n \odot g_n$ and the previous estimate of v.

Given θ_n and a batch $I_n \subseteq \{1, \ldots, N\}$ with $|I_n| = m$ the update consists of the following steps

- compute $g_n = \frac{1}{m} \sum_{i \in I_n} \nabla_\theta L(x_i, \theta_n)$
- compute $\rho_n = \beta_1 \rho_{n-1} + (1 - \beta_1) g_n$
- compute $v_n = \beta_2 v_{n-1} + (1 - \beta_2) g_n \odot g_n$
- update the parameter $\theta_{n+1} = \theta_n - \gamma_n \rho_n / (\sqrt{v_n} + 10^{-8})$.

The vectors ρ_0 and v_0 are usually initialized to be 0. The tuning parameters $\beta_1, \beta_2 \in [0, 1)$ control the memory of the first and second moments, respectively. The $\sqrt{v_n}$ and the division in the last step are coordinate wise. The constant 10^{-8} could, of course, be chosen differently, but is just a safeguard against division-by-zero.

The algorithm above is known as *Adam* (adaptive moment estimation), and was introduced and analyzed by Kingma and Ba (2014). They include so-called bias-correction steps that upscale ρ_n and v_n by the factors $1/(1 - \beta_1^n)$ and $1/(1 - \beta_2^n)$, respectively. These steps are not difficult to implement but are left out in the implementation below for simplicity. It is also possible to replace the $\sqrt{v_n}$ by other powers v_n^q. The choice of $q = 1$ instead of $q = 1/2$ makes the algorithm (more) invariant to scale changes. Again, for simplicity we will only implement the algorithm with $q = 1/2$.

The `adam()` function below is a function factory just as `momentum()`, which returns a function doing the Adam epoch update loop with an enclosing environment used for storage of `rho` and `v`.

```
adam <- function() {
  rho <- v <- 0
  function(
    par,
    samp,
    gamma,
    grad,
    m = 50,           # Mini-batch size
    beta1 = 0.9,      # Momentum memory
    beta2 = 0.9,      # Second moment memory
    ...
  ) {
    M <- floor(length(samp) / m)
    for(j in 0:(M - 1)) {
```

```
      i <- samp[(j * m + 1):(j * m + m)]
      gr <- grad(par, i, ...)
      rho <<- beta1 * rho + (1 - beta1) * gr
      v <<- beta2 * v + (1 - beta2) * gr^2
      par <- par - gamma * (rho / (sqrt(v) + 1e-8))
    }
    par
  }
}
```

We run the stochastic gradient algorithm with Adam epochs and mini-batches of size 1000 with a fixed learning rate of 0.01 and a power law decay schedule interpolating between a learning rate of 0.5 and 0.002. The theoretical results by Kingma and Ba (2014) support a decay schedule proportional to $1/\sqrt{n}$, thus we take $a = 0.5$ below.

```
SG_tracer$clear()
SG(
  par = par0,
  N = nrow(X),
  gamma = 1e-2,
  epoch = adam(),
  grad = ls_grad,
  m = 1000,
  maxiter = 150,
  cb = SG_tracer$tracer
)
SG_trace_adam <- summary(SG_tracer)
SG_tracer$clear()
SG(
  par = par0,
  N = nrow(X),
  gamma = decay_scheduler(gamma0 = 0.5, gamma1 = 2e-3, a = 0.5, n1 = 150),
  epoch = adam(),
  grad = ls_grad,
  m = 1000,
  maxiter = 150,
  cb = SG_tracer$tracer
)
SG_trace_adam_decay <- summary(SG_tracer)
```

Figure 9.8 shows that both runs of the Adam implementation converge faster initially than any of the other algorithms. Eventually they both level off when the error is around 10^{-3} and from this point on they fluctuate randomly. What is not shown is that Adam is also somewhat more robust to changes of the learning rate and rescaling of the parameters. Though Adam has more tuning parameters, it is easier to find good values of those parameters that will lead to fast convergence.

9.3 Stochastic gradient algorithms with Rcpp

As pointed out toward the end of Section 9.1.4, the implementations of stochastic gradient algorithms in R suffer from some shortcomings. In this section we will explore how either

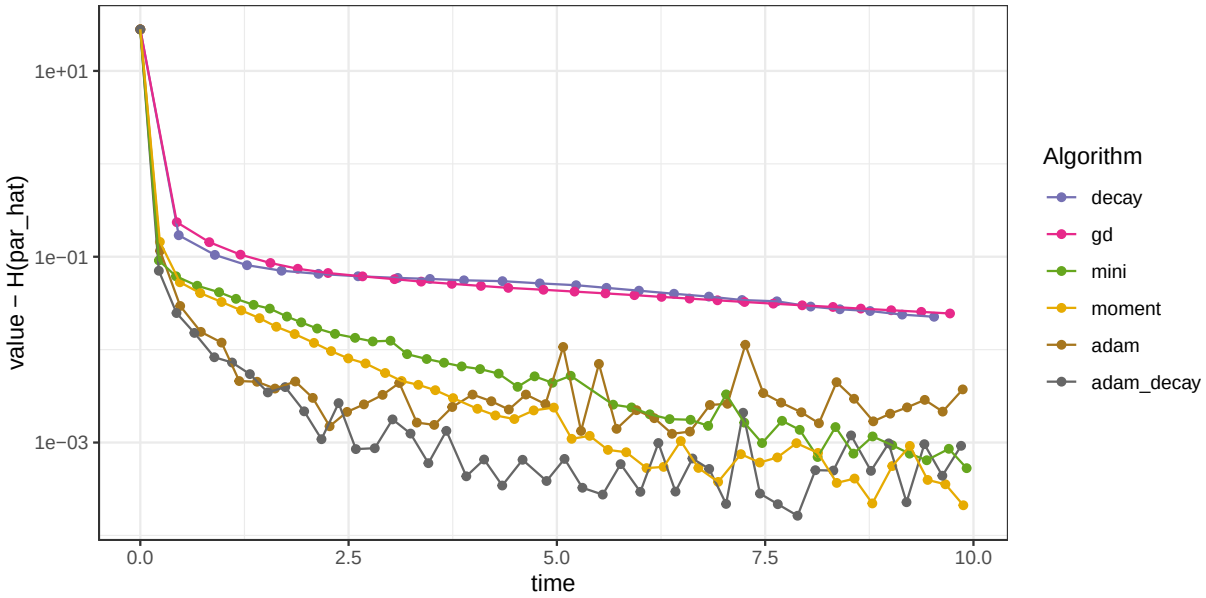


Figure 9.8: Convergence of squared error loss on the news article data set for six algorithms: basic stochastic gradient with a power law decay schedule (decay), gradient descent (gd), mini-batch stochastic gradient with a batch size of 1000 and a fixed learning rate either without momentum (mini) or with momentum (moment), and the Adam algorithm either with a fixed learning rate (adam) or a power law decay schedule (adam_decay).

parts of the algorithms or entire algorithms can be moved to C++ via Rcpp.

The modularity of the `SG()` implementation makes it easy to replace the implementation of either the gradient computation or the entire epoch loop by a C++ implementation, while retaining the overall control of the algorithm and the resampling in R. This is explored first and consists mostly of translating the numerical linear algebra of the gradient computations into C++ code. We can then easily test, compare and benchmark the implementations using the R implementation as a reference.

In the second part of this section the entire mini-batch stochastic gradient algorithm is translated into C++. This has a couple of notable consequences. First, we need access to a sampler in C++ that can do the randomization. While there are various C++ interfaces to an equivalent of `sample()`, some considerations need to go into an appropriate choice. Second, we have to give up on tracing as otherwise implemented. Though it is possible to implement callback of an R function from a C++ function, a tracer will not have the same access to the calling environment as in the R implementation. Thus for performance assessment we will rely on benchmarking of the entire algorithm.

For the C++ implementations we need to give up on some of the abstractions that R provides, though we will benefit from Rcpp data types like `NumericVector` and `NumericMatrix`. In a final implementation we will use RcppArmadillo to regain an abstract approach to numerical linear algebra via the C++ library Armadillo.

9.3.1 Gradients and epochs in Rcpp

We first implement the computation of the mini-batch gradient in Rcpp for the linear model, thus replacing the R implementation of the gradient in `ls_model()`. The implemented function takes the model matrix X and the target variable y as arguments in addition to the parameter and the subset of indices representing the mini-batch. The matrix-vector multiplications for

computing predicted values, residuals and ultimately the gradient are replaced by two double for-loops.

```
// [[Rcpp::export]]
NumericVector lin_grad(
    NumericVector beta,
    IntegerVector ii,
    NumericMatrix X,
    NumericVector y
) {
  int m = ii.length(), p = beta.length();
  NumericVector grad(p), yhat(m);
  // Shift indices one down due to zero-indexing in C++
  IntegerVector iii = clone(ii) - 1;

  for(int i = 0; i < m; ++i) {
    for(int j = 0; j < p; ++j) {
      yhat[i] += X(iii[i], j) * beta[j];
    }
  }
  for(int i = 0; i < m; ++i) {
    for(int j = 0; j < p; ++j) {
      grad[j] += X(iii[i], j) * (yhat[i]- y[iii[i]]);
    }
  }
  return grad / m;
}
```

Note how `clone(ii)` is used to create a copy of the index vector `ii` before it is shifted by one. When R objects like vectors and matrices are passed as arguments to a C++ function, only a pointer to the data in the object is effectively passed. A modification of the resulting Rcpp object within the body of the C++ function will be reflected in R – most likely as an unwanted side effect. To prevent this we can create a copy using `clone()` of all data in the object – a so-called deep copy.

Since only a pointer to a matrix is passed when calling a C++ function, no notable cost is associated with passing the entire model matrix as argument. On the contrary, if we were to pass a subset of the model matrix to the C++ function from R, data would inevitably be copied. And since slicing of the matrix in C++ does not create a copy of data, the implementation of `lin_grad()` avoids the data copying problem that matrix slicing in R incurred on the gradient computations.

We could test `lin_grad()` by comparing it to `ls_grad()` for different choices of parameter vectors and indices. A simple way to achieve this is to run the mini-batch stochastic gradient algorithm with either implementation of the gradient for one epoch and compare the results. Note that we make sure the algorithms use the same mini-batches by setting the seed.

```
set.seed(10)
par1 <- SG(
  par = par0,
  grad = ls_grad,
  N = nrow(X),
  gamma = 0.01,
  maxiter = 1
```

```
)
set.seed(10)
par2 <- SG(
  par = par0,
  grad = lin_grad,
  N = nrow(X),
  gamma = 0.01,
  maxiter = 1,
  X = X,
  y = y
)
range(par1 - par2)
```

```
## [1] 0 0
```

The differences are all 0, thus the test indicates that `lin_grad()` implements the same computation as `ls_grad()`.

We can also move the entire epoch loop to C++, thus replacing the `batch()` R function by Rcpp function `lin_batch()` below. This is achieved simply by adding one outer loop to `lin_grad()` above, which loops over mini-batches and updates the parameter vector.

```
// [[Rcpp::export]]
NumericVector lin_batch(
    NumericVector par,
    IntegerVector ii,
    double gamma,
    NumericMatrix X,
    NumericVector y,
    int m = 50
) {
  int p = par.length(), N = ii.length();
  int M = floor(N / m);
  NumericVector grad(p), yhat(N), beta = clone(par);
  IntegerVector iii = clone(ii) - 1;

  for(int j = 0; j < M; ++j) {
    for(int i = j * m; i < (j + 1) * m; ++i) {
      for(int k = 0; k < p; ++k) {
        yhat[i] += X(iii[i], k) * beta[k];
      }
    }
    for(int k = 0; k < p; ++k) {
      grad[k] = 0;
      for(int i = j * m; i < (j + 1) * m; ++i) {
        grad[k] += X(iii[i], k) * (yhat[i] - y[iii[i]]);
      }
    }
    beta = beta - gamma * (grad / m);
  }
  return beta;
}
```

We can test the implementation of `lin_batch()` just as `lin_grad()` by running the stochastic gradient algorithm for one epoch. We note that the `lin_batch()` function should be passed to `SG()` as the `epoch` argument, and that no `grad` argument is required. The gradient computations are hard coded into the implementation of this epoch function.

```
set.seed(10)
par1 <- SG(
  par = par0,
  grad = ls_grad,
  N = nrow(X),
  gamma = 0.01,
  maxiter = 1
)
set.seed(10)
par2 <- SG(
  par = par0,
  epoch = lin_batch,
  N = nrow(X),
  gamma = 0.01,
  maxiter = 1,
  X = X,
  y = y
)
range(par1 - par2)
```

```
## [1] 0 0
```

We see that these differences are also zero, and the test indicates that `lin_batch()` implements the same computation as `batch()` with either of the two implementations of the gradient.

To investigate the effect of the C++ implementation, the mini-batch stochastic gradient algorithm is run with the Rcpp implementations of either the gradient or the entire epoch using a mini-batch size of 1000 and a constant learning rate $\gamma = 0.05$, which can be compared to the pure R implementation. In addition, we run the algorithm with mini-batch size one and $\gamma = 5 \times 10^{-5}$ using the Rcpp implementation of the epoch.

Figure 9.9 shows, unsurprisingly, that implementing the gradient as a C++ function makes the convergence faster compared to the pure R implementation, but turning the entire epoch into a C++ function results in little if any further improvement. However, the figure also shows that the basic stochastic gradient algorithm (mini-batch size equal to one) converges at a rate comparable to that of the mini-batch algorithm with mini-batch size 1000 when the entire epoch is in C++. Thus any benefit of the mini-batch algorithm with mini-batch size 1000 seems to be largely due to the inefficiency of the basic algorithm in pure R. Experiments using the `lin_batch()` epoch implementation (not shown) suggest that the mini-batch algorithm does have a real edge over the basic algorithm, but with much smaller mini-batch sizes, e.g. $m = 20$, cf Exercise 9.4.

9.3.2 Full Rcpp implementations

In this section we convert the entire stochastic gradient algorithm for the linear model into a C++ function. Two considerations come up when doing so:

- How do we sample with or without replacement in C++?

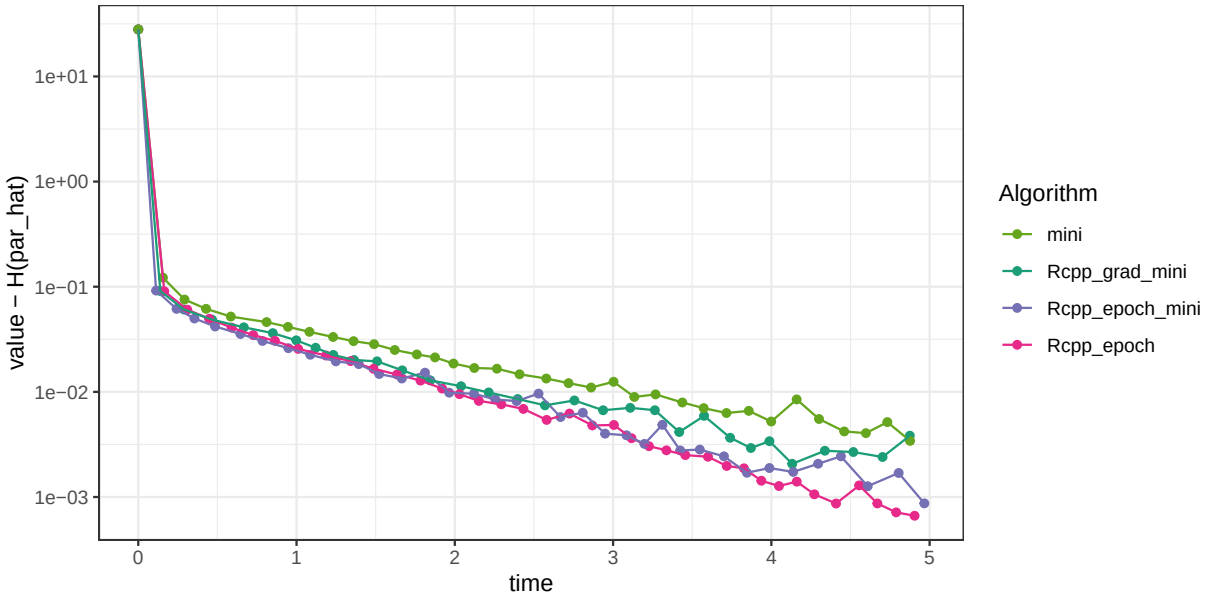


Figure 9.9: Convergence of squared error loss on the news article data set for four algorithms: R implementation of mini-batch stochastic gradient with a batch size of 1000 and a fixed learning rate (mini), the same mini-batch algorithm but with the gradient implemented in Rcpp (Rcpp_grad_mini) and with the entire epoch implemented in Rcpp (Rcpp_epoch_mini), and the basic stochastic gradient algorithm with the entire epoch implemented in Rcpp (Rcpp_epoch).

- Is it possible in C++ to implement the gradient computations using numerical linear algebra instead of two double for-loops?

We deal with the first question first, that is, how to sample from a finite set of size d in C++ such as `sample()` does in R. Though it may seem like a rather trivial problem, it is not entirely trivial to implement sampling from a general distribution on a finite set of size d that is both correct and efficient as a function of d. A non-exhaustive list of ways to sample from discrete sets using Rcpp is:

- The R function `sample()` can be retrieved via `Function sample("sample");` in Rcpp, which makes the function callable from C++ code. There is, however, an overhead associated with calling R functions from C++, which we prefer to avoid.
- There is an Rcpp sugar implementation of `sample()`. It generates different samples than the R function `sample()`.
- There is an RcppArmadilloExtension implementation of `sample()`. See also the blog post Sampling Integers. It requires the R package RcppArmadillo, and it also generates different samples than the R function `sample()`.
- There is the function `dqsample.int()` from the dqrng package dealt with in Section 4.1.2. It requires the R package dqrng and runs on an independent stream of pseudo random numbers then R's RNG.

The Rcpp sugar and the Rcpp Armadillo versions of `sample()` are touched on in the Rcpp introduction vignette, but they are not documented in detail. Both use R's stream of pseudo random numbers, but they nevertheless generate sequences that differ from each other and from `sample()`. To reproduce results in R, we would have to write an interface to those C++ functions if we were to use them. Since the dqrng package provides an interface to `dqsample.int()` from R as well as from C++, we will use that sampler in our implementations.

If we only want to generate random permutations, there are a couple of additional alternatives. There is `randperm()` from the Armadillo library, and there is also `std::random_shuffle()` from the C++ Standard Library. The implementation of `std::random_shuffle()` is, however, not specified by the standard and can be compiler dependent. Both of these alternatives will also run on an independent stream of pseudo random numbers then
R's RNG, and will not be considered any further.

The full Rcpp implementation of the stochastic gradient algorithm is now a simple extension of `lin_batch()` with one additional outer loop over the epochs and a single line calling `dqsample_int()` to sample the random permutation.

```
// [[Rcpp::depends(dqrng)]]
// [[Rcpp::export]]
NumericVector SG_Rcpp(
    NumericVector par,
    int N,
    NumericVector gamma,
    NumericMatrix X,
    NumericVector y,
    int m = 50,
    int maxiter = 100
) {
  int p = par.length(), M = floor(N / m);
  NumericVector grad(p), yhat(N), beta = clone(par);
  IntegerVector ii;

  for(int l = 0; l < maxiter; ++l) {
    // Note that dqsample_int samples from {0, 1, ..., N - 1}
    ii = dqrng::dqsample_int(N, N);
    for(int j = 0; j < M; ++j) {
      for(int i = j * m; i < (j + 1) * m; ++i) {
        yhat[i] = 0;
        for(int k = 0; k < p; ++k) {
          yhat[i] += X(ii[i], k) * beta[k];
        }
      }
      for(int k = 0; k < p; ++k) {
        grad[k] = 0;
        for(int i = j * m; i < (j + 1) * m; ++i) {
          grad[k] += X(ii[i], k) * (yhat[i] - y[ii[i]]);
        }
      }
      beta = beta - gamma[l] * (grad / m);
    }
  }
  return beta;
}
```

We test the implementation by comparing it to the pure R implementation, but using `dqsample_int() / dqsample.int()` to sample the same permutations in both cases.

```
dqset.seed(10)
par1 <- SG(
```

```
  par = par0,
  grad = ls_grad,
  N = nrow(X),
  gamma = 1e-3,
  maxiter = 10,
  sampler = dqrng::dqsample.int
)
dqset.seed(10)
par2 <- SG_Rcpp(
  par = par0,
  N = nrow(X),
  gamma = rep(1e-3, 10),
  X = X,
  y = y,
  maxiter = 10
)
range(par1 - par2)
```

```
## [1] 0 0
```

The two most notable overall differences between `SG()` and `SG_Rcpp()` is that `SG_Rcpp()` is low level and very problem specific, while `SG()` is high level and generic. It is surely possible to refactor the C++ code to write a more generic stochastic gradient algorithm, that would rely on other functions for computing e.g. the gradient. However, to retain performance those functions would need to be implemented in C++ as well.

To achieve a more high level implementation, we can use the RcppArmadillo package, which provides an interface to numerical linear algebra from the C++ library Armadillo. In this way we can use a matrix-vector style implementation similar to the R implementation but without excessive data copying. It is, however, necessary to use various data types from the Armadillo library, e.g. matrices and column vectors, and it is also necessary to type cast the sequence of integers from `dqsample_int()` as a `uvec` data type.

```
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
arma::colvec SG_arma(
    NumericVector par,
    int N,
    NumericVector gamma,
    const arma::mat& X,
    const arma::colvec& y,
    int m = 50,
    int maxiter = 100
) {
  int p = par.length(), M = floor(N / m);
  arma::colvec grad(p), yhat(N), beta = clone(par);
  uvec ii, iii;

  for(int l = 0; l < maxiter; ++l) {
    ii = as<arma::uvec>(dqrng::dqsample_int(N, N));
    for(int j = 0; j < M; ++j) {
      iii = ii.subvec(j * m, (j + 1) * m - 1);
```

```
      beta = beta - gamma[l] * (X.rows(iii).t() * (X.rows(iii) * beta - y(iii)) / m);
    }
  }
  return beta;
}
```

The function returns a column vector, which R represents as a $p \times 1$ matrix, whereas all the other implementations return a vector. But besides this difference, a test will show that `SG_Rcpp()` computes the same updates as all other implementations when `dqsample_int()` is used.

We benchmark and compare all five implementations of the mini-batch stochastic gradient algorithm by running them for 10 epoch iterations with a fixed learning rate of $\gamma = 10^{-4}$ and the default mini-batch size $m = 50$.

```
bench::mark(
  SG = SG(par0, nrow(X), 1e-4, maxiter = 10, grad = ls_grad),
  SG_lin_grad = SG(par0, nrow(X), 1e-4, maxiter = 10, grad = lin_grad, X = X, y = y),
  SG_lin_batch = SG(par0, nrow(X), 1e-4, lin_batch, maxiter = 10, X = X, y = y),
  SG_Rcpp = SG_Rcpp(par0, nrow(X), rep(1e-4, 10), X = X, y = y, maxiter = 10),
  SG_arma = SG_arma(par0, nrow(X), rep(1e-4, 10), X = X, y = y, maxiter = 10),
  check = FALSE, iterations = 5
)
```

```
## # A tibble: 5 x 6
##   expression        min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 SG              888ms    888ms      1.13  183.18MB     4.51
## 2 SG_lin_grad     403ms    407ms      2.43    43.9MB     1.62
## 3 SG_lin_batch    293ms    321ms      3.12    9.13MB     0
## 4 SG_Rcpp         302ms    304ms      3.30    1.82MB     0
## 5 SG_arma         494ms    502ms      1.99    4.54MB     0
```

The pure R implementation (SG) is, unsurprisingly, the slowest, but the full Rcpp implementation (SG_Rcpp) is less then a factor 3 faster. The Armadillo based implementation is somewhat slower, though still faster than the pure R implementation.

We rerun all algorithms, but this time with the mini-batch size $m = 1$. Because `SG()` allocates a lot of memory, the default memory tracking of `mark()` will become prohibitively slow, thus memory tracking is disabled.

```
bench::mark(
  SG = SG(par0, nrow(X), 1e-4, maxiter = 10, m = 1, grad = ls_grad),
  SG_lin_grad = SG(par0, nrow(X), 1e-4, maxiter = 10, m = 1, grad = lin_grad, X = X, y = y),
  SG_lin_batch = SG(par0, nrow(X), 1e-4, lin_batch, maxiter = 10, m = 1, X = X, y = y),
  SG_Rcpp = SG_Rcpp(par0, nrow(X), rep(1e-4, 10), X = X, y = y, maxiter = 10, m = 1),
  SG_arma = SG_arma(par0, nrow(X), rep(1e-4, 10), X = X, y = y, maxiter = 10, m = 1),
  check = FALSE, memory = FALSE, iterations = 5
)
```

```
## # A tibble: 5 x 6
##   expression        min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 SG              5.17s    5.39s     0.174        NA     4.34
```

```
## 2 SG_lin_grad     2.34s    2.53s     0.389        NA     3.03
## 3 SG_lin_batch 319.72ms 390.71ms     2.59         NA     0
## 4 SG_Rcpp      357.75ms 380.99ms     2.64         NA     0
## 5 SG_arma      699.18ms 707.28ms     1.39         NA     0
```

This reveals some bigger differences. With a fixed number of epoch iterations, the total amount of computations does not change substantially when changing the mini-batch size from 50 to 1, but the run times of `SG()` and `SG_lin_grad()` increase notably. The C++ implementations `SG_lin_batch()` and `SG_Rcpp` mostly retain the run time they had for $m = 50$. The runtime of the Armadillo implementation is not affected that much either, but it is still slower than the two other C++ based implementations and now by about a factor 2.

In summary, when the mini-batches are not too small, the improvements on runtime by using Rcpp were relatively small – about a factor 2 to 3. However, when the mini-batch size is small or even one, the runtime was about a factor 10 smaller when using either the full Rcpp implementation or the Rcpp implementation of the entire epoch update. Using `lin_batch()` for the epoch updates strikes a good compromise in this particular example, by being runtime competitive but with the main control structures of the algorithm in R.

9.4 Exercises

Exercise 9.1. Consider the loss

$$L((y,z),\theta) = \mu(z,\theta) - y\log(\mu(z,\theta))$$

for a model, $\mu(z,\theta)$, of the conditional mean, $E(Y \mid Z = z)$, of Y given $Z = z$. See also Example 9.2.

Show that the function

$$a \mapsto a - b\log(a)$$

for fixed b attains it unique minimum in $(0,\infty)$ for $a = b$.

Then show the lower bound on the risk

$$H(\theta) = E(L((Y,Z),\theta)) \geq E\Big(E(Y \mid Z) - E(Y \mid Z)\log(E(Y \mid Z))\Big),$$

and conclude that if $E(Y \mid Z = z) = \mu(z,\theta_0)$ for some θ_0, then θ_0 is a global minimizer of the risk.

Exercise 9.2. Consider the same online setup as in Example 9.3 but with

$$Y_i \mid Z_i = z_i \sim \mathcal{N}(e^{\beta_0+\beta_1 z_i}, 5)$$

instead. That is, the conditional mean value model is still log-linear, but the distribution is Gaussian instead of Poisson.

Replicate the online learning simulation as in Example 9.3 using the Poisson log-likelihood gradient and the squared error gradient. You may want to experiment with different learning rates. How does the change of the distribution affect the convergence of the algorithms?

Exercise 9.3. Show that for the linear model and the squared error loss, the hessian of the objective function is

$$D^2 H_N(\beta) = X^T X$$

where X is the model matrix. Compute the eigenvalues of this matrix using the news data from Section 9.1.4 before and after standardization. What are the conditioning numbers of the matrices, and how do their values relate to convergence of gradient based methods? See also Section 7.1.1.

Exercise 9.4. Make sure that you have a working implementation of the mini-batch stochastic gradient algorithm for the news data from Section 9.1.4 with the `lin_batch()` Rcpp implementation of the epoch updates. Set up an experiment where you run the algorithm over a grid of mini-batch sizes and learning rates. Run the algorithm until the absolute deviation from the minimum of the empirical squared error loss is less than 0.01 and determine the optimal choice of the tuning parameters.

Appendix A

R programming

This appendix on R programming is a brief overview of important programming constructs and concepts in R that are used in the book. For a detailed and much more extensive coverage of R as a programming language the reader is referred to the book Advanced R.

Depending on your background, the appendix can serve different purposes. If you already have experience with programming in R, this appendix can serve as a brush up on some basic aspects of the R language that are used throughout the book. If you have experience with programming in other languages than R, it can serve as an introduction to typical R programming techniques, where some may differ from what you know from other languages. If you have little prior experience with programming, this appendix can teach you the most important things for getting started with the book, but you are encouraged to follow up with additional and more detailed material like Advanced R.

For everybody, this appendix covers specific topics relevant for reading the book. These topics include

- data types, comparisons and numerical precision
- vectorized computations
- functions, environments and function factories
- performance assessment and improvement
- S3 objects and methods

Several important topics, such as S4 and R6 objects, expressions, and data wrangling, are not covered in any detail. The book is on implementations of correct and efficient numerical algorithms used in statistics, and this is reflected in the topics covered in this appendix.

A.1 Data structures

The fundamental data structure in R is a vector. Even variables that look and behave like single numbers are vectors of length one. Vectors come in two flavors: *atomic vectors* and *lists*.

An atomic vector is an indexed collection of data elements that are all of the same type, e.g.

- integers
- floating point numbers
- logical values
- character strings

A list is an indexed collection of elements without any type restrictions on the individual elements. An element in a list can, for instance, be a list itself.

A.1.1 Atomic vectors

You can construct a vector in R by simply typing in its elements, e.g.

```
first_vector <- c(1, 2, 3)  # Note the 'c'
first_vector
```

```
## [1] 1 2 3
```

The constructed vector contains the numbers 1, 2 and 3. We use the classical assignment operator `<-` throughout, while R supports using `=` if you prefer. The `c()` used on the right hand side of the assignment is short for *combine* (or concatenate), and it is also used if you combine two vectors into one.

```
second_vector <- c(4, 5, 6)
c(first_vector, second_vector)
```

```
## [1] 1 2 3 4 5 6
```

There are several convenient techniques in R for constructing vectors of various regular nature, e.g. sequences. The following example shows how to construct a vector containing the integers from 1 to 10. The type of the vector is `integer` indicating that the elements of the vector are stored as integers.

```
integer_vector <- 1:10
integer_vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
typeof(integer_vector)  # `typeof` reveals the internal storage mode
```

```
## [1] "integer"
```

We can access the individual elements as well as subsets of a vector by indices using square brackets.

```
integer_vector[3]
```

```
## [1] 3
```

```
integer_vector[first_vector]  # The first three elements
```

```
## [1] 1 2 3
```

The function `seq` generalizes the colon operator, `1:10`, as a way to generate regular sequences. The following example shows how to generate a sequence, `double_vector`, from 0.1 to 1.0 with increments of size 0.1. The type of the resulting vector is `double`, which indicates that the elements of `double_vector` are stored as doubles. That is, the numbers are stored as floating point numbers using 64 bits of storage with a precision of just about 16 digits.

```
double_vector <- seq(0.1, 1, by = 0.1)
double_vector
```

```
##  [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
typeof(double_vector)
```

```
## [1] "double"
```

Integers are often stored as or coerced into doubles automatically. The supposedly integer vector, `first_vector`, is actually of type `double`.

```
typeof(first_vector)
```

```
## [1] "double"
```

In R, numerical data of either type integer or double is collectively referred to as *numerics* and have mode (and class) `numeric`. This is confusing, in particular because "numeric" is used also as pseudonym of the *type* double, but it is rarely a practical problem. A vector of type integer or double is often just said to by a `numeric`, and the function `is.numeric()` reflects this.

It is possible to insist that integers are actually stored as integers by appending `L` to each integer, e.g.

```
typeof(c(1L, 2L, 3L))
```

```
## [1] "integer"
```

When sequences are generated using the colon operator with the endpoints being integers, as in `1:10`, the result will be a vector of type integer. This is an exception. Apparent integers are usually – and silently – converted into doubles if they are not explicitly marked as integers by `L`.

Vectors of any length can be created by the generic function `vector()`, or by type specific functions such as `numeric()` that creates vectors of type double.

```
vector("numeric", length = 10)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
numeric(10)
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

Both vectors above are of type double and of length 10 and initialized with all elements being 0.

A *logical vector* is another example of a useful atomic vector. The default type of a vector created by `vector()` is logical, with all elements being `FALSE`.

```
vector(length = 10)
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Logical vectors are encountered when we compare the elements of one vector to another vector or to a number.

```
logical_vector <- integer_vector > 4
logical_vector
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
typeof(logical_vector)
```

```
## [1] "logical"
```

While a logical vector has its own type and is stored efficiently as such, it behaves in many ways as a numeric vector with `FALSE` equivalent to 0 and `TRUE` equivalent to 1. If we want to compute the relative frequency of elements in `integer_vector` that are are (strictly) larger than 4, say, we can simply take the mean of `logical_vector`.

```
mean(logical_vector)
```

```
## [1] 0.6
```

The mean of a logical vector behaves as if the logical values are coerced into zeros and ones before the mean is computed.

A final example of an atomic vector is a *character vector*. In this example, the vector is combined from 6 individual strings to form a vector of length 6. Combining strings into a vector does not paste the strings together – it forms a vector, whose elements are the individual strings.

```
character_vector <- c("A", "vector", "of", "length", 6, ".")
character_vector
```

```
## [1] "A"      "vector" "of"     "length" "6"      "."
```

```
typeof(character_vector)
```

```
## [1] "character"
```

The type of the vector is character. Elements of a vector of type character are strings. Note how the numeric value 6 in the construction of the vector was automatically coerced into the string `"6"`.

It is possible to paste together the strings in a character vector.

```
paste(character_vector, collapse = " ")  # Now a character vector of length 1!
```

```
## [1] "A vector of length 6 ."
```

It is likewise possible to split a string according to a pattern. For instance into its individual characters.

```
strsplit(character_vector[2], split = "")  # Split "vector" into characters
```

```
## [[1]]
## [1] "v" "e" "c" "t" "o" "r"
```

Various additional string operations are available – see `?character` for more information.

In summary, atomic vectors are the primitive data structures used in R, with elements being accessible via indices (random access). Typical vectors contain numbers, logical values or strings. There is no declarations of data types – they are inferred from data or computations. This is a flexible type system with many operations in R silently coercing elements in a vector from one type to another.

A.1.2 Comparisons and numerical precision

We can compare vectors using the equality operator `==`, which compares two vectors element-by-element. The result of the following comparison might be a surprise at first sight.

```
double_vector[2:3]
```

```
## [1] 0.2 0.3
```

```
double_vector[2:3] == c(0.2, 0.3)
```

```
## [1]  TRUE FALSE
```

The result of the comparison is a vector of length 2 containing the logical values `TRUE FALSE`, but `double_vector[2:3]` appears to be equal to `c(0.2, 0.3)`. The difference shows up if we increase the number of printed digits from the default (which is 7) to 20.

```
c(0.2, 0.3)
```

```
## [1] 0.2000000000000000111 0.2999999999999999889
```

```
double_vector[2:3]
```

```
## [1] 0.20000000000000001110 0.30000000000000004441
```

The 0.3 produced by `seq` is computed as `0.1 + 0.1 + 0.1`, while the 0.3 in the vector `c(0.2, 0.3)` is converted directly into a double precision number. The difference arises because neither 0.1 nor 0.3 are exactly representable in the binary numeral system, and the arithmetic operations induce rounding errors. The function `numToBits` can reveal the exact difference in the three least significant bits.

```
numToBits(0.3)[1:3]
```

```
## [1] 01 01 00
```

```
numToBits(0.1 + 0.1 + 0.1)[1:3]
```

```
## [1] 00 00 01
```

Differences in the least significant bits are tolerable when we do numerical computations but can be a nuisance for equality testing. When comparing vectors containing doubles we are therefore often interested in testing for approximate equality instead using e.g.

```
all.equal(double_vector, c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0))
```

```
## [1] TRUE
```

The function `all.equal` has a tolerance argument controlling if numerical differences will be regarded as actual differences. Another way of comparing numerical vectors is by computing the range of their difference

```
range(double_vector - c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0))
```

```
## [1] 0.000000e+00 1.110223e-16
```

This shows the largest positive and negative difference. Usually, the size of the differences should be assessed relative to the magnitudes of the numbers compared. For numbers approximately of magnitude 1, differences of the order $2^{-52} \simeq 2.22 \times 10^{-16}$ (the "machine epsilon") can safely be regarded as rounding errors. The default (relative) tolerance of `all.equal` is the more permissive number $\sqrt{2^{-53}} \simeq 1.5 \times 10^{-8}$. This number is an *ad hoc* but commonly used tolerance. See `?all.equal` for more details.

A.1.3 Lists

A.1.4 Other data structures

Factors, dates, data frames

A.2 Functions

When you write an R script and source it, the different R expressions in the script are evaluated sequentially. Intermediate results may be stored in variables and used further down in the script. It may not be obvious, but all the expressions in the script are, in fact, function calls. Your script relies on functions implemented in either core R packages or in other installed packages.

It is possible to use R as a scripting language without ever writing your own functions, but writing new R functions is how you extend the language, and it is how you modularize and reuse your code in an efficient way. As a first example, we implement the Gaussian kernel with bandwidth h (see Section 2.2)

$$K_h(x) = \frac{1}{h\sqrt{2\pi}} e^{-\frac{x^2}{2h^2}}.$$

We call the function `gauss()` to remind us that this is the Gaussian kernel.

```
gauss <- function(x, h = 1) {
  exp(- x^2 / (2 * h^2)) / (h * sqrt(2 * pi))
}
```

The *body* of the function is the R expression `exp(- x^2 / (2 * h^2)) / (h * sqrt(2 * pi))`. When the function is called with specific numeric values of its two *formal arguments* `x` and `h`, the body is evaluated with the formal arguments replaced by their values. The value of the (last) evaluated expression in the body is returned by the function. The bandwidth argument `h` is given a default value `1`, so if that argument is not specified when the function is called, it gets the value `1`.

The Gaussian kernel with bandwidth h is the Gaussian density with standard deviation h. Thus we can compare our implementation with `dnorm()` in R that computes the Gaussian density.

```
c(gauss(1), dnorm(1))
```

```
## [1] 0.2419707 0.2419707
```

```
c(gauss(0.1, 0.1), dnorm(0.1, sd = 0.1))  # Bandwidth changed to 0.1
```

```
## [1] 2.419707 2.419707
```

For those two cases the functions compute the same (at least, up to the printed precision). Note how the formal argument `sd` is given the value `0.1` in the second call of `dnorm()`. The argument `sd` is the third argument of `dnorm()`, but we don't need to specify the second, as in `dnorm(0.1, 0, 0.1)`, to specify the third. We can do so by its name, in which case the second argument in `dnorm()` gets its default value `0`.

There are several alternative ways to implement the Gaussian kernel that illustrate how functions can be written in R.

```
# A one-liner without curly brackets
gauss_one_liner <- function(x, h = 1)
  exp(- x^2 / (2 * h^2)) / (h * sqrt(2 * pi))

# A stepwise implementation computing the exponent first
gauss_step <- function(x, h = 1) {
  exponent <- (x / h)^2 / 2
```

```
  exp(- exponent) / (h * sqrt(2 * pi))
}

# A stepwise implementation with an explicit return statement
gauss_step_return <- function(x, h = 1) {
  exponent <- (x / h)^2 / 2
  value <- exp(- exponent) / (h * sqrt(2 * pi))
  return(value)
}
```

The following small test shows two cases where all implementations compute the same.

```
c(gauss(1), gauss_one_liner(1), gauss_step(1), gauss_step_return(1))
```

```
## [1] 0.2419707 0.2419707 0.2419707 0.2419707
```

```
c(
  gauss(0.1, 0.1),
  gauss_one_liner(0.1, 0.1),
  gauss_step(0.1, 0.1),
  gauss_step_return(0.1, 0.1)
)
```

```
## [1] 2.419707 2.419707 2.419707 2.419707
```

A function should always be tested. A test is a comparison of the return value of the function with the expected value for some specific argument(s). The expected value can either be computed by hand or by another implementation – as in the comparisons of `gauss()` and `dnorm()`. Such tests cannot prove that the implementation is correct, but discrepancies can help you to catch and correct bugs. Remember that when testing numerical computations that use floating point numbers we cannot expect exact equality. Thus tests should reveal if return values are within an acceptable tolerance of the expected results.

Larger pieces of software – such as an entire R package – should include a number of tests of each function it implements. This is known as *unit testing* (each unit, that is, each function, is tested), and there are packages supporting the systematic development of unit tests for R package development. A comprehensive set of unit tests also helps when functions are rewritten to e.g. improve performance or extend functionality. If the rewritten function passes all tests, chances are that we didn't break anything by rewriting the function.

It is good practice to write functions that do one well defined computation and to keep the body of a function relatively small. Then it is easier to reason about what the function does and it is easier to comprehensively test it. Complex behavior is achieved by composing small and well tested functions.

To summarize:

- The body is enclosed by curly brackets `{}`, which may be left out if the body is only one line.
- The function returns the value of the last expression in the body except when the body contains an explicit return statement.
- Formal arguments can be given default values when the function is implemented, and arguments can be passed to the function by position as well as by name.
- Functions should do a single well defined computation and be well tested.

A.2.1 Vectorization

In Section A.1 it was shown how comparison operators work in a vectorized way. In R, comparing a vector to another vector or a number leads to element-by-element comparisons with a logical vector as the result. This is one example of how many operations and function evaluations in R are natively vectorized, which means that when the function is evaluated with a vector argument the function body is effectively evaluated for each entry in the vector.

Our `gauss()` function is another example of a function that automatically works as a vectorized function.

```
gauss(c(1, 0.1), c(1, 0.1))
```

```
## [1] 0.2419707 2.4197072
```

It works as expected for the vector input because all the functions in the body of `gauss()` are vectorized, that is, the arithmetic operators are vectorized, the square root is vectorized and the exponential function is vectorized.

It is good practice to write R programs that use vectorized computations whenever possible. The alternative for-loop can be much slower. Several examples in the book illustrate the computational benefits of vectorized implementations. It may, however, not always be obvious how to correctly implement a vectorized function.

Suppose we want to implement the following function

$$\overline{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i)$$

for a data set $x_1, \ldots, x_n$ and K_h the Gaussian kernel. This is the Gaussian kernel density estimator considered in Section 2.4. A straight forward implementation is

```
xs <- rnorm(10)  # A data set with 10 observations
f_bar <- function(x, h)
  mean(gauss(x - xs, h))
```

This implementation works correctly when `x` and `h` are single numbers but when e.g. `x` is a vector, the function returns a number (not a vector) unrelated to $\overline{f}_h(0)$ and $\overline{f}_h(1)$.

```
c(f_bar(0, 1), f_bar(1, 1))
```

```
## [1] 0.3128259 0.2154763
```

```
f_bar(c(0, 1), 1) # Computation is not vectorized
```

```
## [1] 0.2940242
```

A quick fix is the following explicit vectorization

```
f_bar_vec <- Vectorize(f_bar)
```

The R function `Vectorize()` is a function operator, which takes a function as argument and returns a function. In this case a vectorized version of the input function. That is, `f_bar_vec()` can be applied to a vector, which results in applying `f_bar()` to each element of the vector. We test that `f_bar_vec()` works correctly – both when the first and the second argument is given as a vector.

```
f_bar_vec(c(0, 1), 1)
```

```
## [1] 0.3128259 0.2154763
```

```
c(f_bar(0, 1), f_bar(1, 1)) # Same result as the vectorized computation
```

```
## [1] 0.3128259 0.2154763
```

```
c(f_bar(1, 1), f_bar(1, 0.1))
```

```
## [1] 2.154763e-01 4.338004e-07
```

```
f_bar_vec(1, c(1, 0.1)) # Same result as the vectorized computation
```

```
## [1] 2.154763e-01 4.338004e-07
```

The function `Vectorize()` basically works by looping over the elements in the vector argument(s) and applying the function to each element. For prototyping and quick implementations it can be very convenient, but it is not a shortcut to efficient vectorized computations.

`Vectorize()` is used in Section 2.4 to implement $\overline{f}_h$ based on `dnorm()`. The purpose of that implementation is to be able to compute $\overline{f}_h(x)$ for arbitrary x in a vectorized way, so that the function can be used together with `curve()` and in the likelihood computations of that section. The implementations in Section 2.2.1 differ by computing and returning

$$\overline{f}_h(\tilde{x}_1), \ldots, \overline{f}_h(\tilde{x}_m).$$

That is, those implementations return the evaluations of $\overline{f}_h$ in an equidistant grid $\tilde{x}_1, \ldots, \tilde{x}_m$ and not $\overline{f}_h$ itself.

A.2.2 Environments

Something

Use `xs` above as example.

A.2.3 Function factories

Something

The random number streams from Chap. 4. Include implementation in package.

A.3 Performance

Something

Mention parallel computations

A.3.1 Tracing

A.3.2 Rcpp

Use the mean_numeric and mean_complex examples.

A.4 Objects and methods

Something

A.5 Exercises

Functions

Exercise A.1. Explain the result of evaluating the following R expression.

```
(0.1 + 0.1 + 0.1) > 0.3
```

```
## [1] TRUE
```

Exercise A.2. Write a function that takes a numeric vector `x` and a threshold value `h` as arguments and returns the vector of all values in `x` greater than `h`. Test the function on `seq(0, 1, 0.1)` with threshold 0.3. Have the example from Exercise A.1 in mind.

Exercise A.3. Investigate how your function from Exercise A.2 treats missing values (`NA`), infinite values (`Inf` and `-Inf`) and the special value "Not a Number" (`NaN`). Rewrite your function (if necessary) to exclude all or some of such values from `x`.

Hint: The functions `is.na`, `is.nan` and `is.finite` are useful.

Histograms with non-equidistant breaks

The following three exercises will use a data set consisting of measurements of infrared emissions from objects outside of our galax. We will focus on the variable F12, which is the total 12 micron band flux density.

```
infrared <- read.table("data/infrared.txt", header = TRUE)
F12 <- infrared$F12
```

The purpose of this exercise is two-fold. First, you will get familiar with the data and see how different choices of visualizations using histograms can affect your interpretation of the data. Second, you will learn more about how to write functions in R and gain a better understanding of how they work.

Exercise A.4. Plot a histogram of `log(F12)` using the default value of the argument `breaks`. Experiment with alternative values of `breaks`.

Exercise A.5. Write your own function, called `my_breaks`, which takes two arguments, `x` (a vector) and `h` (a positive integer). Let `h` have default value `5`. The function should first sort `x` into increasing order and then return the vector that: starts with the smallest entry in `x`; contains every *h*th unique entry from the sorted `x`; ends with the largest entry in `x`.

For example, if `h = 2` and `x = c(1, 3, 2, 5, 10, 11, 1, 1, 3)` the function should return `c(1, 3, 10, 11)`. To see this, first sort `x`, which gives the vector `c(1, 1, 1, 2, 3, 3, 5, 10, 11)`, whose unique values are `c(1, 2, 3, 5, 10, 11)`. Every second unique entry is `c(1, 3, 10)`, and then the largest entry `11` is concatenated.

Hint: The functions `sort` and `unique` can be useful.

Use your function to construct *breakpoints* for the histogram for different values of `h`, and compare with the histograms obtained in Exercise A.4.

Exercise A.6. If there are no ties in the data set, the function above will produce breakpoints with `h` observations in the interval between two consecutive breakpoints (except the last two perhaps). If there are ties, the function will by construction return unique breakpoints, but there may be more than `h` observations in some intervals.

The intention is now to rewrite `my_breaks` so that if possible each interval contains `h` observations.

Modify the `my_breaks` function with this intention and so that is has the following properties:

- All breakpoints must be unique.
- The range of the breakpoints must cover the range of `x`.
- For two subsequent breakpoints, a and b, there must be at least `h` observations in the interval $(a, b]$, provided `h < length(x)`. (With the exception that for the first two breakpoints, the interval is $[a, b]$.)

Functions and objects

The following exercises build on having implemented a function that computes breakpoints for a histogram either as in Exercise A.5 or as in Exercise A.6.

Exercise A.7. Write a function called `my_hist`, which takes a single argument `h` and plots a histogram of `log(F12)`. Extend the implementation so that any additional argument specified when calling `my_hist` is passed on to `hist`. Investigate and explain what happens when executing the following function calls.

```
my_hist()
my_hist(h = 5, freq = TRUE)
my_hist(h = 0)
```

Exercise A.8. Modify your `my_hist` function so that it returns an object of class `my_histogram`, which is not plotted. Write a print method for objects of this class, which prints just the number of cells.

Hint: It can be useful to know about the function `cat`.

How can you assign a class label to the returned object so that it is printed using your new print method, but it is still plotted as a histogram when given as argument to `plot`?

Exercise A.9. Write a `summary` method that returns a data frame with two columns containing the midpoints of the cells and the counts.

Exercise A.10. Write a new `plot` method for objects of class `my_histogram` that uses `ggplot2` for plotting the histogram.

Functions and environments

The following exercises assume that you have implemented a `my_hist` function as in Exercise A.7.

Exercise A.11. What happens if you remove that data and call `my_hist` subsequently? What is the environment of `my_hist`? Change it to a new environment, and assign (using the function `assign`) the data to a variable with an appropriate name in that environment. Once this is done, check what now happens when calling `my_hist` after the data is removed from the global environment.

Exercise A.12. Write a function that takes an argument `x` (the data) and returns a function, where the returned function takes an argument `h` (just as `my_hist`) and plots a histogram (just as `my_hist`). Because the return value is a function, we may refer to the function as a function factory.

What is the environment of the function created by the function factory? What is in the environment? Does it have any effect when calling the function whether the data is altered or removed from the global environment?

Exercise A.13. Evaluate the following function call:

```
tmp <- my_hist(10, plot = FALSE)
```

What is the type and class of `tmp`? What happens when `plot(tmp, col = "red")` is executed? How can you find help on what `plot` does with an object of this class? Specifically, how do you find the documentation for the argument `col`, which is not an argument of `plot`?

Bibliography

Demmler, A. and Reinsch, C. (1975). Oscillation matrices with spline smoothing. *Numerische Mathematik*, 24(5):375–382.

Fernandes, K., Vinagre, P., and Cortez, P. (2015). A proactive intelligent decision support system for predicting the popularity of online news. In *Proceedings if the 17th EPIA 2015 - Portuguese Conference on Artificial Intelligence*.

Gürbüzbalaban, M., Ozdaglar, A., and Parrilo, P. A. (2019). Why random reshuffling beats stochastic gradient descent. *Mathematical Programming*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

Lai, T. L. (2003). Stochastic approximation: invited paper. *Ann. Statist.*, 31(2):391–406.

Lauritzen, S. L. and Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 50(2):157–224.

L'Ecuyer, P. and Simard, R. (2007). Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4):22:1–22:40.

Marsaglia, G. (2003). Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6.

Marsaglia, G. and Tsang, W. W. (2000). A simple method for generating gamma variables. *ACM Trans. Math. Softw.*, 26(3):363–372.

Nocedal, J. and Wright, S. J. (2006). *Numerical optimization*. Springer Series in Operations Research and Financial Engineering. Springer, New York, second edition.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407.

Sheather, S. J. and Jones, M. C. (1991). A reliable data-based bandwidth selection method for kernel density estimation. *Journal of the Royal Statistical Society. Series B (Methodological)*, 53(3):683–690.

Silverman, B. W. (1986). *Density Estimation for Statistics and Data Analysis*. Chapman and Hall/CRC.

Tsybakov, A. B. (2009). *Introduction to nonparametric estimation*. Springer Series in Statistics. Springer, New York.

Vinther, B. M., Andersen, K. K., Jones, P. D., Briffa, K. R., and Cappelen, J. (2006). Extending greenland temperature records into the late eighteenth century. *Journal of Geophysical Research: Atmospheres*, 111(D11).